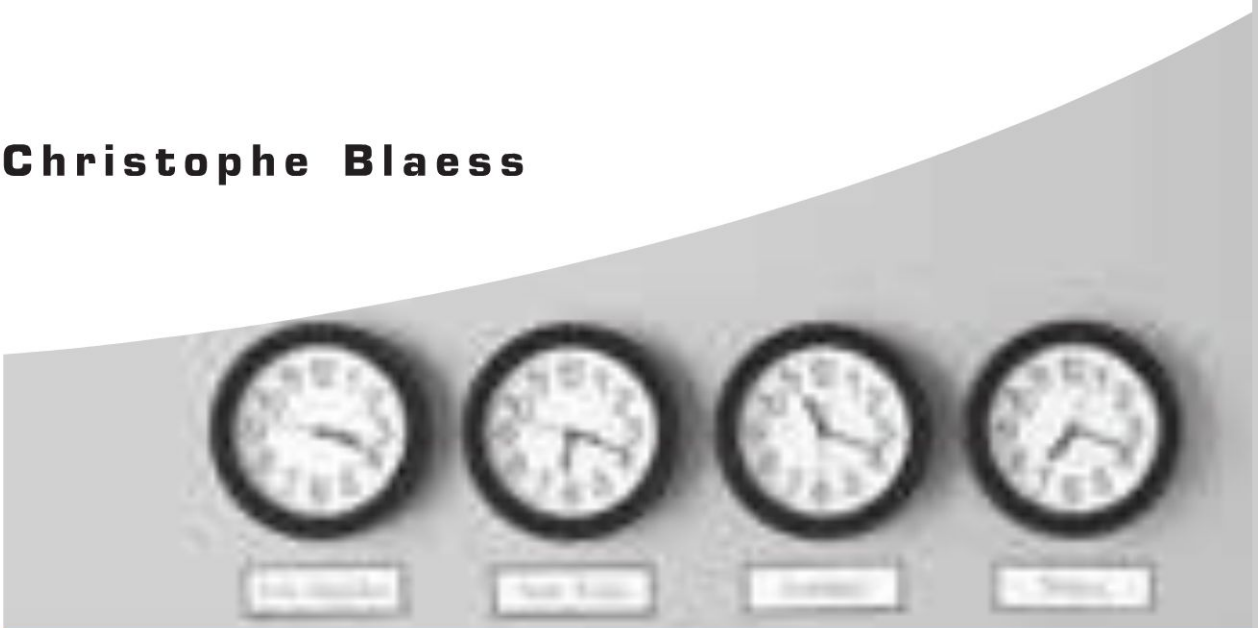
2e édition

Solutions temps réel sous Linux

Cas pratique : le Raspberry Pi

Christophe Blaess



Copyright © 2015 Eyrolles.

EYROLLES

Solutions temps réel sous Linux

2e édition

Avec 50 exercices corrigés

Comprendre le fonctionnement de l'ordonnanceur et du noyau

Pour concevoir un système équilibré, stable et réactif aux événements externes, il est indispensable de bien comprendre le rôle et l'organisation de ses divers composants. C'est l'un des premiers buts de ce livre, qui détaille et commente les interactions, les activations et les commutations des tâches. De très nombreux exemples illustrant le propos permettront au lecteur de réaliser ses propres expériences sur son poste Linux.

Bâtir un système temps réel sous contraintes temporelles fortes

Pour construire une application temps réel sous Linux, l'architecte logiciel doit choisir entre différentes solutions, un choix crucial qui influera sensiblement sur les limites de fonctionnement de son application. Dans cet ouvrage, l'auteur étudie les environnements libres pouvant répondre à des contraintes temporelles plus ou moins fortes et propose des outils pour valider le comportement des tâches face à des charges logicielles ou interruptives importantes. Augmentée et mise à jour, la deuxième édition a pour support d'expérimentation le nano-ordinateur Raspberry Pi. Elle étudie également la mise en œuvre du temps réel sur la carte BeagleBone Black, de portée plus industrielle.

Au sommaire

Multitâche et commutation • Multitâche sous Linux • Systèmes multiprocesseurs • États des tâches • **Interruptions, exceptions et appels système** • Mode noyau • Interruptions et exceptions • Appels système • Threads du noyau • **Ordonnancement temps partagé et priorités** • Temps partagé • Configuration des priorités • **Limitations de l'ordonnancement temps partagé** • Mesure du temps • Tâches périodiques • Préemption des tâches • **Principes du temps réel** • Traitement direct dans le noyau • Temps réel sous Linux • **Performances du temps réel souple** • Timers temps réel • Temps de commutation • Préemptibilité du noyau • **Problèmes temps réel classiques** • Démarrage en Round Robin • Inversion de priorité • Prise de mutex • **Limites et améliorations du temps réel Linux** • Traitement des interruptions • PREEMPT-RT • Outils de mesure des performances • Économies d'énergie • **Extensions temps réel de Linux** • Les nanokernels temps réel • Installation de Xenomai • Expériences avec Xenomai • **Programmer avec Xenomai** • Programmation de tâches simples • Alarmes et tâches périodiques • Synchronisation des tâches • **Traitement des interruptions** • Programmation d'un driver • Interruptions avec Xenomai.

À qui s'adresse cet ouvrage ?

– Aux développeurs, architectes logiciels et ingénieurs devant mettre en œuvre des applications temps réel sous Linux
– Aux décideurs et industriels souhaitant installer un système temps réel sous Linux
– Aux étudiants en informatique

C. Blaess

Expert reconnu de Linux dans l'industrie, Christophe Blaess conçoit et met en œuvre des systèmes embarqués industriels, et propose au sein de la société Logilin qu'il a créée en 2004 des prestations d'ingénierie et de conseil dans différents domaines liés à Linux. Soucieux de partager ses connaissances et son savoir-faire, il dispense également des formations professionnelles (Linux temps réel, Linux embarqué avec Yocto, écriture de *drivers*, programmation système...), co-organise des rencontres dédiées aux systèmes libres embarqués *(Paris Embedded Meetup)* et publie régulièrement des articles sur son blog et dans des revues spécialisées.

Sur le site http://christophe.blaess.fr
– Téléchargez le code source des exemples
– Consultez les corrigés des exercices et de nombreux documents complémentaires
– Dialoguez avec l'auteur

Copyright © 2015 Eyrolles.

Solutions temps réel sous Linux

2e édition

Copyright © 2015 Eyrolles.

Dans la collection *Les guides de formation Tsoft*

J.-F. Bouchaudy. – **Linux Administration. Tome 1 : les bases de l'administration système**.
N°14082, 3e édition, 2014, 690 pages.

J.-F. Bouchaudy. – **Linux Administration. Tome 2 : administration système avancée**.
N°12882, 2e édition, 2011, 504 pages.

J.-F. Bouchaudy. – **Linux Administration. Tome 3 : sécuriser un serveur Linux**.
N°13462, 2e édition, 2012, 400 pages.

J.-F. Bouchaudy. – **Linux Administration. Tome 4 : installer et configurer des serveurs Web, mail ou FTP sous Linux.**
N°13790, 2e édition, 2013, 420 pages.

Autres ouvrages

P. Ficheux. – **Linux embarqué.**
N°13482, 4e édition, 2012, 378 pages.

C. Blaess. – **Développement système sous Linux.** *Ordonnancement multitâches, gestion mémoire, communications, programmation réseau. À paraître.*
N°14207, 4e édition, 2016, 1004 pages.

C. Blaess. – **Shells Linux et Unix par la pratique.**
N°13579, 2e édition, 2012, 296 pages.

I. Hurbain, E. Dreyfus. – **Mémento Unix/Linux**
N°13306, 2e édition, 2011, 14 pages.

Copyright © 2015 Eyrolles.

Solutions temps réel sous Linux

2e édition

Christophe Blaess

Copyright © 2015 Eyrolles.

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Copyright © 2015 Eyrolles.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.
© Groupe Eyrolles, 2012, 2016, ISBN : 978-2-212-14208-2

Avant-propos

La mise au point d'un système temps réel, construit sur des solutions logicielles libres peut sembler *a priori* assez compliquée.

Quel noyau choisir ? Quelle version ? Faut-il ajouter des extensions ? Quelles performances peut-on espérer en retour ? Quel sera le comportement du système en cas de pics de charge logicielle ? De charge d'interruption ? Toutes ces questions – qui se poseront à un moment ou un autre – peuvent décourager l'architecte système face à la multitude d'offres libres ou commerciales se réclamant de « Linux temps réel ».

Dans ce livre, j'ai souhaité aider le concepteur, le chef de projet, le développeur, à bien saisir les éléments à prendre en considération lors de la mise au point d'un système où les performances d'ordonnancement et les temps de réponse aux interruptions sont importants. Le propos s'articulera donc sur le fonctionnement et les possibilités temps réel de Linux et de ses extensions libres. Nous n'aborderons ni les spécificités des distributions Linux (Ubuntu, Debian, Fedora, etc.) qui encadrent le noyau standard, ni les solutions commerciales (comme *Suse Linux Enterprise Real Time*, ou *Red Hat Enterprise MRG*) construites autour de Linux et de ses extensions temps réel auxquelles elles apportent essentiellement un support technique et une validation sur des plates-formes définies.

Dans un premier temps (chapitres 1 et 2), nous examinerons quelques concepts de base concernant le **multitâche** sous Linux et les interactions entre l'espace utilisateur (dans lequel les applications s'exécutent) et l'espace kernel (contenant le code du noyau proprement dit).

Pour comprendre les enjeux de l'ordonnancement, nous commencerons par observer dans les chapitres 3 et 4 les principes et les limites du **temps partagé** (le type d'ordonnancement employé par défaut pour toutes les tâches usuelles).

Pour un comportement prédictible, le temps partagé n'est pas satisfaisant, aussi basculerons-nous aux chapitres 5, 6 et 7 sur un ordonnancement temps réel souple *(soft real time)*. Ces chapitres seront également l'occasion d'aborder des points importants pour les systèmes temps réel : préemptibilité du noyau, granularité et précision des *timers*, inversions de priorités…

Dans le chapitre 8, nous examinerons les limites du temps réel souple de Linux, ainsi que certaines améliorations possibles, comme l'utilisation du **patch PREEMPT_RT**. Nous y étudierons également des outils de mesure des performances sous une charge très élevée en processus comme en interruptions.

Copyright © 2015 Eyrolles.

Pour obtenir un comportement temps réel encore plus prédictible, relevant du **temps réel strict** *(hard real time)*, nous aborderons aux chapitres 9 et 10 l'utilisation de l'extension **Xenomai**.

Enfin, le chapitre 11 nous permettra d'écrire du code s'exécutant dans le noyau Linux, avec un bref aperçu de l'écriture d'un driver et surtout de la **gestion des interruptions**. Nous découvrirons à cette occasion une seconde interface de programmation, nommée RTDM, proposée par Xenomai.

Lors de la mise au point d'un système temps réel avec Linux, on s'aperçoit d'une très grande variation des performances en fonction du matériel utilisé. Les fluctuations d'un timer peuvent facilement évoluer dans un rapport de 1 à 10 entre deux architectures différentes (processeur, périphériques, bus, contrôleur d'interruptions), de même que le temps de déclenchement d'une interruption ou la durée de préemption d'une tâche. Il n'est donc pas possible de fournir des résultats chiffrés absolus indépendamment de la plate-forme cible envisagée. Toutes les valeurs que nous observerons dans ce livre seront donc à prendre relativement les unes par rapport aux autres et non pas comme des valeurs absolues applicables sur une autre architecture.

La plupart des exemples ont été exécutés sur un petit PC portable aux performances plutôt limitées. On trouvera aussi régulièrement des références à la plate-forme Raspberry Pi 2 : je trouve en effet que grâce à ses ports d'entrées-sorties très accessibles, cette carte bon marché se prête particulièrement bien à des expérimentations et à des premiers prototypages (je suis plus réservé sur son utilisation en tant que plate-forme de déploiement final). J'encourage le lecteur à s'en procurer un exemplaire (ou une carte dans le même esprit comme les Beaglebone Black, Banana Pro, CubieBoard, etc.) et à se familiariser avec les GPIO, interruptions, compilation de noyaux Linux, etc.

J'insiste beaucoup sur la nécessité de réaliser des expériences directement sur la plate-forme cible prévue (ou un équivalent) et, pour cela, nous allons construire de nombreux outils de mesure et de vérification. Plus de 60 exemples (disponibles sur mon site web à l'adresse : http://christophe.blaess.fr) sont présentés dans le livre ; je vous encourage à les télécharger, les compiler, les tester et me faire part de vos remarques éventuelles. Chaque chapitre se termine par quelques exercices permettant de mettre en application les concepts abordés. Leur niveau de difficulté est indiqué par un nombre d'étoiles et leurs corrigés sont disponibles avec les exemples du livre à l'adresse mentionnée ci-dessus.

J'anime fréquemment des sessions de formation et des séminaires sur les aspects industriels de Linux (embarqué, temps réel, drivers) et de nombreux participants m'ont aidé, par leur motivation et leurs questions, à améliorer mes exemples, à approfondir les expérimentations et finalement à mieux comprendre les mécanismes internes de Linux et Xenomai. Je les en remercie chaleureusement.

Copyright © 2015 Eyrolles.

Table des matières

CHAPITRE 1

Multitâche et commutation 1

Multitâche sous Linux 1

Création de processus 3

Parallélisme multithreads 5

Systèmes multiprocesseurs 8

Multiprocesseurs, multicœurs et hyperthreading 8

Affinité d'une tâche 10

États des tâches 17

Ordonnancement 19

Préemption 20

Conclusion 20

Points clés 21

Exercices 21

Exercice 1 (*) 21

Exercice 2 (**) 21

Exercice 3 (**) 22

Exercice 4 (***) 22

CHAPITRE 2

Interruptions, exceptions et appels système 23

Mode noyau 23

Interruptions 24

Principe 24

Entrées-sorties sans interruptions 25

Copyright © 2015 Eyrolles.

Entrées-sorties avec interruptions 26
Interruptions sous Linux 27
Routage des interruptions 28
Exceptions 30
Principe 30
Fichier core 31
Appels système 34
Suivi d'un appel système 34
Threads du noyau 37
Conclusion 39
Points clés 39
Exercices 39
Exercice 1 (*) 39
Exercice 2 (*) 39
Exercice 3 (**) 39
Exercice 4 (**) 40
Exercice 5 (***) 40

CHAPITRE 3

Ordonnancement temps partagé et priorités 41

Temps partagé 41
Principes 41
Ordonnanceur historique 44
Ordonnanceurs du noyau 2.6 45
Ordonnanceur CFS 46
Groupes de processus 47
Autres ordonnanceurs 51
Configuration des priorités 51
Courtoisie des processus 51
Priorités entre threads 53
Conclusion 55
Points clés 55
Exercices 55
Exercice 1 (*) 55
Exercice 2 (*) 56

Copyright © 2015 Eyrolles.

Exercice 3 (**) 56
Exercice 4 (***) 56
Exercice 5 (***) 56

CHAPITRE 4

Limitations de l'ordonnancement temps partagé 57

Mesure du temps 57
Heure Unix avec gettimeofday() 58
Précision des mesures. 58
Horloges Posix 61

Tâches périodiques 64
Timers Unix classiques. 64
Timers Posix. 66
Granularité 68
Précision 70

Préemption des tâches 79

Conclusion 82

Points clés 83

Exercices 83
Exercice 1 (*) 83
Exercice 2 (**) 83
Exercice 3 (**) 83
Exercice 4 (***) 83
Exercice 5 (***) 84

CHAPITRE 5

Principes du temps réel 85

Définitions. 85
Temps réel 85
Classes de temps réel 86
Temps réel absolu 86
Temps réel strict 87
Temps réel strict certifiable 87
Temps réel strict non certifiable 88

Copyright © 2015 Eyrolles.

Temps réel souple 88
Rôles respectifs. 90
Traitement direct dans le noyau 90
Traitement des interruptions. 93
Temps réel sous Linux 94
Échelle des priorités 94
Configuration de l'ordonnancement 96
Processus temps réel. 98
Garde-fou temps réel 100
Threads temps réel 101
Threads en Round Robin 106
Rotation sans Round Robin 109
Temps réel depuis le shell. 110
Ordonnancement EDF (*Earliest Deadline First*) 111
Conclusion 117
Points clés 117
Exercices 118
Exercice 1 (*) 118
Exercice 2 (*) 118
Exercice 3 (**) 118
Exercice 4 (**) 118
Exercice 5 (***) 118

CHAPITRE 6

Performances du temps réel souple 119
Timers temps réel 119
Précisions et fluctuations 119
Granularité des timers 122
Conclusion sur les timers 122
Temps de commutation. 123
Commutation entre threads 123
Commutations entre processus. 127
Comparaison processus et threads 128
Imprévisibilités dues à la mémoire virtuelle 130

Copyright © 2015 Eyrolles.

Préemptibilité du noyau 132
Principes 132
Préemptibilité du noyau standard 134
Connaître la configuration d'un noyau 135
Expériences sur la préemptibilité 136
Conclusion 144
Points clés 145
Exercices 145
Exercice 1 (*) 145
Exercice 2 (**) 145
Exercice 3 (**) 145
Exercice 4 (**) 145
Exercice 5 (***) 146

CHAPITRE 7

Problèmes temps réel classiques 147
Démarrage en Round Robin 147
Barrières Posix 149
Inversion de priorité 150
Principe 150
Héritage de priorité 154
Prise de mutex 156
Comportement en temps réel 159
Reprise de mutex en temps réel 162
Solutions 165
Appel explicite à l'ordonnanceur 167
Conclusion 168
Points clés 169
Exercices 169
Exercice 1 (*) 169
Exercice 2 (**) 169
Exercice 3 (**) 169
Exercice 4 (***) 169
Exercice 5 (***) 169

Copyright © 2015 Eyrolles.

CHAPITRE 8

Limites et améliorations du temps réel Linux 171

Traitement des interruptions 171

PREEMPT_RT 178

Threaded interrupts 181

Fully preemptible kernel 183

Conclusion sur PREEMPT_RT 183

Outils de mesure des performances 184

Cyclictest 184

Hwlatdetect, Hackbench... 185

Économies d'énergie 186

Variation de fréquence d'horloge 187

Heuristique performance 189

Heuristique powersave 190

Heuristique ondemand 190

Conclusion 191

Points clés 191

Exercices 192

Exercice 1 (**) 192

Exercice 2 (**) 192

Exercice 3 (**) 192

Exercice 4 (***) 192

Exercice 5 (***) 192

CHAPITRE 9

Extensions temps réel de Linux 193

Les nanokernels temps réel 193

Principes 193

RTLinux 195

RTAI et Adeos 195

Xenomai 198

Interface de programmation 200

Installation de Xenomai 201

Modification du noyau Linux 201

Copyright © 2015 Eyrolles.

Compilation de Linux modifié 203
Compilation de Xenomai 205
Expériences avec Xenomai 206
Première exploration 206
Programmes de tests 208
Conclusion 211
Points clés 211
Exercices 211
Exercice 1 (*) 211
Exercice 2 (**) 211
Exercice 3 (**) 212
Exercice 4 (***) 212

CHAPITRE 10
Programmer avec Xenomai 213
Programmation de tâches simples 213
Principes 213
Initialisation du processus 215
Création de tâche 216
Compilation et exécution 218
Processus unithread 220
Recherche des changements de modes 221
Alarmes et tâches périodiques 223
Réveils périodiques 223
Alarmes 232
Watchdog 235
Synchronisation des tâches 237
Sémaphores 237
Mutex 240
Conclusion 245
Points clés 246
Exercices 246
Exercice 1 (*) 246
Exercice 2 (**) 246

Copyright © 2015 Eyrolles.

Exercice 3 (**) 246
Exercice 4 (***) 246

CHAPITRE 11

Traitement des interruptions 247

Programmation d'un driver 247
Squelette d'un module 247
Structure d'un driver 249
Traitement des interruptions. 255
Traitement en threaded interrupt 261

Interruptions avec Xenomai 265
Real Time Driver Model 265
Interruptions avec RTDM 271

Conclusion 275

Points clés 276

Exercices 276
Exercice 1 (*) 276
Exercice 2 (**) 276
Exercice 3 (**) 276
Exercice 4 (***) 276

CONCLUSION

État des lieux et perspectives 277

Situation actuelle 277
Linux « vanilla » 277
Patch PREEMPT_RT. 278
Xenomai 278
Mesures 278

Perspectives 279

ANNEXEA

Compilation d'un noyau 281

Préparation des sources 281

Configuration de la compilation 283
Principes. 283

Copyright © 2015 Eyrolles.

Interfaces utilisateur 285
Options de compilation 286
Compilation et installation 287
Compilation croisée 287
Compilation native 288

ANNEXE B

Bibliographie 291
Livres 291
Articles 292
Sites web 292

Index 295

Copyright © 2015 Eyrolles.

Copyright © 2015 Eyrolles.

1

Multitâche et commutation

L'ordonnancement sous Linux est avant tout une affaire de gestion des tâches en attente. Sur un poste de travail courant, il tourne en permanence une bonne centaine de processus, dont la plupart sont endormis, en attente d'événements extérieurs (actions de l'utilisateur, données provenant du réseau, etc.), et quelques-uns seulement sont actifs à un moment donné. Dans ce chapitre, nous allons examiner la représentation et l'état des tâches, ainsi que la notion d'ordonnancement préemptif.

Multitâche sous Linux

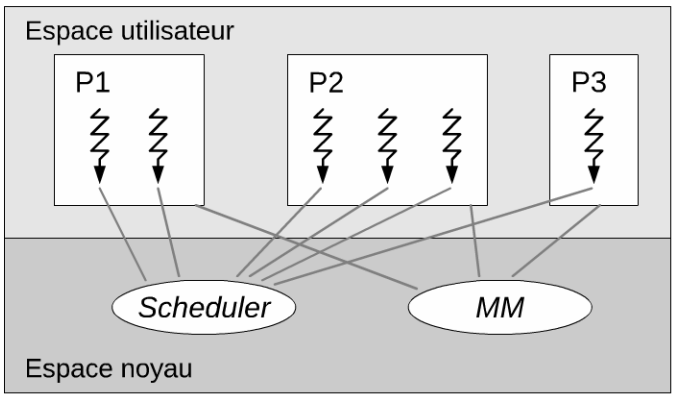
Le fonctionnement multitâche s'exprime traditionnellement sur les systèmes Unix sous forme de **processus**. Pourtant, l'ordonnancement n'est que très faiblement concerné par les processus, et repose en réalité sur la notion de **threads**. Un processus correspond plus exactement à un espace indépendant de mémoire virtuelle, dans lequel un ou plusieurs threads s'exécutent. La notion de threads, notamment leur normalisation par le standard Posix, est relativement récente (années 1990) dans l'histoire des systèmes Unix.

Sur la figure 1-1, nous voyons que la commutation entre les threads est gérée par l'ordonnanceur *(scheduler)* tandis que la gestion des espaces de mémoire virtuelle des processus est assurée par le gestionnaire de mémoire MM *(Memory Manager)* du noyau. Nous pouvons remarquer que le processus `P3` ne comporte qu'un seul thread, c'est le cas de nombreuses applications classiques sous Linux.

Copyright © 2015 Eyrolles.

Figure 1-1

Processus et threads



On peut noter également sur cette figure une frontière entre l'espace utilisateur – celui dans lequel s'exécutent toutes les applications –, et l'espace noyau. Cette frontière est très importante et repose sur une transition du mode de fonctionnement du processeur. Linux s'exécute sur des microprocesseurs qui disposent d'au moins deux modes de fonctionnement.

- Un mode **privilégié**, dans lequel le processeur peut exécuter n'importe quelle opération de son jeu d'instructions défini par le constructeur. Il peut également y réaliser des entrées-sorties directes vers les périphériques matériels externes. Enfin, il lui est possible de modifier la configuration de la mémoire virtuelle à travers le composant MMU (dont nous reparlerons plus loin).
- Un mode **restreint**, dans lequel le processeur est limité à un sous-ensemble de son jeu d'instructions. Dans ce cas, il ne peut accéder qu'à certaines opérations d'entrées-sorties et à certaines plages de mémoire virtuelle qui ont été explicitement configurées depuis le mode privilégié.

Le code du noyau s'exécute toujours en mode privilégié, celui des applications (même celles qui disposent des droits de l'administrateur *root*) uniquement en mode restreint. Ainsi, un processus en mode utilisateur ne pourra ni accéder indûment au matériel, ni toucher des pages de mémoire qui ne lui auraient été volontairement et explicitement accordées par le noyau. Toute tentative de violer ces limites (par exemple, en essayant d'exécuter une instruction assembleur réservée au mode privilégié ou en accédant à une page mémoire non attribuée) se solderait immédiatement par la levée d'une **exception**, c'est-à-dire une interruption interne – aussi appelée une « trappe » sur certains systèmes d'exploitation – qui rendrait immédiatement le contrôle au noyau afin qu'il prenne des dispositions adéquates (comme tuer le processus coupable ou au contraire lui attribuer les ressources demandées).

Lorsqu'un processus s'exécutant en mode utilisateur désire obtenir un accès à un périphérique matériel, par exemple, il devra demander au noyau de réaliser pour lui les opérations voulues (lecture, écriture, paramétrage, projection en mémoire...) en invoquant un **appel système**.

Ces appels système sont des routines d'assistance, où le processus utilisateur sous-traite au noyau certaines opérations nécessitant des privilèges. Avant de réaliser le travail demandé, ce dernier vérifiera que le processus dispose bien de toutes les autorisations adéquates.

Copyright © 2015 Eyrolles.

Création de processus

La création d'un nouveau processus s'effectue *via* l'appel système `fork()`. Celui-ci duplique le processus appelant (traditionnellement appelé « père ») pour créer un nouveau processus (« fils »). Les deux processus continuent leurs progressions en parallèle à partir du point de retour de `fork()`. Le programme exécuté est le même dans les deux processus, mais leur comportement peut s'appuyer sur le code de retour de `fork()` afin de réaliser des opérations différentes. Dans le processus père, `fork()` renvoie le PID *(Process IDentifier)* du fils nouvellement créé, tandis que dans le fils, `fork()` renvoie toujours zéro.

```
pid_t fork (void);
```

La fin du processus survient lors d'un appel `exit()` – ou le retour de la fonction `main()` – ou si le processus reçoit un signal fatal qu'il ne traite pas. Le processus père peut attendre la fin de son fils en invoquant `waitpid()`. Cet appel système lui permet de connaître, par l'intermédiaire d'une variable entière, les conditions de terminaison de son fils (a-t-il appelé `exit()` ? si oui, avec quel argument ? sinon, a-t-il été tué par un signal ? lequel ? etc.).

```
void exit (int status);
pid_t waitpid (pid_t fils, int * status, int options);
```

Enfin, un processus peut charger dans sa mémoire un nouveau code exécutable, abandonnant totalement son programme précédent pour commencer le déroulement d'une nouvelle fonction `main()`. Ceci s'effectue avec l'une des fonctions de la famille `exec()`, dont seul `execve()` est réellement un appel système, les autres étant des fonctions de bibliothèques qui arrangent leurs arguments avant de l'invoquer.

```
int execve (const char *fichier, char *const argv[],
       char *const envp[]);
int execl  (const char *fichier, const char *arg,...);
int execle (const char *fichier, const char *arg, ...,
       char * const envp[]);
int execlp (const char *fichier, const char *arg, ...);
int execv  (const char *fichier, char *const argv[]);
int execvp (const char *fichier, char *const argv[]);
```

Avec cet ensemble de primitives système – `fork()`, `execve()`, `exit()`, `waitpid()` –, on peut organiser tout le multitâche classique Unix fondé sur des processus. Bien sûr, des fonctions de bibliothèques comme `system()` ou `posix_spawn()` simplifient le travail du programmeur en encadrant ces appels système et rendent plus aisé le démarrage d'un nouveau processus.

Voici un petit programme qui se présente comme un shell (très) minimal, il lit des lignes de commandes et les fait exécuter par un processus fils.

```
exemple-processus :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
```

Copyright © 2015 Eyrolles.

```
#define LG_LIGNE 256

int main(void)
{
  char ligne[LG_LIGNE];

  while(1) {
    // Afficher un symbole d'invite (prompt)
    fprintf(stderr, "--> ");
    // Lire une ligne de commandes
    if (fgets(ligne, LG_LIGNE, stdin) == NULL)
      break;
    // Supprimer le retour chariot final
    ligne[strlen(ligne)-1] = '\0';
    // Lancer un processus
    if (fork() == 0) {
      // --- PROCESSUS FILS ---
      // Exécuter la commande
      execlp(ligne, ligne, NULL);
      // Message d'erreur si on échoue
      perror(ligne);
      exit(EXIT_FAILURE);
    } else {
      // --- PROCESSUS PÈRE ---
      // Attendre la fin de son fils
      waitpid(-1, NULL, 0);
      // Et reprendre la boucle
    }
  }
  fprintf(stderr, "\n");
  return EXIT_SUCCESS;
}
```

Lors de son exécution, on peut lui faire réaliser quelques commandes simples :

```
$ ./exemple-processus
--> ls
exemple-processus  exemple-processus.c    Makefile
--> date
dim. mars 27 22:29:06 CEST 2011
--> who
cpb      tty1         2011-03-27 19:12 (:0)
cpb      pts/0        2011-03-27 21:52 (:0.0)
cpb      pts/1        2011-03-27 21:56 (:0.0)
-->
```

Copyright © 2015 Eyrolles.

Toutefois, dès que l'on essaie de passer des arguments sur la ligne de commandes, la fonction `execlp()` recherche un fichier exécutable du nom complet (y compris les espaces et arguments) et échoue :

```
--> ls -l
ls -l: No such file or directory
-->   (Contrôle-D)
$
```

Si l'on souhaitait écrire un vrai interpréteur de commandes, il faudrait analyser la ligne saisie, découper les mots, etc. Ce petit programme est toutefois intéressant dans sa gestion des primitives de base du multitâche Unix. Ce sont les seules dont on disposait de manière standard jusque dans les années 1990 environ.

Parallélisme multithreads

Au cours des années 1980, plusieurs implémentations ont été proposées pour obtenir un mécanisme multitâche léger, fonctionnant à l'intérieur de l'espace mémoire d'un processus. Certaines s'appuyaient sur une commutation entre tâches organisées au sein même du processus – par une bibliothèque –, tandis que d'autres réclamaient une extension des primitives Unix classiques pour permettre à plusieurs tâches de partager le même espace mémoire. Dans les années 1990, une volonté d'uniformisation de l'API des systèmes Unix a donné naissance à la norme Posix, dont une section *(Posix.1c)* était consacrée aux threads. Cette série de fonctions permet de gérer des *Posix Threads*, aussi appelés « Pthreads ».

La création d'un nouveau thread s'obtient en appelant `pthread_create()` à qui on indique la fonction sur laquelle le thread nouvellement créé devra démarrer. L'identifiant du thread (de type `pthread_t`) sera renseigné durant cet appel. On peut également préciser des attributs spécifiques pour le thread et un argument pour la fonction à exécuter. Nous verrons ultérieurement des attributs (enregistrés dans l'objet `pthread_attr_t`) ; pour le moment nous nous contenterons de passer un pointeur `NULL` en second argument de `pthread_create()`.

```
int pthread_create (pthread_t * thread,
                    pthread_attr_t * attr,
                    void * (*fonction) (void *),
                    void * argument);
```

Dès que la fonction `pthread_create()` se termine avec succès, nous savons qu'un nouveau fil d'exécution se déroule dans notre processus.

La fin de ce thread se produira lorsqu'il invoquera `pthread_exit()` ou terminera sa fonction principale par un `return`, en renvoyant un pointeur (éventuellement `NULL` s'il n'a rien de particulier à retourner).

```
void pthread_exit (void * valeur);
```

Copyright © 2015 Eyrolles.

Le pointeur renvoyé lors de la terminaison peut être récupéré par n'importe quel autre thread qui invoque `pthread_join()`.

```
int pthread_join (pthread_t thread, void ** valeur);
```

Dans l'exemple suivant, le thread `main()` de notre programme va démarrer autant de nouveaux threads qu'on lui a passé d'arguments sur sa ligne de commandes. Chacun d'entre eux recevra en paramètre de sa fonction principale un nombre (passé – à travers un *cast* – dans un pointeur). Chaque thread calculera alors la factorielle de son nombre en effectuant des boucles et en nous affichant sa progression. Le résultat du calcul sera renvoyé à la fin de la fonction du thread, et récupéré dans le thread `main()`. L'intérêt de ce programme est d'utiliser les différentes primitives que nous avons présentées précédemment.

exemple-threads.c :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * factorielle(void * arg)
{
    int i;
    long n = (long) arg;
    long resultat = 1;
    for (i = 2; i <= n; i ++) {
        resultat = resultat * i;
        fprintf(stderr, "%ld! : en calcul...\n", n);
        sleep(1);
    }
    return (void *) resultat;
}

int main(int argc, char * argv[])
{
    pthread_t * threads = NULL;
    void * retour;
    int i;
    long n;
    // Vérification des arguments
    if (argc < 2) {
        fprintf(stderr, "usage: %s valeurs..\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // Allocation d'un tableau d'identifiants
    threads = calloc(argc-1, sizeof(pthread_t));
    if (threads == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }
```

Copyright © 2015 Eyrolles.

```
    fprintf(stderr, "main(): lancement des threads\n");
    for (i = 1; i < argc; i ++) {
        // Récupération de l'argument numérique
        if (sscanf(argv[i], "%ld", & n) != 1) {
            fprintf(stderr, "%s: invalide\n", argv[i]);
            exit(EXIT_FAILURE);
        }
        // Lancement du thread
        if (pthread_create(& (threads[i-1]), NULL,
                        factorielle, (void *) n) != 0) {
            fprintf(stderr,
               "Impossible de demarrer le thread %d\n", i);
            exit(EXIT_FAILURE);
        }
    }
    fprintf(stderr, "main(): tous threads lances\n");
    for (i = 1; i < argc; i ++) {
        // Attente du thread
        pthread_join(threads[i-1], & retour);
        fprintf(stderr, "main(): %s! = %ld\n",
                 argv[i], (long) retour);
    }
    fprintf(stderr, "main(): tous threads termines\n");
    free(threads);
    return EXIT_SUCCESS;
}
```

Voici un petit exemple d'exécution :

```
$ ./exemple-threads 3 5 7
main(): lancement des threads
main(): tous threads lances
7! : en calcul...
5! : en calcul...
3! : en calcul...
7! : en calcul...
5! : en calcul...
main(): 3! = 6
7! : en calcul...
5! : en calcul...
7! : en calcul...
main(): 5! = 120
7! : en calcul...
main(): 7! = 5040
main(): tous threads termines
$
```

Copyright © 2015 Eyrolles.

Systèmes multiprocesseurs

Multiprocesseurs, multicœurs et hyperthreading

Une part importante (et croissante) des ordinateurs actuels offre un degré plus ou moins avancé de parallélisation physique des traitements. À cet égard, le noyau Linux considère qu'il existe deux types de systèmes : les machines uniprocesseur sur lesquelles un seul fil d'exécution est présent à un moment donné, et les machines multiprocesseurs symétriques (SMP) qui permettent l'exécution parallèle de plusieurs tâches. Ceci regroupe les plates-formes :

- multiprocesseurs contenant réellement plusieurs processeurs physiques distincts ;
- multicœurs : un seul processeur est présent, mais il dispose de plusieurs décodeurs d'instructions travaillant en parallèle ;
- *hyperthreading*[1] : un seul processeur assure une commutation entre deux séquences d'instructions différentes.

On retrouve bien sûr des combinaisons de ces différentes possibilités. Ainsi, une machine disposant de deux processeurs physiques différents, chacun d'eux comportant deux cœurs et chaque cœur étant hyperthreadé, donne à l'utilisateur la sensation de disposer de huit processeurs virtuels différents.

Pour connaître les caractéristiques du processeur, telles qu'elles ont été détectées par le noyau Linux, on peut interroger le pseudo-fichier `/proc/cpuinfo`. Voici un exemple sur un processeur Intel à quatre cœurs :

```
$ cat /proc/cpuinfo
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 58
model name  : Intel(R) Core (TM) i3-3100M CPU @ 2.40 GHz
stepping    : 9
cpu MHz     : 1273.406
cache size  : 3072 KB
physical id : 0
siblings    : 4
core id     : 0
cpu cores   : 2
[...]
processor   : 1
[...]
processor   : 2
[...]
processor   : 3
```

1. Le mot *hyperthreading* correspond à une implémentation particulière (celle d'Intel) d'un concept plus général nommé *Simultaneous Multi Threading*. Je préfère néanmoins conserver ce terme car il est beaucoup plus répandu.

Copyright © 2015 Eyrolles.

```
vendor_id   : GenuineIntel
cpu family  : 6
model       : 58
model name  : Intel(R) Core(TM) i3-3110M CPU @ 2.40GHz
stepping    : 9
cpu MHz     : 1200.093
cache size  : 3072 KB
physical id : 0
siblings    : 4
core id     : 1
cpu cores   : 2
[…]
$
```

Nous voyons que nous disposons de quatre processeurs virtuels (lignes `processor : 0` à `processor : 3`). Toutefois, il s'agit du même processeur physique (lignes `physical id : 0` identiques) qui contient deux cœurs distincts (lignes `core id : 0` et `core id : 1`). Chacun des cœurs est hyperthreadé. On peut remarquer que la fréquence CPU est différente sur des deux cœurs affichés, en effet sur de nombreux processeurs cette fréquence est modifiable dynamiquement. Ici, le noyau Linux la fait évoluer en fonction de la charge du système. Nous reviendrons sur ce sujet dans le chapitre 8.

La commande `lscpu`, disponible sur de nombreux systèmes Linux, présente de manière plus lisible le contenu de `/proc/cpuinfo` :

```
$ lscpu
Architecture:          x86_64
Mode(s) opératoire(s) des processeurs : 32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) par cœur :   2
Cœur(s) par socket :   2
Socket(s):             1
Nœud(s) NUMA :         1
Identifiant constructeur : GenuineIntel
Famille de processeur : 6
Modèle :               58
Révision :             9
Vitesse du processeur en MHz : 1199.906
BogoMIPS:              4789.00
Virtualisation :       VT-x
Cache L1d :            32K
Cache L1i :            32K
Cache L2 :             256K
Cache L3 :             3072K
NUMA node0 CPU(s):     0-3
$
```

Copyright © 2015 Eyrolles.

Parfois `lscpu` est moins volubile, en voici un exemple d'exécution sur Raspberry Pi 2 :

```
$ lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
$
```

Au sein d'une application, il est possible d'appeler la fonction :

```
#include <unistd.h>
long sysconf (int nom);
```

Toutefois, avec l'argument `_SC_NPROCESSORS_ONLN`, il est important de savoir que cette fonctionnalité n'est pas normalisée et ne fonctionnera peut-être que sous Linux. En voici un exemple :

```
exemple-sysconf.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Nombre de CPU: %ld\n",
            sysconf(_SC_NPROCESSORS_ONLN));
    return EXIT_SUCCESS;
}
```

Qui nous donne :

```
$ ./exemple-sysconf
Nombre de CPU: 4
$
```

Affinité d'une tâche

L'ordonnanceur de Linux est capable de placer (et de déplacer) des tâches sur les processeurs virtuels du système.

Comme représenté sur la figure 1-2, nous lançons une boucle infinie et active (consommatrice de temps CPU) depuis un shell. L'utilitaire GkrellM *(Gnome Krell Monitor)* – que je recommande vivement – présent à droite de la fenêtre affiche graphiquement les activités des quatre cœurs du processeur, mises à jour toutes les secondes. Nous voyons que toute la charge liée à la boucle active est d'abord portée par le cœur 0, puis l'ordonnanceur décide de migrer la tâche (pour répartir la charge globale sur les processeurs disponibles) vers le cœur 2. On

Copyright © 2015 Eyrolles.

remarque un bref passage sur le cœur 1 pendant quelques secondes et un retour prolongé sur le cœur 2.

Figure 1-2

Migration de tâches

Cette migration intervient dynamiquement et spontanément, sans que le processus concerné ne s'en rende compte.

Nous pouvons vérifier à tout moment le processeur sur lequel s'exécute une tâche avec la fonction (spécifique GNU) :

```
int sched_getcpu (void);
```

qui nous renvoie le numéro de processeur depuis lequel elle a été invoquée, ou `-1` si elle ne peut le déterminer.

Le petit programme suivant va vérifier en permanence son emplacement et nous indiquer lorsqu'il détectera des migrations :

exemple-sched-getcpu.c :

```
#define _GNU_SOURCE // sched_getcpu() extension GNU
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

int main(void)
{
    int n;
    int precedent = -1;
    time_t heure;
    struct tm * tm_heure;
```

Copyright © 2015 Eyrolles.

```
    while (1) {
        n=sched_getcpu();
        if (n == -1) {
            perror("sched_getcpu");
            exit(EXIT_FAILURE);
        }
        if (precedent == -1)
            precedent = n;
        if (n != precedent) {
            heure = time(NULL);
            tm_heure = localtime(& heure);
            print("%02d:%02d:%02d migration %d -> %d\n",
                  tm_heure->tm_hour, tm_heure->tm_min,
                  tm_heure->tm_sec,
                  precedent, n);
            precedent = n;
        }
    }
    return EXIT_SUCCESS;
}
```

Nous voyons que le noyau déplace le processus alternativement sur les quatre cœurs en fonction de la charge du système.

```
$ ./exemple-sched-getcpu
08:29:40 migration 3 -> 0
08:30:01 migration 0 -> 2
08:30:07 migration 2 -> 0
08:30:12 migration 0 -> 1
08:30:12 migration 1 -> 2
08:30:15 migration 2 -> 3
08:30:21 migration 3 -> 0
  (Contrôle-C)
$
```

Il nous est également possible de contrôler l'emplacement d'un processus, avant son lancement ou pendant son exécution. Pour cela, la commande shell `taskset` est très utile. Toutefois, son utilisation n'est pas vraiment intuitive. En voici quelques exemples :

```
$ taskset -c 1  ./commande
        // Lance la commande sur le CPU 1
$ taskset 0,2  ./commande
        // Autorise la commande à s'exécuter sur les CPU 0 et 2
$ taskset -pc 0  1234
        // Migre le processus 1234 sur le CPU 0
$ taskset -p 1234
        // Affiche l'affinité du processus 1234
```

Copyright © 2015 Eyrolles.

L'affinité d'une tâche est la liste des CPU sur lesquels elle peut s'exécuter. On peut la consulter ou la fixer à l'aide des fonctions suivantes :

```
int sched_setaffinity (pid_t             pid,
                       size_t            taille,
                       const cpu_set_t * cpuset);

int sched_getaffinity (pid_t       pid,
                       size_t      taille,
                       cpu_set_t * cpuset);
```

Ces fonctions sont des extensions GNU – non portables sur d'autres systèmes Unix – qui nécessitent donc de définir la constante symbolique `_GNU_SOURCE` avant d'inclure `<sched.h>`.

Le second argument de ces fonctions correspond à la taille du type de donnée `cpu_set_t` pour le système.

Les listes de CPU sont représentées par les variables de type `cpu_set_t`, que l'on manipule avec les fonctions suivantes :

```
void CPU_ZERO  (cpu_set_t * ensemble);
void CPU_SET   (int cpu, cpu_set_t * ensemble);
void CPU_CLR   (int cpu, cpu_set_t * ensemble);
int  CPU_ISSET (int cpu, cpu_set_t * ensemble);
```

La fonction `CPU_ZERO()` permet de vider un ensemble, `CPU_SET()` et `CPU_CLR()` permettent respectivement d'insérer ou de retirer un CPU d'un ensemble, enfin `CPU_ISSET()` permet de vérifier si un CPU est mentionné dans un ensemble ou non.

Voici un petit exemple d'accrochage d'un processus sur un CPU donné en argument. On notera que la valeur 0 en premier argument de `sched_setaffiny()` indique que l'on fixe le masque d'affinité pour le processus appelant.

exemple-sched-setaffinity.c :

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

int main(int argc, char *argv[])
{
    cpu_set_t  cpuset;
    int cpu;

    // Lire le numéro de CPU dans le premier argument
    if ((argc != 2) || (sscanf(argv[1],"%d", &cpu)!=1)) {
        fprintf(stderr, "usage: %s cpu\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // Remplir l'ensemble avec le CPU indique
```

Copyright © 2015 Eyrolles.

```
    CPU_ZERO(& cpuset);
    CPU_SET(cpu, &cpuset);

    // Fixer l'affinité
    if (sched_setaffinity(0, sizeof(cpuset), &cpuset)!=0){
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    while (1) {
        printf("Je suis sur le CPU %d\n", sched_getcpu());
        sleep(1);
    }
    return EXIT_SUCCESS;
}
```

Quelques exemples d'exécution, sur un système à quatre cœurs :

```
$ ./exemple-sched-setaffinity 0
Je suis sur le CPU 0
Je suis sur le CPU 0
Je suis sur le CPU 0
Je suis sur le CPU 0
  Contrôle-C
$ ./exemple-sched-setaffinity 3
Je suis sur le CPU 3
Je suis sur le CPU 3
Je suis sur le CPU 3
Je suis sur le CPU 3
  Contrôle-C
$ ./exemple-sched-setaffinity 4
4: Invalid argument
$
```

On remarque que `sched-setaffinity()` échoue si le numéro de CPU est invalide.

Il est également possible de fixer l'affinité d'un thread dans un processus. On peut ainsi tirer parti du parallélisme physique du système au sein d'une application.

Ces fonctions ne sont pas décrites dans la norme Pthreads *(Posix.1c)*, aussi leur ajoute-t-on le suffixe `_np` *(non Posix)* pour indiquer leur manque de portabilité. On peut fixer – et lire – l'affinité d'un thread dynamiquement avec :

```
int pthread_setaffinity_np (pthread_t         thread,
                            size_t            taille,
                            const cpu_set_t * cpuset);
int pthread_getaffinity_np (pthread_t   thread,
                            size_t      taille,
                            cpu_set_t * cpuset);
```

Copyright © 2015 Eyrolles.

On peut aussi fixer l'affinité d'un futur thread avant sa création. Pour cela, on initialise un objet `pthread_attr_t` qui contient les attributs du thread à créer et on passe cette structure en second argument de `pthread_create()`.

Pour remplir la structure d'attributs, on utilise les fonctions :

```
int pthread_attr_setaffinity_np (
                            pthread_attr_t    attr,
                            size_t            taille,
                            const cpu_set_t * cpuset);
int pthread_attr_getaffinity_np (
                            pthread_attr_t   attr,
                            size_t           taille,
                            cpu_set_t *      cpuset);
```

Dans l'exemple suivant, nous allons lancer autant de threads en parallèle qu'il y a de processeurs disponibles. Nous commençons par le CPU 0, puis incrémentons le numéro jusqu'à ce que l'appel `pthread_create()` échoue. Notons que la variable de type `pthread_t` est écrasée à chaque appel avec le nouvel identifiant affecté au thread créé.

exemple_pthread_attr_setaffiniy.c :

```
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

void * fonction(void * arg)
{
    long num = (long) arg;
    while (1) {
        printf("[Thread %ld] Je suis sur le CPU %d\n",
                num, sched_getcpu());
        sleep(5);
    }
}

int main(void)
{
    cpu_set_t       cpuset;
    pthread_t       thr;
    pthread_attr_t  attr;
    long num;

    num = 0;
    while (1) {
        // Initialiser avec les attributs par défaut
        pthread_attr_init(& attr);
        // Préparer le cpuset
        CPU_ZERO(&(cpuset));
```

Copyright © 2015 Eyrolles.

```
        CPU_SET(num, &(cpuset));
        // Fixer l'affinité
        pthread_attr_setaffinity_np(& attr,
                                    sizeof(cpu_set_t),
                                    & cpuset);
        // Lancer le thread
        if (pthread_create(& thr, & attr, fonction,
                           (void *) num) != 0)
            break;
        num++;
    }
    // Terminer le thread main en continuant les autres
    pthread_exit(NULL);
}
```

Exécutons ce programme sur une machine possédant quatre cœurs :

```
$ ./exemple-pthread-attr-setaffinity
[Thread 0] Je suis sur le CPU 0
[Thread 1] Je suis sur le CPU 1
[Thread 2] Je suis sur le CPU 2
[Thread 3] Je suis sur le CPU 3
[Thread 0] Je suis sur le CPU 0
[Thread 1] Je suis sur le CPU 1
[Thread 2] Je suis sur le CPU 2
[Thread 3] Je suis sur le CPU 3
[Thread 1] Je suis sur le CPU 1
[Thread 3] Je suis sur le CPU 3
[Thread 2] Je suis sur le CPU 2
[Thread 0] Je suis sur le CPU 0
   Contrôle-C
$
```

On peut remarquer que l'ordre d'affichage des messages n'est pas figé. Les threads faisant des accès simultanés à la console, le noyau est obligé de se livrer à un arbitrage qui peut évoluer en fonction de l'activité du système. Nous allons examiner les différents états dans lesquels peuvent se trouver les tâches et les transitions possibles entre ces états.

Il est important de noter que l'affinité ainsi fixée ne présente pas de caractère obligatoire pour la tâche. À tout moment, il lui est possible de modifier son affinité et de se déplacer ainsi vers un autre processeur. Ce principe est tout à fait suffisant pour les applications temps réel, où toutes les tâches d'un système sont habituellement configurées pour un fonctionnement optimal. Si on désire verrouiller impérativement une tâche sur un CPU – ou un ensemble de CPU – sans qu'elle puisse en sortir ensuite, on utilisera plutôt le mécanisme des *cpuset*. Un exemple clair est présenté dans la page de manuel `cpuset(7)` accessible ainsi :

```
$ man 7 cpuset
```

Copyright © 2015 Eyrolles.

États des tâches

Utilisons la commande `ps aux` sur un système Linux de type bureautique :

```
$ ps aux
USER  PID %CPU %MEM [...]  STAT  START   TIME  COMMAND
root    1  0.1  0.0        Ss    14:17   1:14  /sbin/init
root    2  0.0  0.0        S     14:17   0:12  [kthreadd]
root    3  0.0  0.0        S     14:17   0:36  [ksoftirqd/0]
[...]
cpb  2760  0.4  0.6        Sl    20:47   0:01  gnome-term
cpb  2767  0.0  0.0        S     20:47   0:00  gnome-pty-h
cpb  2768  0.1  0.0        Ss    20:47   0:00  bash
cpb  2807  2.0  0.0        R+    20:48   0:00  ps aux
[~]$
```

Nous observons la présence de quelques dizaines de processus, décrits par divers champs. Les lignes précédentes ont été éditées pour supprimer quelques champs peu intéressants pour le moment.

Comme nous souhaitons des informations sur l'ordonnancement, nous préférons obtenir des statistiques sur les tâches du système, c'est-à-dire des threads. Pour cela, la commande `ps maux` est beaucoup plus utile car elle décrit l'activité des différents threads au sein de chaque processus.

```
$ ps maux
USER  PID %CPU %MEM [...]  STAT  START   TIME COMMAND
root    1  0.0  0.0          -   14:17   1:14 /sbin/init
root    -  0.0    -          Ss  14:17   1:14 -
root    2  0.0  0.0          -   14:17   0:12 [kthreadd]
root    -  0.0    -          S   14:17   0:12 -
root    3  0.0  0.0          -   14:17   0:36 [ksoftirqd/0]
root    -  0.0    -          S   14:17   0:36 -
[...]
cpb  2760  0.2  0.6          -   20:47   0:01 gnome-term
cpb     -  0.2    -          Sl  20:47   0:01 -
cpb     -  0.0    -          Sl  20:47   0:00 -
cpb     -  0.0    -          Sl  20:47   0:00 -
cpb  2767  0.0  0.0          -   20:47   0:00 gnome-pty-h
cpb     -  0.0    -          S   20:47   0:00 -
cpb  2768  0.0  0.0          -   20:47   0:00 bash
cpb     -  0.0    -          Ss  20:47   0:00 -
cpb  3015  4.0  0.0          -   20:52   0:00 ps maux
cpb     -  4.0    -          R+  20:52   0:00 -
$
```

Nous voyons, pour chaque processus, une première ligne – avec le champ PID rempli – récapitulant les informations globales, puis une ou plusieurs lignes pour chacun des threads. Les champs n'ayant pas de valeur spécifique pour le thread – par exemple, le PID – contiennent un tiret.

Copyright © 2015 Eyrolles.

Nous pouvons voir que tous les processus disposent au moins d'un thread (celui qui exécute la fonction `main()`) et que certains d'entre eux (par exemple, `gnome-terminal` pour ses onglets) disposent de plusieurs threads.

Dans les statistiques fournies, nous remarquons la présence d'un champ `STAT` nous renseignant sur l'état du thread. Ce qui nous concerne ici est la première lettre de ce champ. On peut y rencontrer les états suivants.

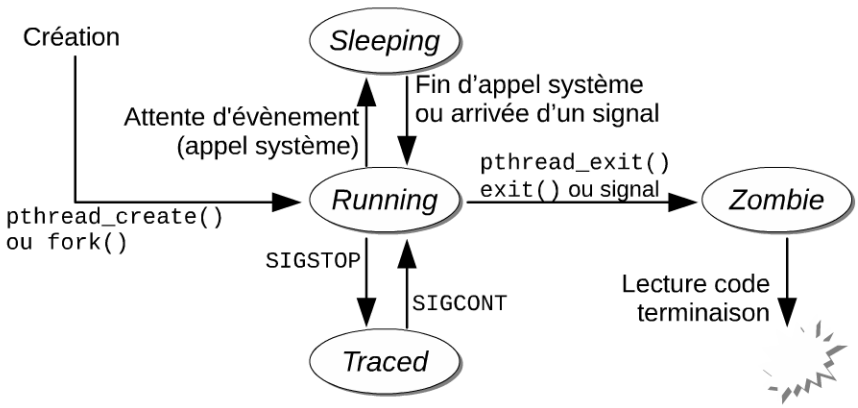
Lettre	État	Signification
`S`	*Sleeping*	Endormi. Le thread est inactif, il est bloqué dans un appel système, probablement en attente de données extérieures. Dès que les données seront disponibles, il sera réveillé et passera en état *Running*. Si un signal est envoyé au thread, il sera réveillé prématurément pour traiter le signal. En l'absence de traitement correct du signal, le thread sera tué.
`R`	*Running*	En exécution. Le thread est actif, il réalise des opérations et consomme du temps CPU.
`D`	*Down*	Bloqué. Cet état très rare signifie que le thread est endormi dans un sommeil profond en attente d'événements externes. Il n'est pas possible de le réveiller prématurément en lui envoyant un signal, même le fameux signal 9 (`SIGKILL`).
`T`	*Traced*	Suivi. Ce mode est utilisé par les débogueurs pour geler le thread. Il ne s'exécute plus mais ne disparaît pas pour autant. Le passage à l'état *Traced* s'obtient la plupart du temps sur l'arrivée d'un signal `SIGSTOP`. Le thread reprendra son exécution normale sur réception d'un signal `SIGCONT`.
`Z`	*Zombie*	Le thread s'est terminé (volontairement avec `pthread_exit()` ou `exit()`, involontairement sur réception d'un signal). Il a laissé un code de terminaison qui persiste jusqu'à ce qu'il soit lu par un autre thread ou par le processus père.

Les transitions entre les états d'une tâche sont toujours assurées par le noyau. La plupart du temps, elles s'expriment comme la conséquence de l'arrivée d'une interruption. Nous reviendrons sur ce sujet dans le prochain chapitre.

La figure 1-3 présente les différents états possibles, ainsi que leurs transitions habituelles. Par souci de simplification, on n'y distingue pas l'état *Sleeping* de l'état *Down*, la tâche étant en sommeil dans les deux cas (dans le second état, elle ne peut toutefois pas être réveillée par l'arrivée d'un signal).

Copyright © 2015 Eyrolles.

Figure 1-3
États d'une tâche



Ordonnancement

Bien entendu, lorsque nous disposons de quatre CPU, par exemple, il peut y avoir jusqu'à quatre tâches actives *(Running)* simultanément. Toutefois, sur la plupart des systèmes, il arrive que le nombre de tâches demandé excède le nombre de processeurs disponibles. Il faut donc organiser des commutations entre ces tâches, c'est le rôle de l'ordonnanceur *(scheduler)*.

L'ordonnanceur profite de la mise en sommeil d'une tâche (qui attend un événement externe, par exemple) pour en activer une autre. Mais il faut également disposer d'un nouvel état pour représenter une tâche qui est prête à s'exécuter, et en attente de la libération du processeur. C'est l'état *Runnable* (Prêt) qui s'intercale entre les états *Sleeping* (ou *Down*) et *Running*.

En zoomant sur la partie centrale de la figure 1-3, celle-ci devient donc la figure ci-après.

Lorsqu'une tâche est réveillée (ou lorsqu'elle démarre, même si ce n'est pas visible sur la figure 1-4), elle passe d'abord dans un état d'attente d'où l'ordonnanceur l'extraira pour la placer sur le processeur. Il peut y avoir plusieurs tâches prêtes simultanément (on peut prendre l'image d'une salle d'attente), et le scheduler va devoir choisir une tâche à activer. On parle d'un mécanisme d'élection. Avec un ordonnancement temps partagé, toute la qualité du scheduler, toute son intelligence, résidera dans la pertinence de ce choix. Nous reviendrons sur ce point au chapitre 3.

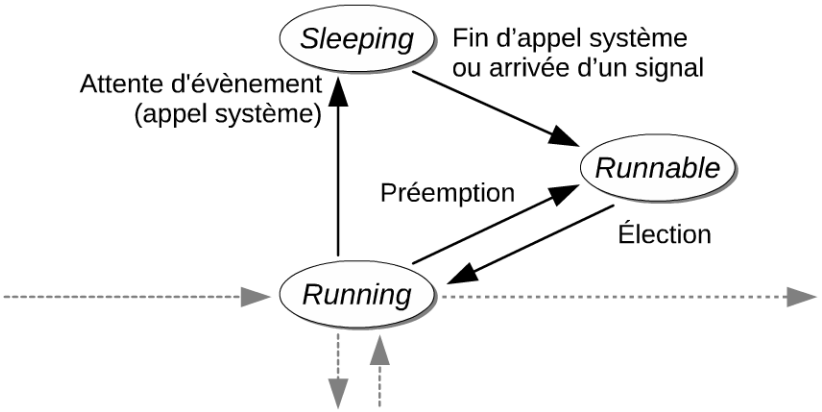
Notons que depuis l'extérieur du kernel, il n'est pas possible de distinguer une tâche qui est *Runnable* d'une tâche qui est effectivement *Running*. Ainsi, `ps` nous les présente-t-il toutes deux avec la lettre `'R'`. Lorsque plusieurs tâches apparaissent avec cet état `'R'` sur un système uniprocesseur, il n'y en a évidemment qu'une seule dans l'état *Running*, les autres étant en *Runnable*.

Le nombre de tâches prêtes à un moment donné est appelé « charge du système ». On effectue généralement une moyenne sur quelques dizaines de secondes pour avoir une valeur significative. S'il y a eu en moyenne 2,5 tâches *Runnable* sur une période donnée, on dit que la charge fut de 250 %. Naturellement, il faut rapporter cette valeur au nombre de processeurs présents pour qu'elle soit représentative.

Copyright © 2015 Eyrolles.

Figure 1-4

État Prêt (Runnable)



Préemption

Lorsqu'une tâche s'est exécutée pendant un temps prolongé au détriment des autres, le système peut décider de lui retirer temporairement le processeur et de la réinjecter dans la liste des tâches *Runnable*. On dit que la tâche a été préemptée, et plus généralement qu'il s'agit d'un système multitâche préemptif.

À l'inverse, un ordonnancement peut être collaboratif, c'est-à-dire que chaque tâche va volontairement (lors d'un appel à la boucle d'événements de l'interface graphique, par exemple) céder régulièrement le CPU – quitte à le retrouver immédiatement si elle est la seule en activité. Un exemple typique était le système Microsoft Windows 3.1.

La plupart des ordonnancements temps partagé actuels sont préemptifs. En revanche, la préemption peut être gênante pour des systèmes temps réel, aussi verrons-nous qu'il est possible de configurer le comportement des tâches temps réel.

Conclusion

Dans ce chapitre, nous avons mis en place les éléments essentiels du multitâche, vu de l'espace utilisateur. En ce qui concerne l'API disponible sous Linux pour la programmation multithreads ou multiprocessus, on trouvera des éléments détaillés dans [BLAESS 2011], *Développement système sous Linux*. Le prochain chapitre va nous permettre de mieux comprendre les transitions entre les états des tâches et les interactions entre l'espace applicatif (espace utilisateur) et celui du noyau.

Copyright © 2015 Eyrolles.

Points clés

Les processus représentent avant tout des espaces mémoire indépendants. Il y a une filiation entre processus. La fin d'un processus est notifiée à son créateur. Les threads d'un processus s'exécutent dans le même espace mémoire et partagent les variables globales.

Le noyau Linux gère de manière homogène les systèmes multiprocesseurs, multicœurs et les processeurs hyperthreads. Il fait migrer des tâches d'un processeur à l'autre en fonction de la charge du système. On peut fixer l'affinité d'une tâche (thread ou processus complet), c'est-à-dire son attachement à un ou plusieurs processeurs.

Pour partager le temps processeur disponible, les tâches sont ordonnancées et alternent entre les états *Sleeping* (endormie, en attente de réveil), *Running* (active, en exécution) et *Runnable* (prête à s'exécuter, attendant la disponibilité du processeur).

Exercices

Les quelques exercices présentés en fin de chapitre permettent de revenir sur les concepts abordés précédemment. Les explications et codes sources des solutions sont disponibles sur le site web de l'auteur :

http://christophe.blaess.fr/

Exercice 1 (*)

En consultant le fichier `/proc/cpuinfo`, déterminez l'architecture des différents systèmes Linux à votre disposition (uniprocesseur ? multiprocesseurs ? nombre de processeurs physiques, nombre de cœurs, présence d'*hyperthreading* ?)

Exercice 2 (**)

Écrivez un petit programme qui démarre autant de threads qu'il y a de processeurs (au sens large, en comptant les cœurs et l'*hyperthreading*). Chaque thread devra se placer sur un processeur distinct, puis ils entameront à tour de rôle des séquences de boucles actives comme :

```
for (i = 0 ; i < 1000000000 ; i ++)
    ;
sleep(10);
```

En examinant régulièrement le résultat de la commande `ps maux`, observez l'activité sur chaque processeur. Je vous conseille par ailleurs d'installer sur votre système l'utilitaire GkrellM qui vous permettra de suivre en temps réel la charge de chaque processeur.

Copyright © 2015 Eyrolles.

Exercice 3 (**)

Créez un programme qui exécute une boucle active en incrémentant un compteur, puis se met en sommeil pendant quelques secondes – avec l'appel système `sleep()`. En exécutant `ps aux` depuis un autre terminal, observez bien les états *Running* et *Sleeping*. Envoyez à votre processus un signal `STOP` avec la commande :

```
$ kill -STOP  (pid_du_processus)
```

Quel est son état ?

Envoyez-lui le signal `CONT` avec une commande similaire. Que se passe-t-il ?

Exercice 4 (***)

Écrivez un programme qui crée un processus fils avec l'appel `fork()`. Ce processus fils doit se terminer au bout de dix secondes, tandis que le père va boucler autour de l'appel `sleep(1)`. Vérifiez le déroulement depuis un autre terminal avec `ps aux`. Que devient le processus fils après sa terminaison ? Comment s'en débarrasser et quel est le mécanisme sous-jacent ?

Copyright © 2015 Eyrolles.

2

Interruptions, exceptions et appels système

Le principe des interruptions est un élément clé des systèmes temps réel car il permet quasi instantanément de consacrer temporairement les ressources du processeur à la réponse à un événement externe. Du temps de réponse à une interruption et de la constance de ce temps de réponse dépendra la qualité d'un système temps réel.

Mode noyau

Nous avons vu dans le chapitre précédent que Linux fait une grande distinction entre deux modes de fonctionnement : le mode **utilisateur** (restreint) et le mode **noyau** (privilégié) – aussi appelé « mode **superviseur** ». Dans le premier mode, les possibilités (jeu d'instructions assembleur, accès à la mémoire, entrées-sorties) sont restreintes, tandis que dans le second mode, aucune limite n'est imposée au logiciel. Il faut bien comprendre que cette séparation entre mode utilisateur et mode superviseur a une véritable représentation matérielle. Le processeur dispose d'un bit dans son registre d'état qui décrit le mode dans lequel il se trouve et c'est donc le processeur lui-même qui limitera ses possibilités lorsque le bit sera dans l'état « utilisateur » et ignorera les limitations si le bit est en l'état « superviseur ». Bien sûr, le détail de cette implémentation varie suivant les processeurs, mais le principe reste le même.

Sous Linux, toutes les applications s'exécutent dans le mode utilisateur (même celles qui ont les droits *root*) et seul le code du noyau s'exécute en mode superviseur.

Copyright © 2015 Eyrolles.

Le passage du mode superviseur au mode utilisateur s'effectue simplement en basculant le bit d'état du processeur. Comme celui-ci est autorisé à utiliser l'ensemble de son jeu d'instructions lorsqu'il se trouve en mode noyau, il lui est possible de modifier à volonté son registre d'état.

Lorsque le processeur se trouve en mode utilisateur, il peut revenir en mode superviseur dans trois cas.

- Le programme utilisateur a besoin de sous-traiter au noyau la réalisation de certaines tâches privilégiées (entrées-sorties, configuration de la mémoire virtuelle, etc.) : il réalise un **appel système**.
- Le processeur détecte une erreur de fonctionnement due au programme en cours d'exécution (mauvais accès mémoire, tentative d'utiliser une instruction assembleur interdite en mode utilisateur, opération arithmétique impossible, etc.). Il lève alors une **exception** qui va redonner instantanément le contrôle au noyau et lui permettre d'envoyer un signal (probablement fatal) au processus fautif.
- L'environnement extérieur (périphériques) doit indiquer au processeur l'occurrence d'une situation spécifique (données disponibles, erreurs, modification d'état d'un capteur, etc.) et déclenche une **interruption** qui va donner également le contrôle au noyau pour qu'il puisse répondre à l'événement survenu avant de reprendre son travail habituel.

Suivant les processeurs, le détail exact de fonctionnement varie, mais pour la plupart d'entre eux, ces trois cas seront traités de manière similaire en utilisant le principe général des interruptions.

Interruptions

Principe

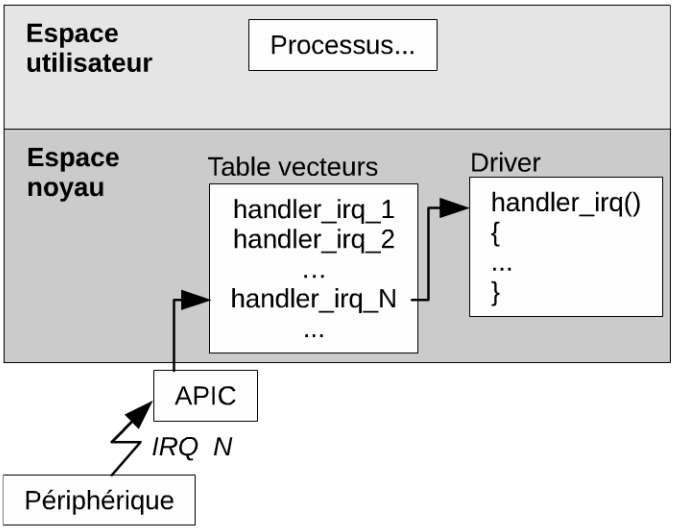
Supposons qu'un périphérique externe (un bouton, par exemple) ait besoin de signaler au système une modification de son état (un utilisateur vient d'appuyer sur le bouton). Il va envoyer un signal électrique à l'APIC *(Advanced Programmable Interrupt Controler)* du système. Ce composant – dont les fonctionnalités sont plus ou moins élaborées suivant la complexité de l'architecture matérielle – identifiera le changement d'état (par exemple, une transition montante de 0V à 5V ou, au contraire, un front descendant de 5V à 0V) et enverra à son tour une impulsion électrique sur une broche IRQ (*Interrupt Request* – demande d'interruption) du processeur.

Ce dernier va interrompre son travail en cours, mémoriser dans la pile l'état de ses registres et l'adresse de l'instruction en cours d'exécution, puis demander au contrôleur d'interruption ce qu'il se passe. L'APIC lui indiquera alors le numéro de l'IRQ survenue (numéro qui dépend du périphérique) et le processeur exécutera le code se trouvant à l'adresse mémoire correspondant à ce numéro. Pour cela, il consultera une table dite « table des vecteurs d'interruptions » qui contient l'adresse des routines à exécuter en fonction du numéro d'IRQ.

Copyright © 2015 Eyrolles.

Figure 2-1

Déclenchement d'une interruption



Une fois le traitement de l'interruption terminé, le processeur pourra reprendre le traitement qu'il était en train de réaliser auparavant.

Pour comprendre l'intérêt des interruptions, considérons un instant un système interactif qui n'en dispose pas.

Entrées-sorties sans interruptions

Les premiers systèmes interactifs étaient basés sur le principe de la « super boucle » *(super loop)*, c'est-à-dire que le programme passait son temps à examiner chacune de ses entrées l'une après l'autre et à réagir en conséquence. Cette scrutation *(polling)* s'exécutait en permanence, depuis la mise sous tension jusqu'à l'arrêt du système.

Un programme pour un tel système pouvait, par exemple, s'écrire comme suit – de manière très imagée :

```
int main (void)
{
    initialisation_du_systeme();

    while (1) { // Boucle infinie
        if (entree_capteur_1() != 0) {
            traitement_entree_1();
        }
        if (entree_capteur_2() != 0) {
            traitement_entree_2();
        }
        [...]
    }
    return 0; // Jamais atteint
}
```

Copyright © 2015 Eyrolles.

On imagine que les fonctions appelées dans cet exemple sont définies un peu plus haut dans le programme.

Ce principe que l'on rencontrait couramment dans les automates programmables industriels (*Programmable Logic Controler* – PLC), et qui est encore adopté parfois dans la programmation sur microcontrôleurs présente des avantages et des inconvénients.

En ce qui concerne les avantages, on notera la solidité du code. La modification d'une entrée déclenche automatiquement un traitement, de façon lisible et facilement extensible. Les états peuvent être prévus et répertoriés sous forme de graphes, et des outils et méthodologies d'analyse permettent d'envisager les transitions entre chaque situation d'entrée.

Par ailleurs, on distingue deux inconvénients principaux : le temps d'exécution et la consommation électrique. Le délai qui s'écoule entre la modification d'une entrée et la réponse par le système est difficilement prévisible. Cette durée est limitée au maximum par l'exécution de toute la boucle mais celle-ci est fonction du nombre de capteurs présents. L'ajout d'une nouvelle entrée peut modifier le temps de réponse à un événement détecté par un autre capteur. En d'autres termes, la prédictibilité du temps de réponse à un élément externe n'est pas assurée. Ce critère est pourtant, nous le verrons plus loin, essentiel pour les applications temps réel.

Le second problème avec ce type de programmation est la boucle incessante que parcourt le programme. Ne s'interrompant jamais, le processeur consomme de l'énergie en permanence et s'échauffe. Ceci nécessite un système de refroidissement (ailettes ou ventilateurs) qui occupe du volume et consomme éventuellement de l'électricité à son tour. Tout ceci s'avère fortement problématique dans le cas d'un système embarqué fonctionnant sur batteries. On notera en outre que le processeur – comme tout composant électrique – s'use et vieillit d'autant plus vite que sa température reste élevée.

Entrées-sorties avec interruptions

Si les modifications d'entrées du système peuvent être signalées par l'intermédiaire d'un contrôleur d'interruption qui déroute l'activité du processeur, le comportement est bien différent.

Le programme peut devenir, de manière tout aussi schématique que le précédent :

```
void handler_1 (void)
{
    acquitement_interruption_1();
    traitement_entree_1();
}


void handler_2 (void)
{
    acquitement_interruption_2();
    traitement_entree_2();
}
```

Copyright © 2015 Eyrolles.

```
int main(void)
{
    initialisation_du_système();

    installation_handler(INTERRUPT_1, handler_1);
    installation_handler(INTERRUPT_2, handler_2);

    while(1) {
        sommeil();
    }
}
```

Cette fois, dès que les gestionnaires d'interruption ont été installés (c'est-à-dire inscrits dans la table que consulte le processeur pour savoir quel traitement réaliser lorsque survient une IRQ), le programme se compose d'une simple boucle infinie autour d'une fonction que j'ai nommée `sommeil()` qui arrête le processeur jusqu'à l'arrivée d'une nouvelle interruption.

Suivant les processeurs, il existe différents types de sommeils, pour les plus courants d'entre eux, on retrouve des états typiquement nommés C0, C1, C2 et C3.

Nom	Signification
C0	Le processeur est actif, il exécute du code et consomme de l'énergie.
C1	L'horloge interne du processeur est arrêtée, ce dernier n'exécute plus d'instructions. Il consomme donc moins d'énergie et peut se réveiller très vite.
C2	Les horloges interne et externe (celles du bus) sont arrêtées, le redémarrage sera un peu plus long qu'en mode C1.
C3	Les horloges sont arrêtées et le processeur vide éventuellement ses caches (mémoire statique). Il consomme très peu d'énergie mais son redémarrage sera plus long.

Lorsque survient une interruption, le processeur est donc réveillé et se branche sur le *handler* associé, qui pourra acquitter l'interruption (c'est-à-dire la valider dans le contrôleur APIC) et procéder au traitement en conséquence.

Dans ce schéma, nous voyons que la consommation énergétique est moindre, le processeur n'étant actif que s'il doit réellement réaliser un travail. Ceci est particulièrement important pour les systèmes embarqués. De plus, le temps s'écoulant entre le déclenchement d'une interruption et le traitement correspondant ne dépend que de la vitesse du processeur et pas du nombre d'entrées qu'il est capable de gérer. Nous aurons de meilleures performances en ce qui concerne l'aspect temps réel.

Interruptions sous Linux

Linux utilise beaucoup les interruptions pour gérer les commutations entre mode utilisateur et mode noyau. Bien entendu, il s'en sert également pour communiquer avec le monde extérieur et recevoir des notifications à chaque changement d'état d'un périphérique.

Copyright © 2015 Eyrolles.

Le fichier `/proc/interrupts` est très intéressant car il présente la liste des interruptions gérées par le noyau, ainsi que le nombre de déclenchements de chacune d'elles, répartis sur chaque CPU ou cœur, depuis le démarrage du système. En voici un extrait :

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:        233       2750        211         52 IO-APIC-edge      timer
  1:          1          0          1          0 IO-APIC-edge      i8042
  8:          0          0          0          1 IO-APIC-edge      rtc0
  9:          0          0          0          0 IO-APIC-fasteoi   acpi
 12:          0          2          1          1 IO-APIC-edge      i8042
 16:     600778      29099        129        130 IO-APIC-fasteoi   uhci_hcd:u
 17:          0          0          0          0 IO-APIC-fasteoi   uhci_hcd:u
 18:       8912       5325          2          0 IO-APIC-fasteoi   firewire_o
 20:    2026994    2200738       1043       1033 IO-APIC-fasteoi   ata_piix,
 22:     175070      11321          6          4 IO-APIC-fasteoi   ehci_hcd:u
 23:   43770059    2302203        163        175 IO-APIC-fasteoi   ehci_hcd:u
 43:    4705950         25         23         25 PCI-MSI-edge      em1
 44:         78         79      30868      14149 PCI-MSI-edge      hda_intel
[...]
$
```

Pour chaque interruption (dont le numéro figure en première colonne), on trouve le nombre d'occurrences sur chaque CPU, puis le type d'interruption (ici, on remarque le contrôleur APIC des entrées-sorties et le contrôle des interruptions du bus PCI). Viennent ensuite la liste des drivers chargés de gérer chaque interruption. Il est possible que le même numéro d'interruption soit utilisé par plusieurs périphériques différents. Dans ce cas, les handlers des drivers seront invoqués successivement, chacun devant déterminer – en interrogeant son matériel – si l'interruption le concerne ou non.

Notons qu'il s'agit ici des interruptions pour lesquelles un gestionnaire a été installé (par un driver, par exemple), mais que l'APIC peut être capable de gérer bien d'autres lignes d'IRQ non mentionnées ici.

Rappelons que tout le contenu de `/proc` est entièrement virtuel, le noyau nous présente ainsi sous forme de pseudo fichiers des informations internes. Si l'on examine en boucle le contenu de `/proc/interrupts`, on verra les valeurs évoluer dynamiquement au gré des interruptions qui surviennent.

Routage des interruptions

Dans le cas d'un système multiprocesseur ou multicœur, une demande d'interruption (en provenance d'un périphérique externe) transmise par l'APIC sera traitée par l'un des processeurs ou cœurs. La répartition des interruptions sur les différents processeurs ou cœurs peut se faire à différents niveaux.

Certaines interruptions sont physiquement associées à un processeur particulier et ne pourront pas être redirigées.

Copyright © 2015 Eyrolles.

Le kernel Linux lui-même peut offrir une répartition des interruptions pour essayer d'équilibrer la charge de travail entre les différents processeurs. Cette solution n'est plus trop utilisée pour les systèmes récents car l'arbitrage effectué par le noyau était trop simpliste.

Il existe un démon – un processus utilisateur – nommé `irqbalance`, qui peut effectuer la répartition en optimisant la charge des différents processeurs en fonction de l'architecture sous-jacente. Par exemple, sur un système de type Quad-Core Intel, les cœurs 1 et 2 d'une part, et 3 et 4 d'autre part, partagent leurs caches ainsi que l'accès à la mémoire. Il est donc plus judicieux de traiter des interruptions simultanées sur les cœurs 1 et 3.

Le démon `irqbalance` est présent sur la majorité des distributions Linux actuelles. Toutefois, dans le cas de systèmes embarqués particuliers, il peut être intéressant de diriger spécifiquement certaines interruptions sur des processeurs donnés. On peut ainsi imaginer de conserver un cœur qui ne fait que du traitement de type calcul sans être interrompu, un cœur qui traite uniquement certaines interruptions bien précises pour lesquelles un temps de réponse garanti est nécessaire, et enfin deux cœurs réalisant le reste du travail et répondant aux interruptions classiques (réseau, disque, etc.) pour lesquelles aucune contrainte absolue n'est imposée.

Voyons pour cela le contenu du répertoire `/proc/irq/`.

```
# cd /proc/irq/
# ls
0  1  10  11  12  13  14  15  16  17  18  2  20  22  23  3  4  43  44  5  6  7  8  9
default_smp_affinity
#
```

Il existe donc un sous-répertoire par interruption que l'APIC est susceptible de gérer. Le contenu exact du sous-répertoire peut varier suivant les versions du noyau Linux. Dans les noyaux assez récents, on trouve entre autres un fichier `smp_affinity` qui contient le masque binaire des processeurs vers lesquels l'interruption peut être dirigée.

```
# cat 23/smp_affinity
03
#
```

Le masque hexadécimal étant `0x03`, soit `00000011` en binaire, l'interruption 23 (correspondant au clavier USB sur mon poste de travail) pourra être envoyée sur les deux premiers processeurs. Modifions cette valeur pour n'autoriser le traitement que par les troisième et quatrième processeurs en écrivant `00001100`, soit `0x0C` avec :

```
# echo C > 23/smp_affinity
# cat 23/smp_affinity
0C
#
```

Copyright © 2015 Eyrolles.

À partir de cet instant, l'interruption 23 ne sera traitée que sur les processeurs 3 et 4, ceci jusqu'au redémarrage du système, sauf si le démon `irqbalance` se déclenche et vient modifier l'affinité de cette interruption.

Certaines interruptions ne peuvent pas être redirigées, c'est le cas de l'interruption 0 sur mon poste :

```
# cat 0/smp_affinity
ff
# echo 1 > 0/smp_affinity
-bash: echo: erreur d'écriture : Erreur d'entrée/sortie
#
```

Toutefois, pour l'essentiel des lignes d'IRQ, le réglage de l'affinité est possible. Aussi, nous pourrons choisir précisément quels cœurs traiteront les interruptions et quels cœurs seront dédiés à des tâches de traitement ou de calcul.

Exceptions

Principe

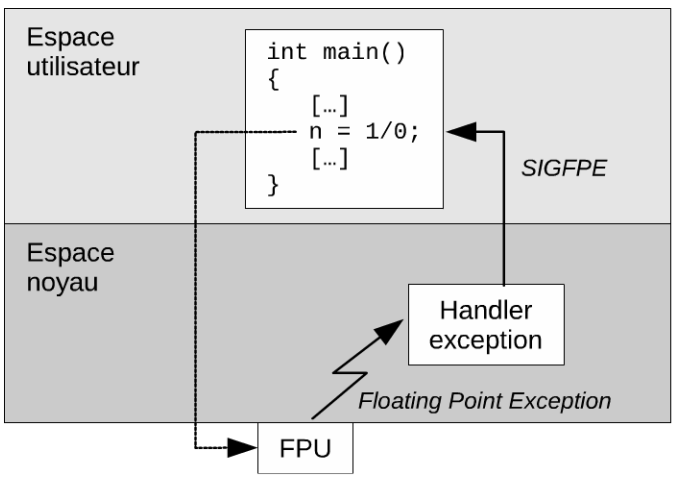
Un second mécanisme d'interaction du processeur avec son environnement existe : il s'agit des exceptions. Très semblables aux interruptions, les exceptions – appelées parfois « interruptions synchrones » – surgissent lorsque certaines conditions d'erreur sont rencontrées à cause du code en cours d'exécution.

Les exceptions sont générées par le processeur lui-même, sans l'intervention de périphérique externe.

Imaginons par exemple qu'un processus, durant une phase de calcul en espace utilisateur, effectue une tentative de division par zéro.

Figure 2-2

Déclenchement d'une exception



Copyright © 2015 Eyrolles.

La FPU *(Floating Point Unit)*, partie du processeur chargée de réaliser les calculs, va détecter une erreur irréparable et lèvera une exception *(Floating Point Exception)*. Le processeur procédera, comme lors de la réception d'une interruption, en sauvegardant ses opérations en cours et en se branchant vers un gestionnaire précédemment installé. Le gestionnaire se trouve naturellement dans le noyau, et a pour mission d'analyser l'opération invalide et de réagir en conséquence. La plupart du temps, il enverra au processus fautif un signal logiciel nommé `SIGFPE` *(Signal Floating Point Error)*. Si le signal n'est pas capturé par le processus, ce dernier sera tué instantanément.

Il existe principalement deux autres situations où une erreur de traitement déclenche une exception : l'instruction interdite et le pointeur invalide.

Imaginons, par exemple, que le processeur exécute un programme en mode utilisateur. Nous avons vu que le registre d'état du processeur indique qu'il fonctionne en mode restreint. Supposons à présent, que le processeur rencontre un code assembleur représentant une opération réservée au mode privilégié, ou un code ne correspondant à aucune opération. Ceci est typique d'un programme dont la pile a été corrompue et qui tente d'exécuter du code à une adresse mémoire contenant des données. La CPU *(Central Processing Unit)* du processeur lèvera une exception qui, traitée par un gestionnaire adapté, provoquera l'envoi du signal `SIGILL` *(Signal Illegal Instruction)* sur le processus fautif.

Enfin, le cas le plus fréquent est celui où le processus accède à une adresse de sa mémoire virtuelle à laquelle aucune plage d'adresses physiques n'a été rattachée. La MMU *(Memory Managment Unit)* lèvera l'exception PFE *(Page Fault Exception)* à son tour, entraînant l'exécution d'un handler qui enverra le signal `SIGSEGV` *(Signal Segmentation Violation)* vers le processus.

Notons toutefois que l'émission de ce signal ne se produit que si le processus a effectivement commis une erreur (utilisation d'un pointeur non initialisé, par exemple). Mais l'exception PFE est déclenchée très fréquemment, car c'est un élément important de la gestion de la mémoire virtuelle. C'est sur l'arrivée de cette exception, par exemple, que l'on effectue la véritable allocation dynamique de mémoire ou le chargement du fichier exécutable d'un programme.

Fichier core

Lorsqu'une exception entraîne l'émission d'un signal fatal (comme `SIGSEGV`, `SIGILL` ou `SIGFPE`) qui tue un processus, le noyau peut enregistrer dans un fichier une copie intégrale de la mémoire virtuelle du processus afin de permettre une analyse post-mortem à l'aide d'un débogueur. Ces fichiers sont traditionnellement nommés *core*, du nom des petits anneaux employés dans les mémoires magnétiques encore en utilisation lorsque les premiers systèmes Unix sont apparus dans les années 1970.

Le support qu'offre un fichier `core` pour assister le développeur lors des phases de débogage est très précieux. En effet, c'est une image exacte du processus qui est enregistrée à l'instant même où le matériel signale une erreur incorrigible. Dans le cas d'un accès mémoire illégal, par exemple, le fichier `core` nous indiquera la ligne du code source où l'erreur s'est produite, et nous permettra d'inspecter les pointeurs et variables pour déceler l'accès erroné. Voyons un exemple de programme qui tente d'écrire dans un pointeur `NULL` :

Copyright © 2015 Eyrolles.

```
exemple-crash.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    char * pointeur = NULL;

    fprintf(stderr, "Avant ecriture dans le pointeur\n");
    pointeur [0] = 'X';
    fprintf(stderr, "Apres ecriture dans le pointeur\n");
    return EXIT_SUCCESS;
}
```

Bien entendu, nous ne voyons que le premier message lors de l'exécution car le programme reçoit un signal `SIGSEGV` dès sa tentative de déréférencement du pointeur.

```
$ ./exemple-crash
Avant ecriture dans le pointeur
Erreur de segmentation
$
```

Le message « Erreur de segmentation » provient du shell, qui lit le code de terminaison de son fils et voit qu'il a été tué par le signal `SIGSEGV`. Il nous avertit donc de cette condition d'erreur. Toutefois, aucun fichier n'a été généré.

En fait, les fichiers `core` peuvent être très encombrants (plusieurs dizaines voire centaines de mégaoctets parfois). D'autant que sur certaines distributions Linux, le fichier a pour nom « core » suivi d'un point et du PID du processus qui a été tué (ceci est configurable *via* `/proc/sys/kernel/core_uses_pid`). Aussi, il peut y en avoir plusieurs dans le même répertoire. En outre, un fichier `core` n'est jamais utile à l'utilisateur (qu'il a plutôt tendance à exaspérer), ni à l'administrateur système. Seul le développeur peut lui trouver une utilité, mais seulement dans des périodes de débogage bien précises.

Aussi, sur la plupart des systèmes Linux, les fichiers `core` ne sont pas créés par défaut, et il faudra les demander explicitement. La commande `ulimit` du shell permet de fixer des limitations de ressources pour les commandes lancées à partir de ce shell. Entre autres, on peut restreindre l'usage de la mémoire, du temps CPU, du nombre de fichiers ouverts simultanément, etc. Avec l'option `-c` de cette commande, c'est la taille maximale des fichiers `core` qui est configurée. Or, celle-ci est généralement fixée à zéro à l'initialisation du shell. Essayons de la modifier et de relancer la commande précédente :

Copyright © 2015 Eyrolles.

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 20
file size               (blocks, -f) unlimited
pending signals                 (-i) 16382
[...]
max user processes              (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
$ ulimit -c unlimited
$ ./exemple-crash
Avant ecriture dans le pointeur
Erreur de segmentation (core dumped)
$ ls -l core
-rw------- 1 cpb cpb 221184 2011-09-13 23:14 core
$
```

Cette fois, nous avons obtenu un fichier `core`. Celui-ci peut être examiné à l'aide du débogueur GDB (ou d'un environnement de développement servant de frontal graphique à GDB comme Eclipse, NetBeans, Code::Blocks, etc.)

```
$ gdb ./exemple-crash core
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
[...]
Core was generated by `./exemple-crash'.
Program terminated with signal 11, Segmentation fault.
#0  0x08048451 in main () at exemple-crash.c:10
10          pointeur [0] = 'X';
(gdb) print pointeur
$1 = 0x0
(gdb) quit
$
```

Nous voyons que GDB nous indique précisément la ligne de code source où s'est produit l'accès ayant levé l'exception de faute de page. De plus, nous pouvons examiner le contenu de variable comme `pointeur`. La présentation de GDB pour afficher le contenu des variables est *a priori* surprenante : `$1` indique qu'il s'agit de la première évaluation d'expression, et `0x0` correspond à un pointeur – donc présenté en hexadécimal – de valeur nulle.

> Si vous ne réussissez pas à produire de fichier `core` même après configuration de `ulimit`, assurez-vous que le fichier `/proc/sys/kernel/core_pattern` ne contienne que le mot `core`.

Le principe des exceptions est donc un moyen simple et efficace pour rendre la main au noyau lorsqu'une situation anormale est détectée mettant en cause le code de l'espace utilisateur. On

Copyright © 2015 Eyrolles.

l'utilise également pour organiser le support de la mémoire virtuelle des processus, nous en reparlerons plus loin.

Appels système

Lorsqu'un processus désire réaliser une opération réclamant des accès aux ressources matérielles du système (par exemple, écrire dans un fichier ou lire un octet depuis un port série), il doit sous-traiter le travail au noyau qui vérifiera les autorisations dont dispose le processus et effectuera – ou non – l'action demandée.

Pour cela, le kernel Linux propose ses services par l'intermédiaire de fonctions spécifiques appelées « appels système ».

Dans l'espace utilisateur, l'appel système est implémenté dans la bibliothèque C, ce qui explique que cette dernière soit indispensable sur tout système Linux même si les applications ne sont pas écrites en C. Lors de la construction d'un système embarqué, l'installation de la bibliothèque GlibC (ou uClibC) vient immédiatement après la génération du noyau. Par exemple, l'appel système `write()`, qui sert à écrire dans n'importe quel descripteur de fichier, est implémenté directement dans la libC et s'occupera d'invoquer le noyau.

Au sein de ce dernier, l'appel système est une routine nommée `sys_write()` qui se trouve dans le fichier `fs/read_write.c` des sources du noyau.

> Dans les noyaux récents, le nom `sys_write` n'apparaît plus directement dans l'en-tête de la fonction, il est construit par la macro `SYSCALL_DEFINE3()`.

Pour passer de la fonction `write()` de la bibliothèque C – exécutée par du code en mode utilisateur – à la fonction `sys_write()` implémentée dans le noyau, on utilise sur la plupart des systèmes des interruptions logicielles.

Une interruption logicielle est une instruction assembleur que le programme exécute normalement depuis l'espace utilisateur. Toutefois, lorsque l'instruction est décodée par le processeur, tout se passe comme si une interruption matérielle était survenue : le processeur sauvegarde ses registres, passe en mode superviseur et se branche à l'adresse indiquée dans la table des vecteurs d'interruption. Cette routine se trouve naturellement dans l'espace kernel et va réaliser le travail demandé, après avoir vérifié les permissions du processus appelant. Une fois l'opération terminée, au retour du gestionnaire d'interruption, le contrôle repasse au processus en mode utilisateur et l'exécution reprend juste après l'instruction de l'interruption logicielle.

Suivi d'un appel système

Essayons de suivre la trace d'un appel système depuis l'espace processus jusqu'au sein du noyau Linux. Pour cela, prenons un petit programme qui affiche un message sur sa sortie standard (sans utiliser les flux de la bibliothèque C, mais seulement les descripteurs de fichiers).

Copyright © 2015 Eyrolles.

```
exemple-appel-systeme.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
    char * message = "Hello\n";
    write (STDOUT_FILENO, message, strlen(message));
    return EXIT_SUCCESS;
}
```

Nous allons centrer notre suivi sur l'appel système `write()`. Il est important de compiler le programme avec l'option `-static` de `gcc`, ce qui peut nécessiter, suivant les distributions, l'installation d'un package supplémentaire *libc-static*. Ainsi, nous sommes certains que l'implémentation de `write()` côté utilisateur est entièrement intégrée dans le fichier exécutable et nous pourrons l'examiner avec `objdump`.

```
$ gcc -o exemple-appel-systeme -Wall -g -static exemple-appel-systeme.c
$ objdump -D exemple-appel-systeme | less
```

Le code assembleur produit dépend des versions de la bibliothèque C employée et peut donc varier légèrement. Il s'agit ici d'un exemple réalisé sur un PC embarqué 32 bits.

Nous allons rechercher successivement plusieurs chaînes de caractères dans la sortie de `objdump`. Commençons par `main` :

```
080482c0 <main>:
 80482c0: 55                   push %ebp
 80482c1: 89 e5                mov  %esp,%ebp
[...]
 80482d8: e8 b3 6f 00 00       call 804f290 <strlen>
 80482dd: 89 44 24 08          mov  %eax,0x8(%esp)
 80482e1: 8b 44 24 1c          mov  0x1c(%esp),%eax
 80482e5: 89 44 24 04          mov  %eax,0x4(%esp)
 80482e9: c7 04 24 01 00 00 00 movl $0x1,(%esp)
 80482f0: e8 bb ad 00 00       call 80530b0 <_libc_write>
 80482f5: b8 00 00 00 00       mov  $0x0,%eax
 80482fa: c9                   leave
 80482fb: c3                   ret
```

On remarque l'appel à la fonction `libc_write`, précédé de quelques instructions qui placent successivement dans la pile du procesus : le retour de la fonction `strlen` (à l'adresse `80482dd`), le pointeur sur la chaîne de caractères (en `80482e5`) et le numéro du descripteur `STDOUT` de valeur `1` (en `80482e9`), tout ceci représentant les arguments de la fonction `write()`.

Copyright © 2015 Eyrolles.

Comme nous avons compilé notre processus de manière statique, nous retrouvons l'implémentation de `libc_write` un peu plus loin :

```
080530b0 <_libc_write>:
[...]
 80530bb: 8b 54 24 10       mov  0x10(%esp),%edx
 80530bf: 8b 4c 24 0c       mov  0xc(%esp),%ecx
 80530c3: 8b 5c 24 08       mov  0x8(%esp),%ebx
 80530c7: b8 04 00 00 00    mov  $0x4,%eax
 80530cc: ff 15 98 f0 0c 08 call *0x80cf098
 80530d2: 5b                pop  %ebx
 80530d3: 3d 01 f0 ff ff    cmp  $0xfffff001,%eax
 80530d8: 0f 83 02 20 00 00 jae 80550e0 <_syscall_error>
 80530de: c3                ret
```

Après avoir rempli les registres `ebx`, `ecx` et `edx` avec les valeurs extraites de la pile, et le registre `eax` avec la valeur 4 (représentant l'appel système `write()`), le programme va se brancher sur une routine dont l'emplacement est inscrit à l'adresse `0x080cf098`.

```
080cf098 <_dl_sysinfo>:
 80cf098: d0 45 05          rolb   0x5(%ebp)
 80cf09b: 08 00             or     %al,(%eax)
```

Ici, ce ne sont pas les instructions assembleur qui nous intéressent mais bien les valeurs stockées à cet emplacement : `d0`, `45`, `05` et `08`, qui correspondent à une adresse (stockée poids faible en premier sur architecture PC) : `0x080545d0`.

```
080545d0 <_dl_sysinfo_int80>:
 80545d0: cd 80             int $0x80
 80545d2: c3                ret
```

À cette adresse, nous trouvons bien l'instruction `int` qui déclenche une interruption logicielle. Il s'agit de l'interruption `0x80` – sur un PC – qui est utilisée pour entrer dans le noyau quel que soit l'appel système invoqué. La valeur stockée précédemment dans `eax`, qui correspond au numéro de l'appel système, servira, une fois dans le noyau, à sélectionner la routine à appeler.

Le lecteur intéressé pourra suivre la suite de l'appel système en consultant les fichiers suivants du kernel Linux (je considère ici le code source du noyau 3.0) :

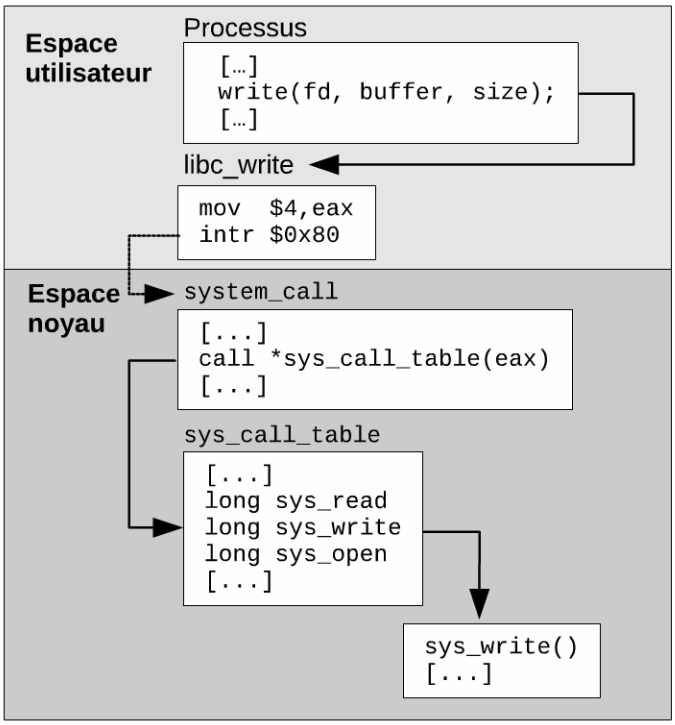
- au *boot*, la fonction `trap_init()` du fichier `arch/x86/kernel/traps.c` installe la routine `system_call()` comme gestionnaire de l'interruption `0x80` (appelée `SYSCALL_VECTOR`) ;
- la routine `system_call` implémentée dans le fichier assembleur `arch/x86/kernel/entry_32.S` appelle la fonction se trouvant au rang correspondant à la valeur `eax` dans la table `sys_call_table` ;
- l'entrée 4 (valeur chargée dans `eax` avant l'interruption) de la table `arch/x86/kernel/syscall_table_32.S` est un pointeur vers la fonction `sys_write()` mentionnée plus haut.

Ainsi, le passage du mode utilisateur au mode noyau par interruption logicielle peut être résumé par la figure 2-3.

Copyright © 2015 Eyrolles.

Figure 2-3

Suivi de l'appel système write()



À noter que sur certaines architectures modernes, des instructions assembleur spécifiques (SYSENTER/SYSEXIT) permettent de s'affranchir du mécanisme d'interruption logicielle en effectuant un branchement direct sur un gestionnaire dans le noyau de manière plus rapide et efficace ; néanmoins le principe reste globalement le même.

Threads du noyau

Une fois le *boot* achevé, le noyau lance le premier processus, *init*, dont le PID vaut 1, pour continuer l'initialisation. Celui-ci lance les premiers services, les démons, les connexions utilisateur, etc.

Ensuite, le contrôle ne reviendra au noyau que sur les événements suivants.

- Un périphérique externe doit signaler quelque chose au système (expiration d'un timer, disponibilité de données reçues, échec d'envoi, modification de l'état d'un capteur, etc.) et déclenche une interruption.
- Un processus utilisateur a commis une erreur nécessitant l'intervention du noyau, le matériel signale le problème par l'intermédiaire d'une exception.
- Une application de l'espace utilisateur doit sous-traiter une opération au mode superviseur et effectue donc un appel système qui se traduit habituellement par une interruption logicielle.

Copyright © 2015 Eyrolles.

Dans tous ces cas, c'est le mécanisme des interruptions (ou assimilé) qui est utilisé. Il s'agit du seul point d'entrée dans le noyau et nous lui accorderons une attention particulière lorsque nous aborderons les ordonnancements temps réel.

À noter qu'il existe du code noyau exécuté indépendamment de ces interruptions : les *kernel threads.*

Les threads du noyau sont des fonctions, exécutées avec les privilèges kernel (mode superviseur) dans l'espace mémoire du noyau, mais ordonnancées au même titre que les tâches du mode utilisateur (processus, threads utilisateur).

Les threads du noyau peuvent être affichés à l'aide de la commande `ps aux`, ils sont indiqués entre crochets :

```
$ ps aux
USER  PID %CPU %MEM  VSZ   RSS TTY STAT START TIME COMMAND
root    1  0.0  0.0  3040  1752 ?   Ss Aug30  0:03 /sbin/init
root    2  0.0  0.0     0     0 ?   S  Aug30  0:00 [kthreadd]
root    3  0.0  0.0     0     0 ?   S  Aug30  1:59 [ksoftirqd/0]
root    6  0.0  0.0     0     0 ?   S  Aug30  0:00 [migration/0]
root    7  0.0  0.0     0     0 ?   S  Aug30  0:00 [migration/1]
root    9  0.0  0.0     0     0 ?   S  Aug30  1:08 [ksoftirqd/1]
root   11  0.0  0.0     0     0 ?   S  Aug30  0:00 [migration/2]
root   13  0.0  0.0     0     0 ?   S  Aug30  0:10 [ksoftirqd/2]
root   14  0.0  0.0     0     0 ?   S  Aug30  0:00 [migration/3]
root   16  0.0  0.0     0     0 ?   S  Aug30  0:11 [ksoftirqd/3]
root   17  0.0  0.0     0     0 ?   S< Aug30  0:00 [cpuset]
root   18  0.0  0.0     0     0 ?   S< Aug30  0:00 [khelper]
root   19  0.0  0.0     0     0 ?   S< Aug30  0:00 [netns]
root   21  0.0  0.0     0     0 ?   S  Aug30  0:01 [sync_supers]
[...]
```

On peut remarquer que les valeurs des colonnes `VSZ` (*Virtual Memory Size*, taille de la mémoire virtuelle) et `RSS` (*Resident Set Size*, portion de la mémoire virtuelle présente en mémoire physique) sont à `0` pour tous ces threads. En effet, ils s'exécutent directement dans la mémoire kernel et ne disposent pas d'espace mémoire propre.

Comme nous le verrons plus loin, il est possible sur les noyaux récents de faire traiter les handlers des interruptions matérielles par des threads du noyau. Ces derniers étant ordonnancés, ils disposent d'une priorité d'exécution configurable que nous pourrons régler en fonction des autres tâches temps réel du système.

La seule différence d'exécution notable entre un kernel thread et une tâche de l'espace utilisateur, est que certains kernel threads peuvent être attachés à des processeurs – ou cœurs – bien précis (par exemple, `ksoftirqd/0`, `ksoftirqd/1`, `migration/1`, `migration/3`, etc., dans l'exemple précédent) et ne pourront pas être déplacés à notre guise.

Copyright © 2015 Eyrolles.

Conclusion

Dans ce chapitre, nous avons observé les points d'entrées du noyau Linux. Plus particulièrement, nous avons remarqué l'importance des interruptions (matérielles ou logicielles) pour l'activation de l'espace kernel.

Dans le chapitre suivant, nous étudierons les méthodes utilisées par le scheduler pour déterminer quelle tâche activer.

Points clés

Pour passer de l'espace utilisateur à l'espace noyau, trois possibilités : l'interruption matérielle – provenant de l'extérieur du système –, l'exception signalant une erreur d'exécution et l'appel système qui permet à un processus utilisateur de demander au noyau de réaliser une opération privilégiée pour son compte.

Linux peut lancer l'exécution de tâches (les kernel threads) fonctionnant avec les privilèges et l'espace mémoire du noyau, tout en étant ordonnancées comme les autres tâches du système.

Il est possible de fixer l'affinité d'une interruption afin que le gestionnaire se déroule sur un processeur spécifique. Ceci permet d'isoler un traitement temps réel des fluctuations liées aux interruptions.

Exercices

Exercice 1 (*)

Observez les interruptions de votre système, en utilisant la commande :

```
# watch -n 0.1 cat /proc/interrupts
```

qui affiche le contenu du fichier `/proc/interrupts` tous les dixièmes de secondes (suivant la localisation de votre système, il vous faudra peut-être utiliser `0,1` plutôt que `0.1`).

Voyez-vous une interruption timer se produire régulièrement ? Avec quelle fréquence ?

Exercice 2 (*)

Déterminez par l'observation quelle interruption se produit lorsque vous déplacez la souris de votre système ou lorsque vous pressez une touche.

Exercice 3 (**)

Si votre système est multiprocesseur, modifiez l'affinité des interruptions identifiées précédemment et vérifiez si elles se déplacent bien sur le processeur choisi.

Copyright © 2015 Eyrolles.

Exercice 4 (**)

Écrivez un programme qui tente de réaliser une division par zéro. Exécutez-le, que se passe-t-il ?

Exercice 5 (***)

Comment pouvez-vous écrire un programme qui déclenche l'interruption `SIGILL` ?

Copyright © 2015 Eyrolles.

3

Ordonnancement temps partagé et priorités

Les tâches s'exécutant sur un système Linux se déroulent typiquement sous un ordonnancement temps partagé. Bien sûr, nous verrons par la suite comment les forcer à s'exécuter en temps réel, mais il est important de comprendre le fonctionnement de l'ordonnancement temps partagé pour ajuster au mieux les performances d'un système interactif réalisant des traitement en temps réel.

Temps partagé

Principes

Linux, en tant que descendant d'Unix, est un système multitâche et multi-utilisateur ordonnancé en temps partagé. L'article original de Dennis Ritchie et Ken Thomson, « The Unix Time-Sharing System », datant de 1974 décrit le premier système Unix et l'aspect temps partagé de ce système multitâche[1].

Le temps partagé, que l'on confond parfois avec le multitâche préemptif, est un mode d'ordonnancement essayant de répartir le plus équitablement possible le temps CPU disponible, afin que plusieurs tâches puissent s'exécuter de manière apparemment simultanée sur le même processeur. Pour cela, un composant du noyau, le scheduler (ordonnanceur), est chargé de commuter l'exécution du CPU entre les différentes tâches prêtes *(Runnable)* comme nous l'avons évoqué au chapitre 1.

Examinons les situations dans lesquelles l'ordonnanceur peut cesser l'exécution d'une tâche T1 pour activer une autre tâche. Il existe principalement trois cas de figure.

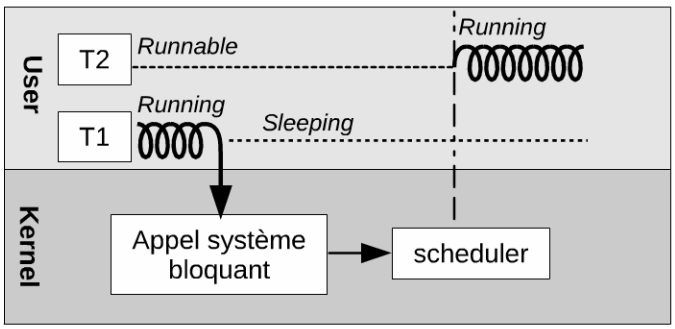
1. On trouve aisément des copies de cet article sur Internet en saisissant son titre dans un moteur de recherche. Il est intéressant de voir comment les concepts exposés sont toujours aussi présents dans Linux plus de 40 ans après sa publication.

Copyright © 2015 Eyrolles.

- La tâche T1 invoque un appel système durant lequel elle est mise en sommeil en attente d'un événement futur (réception de données, lecture d'un fichier, timer, etc.). Le noyau pourra alors activer une autre tâche en attente. Nous pouvons observer cette commutation sur la figure 3-1 où le temps s'écoule de gauche à droite. Initialement, la tâche T1 est *Running* (active) et la tâche T2 est *Runnable* (prête). Lorsque T1 invoque un appel système durant lequel elle s'endort, l'ordonnanceur va alors choisir d'activer T2 qui passe à l'état *Running*. Si aucune tâche n'est prête quand T1 s'endort, l'ordonnanceur peut invoquer une routine spécifique du noyau, que l'on nomme traditionnellement la boucle *idle* (inactive) au sein de laquelle il est possible sur certaines architectures d'arrêter le processeur jusqu'à la prochaine occurrence d'une interruption matérielle.

Figure 3-1

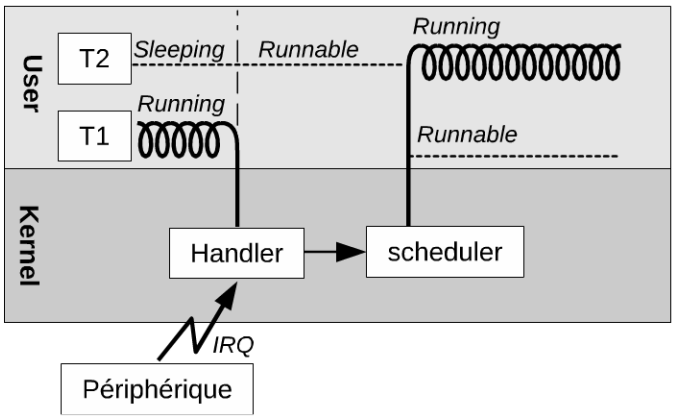
Commutation sur appel système bloquant



- Le deuxième type de commutation se présente quand une interruption survient, indiquant l'occurrence d'un événement (entrée-sortie, timer, etc.) qui réveille une autre tâche que le noyau juge plus prioritaire. Cette dernière est alors activée et T1 retourne l'état *Runnable*. Sur la figure 3-2, la tâche T2 est endormie (dans un appel système bloquant) en attente d'un événement externe (par exemple, l'arrivée d'un octet sur une ligne série). Quand le matériel déclenche une interruption, dont le handler indique que la condition attendue est réalisée, T2 est réveillée et placée dans l'état *Runnable*. Si l'ordonnanceur la juge plus prioritaire que T1, cette dernière est préemptée pour laisser passer T2.

Figure 3-2

Commutation sur interruption externe

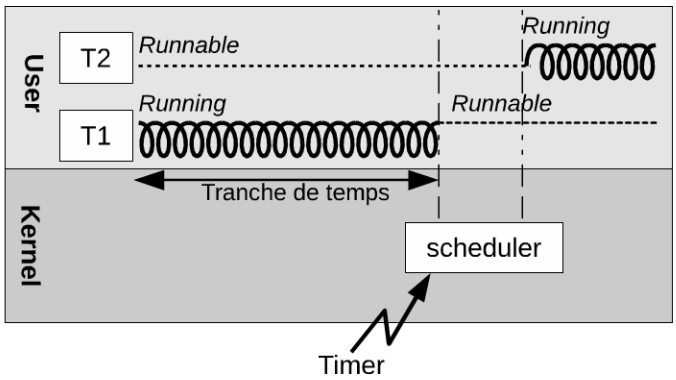


Copyright © 2015 Eyrolles.

- Enfin, le dernier type de commutation se présente ainsi : lorsque le noyau a activé T1, il lui a accordé une certaine tranche de temps *(timeslice)*, dont la longueur varie en fonction de la priorité de la tâche. Au bout de cette durée, si T1 n'a pas cédé le processeur volontairement, elle sera préemptée par le noyau. Si une autre tâche T2 est dans l'état *Runnable*, elle sera activée à son tour. Sinon, T1 sera réactivée. La figure 3-3 illustre cette situation : la tâche T1 est activée et conserve le processeur jusqu'à l'expiration d'un timer qui invoque l'ordonnanceur. Celui-ci juge qu'il est temps d'activer la tâche T2 et de laisser T1 dans l'état *Runnable* jusqu'à l'expiration de la nouvelle tranche de temps accordée à T2, où elle sera probablement réactivée.

Figure 3-3

Commutation sur expiration de timer



Ce dernier exemple de commutation est probablement le mieux connu, bien qu'il ne représente en réalité qu'un cas relativement rare. Il se produit lorsque plusieurs tâches sont actives simultanément pour consommer du temps processeur (jobs de calcul, par exemple). Sur la plupart des systèmes Linux courants, les commutations 1 et 2 sont les plus fréquentes puisque c'est sur l'occurrence d'événements externes *via* les interruptions que la majorité des commutations a lieu.

En résumé, une tâche placée sur le processeur pourra être interrompue soit volontairement en invoquant un appel système bloquant, soit involontairement lors de l'arrivée d'une interruption réveillant une tâche plus prioritaire ou lors de l'expiration d'un timer indiquant qu'elle a consommé toute la tranche de temps qui lui était allouée.

Dans tous les cas, l'ordonnanceur de Linux est mis à contribution pour déterminer quelle sera la prochaine tâche à activer. Dans un environnement à ordonnancement temps partagé, c'est de la qualité de cet ordonnanceur, de la pertinence de ses choix et de sa rapidité de décision que dépendront la souplesse et la fluidité du système pour l'utilisateur.

> Dans un environnement temps réel, il n'y a pas réellement de choix laissé à l'ordonnanceur, il se contente – comme nous le verrons par la suite – de toujours activer la tâche la plus prioritaire.

Sous Linux, plusieurs ordonnanceurs temps partagé ont été implémentés, avec des améliorations successives. Nous allons examiner quelques versions marquantes.

Copyright © 2015 Eyrolles.

Ordonnanceur historique

Depuis les premières versions de Linux, l'ordonnanceur est implémenté par la fonction `schedule()` dans le fichier `kernel/sched.c`.

> Pour étudier confortablement les sources de Linux, je conseille le site web suivant, qui permet de naviguer aisément entre les versions, les répertoires et les fichiers :
> ‣ http://lxr.free-electrons.com

Le premier ordonnanceur a été implémenté dès le noyau 0.0.1 en 1991. Il n'a pas fondamentalement changé avant la version 2.6 diffusée fin 2003 (plus exactement avant le noyau de développement 2.5.0, mais je préfère considérer les versions stables du kernel).

La fonction `schedule()` de cet ordonnanceur passe en revue toutes les tâches dans l'état *Runnable*, calculant pour chacune d'entre elles un poids indiquant à quel point il est intéressant d'activer cette tâche. À partir de la version 1.3.36 du kernel, une fonction spécifique nommée `goodness()` a été introduite pour calculer cette valeur.

Le calcul de la qualité d'une tâche s'effectue ainsi.

- Une tâche déjà activée sur un autre CPU reçoit un poids de -1 000 qui l'empêche d'être sélectionnée.
- Si une tâche est ordonnancée en temps réel, elle reçoit immédiatement un poids de 1 000 additionné à sa priorité. Ceci permet de sélectionner instantanément la tâche temps réel la plus prioritaire, sans qu'elle soit concurrencée par les tâches temps partagé (dont le poids ne peut dépasser 140 environ). Cette prise en compte des ordonnancements temps réel est apparue dans le noyau 1.4 (1.3.43 pour la branche de développement).
- Pour les tâches temps partagé, le poids est initialisé avec le nombre de *ticks* système non consommés dans sa tranche de temps. Nous reviendrons sur les ticks dans le prochain chapitre, la valeur maximale était environ de 80.
- Si la tâche s'exécutait précédemment sur le même CPU, son poids est augmenté d'une valeur arbitraire (15 sur un PC) pour prendre en compte le coût de la migration des processus entre CPU.
- Si la configuration de la mémoire virtuelle de la tâche est déjà chargée dans la MMU du processeur, le poids est incrémenté de 1 pour favoriser très légèrement les commutations entre threads du même processus.
- On ajoute au poids une valeur de priorité fournie par l'utilisateur, allant de 1 à 40, nous la décrirons plus loin.

Cet ordonnanceur fonctionnait bien, mais il posait un problème sur les serveurs comportant un grand nombre de tâches : la fonction `schedule()` devait à chaque invocation passer en revue toutes les tâches prêtes pour choisir celle à activer.

Le temps pris pour se décider était ainsi directement proportionnel au nombre de tâches dans l'état *Runnable*. Si l'on a *N* tâches prêtes, la fonction `schedule()` prendra un temps proportionnel

Copyright © 2015 Eyrolles.

à *N* pour se décider. En algorithmique, on dit que l'ordonnanceur est donc d'une complexité *O(N)*.

Par ailleurs, le comportement des premiers noyaux n'était pas aussi souple que l'on pouvait l'espérer et lors d'une grosse phase de calcul ou de compilation, le fonctionnement de l'interface utilisateur était sensiblement ralenti.

Ordonnanceurs du noyau 2.6

Avec l'arrivée du noyau 2.6, nous avons vu apparaître un nouvel ordonnanceur développé par Ingo Molnár en essayant de mettre l'accent sur trois points.

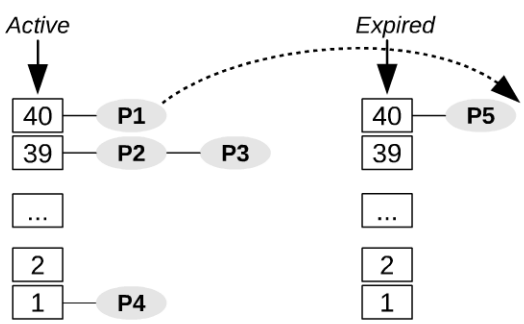
- Linux est largement utilisé sur des postes de travail personnels, au sein desquels ne s'exécutent qu'une à deux tâches simultanées. Inutile de perdre du temps à se décider si le choix ne se réduit qu'à une seule possibilité.
- Il existe également des serveurs utilisant Linux avec des centaines, voire des milliers de tâches actives en même temps. Nous avons indiqué que l'ordonnanceur précédent n'était pas satisfaisant car sa complexité était *O(N)*. La nouvelle fonction `schedule()` est en *O(1)*. Elle s'exécute avec un temps proportionnel à 1, autrement dit en temps constant, quel que soit le nombre de tâches prêtes.
- L'ordonnanceur précédent n'offrait pas une très bonne réactivité lorsque le système était chargé par des tâches de calcul. Le nouvel ordonnanceur doit favoriser les tâches interactives, quitte à pénaliser légèrement les jobs de compilation ou de calcul, afin de garder une souplesse satisfaisante même avec une charge importante.

L'implémentation de cet ordonnanceur s'appuie sur deux tables nommées `active` et `expired` (voir figure 3-4). Chaque table contient 40 listes de tâches, une pour chaque niveau de priorité temps partagé.

Les listes ne contiennent que des tâches dans l'état *Runnable*. Initialement, `expired` est vide et `active` contient toutes les tâches prêtes, classées en fonction de leurs priorités.

Figure 3-4

Premier ordonnanceur du noyau 2.6



Lorsque `schedule()` est appelée, elle n'aura qu'à prendre la première tâche prête en partant du haut de la table `active`. Ceci est implémenté avec un algorithme qui garantit que le temps d'exécution reste constant, quel que soit le nombre de tâches dans la table.

Copyright © 2015 Eyrolles.

La tâche choisie (P1 sur la figure 3-4) va être placée sur le processeur, et le noyau lui accordera une tranche de temps dont la durée variera en fonction de sa priorité (plus une tâche est prioritaire, plus longue est sa tranche de temps). Comme nous l'avons vu chapitre 2, trois cas mènent à l'éviction de P1.

- P1 s'endort dans un appel système bloquant : elle sort donc des listes de tâches actives et `schedule()` passe à la suivante.
- Une autre tâche plus prioritaire se réveille suite à une interruption : la tâche P1 va donc regagner la table `active`, le noyau notant qu'elle a déjà consommé une partie de sa tranche de temps.
- P1 consomme tout le temps qui lui était imparti et un timer indique au noyau qu'il faut la préempter : elle va donc passer dans la table `expired` et `schedule()` activera la suivante.

Peu à peu, `active` va se vider au profit de `expired`, à chaque fois qu'une tâche consomme toute sa tranche de temps. Une fois `active` entièrement vide, le noyau intervertit les deux tables et reprend sa progression.

Cet algorithme présente de nombreux avantages. Nous avons déjà évoqué le fait qu'il soit rapide à exécuter et que sa durée n'augmente pas en fonction du nombre de tâches prêtes. Un autre point important est l'absence totale de « famine » : même la tâche la moins prioritaire est sûre de pouvoir s'exécuter une fois (pendant une durée très courte, il est vrai) entre deux exécutions successives des tâches plus prioritaires (dont les tranches de temps sont nettement plus longues).

Lorsqu'un processus est placé dans la table `expired`, le noyau peut jouer sur sa position en fonction de son comportement précédent. Ainsi, une tâche fortement consommatrice de temps CPU (du calcul, par exemple) sera-t-elle « punie » modérément par l'ordonnanceur, tandis qu'une tâche passant beaucoup de temps endormie dans des appels système bloquants (attente de déplacement de la souris, de pression sur le clavier, etc.) sera considérée comme interactive et légèrement favorisée par rapport aux autres. De cette manière, dès qu'une interruption de déplacement de la souris se déclenchera, l'interface graphique sera activée et pourra réagir instantanément, au détriment des jobs de compilation, par exemple.

Il faut noter que la complexité de la fonction d'insertion dans une table est *O(log(N))* – seule l'extraction d'une tâche dans la table est en *O(1)* – ce qui reste tout à fait raisonnable, même pour un nombre conséquent de tâches parallèles.

Ordonnanceur CFS

Pour le noyau 2.6.23, Ingo Molnár a développé un nouvel ordonnanceur, dont les performances sont jugées meilleures que celles de son prédécesseur. Il s'agit du CFS (*Completely Fair Scheduler*, ordonnanceur totalement équitable), dont le but est de partager le plus équitablement possible le temps CPU disponible entre les tâches.

Le CFS ne gère plus les tâches à travers des listes d'attente classées par priorité. Il les trie en se basant sur la valeur de *virtual runtime* de chaque tâche, correspondant au temps CPU consommé. À chaque invocation de l'ordonnanceur, celui-ci active la tâche ayant le plus manqué de temps CPU (celle dont le *virtual runtime* est le plus court). Si cette tâche s'endort ou si elle est préemptée, on passe à la suivante ayant la valeur de *virtual runtime* la plus faible.

Copyright © 2015 Eyrolles.

Pour organiser les tâches en attente, l'ordonnanceur CFS s'appuie sur une structure de données nommées *red-black tree*. Il s'agit d'arbres binaires dont chaque nœud est coloré en rouge ou en noir. Quelques règles permettent d'ajouter, de supprimer des éléments dans l'arbre ou de le trier avec une complexité *O(log(N))*, ce qui signifie que pour un nombre *N* de tâches en attente, l'extraction ou l'insertion prendront au maximum une durée proportionnelle à *log(N)*. Ceci reste très raisonnable même lorsque *N* croît de manière importante.

Les avantages de l'ordonnanceur CFS sur l'ordonnanceur en *O(1)* précédent sont essentiellement liés aux priorités entre les tâches. Le CFS les prend en considération dans le calcul du temps attribué à la tâche avant préemption. Comparativement à son prédécesseur, le CFS offre les bénéfices suivants.

- L'influence de la priorité fixée par l'utilisateur sur la durée accordée à la tâche avant sa préemption est uniforme : augmenter d'une unité la priorité multiplie le temps disponible par 1,1.
- Les durées avant préemption varient nettement avec les priorités temps partagé, ce qui rend ces dernières beaucoup plus sensibles qu'avec l'ordonnanceur précédent.
- En conséquence, il est possible de faire fonctionner des applications imposant des contraintes temporelles (lecteur vidéo, par exemple) en temps partagé avec une priorité élevée, alors qu'auparavant il était indispensable de les exécuter selon un ordonnancement temps réel.

Une autre évolution de l'ordonnanceur Linux est d'être devenu relativement générique et d'appeler des routines qui dépendent de modules chargés dans le noyau. Ceci devrait permettre d'expérimenter plus facilement les évolutions du scheduler.

Groupes de processus

Les premières versions de l'ordonnanceur CFS assuraient une répartition équitable du temps CPU entre les tâches (en prenant bien sûr en considération les priorités temps partagé que nous décrirons plus loin). Toutefois, ceci pose un problème dans le cas d'un serveur multi-utilisateur ou même du poste de travail d'un développeur : si on lance un gros job de calcul ou de compilation avec de nombreuses tâches en parallèle (par exemple, un `make -j 64` dans la compilation du noyau Linux), cela va pénaliser le fonctionnement interactif du système ou des applications multimédias (lecteur vidéo, par exemple).

Des améliorations proposées par Srivatsa Vaddagiri ont permis de prendre en compte les groupes de processus. Cette notion de groupe (qui jusqu'alors ne servait principalement qu'au shell pour envoyer des signaux à des ensembles de processus lancés en arrière-plan) va permettre de répartir plus équitablement le temps CPU entre les différentes activités du système. Si cinq processus de calcul appartiennent à un groupe et deux processus à un autre groupe, ceux du premier groupe recevront 10 % du temps CPU et ceux du second 25 %. Ainsi, chaque groupe bénéficiera de 50 % du temps et la répartition se fera équitablement entre les processus du groupe.

Le lecteur intéressé pourra trouver de la documentation sur les groupes de processus dans le répertoire `Documentation/cgroups` des sources du noyau Linux. Ce mécanisme permet de contrôler les ressources (mémoire, temps processeur, CPU, etc.) qui sont affectées à chaque groupe de processus.

Copyright © 2015 Eyrolles.

La gestion des groupes de processus étant assez complexe à organiser pour l'administrateur, une option est apparue dans le noyau 2.6.38, implémentée par Mike Galbraith. Elle permet de regrouper automatiquement les processus en fonction de leur terminal de contrôle.

Si nous démarrons une compilation avec 64 tâches en parallèle dans un terminal, et qu'un lecteur vidéo est en cours de fonctionnement, ce dernier conservera ses 50 % du temps CPU, alors que les tâches de compilation ne recevront chacune que 50/64 = 0,78 %.

Ceci est tout à fait adapté pour le poste de travail d'un programmeur et l'amélioration a été accueillie par un message enthousiaste de Linus Torvalds en novembre 2010 :

[ ] Je dois dire que je suis (très agréablement) surpris par la taille réduite de ce patch, son aspect non intrusif et plutôt élégant. Je suis également très satisfait de ses performances interactives.

J'avoue que mon test est vraiment trivial (lire mes mails dans un navigateur web, me balader un peu sur le système, tout en exécutant un `make -j64` *sur le noyau), mais c'est ce qui me concerne vraiment. Et c'est une* ***énorme*** *amélioration. Une amélioration pour des choses comme la fluidité du déroulement, mais également pour le temps de chargement des pages web. Ça n'aurait sans doute pas dû me surprendre, mais j'ai toujours associé la lenteur de chargement avec les performances réseau. Or, il y a une demande de CPU assez importante pendant le chargement d'une page web pour que le processus se retrouve en manque de temps processeur s'il y a déjà une charge système de plus de 50.*

Donc je pense qu'il s'agit vraiment d'un patch apportant une « amélioration réelle ». Bon travail. L'ordonnancement par groupe passe de « utile pour certains serveurs spécifiques » à « fonctionnalité qui tue ».

Les distributions actuelles ont adoptées ce comportement comme étant celui par défaut sur les stations de travail2. Nous allons à présent vérifier un peu ces concepts avec quelques exemples. Nous allons nous appuyer sur le petit programme simple suivant : il boucle autour de l'appel système `time()` pour lire l'heure en permanence et compte le nombre de boucles qu'il réalise. Lorsque dix secondes sont écoulées, il sort de sa boucle et affiche son nombre d'itérations.

Nous l'exécuterons une fois tout seul pour déterminer le nombre de boucles possibles en une seconde fois sur notre processeur. Ensuite, lorsque nous le réinvoquerons dans d'autres circonstances, nous lui indiquerons ce nombre en argument sur sa ligne de commandes. Il pourra ainsi nous afficher le pourcentage du temps CPU dont il a bénéficié.

> On notera que les mesures sont ici très grossières, avec une précision de l'ordre de la seconde. On peut si besoin affiner les calculs en employant `gettimeofday()` qui fournit le complément horaire en microsecondes.

```
./exemple-taux-cpu.c
#include <stdio.h>
#include <stdlib.h>
```

2. L'option à activer lorsque l'on compile soi-même son kernel est Automatic process group scheduling du menu General setup.

Copyright © 2015 Eyrolles.

```
#include <unistd.h>
#include <time.h>
int main(int argc, char * argv[])
{
    time_t debut;
    long int nb_boucles;
    long int nb_boucles_max;

    debut = time(NULL);

    nb_boucles = 0;
    while (time(NULL) < (debut + 10)) {
        nb_boucles ++;
    }
    printf("[%d] nb_boucles : %ld ", getpid(), nb_boucles);
    if ((argc == 2)
     && (sscanf(argv[1],"%ld", & nb_boucles_max) == 1)){
        printf("(%.0f)",
               100.0 * nb_boucles / nb_boucles_max);
    }
    printf("\n");
    return EXIT_SUCCESS;
}
```

Comme nous voulons étudier les partages de temps CPU, il est important que tous les processus que nous lançons s'exécutent sur le même processeur (même cœur). Aussi, commençons-nous par fixer l'affinité du shell et celle de ses descendants :

```
$ taskset -pc 0 $$
pid 20545's current affinity list: 0-3
pid 20545's new affinity list: 0
$
```

Puis, nous exécutons à plusieurs reprises le programme tout seul pour avoir une estimation du nombre maximal de boucles en dix secondes :

```
$ ./exemple-taux-cpu
[4666] nb_boucles : 81483838
$ ./exemple-taux-cpu
[4673] nb_boucles : 76501287
$ ./exemple-taux-cpu
[4678] nb_boucles : 76312723
$ ./exemple-taux-cpu
[4681] nb_boucles : 81475481
$
```

Copyright © 2015 Eyrolles.

Sur ce système, on réalise donc environ 81 000 000 itérations par dizaines de secondes. Je répète que je ne recherche pas une valeur précise – sinon j'aurai appelé `gettimeofday()` – mais un ordre de grandeur de la répartition du CPU.

Lançons deux tâches, puis trois en parallèle, depuis le même terminal :

```
$ ./exemple-taux-cpu  81000000  &  ./exemple-taux-cpu  81000000
[4700] nb_boucles : 37309646 (46)
[4701] nb_boucles : 37370351 (46)
$ ./exemple-taux-cpu  81000000 & ./exemple-taux-cpu  81000000
    & ./exemple-taux-cpu  81000000
[4711] nb_boucles : 26403300 (33)
[4712] nb_boucles : 26461390 (33)
[4710] nb_boucles : 26410465 (33)
$
```

Les résultats ont été légèrement édités pour supprimer les messages parasites du shell lors du lancement de processus en parallèle et l'affichage de son *prompt* alors qu'un processus tourne encore à l'arrière-plan.

Nous voyons bien que la répartition du temps est de 50 % pour chaque processus lorsqu'il y en a deux, et de 33,3 % lorsqu'il y en a trois.

Vérifions la répartition entre deux terminaux. Dans un nouveau terminal, nous fixons également l'affinité du shell :

```
$ taskset -pc 0 $$
pid 527's current affinity list: 0-3
pid 527's new affinity list: 0
$
```

Puis, nous lançons dix processus dans un terminal :

```
$ ./exemple-taux-cpu 81000000 & ./exemple-taux-cpu 81000000
 & ./exemple-taux-cpu 81000000 & ./exemple-taux-cpu 81000000
 & ./exemple-taux-cpu 81000000 & ./exemple-taux-cpu 81000000
 & ./exemple-taux-cpu 81000000 & ./exemple-taux-cpu 81000000
 & ./exemple-taux-cpu 81000000 & ./exemple-taux-cpu 81000000
[4781] nb_boucles : 3922792 (5)
[4776] nb_boucles : 3946171 (5)
[4774] nb_boucles : 3937505 (5)
[4777] nb_boucles : 3934834 (5)
[4778] nb_boucles : 3945245 (5)
[4772] nb_boucles : 3900060 (5)
[4773] nb_boucles : 3932328 (5)
```

Copyright © 2015 Eyrolles.

```
[4779] nb_boucles : 3956489 (5)
[4775] nb_boucles : 3952508 (5)
[4780] nb_boucles : 3957889 (5)
$
```

Simultanément, nous avons lancé dans un autre terminal :

```
$ sleep 1 ; ./exemple-taux-cpu 81000000  &  ./exemple-taux-cpu  81000000
[4783] nb_boucles : 18805290 (23)
[4782] nb_boucles : 18837704 (23)
$
```

Le sommeil d'une seconde avant démarrage des processus laisse le temps de commuter sur l'autre terminal pour lancer simultanément les deux commandes.

Nous voyons bien que l'ordonnanceur a attribué 50 % du temps CPU à chaque terminal, ce qui se traduit par 5 % pour chacune des dix tâches du premier et 25 % pour les deux processus du second.

Autres ordonnanceurs

Il existe d'autres ordonnanceurs qui n'ont pas été intégrés dans la version standard de Linux : Con Kolivas en a notamment développés plusieurs. Son algorithme RSDL *(Rotating Staircase Deadline Scheduler)* a été supplanté au dernier moment par le CFS qui s'en inspirait, en octobre 2007, ce qui l'a poussé à abandonner le développement noyau pendant plusieurs mois. De retour en septembre 2009, il a présenté un nouvel ordonnanceur, spécialisé pour les postes de travail mono-utilisateur, nommé par provocation BFS *(Brain Fuck Scheduler)* montrant son animosité envers le CFS d'Ingo Molnár. Cet algorithme n'est *a priori* pas destiné à être inclus dans le noyau officiel, mais existe sous forme de patch pour modifier les sources du kernel standard.

Configuration des priorités

Courtoisie des processus

Comme nous l'avons évoqué précédemment, Linux permet à l'utilisateur de donner des priorités aux processus pour en favoriser certains ou en limiter d'autres.

> Attention à ne pas confondre ces priorités, qui concernent uniquement l'ordonnancement temps partagé avec les priorités temps réel que nous examinerons dans la seconde partie de ce livre.

Suivant la tradition des systèmes Unix, l'ordonnanceur Linux propose à l'utilisateur d'indiquer la courtoisie, la gentillesse *(nice)* du processus, c'est-à-dire une valeur qui représente la facilité

Copyright © 2015 Eyrolles.

avec laquelle le processus laisse passer les autres. Traditionnellement, cette valeur – qui progresse donc inversement à la priorité du processus – va de -20 pour un processus très agressif à +19 pour un processus très gentil. Généralement, les processus sont lancés par défaut avec la valeur 0 et il est nécessaire d'avoir les privilèges *root* pour pouvoir diminuer la gentillesse (augmenter la priorité) d'un processus.

On voit bien, à ce propos, qu'Unix s'est développé essentiellement dans des laboratoires et centres de recherches, où il était important, lorsqu'on lançait un long job de calcul intensif sur le serveur, de ne pas pénaliser les autres utilisateurs connectés. Pour cela, il était très facile d'utiliser la commande `nice` depuis le shell pour rendre notre processus plus gentil au niveau de l'ordonnancement. Ainsi, il s'exécutait lors d'un temps légèrement plus long, mais on évitait les récriminations des autres utilisateurs présents.

Depuis le shell, on utilise :

```
$ nice -n increment ./commande arguments...
```

pour lancer la commande (et ses arguments) avec une valeur de gentillesse augmentée de l'incrément précisé.

Seul *root* peut faire descendre la valeur de gentillesse en dessous d'une limite – configurable – valant généralement zéro. Cette limite peut être consultée (et modifiée) avec la commande `ulimit -e`. Toutefois, cette commande ne fournit pas directement la valeur limite, mais 20-*limite*. Si `ulimit -e` renvoie 15, cela signifie qu'un processus doit avoir les privilèges *root* pour descendre sous la valeur `nice` 20-15 = 5. Par défaut, `ulimit -e` renvoie 20, donc la limite est 20-20 = 0.

Reprenons notre exemple précédent, en lançant deux tâches en parallèle, avec des valeurs `nice` différentes.

```
$ ./exemple-taux-cpu 81000000 & nice -n 5 ./exemple-taux-cpu 81000000
[4861] nb_boucles : 58885942 (73)
[4862] nb_boucles : 19851775 (25)
$ ./exemple-taux-cpu 81000000 & nice -n 10 ./exemple-taux-cpu 81000000
[4883] nb_boucles : 72086172 (89)
[4884] nb_boucles : 7829811 (10)
$ ./exemple-taux-cpu 81000000 & nice -n 19 ./exemple-taux-cpu  81000000
[4909] nb_boucles : 77825517 (96)
[4910] nb_boucles : 1147423 (1)
$
```

Nous constatons qu'il y a de nettes différences de temps CPU accordés aux tâches dont les valeurs `nice` sont différentes, même lorsque l'incrément ne vaut que 5. Voyons ce que donne l'exécution de deux processus, dont les priorités sont aux deux extrêmes de l'échelle permise. Pour cela, nous prenons les droits *root* :

Copyright © 2015 Eyrolles.

```
# nice -n -20 ./exemple-taux-cpu 81000000 & nice -n 19 ./exemple-taux-cpu 81000000
[5045] nb_boucles : 78654399 (97)
[5046] nb_boucles : 15786 (0)
#
```

Le premier processus (le plus agressif) a pu réaliser près de 5 000 fois le nombre d'itérations du second !

Il est possible de modifier, ou de consulter, la courtoisie d'un processus avec l'un des appels système suivants :

```
#include <unistd.h>
int nice (int increment);
```

qui ajoute l'incrément précisé à la gentillesse du processus appelant (et renvoie sa nouvelle valeur) :

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int type, int ident);
int setpriority (int type, int ident, int nice);
```

Dans ces deux derniers appels système, on indique à travers l'argument `type` les processus concernés. S'il vaut `PRIO_PROCESS`, le second argument est le PID du processus visé, s'il vaut `PRIO_PGRP`, l'identifiant est un PGID *(Process Group Identifier)*, et si le type est `PRIO_USER`, le deuxième argument est un UID *(User Identifier)*.

Comme précédemment, on indique en réalité une valeur de courtoisie allant de -20 (le plus prioritaire) à +19 (le plus gentil).

Priorités entre threads

Nous verrons plus loin qu'il est possible d'ordonnancer certains ou tous les threads d'un processus en temps réel et de fixer précisément la priorité de chacun d'entre eux.

En ce qui concerne l'ordonnancement temps partagé, les choses sont plus complexes. La norme Posix précise que la courtoisie fixée avec `setpriority()` concerne tous les threads du processus. Ainsi, il n'est pas possible de fournir de priorité temps partagé pour les différents threads.

La bibliothèque NPTL *(Native Posix Threads Library)*, utilisée actuellement dans la majeure partie des systèmes Linux (du moins ceux qui emploient la GNU LibC), prend le parti de s'éloigner sur ce point de la norme, en autorisant la fonction `nice()` à modifier uniquement la courtoisie du thread appelant.

Voici un exemple dans lequel nous créons cinq threads, chacun prenant une valeur de gentillesse spécifique, puis effectuant des boucles pendant 10 secondes. Le thread principal attend la fin de tous les autres, en récupérant leurs nombres d'itérations, et présente les résultats :

Copyright © 2015 Eyrolles.

exemple-nice-threads.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

void * fonction_thread (void * arg)
{
    long nice_value = (long) arg;
    time_t debut;
    long nb_iterations  = 0;

    nice(nice_value);
    debut = time(NULL);

    while (time(NULL) < (debut + 10))
        nb_iterations ++;

    return (void *) nb_iterations;
}

#define NB_THREADS  5

int main(void)
{
    int i;
    void * resultat;
    long nice_value[NB_THREADS];
    pthread_t thread[NB_THREADS];
    long total_iterations = 0;
    long nb_iterations[NB_THREADS];

    for (i = 0; i < NB_THREADS; i ++)
        nice_value[i] = (i+1) * (20 / NB_THREADS);

    for (i = 0; i < NB_THREADS; i ++)
        pthread_create(&(thread[i]),NULL, fonction_thread,
                       (void *) nice_value[i]);
    for (i = 0; i < NB_THREADS; i ++) {
        pthread_join(thread[i], & resultat);
        nb_iterations[i] = (long) resultat;
        total_iterations += nb_iterations[i];
    }
    for (i = 0; i < NB_THREADS; i ++)
        printf("[%ld] %ld (%.0f)\n", nice_value[i],
               nb_iterations[i],
               100.0*nb_iterations[i]/total_iterations);
    return EXIT_SUCCESS;
}
```

Copyright © 2015 Eyrolles.

L'exécution confirme bien que les threads de la NPTL peuvent recevoir des priorités temps partagé différentes :

```
$ ./exemple-nice-threads
[4] 33857744 (60)
[8] 13777677 (24)
[12] 5601120 (10)
[16] 2320758 (4)
[20] 1204927 (2)
$
```

Notez bien que ce comportement est spécifique et ne sera pas portable sur d'autres Unix, ni sur des systèmes Linux (embarqués, par exemple) employant d'autres bibliothèques C.

Conclusion

Nous avons étudié le fonctionnement de l'ordonnancement temps partagé sous Linux. Dans le prochain chapitre, nous allons voir ses limitations, afin de savoir quelles restrictions pourront nous pousser à préférer un ordonnancement temps réel pour développer une application interactive.

Points clés

L'ordonnancement temps partagé a beaucoup évolué au fil des versions du noyau Linux. Souple, fluide, équitable, le modèle actuel est performant, tant pour les stations de travail personnelles que pour les serveurs à haute charge.

La répartition du temps processeur s'effectue d'abord au niveau des groupes de processus, puis tâche par tâche dans chaque groupe.

On peut fixer la courtoisie – c'est-à-dire l'opposé de la priorité – d'un processus et même d'un thread (spécificité de Linux).

Exercices

Exercice 1 (*)

Utilisez le programme `exemple-taux-cpu` présenté précédemment pour déterminer le nombre de boucles que peut effectuer votre processeur en 10 secondes. Lancez-en deux, puis trois exemplaires en parallèle. Vérifiez les variations sur le nombre de boucles effectuées. Pensez à fixer l'affinité de vos processus sur le même CPU.

Copyright © 2015 Eyrolles.

Exercice 2 (*)

Utilisez la commande `nice` pour préfixer un exemplaire du programme, tandis que vous lancerez l'autre exemplaire à sa priorité naturelle. Voyez-vous une différence ?

Exercice 3 (**)

Écrivez un script shell pour lancer successivement le programme de mesure des boucles aux différents niveaux de `nice` entre 1 et +19 (ou -20 et +19 avec les droits *root*) en parallèle avec une autre instance au niveau `nice` 0. Observez l'échelle des priorités en fonction du niveau de courtoisie des processus.

Exercice 4 (***)

Comparez les résultats de l'exercice précédent sur des noyaux Linux de différentes générations et avec différentes options de compilation :

- Noyau Linux 2.6 avant le 2.6.23 ;
- Linux postérieur au 2.6.23 et antérieur au 2.6.38 ;
- Noyau postérieur au 2.6.38 compilé avec ou sans l'autogroupement des processus ;
- ...

Exercice 5 (***)

En vous appuyant sur la documentation du noyau, placez plusieurs tâches dans des *cgroups* distincts dont les affinités seront disjointes. Peut-on toujours déplacer les processus avec `taskset` ? avec `sched_setaffinity` ? d'un *cgroup* à l'autre ?

Copyright © 2015 Eyrolles.

4

Limitations de l'ordonnancement temps partagé

Dans le chapitre précédent, nous avons examiné les fondements de l'ordonnancement temps partagé. Efficace et performant, il est parfaitement adapté aux tâches courantes d'un poste de développement ou de bureautique. Pourtant, certaines applications (*media player*, entrées-sorties, etc.) s'en satisfont difficilement. Nous allons essayer de comprendre les limites qui handicapent le scheduler temps partagé de Linux pour ces applications.

Mesure du temps

Tout d'abord, nous allons devoir nous appuyer sur des mesures horaires pour vérifier la durée d'opérations critiques. Aussi, devons-nous trouver des outils capables de nous fournir l'heure assez précisément.

La fonction `time()` bien connue fournit l'heure dans un type entier `time_t` qui contient le nombre de secondes écoulées depuis le 1er janvier 1970. Pratique pour mesurer des opérations à l'échelle humaine (*uptime* depuis le *boot*, durée d'exécution d'une grosse compilation, etc.), elle n'est pas assez pointue pour des tâches informatiques précises.

```
time_t time (time_t * heure);
```

Copyright © 2015 Eyrolles.

Heure Unix avec gettimeofday()

La fonction `gettimeofday()` est plus utile car elle remplit une structure `timeval` qui contient un champ `tv_sec` avec la même valeur que `time()` précédemment, mais également un champ `tv_usec` qui contient le complément en microsecondes.

```
#include <sys/time.h>
int gettimeofday (struct timeval * tv,
                  struct timezone * tz);
```

Le second argument de `gettimeofday()` permet de lire le fuseau horaire pour lequel le système est configuré. En pratique, cette valeur est rarement utilisée et lors de la plupart des appels de `gettimeofday()`, le second argument est `NULL`.

Précision des mesures

Nous pouvons nous interroger sur la précision des mesures fournies par `gettimeofday()`. Celle-ci varie sensiblement suivant les systèmes. Voici un petit exemple qui va boucler trente fois (valeur arbitraire) pour lire l'heure et afficher ensuite ses résultats. L'affichage n'est fait qu'après la phase de mesure pour éviter les perturbations liées à la communication avec le terminal.

```
exemple-gettimeofday-01.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>


#define NB_MESURES 30

int main(int argc, char * argv[])
{
    struct timeval tv[NB_MESURES];
    int i;
    for (i = 0; i < NB_MESURES; i ++)
        gettimeofday(& (tv[i]), NULL);
    for (i = 0; i < NB_MESURES; i ++)
        printf("%2d : %ld.%06ld\n", i,
               tv[i].tv_sec, tv[i].tv_usec);
    return EXIT_SUCCESS;
}
```

On peut remarquer le formatage un peu particulier dans le `printf()` : j'affiche le champ « secondes » de la structure `timeval`, puis un point décimal suivi du champ « microsecondes ». Comme ce dernier est compris entre 0 et 999999 (valeur imposée par la norme Posix), je fais précéder d'autant de zéros que nécessaire pour qu'il tienne sur six chiffres.

Voici un exemple d'exécution sur un PC portable :

Copyright © 2015 Eyrolles.

```
$ ./exemple-gettimeofday-01

 0 : 1436801387.696325
 1 : 1436801387.696326
 2 : 1436801387.696326
 3 : 1436801387.696326
 4 : 1436801387.696326
 5 : 1436801387.696326
 6 : 1436801387.696326
 7 : 1436801387.696326
 8 : 1436801387.696326
 9 : 1436801387.696326
10 : 1436801387.696326
11 : 1436801387.696326
12 : 1436801387.696326
13 : 1436801387.696327
14 : 1436801387.696327
15 : 1436801387.696327
16 : 1436801387.696327
17 : 1436801387.696327
18 : 1436801387.696327
19 : 1436801387.696327
20 : 1436801387.696327
21 : 1436801387.696327
22 : 1436801387.696327
23 : 1436801387.696327
24 : 1436801387.696327
25 : 1436801387.696327
26 : 1436801387.696328
27 : 1436801387.696328
28 : 1436801387.696328
29 : 1436801387.696328
[chapitre-04]$
```

Nous pouvons remarquer que les appels successifs renvoient des valeurs parfaitement régulières, les microsecondes se déroulent sans saut (donc notre programme n'a pas été préempté). Par ailleurs, on réussit à faire treize appels à `gettimeofday()` par microseconde. Sur une machine plus puissante, on observera couramment une vingtaine d'appels par microseconde.

Cet appel système est bien adapté pour horodater des événements avec une précision de l'ordre de la microseconde sur ce système.

Voici à présent un extrait d'une exécution sur un Raspberry Pi modèle 1 :

```
 4 : 1411876365.577818
 5 : 1411876365.577818
 6 : 1411876365.577819
 7 : 1411876365.577820
```

Copyright © 2015 Eyrolles.

```
 8 : 1411876365.577821
 9 : 1411876365.577821
10 : 1411876365.577822
11 : 1411876365.577823
12 : 1411876365.577824
13 : 1411876365.577824
14 : 1411876365.577825
15 : 1411876365.577827
16 : 1411876365.577827
17 : 1411876365.577828
18 : 1411876365.577828
19 : 1411876365.577829
20 : 1411876365.577830
```

Cette fois, nous constatons que la précision des mesures n'est plus aussi régulière. Parfois deux invocations successives (4 et 5, 8 et 9, etc.) se déroulent durant la même microseconde, parfois il peut s'écouler plus d'une microseconde entre deux appels (14 et 15).

Nous allons encore exécuter ce même programme sur une autre plate-forme à processeur Arm pour Linux embarqué (Igep v.2). Voici un extrait du résultat :

```
# ./exemple-gettimeofday-arm
 0 : 225.572906
 1 : 225.572906
 2 : 225.572906
 3 : 225.572906
 4 : 225.572906
 5 : 225.572906
 6 : 225.572906
 7 : 225.572906
 8 : 225.572937
 9 : 225.572937
10 : 225.572937
11 : 225.572937
[...]
25 : 225.572937
26 : 225.572967
27 : 225.572967
28 : 225.572967
29 : 225.572967
#
```

Les valeurs sont nettement moins précises : en fait, nous voyons que le noyau ne peut nous fournir l'heure qu'avec une granularité de 30 microsecondes environ, même si l'appel système proprement dit dure moins longtemps (environ 1,7 microsecondes).

Il n'est donc pas inutile de vérifier, avec le petit programme précédent, la précision de `gettimeofday()` directement sur le système cible.

Copyright © 2015 Eyrolles.

Horloges Posix

La norme Posix a introduit quelques fonctions permettant de mesurer l'heure avec une précision parfois meilleure que `gettimeofday()`.

```
#include <time.h>
int clock_getres  (clockid_t horloge,
                   struct timespec * resolution);
int clock_gettime (clockid_t horloge,
                   struct timespec * heure);
int clock_settime (clockid_t horloge,
                   const struct timespec * heure);
```

`clock_getres()` permet de connaître la résolution avec laquelle l'horloge précisée en premier argument nous fournit l'heure. Attention, il ne s'agit pas de la précision de la mesure, cela signifie simplement que toutes les valeurs fournies seront des multiples de cette résolution.

`clock_gettime()` lit l'heure et `clock_settime()` la configure. Ce dernier appel nécessite généralement les privilèges *root*.

Les horloges précisées en premier argument peuvent prendre l'une des valeurs suivantes :

Valeur	Signification
`CLOCK_REALTIME`	L'horloge du système qui fournit l'heure absolue réelle (réglable).
`CLOCK_MONOTONIC`	Une horloge non modifiable, dont la valeur initiale au *boot* est laissée à la discrétion du noyau.
`CLOCK_PROCESS_CPUTIME_ID`	Une horloge dont l'identifiant est fourni par `pthread_getcpuclockid()`, qui indique le temps CPU consommé par l'ensemble des threads du processus.
`CLOCK_THREAD_CPUTIME_ID`	Une horloge dont l'identifiant est fourni par `clock_getcpuclockid()`, indiquant le temps CPU consommé par le thread.
`CLOCK_MONOTONIC_RAW` (spécifique Linux, non Posix)	Une horloge renvoyant l'heure – non modifiable – fournie par le matériel.

Voici un petit exemple permettant d'étudier les différentes horloges.

exemple-clock-gettime-01.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void affiche_caracteristiques(const char * nom,
                              clockid_t clock)
{
    struct timespec res;
    struct timespec hre;
```

Copyright © 2015 Eyrolles.

```
    printf("%s : \n", nom);
    clock_getres  (clock, & res);
    clock_gettime (clock, & hre);
    printf("   resolution : %ld.%09ld\n",
                 res.tv_sec, res.tv_nsec);
    printf("   heure :      %ld.%09ld\n",
                 hre.tv_sec, hre.tv_nsec);
}

int main(void)
{
    affiche_caracteristiques("CLOCK_REALTIME",
                             CLOCK_REALTIME);
    affiche_caracteristiques("CLOCK_MONOTONIC",
                             CLOCK_MONOTONIC);
    affiche_caracteristiques("CLOCK_PROCESS_CPUTIME_ID",
                             CLOCK_PROCESS_CPUTIME_ID);
    affiche_caracteristiques("CLOCK_THREAD_CPUTIME_ID",
                             CLOCK_THREAD_CPUTIME_ID);
    affiche_caracteristiques("CLOCK_MONOTONIC_RAW",
                             CLOCK_MONOTONIC_RAW);
    return EXIT_SUCCESS;
}
```

L'exécution montre que sur ce système, la résolution est la nanoseconde ; les différentes horloges ont des origines variées : 01/01/1970 pour `CLOCK_REALTIME`, heure de *boot* pour `CLOCK_MONOTONIC` et `CLOCK_MONOTONIC_RAW`, et heure de démarrage du processus ou du thread pour les deux autres.

```
$ ./exemple-clock-gettime-01
CLOCK_REALTIME :
   resolution : 0.000000001
   heure :      1316672165.783063186
CLOCK_MONOTONIC :
   resolution : 0.000000001
   heure :      142870.818423848
CLOCK_PROCESS_CPUTIME_ID :
   resolution : 0.000000001
   heure :      0.001803095
CLOCK_THREAD_CPUTIME_ID :
   resolution : 0.000000001
   heure :      0.001821238
CLOCK_MONOTONIC_RAW :
   resolution : 0.000000000
   heure :      142870.818356852
$
```

Pour vérifier si la précision des mesures est meilleure que la microseconde, nous allons réaliser le même travail qu'avec `gettimeofday()` : appel en boucle, puis affichage des valeurs.

Copyright © 2015 Eyrolles.

```
exemple-clock-gettime-02.c  :
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


int main(void)
{
    int i;
    struct timespec mesure[30];
    for (i = 0; i < 30; i ++)
        clock_gettime(CLOCK_REALTIME, &(mesure[i]));
    for (i = 0; i < 30; i ++)
        fprintf(stdout, "%ld.%09ld\n",
                mesure[i].tv_sec, mesure[i].tv_nsec);
    return EXIT_SUCCESS;
}
```

En exécutant ce programme sur le PC portable qui nous donnait treize appels par microseconde avec `gettimeofday()`, nous obtenons :

```
$ ./exemple-clock-gettime-02

1436801515.063343311
1436801515.063343480
1436801515.063343559
1436801515.063343642
1436801515.063343721
1436801515.063343804
1436801515.063343883
1436801515.063343965
[...]

1436801515.063345499
1436801515.063345582
1436801515.063345661
1436801515.063345744
$
```

En fait, ces résultats ne sont pas très parlants, mais nous pouvons afficher sur chaque ligne, grâce à un petit script nommé `Awk`, la différence par rapport à la ligne précédente :

```
$ ./exemple-clock-gettime-02 | awk '(NR>1){printf "%.9f\n", $1-p} {p=$1}'
0.000000238
0.000000238
0.000000000
0.000000000
0.000000238
0.000000000
[...]
```

Copyright © 2015 Eyrolles.

```
0.000000000
0.000000000
0.000000238
0.000000000
0.000000000
0.000000238
[...]
$
```

Les mesures ont une précision inférieure à 0,25 microseconde, ce qui est correct pour un système de ce type. Avec une machine plus puissante, on pourrait atteindre une précision de 50 nanosecondes.

Tâches périodiques

La plupart des systèmes devant répondre à des contraintes temporelles relativement fortes s'appuient sur des tâches périodiques, qui doivent s'exécuter avec la meilleure granularité et la moindre variabilité possibles.

Il existe plusieurs appels système permettant de réaliser des actions périodiques. La première méthode est basée sur l'appel système `setitimer()`, classique sous Unix, la seconde utilise `timer_create()`, normalisé par l'extension de Posix consacrée au temps réel. Ces deux fonctions s'appuient sur les mêmes mécanismes dans le noyau et offriront donc des performances similaires. Notons toutefois que `setitimer()` ne permet d'enregistrer qu'un seul timer par processus – ou plutôt trois timers, mais ils sont complémentaires – tandis que `timer_create()` permet d'en utiliser plusieurs (la limite dépend du système).

Timers Unix classiques

La fonction `setitimer()` active un timer qui enverra régulièrement un signal au processus appelant. Ce dernier devra capturer le signal et effectuer le travail périodique dans sa routine de traitement.

```
#include <sys/time.h>

int setitimer (int type, const struct itimerval * nouveau,
               struct itimerval * precedent);
```

La structure `itimerval` contient deux champs.

Champs	Contenu
`it_value`	Durée avant le déclenchement du premier timer.
`it_interval`	Durée entre deux déclenchements successifs.

Le premier argument de cette fonction précise les conditions dans lesquelles les durées indiquées ci-dessus sont décomptées.

Copyright © 2015 Eyrolles.

Type	Signification
ITIMER_REAL	Les durées sont considérées comme réelles, mesurées à partir de l'horloge système. Le signal envoyé à expiration sera le signal SIGALRM.
ITIMER_VIRTUAL	Les durées ne sont décomptées que lorsque le processus est en cours d'exécution. Le signal est SIGVTALRM.
ITIMER_PROF	Les durées concernent uniquement les moments où le processus s'exécute en mode utilisateur. Le signal envoyé est SIGPROF.

Le dernier argument de `setitimer()` permet de récupérer la programmation précédente si l'on souhaite la restaurer ultérieurement. Bien entendu, si nous ne sommes pas intéressés par cette valeur, il suffira de transmettre un pointeur `NULL`.

Voici un petit exemple d'utilisation avec un timer fonctionnant à la seconde (nous augmenterons nettement la fréquence par la suite). Lorsque `SIGALRM` arrive sur le processus, la fonction gérant ce signal va afficher un astérisque sur la sortie du processus.

```
exemple-setitimer-01.c :
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

void handler_signal(int inutilise)
{
    fprintf(stderr, "*");
}

int main(int argc, char * argv[])
{
    struct itimerval timer;
    // Installer le handler du signal SIGALRM
    signal(SIGALRM, handler_signal);
    // Préparer la structure itimerval
    timer.it_interval.tv_sec = 1;
    timer.it_interval.tv_usec = 0;
    timer.it_value = timer.it_interval;
    // Programmer le timer
    setitimer(ITIMER_REAL, & timer, NULL);
    while (1)
        pause();
    return EXIT_SUCCESS;
}
```

Copyright © 2015 Eyrolles.

À l'exécution, un astérisque est affiché toutes les secondes jusqu'à l'arrêt par l'utilisateur :

```
$ ./exemple-setitimer-01
************************************ (Contrôle-C)
$
```

Timers Posix

Les timers proposés par la norme Posix reposent sur un principe légèrement différent, même si les fonctionnalités sous-jacentes sont les mêmes dans le noyau.

On commence par créer un timer avec la fonction :

```
#include <time.h>

int timer_create(clockid_t horloge,
                 struct sigevent * notification,
                 timer_t * timer);
```

Cette fonction initialise et remplit le pointeur passé en troisième argument. Le premier argument indique au noyau sur quel type d'horloge système il doit construire notre timer – les horloges sont identiques à celles que nous avons vues pour `clock_gettime()` – et le deuxième argument décrit comment notre processus gérera l'occurrence du timer.

Les champs de la structure `sigevent` ont les significations suivantes.

Champs	Signification
`sigev_notify`	Type de notification : - `SIGEV_NONE` : aucune ; - `SIGEV_SIGNAL` : délivrance d'un signal ; - `SIGEV_THREAD` : démarrage d'un thread ; - `SIGEV_THREAD_ID` : envoi d'un signal a un thread spécifique.
`sigev_signo`	Si notification par signal : numéro du signal à délivrer. On conseille généralement d'utiliser un signal temps réel.
`sigev_value`	Si notification par signal temps réel : valeur qui accompagne le signal. Si notification par thread : valeur passée en argument de la fonction.
`sigev_notify_function`	Si notification par thread : fonction à démarrer.
`sigev_notify_attributes`	Si notification par thread : attributs du nouveau thread.
`sigev_notify_thread_id`	Si notification `THREAD_ID` : identifiant du thread devant recevoir le signal.

La notification par démarrage d'un thread est assurée par la GlibC alors que l'envoi d'un signal est réalisé par le noyau. Cette dernière méthode est peut-être légèrement plus efficace.

Une fois le timer créé, on le configure en indiquant sa période et le délai avant le premier déclenchement à l'aide de l'appel :

Copyright © 2015 Eyrolles.

```
int timer_settime (timer_t timer, int absolu,
                   const struct itimerspec * spec,
                   struct itimerspec * precedente);
```

Le premier argument est le timer obtenu auparavant, le second argument un indicateur précisant si la structure `itimerspec` contient une durée ou une heure absolue de déclenchement (en général, on utilise une durée, et le second argument est nul).

La structure `itimerspec` ressemble à la structure `itimerval` vue précédemment, à la différence que les temps sont compris dans des structures `timerspec` dont la résolution est en nanosecondes plutôt qu'en microsecondes.

Voici un petit exemple où l'on programme deux timers dans le même processus, le premier avec une période d'une seconde, le second avec une période d'un quart de seconde.

```
exemple-timer-create-01.c :
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>

void handler_signal_1(int inutilise)
{
    fprintf(stderr, "1 ");
}

void handler_signal_2(int inutilise)
{
    fprintf(stderr, "2 ");
}

int main(int argc, char * argv[])
{
    timer_t timer_1, timer_2;
    struct sigevent event_1, event_2;
    struct itimerspec spec_1, spec_2;

    // Installer les handlers
    signal(SIGRTMIN+1, handler_signal_1);
    signal(SIGRTMIN+2, handler_signal_2);

    // Indiquer la notification désirée
    event_1.sigev_notify = SIGEV_SIGNAL;
    event_1.sigev_signo  = SIGRTMIN+1;

    // Configurer la période du timer
    spec_1.it_interval.tv_sec  = 1;
    spec_1.it_interval.tv_nsec = 0;
    spec_1.it_value = spec_1.it_interval;
```

Copyright © 2015 Eyrolles.

```
    // Allouer le timer
    if (timer_create(CLOCK_REALTIME, &event_1,
                     &timer_1) != 0) {
        perror("timer_create");
        exit(EXIT_FAILURE);
    }

    // Mêmes opérations pour le second timer
    event_2.sigev_notify = SIGEV_SIGNAL;
    event_2.sigev_signo  = SIGRTMIN+2;
    spec_2.it_interval.tv_sec  = 0;
    spec_2.it_interval.tv_nsec = 250000000; // 0,25 sec.
    spec_2.it_value = spec_2.it_interval;
    if (timer_create(CLOCK_REALTIME, &event_2,
                     &timer_2) != 0) {
        perror("timer_create");
        exit(EXIT_FAILURE);
    }

    // Programmer les timers
    if ((timer_settime(timer_1, 0, &spec_1, NULL) != 0)
     || (timer_settime(timer_2, 0, &spec_2, NULL) != 0)) {
        perror("timer_settime");
        exit(EXIT_FAILURE);
    }
    while (1)
        pause();
    return EXIT_SUCCESS;
}
```

À l'exécution, nous voyons bien que le processus reçoit les deux timers :

```
$ ./exemple-timer-create-01
2 2 2 2 1 2 2 2 2 1 2 2 2 2 1 2 2 2 2 1 2 2 2 2 1 2 2 2 2 1 2 2 2 2 1 2 2 2
(Contrôle-C)
$
```

Granularité

La granularité des timers sous Linux a beaucoup évolué au cours de son histoire, et dépend encore sensiblement des architectures et des options de compilation. Il est donc intéressant de s'arrêter un instant sur les étapes de la gestion des timers par le noyau Linux.

Dans les premières versions de Linux, les timers logiciels étaient gérés par une interruption périodique – aussi nommée tick –, configurée au démarrage du système. Sur l'architecture PC, cette interruption était déclenchée par un composant timer matériel, que le noyau programmait

Copyright © 2015 Eyrolles.

au *boot* avec une fréquence de 100 Hz. Ainsi, le handler de l'interruption (IRQ 0) était invoqué toutes les 10 ms. Son rôle était alors de gérer toutes les opérations temporelles revenant au noyau :

- maintien de l'heure système et mise à jour du compteur des *jiffies* (ticks écoulés depuis le *boot*) ;
- décompte des tranches de temps accordées aux processus avant préemption ;
- vérification des délais restants lorsqu'un processus est bloqué dans un appel système temporisé – comme `sem_timedwait()` pour les sémaphores ;
- envoi des signaux aux processus dont les timers logiciels ont expirés ;
- etc.

La granularité des timers logiciels était donc de 10 ms. Toutes les durées des opérations périodiques devaient être des multiples de cette durée.

Le manque de finesse dans la configuration des timers a poussé les développeurs du noyau à faire évoluer la fréquence de l'interruption sous-jacente dans les premières versions du noyau 2.6 en 2004. Les améliorations matérielles survenues permettaient d'utiliser une fréquence de 1 kHz, et les timers pouvaient programmer des périodes multiples de 1 ms.

Néanmoins, cette fréquence devenait trop élevée pour certaines machines, notamment les systèmes embarqués pour lesquels Linux représentait une alternative de plus en plus intéressante. À partir du noyau 2.6.10 (fin 2004), la fréquence est devenue configurable au moment de la compilation du noyau. On pouvait choisir entre 100 Hz, 250 Hz et 1 kHz. Rapidement une valeur supplémentaire 300 Hz est apparue car c'est un multiple entier de 50 Hz et de 60 Hz, les fréquences utilisées par les applications multimédias en Europe et aux États-Unis. Avec un timer matériel à 300 Hz, réaliser un traitement périodique à 50 Hz ou 60 Hz est simple et efficace.

Lors de la configuration du noyau avant compilation (voir annexe A), le choix de la fréquence s'effectue dans le menu Processor Type and Features (ou suivant les architectures : Kernel Features, Kernel Options, etc.), sous-menu Timer Frequency.

L'inconvénient principal du tick, tel qu'il a été employé dans toutes ces versions de Linux, est lié à la consommation du processeur. Lorsque le système n'a rien à faire, il est possible d'endormir le processeur en attente d'une interruption (chapitre 2). Ainsi, le processeur consomme beaucoup moins d'énergie et refroidit. Il sera automatiquement réveillé par une interruption dès qu'un utilisateur pressera une touche ou déplacera la souris, dès qu'un paquet réseau sera reçu sur une interface ou un événement détecté sur une carte de communication externe, etc. Malheureusement, avec le principe du tick, le noyau ne reste jamais endormi plus d'une milliseconde, puisque l'interruption timer le réveille pour vérifier s'il doit réaliser une des tâches décrites dans la liste vue plus haut.

Avec les améliorations du matériel, il est alors devenu envisageable de ne plus utiliser d'interruption périodique, mais de programmer le timer physique à chaque fois que l'on endort le processeur, en planifiant un réveil pour la prochaine tâche à réaliser. Ainsi, le sommeil peut durer plus longtemps, jusqu'à l'expiration du premier timer logiciel ou l'arrivée d'une interruption externe. Ce principe (nommé les *dynamic ticks*) a été intégré en 2006.

Copyright © 2015 Eyrolles.

Jusqu'alors, ce mécanisme n'était pas possible en raison du temps (trop long) de programmation des timers matériels et des fluctuations entre l'instant de leur programmation et le déclenchement de la première interruption.

L'évolution suivante consistait à supprimer complètement les ticks, le noyau s'appuyant en permanence sur la mise à jour du timer matériel disponible (programmation des délais d'exécution accordés aux tâches, expiration des *timeouts* des appels système, déclenchement des signaux des timers logiciels, etc.).

> L'option *tickless* est parfois déconseillée pour les systèmes temps réel car elle laisse le processeur s'endormir et descendre dans des états de sommeil profond (comme nous en avons parlé dans le chapitre précédent), ce qui augmente le temps de réveil et donc de réponse aux interruptions. Suivant les situations, il faudra réaliser un arbitrage entre les besoins temps réels (rapidité de réveil après occurrence d'une interruption) et les contraintes liées aux systèmes embarqués (consommation la plus économique possible).

Simultanément, nous avons assisté à l'intégration du patch HR-Timers *(High-Resolution Timers)* permettant des programmations de précision (à condition toutefois qu'elles soient supportées par le matériel).

Les noyaux Linux actuels (4.0 par exemple) peuvent être configurés à la compilation comme *tickless* (sans tick), *tickless idle* (ticks uniquement durant l'activité du système) ou *periodic timer ticks* (ticks réguliers et permanents) ; ces options se trouvent, dans le sous-menu Timers subsystem du menu General setup. Les deux modes *tickless* permettent d'optimiser l'utilisation du processeur (les tâches sont moins interrompues) et d'économiser la consommation lors des mises en sommeil.

En outre, grâce au HR-Timers, il est à présent possible de bénéficier de timers logiciels de période inférieure à la milliseconde ou de sommeils ponctuels avec `nanosleep()` de durées précises à quelques microsecondes près.

Précision

La granularité des timers est aujourd'hui inférieure à la milliseconde, mais qu'en est-il de leur précision ? Quelle sont les fluctuations à attendre dans le déclenchement des opérations périodiques ? On utilise pour nommer cette imprécision dans l'instant exact de déclenchement le terme de « gigue », ou plus couramment le terme anglais *jitter.*

Nous allons construire un petit exemple qui programme un timer Posix avec une période fournie en argument sur la ligne de commandes. À chaque réception de signal, nous lirons l'heure et calculerons la différence avec le signal précédent. Au bout de cinq secondes, nous afficherons sur la sortie standard toutes les mesures.

exemple-timer-create-02.c :

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
```

Copyright © 2015 Eyrolles.

```
static timer_t timer = (timer_t) 0;
static long int * mesure = NULL;
static int nb_mesures = 0;
static int nb_total_mesures = 0;

void handler_signal(int inutilise)
{
  struct timespec heure;
  static struct timespec preced = { 0, 0} ;

  clock_gettime (CLOCK_REALTIME, & heure);
  if (preced.tv_sec > 0) {
    mesure[nb_mesures]  = heure.tv_sec  - preced.tv_sec;
    mesure[nb_mesures] *= 1000000000;
    mesure[nb_mesures] += heure.tv_nsec - preced.tv_nsec;
    nb_mesures ++;
  }
  preced = heure;
  if (nb_mesures == nb_total_mesures)
    timer_delete(timer);
}


int main(int argc, char * argv[])
{
  int i;
  long int periode;
  struct sigevent event;
  struct itimerspec spec;

  if ((argc != 2)
   || (sscanf(argv[1], "%ld", & periode) != 1)){
    fprintf(stderr, "usage: %s periode_en_microsec\n",
                    argv[0]);
    exit(EXIT_FAILURE);
  }

  // Allouer le tableau de mesures
  nb_total_mesures = 5000000 / periode; // 5 secondes
  if ((mesure = calloc(nb_total_mesures,
                    sizeof(struct timespec))) == NULL){
    perror("calloc");
    exit(EXIT_FAILURE);
  }

  // Configurer le timer
  signal(SIGRTMIN, handler_signal);
  event.sigev_notify = SIGEV_SIGNAL;
  event.sigev_signo  = SIGRTMIN;
```

Copyright © 2015 Eyrolles.

```
    periode = periode * 1000;  // en nanosecondes
    spec.it_interval.tv_sec  = periode / 1000000000;
    spec.it_interval.tv_nsec = periode % 1000000000;
    spec.it_value = spec.it_interval;
    // Allouer le timer
    if (timer_create(CLOCK_REALTIME, &event, &timer) != 0){
      perror("timer_create");
      exit(EXIT_FAILURE);
    }
    // Programmer le timer
    if (timer_settime(timer, 0, &spec, NULL) != 0) {
      perror("timer_settime");
      exit(EXIT_FAILURE);
    }
    // Attendre le dernier signal
    while (nb_mesures < nb_total_mesures)
      pause();
    // Afficher les mesures en microsecondes
    for (i = 0; i < nb_total_mesures; i ++)
          printf("%ld\n", mesure[i]/1000);
      return EXIT_SUCCESS;
}
```

Les résultats que fournit ce programme (exécuté sur un Raspberry Pi 2) ne sont pas très intéressants tels quels, il nous faut les traiter à nouveau pour obtenir des informations pertinentes. Nous allons donc sauvegarder les valeurs dans un fichier par une redirection de la sortie du processus.

```
$ ./exemple-timer-create-02  1000 > data-01.txt
     (cinq secondes s'écoulent...)
$ cat data-01.txt

945
994
997
1006
994
998
999
1000
998
1000
[...]
$
```

Nous pouvons à présent traiter ces données pour en extraire quelques informations statistiques. Le programme suivant travaille sur un flux quelconque de valeurs, et affiche finalement les valeurs minimale, maximale et moyenne ainsi que l'écart-type. Nous le réutiliserons à nouveau dans les chapitres à venir.

Copyright © 2015 Eyrolles.

```
calculer-statistiques.c  :
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LG_LIGNE   1024

int main(int argc, char * argv[])
{
  // L'algorithme implémenté par ce programme est
  // présenté sur la page Wikipédia "Algorithms
  // for Calculating Variance". Il est construit
  // autour d'une idée de Donald E. Knuth décrite
  // dans le second volume de "The Art of Computer
  // Programming".

  char ligne[LG_LIGNE];

  double mesure;
  int nb_mesures  = 0;
  double minimum;
  double maximum;
  double moyenne  =  0.0;
  double variance =  0.0;
  double delta;
  double ecart_type;

  while (fgets(ligne, LG_LIGNE, stdin) != NULL) {
    if (sscanf(ligne, "%lf", & mesure) != 1) {
      fprintf(stderr, "Erreur de lecture, ligne %d\n",
              nb_mesures+1);
      exit(EXIT_FAILURE);
    }
    nb_mesures ++;
    if ((nb_mesures == 1) || (minimum > mesure))
      minimum = mesure;
    if ((nb_mesures == 1) || (maximum < mesure))
      maximum = mesure;
    delta = mesure - moyenne;
    moyenne  += delta / nb_mesures;
    variance += delta * (mesure - moyenne);
  }
  if (nb_mesures > 1)
    variance = variance / (nb_mesures - 1);

  ecart_type = sqrt(variance);

  printf(" Nb mesures = %d\n Minimum = %.0lf\n"
         " Maximum = %.0lf\n" Moyenne = %.0lf\n"
```

Copyright © 2015 Eyrolles.

```
          " Ecart-type = %.0lf\n",
           nb_mesures, minimum, maximum,
           moyenne, ecart_type);

  return EXIT_SUCCESS;
}
```

Voici les éléments statistiques obtenus à partir des valeurs précédentes :

```
$ ./calculer-statistiques < data-01.txt
 Nb mesures = 5000
 Minimum = 871
 Maximum = 1224
 Moyenne = 999
 Ecart-type = 7
$
```

Nous observons que les déclenchements du timer sont bien espacés en moyenne de 1 milliseconde, ce qui correspond parfaitement à la période demandée. La précision de chaque déclenchement semble relativement bonne, puisque l'écart-type par rapport à cette valeur de consigne est de 7 microsecondes. De même, les valeurs extrêmes ne sont pas exagérément éloignées, il ne s'est jamais écoulé plus de 1,22 milliseconde entre deux déclenchements successifs du timer.

Ces informations sont utiles, mais il est généralement préférable en ingénierie temps réel de disposer d'une représentation graphique où l'on voit en abscisse les valeurs mesurées et en ordonnée le nombre de mesures concernées. Pour cela, le petit programme suivant va prendre nos données mesurées et sortir un tableau permettant de construire un histogramme.

```
calculer-histogramme.c  :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LG_LIGNE   1024

int main(int argc, char * argv[])
{
  char ligne[LG_LIGNE];
  int nb_mesures;
  int nb_echantillons;

  double mesure;
  double minimum = -1;
  double maximum = 0;
  double largeur_echantillon;
  int *  echantillons;
```

Copyright © 2015 Eyrolles.

```
  int i;

  if ((argc != 4)
   || (sscanf(argv[1], "%d", & nb_echantillons) != 1)
   || (sscanf(argv[2], "%lf", & minimum) != 1)
   || (sscanf(argv[3], "%lf", & maximum) != 1)) {
    fprintf(stderr, "usage: %s nb_echantillons minimum"
                    " maximum\n", argv[0]);
    exit(EXIT_FAILURE);
  }
  if (nb_echantillons <= 0) {
    fprintf(stderr, "Valeur invalide : %d\n",
            nb_echantillons);
    exit(EXIT_FAILURE);
  }

  largeur_echantillon = (maximum - minimum)
                        / nb_echantillons;
  echantillons = calloc(nb_echantillons, sizeof(int));

  nb_mesures = 0;
  while (fgets(ligne, LG_LIGNE, stdin) != NULL) {
    if (sscanf(ligne, "%lf", & mesure) != 1) {
      fprintf(stderr, "Erreur de lecture, ligne %d\n",
              nb_mesures+1);
      exit(EXIT_FAILURE);
    }
    i = (int) (mesure - minimum )/ largeur_echantillon;
    if ((i < 0) || (i >= nb_echantillons)) {
      fprintf(stderr, "Valeur invalide ligne %d\n",
                      nb_mesures + 1);
      exit(EXIT_FAILURE);
    }
    echantillons[i] ++;
  }
  for (i = 0; i < nb_echantillons; i ++)
    printf("%.0lf %d\n",
           minimum + (i + 0.5) * largeur_echantillon,
           echantillons[i]);
  return EXIT_SUCCESS;
}
```

Copyright © 2015 Eyrolles.

Appliqué à notre fichier `data1` précédent, le programme donne :

```
$ ./calculer-histogramme 100 750 1250 < data-01.txt
752 0
758 0
762 0
768 0
[...]
868 0
872 1
878 1
882 0
888 1
[...]
988 11
992 300
998 2745
1002 1557
1008 356
1012 6
1018 0
1022 1
1028 0
[...]
```

Les valeurs fournies sur la ligne de commandes (nombre d'intervalles et limites minimale et maximale) sont déterminées arbitrairement à partir des résultats du calcul statistique précédent.

Pour visualiser graphiquement le résultat, je vous propose d'utiliser l'outil `gnuplot` grâce à ce petit script :

afficher-histogramme.sh :

```
#! /bin/sh

if [ $# -lt 2 ]
then
  echo "usage: $0 <fic-histogramme> <titre> [<sortie>]"
    exit 1
fi

if [ $# -eq 3 ]
then
    output="$3"
else
    output="|display gif:-"
fi

gnuplot <<- EOF
    set terminal gif size 800,600
    set title "$2"
```

Copyright © 2015 Eyrolles.

```
    set output '$output'
    # Echelle logarithmique en ordonnee
    set logscale y 10
    # Afficher le contenu du fichier
    plot [] [0.1:] "$1" notitle with boxes
    # Attendre click de souris
    pause mouse
EOF
```

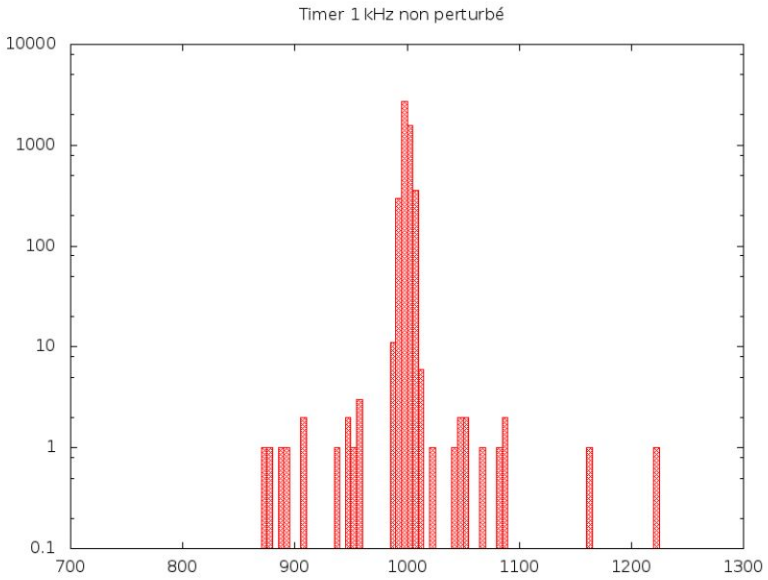
Nous relançons le programme précédent avec une redirection de sa sortie standard dans un fichier, que nous soumettons à notre script.

```
$ ./calculer-histogramme 100 750 1257 < data-01.txt > histo-01.txt
$ ./afficher-histogramme.sh histo-01.txt "Timer 1kHz non perturbé
```

L'histogramme représenté à la figure 4-1 s'affiche alors. Les durées entre déclenchements successifs du timer (mesurés en nanosecondes) sont indiquées en abscisse, leurs nombres d'occurrences en ordonnée. Notez que l'échelle des ordonnées est logarithmique afin de laisser apparaître les occurrences uniques des valeurs extrêmes.

Figure 4-1

Timer 1 kHz non perturbé



Ce graphique nous présente un résultat correct pour un système temps partagé : une large majorité des mesures sont très proches de la valeur de consigne et quelques écarts existent, en nombre limité. En outre, la dispersion est relativement faible, puisque la totalité des mesures se trouve dans un intervalle dont la largeur est d'environ 50 % de la période demandée.

Nous allons à présent *stresser* un peu notre programme en chargeant son processeur par d'autres tâches. Le petit programme suivant perturbe le fonctionnement du système en effectuant régulièrement des boucles actives, puis en dormant quelques microsecondes.

Copyright © 2015 Eyrolles.

Attention, sur certains systèmes, les mesures avec un processeur chargé au maximum peuvent être trompeuses. Nous avons évoqué dans le chapitre 2, section « Entrées-sorties avec interruptions », les modes de sommeil des processeurs. Si un CPU est endormi dans un sommeil profond, son temps de démarrage après réveil par un signal peut être plus long que celui d'un CPU chargé par une boucle active moins prioritaire que le gestionnaire de signal. Il convient donc, pour perturber efficacement un processeur, d'alterner des périodes d'activité intense et des périodes de sommeil.

```
exemple-perturbateur.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

int main(void)
{
  struct timeval heure;
  struct timeval debut_boucle;
  struct timeval debut_programme;
  long long int  difference;

  gettimeofday(& debut_programme, NULL);

  while (1) {
    gettimeofday(& debut_boucle, NULL);
    do {
      gettimeofday(& heure, NULL);
      difference  = heure.tv_sec - debut_boucle.tv_sec;
      difference *= 1000000;
      difference += heure.tv_usec - debut_boucle.tv_usec;
    } while (difference < 100);

    usleep(20);

    difference  = heure.tv_sec - debut_programme.tv_sec;
    difference *= 1000000;
    difference += heure.tv_usec - debut_programme.tv_usec;

    if ((heure.tv_sec - debut_programme.tv_sec) > 10)
      break;
  }
  return EXIT_SUCCESS;
}
```

Nous lançons le perturbateur sur un terminal et la mesure des timers sur un autre, en s'assurant – grâce à taskset – qu'ils soient tous deux sur le même processeur.

Copyright © 2015 Eyrolles.

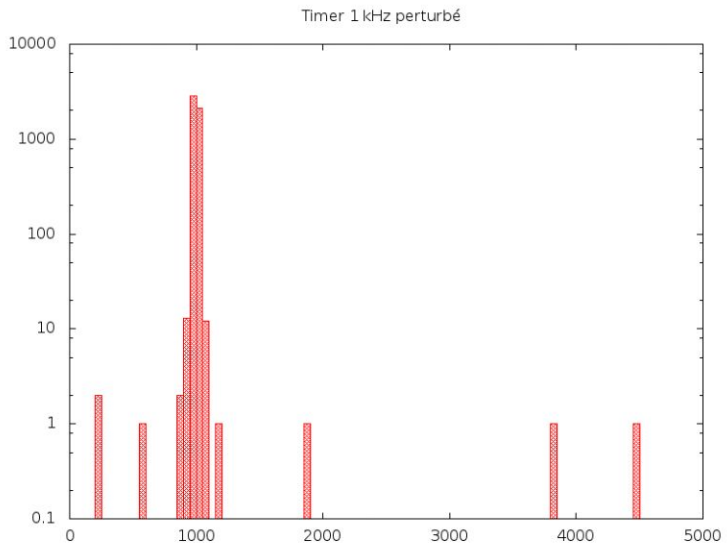
```
                          $ taskset -c 0 ./exemple-perturbateur
$ taskset -c 0 ./exemple-timer-create-02 1000 > data-02.txt
$
$
$ ./calculer-statistiques < data-02.txt
 Nb mesures = 5000
 Minimum = 211
 Maximum = 4489
 Moyenne = 1000
 Ecart-type = 67
$
```

Les résultats semblent moins bons. En les vérifiant sur l'histogramme, nous obtenons le graphique représenté à la figure 4-2. Attention, les valeurs en abscisse ont été sensiblement étendues vers la droite. En effet, on note non seulement une dispersion un peu plus large autour de la valeur de consigne (1 milliseconde), mais également des valeurs extrêmes s'étendant jusqu'à 4,5 millisecondes.

Un timer dont le traitement est réalisé avec un ordonnancement temps partagé n'est donc visiblement pas capable de répondre de manière fiable à des contraintes temporelles.

Figure 4-2

Timer 1 kHz perturbé



Préemption des tâches

Nous avons observé les limites du temps partagé en ce qui concerne le comportement périodique. Nous savons également que les tâches peuvent être préemptées à tout moment pour laisser passer d'autres tâches jugées plus prioritaires par l'ordonnanceur. Nous allons observer – et mesurer – ce phénomène. Pour cela, nous allons utiliser un petit programme qui boucle en

Copyright © 2015 Eyrolles.

observant l'évolution de l'heure, mesurée avec `gettimeofday()` – que l'on pourrait remplacer par `clock_gettime()` avec des résultats similaires. Lorsque le programme détecte une durée supérieure à un certain seuil (fourni en argument sur sa ligne de commande) entre deux lectures successives, il note les valeurs. Au bout de cinq secondes (ou après 100 préemptions), il se termine et affiche les résultats.

exemple-gettimeofday-02.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>


#define MAX_MESURES 100

int main(int argc, char * argv[])
{
    long int seuil;
    struct timeval heure;
    struct timeval debut;
    struct timeval precedente;
    struct timeval avant_preemption[MAX_MESURES];
    struct timeval apres_preemption[MAX_MESURES];
    long long int difference;

    int nb_mesures = 0;
    int i;

    if ((argc != 2)
     || (sscanf(argv[1], "%ld", & seuil) != 1)) {
        fprintf(stderr, "usage: %s seuil-en-microsec\n",
                        argv[0]);
        exit(EXIT_FAILURE);
    }

    gettimeofday(& debut, NULL);
    precedente = debut;

    do {
        gettimeofday(& heure, NULL);
        difference  = heure.tv_sec - precedente.tv_sec;
        difference *= 1000000;
        difference += heure.tv_usec - precedente.tv_usec;
        if (difference >= seuil) {
            avant_preemption[nb_mesures] = precedente;
            apres_preemption[nb_mesures] = heure;
            nb_mesures ++;
            if (nb_mesures >= MAX_MESURES)
                break;
        }
```

Copyright © 2015 Eyrolles.

```
        precedente = heure;
    } while (heure.tv_sec - debut.tv_sec <= 5);
    for (i = 0; i < nb_mesures ; i ++)
        printf("[%d] %ld.%06ld  ->  %ld.%06ld\n",
               i + 1,
               avant_preemption[i].tv_sec,
               avant_preemption[i].tv_usec,
               apres_preemption[i].tv_sec,
               apres_preemption[i].tv_usec);

    return EXIT_SUCCESS;
}
```

Lorsque nous l'exécutons, même avec une priorité temps partagé très élevée (valeur de `nice` la plus négative possible), nous observons des perturbations dans l'exécution de notre tâche.

```
# nice -n -20 taskset -c 1 ./exemple-gettimeofday-02 15
[1] 1436866259.913814  ->  1436866259.913930
[2] 1436866259.923814  ->  1436866259.923907
[3] 1436866259.933814  ->  1436866259.933858
[4] 1436866259.943814  ->  1436866259.943855
[5] 1436866259.953814  ->  1436866259.953837
[6] 1436866259.963814  ->  1436866259.963835
[7] 1436866259.973814  ->  1436866259.973835
[8] 1436866259.983814  ->  1436866259.983838
[9] 1436866259.993814  ->  1436866259.993836
[10] 1436866260.003814  ->  1436866260.003834
[11] 1436866260.013814  ->  1436866260.013835
[12] 1436866260.023814  ->  1436866260.023839
[13] 1436866260.033814  ->  1436866260.033836
[14] 1436866260.043814  ->  1436866260.043836
[15] 1436866260.053814  ->  1436866260.053835
[16] 1436866260.063814  ->  1436866260.063835
[17] 1436866260.073814  ->  1436866260.073836
[18] 1436866260.083814  ->  1436866260.083832
[19] 1436866260.093814  ->  1436866260.093842
[20] 1436866260.103814  ->  1436866260.103836
[21] 1436866260.113814  ->  1436866260.113836
[22] 1436866260.123814  ->  1436866260.123836
[23] 1436866260.133814  ->  1436866260.133835
[24] 1436866260.143814  ->  1436866260.143835
[25] 1436866260.153814  ->  1436866260.153848
[...]
#
```

Copyright © 2015 Eyrolles.

Nous avons placé ici le seuil à 15 microsecondes. Les préemptions se présentent très régulièrement toutes les 10 millisecondes. Il s'agit bien entendu du tick du système. La mesure est faite sur un Raspberry Pi 2, dont le tick est configuré avec une fréquence de 100 Hz.

L'exécution s'est faite sur le cœur numéro 1. Réitérons en plaçant le programme sur le cœur numéro 0.

```
# nice -n 20 taskset -c 0 ./exemple-gettimeofday-02 15
[1] 1436866485.003814  ->  1436866485.003942
[2] 1436866485.006476  ->  1436866485.006501
[3] 1436866485.006603  ->  1436866485.006622
[4] 1436866485.013814  ->  1436866485.013861
[5] 1436866485.014478  ->  1436866485.014497
[6] 1436866485.014605  ->  1436866485.014621
[7] 1436866485.023814  ->  1436866485.023837
[8] 1436866485.030480  ->  1436866485.030495
[9] 1436866485.033814  ->  1436866485.033875
[10] 1436866485.038481  ->  1436866485.038497
[11] 1436866485.038608  ->  1436866485.038623
[12] 1436866485.043814  ->  1436866485.043839
[13] 1436866485.046482  ->  1436866485.046497
[14] 1436866485.046609  ->  1436866485.046624
[15] 1436866485.053814  ->  1436866485.053837
[16] 1436866485.054483  ->  1436866485.054498
[17] 1436866485.063815  ->  1436866485.063839
[18] 1436866485.070484  ->  1436866485.070499
[19] 1436866485.073814  ->  1436866485.073840
[20] 1436866485.078486  ->  1436866485.078501
[21] 1436866485.083814  ->  1436866485.083837
[22] 1436866485.086487  ->  1436866485.086502
[23] 1436866485.093814  ->  1436866485.093837
[24] 1436866485.094487  ->  1436866485.094503
[25] 1436866485.103814  ->  1436866485.103837
[...]
```

Nous voyons alors de très nombreuses préemptions de notre tâche. En effet, par défaut, l'essentiel des interruptions (réseau, carte SD, SPI, GPIO...) est traité par le Raspberry Pi 2 sur son cœur 0.

Conclusion

Dans ce chapitre, nous avons examiné les limites des ordonnancements temps partagé : très bien adaptés pour organiser l'exécution des tâches courantes sur un poste de travail ou sur un serveur réseau, ils ne permettent pas un contrôle fin des temporisations, ni une exécution ininterrompue des tâches, aussi importantes soient-elles.

Pour améliorer les réponses temporelles de nos processus, il nous faudra donc nous tourner vers les mécanismes temps réel.

Copyright © 2015 Eyrolles.

Points clés

- Le noyau Linux actuel nous permet de mesurer l'heure avec une précision de l'ordre de la microseconde (ceci dépend beaucoup du matériel).
- Avec une compilation adéquate (options High-Resolution Timers et Tickless System), nous pouvons obtenir des timers logiciels dont la période se mesure en dizaine ou centaines de microsecondes.
- L'ordonnancement temps partagé induit une fluctuation *(jitter)* des timers et des préemptions inévitables des tâches.

Exercices

Exercice 1 (*)

Utilisez le programme `exemple-gettimeofday.c` pour vérifier la précision de la mesure horaire sur votre ordinateur. Combien de temps dure l'appel système ?

L'utilisation de la commande `nice` pour modifier la priorité temps partagé influe-t-elle sur les résultats ?

Exercice 2 (**)

Utilisez le programme `exemple-timer-create.c` pour créer un timer à la milliseconde. Observez les résultats fournis par le programme, puis envoyez-les dans le programme `calculer-statistiques.c` pour analyser le comportement du timer.

Exercice 3 (**)

Utilisez les programmes `exemple-timer-create.c`, `calculer-histogramme.c` et le script `afficher-histogramme.sh` pour visualiser graphiquement les fluctuations du timer.

Jusqu'à quelle période pouvez-vous faire descendre le timer en conservant des résultats acceptables (par exemple, un écart-type inférieur au dixième de la période) ?

Exercice 4 (***)

Vérifiez le comportement du timer programmé dans l'exemple précédent en présence d'un programme parasite, comme `element-perturbateur.c` ou en présence d'un grand nombre d'interruptions (ceci peut s'obtenir avec la commande `hdparm` ou en utilisant un `ping` *flood* depuis un autre poste).

Copyright © 2015 Eyrolles.

Exercice 5 (***)

Utilisez le programme `exemple-gettimeofday-02` pour détecter les préemptions supérieures à 1 milliseconde.

S'en produit-t-il si vous laissez le programme seul sur un système au repos ? Si vous utilisez le système (mouvement de fenêtres graphiques, exécution de commandes shell...) pendant son exécution ? Si vous lancez sur le même processeur une application gourmande en CPU au démarrage (Eclipse, Open Office.org, Firefox, etc.) ?

Copyright © 2015 Eyrolles.

5

Principes du temps réel

Ayant observé jusqu'à présent les limites de l'ordonnancement temps partagé, nous pouvons être amenés à vouloir améliorer la fiabilité de nos tâches, et notamment leurs temps de réponse face à un événement. Nous allons pour cela examiner dans ce chapitre et les suivants, les possibilités de traitement temps réel offertes par le noyau Linux classique (sans extension particulière).

Définitions

Temps réel

Il existe de nombreuses définitions concernant les systèmes temps réel, certaines s'appuient sur la prédictibilité du temps d'exécution d'une tâche, d'autres sur les priorités et les mécanismes d'ordonnancement, d'autres encore mettent l'accent sur la précision des caractéristiques temporelles.

Je retiendrai une définition qui convient à l'idée que je me fais d'un système temps réel appliqué :

« Un système informatique est soumis à des contraintes temps réel si l'instant auquel il parvient au terme d'une opération entre en considération dans la validité du résultat de cette opération. »

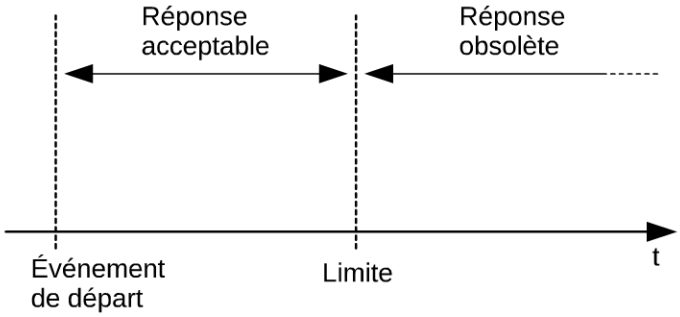
En pratique, on considère qu'une opération est initialisée par un événement et que l'on dispose d'un certain délai pour la terminer. L'événement de départ peut être interne (déclenchement d'un timer, détection d'une valeur spécifique d'un paramètre, message provenant d'une autre tâche...) ou externe (interruption matérielle, changement d'état sur un capteur...). Si le système effectue

Copyright © 2015 Eyrolles.

le traitement avant le délai qui lui était imparti, le résultat de l'opération est correct, sinon il est considéré comme invalide.

Figure 5-1

Contrainte temporelle

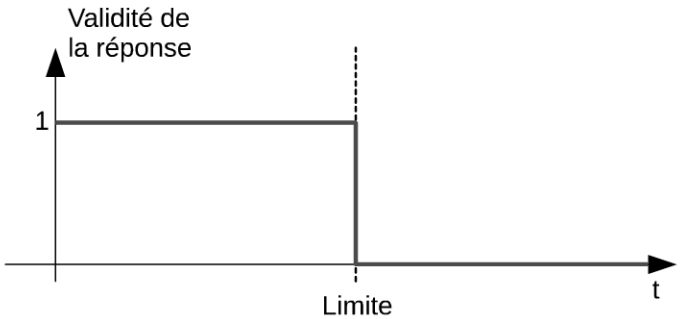


Ceci est représenté sur la figure 5-1 où se trouvent placés sur l'axe temporel l'événement de départ (timer, interruption, signal, etc.) et la limite au-delà de laquelle la réponse du système n'est plus acceptable.

Nous pouvons représenter une courbe de validité de la réponse : entre l'événement déclencheur et la limite, la réponse sera valide (si le traitement est correct bien sûr), après elle ne le sera plus. Un exemple de cette courbe est représenté à la figure 5-2.

Figure 5-2

Validité en temps réel strict



Classes de temps réel

On caractérise souvent les systèmes temps réel en s'appuyant sur deux catégories, deux classes : ceux capables de répondre à des contraintes strictes (temps réel dur – *hard realtime*) et ceux ne pouvant répondre qu'à des contraintes légèrement relâchées (temps réel souple ou mou – *soft realtime*). J'ai personnellement tendance à élargir un peu cette classification en employant quatre catégories : temps réel « absolu », « strict certifiable », « strict non certifiable » et « souple ».

Temps réel absolu

Dans cette catégorie, je classe les systèmes où la durée d'un traitement – y compris la réaction à une sollicitation externe – est parfaitement connue (avec une précision de l'ordre de la dizaine de

Copyright © 2015 Eyrolles.

nanosecondes, par exemple) et ne varie pas en fonction de l'activité du système. Les fluctuations infimes pourraient être dues aux dérives thermiques des caractéristiques des composants.

Il s'agit essentiellement de systèmes purement électroniques (FPGA, PAL, PLD…), sans intervention de programmation logicielle. Ce type de temps réel ne peut pas être atteint avec un microprocesseur.

Temps réel strict

À la différence de la classe précédente, les systèmes temps réel stricts *(hard realtime)* ne garantissent pas une durée exacte de réaction, mais fournissent une borne supérieure – une limite – pour cette durée.

La courbe de la figure 5-2 est révélatrice de contraintes temps réel strictes. Une fois la limite atteinte, la validité de la réponse s'effondre, entraînant un comportement catastrophique (au sens scientifique du terme) quant à ses conséquences.

Généralement, la limite et les contraintes qu'elle suppose sont impliquées par le monde extérieur (axe motorisé arrivant en butée, seuil haut de remplissage d'un réservoir, créneau d'émission d'un message radio, etc.).

Contrairement à une idée souvent exprimée, mais erronée, le temps réel n'englobe pas la notion de vitesse, mais plutôt de prédictibilité. Certains systèmes temps réel peuvent impliquer des calculs ou des traitements longs (des minutes, voire des heures), mais la contrainte externe s'exprime sous forme d'une heure absolue à ne pas dépasser (fin d'un concours de calcul, clôture de Wall Street, etc.)

Naturellement, dans la plupart des cas où s'expriment de véritables contraintes temps réel strictes, le système doit répondre dans des délais brefs, comptés en micro ou millisecondes.

Temps réel strict certifiable

Un enjeu essentiel des systèmes temps réel stricts est de pouvoir piloter des systèmes critiques, pour lesquels il est indispensable de pouvoir *prouver* que le programme ne dépassera jamais la limite garantie, quels que soient la charge du système, le nombre d'interruptions survenues et le traitement à réaliser.

Pour cela, il faut non seulement détailler toutes les opérations que l'on effectue en réponse à l'événement survenu – en vérifiant les limites maximales des boucles, les portions les plus longues des tests, en additionnant les nombres de cycles par instructions assembleurs exécutées –, mais il faut également que le système d'exploitation soit capable de garantir qu'il laissera notre traitement sans l'interrompre – ou du moins en donnant une limite maximale au nombre et à la durée des interruptions ; ceci en vue d'obtenir des certifications de sécurité (domaines ferroviaire, aéronautique, automobile, médical…). Ce n'est qu'à ce prix qu'un système temps réel strict pourra piloter un dispositif responsable de vies humaines.

Disons-le clairement : Linux n'est – à l'heure actuelle – pas capable de répondre aux critères des systèmes temps réel stricts certifiables. Le noyau Linux seul contient environ dix-neuf millions de lignes de code, sans compter les bibliothèques et utilitaires indispensables de l'espace

Copyright © 2015 Eyrolles.

utilisateur. Personne n'est prêt à parcourir tous les chemins logiques possibles de ces millions de lignes de code pour s'assurer qu'une limite temporelle ne sera jamais dépassée.

Les systèmes capables de répondre aux contraintes du temps réel strict certifiable sont généralement construits autour de microcontrôleurs ; ils sont souvent monotâches ou ne comportent qu'un nombre très limité (et figé) de tâches.

La possibilité de réaliser des systèmes temps réel stricts certifiables en employant des microprocesseurs modernes est même sujet à controverse. De nombreux sous-systèmes de ces microprocesseurs limitent la prévisibilité des temps de réponse : MMU, caches mémoire, pipelines d'instruction, protocoles de synchronisation des caches pour les processeurs multicœurs, réduction de consommation d'énergie, etc.

Si l'on souhaite développer une application temps réel stricte certifiable en utilisant un système d'exploitation libre, les solutions seront à rechercher vers des systèmes comme FreeRTOS (ordonnanceur minimal) ou RTEMS *(Real Time Executive for Multiprocessor Systems)* implémentés sur des processeurs simples ou des microcontrôleurs.

Temps réel strict non certifiable

Les systèmes temps réel stricts non certifiables ont pour vocation de répondre à des limites identiques aux systèmes précédents. Néanmoins, leur complexité interne ne permet pas de *prouver formellement* qu'ils répondront toujours aux contraintes imposées.

En pratique, on peut parfaitement utiliser ces systèmes dès que le dépassement éventuel d'une limite temporelle n'implique pas de conséquence dramatique (notamment sur une vie humaine).

Dans ces circonstances, on se contente de vérifier que le système répond aux exigences du cahier des charges, sans toutefois pouvoir le prouver formellement. Il s'agit d'observer le système pendant un temps suffisamment long, avec des charges système élevées – nous reparlerons de ce thème – et de s'assurer qu'aucun dépassement n'est survenu.

Le noyau Linux classique seul n'est pas capable d'offrir ce type de garantie, néanmoins l'adjonction d'une extension comme Xenomai, que nous étudierons plus loin, lui permet de rentrer dans cette catégorie.

Temps réel souple

Dans les systèmes temps réel souples[1] *(soft realtime)*, on n'impose pas un comportement valable dans le pire des cas *(worst case)*, comme précédemment, mais simplement que le système consacre toute son énergie *(best effort)* à réaliser le travail attendu.

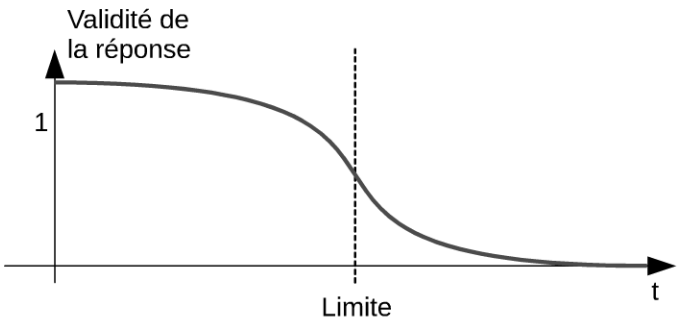
La courbe précédente de validité du résultat peut être quelque peu lissée (figure 5-3).

1. Certains préfèrent parler de « temps réel mou » au lieu de « temps réel souple ». Je trouve personnellement l'adjectif « mou » peu adapté à la notion de temps réel, malgré son clin d'œil à Salvador Dalí.

Copyright © 2015 Eyrolles.

Figure 5-3

Validité en temps réel souple



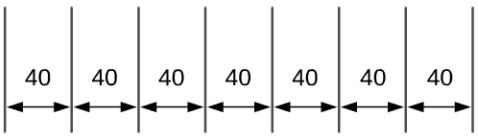
Cette fois, la validité de résultat est d'autant meilleure qu'elle est rapide. Une réponse quasi instantanée est considérée comme très bonne. Plus on se rapproche de la limite, moins la réponse est acceptable. Toutefois, même après avoir dépassé notre limite, on attribuera une « faible validité » à notre réponse.

Comment peut-on attribuer une valeur à la validité d'une réponse qui ne soit pas « vraie » ou « fausse » ? Simplement, on s'intéressera à des conditions d'exécution moyennes, et non au pire des cas. Prenons l'exemple d'un système devant diffuser des trames vidéo, par exemple pour une application de vidéoconférence. Dans notre cas, l'événement déclencheur est un timer fourni par le système et le logiciel répond en émettant son image dans un délai limité.

Nous souhaiterions que notre programme diffuse les trames avec une fréquence de 25 images/seconde, soit 40 ms entre chaque émission, comme sur la figure 5-4.

Figure 5-4

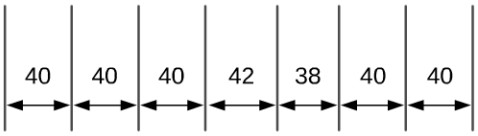
Système non perturbé



Supposons que suite à une perturbation relativement rare (coïncidence d'interruptions multiples), une image soit légèrement retardée, à 42 ms de la précédente au lieu de 40 ms (figure 5-5).

Figure 5-5

Temps réel souple perturbé



Dans un environnement temps réel souple, comme Linux, le système corrigera l'écart à l'image suivante, quitte à l'avancer par rapport à celle en retard. Ainsi, le nombre moyen d'images par seconde restera constant à 25.

Nous nous intéressons dans ce cas au comportement moyen du système et non au comportement individuel de chaque événement.

Une chose importante également : si un système est considéré comme soumis à des contraintes temps réel souples, le concepteur n'est plus obligé de *prouver* son bon comportement quelles que soient les circonstances, mais il peut le *vérifier*. Il construit une maquette du système, la

Copyright © 2015 Eyrolles.

chargeant au maximum, tant au niveau des applications et services logiciels qu'au niveau des interruptions matérielles et trafics de données (réseau, liaison série…) et vérifie sa bonne tenue dans ces conditions extrêmes.

Rôles respectifs

La limite entre les domaines d'application des systèmes temps réel souples et stricts (non certifiables en ce qui concerne Linux) n'est pas facile à définir. Il s'agit avant tout des contraintes imposées par l'environnement physique avec lequel le logiciel interagit et des tolérances aux variations temporelles.

Si le temps de réponse demandé est à l'échelle humaine (allumage d'un voyant d'acquittement suite à une pression sur un bouton), il ne fait nul doute qu'un système temps réel souple – voire temps partagé dont les tâches ont des valeurs de priorité élevées – aura une réactivité largement suffisante. Dans ce cas, les limites temporelles seront de l'ordre de la dizaine de millisecondes, avec une tolérance de plusieurs millisecondes.

Lorsque le système doit superviser ou piloter des dispositifs électromécaniques externes, les contraintes sont généralement beaucoup plus fortes. Même si les délais de réaction sont parfois relativement longs (plusieurs millisecondes ou dizaines de millisecondes), les limites doivent être impérativement respectées (imaginons le cas d'un moteur devant être arrêté avant qu'un axe arrive en butée).

Naturellement, il est souvent possible d'utiliser un système offrant des possibilités de temps réel souples, comme Linux, même lorsque le problème est clairement relatif aux concepts du temps réel strict si le délai limite de terminaison d'une tâche est très nettement supérieur au plus long délai de réponse observé. Toutefois, il convient d'être bien conscient que dans ce cas, nous n'avons jamais de garantie absolue du respect de la limite. Il faut évaluer les conséquences d'un éventuel retard d'exécution très rare (et très improbable) et en accepter le prix.

Traitement direct dans le noyau

J'ai remarqué une question récurrente parmi mes clients et les participants à mes sessions de formation traitant du temps réel : « Et si nous faisions tout le traitement directement dans le kernel Linux, les temps de réponse seraient-ils plus prédictibles ? » Je formule habituellement une réponse un peu ambiguë comme « Oui peut-être, mais ce ne serait probablement pas une bonne solution ».

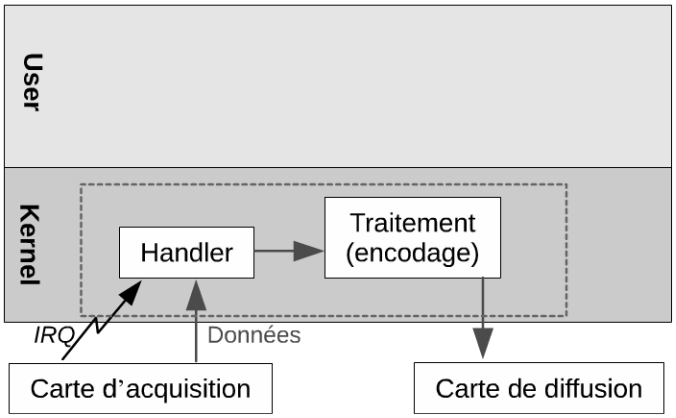
L'idée dissimulée derrière cette question est généralement d'écrire un driver spécifique avec une sorte d'énorme gestionnaire d'interruptions qui ferait tout le traitement, avec un comportement le plus déterministe possible. Je déconseille cette approche car, non seulement elle ne s'insère pas bien dans la philosophie du système Linux, mais elle risque également de pénaliser au final certains composants du noyau diminuant la fiabilité globale du système.

Prenons un exemple assez courant : une carte d'acquisition graphique fournit un flux d'images vidéo que l'on souhaite encoder et envoyer vers un périphérique de distribution spécifique.

Copyright © 2015 Eyrolles.

Figure 5-6

Traitement dans le noyau



Dans la première approche, schématisée sur la figure 5-6, la carte d'acquisition déclenche une interruption sur réception d'une trame de données (image vidéo), qui est lue par le handler de l'interruption, traitée par un algorithme d'encodage, et transmise dans le buffer de sortie d'une carte de diffusion.

Ceci nécessite d'écrire deux drivers ou un seul gros driver encadrant tout le traitement, et présente plusieurs inconvénients.

- Le développement dans le kernel Linux n'est pas plus compliqué que dans l'espace utilisateur, mais il convient de faire attention à de nombreux paramètres (protection des variables globales, réentrance des appels système, etc.) qui compliquent rapidement le projet. En outre, la mise au point et la validation du code sont rendues beaucoup plus complexes par l'absence d'outils de débogage simples et le manque total de protection (isolation mémoire, accès aux périphériques, etc.).
- Le noyau Linux est sous licence GPL. Lorsqu'on fournit un système à l'utilisateur, il doit obligatoirement disposer des sources du kernel, ainsi que des éventuelles modifications qui y ont été apportées. Or, dans le monde industriel, il est rare d'accepter de fournir les sources des traitements centraux (par exemple, encodage vidéo) qui représentent une part importante du travail de recherche et développement.

Il est possible d'insérer du code propriétaire dans le noyau Linux par l'intermédiaire des modules (fichiers `*.ko` *(kernel object)* que l'on charge dynamiquement avec la commande `insmod`), mais il existe quand même des restrictions de licences, et le module ne peut être utilisé qu'avec le noyau exact pour lequel il a été compilé, ce qui limite les possibilités de diffusion du fichier binaire.

- S'il est possible, dans le noyau, d'assurer le déroulement ininterrompu d'une portion de code (en coupant les interruptions, par exemple, et en interdisant temporairement la préemption), ceci va à l'encontre des préceptes d'écriture des drivers et rend la maintenance difficile au gré des évolutions de Linux.
- Enfin, ce principe permet de rendre une opération plus prioritaire que tout le système, mais ne permet pas de programmer plusieurs tâches en profitant des mécanismes de priorité offerts par Linux.

Copyright © 2015 Eyrolles.

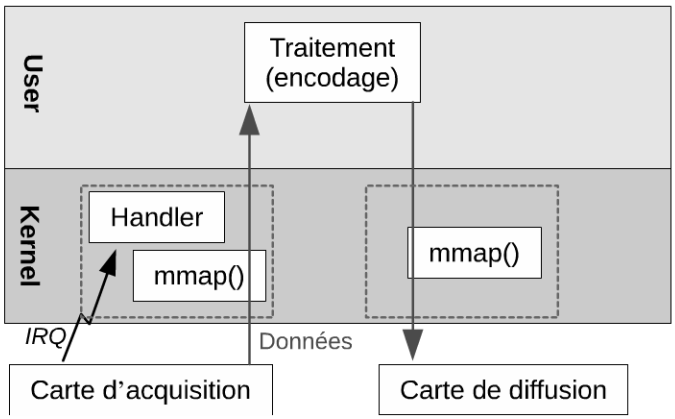
Pour ce genre de projet, je conseillerai plutôt une approche légèrement différente, représentée sur la figure 5-7.

Un premier driver très simple gère les accès à la carte d'acquisition et implémente quelques appels système, dont la méthode `mmap()`. Ce genre de driver est facile à écrire, de nombreux exemples existent – nous en verrons un dans le chapitre 11 – et il ne contient aucun aspect « intelligent » ou « secret », ce qui permet de le placer sous licence GPL sans problème.

Un second driver – tout aussi simple – gère la carte de diffusion en implémentant, entre autres, l'appel système `mmap()`.

Figure 5-7

Traitement en espace utilisateur



Un processus, fonctionnant en espace utilisateur, va demander au premier driver de lui faire une projection directe de la mémoire de la carte d'acquisition dans son espace d'adressage grâce à `mmap()`. Symétriquement, il demandera une projection dans un autre emplacement de son espace d'adressage de la mémoire de la carte de diffusion. Ensuite, le programme effectuera en boucle les traitements, se mettant en sommeil sur un appel système bloquant comme `select()` jusqu'à ce qu'une image soit disponible dans sa mémoire, puis il assurera l'encodage en écrivant dans la mémoire du second driver, et préviendra ce dernier grâce à un appel comme `ioctl()`.

Les avantages de ce système sont les suivants.

- Les codes sources des drivers sont simples et peuvent être diffusés sans aucune limitation. Ils sont construit sur des exemples disponibles tant dans les sources du kernel Linux que sur des sites Internet de développeurs.
- Le code contenant l'algorithme d'encodage se trouve dans l'espace utilisateur où aucune restriction de licence ne s'applique. Il peut donc se trouver sous une licence propriétaire sans aucune divulgation de code.
- Le code principal du programme se situant dans l'espace utilisateur, sa mise au point est facilitée par les nombreux outils disponibles. De plus, les limitations encadrant le déroulement des processus protègent le système et les autres applications de toute erreur de codage.
- Le processus s'exécutant sous le contrôle de l'ordonnanceur, il sera possible de lui donner une priorité (temps partagé ou temps réel) qui garantira son interaction avec les autres tâches.

Copyright © 2015 Eyrolles.

L'intérêt d'utiliser Linux comme système d'exploitation temps réel plutôt qu'un système spécialisé comme VxWorks, VRTX, etc., réside à mon avis dans les quatre points suivants.

- La gratuité et la liberté du système, liées à sa licence GPL : il est important à mon sens de respecter l'esprit Open Source, et de ne placer dans le kernel Linux que du code dont on est prêt à fournir les sources aux utilisateurs.
- La richesse de l'API, qui offre des mécanismes de communication et de synchronisation entre threads (mutex, variables conditions...), entre processus (files de messages, mémoire partagée, sémaphores, *pipes*...) et même entre systèmes distants (sockets TCP/IP ou UDP/IP). Pour profiter de cette interface de programmation, il est préférable de l'aborder depuis l'espace utilisateur.
- La protection offerte par la mémoire virtuelle, et plus particulièrement par le système de pagination s'appuyant sur la MMU *(Memory Managment Unit)* du processeur. Seuls les processus utilisateurs ont cette garantie d'étanchéité absolue de leur espace mémoire.
- Enfin, l'environnement GNU/Linux lui-même s'avère particulièrement riche et complet, offrant de nombreux serveurs (FTP, HTTP, SSH, SNMP) très utiles dans des environnements industriels ou scientifiques, ainsi qu'un nombre élevé de bibliothèques et de langages de programmation.

Pour ces raisons, je conseille systématiquement de développer le cœur des applications temps réel dans l'espace utilisateur, et de n'ajouter au noyau que le strict minimum nécessaire pour gérer le matériel spécifique. Si une tâche doit être obligatoirement réalisée avec les privilèges du noyau (pour la gestion des entrées-sorties physiques ou des interruptions), on peut se tourner vers les kernel threads, qui présentent l'avantage d'être ordonnancés et de pouvoir ainsi bénéficier de priorités temps réel.

Traitement des interruptions

Il est important de noter qu'il n'est pas possible depuis l'espace utilisateur de masquer ou démasquer les interruptions, ni de les traiter directement en installant un handler.

Si le travail à réaliser lors de l'occurrence d'une interruption est court et doit être effectué très rapidement, on peut concevoir d'écrire un gestionnaire spécifique, que l'on installera dans le noyau à la compilation, ou sous forme de module chargé dynamiquement.

Il est aussi possible d'écrire un petit driver minimal qui permettra de réveiller un processus utilisateur (ordonnancé en temps réel ou non) en attente sur l'interruption. Nous en verrons un exemple ultérieurement.

Copyright © 2015 Eyrolles.

Temps réel sous Linux

Échelle des priorités

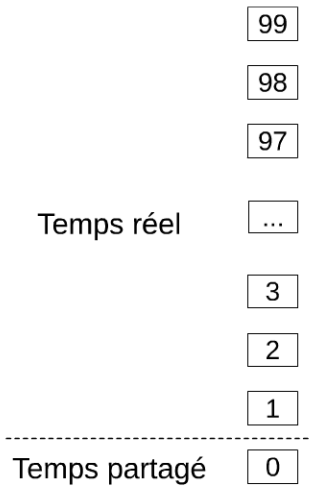
Les tâches temps réel disposent d'une échelle des priorités spécifique, totalement indépendante des tâches temps partagé. Sous Linux, les priorités temps réel se trouvent dans l'intervalle [1, 99]. Une tâche évoluant avec la priorité 99 est donc la plus prioritaire.

On peut considérer – avec un léger abus de langage – que la priorité temps réel 0 correspond à l'ordonnancement temps partagé, gérant lui-même ses propres priorités à travers la valeur de courtoisie *(nice)*. On peut observer la frontière entre ces deux classes d'ordonnancement sur la figure 5-8.

Il est important de bien comprendre que l'ordonnancement (temps réel ou temps partagé) se choisit tâche par tâche. Par conséquent, le choix peut s'effectuer au niveau d'un processus, ou même d'un thread dans un processus.

Figure 5-8

Échelle des priorités



Le principe général est qu'une tâche d'une priorité donnée ne pourra jamais être préemptée ou laissée en attente *(Runnable)* tandis qu'une tâche de priorité moindre dispose du processeur.

Une tâche de priorité 50 ne laissera jamais s'exécuter une tâche de priorité 49 et encore moins une tâche temps partagé.

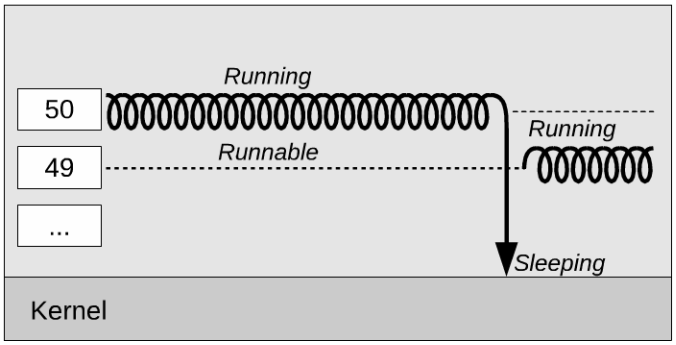
> Attention, cette dernière affirmation n'est pas tout à fait exacte, nous en reparlerons plus loin.

Pour que la tâche de priorité 49 puisse s'exécuter, il faudra que celle de priorité 50 s'endorme volontairement (ou se termine) comme représenté sur la figure 5-9.

Copyright © 2015 Eyrolles.

Figure 5-9

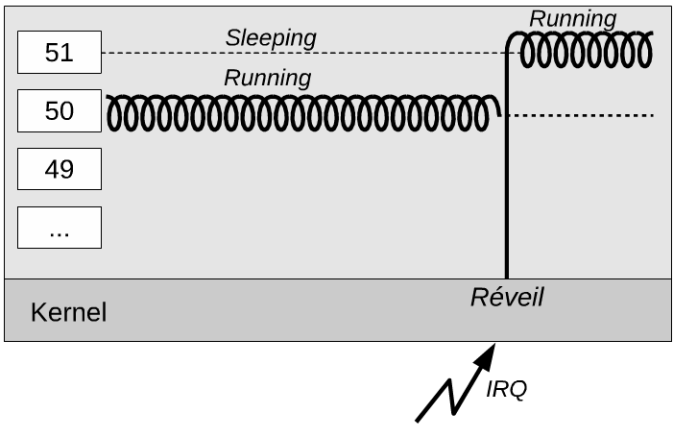
Cession de CPU en temps réel



En revanche, notre tâche de priorité 50 sera immédiatement préemptée si une tâche de priorité 51 se réveille (suite à une interruption ou au déclenchement d'un timer, par exemple), comme nous pouvons le voir sur la figure 5-10. Il lui faudra attendre l'endormissement ou la terminaison de la tâche 51 pour reprendre son exécution.

Figure 5-10

Préemption en temps réel



Il subsiste malgré tout une interrogation lorsque deux tâches se trouvent sur le même niveau de priorité. Pour préciser le comportement, il existe deux types d'ordonnancements temps réel normalisés par la norme Posix.

- Fifo (*First In, First Out*, premier arrivé, premier servi) : une tâche sous ordonnancement Fifo ne peut être préemptée que par une tâche *de priorité strictement supérieure* qui devient prête.
- Round Robin (tourniquet) : lorsqu'une tâche sous ordonnancement Round Robin est activée, on lui accorde un délai au bout duquel elle sera préemptée pour laisser passer les autres tâches *de même priorité* en attente. Si aucune tâche n'est prête, la tâche initiale reprend son exécution normalement.

Répétons-le : même en Round Robin, une tâche ne laissera jamais s'exécuter une tâche de priorité inférieure.

Copyright © 2015 Eyrolles.

Configuration de l'ordonnancement

Pour fixer ou consulter le type d'ordonnancement d'un processus, nous utilisons les appels système :

```
#include <sched.h>

int sched_setscheduler (pid_t cible,
                int ordonnancement,
                const struct sched_param * parametre);

int sched_getscheduler (pid_t cible);

int sched_setparam (pid_t cible,
                const struct sched_param *parametre);

int sched_getparam (pid_t cible,
                    struct sched_param * parametre);
```

Le premier argument est l'identifiant du processus concerné. En effet, un processus peut tout à fait modifier l'ordonnancement d'un autre processus. Si l'on indique `0` en premier argument, c'est au contraire son propre ordonnancement que l'on modifie.

Le second argument de `sched_setscheduler()` ou le retour de `sched_getscheduler()` représente le type d'ordonnancement parmi les trois valeurs normalisées par Posix :

Constante	Signification
`SCHED_FIFO`	Ordonnancement temps réel Fifo. La tâche ne peut être préemptée que par une tâche de priorité strictement supérieure.
`SCHED_RR`	Ordonnancement temps réel Round Robin. La tâche est interrompue régulièrement pour laisser s'exécuter les autres tâches de même priorité.
`SCHED_OTHER`	Ordonnancement temps partagé. Le comportement n'est pas normalisé par Posix. Sous Linux 3.0, il s'agit de l'ordonnancement *CFS*.

Il existe d'autres types d'ordonnancements temps partagé (`SCHED_BATCH`, `SCHED_IDLE`) spécifiques à Linux et dont le comportement n'est pas constant selon les versions du kernel.

Le troisième argument de `sched_setscheduler()` ou le deuxième de `sched_getparam()` et `sched_setparam()` est un pointeur sur une structure de paramètres. Cette structure ne contient qu'un seul champ normalisé.

Champ	Signification
`sched_priority`	La priorité temps réel dans l'intervalle [1,99] pour les ordonnancements Fifo et Round Robin, et valant 0 pour l'ordonnancement temps partagé Other.

Copyright © 2015 Eyrolles.

Lorsque nous fixerons un nouvel ordonnancement, avec `sched_setscheduler()`, nous nous contenterons de remplir ce champ `sched_priority`. Sous Linux, il s'agit bien de l'unique champ de la structure, toutefois il est possible que d'autres systèmes ajoutent leurs propres paramètres (on peut imaginer, par exemple, une notion de *deadline* pour d'autres ordonnancements). Aussi, pour assurer la portabilité d'un programme, prendra-t-on soin, lorsque l'on voudra modifier la priorité, de commencer par lire la structure avec `sched_getparam()`, de modifier le champ `sched_priority`, puis de réécrire la structure avec `sched_setparam()`.

J'ai indiqué que les valeurs limites pour les priorités temps réel étaient dans l'intervalle [1, 99]. Ceci est spécifique à Linux, et un programme se voulant portable devrait déterminer correctement l'échelle des priorités en invoquant :

```
int sched_get_priority_max (int ordonnancement);
int sched_get_priority_min (int ordonnancement);
```

Voici une vérification des valeurs indiquées précédemment :

```
exemple-sched-get-priorities.c :
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("SCHED_FIFO :  [%d - %d]\n",
           sched_get_priority_min(SCHED_FIFO),
           sched_get_priority_max(SCHED_FIFO));
    printf("SCHED_RR :    [%d - %d]\n",
           sched_get_priority_min(SCHED_RR),
           sched_get_priority_max(SCHED_RR));
    printf("SCHED_OTHER : [%d - %d]\n",
           sched_get_priority_min(SCHED_OTHER),
           sched_get_priority_max(SCHED_OTHER));
    return EXIT_SUCCESS;
}
```

Les valeurs attendues :

```
$ ./exemple-sched-get-priorities
SCHED_FIFO :  [1 - 99]
SCHED_RR :    [1 - 99]
SCHED_OTHER : [0 - 0]
$
```

Il est important de savoir que pour soumettre un processus à un ordonnancement temps réel, avec `sched_setscheduler()`, l'appelant doit disposer des droits *root*. En effet, cette opération est potentiellement dangereuse, puisque la tâche temps réel peut se mettre à dévorer tout le temps CPU disponible et créer ainsi un *déni de service* sur le système.

Copyright © 2015 Eyrolles.

Processus temps réel

Nous allons à présent en étudier un exemple. Pour cela, nous allons créer un processus père qui va lancer autant de processus fils qu'il y a de CPU disponibles sur le système. Chacun d'entre eux fixera son affinité (comme nous l'avons vu au chapitre 1) afin d'occuper tous les processeurs. Ensuite, ils passeront en temps réel Fifo de priorité 99, avant d'attaquer une boucle active infinie :

```
while (1)
     ;
```

Bien sûr, si nous nous lançons directement dans cette boucle, tous les CPU seront saturés et nous ne pourrons pas reprendre la main sur notre machine. Notre seul recours sera l'interrupteur Marche/Arrêt. Pour éviter ceci, nous allons programmer une alarme de 15 secondes avant de rentrer dans la boucle.

Au bout des 15 secondes, le noyau enverra automatiquement un signal `SIGALRM` à chaque processus qui a programmé l'alarme. Comme nous ne faisons rien pour gérer ce signal, les processus en boucle seront tués, et nous reprendrons le contrôle de notre système. Voici une implémentation :

exemple-boucle-temps-reel.c :

```
#define _GNU_SOURCE  // Pour avoir sched_setaffinity
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void fonction_fils (int cpu);

int main(void)
{
    int nb_cpus = sysconf(_SC_NPROCESSORS_ONLN);
    int cpu;
    for (cpu = 0; cpu < nb_cpus; cpu ++) {
        if (fork() == 0) {
            fonction_fils(cpu);
        }
    }
    /* Attendre la fin des fils */
    for (cpu = 0; cpu < nb_cpus; cpu ++)
        waitpid(-1, NULL, 0);
    return EXIT_SUCCESS;
}


void fonction_fils (int cpu)
{
    struct sched_param param;
    cpu_set_t  cpuset;
```

Copyright © 2015 Eyrolles.

```
    /* Se placer sur le CPU indiqué */
    CPU_ZERO(& cpuset);
    CPU_SET(cpu, &cpuset);
    if (sched_setaffinity(0, sizeof(cpuset),
                          & cpuset) !=0) {
        perror("sched_setaffinity");
        exit(EXIT_FAILURE);
    }

    /* Passer en temps réel Fifo 99 */
    param.sched_priority = 99;
    if (sched_setscheduler(0, SCHED_FIFO, &param) != 0){
        perror("sched_setscheduler");
        exit(EXIT_FAILURE);
    }
    /* Dormir une seconde pour être sûr que les autres
       fils sont sur leurs CPU respectifs */
    sleep(1);
    /* Programmer une alarme dans 15 secondes */
    alarm(15);
    /* Boucler activement */
    while (1)
        ;
}
```

Avant de lancer ce programme, je dois vous avertir que certains environnements graphiques n'aiment pas être privés de temps CPU pendant plusieurs secondes et peuvent se comporter bizarrement (envoyer des rafales de retours chariot dans les terminaux texte, par exemple). De même, j'ai constaté des déconnexions d'adaptateur Wi-Fi si le temps dépasse 20 à 30 secondes. Je vous conseille donc de tester le programme sur un poste de travail indépendant, que vous pourrez réinitialiser sans gêner d'autres utilisateurs.

Un bon réflexe à acquérir lorsqu'on se prépare à démarrer un programme d'expérimentation en temps réel est de lancer auparavant depuis le shell la commande `sync`. Celle-ci s'assurera que tous les contenus des buffers de cache disque en attente ont été transmis aux pilotes de périphériques. On limite ainsi la perte de données en cas de nécessité de réinitialisation abrupte.

Allons-y !

```
$ sync
$ ./exemple-boucle-temps-reel
sched_setscheduler: Operation not permitted
sched_setscheduler: Operation not permitted
sched_setscheduler: Operation not permitted
sched_setscheduler: Operation not permitted
$
```

Copyright © 2015 Eyrolles.

Et oui, le passage en temps réel échoue si nous n'avons pas les droits *root*.

```
$ su
Password:
# ./exemple-boucle-temps-reel
#
```

Après une période de 15 secondes (qui paraît beaucoup plus longue la première fois, c'est normal...) pendant laquelle tout le système était entièrement gelé, nous reprenons normalement la main sur notre système.

Mais, est-ce bien vrai ? Le système est-il effectivement gelé pendant 15 secondes ? Recommencez plusieurs fois l'expérience et vous pourrez remarquer que le pointeur de la souris arrive à se déplacer (de façon très saccadée), que le curseur du shell continue à clignoter et que l'horloge graphique avance (en sautant parfois quelques secondes).

Depuis de nombreuses années, je réalise cette petite expérience lors de sessions de formation sur Linux temps réel. Imaginez ma surprise, au printemps 2008, en lançant un tel programme, de voir les applications tournant en ordonnancement temps partagé (comme l'utilitaire *GkrellM* dont nous avons parlé au chapitre 1) continuer à s'exécuter – ralenties, il est vrai – malgré les boucles temps réel Fifo de priorités très élevées.

Garde-fou temps réel

Depuis la version 2.6.25 de Linux, sortie en avril 2008, une petite partie (5 %) du temps CPU disponible est réservée aux tâches temps partagé, même lorsque des tâches temps réel s'exécutent !

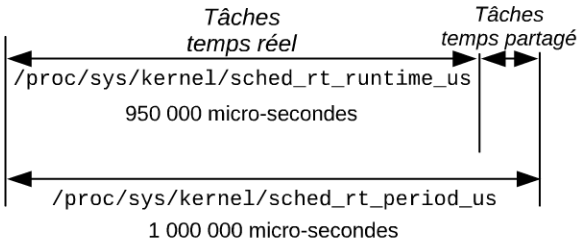
Ce comportement – qui avait été introduit sans grande publicité dans le nouveau noyau – viole bien entendu les principes mêmes des systèmes temps réel. Il est conçu comme une sorte de garde-fou qui protège l'usager inconscient contre les boucles infinies qu'il aurait malencontreusement lancées sous un ordonnancement temps réel. Bien sûr, cela peut s'avérer très utile lors de la mise au point d'un système temps réel ou pour son débogage. Toutefois, j'estime que c'est une option qui ne devrait pas être active par défaut. Les noyaux Linux depuis le 2.6.25 adoptent une attitude ultra prudente, préférant, par défaut, pénaliser le comportement temps réel plutôt que de risquer un déni de service par une boucle infinie.

Heureusement, il est possible de débrayer ce garde-fou, en écrivant dans un pseudofichier de `/proc` – même si l'on peut regretter qu'il n'y ait pas une option pour désactiver ce comportement par défaut lors de la compilation du noyau. Le fichier `/proc/sys/kernel/sched_rt_period_us` représente une période, exprimée en microsecondes, que partagent les tâches temps réel et les tâches temps partagé. Son contenu par défaut est `1.000.000`, ce qui correspond à 1 seconde. Dans cette période, on réserve aux tâches temps réel la durée indiquée dans `/proc/sys/kernel/sched_rt_runtime_us`. Par défaut, cette durée vaut `950.000`. Ainsi, chaque seconde, on réserve 50 millisecondes pour les tâches temps partagé, et les tâches temps réel peuvent consommer jusqu'à 950 millisecondes.

Copyright © 2015 Eyrolles.

Figure 5-11

Garde-fou du noyau Linux



Afin de désactiver ce comportement, il suffit d'écrire la valeur -1 dans le pseudo-fichier `sched_rt_runtime_us`. Je conseille de réaliser ceci dès le *boot* du système, dans un script de démarrage par exemple. On peut réaliser cette opération aisément à l'aide de la commande `sysctl` qui fonctionne également lorsqu'on doit la préfixer de `sudo`.

Testons donc nos boucles temps réel parallèles sans laisser de temps CPU pour les autres tâches :

```
# sysctl kernel.sched_rt_runtime_us=-1
kernel.sched_rt_runtime_us = -1
# ./exemple-boucle-temps-reel
#
```

Cette fois, aucune activité ne peut s'exécuter pendant 15 secondes, le système est complètement figé.

Threads temps réel

Jusqu'à présent, nous nous sommes contentés de placer des processus en temps réel. Il est toutefois possible de configurer individuellement l'ordonnancement des threads d'un processus.

Les fonctions concernées sont les suivantes :

```
#include <pthread.h>

int pthread_getschedparam (pthread_t thread,
                           int * ordonnancement,
                     struct sched_param * parametres);

int pthread_setschedparam (pthread_t thread,
                           int ordonnancement,
                 const struct sched_param * parametres);
```

qui permettent de modifier le type d'ordonnancement et la priorité d'un thread existant, ainsi que :

```
int pthread_attr_getschedparam (
                    const pthread_attr_t * attributs,
                    struct sched_param * parametres);
```

Copyright © 2015 Eyrolles.

```
int pthread_attr_setschedparam (
                    pthread_attr_t * attributs,
              const struct sched_param * parametres);

int pthread_attr_getschedpolicy (
                    const pthread_attr_t * attributs,
                    int * ordonnancement);
int pthread_attr_setschedpolicy (
                    pthread_attr_t *attributs,
                    int ordonnancement);
```

Ces fonctions permettent de manipuler un objet « attributs », pour fixer les paramètres désirés dès la création d'un nouveau thread. Schématiquement, le principe est le suivant :

```
pthread_t  thr;
pthread_attr_t  attributs;
struct sched param parametres;

// Fixer les valeurs par défaut pour tous les attributs
pthread_attr_init(& attributs);

// Choisir la priorité temps réel
parametres.sched_priority = 50;

// Inscrire l'ordonnancement dans les attributs
pthread_atttr_setschedpolicy(& attributs, SCHED_FIFO);

// Inscrire la priorité dans les attributs
pthread_attr_setschedparam(& attributs, & parametres);

[...]

// Créer un thread avec les attributs précisés
pthread_create(& thr, & attributs, fonction_thr, NULL);
```

L'objet `attributs` n'est pas modifié lors du `pthread_create()` et n'est plus utilisé par le système une fois le thread lancé. Il est donc possible de le réutiliser, en le modifiant éventuellement, pour créer d'autres threads. Ensuite, il est conseillé d'appeler :

```
void pthread_attr_destroy (pthread_attr_t * attributs);
```

pour libérer ses éventuelles ressources internes (cette fonction n'a aucun effet sous Linux, mais garantit la portabilité du programme).

Attention, lorsqu'un thread est créé par cette méthode, il hérite par défaut de l'ordonnancement de son thread créateur. Autrement dit, il ne tient pas compte des éléments d'ordonnancement que nous avons fixés dans les attributs. Pour confirmer que nous voulons bien utiliser ces paramètres, il faut l'indiquer dans un attribut supplémentaire :

```
int pthread_attr_getinheritsched (
```

Copyright © 2015 Eyrolles.

```
                const pthread_attr_t * attributs,
                int * heritage);

int pthread_attr_setinheritsched (
                pthread_attr_t * attributs,
                int heritage) ;
```

Les valeurs possibles pour le second paramètre sont :

- PTHREAD_INHERIT_SCHED (valeur par défaut) : chaque thread hérite de l'ordonnancement du thread qui le crée ;
- PTHREAD_EXPLICIT_SCHED : le thread créé est ordonnancé en fonction des paramètres indiqués dans l'objet « attributs ».

Il est donc nécessaire dans le code précédent d'ajouter la ligne suivante :

```
pthread_attr_setinheritsched(& attributs,
                             PTHREAD_EXPLICIT_SCHED);
```

avant le pthread_create().

Il existe une dernière fonction concernant les threads et le temps réel :

```
int pthread_attr_getscope (
                const pthread_attr_t * attributs,
                int * portee);

int pthread_attr_setscope (
                pthread_attr_t * attributs,
                int portee);
```

Sur certains systèmes, la configuration de l'ordonnancement s'applique uniquement aux threads entre eux au sein du même processus. Les priorités sont donc relatives à celles du processus englobant. Un thread pourra avoir une priorité 99 par rapport à ses frères (et bénéficier ainsi de l'ordonnancement le plus favorable), mais se trouver limité vis-à-vis des autres tâches du système à la priorité de son processus conteneur.

Sur d'autres environnements, à l'inverse, la priorité d'un thread est relative à l'ordonnancement global du système, quelle que soit la priorité du thread initial du processus.

La norme Posix a donc prévu de pouvoir choisir la portée de l'ordonnancement d'un thread suivant l'un des deux comportements, en indiquant dans la valeur du second paramètre les fonctions suivantes :

- PTHREAD_SCOPE_SYSTEM : l'ordonnancement est global, un thread peut accéder à toutes les priorités du système ;
- PTHREAD_SCOPE_PROCESS : l'échelle des valeurs de priorité est limitée par celle du processus.

Sous Linux, seule la portée globale (PTHREAD_SCOPE_SYSTEM) est disponible. C'est la plus intéressante pour les systèmes temps réel.

Copyright © 2015 Eyrolles.

Il est possible d'obtenir un équivalent approximatif de `PTHREAD_SCOPE_PROCESS` (par exemple, pour le portage d'une application développée sur un autre système) en s'appuyant sur les `cgroups`, mais cela sort du cadre de ce livre.

Nous allons construire un petit exemple où quatre threads vont se concurrencer pour le temps CPU. Chacun d'entre eux affichera son heure de démarrage, effectuera une boucle active de calcul, puis affichera son heure de terminaison. Ils seront placés à des priorités temps réel différentes. Toutefois, les threads les plus prioritaires démarreront après les autres, afin de laisser aux moins prioritaires le temps de commencer avant d'être préemptés.

Les portées des deux boucles imbriquées dans la `fonction_thread()` devront être calibrées pour durer quelques secondes.

exemple-threads-temps-reel.c :

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

#define NB_THREADS 4

void * fonction_thread(void * arg)
{
  int numero = (int) arg;
  int i, j;

  sleep(numero*2);

  fprintf(stderr, "Debut thread %d a %ld\n",
                  numero, time(NULL));
  for (i = 0; i < 100000; i ++)
    for (j = 0; j < 10000; j ++)
        ;
  fprintf(stderr, "Fin thread %d a %ld\n",
                  numero, time(NULL));
  return NULL;
}

int main(void)
{
  int i;
  int err;
  pthread_attr_t attr;
  struct sched_param param;
  pthread_t thr[NB_THREADS];
```

Copyright © 2015 Eyrolles.

```
  pthread_attr_init (& attr);
  if ((err = pthread_attr_setschedpolicy(& attr,
                                  SCHED_FIFO)) != 0) {
    fprintf(stderr, "setschedpolicy: %s\n",
                    strerror(err));
    exit(EXIT_FAILURE);
  }
  if ((err = pthread_attr_setinheritsched(& attr,
                    PTHREAD_EXPLICIT_SCHED)) != 0) {
    fprintf(stderr, "setinheritsched: %s\n",
                    strerror(err));
    exit(EXIT_FAILURE);
  }
  for (i = 0; i < NB_THREADS; i ++) {
    param.sched_priority = (i + 1) * 10;
    if ((err = pthread_attr_setschedparam(& attr,
                                    & param)) != 0) {
      fprintf(stderr, "setschedparam: %s\n",
                      strerror(err));
      exit(EXIT_FAILURE);
    }
    if ((err = pthread_create(& (thr[i]), & attr,
              fonction_thread, (void *) (i+1))) != 0) {
      fprintf(stderr, "pthread_create: %s\n",
                      strerror(err));
      exit(EXIT_FAILURE);
    }
  }
  for (i = 0; i < NB_THREADS; i ++)
    pthread_join(thr[i], NULL);

  return EXIT_SUCCESS;
}
```

À l'exécution, nous constatons que les threads les plus prioritaires se terminent bien en premier, bien qu'ils aient démarrés avant les autres. Bien entendu, pour qu'ils soient en concurrence, il convient de tous les placer sur le même processeur.

```
# taskset -c 0 ./exemple-threads-temps-reel
Debut thread 1 a 1317907166
Debut thread 2 a 1317907168
Debut thread 3 a 1317907170
Debut thread 4 a 1317907172
Fin thread 4 a 1317907175
Fin thread 3 a 1317907176
Fin thread 2 a 1317907177
Fin thread 1 a 1317907178
#
```

Copyright © 2015 Eyrolles.

Threads en Round Robin

En général, lors de la conception d'un système, on essaye d'affecter une seule tâche par niveau de priorité. L'échelle proposée par Linux [1, 99] est suffisamment large pour la plupart des applications que l'on construira avec ce système.

Les tâches les plus urgentes, et celles tolérant le moins de fluctuations temporelles, seront placées à des niveaux de priorité les plus élevés. Les autres processus ou threads d'importance moindre, ou acceptant plus facilement des jitters, se retrouveront à des niveaux de priorité plus faibles.

Lorsque des tâches ont des niveaux équivalents de criticité ou d'urgence, on leur donne des priorités proches. Toutefois, il existe des cas où l'on est obligé de fixer plusieurs tâches sur le même niveau de priorité. C'est le cas notamment lorsque le nombre de tâches n'est pas connu à l'avance, mais est déterminé à l'initialisation du système (nombre variable de ports de communication à gérer, par exemple) ou même dynamiquement (à chaque connexion d'un nouveau client distant).

Si les tâches de même niveau ont des traitements conséquents à réaliser, l'ordonnancement Fifo n'est pas nécessairement adapté, car lorsqu'une tâche prend le CPU et commence son traitement, toutes les autres seront gelées pendant toute la durée de son travail. Pour ces situations, on préférera un ordonnancement Round Robin, où une tâche peut s'exécuter pendant une tranche de temps au bout de laquelle elle est préemptée et placée à la fin de la liste des tâches de mêmes priorités.

Nous allons reprendre l'exemple précédent avec quatre threads, placés cette fois au même niveau de priorité. Les threads boucleront autour de l'appel système `gettimeofday()`. Comme nous l'avons observé au chapitre 4, ceci nous permettra de détecter les préemptions de chaque thread. Le seuil de détection est ici fixé arbitrairement à 1 milliseconde.

```
exemple-threads-rr.c :
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

#define NB_THREADS 4

void * fonction_thread(void * arg)
{
  int numero = (int) arg;

  struct timeval debut, precedente, heure;
  long long int difference;

  sleep(2);

  gettimeofday(& debut, NULL);
```

Copyright © 2015 Eyrolles.

```
  heure = debut;

  while (1) {
    do {
      precedente = heure;
      gettimeofday(& heure, NULL);
      // Boucle limitée à 10 secondes
      if ((heure.tv_sec - debut.tv_sec) > 10)
        pthread_exit(NULL);
      difference  = heure.tv_sec - precedente.tv_sec;
      difference *= 1000000;
      difference += heure.tv_usec - precedente.tv_usec;
    } while (difference < 1000); // 1 milliseconde
    fprintf(stdout, "[%ld.%06ld] %d preempte\n",
           precedente.tv_sec, precedente.tv_usec, numero);
    fprintf(stdout, "[%ld.%06ld] %d active\n",
           heure.tv_sec, heure.tv_usec, numero);
  }

  return NULL;
}

int main(void)
{
  int i;
  int err;
  pthread_attr_t attr;
  struct sched_param param;
  pthread_t thr[NB_THREADS];

  pthread_attr_init (& attr);
  if ((err = pthread_attr_setschedpolicy(& attr,
                                         SCHED_RR)) != 0) {
    fprintf(stderr, "setschedpolicy: %s\n",
                    strerror(err));
    exit(EXIT_FAILURE);
  }
  if ((err = pthread_attr_setinheritsched(& attr,
                          PTHREAD_EXPLICIT_SCHED)) != 0) {
    fprintf(stderr, "setinheritsched: %s\n",
                     strerror(err));
    exit(EXIT_FAILURE);
  }
  param.sched_priority = 40;
  if ((err = pthread_attr_setschedparam(& attr,
                                        & param)) != 0) {
    fprintf(stderr, "setschedparam: %s\n", strerror(err));
    exit(EXIT_FAILURE);
  }
```

Copyright © 2015 Eyrolles.

```
  for (i = 0; i < NB_THREADS; i ++) {
    if ((err = pthread_create(& (thr[i]), & attr,
                fonction_thread, (void *) (i+1))) != 0) {
      fprintf(stderr, "pthread_create: %s\n",
                                  strerror(err));
      exit(EXIT_FAILURE);
    }
  }
  for (i = 0; i < NB_THREADS; i ++)
    pthread_join(thr[i], NULL);

  return EXIT_SUCCESS;
}
```

Lors de l'exécution, les affichages dus aux différents threads ne se produisent pas par ordre chronologique, aussi utiliserons-nous l'utilitaire `sort` pour mieux visualiser les périodes :

```
# taskset -c 0 ./exemple-threads-rr | sort
[1317912293.800457] 1 preempte
[1317912294.800457] 2 preempte
[1317912295.800461] 3 preempte
[1317912296.800462] 4 preempte
[1317912296.800470] 1 active
[1317912296.900457] 1 preempte
[1317912296.900463] 2 active
[1317912297.000458] 2 preempte
[1317912297.000463] 3 active
[1317912297.100457] 3 preempte
[1317912297.100463] 4 active
[1317912297.200457] 4 preempte
[1317912297.200462] 1 active
[1317912297.300460] 1 preempte
[1317912297.300462] 2 active
[1317912297.400460] 2 preempte
[1317912297.400461] 3 active
[1317912297.500460] 3 preempte
[1317912297.500462] 4 active
[1317912297.600460] 4 preempte
[1317912297.600462] 1 active
[1317912297.700457] 1 preempte
[1317912297.700461] 2 active
[...]
[1317912305.000040] 4 active
#
```

Les quatre premières lignes sont incohérentes, ce qui est normal car les threads démarrent simultanément. Ensuite, nous observons bien une exécution régulière 1 – 2 – 3 – 4 – 1 – 2 – 3 – 4

Copyright © 2015 Eyrolles.

Les threads s'exécutent par tranches de 100 millisecondes. Cette durée dépend du processus lui-même et peut être consultée par l'appel système :

```
int sched_rr_get_interval (pid_t processus,
                  struct timespec * intervalle);
```

Toutefois, il n'existe pas de fonction standard pour fixer la tranche de temps utilisée dans les ordonnancements temps réel Round Robin.

Nous pouvons observer également le temps de commutation entre deux threads du même processus : environ 6 microsecondes sur ce système. Nous reviendrons sur ce point dans le prochain chapitre.

Rotation sans Round Robin

L'ordonnancement `SCHED_RR` permet d'activer successivement les différentes tâches à un niveau donné. Notons par ailleurs que si l'une des tâches est ordonnancée en `SCHED_FIFO`, dès que les précédentes lui auront cédé le CPU, elle le conservera sans préemption jusqu'à s'endormir volontairement. Seules les tâches en Round Robin sont préemptées, pas celles en Fifo.

Le Round Robin pose malgré tout un problème dans certains cas : la préemption peut survenir à n'importe quel moment et il est impossible de fixer des portions de code critiques durant lesquelles il ne faut pas interrompre la tâche.

Ce souci survient souvent dans les applications qui doivent dialoguer avec du matériel et respecter une temporisation assez précise entre des opérations d'entrées-sorties.

Lorsque l'on veut faire « tourner » des tâches consommatrices de CPU en parallèle au même niveau de priorité temps réel, il existe néanmoins une solution : l'appel système.

```
int sched_yield (void);
```

Ce dernier a pour effet de suspendre la tâche courante et de la placer en fin de file des tâches en attente de sa priorité. Si aucune tâche de même priorité n'est prête à s'exécuter, l'appel système revient immédiatement sans modification pour la tâche appelante.

> Sur certains systèmes temps réel classiques, l'habitude était d'utiliser un sommeil de durée nulle `sleep(0)` pour réaliser cette opération.

Notre rotation entre tâches sera donc organisée « manuellement », chacune réalisant une boucle du type :

```
while ( ! condition_de_sortie) {
  // Réaliser le traitement critique
  [...]
  // Céder le CPU aux autres tâches de même priorité
  sched_yield();
}
```

Copyright © 2015 Eyrolles.

Naturellement, pour ne pas être préemptées intempestivement, toutes les tâches seront ordonnancées en `SCHED_FIFO`.

Temps réel depuis le shell

Lorsque l'on désire lancer un processus en temps réel, mais qu'il n'a pas été spécifiquement prévu pour – ne faisant pas appel à `sched_setschedparam()`, par exemple – il existe une solution simple et élégante : l'utilitaire `chrt` disponible sur la plupart des distributions Linux.

> J'ai découvert cet utilitaire tardivement – en 2008 je crois – alors que j'employais depuis des années un équivalent « fait maison ». Convaincu de l'utilité de mon programme, je me suis amusé à l'intégrer dans le package Busybox qui sert sur la plupart des plates-formes Linux embarqué. Une fois mon portage terminé, et avant de le soumettre aux mainteneurs de Busybox, j'ai recherché un nom cohérent pour la commande (je l'appelais `set-rt-priority` jusque là mais c'était un peu long). En parcourant la liste des applets de Busybox, mon attention a été attirée par la commande `chrt` que je ne connaissais pas. J'ai alors découvert que j'avais réinventé un utilitaire qui existait déjà !

L'interface de `chrt` n'est pas très intuitive. Elle fonctionne un peu comme `taskset` que nous avons examinée au chapitre 1 et qui permet de fixer l'affinité d'un processus. Voici quelques exemples d'utilisation :

```
# chrt  -f  50  ./commande
     lance la commande en ordonnancement Fifo priorité 50
# chrt  -r  10  ./commande
     lance la commande en Round Robin priorité 10
# chrt  -o  0   ./commande
     lance la commande en temps partagé (Other) priorité 0
# chrt -p  12345
     affiche ordonnancement et priorité du processus PID 12345
# chrt  -pf 50  12345
     passe le processus de PID 12345 en Fifo priorité 50
# chrt  -pr 30  12345
     passe le processus de PID 12345 en RR priorité 50
```

Bien entendu, lorsqu'il s'agit de lancer ou passer une tâche en ordonnancement temps réel, les droits *root* sont nécessaires pour exécuter `chrt`.

Il est possible, par exemple, d'utiliser `chrt` pour améliorer la fluidité de certaines applications multimédias (vidéo, audio, etc.).

Voici un exemple :

Copyright © 2015 Eyrolles.

```
# chrt -pf 80 $$
# mplayer video-01.avi
Creating config file: /root/.mplayer/config
MPlayer 1.0rc4-4.5.2 (C) 2000-2010 MPlayer Team
[...]
Starting playback...
Movie-Aspect is 2.35:1 - prescaling to correct movie aspect.
A:  28.9 V:  28.9 A-V: -0.000 ct: -0.041 694/694  2%  1%  0.3% 1 0
(Contrôle-C)
Exiting... (Quit)
#
```

On remarquera que j'ai commencé par passer mon shell (dont le PID est toujours contenu dans la variable spéciale `$$`) en temps réel Fifo, priorité 80. Ensuite, j'ai lancé le lecteur vidéo `mplayer` qui hérite bien sûr de l'ordonnancement de son père et dispose dans ce cas d'une grande stabilité de ses timers de synchronisation, même en cas de forte charge du système.

Ordonnancement EDF *(Earliest Deadline First)*

Un nouveau type d'ordonnancement a été intégré courant 2014 dans le noyau Linux (dans la version 3.14). Il n'est pas défini par la norme Posix, et nécessite un appel système spécifique pour sa mise en œuvre.

L'ordonnancement s'appelle `SCHED_DEADLINE`. Il s'agit d'une composition entre les algorithmes EDF *(Earliest Deadline First)* et CBS *(Constant Bandwidth Server).*

Une tâche soumise à l'ordonnancement `SCHED_DEADLINE` est dotée de trois paramètres, tous exprimés en nanosecondes :

- *runtime :* une durée d'exécution maximale pour la tâche, c'est le temps CPU maximum nécessaire pour réaliser le traitement, on peut l'estimer ou le calculer par des itérations préalables ;
- *deadline :* une date de réalisation à laquelle la tâche doit être terminée, elle est mesurée à partir du début de l'activation (réveil) de la tâche ;
- *period :* la durée d'un cycle d'exécution de la tâche. Celle-ci peut en effet être périodique, et les paramètres précédents s'appliqueront pour chaque itération.

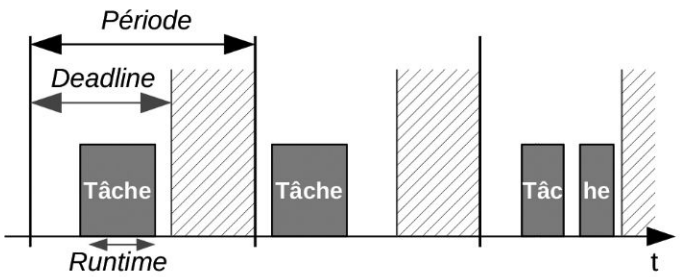
L'algorithme CBS permet au noyau de garantir la disponibilité d'une durée au moins égale à *runtime* durant une période de durée *deadline* après le début de chaque période. Ceci est réalisé en calculant statiquement les durées d'exécution de toutes les tâches concernées par cet ordonnancement et en s'assurant qu'elles pourront respecter les contraintes demandées.

Rien n'empêche le noyau de préempter temporairement une tâche `SCHED_DEADLINE`, à condition de lui rendre le contrôle suffisamment tôt pour qu'elle puisse terminer son activité avant la limite *deadline*. C'est le cas sur la figure 5-12, lors de l'exécution du troisième cycle.

Copyright © 2015 Eyrolles.

Figure 5-12

Ordonnancement SCHED_DEADLINE



L'algorithme EDF sert à sélectionner une tâche lorsque le *scheduler* est invoqué. En l'occurrence, il activera celle dont la limite d'exécution est la plus proche.

Pour basculer une tâche en ordonnancement SCHED_DEADLINE, nous ne pouvons pas utiliser le classique appel sched_setscheduler() car il ne permet pas de fixer les trois arguments précédents.

> En principe, rien n'interdirait d'ajouter des champs dans la structure sched_param, mais comme le type d'ordonnancement n'est pas standardisé par Posix, les développeurs de Linux ont préféré ne pas la modifier et ajouter un appel système plus général, qui, par ailleurs, permet de configurer également les autres ordonnancements.

```
int sched_setattr (pid_t tâche, const struct sched_attr * attr, unsigned int flags);
int sched_gettattr (pid_t tâche, struct sched_attr * attr, unsigned int size, unsigned int flags);
```

La structure sched_attr contient les champs nécessaires pour configurer tous les ordonnancements actuels de Linux.

Champ	Signification
sched_policy	La classe d'ordonnancement : SCHED_FIFO, SCHED_RR, SCHED_OTHER ou SCHED_DEADLINE.
sched_priority	La priorité temps réel désirée pour les ordonnancements SCHED_FIFO ou SCHED_RR. La valeur zéro est obligatoire pour les autres ordonnancements.
sched_nice	La « gentillesse » temps partagé d'une tâche en ordonnancement SCHED_OTHER. Zéro pour les autres.
sched_runtime	La durée maximale estimée (en nanosecondes) pour l'exécution de la tâche, seulement en ordonnancement SCHED_DEADLINE.
sched_deadline	La durée (en nanosecondes) entre l'activation de la tâche SCHED_DEADLINE et l'instant où elle doit avoir terminé son travail.
sched_period	La période du cycle d'activation (en nanosecondes) d'une tâche sous ordonnancement SCHED_DEADLINE.

Copyright © 2015 Eyrolles.

Champ	Signification
sched_flags	Le seul attribut accepté aujourd'hui dans ce champs est SCHED_FLAG_RESET_ON_FORK, qui indique qu'un éventuel processus fils futur n'héritera pas de l'ordonnancement de son parent. Cet attribut est obligatoire pour l'ordonnancement SCHED_DEADLINE.
size	Ce champ doit être initialisé avec sizeof(struct sched_attr) pour gérer les évolutions futures des paramètres.

Le paramètre flags des appels système sched_setattr() et sched_getattr() doit être nul, le paramètre size de sched_getattr() est la taille de la zone mémoire passée en second argument, qui doit être au moins égale à sizeof(struct sched_attr).

Pour l'ordonnancement SCHED_DEADLINE, le paramètre sched_period doit être supérieur à sched_deadline, qui doit à son tour être supérieur à sched_runtime.

Au moment de la rédaction de ces lignes, la bibliothèque C installée sur la distribution que j'utilise ne connaît pas encore les appels système sched_getattr() et sched_setattr(). Pour pouvoir faire fonctionner l'exemple suivant, il faut les déclarer explicitement en employant l'appel système syscall(). Il s'agit de la portion de code encadrée par un #ifndef SCHED_DEADLINE dans l'exemple suivant. Les numéros des appels système varient d'une architecture à l'autre. J'ai mentionné ici les valeurs pour les architectures les plus courantes. Il faudra éventuellement les adapter pour de nouvelles plates-formes en consultant le fichier de définition des appels système sous le répertoire arch/<architecture>/kernel du noyau.

L'exemple ci-dessous bascule le processus sous ordonnancement SCHED_DEADLINE, puis crée un timer avec la période indiquée. Dans la fonction de traitement du timer, nous affichons en permanence l'écart minimal et l'écart maximal observés entre deux déclenchements successifs.

exemple-sched-deadline.c :

```
#define _GNU_SOURCE

#include <sched.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>

#ifndef SCHED_DEADLINE

  #define SCHED_DEADLINE  6

  #include <sys/syscall.h>

  // Définitions pour x86_64
  #define SCHED_SETATTR_SYSCALL_NB 314
  #define SCHED_GETATTR_SYSCALL_NB 315
```

Copyright © 2015 Eyrolles.

```
  // Définitions pour i386
  //#define SCHED_SETATTR_SYSCALL_NB 351
  //#define SCHED_GETATTR_SYSCALL_NB 352

  // Définitions pour ARM
  //#define SCHED_SETATTR_SYSCALL_NB 380
  //#define SCHED_GETATTR_SYSCALL_NB 381

  // Définitions pour PowerPC
  //#define SCHED_SETATTR_SYSCALL_NB 355
  //#define SCHED_GETATTR_SYSCALL_NB 356

  struct sched_attr {
    __uint32_t size;

    __uint32_t sched_policy;
    __uint64_t sched_flags;

    /* SCHED_OTHER */
    __int32_t sched_nice;

    /* SCHED_FIFO, SCHED_RR */
    __uint32_t sched_priority;

    /* SCHED_DEADLINE */
    __uint64_t sched_runtime;
    __uint64_t sched_deadline;
    __uint64_t sched_period;
  };

  #define SCHED_FLAG_RESET_ON_FORK  1

  long sched_getattr(pid_t pid, struct sched_attr * attr,
                                unsigned int size, unsigned int flags)
  {
    return syscall(SCHED_GETATTR_SYSCALL_NB, pid, attr, size, flags);
  }


  long sched_setattr(pid_t pid, struct sched_attr * attr, unsigned int flags)
  {
    return syscall(SCHED_SETATTR_SYSCALL_NB, pid, attr, flags);
  }
#endif


static void rtmin_handler(int unused);


int main(int argc, char * argv[])
```

Copyright © 2015 Eyrolles.

```
{
  long int runtime;
  long int deadline;
  long int period;
  timer_t timer;
  struct sched_attr attr;
  struct sigevent  event;
  struct itimerspec spec;

  if ((argc < 4)
   || (sscanf(argv[1], "%ld", &runtime)  != 1)
   || (sscanf(argv[2], "%ld", &deadline) != 1)
   || (sscanf(argv[3], "%ld", &period)   != 1)) {
    fprintf(stderr, "usage: %s <runtime> <deadline> <period> (all in usec)\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  memset(& attr, 0, sizeof(struct sched_attr));
  attr.size           = sizeof(struct sched_attr);
  attr.sched_flags    = SCHED_FLAG_RESET_ON_FORK;
  attr.sched_policy   = SCHED_DEADLINE;
  attr.sched_runtime  = runtime * 1000;
  attr.sched_deadline = deadline * 1000;
  attr.sched_period   = period * 1000;
  if (sched_setattr(0, & attr, 0) != 0) {
    perror("sched_setattr");
    exit(EXIT_FAILURE);
  }

  event.sigev_notify = SIGEV_SIGNAL;
  event.sigev_signo  = SIGRTMIN;
  signal(SIGRTMIN, rtmin_handler);

  spec.it_interval.tv_sec  = period / 1000000;
  spec.it_interval.tv_nsec = (period % 1000000) * 1000; // in nsec.
  spec.it_value = spec.it_interval;

  if (timer_create(CLOCK_REALTIME, & event, & timer) != 0) {
    perror("timer_create");
    exit(EXIT_FAILURE);
  }

  if (timer_settime(timer, 0, & spec, NULL) != 0) {
    perror("timer_settime");
    exit(EXIT_FAILURE);
  }
  while(1)
    pause();

  return EXIT_SUCCESS;
}
```

Copyright © 2015 Eyrolles.

```
static void rtmin_handler(int unused)
{
  struct timeval tv;
  static struct timeval prev = {0, 0};
  long int delay;
  static long int min_delay = -1;
  static long int max_delay = -1;
  int change;

  gettimeofday(& tv, NULL);
  if (prev.tv_sec > 0) {
    delay  = (tv.tv_sec - prev.tv_sec) * 1000000
           + (tv.tv_usec - prev.tv_usec);
    change = 0;
    if ((min_delay < 0) || (delay < min_delay)) {
      min_delay = delay;
      change = 1;
    }
    if ((max_delay < 0) || (delay > max_delay)) {
      max_delay = delay;
      change = 1;
    }
    if (change)
      fprintf(stdout, "Min=%ld, Max=%ld\n", min_delay, max_delay);
  }
  prev = tv;
}
```

Il faut noter une particularité pour un processus ordonnancé en `SCHED_DEADLINE` : son affinité CPU ne doit pas être contrainte. En d'autres termes, il ne faut pas utiliser `taskset` par exemple pour interdire l'exécution sur certains cœurs, sinon le noyau refusera le passage en ordonnancement `SCHED_DEADLINE`.

Voici un exemple d'exécution sur Raspberry Pi 2, où nous demandons un timer toutes les dix millisecondes, le handler associé doit s'exécuter dans les cinq premières millisecondes de chaque cycle, avec une durée d'exécution maximale (largement surestimée) d'une milliseconde. Les arguments de ligne de commande et les résultats sont en microsecondes.

Le programme n'affiche de nouvelles lignes que lorsque le minimum ou le maximum observés changent.

```
# ./exemple-sched-deadline  1000  5000  10000
[0] Min=9973, Max=9973
[0] Min=9973, Max=10013
[9] Min=9973, Max=10015
[10] Min=9973, Max=10017
[11] Min=9973, Max=10020
[12] Min=9973, Max=10037
```

Copyright © 2015 Eyrolles.

```
[24] Min=9973, Max=10071
[24] Min=9937, Max=10071
[25] Min=9937, Max=10149
[25] Min=9862, Max=10149
```

En quelques secondes (quelques centaines d'itérations), les valeurs minimales et maximales se stabilisent. Logiquement, l'écart entre deux timers ne doit pas dépasser quinze millisecondes (si la première occurrence tombe au tout début du cycle et la seconde à la limite tolérée du cycle suivant). Nous voyons qu'avec un maximum de 10 149 microsecondes, la limite est respectée.

Si on laisse le programme tourner sur un système normalement chargé (même avec des tâches temps réel de haute priorité), la limite est bien respectée. On remarque toutefois des dérives dès que la charge en interruptions matérielles est importante. Nous examinerons dans le prochain chapitre si ce comportement est identique pour les ordonnancements temps réel `SCHED_FIFO` et `SCHED_RR`.

Conclusion

Nous avons observé dans ce chapitre les bases de l'ordonnancement temps réel sous Linux. J'encourage vivement le lecteur à se livrer à ses propres expériences, en réalisant des petites boucles temporisées ou protégées par `alarm()`, afin de se rendre compte des effets sur la réactivité du système.

Notre prochain chapitre va s'attacher à étudier les limites du temps réel souple sous Linux.

Points clés

- On peut ordonnancer des tâches (processus complets ou threads uniques) en temps réel dans une échelle allant de 1 à 99. Les tâches temps partagé classiques ont l'équivalent d'une priorité temps réel nulle.
- Il existe deux ordonnancements temps réel sous Linux : Fifo ou Round Robin. Une tâche Fifo ne peut être préemptée que par une tâche de priorité strictement supérieure. En Round Robin, une tâche s'interrompt périodiquement pour laisser s'exécuter les éventuelles autres tâches de même priorité.
- Sous Linux, un garde-fou conserve une partie du temps processeur pour les tâches temps partagé même quand une tâche temps réel s'exécute. Pour le désactiver, on configure `/proc/sys/kernel/sched_rt_runtime_us`.
- L'utilitaire `chrt` permet de fixer l'ordonnancement d'un processus depuis la ligne de commandes du shell.
- Le nouvel ordonnancement `SCHED_DEADLINE` apparu dans le noyau 3.14 permet de cadrer précisément les limites temporelles d'exécution d'une tâche (généralement cyclique). Bien qu'il ne soit pas classé dans les ordonnancements temps réel classiques, il a une priorité plus grande que ces derniers, puisque la portion de temps CPU nécessaire est réservée. L'utilisation d'un appel système spécifique est un handicap pour cet ordonnancement sur les systèmes où la bibliothèque C n'en tient pas encore compte.

Copyright © 2015 Eyrolles.

Exercices

Les exercices suivants peuvent, en cas d'erreur, nécessiter d'éteindre « brutalement » votre machine. Assurez-vous avant de les démarrer d'avoir sauvegardé toutes les données importantes des applications en cours. Pensez également à lancer la commande `sync` depuis le shell avant de démarrer un programme susceptible de boucler indéfiniment. Enfin, il est évident qu'il ne faut pas effectuer ces manipulations si d'autres utilisateurs sont connectés sur le même système.

Exercice 1 (*)

Utilisez la commande `chrt` pour examiner l'ordonnancement des processus présents sur votre système. En voyez-vous avec un ordonnancement temps réel ? Examinez plus particulièrement les threads du noyau (présentés entre crochets au début des résultats de la commande `ps aux`).

Exercice 2 (*)

Essayez de passer votre shell en temps réel avec `chrt`. Quels sont les droits nécessaires ? Pourquoi ? Observez-vous des différences sur le comportement des commandes que vous passez depuis ce shell ?

Exercice 3 (**)

Passez en temps réel (Fifo ou Round Robin) avec une priorité 10 tous les processus de votre système précédemment identifiés comme étant ordonnancés en temps partagé.

Constatez-vous un fonctionnement différent de votre environnement ?

Exercice 4 (**)

Lancez le programme `exemple-boucle-temps-reel`. Le système est-il actif ? Gelé ? Totalement gelé ? Pourquoi ?

Jouez (avec des redirections de `echo`) sur le contenu du pseudo-fichier `/proc/sys/kernel/sched_rt_runtime_us` pour garantir un vrai ordonnancement temps réel à la boucle.

Exercice 5 (***)

Lancez une boucle temps réel avec une priorité Fifo 40 (après configuration correcte de `sched_rt_runtime_us`).

Au préalable, montez à la priorité Fifo 50 les processus nécessaires pour conserver le contrôle sur votre système (serveur X-Window, environnement graphique, etc.).

Copyright © 2015 Eyrolles.

6

Performances du temps réel souple

Nous avons vu que nous disposons d'outils nous permettant de fixer – tant depuis le shell qu'au sein d'une application – l'ordonnancement des tâches en suivant des politiques Fifo ou Round Robin, avec des priorités dans l'échelle [1, 99]. Nous allons maintenant essayer de mesurer les performances de ce temps réel souple et d'en déterminer les limites d'application.

Timers temps réel

Précisions et fluctuations

Notre première expérience va consister à vérifier si les tâches temps réel bénéficient d'une meilleure précision pour réaliser des opérations périodiques.

Reprenons le programme `exemple-timer-create-02.c` du chapitre 4 et exécutons-le sous un ordonnancement Fifo de haute priorité en sauvegardant ses résultats dans un fichier :

```
# taskset -pc 0 $$
pid 3189's current affinity list: 0-3
pid 3189's new affinity list: 0
# chrt -f 99  ./exemple-timer-create-02 1000 > data-rt-01.txt
# ./calculer-statistiques < data-rt-01.txt
```

Copyright © 2015 Eyrolles.

```
 Nb mesures = 5000
 Minimum = 930
 Maximum = 1063
 Moyenne = 999
 Ecart-type = 5
#
```

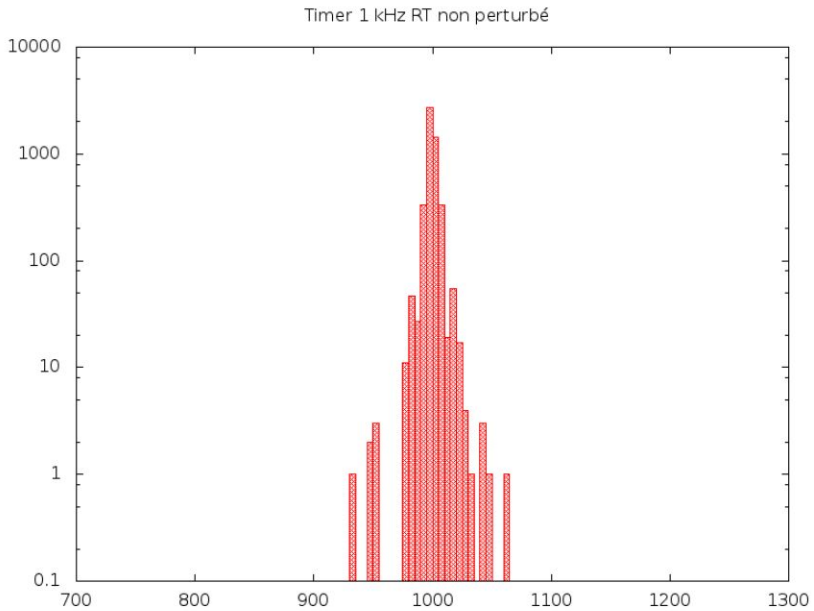
Nous voyons que les statistiques sont meilleures que pour le temps partagé (sans perturbation), où nous avions obtenu un écart maximal de 1 224 microsecondes entre deux déclenchements du timer, et un écart-type de 7 microsecondes par rapport à la valeur moyenne d'une milliseconde.

Un graphique nous permettra de mieux voir les fluctuations :

```
# ./calculer-histogramme 100 750 1250 < data-rt-01.txt > histo-rt-01.txt
# ./afficher-histogramme.sh histo-rt-01.txt "Timer 1 kHz RT non perturbé" figure-6-1.png
```

Figure 6-1

Timer temps réel non perturbé



Si on le compare à la figure 4-1 du chapitre 4, nous voyons que les écarts par rapport à la moyenne sont du même ordre, mais qu'il n'y a plus d'occurrences déviant sur les côtés.

Essayons à présent la même expérience avec un programme perturbateur (en temps partagé bien sûr) exécuté sur un autre terminal, sur le même processeur.

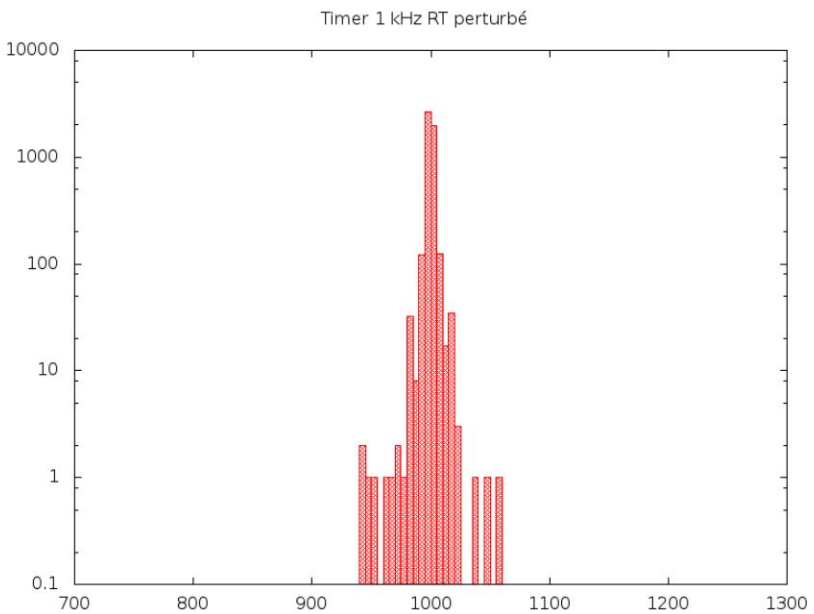
Copyright © 2015 Eyrolles.

```
$ taskset -pc 0 $$
                                $ ./exemple-perturbateur
# chrt -f 99  ./exemple-timer-create-02 1000 > data-rt-02.txt
# ./calculer-statistiques < data-rt-02.txt
 Nb mesures = 5000
 Minimum = 943
 Maximum = 1058
 Moyenne = 999
 Ecart-type = 4
# ./calculer-histogramme 100 750 1250 < data-rt-02.txt > histo-rt-02.txt
# ./afficher-histogramme.sh histo-rt-02.txt "Timer 1 kHz RT perturbé"
```

Les valeurs sont bien stables et comparables aux précédentes. L'écart-type est de 4 microsecondes, alors qu'au chapitre 4 nous avions un écart-type quinze fois plus élevé dans les mêmes circonstances.

Figure 6-2

Timer temps réel avec perturbation



Je vous encourage à comparer ce graphique avec la figure 4-2 du chapitre 4, où nous avions été obligé d'étirer l'axe des abscisses vers la droite jusqu'à 5 millisecondes.

Les résultats précédents ont été obtenus sur le cœur 0 d'un Raspberry Pi 2. C'est la pire des situations puisque par défaut la plupart des interruptions sont traitées sur ce cœur. Comme nous l'avons vu dans le chapitre 2, il serait possible de rediriger les traitements d'interruption sur d'autres cœurs.

Copyright © 2015 Eyrolles.

Granularité des timers

Étant donné les fluctuations observées lors de nos manipulations en temps partagé, nous n'avons pas cherché à vérifier la finesse que nous pouvons attendre lors de déclenchements périodiques.

Cette fois, nous pouvons essayer de diminuer la période des timers temps réel et vérifier la précision des déclenchements. Ceci se réalise ainsi :

```
# chrt -f 99 ./exemple-timer-create-02 1000 | ./calculer-statistiques
 Nb mesures = 5000
 Minimum = 997
 Maximum = 1002
 Moyenne = 999
 Ecart-type = 1
#
```

Pour éviter les collisions avec les autres interruptions du système, j'ai choisi de placer ce test sur le cœur numéro 2 d'un Raspberry Pi 2.

J'ai regroupé dans le tableau suivant les résultats de plusieurs essais successifs.

Période demandée	Moyenne	Mini	Maxi	Écart-type
1000	999	962	1014	3
500	499	411	590	3
100	99	54	197	1
50	49	13	100	3
10	21	12	163	10
5	25	12	195	12

Nous voyons que notre tâche périodique peut arriver à fonctionner raisonnablement jusqu'à des périodes de quelques dizaines de microsecondes (à condition d'être ordonnancée en temps réel avec une priorité suffisante). Les valeurs moyennes sont toujours bonnes et les écarts-types (qui sont significatifs des fluctuations du timer) restent faibles. Toutefois, il faut noter que si un écart-type de 3 microsecondes représente 0,3 % d'un timer à la milliseconde, il correspond également à 6 % d'un timer à 50 microsecondes.

Conclusion sur les timers

Je considère que le temps réel souple est très adapté pour faire fonctionner des tâches périodiques de quelques centaines de microsecondes à quelques millisecondes. Naturellement, il faut être conscient que ces timers pourront être retardés occasionnellement de plusieurs centaines de microsecondes, et que le système corrigera la fluctuation sur les itérations suivantes.

Copyright © 2015 Eyrolles.

On notera que ces valeurs sont très différentes d'un processeur à l'autre. Il sera nécessaire dans la phase d'étude d'un projet temps réel de mesurer sur le système cible les valeurs (à l'aide des petits exemples précédents, par exemple).

La granularité des timers dépend également de l'option High Resolution Timers Support du menu Processor Type and Features lors de la compilation du noyau.

Pour vérifier si cette option est activée sur un système en cours de fonctionnement, il suffit de saisir :

```
# zcat /proc/config.gz | grep HIGH_RES_TIMERS
CONFIG_HIGH_RES_TIMERS=y
```

Cela implique naturellement que le support pour le fichier `/proc/config.gz` ait été activé dans le noyau.

Temps de commutation

Un point important dans les systèmes temps réel est de connaître le temps de commutation entre deux tâches synchronisées sur une ressource partagée (mutex, sémaphore, etc.) Les fournisseurs de systèmes temps réels stricts propriétaires indiquent généralement ces valeurs dans les spécifications de leurs produits, garantissant des durées maximales de commutation.

Dans les systèmes temps réels souples, nous n'avons pas de garantie aussi absolue sur les durées de commutation. Nous pouvons toutefois les mesurer et valider ainsi par l'observation le comportement de notre système.

Commutation entre threads

Le temps de commutation le plus simple à mesurer est celui entre deux threads d'un même processus. Pour cela, il suffit que les deux threads se synchronisent sur le même mutex. Ils partagent également une variable globale `struct timeval`.

Le principe est le suivant : le thread 2 tient le mutex à l'origine, et on répète les opérations suivantes en boucle.

Thread 1	Thread 2
Demande le mutex	Attend 10 ms
	Inscrit l'heure dans la `struct timeval` partagée
	Relâche le mutex
Obtient le mutex	
Lit l'heure et calcule la durée de commutation	Demande le mutex
Relâche le mutex	Obtient le mutex
Attend 10 ms	

Copyright © 2015 Eyrolles.

Pour être sûr que le thread 2 ne récupère pas instantanément le mutex après l'avoir relâché, nous rendons le thread 1 sensiblement plus prioritaire. Ce dernier devra alors utiliser un petit sommeil (10 ms, par exemple) pour que le thread 2 ait le temps de reprendre le mutex lorsqu'il le libère.

> Nous étudierons au chapitre 7 une autre méthode permettant de mieux se synchroniser sur un mutex.

Voici un petit exemple :

```
exemple-commutation-threads.c
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

static pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

#define NB_COMMUTATIONS_MAX  1000
static int nb_commutations = 0;
static long int durees[NB_COMMUTATIONS_MAX];

static struct timeval avant_commutation;


void * thread_haute_prio (void * unused)
{
  struct timeval apres_commutation;
  usleep(100000); // ms
  while (nb_commutations < NB_COMMUTATIONS_MAX) {
    pthread_mutex_lock(& mutex);
    gettimeofday(& apres_commutation, NULL);
    durees[nb_commutations]  = apres_commutation.tv_sec
                             - avant_commutation.tv_sec;
    durees[nb_commutations] *= 1000000;
    durees[nb_commutations] += apres_commutation.tv_usec
                             - avant_commutation.tv_usec;
    nb_commutations ++;
    pthread_mutex_unlock(& mutex);
    usleep(10000); // 10 ms
  }
  return NULL;
}
```

Copyright © 2015 Eyrolles.

```
void * thread_basse_prio (void * unused)
{
  pthread_mutex_lock(& mutex);
  usleep(200000); // 200 ms
  while (nb_commutations < NB_COMMUTATIONS_MAX) {
    gettimeofday(& avant_commutation, NULL);
    pthread_mutex_unlock(& mutex);
    pthread_mutex_lock(& mutex);
    usleep(10000); // 10 ms
  }
  pthread_mutex_unlock(& mutex);
  return NULL;
}

int main(void)
{
  int i;
  int err;
  pthread_attr_t attr;
  struct sched_param param;
  pthread_t thr[2];

  pthread_attr_init (& attr);
  if ((err = pthread_attr_setschedpolicy(& attr,
                                         SCHED_FIFO)) != 0) {
    fprintf(stderr,"setschedpolicy: %s\n",
                    strerror(err));
    exit(EXIT_FAILURE);
  }
  if ((err = pthread_attr_setinheritsched(& attr,
                          PTHREAD_EXPLICIT_SCHED)) != 0) {
    fprintf(stderr, "setinheritsched: %s\n",
                      strerror(err));
    exit(EXIT_FAILURE);
  }

  param.sched_priority = 20;
  if ((err = pthread_attr_setschedparam(& attr,
                                        & param)) != 0) {
    fprintf(stderr, "setschedparam: %s\n",strerror(err));
    exit(EXIT_FAILURE);
  }
  if ((err = pthread_create(& (thr[0]), & attr,
                        thread_basse_prio, NULL)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(err));
    exit(EXIT_FAILURE);
  }
```

Copyright © 2015 Eyrolles.

```
  param.sched_priority = 80;
  if ((err = pthread_attr_setschedparam(& attr,
                                        & param)) != 0) {
    fprintf(stderr, "setschedparam: %s\n",strerror(err));
    exit(EXIT_FAILURE);
  }
  if ((err = pthread_create(& (thr[1]), & attr,
                           thread_haute_prio, NULL)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(err));
    exit(EXIT_FAILURE);
  }

  pthread_join(thr[0], NULL);
  pthread_join(thr[1], NULL);

  for (i = 0; i < NB_COMMUTATIONS_MAX; i ++)
    printf("%ld\n", durees[i]);

  return EXIT_SUCCESS;
}
```

Nous lançons le programme (avec les deux threads sur le même CPU), qui affiche les temps de commutation mesurés :

```
# taskset -pc 2 $$
# ./exemple-commutation-threads
12
13
12
13
12
13
13
13
13
12
[...]
#
```

Naturellement, les conditions qui nous intéressent pour le temps réel sont les cas extrêmes, aussi réutilisons-nous nos outils développés au chapitre 4 :

```
# ./exemple-commutation-threads > data-threads-01.txt
# ../chapitre-04/calculer-statistiques < data-threads-01.txt
 Nb mesures = 1000
 Minimum = 11
 Maximum = 53
 Moyenne = 12
 Ecart-type = 2
```

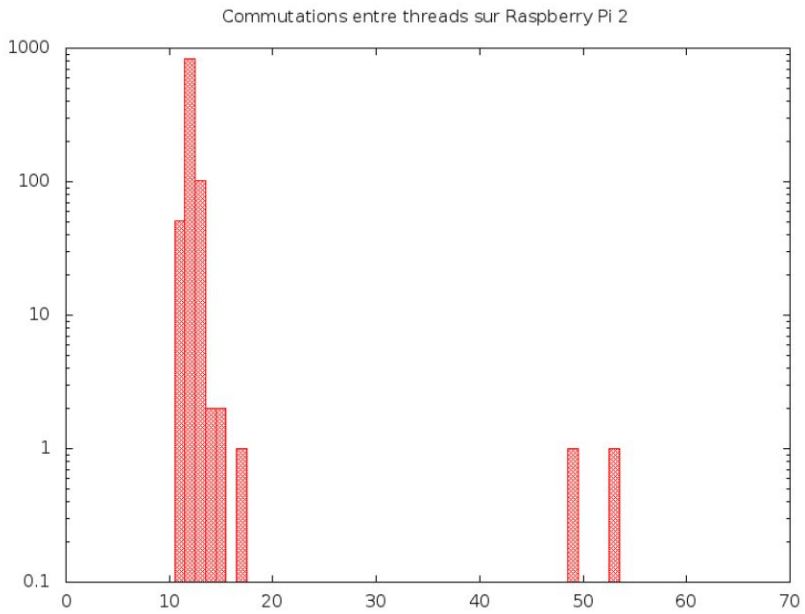
Copyright © 2015 Eyrolles.

```
# ../chapitre-04/calculer-histogramme 60 1 60 < data-threads-01.txt >
histo-threads-01.txt
# ../chapitre-04/afficher-histogramme.sh histo-threads-01.txt "Commutations entre
threads sur Raspberry Pi 2"
#
```

Les résultats sont représentés sur l'histogramme de la figure 6-3. On y remarque que les temps de commutation sont bien constants et plutôt rapides (deux commutations vers 50 microsecondes probablement dues à l'occurrence d'une interruption, une en 17 microsecondes et les 997 autres entre 11 et 15 microsecondes). Ceci est très satisfaisant, mais il faut noter que le système n'était pas particulièrement chargé, et il serait sûrement nécessaire de valider ces données sur un système cible avec une charge processeur importante.

Figure 6-3

Temps de commutation entre threads



Commutations entre processus

Comparativement, nous allons examiner les commutations entre processus au lieu des threads. Le principe est le même mais le code source est plus long car il faut utiliser :

- une zone de mémoire partagée pour stocker l'heure avant commutation ;
- une zone de mémoire partagée pour stocker le tableau des durées de commutation ;
- une zone de mémoire partagée pour le nombre de commutations mesurées ;
- un sémaphore nommé pour la synchronisation.

Cette fois, les commutations s'effectuent entre deux processus fils du programme principal. Je ne vais pas développer tout le code ici, vous le trouverez sur mon site web, de même que les codes sources des autres exemples de cet ouvrage.

Copyright © 2015 Eyrolles.

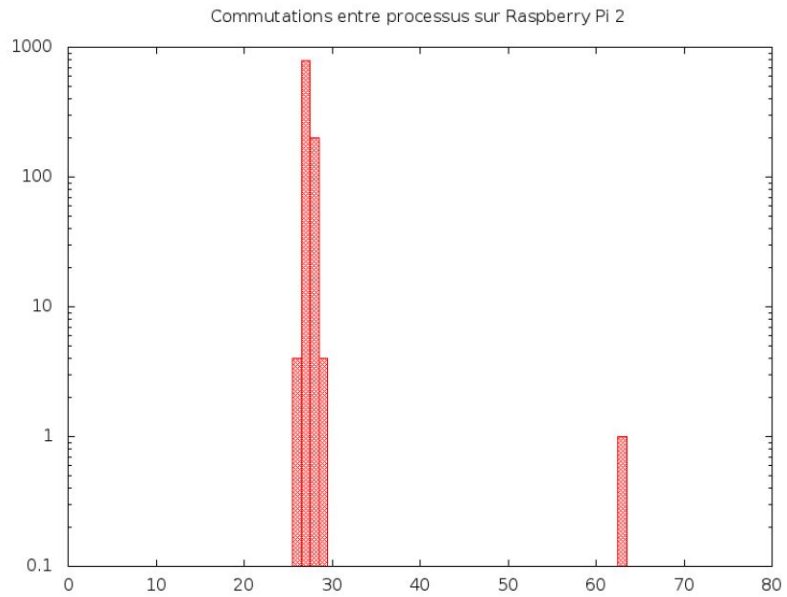
Les résultats sont les suivants :

```
# ./exemple-commutation-processus > data-process-01.txt
# ../chapitre-04/calculer-statistiques < data-process-01.txt
 Nb mesures = 1000
 Minimum = 26
 Maximum = 63
 Moyenne = 27
 Ecart-type = 1
# ../chapitre-04/calculer-histogramme 100 0 150 < data-process-01.txt > histo-
process-01.txt
# ../chapitre-04/afficher-histogramme.sh histo-process-01.txt "Commutations entre
processus sur Raspberry Pi 2"
```

Effectivement, les durées moyennes et maximales sont plus longues, comme le montre l'histogramme de la figure 6-4.

Figure 6-4

Temps de commutation entre processus



Comparaison processus et threads

Le temps de commutation entre deux processus est légèrement plus long qu'entre deux threads du même processus. Ceci est tout à fait normal, car lors de la commutation entre deux processus, il est nécessaire de modifier la configuration de la mémoire virtuelle.

Lorsqu'un processus s'exécute, il voit un espace d'adressage linéaire immense que l'on nomme mémoire virtuelle (de 4 Go sur un processeur 32 bits, à 256 To sur les processeurs 64 bits actuels en mode dit « canonique »). Cet espace est découpé en pages (classiquement 4 Ko, mais d'autres configurations existent comme 2 Mo) et chacune d'entre elles peut avoir une correspondance ou

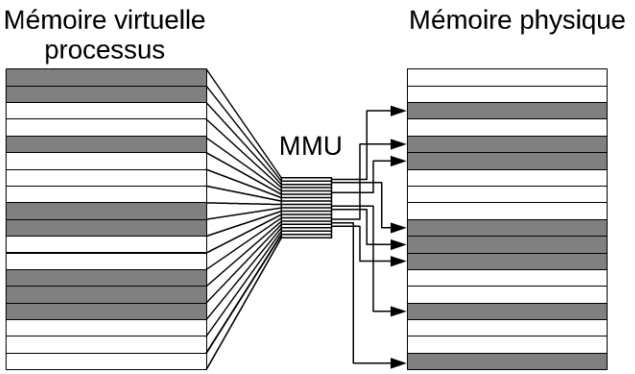
Copyright © 2015 Eyrolles.

non en mémoire physique. Cette correspondance est assurée par une sous-partie du processeur nommée MMU *(Memory Managment Unit)*, qui peut être imaginée comme une sorte de table indiquant pour chaque page de mémoire virtuelle le numéro de la page de mémoire physique correspondant, ou un repère précisant que cette page virtuelle n'a pas de correspondance en mémoire physique. Toute tentative de lecture ou d'écriture dans une telle page déclencherait une exception « faute de page » qui avertirait le noyau d'un accès mémoire illégal (se soldant la plupart du temps par un signal fatal pour le processus, mais aussi parfois par de l'allocation mémoire à la volée).

Chaque processus dispose de sa propre configuration pour la table MMU. Sauf cas particulier (mémoire partagée), ces configurations sont organisées pour garantir qu'aucune page de mémoire physique ne risque d'être modifiée par deux processus distincts. C'est ce qui assure l'étanchéité de la mémoire d'un processus.

Figure 6-5

Mémoire virtuelle d'un processus



Lorsque l'ordonnanceur commute entre deux tâches se trouvant dans deux espaces mémoire (deux processus) distincts, il doit reconfigurer le contenu de la MMU. Or, cette modification de la configuration de la MMU a un coût non négligeable, principalement en termes de cache mémoire.

À l'inverse, deux threads d'un même processus partagent par définition la même mémoire virtuelle. La seule différence dans la configuration de leur espace mémoire est que chaque thread dispose d'une pile personnalisée pour stocker l'adresse de retour lorsqu'il entre dans une fonction, les paramètres de la fonction et pour allouer les variables automatiques (locales).

La commutation entre deux threads d'un même processus est donc beaucoup plus rapide puisqu'elle ne sollicite que les registres du processeur (dont le pointeur de pile).

Lorsque l'on porte sous Linux des projets ayant été conçus initialement pour des systèmes temps réel classiques (VxWorks, VRTX, pSOS, etc.), on est souvent amené à développer un gros processus qui contient tout le projet, en répartissant toutes les tâches sous forme de threads. Ceci présente l'avantage de projeter les concepts (mémoire partagée, mutex, etc.) que l'on trouve sur ces systèmes embarqués, qui n'ont généralement pas de notion de mémoire virtuelle. En outre, la commutation entre threads est une opération efficace et dont la durée est plus déterministe que celle entre processus.

Copyright © 2015 Eyrolles.

Un inconvénient toutefois : si un thread commet une erreur fatale liée à un bogue (accès mémoire, division par zéro, etc.), c'est tout le processus – l'ensemble des threads – qui est interrompu de façon abrupte...

Imprévisibilités dues à la mémoire virtuelle

La gestion de la mémoire virtuelle par Linux est une mécanique efficace et perfectionnée, dont le fonctionnement est régulièrement retravaillé au gré des évolutions du noyau.

Nous avons mentionné le cas classique où l'accès à une page de mémoire virtuelle notée comme absente dans la MMU déclenche une exception « faute de page » suivie d'un signal `SIGSEGV` tuant le processus. Ceci n'est vrai que lorsque l'accès mémoire était véritablement illégal. Mais il existe de nombreux autres cas où la faute de page est liée à une opération tout à fait valide et où le comportement du noyau s'avère totalement différent. Voyons quelques exemples simples :

- Lorsque le kernel doit démarrer l'exécution d'un nouveau programme – *via* l'appel système `execve()` – il ne charge pas immédiatement en mémoire tout le code exécutable et les bibliothèques, ceci formant parfois un ensemble très volumineux de plusieurs dizaines de Mo. Le noyau projette ces fichiers dans la mémoire virtuelle du processus. Les pages sont toujours notées absentes dans la MMU, mais il existe une table dans le kernel qui indique qu'elles devront à terme contenir le code se trouvant dans le fichier exécutable ou les bibliothèques externes. Dès que le processus essaye d'entrer dans une routine située dans une page absente, l'exception « faute de page » conduit à l'allocation d'une page en mémoire physique et au chargement depuis le disque du code nécessaire. Au retour de l'exception, le processus continue son déroulement tout à fait normalement.
- Chaque processus – et même chaque thread – utilise une pile pour stocker les paramètres, variables locales (automatiques) et adresses de retour des fonctions. La pile n'est pas allouée de manière statique au début du processus, mais elle s'étend à la demande au cours de son exécution. Lorsque l'empilement d'une nouvelle donnée provoque le passage à une nouvelle page, absente jusqu'alors dans la MMU, le noyau alloue dynamiquement une page de mémoire physique (en vérifiant si la limite de taille de pile n'est pas atteinte) et l'affecte au processus.
- Le principe du *swap* est généralement assez bien connu : lorsque le noyau doit libérer de la mémoire physique de manière urgente, il sélectionne des pages peu utilisées d'un processus et les transfère sur le disque (dans une partition ou un fichier dédiés). Ensuite, les pages correspondantes de la mémoire virtuelle de ce processus sont notées absentes. Si le processus tente à nouveau d'accéder à ces données *swappées*, le kernel profitera de l'exception « faute de page » pour les recharger en mémoire physique (quitte à en *swapper* d'autres). Aujourd'hui, la quantité de mémoire disponible étant beaucoup plus grande que sur les premières versions de Linux, le rôle du *swap* n'est plus aussi crucial. Dans les systèmes embarqués et temps réel, on ne configure généralement aucune zone de *swap*.

Il existe encore de nombreuses autres situations où la mémoire virtuelle met à contribution l'exception « faute de page » pour réaliser des opérations dynamiquement (allocation différée lors d'un `malloc()`, projection d'une zone partagée avec `mmap()`, etc.).

Copyright © 2015 Eyrolles.

Cette méthode de gestion avancée de la mémoire, en s'appuyant sur l'exception « faute de page », est très performante et fonctionne parfaitement pour un poste de travail ou un serveur. Mais elle peut s'avérer très gênante dans le cas d'une application temps réel.

Imaginez que notre processus vient à peine de démarrer. Il a basculé sous ordonnancement temps réel et invoque une fonction de traitement critique. Toutefois, cette fonction peut se trouver sur une page virtuelle qui n'est pas encore présente en mémoire physique. C'est lors de l'exception « faute de page » que le noyau va lancer le chargement depuis le fichier exécutable. Le temps de démarrage de notre routine critique vient subitement de changer d'ordre de grandeur pour passer de la microseconde à la milliseconde !

Le problème n'est pas tant la durée de chargement du code exécutable (qu'il faut bien réaliser à un moment ou à un autre), mais l'imprévisibilité de la situation, due notamment au comportement des autres processus et au niveau de saturation de la mémoire physique.

Un second phénomène peut contrarier la prédictibilité du temps d'accès à la mémoire : lorsqu'un processus crée un fils en appelant `fork()`, le noyau est censé réaliser une copie de la mémoire du processus père. Toutefois, pour améliorer l'efficacité du système, le kernel n'effectue pas de copie de la mémoire physique. Les pages de mémoire virtuelle des deux processus, père et fils, pointent vers les mêmes pages de mémoire physique, mais elles sont alors verrouillées en lecture seule dans la MMU. Tant que les deux processus ne font que de la consultation de ces pages (code exécutable, tables de paramètres constants, etc.), il n'y a aucune duplication de mémoire.

Si l'un des deux processus tente de modifier une donnée, la MMU voit un conflit avec la configuration lecture seule de la page concernée et lève une exception « faute de page ». Le kernel s'aperçoit que l'exception s'est produite lors d'un accès à une page qu'il avait notée comme *Copy-On-Write* (copie à l'écriture) et va réaliser seulement à ce moment la duplication de la page de mémoire physique, puis l'attribution de chacune des copies à l'un des processus et enfin le déverrouillage pour en autoriser l'accès lecture écriture.

Encore une fois, ce mécanisme très efficace peut perturber le déterminisme de l'exécution d'une application temps réel. Pourtant, le noyau Linux nous propose une solution :

```
#include <sys/mman.h>
int mlockall (int flags);
int munlockall (void);
```

L'appel système `mlockall()` charge et verrouille en mémoire physique toutes les pages de mémoire virtuelle qui y sont déjà (si `flags` contient `MCL_CURRENT`), ainsi que celles qui devraient être chargées ultérieurement (si `flags` contient `MCL_FUTURE`) par le mécanisme de la copie à l'écriture ou du chargement à la demande du code exécutable.

L'appel `mlockall()` peut échouer si la quantité de mémoire disponible n'est pas suffisante, ou si les limites fixées pour le processus ont été atteintes.

Je conseille de l'employer systématiquement dans les processus temps réel, afin d'éviter les retards imprévisibles dus à la mémoire virtuelle. Un petit souci se pose toutefois pour garantir l'allocation de la pile du processus. J'utilise généralement une fonction d'initialisation qui va réserver la taille voulue dans la pile et appeler `mlockall()` :

Copyright © 2015 Eyrolles.

```
#define TAILLE_PILE_VOULE   32768

void verrouiller_memoire ()
{
    unsigned char tableau [TAILLE_PILE_VOULUE];
    memset(tableau, 0, taille_pile_voulue);
    if (mlockall(MCL_CURRENT | MCL_FUTURE) != 0) {
        perror("mlockall()");
        exit(EXIT_FAILURE);
    }
}
```

Cette fonction pourra être appelée après passage en temps réel, avant d'entamer les opérations critiques.

Nous verrons dans les chapitres ultérieurs que cette opération de verrouillage en mémoire physique, qui est importante pour le temps réel souple, pourra être indispensable pour les extensions temps réel strictes.

Préemptibilité du noyau

Il existe dans l'univers des systèmes d'exploitation temps réel une notion parfois mal comprise mais très importante : celle de préemptibilité du noyau.

Avant tout, une précision de vocabulaire : Linux a toujours été – et demeure – un système d'exploitation à ordonnancement *préemptif*. Le noyau est capable à tout moment de préempter une tâche pour exécuter un traitement urgent (sur demande d'interruption, par exemple) ou commuter sur une autre tâche plus prioritaire qui vient de se réveiller. Nous l'avions observé sur la figure 3-2 du chapitre 3.

Depuis le noyau 2.6, une option a été intégrée permettant de rendre le noyau *préemptible*, c'est-à-dire susceptible d'être lui-même préempté par une tâche urgente.

Principes

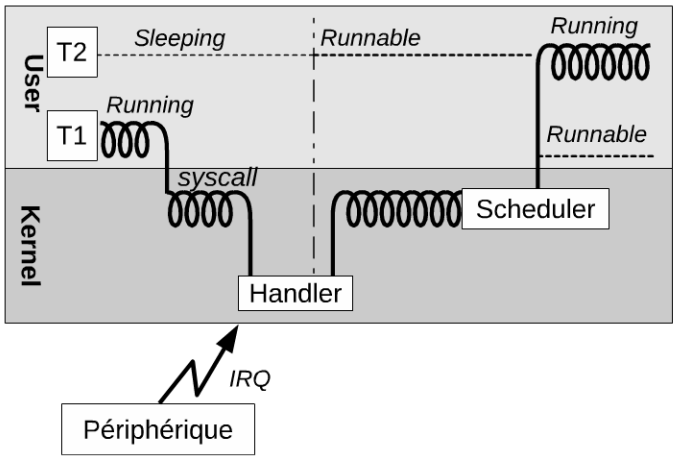
Prenons d'abord l'exemple d'un noyau non préemptible et reportons-nous à la figure 6-6. Nous observons deux tâches : T1 (de faible priorité) et T2 (de haute priorité).

T2 est endormie en attente d'un événement extérieur, alors que T1 est active. Nous savons que si T2 devient prête alors que T1 s'exécute en espace utilisateur, T1 sera immédiatement préemptée (voir chapitre 3, figure 3-2).

Copyright © 2015 Eyrolles.

Figure 6-6

Noyau non préemptible



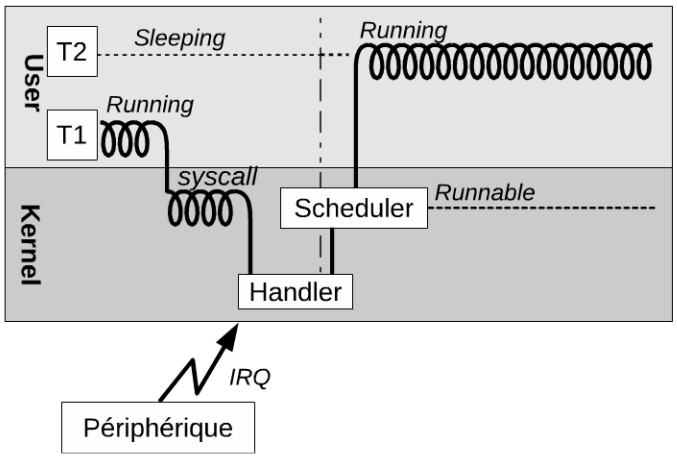
Ici, T1 va effectuer un appel système et entrer ainsi dans le noyau. Supposons qu'il s'agisse d'un appel système relativement long (nécessitant de réorganiser des tables internes, par exemple), mais durant lequel la tâche T1 n'est pas endormie.

À présent, une interruption survient, indiquant que l'événement attendu par T2 s'est produit. Le handler gérant cette interruption fait passer T2 à l'état prête *(Runnable)*. Toutefois, au retour de ce handler, c'est l'appel système s'exécutant pour le compte de T1 qui doit d'abord se terminer. Puis, au retour dans l'espace utilisateur, l'ordonnanceur est invoqué et la commutation se fait vers T2 qui devient *Running* tandis que T1 est *Runnable*.

Voyons à présent le comportement d'un noyau préemptible face à la même séquence d'événements, comme représenté sur la figure 6-7.

Figure 6-7

Noyau préemptible



T1 débute toujours *Running* et T2 *Sleeping* ; T1 invoque un appel système durant lequel une interruption se produit. Le handler fait passer T2 à l'état *Runnable*.

Copyright © 2015 Eyrolles.

Cette fois, c'est au retour du handler de l'interruption que l'ordonnanceur est invoqué. Aussi peut-on commuter tout de suite sur T2, en laissant T1 *Runnable* dans l'espace kernel. Ce n'est que lorsque T2 s'endormira volontairement que l'appel système suspendu pourra continuer et T1 reprendre son exécution.

Ce second comportement – celui d'un noyau préemptible – est beaucoup plus adapté pour un système temps réel. En effet, nous avons un événement externe (l'interruption) et une action en réponse à ce résultat (l'exécution de T2). Il est manifeste en comparant les deux figures que dans le second cas, la réponse à l'événement arrive plus rapidement que dans le premier cas. Toutefois, ce n'est pas la notion de rapidité qui compte pour une application temps réel mais celle de prédictibilité.

Dans le premier cas, nous sommes incapables de donner une limite au temps s'écoulant entre le déclenchement de l'événement et le début de la réponse, car ce temps ne dépend ni de l'application temps réel (T2) ni du noyau lui-même (durée du handler), mais du comportement d'une autre tâche totalement indépendante. On appelle ce genre de situation, dans laquelle une tâche temps réel est soumise à l'exécution d'une tâche de moindre priorité, une « inversion de priorité ». Nous reviendrons sur ce sujet dans le prochain chapitre.

Au contraire, dans la seconde situation, nous pouvons mesurer la latence, c'est-à-dire le temps s'écoulant entre l'arrivée de l'interruption et le réveil de la tâche T2. Ensuite, sachant que cette durée ne dépend que du handler et de l'ordonnanceur, nous pouvons estimer que la latence restera constante par la suite.

Il faut toutefois reconnaître que la préemptibilité alourdit quelque peu le noyau.

- Tout le code des appels système doit être réentrant, même sur un système uniprocesseur : la tâche T2 de notre exemple peut en effet invoquer le même appel système que celui dans lequel T1 est en attente et utiliser des variables globales ou statiques qu'il faut protéger contre les accès concurrents.
- L'appel de l'ordonnanceur doit se faire à chaque retour d'interruption et non plus à chaque retour dans l'espace utilisateur.
- Pour maintenir une tâche en attente dans le code du kernel, et enregistrer son état (registres processeur, etc.), il faut qu'elle dispose d'une pile personnelle dans l'espace noyau en plus de sa pile dans l'espace utilisateur.

Préemptibilité du noyau standard

Les noyaux Linux antérieurs au 2.6 n'étaient pas préemptibles. Pour obtenir ce comportement, il fallait intégrer un patch spécifique. Le support a été ajouté dans la branche stable 2.6, avec une option de compilation spécifique dans le menu Processor Type and Features de configuration du noyau. Cette option est nommée Preemption Model et peut prendre trois valeurs.

- Preemptible kernel : pour les systèmes devant réagir de manière prévisible aux interruptions, avec un comportement temps réel souple.
- No forced preemption : une large majorité des utilisateurs de Linux ne tire aucun profit de la préemptibilité du noyau. Les serveurs web, les postes de calcul, les bases de données ne réclament pas de latence limitée pour répondre aux interruptions, mais préfèrent optimiser la

Copyright © 2015 Eyrolles.

puissance de traitement du processeur en le déchargeant des opérations inutiles. Pour ceux-là, l'option No forced preemption reprend le comportement d'un noyau non préemptible.

- Voluntary kernel preemption : il s'agit d'une option un peu « tiède », avec laquelle le noyau n'est pas totalement préemptible, mais où des points de préemption sont insérés explicitement autour des fonctions potentiellement longues invoquées depuis les appels système. C'est souvent la configuration adoptée par les distributions ; elle est bien adaptée aux postes de travail avec quelques périphériques externes.

Vous pouvez voir les points de préemption explicites dans le code du noyau Linux, il s'agit de portions comme :

```
if (need_resched()) {
    // nettoyage
    [...]
    // appel de l'ordonnanceur
    schedule();
}
```

Connaître la configuration d'un noyau

Lorsqu'un noyau est compilé et livré à un utilisateur, ce dernier devrait en principe disposer du fichier `.config` décrivant la configuration du kernel. Les distributions proposent généralement ce fichier dans le répertoire `/boot`. Pour savoir quelle option a été choisie en ce qui concerne la préemptibilité, il suffit donc de rechercher les options `CONFIG_PREEMPT_xxx` dans ce fichier :

```
$ grep CONFIG_PREEMPT /boot/config-3.0.0-12-generic
# CONFIG_PREEMPT_RCU is not set
CONFIG_PREEMPT_NOTIFIERS=y
# CONFIG_PREEMPT_NONE is not set
CONFIG_PREEMPT_VOLUNTARY=y
# CONFIG_PREEMPT is not set
$
```

Ici, nous voyons que l'option `VOLUNTARY` est activée, comme sur la plupart des distributions classiques. La commande `uname -a` nous confirme qu'il s'agit d'un noyau compilé pour la distribution Ubuntu.

```
$ uname -a
Linux SRVR-Logilin 3.0.0-12-generic #20-Ubuntu SMP Fri Oct 7 14:50:42 UTC 2011 i686
➥  i686 i386 GNU/Linux
[~]$
```

Sur un système embarqué, le fichier de configuration du noyau n'est pas toujours fourni. Toutefois, il est possible d'intégrer son contenu directement dans la mémoire du noyau, et d'y donner accès *via* le pseudofichier `/proc/config.gz`. En voici un exemple sur une carte à processeur Arm.

Copyright © 2015 Eyrolles.

```
$ ssh root@Pandaboard
root@Pandaboard's password:
[Panda] # zcat /proc/config.gz | grep CONFIG_PREEMPT
CONFIG_PREEMPT_RCU=y
# CONFIG_PREEMPT_NONE is not set
# CONFIG_PREEMPT_VOLUNTARY is not set
CONFIG_PREEMPT=y
[Panda] #
```

Cette fois, l'option `PREEMPT` est activée, ce qui apparaît également dans le résultat de :

```
[Panda] # uname -a
Linux Pandaboard 3.0.0-rc7-cpb #1 SMP PREEMPT Thu Sep 29 14:49:25 CEST 2011 armv7l
➥  GNU/Linux
[Panda] #
```

Expériences sur la préemptibilité

Il n'est pas très simple de mettre en évidence la différence de comportement entre un noyau préemptible et un noyau non préemptible car elle concerne précisément des cas rares et difficiles à reproduire.

Pour cela, nous allons nous intéresser à une interruption déclenchée par le port de communication série RS-232. Nous allons envoyer un caractère sur une liaison série à destination d'un système Linux sur lequel fonctionnera un processus temps réel qui renverra le même caractère dans l'autre sens sur la même liaison série.

L'expérience suivante met en œuvre une carte Raspberry Pi 2 fonctionnant avec un noyau Linux 3.18. Ce choix s'explique par le fait que le Raspberry Pi dispose d'un véritable contrôleur série RS-232 possédant sa propre interruption – et non d'une émulation basée sur un contrôleur USB comme la plupart des PC actuels.

Nous essaierons de mesurer précisément le temps de réponse de notre processus temps réel et de voir s'il varie sous l'influence d'un processus de moindre priorité.

Pour effectuer cette mesure, j'utiliserai un microcontrôleur Texas Instruments MSP 430. Ce dernier fonctionne sans système d'exploitation et nous permet un contrôle direct et sans perturbation de la liaison série. Le microcontrôleur MSP430 est simple et bon marché. Je me servirai d'une petite carte de développement nommée Launchpad qui permet, pour un coût modique, de programmer et de faire fonctionner le microcontrôleur en accédant facilement à ses broches d'entrées-sorties. Le microcontrôleur enverra au Raspberry Pi un caractère et mesurera le temps écoulé jusqu'à la réception de la réponse.

Voici le programme qui est compilé, puis chargé dans le microcontrôleur (à l'aide des utilitaires `msp430-gcc` et `mspdebug` qui sont librement disponibles sous Linux). Mentionnons pour le lecteur non habitué à la programmation de microcontrôleur que les pseudo-variables `WDCTL`, `P1DIR`, `P1OUT`, `P1REN`, `DCOCTL`, `BCSCTL1`, `P1SEL`, `P1SEL2`, `UCA0CTL0`, `UCA0CTL1`, `UCA0BR0`, `UCA0BR1`, `UCA0MCTL`, `IE2`, `IFG2` `UCA0TXBUF` et `UCA0RXBUF` représentent des accès directs aux registres du microcontrôleur. Pour avoir plus d'informations sur leurs rôles, se reporter au « MSP430x2xx User Guide (slau144) ».

Copyright © 2015 Eyrolles.

```
uart-echo-timing.c  :

#include <intrinsics.h>
#include <msp430g2553.h>

void send_char(char c);
void send_long_int(long l);
void delay_ms(unsigned int ms);

int main(void)
{
    long int counter;

    // Stop the watchdog.
    WDTCTL  = WDTPW + WDTHOLD;

    // Leds P1.0 et P1.6 to output, turned off.
    P1DIR |=  BIT0 | BIT6;
    P1OUT &= ~(BIT0 | BIT6);

    // P1.3 button on input with pull-up resistor.
    P1DIR &= ~BIT3;
    P1REN |=  BIT3;
    P1OUT |=  BIT3;

    // Select a 16MHz Basic clock.
    DCOCTL  = 0;
    BCSCTL1 = CALBC1_16MHZ;
    DCOCTL  = CALDCO_16MHZ;

    // Select secondary function for P1.1 (Rx)  and P1.2 (Tx)
    P1SEL   = BIT1 | BIT2;
    P1SEL2  = BIT1 | BIT2;

    // UART, 8bits, no parity, 1 stop,
    UCA0CTL0 = 0;

    // See MSP430x2xx User Guide (slau144j) p.424:
    // BRCLK = 16MHz
    // Baud rate = 115200
    //  -> UCBRx  = 137
    //  -> UCBRSx = 7
    //  -> UCBRFx = 0
    UCA0CTL1 |= UCSSEL_2;
    UCA0BR0  = 138;
    UCA0BR1  = 0;
    UCA0MCTL = UCBRS2 + UCBRS1 + UCBRS0;
    UCA0CTL1 &= ~UCSWRST; // USCI Reset.

    IE2 |= UCA0TXIE | UCA0RXIE;
```

Copyright © 2015 Eyrolles.

```
    // Green led on.
    P1OUT |= BIT6;
    // Wait for button press.
    while ((P1IN & BIT3) != 0)
        ;
    // Green led off.
    P1OUT &= ~BIT6;

    while (1) {
        counter = 0;
        // Red led on.
        P1OUT |= BIT0;
        // Send a request.
        send_char('A');
        // Wait for an answer.
        while(!(IFG2 & UCA0RXIFG)) {
            counter ++;
        }
        // Red led off.
        P1OUT &= ~BIT0;
        if (UCA0RXBUF != 'A')
            continue;
        // Send the timing.
        send_long_int(counter);
        send_char('\r');
        send_char('\n');
        delay_ms(10);
    }
    return 0;
}

void delay_ms(unsigned int ms)
{
    while ((ms--) > 0)
        __delay_cycles(16000);
}

void send_char(char c)
{
    // Wait for Tx empty.
    while(!(IFG2 & UCA0TXIFG))
        ;
    UCA0TXBUF = c;
}

void send_long_int(long l)
{
    if (l > 10)
        send_long_int(l / 10);
    send_char((l%10) + '0');
}
```

Copyright © 2015 Eyrolles.

On peut voir que ce programme, après une pression sur le bouton Start, envoie un octet (caractère `'A'`) sur le port série et boucle – en incrémentant une variable – tant qu'il n'a pas reçu de réponse. La durée de la boucle importe peu. Chaque itération dure environ 1 microseconde, et on pourrait la calibrer précisément mais cela ne présente pas de véritable intérêt. Ce n'est pas le nombre absolu de boucles effectuées qui nous concerne, mais plutôt les variations de ce nombre au cours des essais successifs.

Une fois la réponse obtenue, nous envoyons sur le même port série le nombre de boucles effectuées, et le travail reprend inlassablement.

Le programme qui fonctionne sur le Raspberry Pi 2 sous Linux est le suivant :

exemple-reponse-irq-serie.c :

```
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>

volatile int quitter = 0;

void handler_sigint(int unused)
{
    quitter = 1;
}

int main(int argc, char * argv[])
{
    int fd;
    unsigned char  octet;
    int nb_cycles;
    struct termios parametres;
    struct termios original;

    if (argc != 2) {
        fprintf(stderr, "usage: %s port_serie\n",argv[0]);
        exit(EXIT_FAILURE);
    }

    signal(SIGINT, handler_sigint);

    if ((fd = open(argv[1], O_RDWR | O_NONBLOCK))<0){
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    tcgetattr(fd, & original);
    tcgetattr(fd, & parametres);
    cfmakeraw(& parametres);
    cfsetispeed(& parametres, B115200);
```

Copyright © 2015 Eyrolles.

```
    cfsetospeed(& parametres, B115200);
    parametres.c_iflag |= IGNPAR;
    parametres.c_cflag &= ~(CSIZE | PARENB | CSTOPB);
    parametres.c_cflag |= CS8 | CLOCAL;
    if (tcsetattr(fd, TCSANOW, & parametres) != 0) {
        perror("tcsetattr");
        exit(EXIT_FAILURE);
    }
    parametres.c_cc[VMIN]  = 1;
    parametres.c_cc[VTIME] = 0;
     fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) & ~O_NONBLOCK);

    while (! quitter) {
        if (read(fd, & octet, 1) < 0) {
            perror(argv[1]);
            exit(EXIT_FAILURE);
        }
        if (octet!= 'A')
            continue;
        if (write(fd, & octet, 1) < 0) {
            perror(argv[1]);
            exit(EXIT_FAILURE);
        }
        while (octet != '\n') {
            read(fd, &octet, 1);
           write(STDOUT_FILENO, &octet, 1);
        }
    }
    tcsetattr(fd, TCSANOW, & original);
    close(fd);
    fprintf(stderr, "Bye !\n");
    return EXIT_SUCCESS;
}
```

Le programme attend donc un caractère, le renvoie en écho, récupère les caractères représentant son temps de réponse et les affiche sur sa sortie standard.

Voici un exemple d'exécution :

```
# ./exemple-reponse-irq-serie /dev/ttyAMA0
5938
5908
5918
5915
5923
5923
5926
5924
5923
5926
```

Copyright © 2015 Eyrolles.

```
5927
5920
5922
 [...]
    (Contrôle-C)
Bye !
#
```

Pour perturber le fonctionnement du programme, j'ai réalisé un petit script shell qui charge (avec `insmod`) et décharge (avec `rmmod`) toutes les cinq secondes ce petit module du kernel :

```
module-delay.c  :
#include <linux/module.h>
#include <linux/delay.h>

static int __init module_delay_init (void)
{
    mdelay(100);
    return 0;
}

static void __exit module_delay_exit (void)
{}

module_init(module_delay_init);
module_exit(module_delay_exit);
MODULE_LICENSE("GPL");
```

Ce module effectue une boucle active de 100 millisecondes dès son chargement dans le kernel. Ceci nous permet de disposer d'un appel système suffisamment long pour perturber le temps de réponse à une interruption sur un système non préemptible.

Comme nous en avons l'habitude, étudions les résultats en les redirigeant dans un fichier et utilisons les outils développés au chapitre 4. Tout d'abord, un essai avec un noyau non préemptible :

```
# uname -a
Linux raspberrypi 3.18.16-cpb-no-preempt #1 SMP Fri Jul 17 09:11:35 CEST 2015 armv7l
GNU/Linux
# taskset -pc 1 $$
pid 3222's current affinity list: 0-3
pid 3222's new affinity list: 0-3
#  chrt -f 90 ./exemple-reponse-irq-serie /dev/ttyAMA0 > /run/resultats-non-preempt.
txt
   (Contrôle-C après quelques minutes)
Bye!
#
```

Copyright © 2015 Eyrolles.

J'ai choisi d'enregistrer les résultats des mesures dans un fichier placé dans le répertoire `/run` (système de fichiers `tmpfs` en mémoire RAM), afin d'éviter les perturbations apportées par les écritures sur la carte SD.

Pendant le déroulement du programme, mon script qui charge et décharge le module perturbateur toutes les cinq secondes s'exécute sur le même processeur, avec une priorité temps partagée.

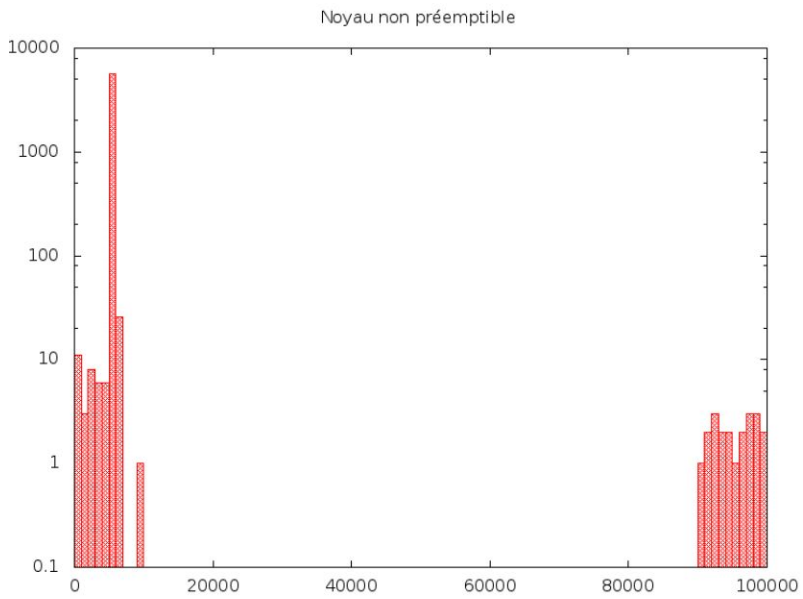
L'analyse du résultat donne :

```
# ../chapitre-04/calculer-statistiques < resultats-non-preempt.txt
 Nb mesures = 5748
 Minimum = 528
 Maximum = 99311
 Moyenne = 6234
 Ecart-type = 5406
# ../chapitre-04/calculer-histogramme 100 0 100000  < resultats-non-preempt.txt
➥  histo-non-preempt.txt
```

Les résultats statistiques paraissent assez catastrophiques : une variabilité entre 500 et 100 000 itérations de boucles, et un écart-type approchant la moyenne des valeurs ! Graphiquement, les résultats sont plus compréhensibles, comme nous le voyons sur la figure 6-8.

Figure 6-8

Réponse en interruption sur noyau non préemptible



Notre système fait un grand écart entre un temps de réponse faible (de l'ordre de 5 000 itérations par boucle, environ 5 millisecondes) et une réponse très retardée par l'appel système perturbateur (100 000 itérations par boucle, à peu près 100 ms).

Voici à présent la réponse avec le même noyau compilé avec l'option préemptible :

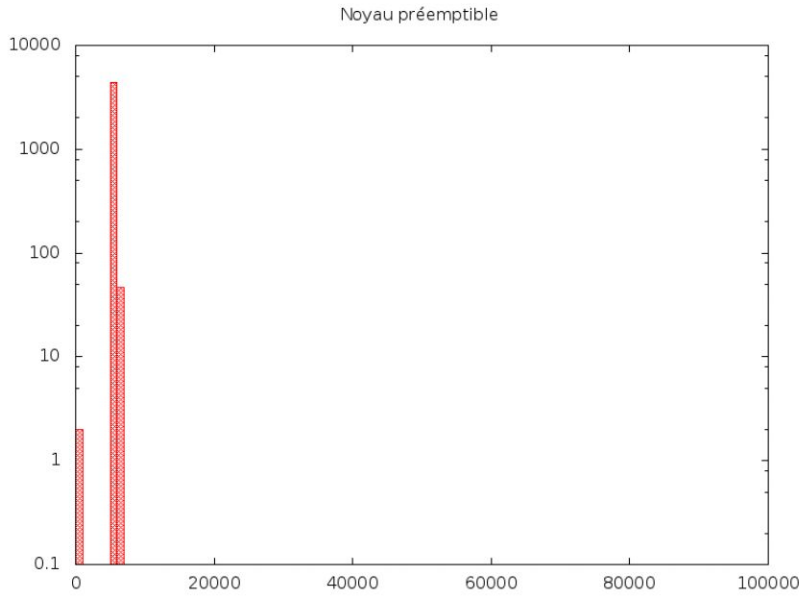
Copyright © 2015 Eyrolles.

```
# uname -a
Linux raspberrypi 3.18.16-cpb-preempt #1 SMP PREEMPT Fri Jul 17 13:19:13 CEST 2015
armv7l GNU/Linux
# chrt -f 90 ./exemple-reponse-irq-serie /dev/ttyAMA0 > /run/resultats-preempt.txt
# ../chapitre-04/calculer-statistiques < resultats-preempt.txt
 Nb mesures = 4460
 Minimum = 534
 Maximum = 6167
 Moyenne = 5928
 Ecart-type = 115
#
```

Voilà qui est beaucoup mieux ! Le résultat est visible sur la figure 6-9 avec la même échelle que précédemment. Un zoom au début de l'axe des abscisses est représenté sur la figure 6-10.

Figure 6-9

Réponse en interruption sur noyau préemptible (1)

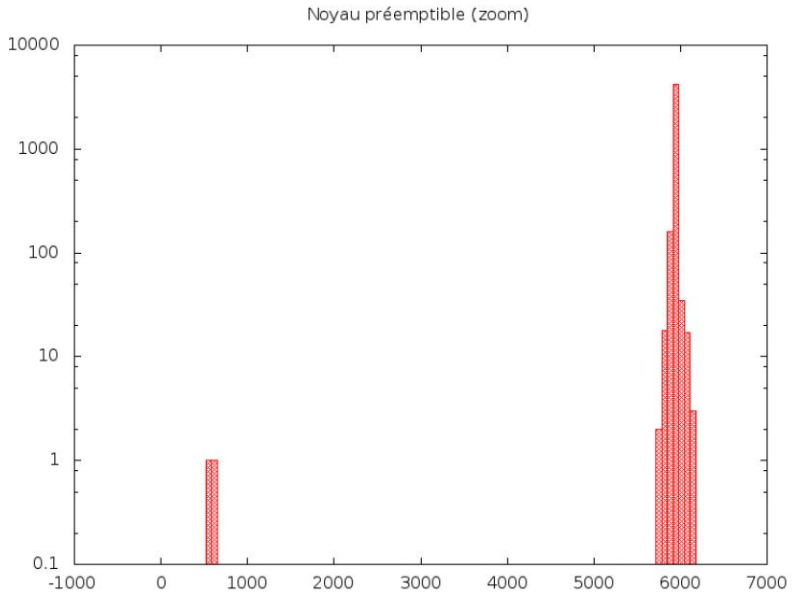


Cette fois, aucune préemption du processus de haute priorité, la réponse à l'interruption est de l'ordre de 5 millisecondes au maximum. La réponse n'est pas réellement plus rapide (elle peut même être très légèrement plus lente en raison de la complexité ajoutée au code noyau), mais elle est plus fiable et prévisible.

Copyright © 2015 Eyrolles.

Figure 6-10

Réponse en interruption sur noyau préemptible (2)



Nous avons indiqué précédemment que la préemptibilité du noyau impose aux appels système d'être réentrants. Ceci signifie que l'appel système laissé en attente pendant l'activation d'une autre tâche est susceptible d'être réinvoqué par cette dernière. Aussi, tout accès à des variables globales ou statiques, par exemple, doit être protégé.

Dans le kernel Linux, plusieurs mécanismes cohabitent pour la protection des données partagées. Les *spinlocks* sont les plus utilisés car ils permettent d'éviter les cas de concurrences entre appels système et gestionnaires d'interruptions quelle que soit la configuration du système (uniprocesseur ou multiprocesseur, préemptible ou non).

Ceci implique que les portions de code durant lesquelles un spinlock a été verrouillé ne sont pas préemptibles. Certes, ces moments doivent être les plus brefs possibles, mais il n'en demeure pas moins que les réponses à certaines interruptions peuvent être légèrement retardées à cause d'un spinlock tenu par une tâche de plus faible priorité. Et ceci, même si le spinlock en question protège une variable qui n'a aucun rapport avec l'interruption.

Pour améliorer ce comportement, le projet *PREEMPT_RT* propose un patch qui rend préemptible la plupart de ces portions de code en remplaçant les spinlocks par des mutex là où c'est possible. Nous en reparlerons au chapitre 8.

Conclusion

Les mécanismes temps réel proposés par le noyau Linux standard offrent, ainsi que nous l'avons observé, des performances intéressantes. Néanmoins, toute application temps réel peut être confrontée à certains problèmes classiques et récurrents (inversion de priorité, reprise de mutex, etc.) comme nous allons l'étudier dans le prochain chapitre.

Copyright © 2015 Eyrolles.

Points clés

- Avec un ordonnancement temps réel souple proposé par le noyau Linux standard, une tâche peut bénéficier de timers dont la période peut descendre à quelques dizaines de microsecondes avec une variabilité, un jitter, de quelques microsecondes. Ceci dépend toutefois très nettement des performances du matériel.
- La commutation entre deux processus prend quelques microsecondes. La commutation entre deux threads du même processus est légèrement plus rapide.
- Pour éviter les délais imprévisibles dus à la gestion de la mémoire virtuelle, il est important d'utiliser l'appel `mlockall()`.
- La préemptibilité optionnelle du noyau, disponible depuis sa version 2.6, est un élément important pour améliorer la fiabilité des systèmes temps réel dans leur réponse aux événements externes.

Exercices

Exercice 1 (*)

Reprenez le programme `exemple-timer-create-02.c` du chapitre 4 et exécutez-le sous un ordonnancement temps réel (avec `chrt`). Est-il plus rapide qu'en temps partagé ?

Exercice 2 (**)

Utilisez le programme `exemple-perturbateur.c` pendant l'exécution de `exemple-timer-create-02.c`. Les résultats de ce dernier sont-ils différents ? Comparez-les avec ceux obtenus au chapitre 4.

Jouez sur la priorité temps réel du processus perturbateur et observez la variabilité du timer. Quelle est la priorité maximale que l'on peut employer sans augmentation du jitter ?

Exercice 3 (**)

Déterminez la fréquence maximale avec laquelle vous pouvez faire fonctionner un timer sur votre système sans que l'écart-type par rapport aux déclenchements prévus dépasse 5 % de la période du timer.

Exercice 4 (**)

Mesurez le temps de commutation entre deux threads du même processus synchronisés sur un mutex. Recommencez la même opération sur deux processus synchronisés sur un sémaphore.

Placez les threads ou processus sur deux processeurs distincts, puis sur le même processeur. Voyez-vous une différence ?

Copyright © 2015 Eyrolles.

Exercice 5 (***)

Mesurez le temps de commutation entre deux processus en utilisant les différentes ressources IPC classiques : files de messages, tubes de communication, signaux temps réels, etc.

Le principe est toujours le même : un processus récupère l'heure système et la fournit à l'autre processus en le réveillant. Ce dernier calculera la différence et l'affichera.

Copyright © 2015 Eyrolles.

7

Problèmes temps réel classiques

Il existe quelques problèmes que l'on rencontre régulièrement avec les systèmes temps réel et auxquels la plupart des développeurs de ces systèmes sont confrontés un jour ou l'autre. Notre premier souci va consister à lancer des threads en ordonnancement Round Robin avec une priorité élevée, puis nous examinerons les problèmes d'inversion de priorités et enfin les prises de mutex.

Démarrage en Round Robin

Reprenons le principe de l'exemple `exemple-threads-rr` du chapitre 5. Nous souhaitons lancer quelques threads en parallèle sur le même CPU, avec un ordonnancement de type Round Robin et une priorité relativement élevée. Chaque thread réalisera une boucle active en notant les heures de démarrage et de terminaison.

exemple-threads-rr-01.c

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void * fonction_thread(void *);

#define NB_THREADS  5
```

Copyright © 2015 Eyrolles.

```
int main(void)
{
  int i;
  int err;
  pthread_attr_t attr;
  struct sched_param param;
  pthread_t thr[NB_THREADS];

  pthread_attr_init (& attr);
  if ((err = pthread_attr_setschedpolicy(& attr,
                              SCHED_RR)) != 0) {
    fprintf(stderr, "setschedpolicy:%s\n",strerror(err));
    exit(EXIT_FAILURE);
  }
  param.sched_priority = 80;
  if ((err = pthread_attr_setschedparam(& attr,
                                  & param)) != 0) {
    fprintf(stderr, "setschedparam:%s\n",strerror(err));
    exit(EXIT_FAILURE);
  }
  if ((err = pthread_attr_setinheritsched(& attr,
                          PTHREAD_EXPLICIT_SCHED)) != 0) {
    fprintf(stderr, "setinheritsched:%s\n",strerror(err));
    exit(EXIT_FAILURE);
  }
  for (i = 0; i < NB_THREADS; i ++) {
    if ((err = pthread_create(& (thr[i]), & attr,
                fonction_thread, (void *) i)) != 0) {
      fprintf(stderr, "create:%s\n", strerror(err));
      exit(EXIT_FAILURE);
    }
  }
  for (i = 0; i < NB_THREADS; i ++)
    pthread_join(thr[i], NULL);

  return EXIT_SUCCESS;
}


void * fonction_thread(void * arg)
{
  int numero = (int) arg;
  int i, j;
  time_t debut, fin;
  time(& debut);
  for (i = 0; i < 100000; i ++)
    for (j = 0; j < 10000; j ++)
      ;
  time(& fin);
  printf("[%d] %ld -> %ld\n",
```

Copyright © 2015 Eyrolles.

```
            numero, debut, fin);
  return NULL;
}
```

Lorsque nous exécutons notre programme, les threads affichent tous leurs heures de début et de fin (en secondes comptées depuis le 01/01/1970).

```
# taskset -pc 0 $$
pid 2470's current affinity list: 3
pid 2470's new affinity list: 0
# ./exemple-threads-rr-01
[0] 1318412845 -> 1318412848
[1] 1318412848 -> 1318412851
[2] 1318412851 -> 1318412854
[3] 1318412854 -> 1318412857
[4] 1318412857 -> 1318412860
#
```

Trouvez-vous que cela ressemble à une exécution en Round Robin ? Moi, non ! Chaque thread s'exécute pendant trois secondes, mais chacun commence uniquement après la fin du précédent. Nous avons un déroulement qui correspond beaucoup plus à un ordonnancement Fifo que Round Robin.

Le problème qui se pose est que notre thread `main()`, celui qui lance les cinq autres, est ordonnancé en temps partagé. À peine a-t-il démarré le premier thread temps réel que celui-ci le préempte, et s'exécute en entier avant même que `main()` puisse créer le second thread.

Le problème serait le même si le thread `main()` était ordonnancé en temps réel avec une priorité inférieure à celles des threads créés. Dans certains cas, on peut se permettre de placer le thread de lancement à une priorité supérieure à celles des threads à créer, mais ce n'est pas toujours possible.

Barrières Posix

Pour pallier ce genre de problème, il existe un mécanisme spécifique : les barrières. On initialise la barrière en indiquant combien de threads elle doit retenir. Ensuite, chaque nouveau thread créé vient d'abord se mettre en attente contre la barrière. Dès qu'ils sont tous présents, la barrière se lève et laisse les tâches s'exécuter en parallèle.

Pour modifier le programme précédent, il suffit d'ajouter les lignes suivantes :

```
exemple-threads-rr-02.c
[...]

pthread_barrier_t  barriere;

int main(void)
{
  [...] // Déclaration des variables
```

Copyright © 2015 Eyrolles.

```
  pthread_barrier_init(& barriere, NULL, NB_THREADS);

  [...] // Lancement des threads
}


void * fonction_thread(void * arg)
{
  int numero = (int) arg;
  int i, j;
  time_t debut, fin;
  pthread_barrier_wait(&barriere);
  time(& debut);
  [...]
```

Cette fois, le déroulement est conforme à nos attentes :

```
# ./exemple-threads-rr-02
[4] 1318413001 -> 1318413016
[0] 1318413002 -> 1318413016
[1] 1318413002 -> 1318413016
[2] 1318413003 -> 1318413016
[3] 1318413003 -> 1318413016
#
```

Inversion de priorité

Il existe un problème récurrent dans les systèmes temps réel : les inversions de priorité. Ceci se produit lorsqu'une tâche de haute priorité est en attente d'une ressource tenue par une tâche de faible priorité (jusqu'ici, tout va bien) et que cette dernière se trouve préemptée par une tâche de priorité moyenne. Tout se passe alors comme si cette tâche de priorité moyenne avait préempté celle de haut niveau.

Principe

Imaginons que nous ayons trois threads T1, T2, et T3 de priorités strictement croissantes, ainsi qu'un mutex M utilisé par certains de nos threads. Initialement, T2 et T3 sont endormis, T1 est actif et le mutex M est libre.

T1	T2	T3	M
Running	*Sleeping*	*Sleeping*	Libre
T1 demande le mutex. Ce dernier étant libre, il peut l'obtenir.			

Copyright © 2015 Eyrolles.

T1	T2	T3	M
Running	*Sleeping*	*Sleeping*	Tenu par T1
T3 se réveille sur un événement externe. Il est plus prioritaire que T1.			
Runnable	*Sleeping*	*Running*	Tenu par T1
T3 demande le mutex M. Celui-ci étant tenu par T1, T3 ne peut l'obtenir et s'endort. T1 reprend son exécution.			
Running	*Sleeping*	*Sleeping*	Tenu par T1, demandé par T3.
T2 se réveille. Il est plus prioritaire que T1 et le préempte donc.			
Runnable	*Running*	*Sleeping*	Tenu par T1, demandé par T3.

Nous sommes alors dans une situation où T3 est soumis à l'exécution de T2 qui doit relâcher le processeur pour que T1 s'exécute jusqu'à libérer le mutex ce qui réveillera enfin T3. Le fait que T3 soit soumis à l'exécution de T1 est parfaitement normal : ils partagent une ressource (le mutex) sur laquelle ils doivent synchroniser leur progression. Ce qui est gênant c'est que T3 soit soumis à T2, car ces tâches n'ont *a priori* rien en commun. La conception du système (probablement erronée) a conduit à placer T2 entre T1 et T3.

Il n'est pas très facile de mettre en évidence ce comportement. Dans l'exemple suivant, le thread T1 crée directement T2 et T3 plutôt qu'ils soient réveillés par des événements externes.

Comme d'habitude, il faut calibrer les boucles actives pour que les traitements durent quelques secondes.

```
exemple-inversion.c  :
#define _XOPEN_SOURCE  500

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_t thr1, thr2, thr3;
pthread_attr_t attr1, attr2, attr3;

void * fonction_1(void *unused);
void * fonction_2(void * unused);
void * fonction_3(void * unused);


void * fonction_1(void *unused)
{
  fprintf(stderr, "T1 demarre\n");
```

Copyright © 2015 Eyrolles.

```
  fprintf(stderr, "T1 demande le mutex\n");
  pthread_mutex_lock(& mutex);
  fprintf(stderr, "T1 tient le mutex\n");

  fprintf(stderr, "T1 cree T3\n");
  pthread_create(& thr3, &attr3, fonction_3, NULL);

  fprintf(stderr, "T1 cree T2\n");
  pthread_create(& thr2, &attr2, fonction_2, NULL);

  fprintf(stderr, "T1 lache le mutex\n");
  pthread_mutex_unlock(& mutex);
  fprintf(stderr, "T1 se termine\n");
  return NULL;
}

void * fonction_2(void * unused)
{
  int i, j;
  fprintf(stderr, "    T2 demarre\n");
  for (i = 0; i < 100000; i ++)
    for (j = 0; j < 10000; j ++)
      ;
  fprintf(stderr, "    T2 se termine\n");
  return NULL;
}

void * fonction_3(void * unused)
{
  int i, j;
  fprintf(stderr, "        T3 demarre\n");
  fprintf(stderr, "        T3 demande le mutex\n");
  pthread_mutex_lock(& mutex);
  fprintf(stderr, "        T3 tient le mutex\n");
  for (i = 0; i < 100000; i ++)
    for (j = 0; j < 10000; j ++)
      ;
  fprintf(stderr, "        T3 lache le mutex\n");
  pthread_mutex_unlock(& mutex);
  fprintf(stderr, "        T3 se termine\n");
  return NULL;
}

int main(int argc, char * argv [])
{
  struct sched_param param;
  pthread_attr_init(& attr1);
  pthread_attr_init(& attr2);
  pthread_attr_init(& attr3);
  pthread_attr_setschedpolicy(& attr1, SCHED_FIFO);
  pthread_attr_setschedpolicy(& attr2, SCHED_FIFO);
```

Copyright © 2015 Eyrolles.

```
  pthread_attr_setschedpolicy(& attr3, SCHED_FIFO);
  param.sched_priority = 10;
  pthread_attr_setschedparam(& attr1, & param);
  param.sched_priority = 20;
  pthread_attr_setschedparam(& attr2, & param);
  param.sched_priority = 30;
  pthread_attr_setschedparam(& attr3, & param);
  pthread_attr_setinheritsched(& attr1,
                          PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setinheritsched(& attr2,
                          PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setinheritsched(& attr3,
                          PTHREAD_EXPLICIT_SCHED);
  pthread_create(& thr1, & attr1, fonction_1, NULL);
  pthread_join(thr1, NULL);
  return 0;
}
```

Il faut que les trois threads se déroulent sur le même CPU. Voici un exemple d'exécution :

```
# ./exemple-inversion
T1 demarre
T1 demande le mutex
T1 tient le mutex
T1 cree T3
        T3 demarre
        T3 demande le mutex
T1 cree T2
    T2 demarre
    T2 se termine
T1 lache le mutex
        T3 tient le mutex
        T3 lache le mutex
        T3 se termine
T1 se termine
#
```

Nous voyons bien que T2 a le temps de s'exécuter entièrement avant que T3 puisse faire son travail nécessitant le mutex.

Que faire pour éviter ces situations ? Il n'y a pas de réponse absolue. La meilleure attitude est de prévoir dès la conception du système une répartition des priorités telle que l'inversion ne puisse pas se produire.

Il existe un cas célèbre, détaillé dans plusieurs articles techniques : celui de la mission *Mars Pathfinder*. Un parallèle approximatif avec le tableau précédent consisterait à considérer que la tâche T1 était une tâche de très faible priorité, chargée de collecter des données météorologiques, et T3, la supervision du bus logiciel chargée d'envoyer les données vers la station. La ressource partagée n'était pas un simple mutex, mais un sémaphore dissimulé dans l'implémentation de deux mécanismes de communication entre les tâches de la bibliothèque VxWorks.

Copyright © 2015 Eyrolles.

Le problème de préemption indirecte de T3 par des tâches de moindre priorité était détecté par le *watchdog* qui, voyant que le bus logiciel n'était pas à jour, réinitialisait le système, redémarrant ainsi le programme de travail au début de la journée. Pour plus de précisions et de détails, on se reportera à l'article [REEVES 1997].

La mission *Pathfinder* a pu être sauvée grâce à un mécanisme, nommé « héritage de priorité », que la NASA a activé à distance. Cet héritage de priorité consiste à élever momentanément la priorité d'une tâche tenant une ressource si une autre tâche plus prioritaire demande cette même ressource.

Héritage de priorité

Il existe une implémentation assez classique, nommée PIP *(Priority Inheritance Protocol)*, qui donne des valeurs de priorité aux mutex et fonctionne avec deux règles. Reprenons notre exemple précédent. Initialement, M a une priorité nulle que j'indique entre parenthèses après son état dans le tableau suivant.

T1	**T2**	**T3**	**M**
Running	*Sleeping*	*Sleeping*	Libre (0)
T1 demande le mutex. Ce dernier étant libre, il peut l'obtenir.			

La première règle de PIP intervient : « un mutex hérite de la priorité la plus élevée parmi celles de tous les threads qui le demandent ». Sa priorité passe donc à 1.

T1	T2	T3	M
Running	*Sleeping*	*Sleeping*	Tenu par T1 (1)
T3 se réveille sur un événement externe. Il est plus prioritaire que T1.			
Runnable	*Sleeping*	*Running*	Tenu par T1 (1)
T3 demande le mutex M. Celui-ci étant tenu par T1, T3 ne peut l'obtenir et s'endort. T1 reprend son exécution.			

En vertu de la première règle, cette fois la priorité du mutex monte à celle de T3. Puis, la seconde règle intervient : « un thread s'exécute avec la priorité la plus élevée parmi celles de tous les mutex qu'il tient ».

Ainsi, la priorité de T1 va brusquement monter à celle de T3.

T1	**T2**	**T3**	**M**
Running Priorité T3	*Sleeping*	*Sleeping*	Tenu par T1, demandé par T3. (3)
T2 se réveille, mais il est moins prioritaire que T1 et ne peut s'exécuter.			
Running Priorité de T3	*Runnable*	*Sleeping*	Tenu par T1, demandé par T3. (3)

Copyright © 2015 Eyrolles.

T1	T2	T3	M
T1 relâche son mutex. Il retombe à sa priorité originale, mais la libération du mutex réveille T3.			
Runnable	*Runnable*	*Running*	Tenu par T3

Ainsi, T3 va pouvoir réaliser son travail avant que T2 soit activé.

Pour activer cet héritage des priorités, il existe un attribut des mutex à configurer. On peut modifier l'exemple précédent en remplaçant l'initialisation statique :

```
exemple-inversion.c  :
[...]
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
[...]
```

par une initialisation dynamique au démarrage :

```
exemple-pip.c  :

[...]

int main(int argc, char * argv [])
{
  struct sched_param param;
  pthread_mutexattr_t mutexattr;

  pthread_mutexattr_init(& mutexattr);
  pthread_mutexattr_setprotocol(& mutexattr,
                                PTHREAD_PRIO_INHERIT);
  pthread_mutex_init(& mutex, & mutexattr);

  [...]
```

Le reste du programme est inchangé. Voyons le comportement à présent :

```
# ./exemple-pip
T1 demarre
T1 demande le mutex
T1 tient le mutex
T1 cree T3
        T3 demarre
        T3 demande le mutex
T1 cree T2
T1 lache le mutex
        T3 tient le mutex
        T3 lache le mutex
        T3 se termine
```

Copyright © 2015 Eyrolles.

```
    T2 demarre
    T2 se termine
T1 se termine
#
```

Cette fois, nous observons que le thread T3 a pu s'exécuter entièrement avant de céder la place à T2.

Il faut noter que si la plupart des systèmes temps réel actuels implémentent l'héritage de priorité, il existe des détracteurs pour ce mécanisme. Entre autres, un célèbre article de Victor Yodaiken *Against Priority Inheritance* (voir Bibliographie) présente plusieurs inconvénients de PIP.

Prise de mutex

Une autre question peut se poser quand plusieurs threads sont en attente pour tenter de prendre un mutex alors que ce dernier est verrouillé : qui va l'obtenir lorsqu'il sera libéré par son actuel détenteur ?

Effectuons un premier essai en temps partagé avec un ensemble de cinq threads en concurrence pour obtenir le même mutex.

```
exemple-prise-mutex-01.c:
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void * fonction_thread (void * arg)
{
    int numero = (int) arg;
    usleep(10000 * numero);
    while (1) {
        fprintf(stderr, "[%d] demande le mutex\n",numero);
        pthread_mutex_lock(& mutex);
        fprintf(stderr, "        [%d] tient le mutex\n",
                                numero);
        sleep(1);
        fprintf(stderr, "[%d] lache le mutex\n",numero);
        pthread_mutex_unlock(& mutex);
        usleep(10000);
    }
    return NULL;
}

#define NB_THREADS 5
int main(void)
```

Copyright © 2015 Eyrolles.

```
{
    int i;
    pthread_t thread[NB_THREADS];

    pthread_mutex_lock(& mutex);
    for (i = 0; i < NB_THREADS; i ++)
        pthread_create(& thread[i], NULL,
                       fonction_thread, (void *) (i+1));
    sleep(1);
    fprintf(stderr, "Liberation initiale du mutex\n");
    pthread_mutex_unlock(& mutex);
    for (i = 0; i < NB_THREADS; i ++)
        pthread_join(thread[i], NULL);
    return EXIT_SUCCESS;
}
```

Les sommeils dans l'exécution des threads ont plusieurs rôles : le premier `usleep()` assure que les threads commencent leurs boucles dans l'ordre de création, le `sleep()`, alors que le mutex est tenu par le thread, permet un affichage régulier et lisible des messages. Enfin, le dernier `usleep()` garantit que le thread qui vient de lâcher un mutex ne le réclame pas à nouveau instantanément. Nous nous concentrerons par la suite sur ce dernier point.

```
$ ./exemple-prise-mutex-01
[1] demande le mutex
[2] demande le mutex
[3] demande le mutex
[4] demande le mutex
[5] demande le mutex
Liberation initiale du mutex
        [1] tient le mutex
[1] lache le mutex
        [2] tient le mutex
[1] demande le mutex
[2] lache le mutex
        [3] tient le mutex
[2] demande le mutex
[3] lache le mutex
        [4] tient le mutex
[3] demande le mutex
[4] lache le mutex
        [5] tient le mutex
[4] demande le mutex
[5] lache le mutex
        [1] tient le mutex
  (Contrôle-C)
$
```

Copyright © 2015 Eyrolles.

Clairement, si l'ordonnancement est en temps partagé, lors de la libération du mutex, le système l'affecte au premier thread dans la file d'attente. Comment cela se déroule-t-il ?

L'appel `pthread_mutex_lock()` est avant tout une fonction de la bibliothèque C. Un système utilisant la bibliothèque NPTL incluse dans la GlibC, tire parti des fonctionnalités offertes par le kernel Linux depuis la version 2.6 : les Futex *(Fast User Space Mutex)*. Lorsqu'un thread essaye de verrouiller un mutex, et que ce dernier est libre, la prise du mutex s'effectue entièrement dans l'espace utilisateur (à l'intérieur de la NPTL, sans passer par le kernel). Il n'y aura d'appel au noyau – opération plus coûteuse en temps – que si le mutex est déjà verrouillé. C'est une très bonne optimisation lorsque la protection des données partagées entre les threads n'entraîne que rarement des situations de contention. Dans notre cas, toutefois, chaque appel `pthread_mutex_lock()` entre dans le noyau et le thread s'endort dans une file d'attente.

De même, l'appel `pthread_mutex_unlock()` de la NPTL peut déverrouiller un mutex en restant simplement dans l'espace utilisateur si aucun autre thread n'est en attente du même mutex. Néanmoins, dans notre exemple, chaque libération du mutex entraînera un appel système qui réveillera tous les threads en attente et les laissera « lutter » pour l'obtention du mutex lors de l'ordonnancement suivant.

Dans un ordonnancement temps partagé, il est logique que les prises de mutex de l'exemple précédent s'effectuent dans l'ordre d'arrivée initiale car tous les threads sont symétriques. Si toutefois il y avait des différences entre eux (valeur de `nice`, consommation du CPU, etc.), l'ordre ne serait plus garanti. Modifions, par exemple, la fonction des threads de manière à ce qu'un seul d'entre eux (le numéro 2) se comporte « bien » vis-à-vis de l'ordonnanceur, les autres réclamant le mutex avec insistance en bouclant autour de `pthread_mutex_trylock()`.

```
exemple-prise-mutex-02.c:
[...]
void * fonction_thread (void * arg)
{
    int numero = (int) arg;
    usleep(10000 * numero);
    while (1) {
        fprintf(stderr, "[%d] demande le mutex\n",numero);
        if (numero != 2)
            while (pthread_mutex_trylock(& mutex) != 0)
                ;
        else
            pthread_mutex_lock(& mutex);
       fprintf(stderr, "        [%d] tient le mutex\n",
                                numero);
[...]
```

À l'exécution, le comportement est très différent. Le thread numéro 2 est visiblement favorisé par l'ordonnanceur et retrouve le mutex plus souvent que les autres qui sont « punis » par le scheduler pour leurs comportements hystériques autour du `pthread_mutex_trylock()`.

Copyright © 2015 Eyrolles.

```
$ ./exemple-prise-mutex-02
[1] demande le mutex
[2] demande le mutex
[3] demande le mutex
[4] demande le mutex
[5] demande le mutex
Liberation initiale du mutex
        [2] tient le mutex
[2] lache le mutex
        [5] tient le mutex
[2] demande le mutex
[5] lache le mutex
        [2] tient le mutex
[5] demande le mutex
[2] lache le mutex
        [4] tient le mutex
[2] demande le mutex
[4] lache le mutex
        [2] tient le mutex
[4] demande le mutex
[2] lache le mutex
        [3] tient le mutex
[2] demande le mutex
[3] lache le mutex
        [2] tient le mutex
  (Contrôle-C)
$
```

Comportement en temps réel

Le comportement observé ci-dessus est le fruit d'un ordonnancement en temps partagé. Nous savons qu'il n'y a aucune garantie particulière offerte par cet ordonnancement, et que son comportement peut varier d'une version à l'autre de Linux ou de la bibliothèque C (Glibc avec NPTL ou uClibc avec Linux Threads). Pour plus de déterminisme, tournons-nous vers un ordonnancement temps réel.

Le programme est légèrement modifié car les threads ne vont plus boucler (sinon le plus prioritaire aura toujours le mutex). Nous allons observer l'ordre de prise du mutex lors de la libération initiale. Commençons avec des threads dont la priorité est égale à leur numéro d'ordre de création.

exemple-prise-mutex-03.c:

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

Copyright © 2015 Eyrolles.

```
static pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

void * fonction_thread (void * arg)
{
    int numero = (int) arg;
    struct sched_param param;

    param.sched_priority = numero;
    if (pthread_setschedparam(pthread_self(), SCHED_FIFO,
                    & param) != 0) {
        perror("setschedparam");
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "[%d] demande le mutex\n", numero);
    pthread_mutex_lock(& mutex);
    fprintf(stderr, "        [%d] tient le mutex\n",
                            numero);
    sleep(1);
    fprintf(stderr, "[%d] lache le mutex\n", numero);
    pthread_mutex_unlock(& mutex);
    return NULL;
}

#define NB_THREADS 5
int main(void)
{
    int i;
    pthread_t thread[NB_THREADS];

    pthread_mutex_lock(& mutex);
    for (i = 0; i < NB_THREADS; i ++) {
        pthread_create(& thread[i], NULL,
                       fonction_thread, (void *) (i+1));
        usleep(10000);
    }
    sleep(1);
    fprintf(stderr, "Liberation initiale du mutex\n");
    pthread_mutex_unlock(& mutex);
    for (i = 0; i < NB_THREADS; i ++)
        pthread_join(thread[i], NULL);
    return EXIT_SUCCESS;
}
```

Exécutons notre exemple :

```
# ./exemple-prise-mutex-03
[1] demande le mutex
[2] demande le mutex
[3] demande le mutex
```

Copyright © 2015 Eyrolles.

```
[4] demande le mutex
[5] demande le mutex
Liberation initiale du mutex
        [5] tient le mutex
[5] lache le mutex
        [4] tient le mutex
[4] lache le mutex
        [3] tient le mutex
[3] lache le mutex
        [2] tient le mutex
[2] lache le mutex
        [1] tient le mutex
[1] lache le mutex
#
```

Les prises successives du mutex se sont bien réalisées en fonction de la priorité du thread. Inversons les priorités pour vérifier que l'ordre initial de demande du mutex n'influe pas.

```
exemple-prise-mutex-04.c:
[...]
 param.sched_priority = 10 - numero;
[...]
```

À l'exécution, l'ordre de prise du mutex est bien celui des priorités décroissantes :

```
# ./exemple-prise-mutex-04
[1] demande le mutex
[2] demande le mutex
[3] demande le mutex
[4] demande le mutex
[5] demande le mutex
Liberation initiale du mutex
        [1] tient le mutex
[1] lache le mutex
        [2] tient le mutex
[2] lache le mutex
        [3] tient le mutex
[3] lache le mutex
        [4] tient le mutex
[4] lache le mutex
        [5] tient le mutex
[5] lache le mutex
#
```

Copyright © 2015 Eyrolles.

Reprise de mutex en temps réel

Un problème récurrent dans les systèmes embarqués temps réel est la nécessité de synchroniser plusieurs threads sur la prise permanente du même mutex. Grossièrement, nous voudrions que chaque thread puisse effectuer une boucle infinie :

```
while (1)  {
          pthread_mutex_lock(& mutex);
          // Traitement critique
             [...]
          pthread_mutex_unlock(& mutex);
}
```

Dans ce cas, tous les threads (dont le nombre peut être variable en fonction de l'évolution du système) seront placés au même niveau de priorité et ordonnancés en temps réel Round Robin, par exemple. Nous espérons ainsi que chacun d'entre eux pourra successivement recevoir le mutex et effectuer son traitement critique.

```
exemple-prise-mutex-05.c:
[...]
void * fonction_thread (void * arg)
{
    int numero = (int) arg;
    struct sched_param param;

    param.sched_priority = 10;
    if (pthread_setschedparam(pthread_self(), SCHED_RR,
                              & param) != 0) {
        perror("setschedparam");
        exit(EXIT_FAILURE);
    }

    while (1) {
        fprintf(stderr, "[%d] demande le mutex\n",numero);
        pthread_mutex_lock(& mutex);
        fprintf(stderr, "        [%d] tient le mutex\n",
                        numero);
        sleep(1);
        fprintf(stderr, "[%d] lache le mutex\n", numero);
        pthread_mutex_unlock(& mutex);
    }
    return NULL;
}
[...]
```

L'exécution semble pourtant décevante…

Copyright © 2015 Eyrolles.

```
# ./exemple-prise-mutex-05
[1] demande le mutex
[2] demande le mutex
[3] demande le mutex
[4] demande le mutex
[5] demande le mutex
Liberation initiale du mutex
        [1] tient le mutex
[1] lache le mutex
[1] demande le mutex
        [1] tient le mutex
[1] lache le mutex
[1] demande le mutex
        [1] tient le mutex
[1] lache le mutex
[1] demande le mutex
        [1] tient le mutex
[1] lache le mutex
[1] demande le mutex
        [1] tient le mutex
[1] lache le mutex
[1] demande le mutex
        [1] tient le mutex
[1] lache le mutex
[1] demande le mutex
        [1] tient le mutex
[1] lache le mutex
[1] demande le mutex
        [1] tient le mutex
 (Contrôle-C)
#
```

Comment se fait-il qu'un seul thread accède au mutex alors qu'ils sont cinq à tenter de l'obtenir simultanément, et que nous les avons placés en ordonnancement Round Robin ? Pour comprendre le fonctionnement, reprenons les verrouillages et déverrouillages des mutex.

Notre thread numéro 1 a obtenu le mutex car il était le premier de son niveau de priorité à s'exécuter. Il le libère ensuite, ce qui se traduit par un appel système car le mutex est réclamé par les quatre autres threads (endormis pour le moment). Durant cet appel système, le kernel réveille les threads en attente, afin qu'ils puissent à nouveau tenter leur chance en réclamant le mutex. Avant de revenir dans l'espace utilisateur, l'ordonnanceur est invoqué, et ce dernier juge que le thread numéro 1, étant en temps réel Round Robin, peut continuer à s'exécuter jusqu'à ce qu'il ait consommé tout le temps CPU alloué à sa tranche de temps (100 millisecondes). Il faut être conscient que le thread ne consomme que très peu de temps processeur, car il passe l'essentiel de son temps en sommeil dans le `sleep()` au milieu de la boucle.

Le thread 1 reprenant son exécution, il peut à nouveau réclamer le mutex qu'il vient juste de libérer, et l'obtenir car les autres threads n'ont pas pu être ordonnancés. Lorsque notre thread

Copyright © 2015 Eyrolles.

va s'endormir dans le `sleep()`, ses confrères vont demander successivement le mutex – déjà verrouillé – et s'endormir l'un après l'autre.

Pour que les autres threads arrivent à prendre le mutex, il faudra attendre que le thread 1 ait consommé l'intégralité de sa tranche de temps CPU, ce qui n'adviendra qu'après un très grand nombre d'itérations si nous conservons le sommeil d'une seconde. Dans l'exemple suivant, nous allons supprimer ce sommeil afin de forcer le thread à consommer du temps CPU. Une variable globale contenant le numéro du dernier thread ayant pris le mutex permettra de voir les transitions horodatées.

```
exemple-prise-mutex-06.c:
[...]
static int dernier_thread = 0;

void * fonction_thread (void * arg)
{
    int numero = (int) arg;
    struct sched_param param;
    struct timeval tv;

    param.sched_priority = 10;
    if (pthread_setschedparam(pthread_self(), SCHED_RR,
                              & param) != 0) {
        perror("setschedparam");
        exit(EXIT_FAILURE);
    }
    while (1) {
        pthread_mutex_lock(& mutex);
        if (dernier_thread != numero) {
            gettimeofday(& tv, NULL);
           fprintf(stderr,"%ld.%06ld: %d tient le mutex\n",
                        tv.tv_sec, tv.tv_usec, numero);
            dernier_thread = numero;
        }
        pthread_mutex_unlock(& mutex);
    }
    return NULL;
}
```

Par ailleurs, pour permettre une sortie du programme, un appel système `alarm()` est ajouté dans la fonction `main()`. Ainsi, le noyau enverra au processus le signal `SIGALRM` au bout de 15 secondes. Ce signal n'étant pas géré, il va tuer notre processus.

```
int main(void)
{
    [...]
    sleep(1);
    alarm(15);
    fprintf(stderr, "Liberation initiale du mutex\n");
    [...]
```

Copyright © 2015 Eyrolles.

Les messages devront être redirigés dans un fichier pour être consultés tranquillement.

```
# ./exemple-prise-mutex-06 2> resultat.txt
Minuterie d'alerte
# cat resultats.txt
Liberation initiale du mutex
1319322465.281375 : 1 tient le mutex
1319322466.479252 : 4 tient le mutex
1319322467.479251 : 1 tient le mutex
1319322467.779246 : 3 tient le mutex
1319322468.779247 : 1 tient le mutex
1319322468.879251 : 5 tient le mutex
1319322469.879246 : 3 tient le mutex
1319322469.979245 : 1 tient le mutex
1319322470.079245 : 4 tient le mutex
1319322470.579252 : 2 tient le mutex
1319322471.579246 : 4 tient le mutex
1319322471.679252 : 5 tient le mutex
1319322471.779245 : 2 tient le mutex
1319322471.979246 : 1 tient le mutex
[...]
#
```

Les résultats ne sont guère concluants. Nous aimerions que les threads alternent correctement leurs exécutions autour de ce mutex.

Solutions

La première solution, qui vient à l'esprit des programmeurs habitués aux systèmes embarqués temps réel « classiques » et à la programmation sur microcontrôleurs est d'ajouter un petit délai entre le relâchement du mutex et sa reprise par le même thread (comme dans l'exemple `exemple-prise-mutex-01.c`).

Typiquement, on utilise un appel `usleep(10)` pour demander une mise en sommeil de 10 microsecondes, voire un appel `usleep(0)` pour forcer un appel au scheduler dans une condition défavorable au processus appelant. En voici un exemple :

```
exemple-prise-mutex-07.c:
[...]
void * fonction_thread (void * arg)
{
    int numero = (int) arg;
    struct sched_param param;

    param.sched_priority = 10;
    if (pthread_setschedparam(pthread_self(), SCHED_FIFO,
                              & param) != 0) {
        perror("setschedparam");
```

Copyright © 2015 Eyrolles.

```
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stderr, "[%d] demande le mutex\n", numero);
        pthread_mutex_lock(& mutex);
        fprintf(stderr, "        [%d] tient le mutex\n",
              numero);
        sleep(1);
        fprintf(stderr, "[%d] lache le mutex\n", numero);
        pthread_mutex_unlock(& mutex);
        usleep(10);
    }
    return NULL;
}
[...]
```

Cette fois, l'ordonnancement s'effectue en mode Fifo, afin d'éviter toute préemption involontaire. Cette solution fonctionne, en voici une démonstration.

```
# ./exemple-prise-mutex-07
[1] demande le mutex
[2] demande le mutex
[3] demande le mutex
[4] demande le mutex
[5] demande le mutex
Liberation initiale du mutex
        [1] tient le mutex
[1] lache le mutex
        [2] tient le mutex
[1] demande le mutex
[2] lache le mutex
        [3] tient le mutex
[2] demande le mutex
[3] lache le mutex
        [4] tient le mutex
[3] demande le mutex
[4] lache le mutex
        [5] tient le mutex
[4] demande le mutex
[5] lache le mutex
        [1] tient le mutex
[5] demande le mutex
[1] lache le mutex
        [2] tient le mutex
[1] demande le mutex
[2] lache le mutex
        [3] tient le mutex
```

Copyright © 2015 Eyrolles.

```
[2] demande le mutex
[3] lache le mutex
        [4] tient le mutex
[3] demande le mutex
[4] lache le mutex
        [5] tient le mutex
[...]
  (Contrôle-C)
#
```

Appel explicite à l'ordonnanceur

La solution précédente fonctionne mais n'est guère élégante ni efficace. Si le thread est le seul à travailler, il effectue incessamment des sommeils inutiles.

Il existe pourtant l'appel système `sched_yield()`, dont nous avons parlé au chapitre 5, qui permet de forcer un appel à l'ordonnanceur en demandant à être placé à la fin de la liste des tâches de la même priorité que l'appelant. Si aucune autre tâche de même priorité n'est prête, le thread appelant est réactivé immédiatement. Voici comment il faut l'utiliser en cette circonstance :

```
exemple-prise-mutex-08.c:
[...]
void * fonction_thread (void * arg)
{
    int numero = (int) arg;
    struct sched_param param;

    param.sched_priority = 10;
    if (pthread_setschedparam(pthread_self(), SCHED_FIFO,
                              & param) != 0) {
        perror("setschedparam");
        exit(EXIT_FAILURE);
    }

    while (1) {
        fprintf(stderr, "[%d] demande le mutex\n", numero);
        pthread_mutex_lock(& mutex);
        fprintf(stderr, "        [%d] tient le mutex\n",
                            numero);
        sleep(1);
        fprintf(stderr, "[%d] lache le mutex\n", numero);
        pthread_mutex_unlock(& mutex);
        sched_yield();
    }
    return NULL;
}
[...]
```

Copyright © 2015 Eyrolles.

L'exécution confirme bien le comportement attendu.

```
# ./exemple-prise-mutex-08
[1] demande le mutex
[2] demande le mutex
[3] demande le mutex
[4] demande le mutex
[5] demande le mutex
Liberation initiale du mutex
        [1] tient le mutex
[1] lache le mutex
        [2] tient le mutex
[1] demande le mutex
[2] lache le mutex
        [3] tient le mutex
[2] demande le mutex
[3] lache le mutex
        [4] tient le mutex
[3] demande le mutex
[4] lache le mutex
        [5] tient le mutex
[4] demande le mutex
[5] lache le mutex
        [1] tient le mutex
[5] demande le mutex
[1] lache le mutex
        [2] tient le mutex
[1] demande le mutex
[2] lache le mutex
        [3] tient le mutex
[...]
#
```

Conclusion

La mise au point d'une application temps réel est une chose complexe et de nombreux « pièges » inattendus peuvent se présenter. Le premier réflexe est souvent d'utiliser des délais, des attentes, pour synchroniser les opérations. Cette méthode, bien que fonctionnelle dans la plupart des cas, n'est pas toujours optimale ni portable. En explorant l'API disponible, le programmeur pourra souvent trouver de meilleures solutions (variables conditions, barrières, héritage de priorité, de-scheduling volontaires, etc.).

Copyright © 2015 Eyrolles.

Points clés

- En temps réel, pour éviter qu'un thread nouvellement créé préempte son créateur et l'empêche de lancer les autres threads équivalents, on peut utiliser les barrières Posix.
- Pour empêcher l'inversion de priorité qui peut se produire sur les mutex, le noyau Linux propose une implémentation du *Priority Inheritance Protocol*, que l'on active à l'initialisation d'un mutex.
- La prise d'un mutex se fait généralement suivant la priorité des threads qui le demandent. Si un thread lâche et redemande le même mutex immédiatement, il est probable qu'il l'obtienne au détriment des autres. Pour réguler cet accès, on peut employer `sched_yield()`.

Exercices

Exercice 1 (*)

Comment mettre en œuvre un mécanisme équivalent à celui des barrières `pthread_barrier_t` pour synchroniser le démarrage de plusieurs processus temps réel distincts ?

Exercice 2 (**)

Reprenez le programme `exemple-inversion.c` et modifiez-le pour employer des processus plutôt que des threads. Le mutex sera remplacé par un sémaphore Posix.

Y a-t-il une inversion de priorité ? Peut-on configurer un héritage de priorité sur le sémaphore ?

Exercice 3 (**)

Sur un système ne disposant pas d'héritage de priorité sur les mutex, comment peut-on éviter les inversions de priorité telles que nous les avons observées ?

Exercice 4 (***)

Quel est l'ordre de prise d'un sémaphore lorsque plusieurs processus de priorités différentes le réclament simultanément ? Lorsqu'il s'agit de processus de même priorité en Round Robin ?

Exercice 5 (***)

Écrivez un programme qui travaille en boucle autour de la prise d'un sémaphore Posix. Lancez-en plusieurs exemplaires en parallèle et assurez-vous que leurs exécutions soient cadencées de manière homogène et régulière.

Copyright © 2015 Eyrolles.

Copyright © 2015 Eyrolles.

8

Limites et améliorations du temps réel Linux

Le temps réel souple de Linux tel que nous l'avons observé jusqu'ici offre de nombreuses possibilités et des performances assez intéressantes. Il y a toutefois quelques limitations parfois gênantes, comme les traitements longs d'interruption au détriment des tâches temps réel. Nous allons étudier ce problème, ainsi qu'une solution offerte par le projet PREEMPT_RT. Nous verrons qu'il est important de pouvoir comparer les performances en fonction des options de configuration, ce que permettent certains outils de tests. Nous examinerons également des solutions pour améliorer le comportement des systèmes embarqués, comme la régulation de la vitesse du processeur.

Traitement des interruptions

Au chapitre 6, nous avons examiné les réponses aux interruptions depuis l'espace utilisateur, lorsque le gestionnaire – installé dans le noyau – réveillait une tâche en attente. Nous avons étudié à titre d'exemple un processus qui s'endormait jusqu'à l'arrivée d'un caractère sur le port série. Grâce à la préemptibilité (optionnelle) du noyau, nous avons vérifié que le temps de réveil de cette tâche n'est pas perturbé par un processus de plus faible priorité réalisant un appel système long.

Nous allons poursuivre sur ce sujet en étudiant le problème réciproque : l'influence de l'arrivée d'une interruption sur une tâche temps réel.

> Attention, il s'agit d'un domaine très actif dans l'évolution actuelle du noyau Linux. Je réalise ci-après des expériences avec un noyau bien particulier (3.18), mais ses successeurs n'auront peut-être pas le même comportement.

Copyright © 2015 Eyrolles.

Réalisons une petite expérience : sur un Raspberry Pi 2, nous allons faire tourner des boucles temps réel de priorité maximale (Fifo 99) sur tous les CPU, et pendant ce temps nous enverrons des *pings* vers cette machine depuis un autre ordinateur du même réseau. Pour que ce processus soit indépendant de l'ordonnanceur, nous allons éviter d'invoquer un appel système et réaliser plutôt une boucle active de comptage. La valeur maximale du comptage dépendra bien sûr des plates-formes. Pour un Raspberry Pi 2, un comptage jusqu'à 500 millions dure une petite dizaine de secondes.

```
exemple-boucle-temps-reel-01.c  :
#define _GNU_SOURCE  // Pour avoir sched_setaffinity
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

    static long int count = 100000000;
    static pthread_barrier_t barrier;

void * fonction_thread(void * arg)
{
    long n;
    long cpu = (long) arg;
    cpu_set_t  cpuset;

    // Passer sur le CPU demandé.
    CPU_ZERO(& cpuset);
    CPU_SET(cpu, &cpuset);
    if (pthread_setaffinity_np(pthread_self(),
                               sizeof(cpuset), & cpuset) !=0) {
        perror("pthread_setaffinity_np");
        exit(EXIT_FAILURE);
    }

    pthread_barrier_wait(& barrier);

    // Exécuter la boucle de comptage.
    for (n = 0; n < count; n ++)
        ;
    return NULL;
}


int main(int argc, char * argv[])
{
    long int nb_cpus = sysconf(_SC_NPROCESSORS_ONLN);
    pthread_t * threads;
    long int cpu;
    struct sched_param param;
```

Copyright © 2015 Eyrolles.

```
    // Initialiser la valeur max. du comptage.
    if (argc > 1) {
        if (sscanf(argv[1], "%ld", & count) != 1) {
            fprintf(stderr, "usage: %s [count-value]\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }
    // Passer en temps réel.
    param.sched_priority = 95;
    if (sched_setscheduler(0, SCHED_FIFO, & param) != 0) {
        perror("sched_setscheduler");
        exit(EXIT_FAILURE);
    }
    // Initialiser la table des threads.
    threads = calloc(nb_cpus, sizeof(pthread_t));
    if (threads == NULL) {
        perror(argv[0]);
        exit(EXIT_FAILURE);
    }

    // Démarrer les threads.
    pthread_barrier_init(& barrier, NULL, nb_cpus);
    for (cpu = 0; cpu < nb_cpus; cpu ++) {
        if (pthread_create(& (threads[cpu]), NULL,
                           fonction_thread, (void *) cpu) != 0) {
            perror(argv[0]);
            exit(EXIT_FAILURE);
        }
    }

    // Attendre la fin des threads.
    for (cpu = 0; cpu < nb_cpus; cpu ++) {
        pthread_join(threads[cpu], NULL);
    }
    free(threads);
    return EXIT_SUCCESS;
}
```

Les deux affichages sont présentés en parallèle sur deux colonnes :

```
# uname -r
3.18.11-v7+
#
                    $ ping 192.168.3.152
                    PING 192.168.3.152 56(84) bytes of data.
                    icmp_req=1 ttl=64 time=0.355 ms
                    icmp_req=2 ttl=64 time=0.306 ms
                    icmp_req=3 ttl=64 time=0.269 ms
                    icmp_req=4 ttl=64 time=0.389 ms
```

Copyright © 2015 Eyrolles.

```
# ./exemple-boucle-temps-reel
                    icmp_req=5  ttl=64 time=0.281 ms
                    icmp_req=6  ttl=64 time=0.265 ms
                    icmp_req=7  ttl=64 time=0.259 ms
                    icmp_req=8  ttl=64 time=0.246 ms
                    icmp_req=9  ttl=64 time=0.248 ms
                    icmp_req=10 ttl=64 time=0.244 ms
                    icmp_req=11 ttl=64 time=0.251 ms
                    icmp_req=12 ttl=64 time=0.240 ms
                    icmp_req=13 ttl=64 time=0.376 ms
#
                    icmp_req=14 ttl=64 time=0.275 ms
                    icmp_req=15 ttl=64 time=0.260 ms
                    icmp_req=16 ttl=64 time=0.268 ms
                         (Contrôle-C)
                    --- 192.168.3.152 ping statistics ---
                    16 packets transmitted, 16 received, 0% packet loss
                    $
```

Sur ce noyau Linux standard, le `ping` depuis une autre machine a continué de fonctionner même pendant l'exécution d'une boucle temps réel de très haute priorité. Que s'est-il passé ?

Je précise que la réponse au `ping` n'est pas traitée directement par la carte réseau mais bien par les couches de protocoles du noyau. Il suffit pour s'en convaincre de voir qu'un :

```
# echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

interdit instantanément toutes les réponses au `ping`, alors que :

```
# echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

les rétablit.

Le processus `exemple-boucle-temps-reel-01`, placé en priorité Fifo 95 est *a priori* totalement protégé contre les préemptions, et son exécution nous semble absolument déterministe et prévisible. Il n'y a aucune autre tâche temps réel plus prioritaire. On a exécuté auparavant :

```
 sysctl kernel.sched_rt_runtime _us=-1
```

pour désactiver le garde-fou présenté dans le chapitre 5. Comme attendu, la console et l'environnement graphique du système semblent totalement gelés.

Toutefois, une interruption matérielle survient. Rappelons-nous qu'il n'est pas possible de couper les interruptions matérielles depuis l'espace utilisateur.

Copyright © 2015 Eyrolles.

Même si sur certaines architectures comme le PC on peut effectivement couper les interruptions depuis un programme avec des opérations comme `iopl(3); asm("cli");...; asm("sti")`, ce n'est ni élégant ni portable. Je ne prendrai donc pas en considération cette possibilité.

L'interruption en question est celle du contrôleur Ethernet. Le noyau va donc activer le handler inclus dans le driver de ce dernier, et celui-ci va enchaîner plusieurs opérations résumées sur la figure 8-1.

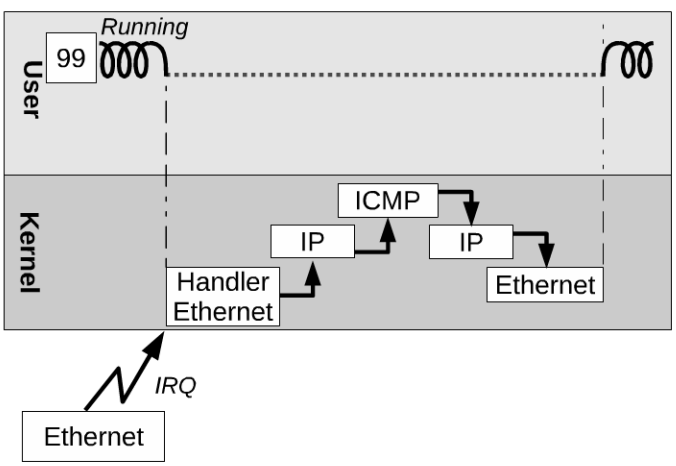
- Il interroge le matériel pour vérifier si des données sont bien arrivées, récupère la trame reçue, puis la transmet à la fonction `netif_rx()` du fichier `net/core/dev.c` qui représente le point d'entrée « bas » de la pile réseau.
- Déterminant qu'il s'agit d'un paquet pour le protocole IPv4, cette routine l'envoie à la fonction `ip_rcv()` du fichier `net/ipv4/ip_input.c`, entrée de la couche IP.
- Cette dernière détermine qu'il s'agit d'un paquet pour le protocole ICMP et le transmet à la fonction `icmp_rcv()` du fichier `net/ipv4/icmp.c`. Voyant qu'il s'agit d'une demande d'écho, cette dernière invoque donc la fonction `icmp_echo()` du même fichier.
- Après vérification de l'autorisation de répondre (si `/proc/sys/kernel/ipv4/icmp_echo_ignore_all` contient bien zéro), cette routine prépare la réponse et le paquet repart dans l'autre sens, redescendant vers la couche IP – qui devra peut-être calculer une route vers la destination.
- Les données préparées par la couche IP vont redescendre vers la couche Ethernet, qui les transmet au matériel en lui demandant de les poster sur le réseau.

N'oublions pas également l'implication éventuelle du firewall *(Net-Filter),* qui intervient à chaque transition entre les couches réseau pour vérifier si les données sont autorisées à poursuivre leur chemin.

Enfin, l'interruption se termine et à l'issue de toutes ces opérations, notre processus temps réel qui pensait s'exécuter de manière exclusive peut reprendre son déroulement...

Figure 8-1

Interruption d'une tâche temps réel



Copyright © 2015 Eyrolles.

En réalité, tout le travail n'est pas réalisé directement dans le gestionnaire d'interruptions ; le noyau Linux considère qu'il faut distinguer deux parties dans les handlers d'interruption : la moitié supérieure *(Top Half)* qui doit être effectuée le plus rapidement possible – et sans que la même IRQ ait la possibilité de se déclencher à nouveau –, et la moitié inférieure *(Bottom Half)* qui peut être légèrement retardée.

Pendant longtemps, les drivers pouvaient s'appuyer essentiellement sur deux mécanismes pour scinder leurs traitements en interruptions : les *tasklets,* qui sont exécutées immédiatement après le retour du handler d'interruption, et les *workqueues,* qui sont exécutées par un kernel thread – donc ordonnancées au même titre que les processus. La situation a légèrement évolué depuis les derniers noyaux 2.6, nous détaillerons cela plus loin.

Le traitement de la pile réseau est effectuée par une *soft-IRQ*, mécanisme très proche des tasklets. Ainsi, tout le traitement de réponse au `ping` est réalisé au détriment des tâches temps réel, quelles que puissent être leurs priorités.

Pour vérifier ce comportement, et essayer de mesurer la durée pendant laquelle notre tâche temps réel est interrompue, nous allons utiliser un petit programme, proche de ceux développés au chapitre 4, qui détecte et affiche les préemptions d'une durée (en microsecondes) supérieure à un certain seuil. Le programme tournant en temps réel, il s'endort volontairement à intervalle régulier afin de laisser le système respirer (afficher les messages, traiter les tâches en attente, etc.).

exemple-interruption-01.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

int main(int argc, char * argv[])
{
    long int seuil;
    struct timeval debut;
    struct timeval heure;
    struct timeval precedente = {0, 0};
    long long int difference;

    if ((argc != 2)
     || (sscanf(argv[1], "%ld", & seuil) != 1)) {
        fprintf(stderr,"usage: %s seuil-en-microsec\n",
                        argv[0]);
        exit(EXIT_FAILURE);
    }

    gettimeofday(& debut, NULL);
    precedente = debut;
    while(1) {
        gettimeofday(& heure, NULL);
        difference  = heure.tv_sec - precedente.tv_sec;
        difference *= 1000000;
```

Copyright © 2015 Eyrolles.

```
        difference += heure.tv_usec - precedente.tv_usec;
        precedente = heure;
        if (difference >= seuil)
            fprintf(stderr, "%lld\n", difference);
        // Pause toutes les cinq secondes
        if (heure.tv_sec - debut.tv_sec > 5) {
            usleep(100000);
            gettimeofday(&debut, NULL);
            precedente = debut;
        }
    }
    return EXIT_SUCCESS;
}
```

Lançons ce programme sur un poste, en temps réel Fifo de priorité 99 ; simultanément, nous allons exécuter un `ping` depuis une autre station du même réseau. Notre programme sera placé sur le cœur 0 d'un Raspberry Pi 2. L'exemple tourne sur un noyau 3.18 :

```
# taskset -p -c 0 $$

# chrt -f 99 ./exemple-interruption-01 100
                          $ ping 192.168.3.152
                          icmp_req=1 ttl=64 time=0.300 ms
115
                          icmp_req=2 ttl=64 time=0.246 ms
136
                          icmp_req=3 ttl=64 time=0.266 ms
120
                          icmp_req=4 ttl=64 time=0.268 ms
115
                          icmp_req=5 ttl=64 time=0.242 ms
112
                          icmp_req=6 ttl=64 time=0.252 ms
112
                          icmp_req=7 ttl=64 time=0.250 ms
127
                             (Contrôle-C)
                          $
   (Contrôle-C)
#
```

Le choix du seuil (100 microsecondes) a été effectué arbitrairement, et peut nécessiter une modification pour réitérer l'expérience sur une autre machine. Ici, nous voyons que notre processus temps réel est préempté pendant 100 à 150 microsecondes à chaque fois qu'un `ping` arrive sur l'interface Ethernet !

Copyright © 2015 Eyrolles.

Naturellement, ceci est contraire à tous les principes du temps réel puisque nous ne maîtrisons absolument pas l'arrivée de cette interruption (qui n'a par ailleurs rien à voir avec l'application temps réel). Ce genre de situation, qui correspond de fait à une inversion de priorité, peut se retrouver dans d'autres circonstances :

- un processus de faible priorité a demandé une lecture depuis le disque et lorsque l'interruption indiquant que les données sont disponibles se déclenche, elle préempte une tâche de haute priorité ;
- un utilisateur déplace la souris ou presse une touche du clavier, ce qui déclenche une interruption (*via* le contrôleur USB, éventuellement) au détriment d'un processus temps réel ;
- certains traitements réalisés par le noyau peuvent demander une consommation importante de CPU : gestion de la mémoire virtuelle, systèmes de fichiers cryptés, etc.

Pour pallier cette faiblesse du noyau Linux, deux approches sont possibles :

- améliorer le traitement des interruptions dans le kernel en transférant une plus grosse part de code dans les *Bottom Halves* ordonnancées ;
- ajouter sous le kernel Linux un cœur de noyau temps réel qui ne lui envoie les interruptions que si aucune tâche prioritaire n'est prête.

La seconde approche est celle des systèmes tels que RTLinux, RTAI ou Xenomai dont nous parlerons dans les prochains chapitres. La première est celle adoptée par le projet PREEMPT_RT.

PREEMPT_RT

L'un des principaux projets destinés à améliorer les performances de Linux pour le temps réel est le projet PREEMPT_RT de Thomas Gleixner, Ingo Molnár et de nombreux autres collaborateurs. On le nomme également Linux-RT.

> Attention à ne pas confondre Linux-RT, projet libre pour améliorer le noyau classique, et RTLinux, produit commercial utilisant un micronoyau temps réel ordonnançant le noyau standard en tâche de fond.

Le projet PREEMPT_RT se présente sous la forme d'un patch que l'on peut appliquer sur un noyau standard pour affiner ses fonctionnalités dans divers domaines touchant au temps réel. Il convient de considérer ce patch à la fois comme une amélioration sensible du comportement de Linux temps réel, mais également comme une sorte de laboratoire où les participants au projet valident des optimisations en attendant qu'elles soient effectivement intégrées au noyau Linux.

De nombreuses contributions de PREEMPT_RT restent assez confidentielles (les diminutions des latences dues au *Virtual File System* et à la gestion de la mémoire, par exemple, ou encore les spinlocks préemptibles). D'autres sont plus visibles, comme la préemptibilité du noyau que nous avons étudiée au chapitre 6 ou les *threaded interrupts* que nous allons observer à présent. Notez que ces éléments ont d'ores et déjà été intégrés dans le noyau standard. Toutefois, la plupart des fonctionnalités qui améliorent le comportement du point de vue temps réel présentent un revers gênant dans d'autres domaines, en général un léger surcoût de temps CPU qui peut devenir prohibitif sur les systèmes parallélisant un grand nombre de tâches (serveurs, calculs, bases de données, etc.). Aussi, la progression des fonctionnalités de PREEMPT_RT dans le noyau standard est assez lente.

Copyright © 2015 Eyrolles.

Ceci est dû pour une faible part au manque de conviction de Linus Torvalds dans l'utilité du temps réel strict sous Linux, et pour une grande part à l'absence dans le développement du noyau des utilisateurs industriels (embarqué, temps réel, etc.), ainsi qu'à leur maigre retour d'expérience sur les mises en œuvre réalisées.

Il est donc souvent intéressant d'utiliser le patch de Thomas Gleixner pour bénéficier des nouvelles améliorations qui n'ont pas encore été intégrées dans le noyau Linux. Ce patch, mis à jour régulièrement, est disponible à l'adresse suivante : http://www.kernel.org/pub/linux/kernel/projects/rt/.

En novembre 2014, Thomas Gleixner a annoncé que le manque de financement de la part du monde industriel ne lui permettait plus de développer le projet PREEMPT_RT à plein temps et qu'il devrait à présent le faire sur son temps libre. Ceci risquant d'impacter la disponibilité du patch pour les nouveaux noyaux, des acteurs industriels ont proposé leur soutien. En octobre 2015, la Linux Foundation a annoncé la création d'une branche *« Real Time Linux Collaborative Project »* en embauchant Thomas Gleixner en tant que *Linux Foundation Fellow* au même titre que Linus Torvalds, Greg Kroah-Hartman ou Richard Purdie.

Voici un exemple d'application de ce patch. Commençons par télécharger un noyau Linux vanilla adapté à notre plate-forme (Raspberry Pi 2) :

```
# uname -r
3.18.11-v7+
# git clone https://github.com/raspberrypi/linux linux
```

Le clonage avec Git du noyau et de tout l'historique de ses modifications est une opération assez longue, variable en fonction du système et du réseau, mais de l'ordre d'une heure. Extrayons la version du noyau Linux qui nous intéresse et vérifions son numéro exact.

```
# cd linux
# git  checkout  rpi-3.18.y
# head Makefile
VERSION = 3
PATCHLEVEL = 18
SUBLEVEL = 16
EXTRAVERSION =
```

Nous pouvons télécharger le patch PREEMPT_RT correspondant.

```
# cd ..
# wget  https://www.kernel.org/pub/linux/kernel/projects/rt/3.18/older/patch-
3.18.16-rt13.patch.xz
# unxz  patch-3.18.16-rt13.patch.xz
# cd  linux
```

Attention à ne pas télécharger `patches-3.18.16...` qui contient tous les petits patchs compilés par Thomas Gleixner pour obtenir le gros patch PREEMPT_RT.

Copyright © 2015 Eyrolles.

Nous allons commencer par créer une branche de travail avant d'appliquer le patch.

```
# git branch rpi-3.18.16-RT
# git checkout rpi-3.18.16-RT
# patch -p1 < ../patch-3.18.16-rt13.patch  --dry-run
[...]
patching file mm/Kconfig
patching file init/Makefile
patching file scripts/mkcompile_h
$
```

Je conseille d'utiliser systématiquement l'option `--dry-run` lors du premier test d'un nouveau patch. Cette option ne modifie pas les sources du noyau, elle vérifie simplement si l'application du patch est possible sans erreur. Nous pouvons voir quelques messages *« Hunk succeeded at ... »* bénins qui indiquent un léger décalage entre le code source dont nous disposons (adapté pour Raspberry Pi) et celui du noyau officiel à partir duquel le patch a été produit.

Si le résultat est correct, comme ici, nous pouvons relancer la commande `patch` sans l'option `--dry-run` :

```
# patch -p1 < ../patch-3.18.16-rt13.patch
[...]
patching file scripts/mkcompile_h
patching file sound/core/pcm_native.c
patching file virt/kvm/async_pf.c
patching file virt/kvm/kvm_main.c
#
```

Nous récupérons la configuration du système en fonctionnement :

```
# zcat /proc/config.gz > ./.config
# make menuconfig
```

La suite de la compilation est identique à la description de l'annexe A. Toutefois, dans le menu de configuration du noyau, une nouvelle option est présente, Fully preemptible kernel, en plus des options de préemptibilité décrites au chapitre 6. Nous en reparlerons plus loin.

Pour identifier un noyau contenant le patch PREEMPT_RT, c'est très simple, il suffit de saisir :

```
# uname -a
Linux raspberrypi 3.18.16-rt13-cpb+ #1 SMP PREEMPT RT Tue Jul 14 23:43:08 CEST 2015
armv7l GNU/Lin ux
```

Le point à remarquer est la présence du flag `RT`. Attention, l'option `PREEMPT` n'indique pas la présence du patch, mais seulement que le noyau a été compilé avec une configuration préemptible.

Copyright © 2015 Eyrolles.

Quel est l'intérêt d'utiliser un noyau modifié par le patch PREEMPT_RT ? Après avoir redémarré sur un kernel modifié par ce patch, nous allons reprendre notre expérience précédente avec les boucles temps réel :

```
# uname -r
3.18.16-rt13-cpb+
                          $ ping 192.168.3.152
                          icmp_req=1  ttl=64 time=0.337 ms
# ./exemple-boucle-temps-reel-01 500000000
                          icmp_req=2  ttl=64 time=8754 ms
                          icmp_req=3  ttl=64 time=7755 ms
                          icmp_req=4  ttl=64 time=6755 ms
                          icmp_req=5  ttl=64 time=5755 ms
                          icmp_req=6  ttl=64 time=4755 ms
                          icmp_req=7  ttl=64 time=3756 ms
                          icmp_req=8  ttl=64 time=2756 ms
                          icmp_req=9  ttl=64 time=1756 ms
                          icmp_req=10 ttl=64 time=756 ms
#
                          icmp_req=11 ttl=64 time=0.411 ms
                          icmp_req=12 ttl=64 time=0.371 ms
                            (Contrôle-C)
                          $
```

Cette fois, notre processus temps réel n'est jamais préempté durant sa boucle active. La commande `ping` continue inlassablement à envoyer des messages toutes les secondes, mais ils ne sont pas traités car notre processus est plus prioritaire. Les réponses aux `ping` ne sont envoyées qu'une fois la boule temps réel achevée. C'est très visible sur la trace précédente en observant les temps de réponse. Une fois la boucle commencée, un premier `ping` a duré 8,7 secondes environ, le suivant (dont la requête a été envoyée une seconde plus tard) 7,7 secondes, puis 6,7 secondes, et ainsi de suite jusqu'à la fin du processus temps réel, et la reprise des réponses normales.

Threaded interrupts

Notre tâche temps réel est bien restée plus prioritaire que le traitement des interruptions. Mais ceci signifie donc que les gestionnaires d'interruptions doivent différer une partie de leur travail pour qu'il soit exécuté avec une priorité donnée.

Nous avons évoqué précédemment le principe des *Top Half* et *Bottom Half*, qui représentent respectivement les traitements urgents – réalisés immédiatement – et ceux moins pressants, qui sont réalisés soit en tasklet, soit par les workqueues dans un kernel thread. Ces dernières sont en train de disparaître au profit des threaded interrupts.

Ce système, que l'on doit au projet PREEMPT_RT, est en cours de déploiement dans le noyau standard depuis le 2.6.30. Lorsqu'un driver désire gérer une interruption, il fournit deux fonctions : la première, appelée directement à l'arrivée de l'IRQ, doit vérifier si l'interruption concerne bien son driver et renvoyer `IRQ_WAKE_THREAD` si c'est le cas – ou `IRQ_NONE` si l'interruption ne vient pas de son matériel. La seconde fonction sera exécutée dans le contexte d'un kernel thread, avec une priorité temps réel Fifo 50 par défaut.

Copyright © 2015 Eyrolles.

Dans le noyau standard, les drivers doivent volontairement utiliser le mécanisme des threaded interrupts, ce qui est encore rare car cela ralentit le traitement des interruptions. Sur un noyau auquel le patch PREEMPT_RT a été appliqué, l'utilisation des threaded interrupts est beaucoup plus généralisée. Pour chercher le numéro de l'interruption Ethernet sur notre système, il suffit de lancer une commande suivante :

```
# watch -n 1 cat /proc/interrupts
```

comme nous l'avons vu au chapitre 2, tandis qu'un `ping` s'exécute depuis une autre machine. On observe alors qu'une ligne augmente régulièrement de deux interruptions par seconde (réception de la requête et fin d'émission de la réponse). Sur mon Raspberry Pi 2, il s'agit de l'interruption 32. J'observe alors la sortie de la commande `ps`.

```
# ps aux
[...]
root   60  0.0  0.0   0   0 ?   S<   03:51  0:00 [iscsi_eh]
root   61  0.0  0.0   0   0 ?   S<   03:51  0:00 [dwc_otg]
root   62  0.9  0.0   0   0 ?   S    03:51  0:05 [irq/32-dwc_otg ]
root   63  0.4  0.0   0   0 ?   S    03:51  0:02 [irq/32-dwc_otg _]
root   64  0.8  0.0   0   0 ?   S    03:51  0:04 [irq/32-dwc_otg _]
root   65  0.0  0.0   0   0 ?   S<   03:51  0:00 [DWC Notificati o]
root   66  0.0  0.0   0   0 ?   S    03:51  0:00 [irq/24-DMA IRQ ]
root   67  0.0  0.0   0   0 ?   S    03:51  0:00 [irq/25-DMA IRQ ]
root   68  0.0  0.0   0   0 ?   S    03:51  0:00 [irq/84-mmc0]
[...]
#
```

Nous voyons que sur ce système doté du patch PREEMPT_RT, les interruptions sont gérées par des kernel threads, ceux de PID 62, 63 et 64 étant dédié à la gestion de l'interruption 32.

L'interruption est ici identifiée comme `dwc_otg` (qui dépend de l'USB), car sur cette carte embarquée, les ports USB et l'interface Ethernet filaire sont gérés par le même contrôleur.

Il est possible de modifier la priorité de cette interruption pour la monter à 98, par exemple, et de réexécuter notre programme de tests (qui fonctionne à la priorité 95) :

```
# chrt -pf 98 62
# chrt -pf 98 63
chrt -pf 98 64
```

Cette fois, le `ping` s'exécute normalement, notre programme étant préempté par le thread traitant l'interruption réseau.

Le mécanisme des threaded interrupts fait lentement son chemin dans le noyau Linux standard. Rappelons-nous que cela pénalise légèrement les traitements, aussi leur intégration complète ne pourra se faire qu'à travers des options de compilation permettant aux utilisateurs n'ayant pas de besoins spécifiquement temps réel de les désactiver.

Copyright © 2015 Eyrolles.

Fully preemptible kernel

Nous avons noté plus haut qu'il existe une option supplémentaire apportée par le patch PREEMPT_RT, nommée Fully preemptible kernel. Le principe est d'améliorer la préemptibilité du code du noyau telle que nous l'avons étudiée au chapitre 6. Nous avions précisé que même avec l'option Preemptible kernel, il restait des portions d'appels système où la préemption était impossible à cause de la prise d'un spinlock, même si ce dernier ne servait pas à protéger une variable concernée par l'interruption.

Le patch PREEMPT_RT rend ces passages préemptibles en remplaçant les spinlocks par des *rt-mutex* et en réduisant au maximum les portions non préemptibles (employant des *raw_spinlock*). En outre, les mutex initiaux du kernel sont à présent dotés d'héritages de priorités. Nous pouvons nous rendre compte de l'amélioration de la préemptibilité du noyau en reprenant notre expérience de réponse en interruption du chapitre 6 avec le kernel sur lequel le patch a été appliqué.

En se remettant dans les mêmes conditions que précédemment, nous obtenons les résultats suivants :

```
# ../chapitre-04/calculer-statistiques < resultats-PREEMPT_RT.txt
 Nb mesures = 5385
 Minimum = 2114
 Maximum = 6147
 Moyenne = 5922
 Ecart-type = 55
$
```

Alors qu'au chapitre 6, avec un noyau Linux standard, nous obtenions :

```
 # ../chapitre-04/calculer-statistiques < resultats-preempt.txt
 Nb mesures = 4460
 Minimum = 534
 Maximum = 6167
 Moyenne = 5928
 Ecart-type = 115
$
```

Les résultats ne sont pas très différents, mais on sent néanmoins que les valeurs sont plus rapprochées autour de la moyenne. L'écart-type, notamment, est divisé par deux. La moyenne est ici très légèrement meilleure, mais il faudrait prendre des mesures beaucoup plus longues pour s'en assurer. On pourrait même s'attendre à ce que la moyenne soit très légèrement moins bonne à cause de la légère surcharge de code induite par PREEMPT_RT.

Conclusion sur PREEMPT_RT

L'utilisation du patch de Thomas Gleixner se justifie souvent par des besoins précis (préemptibilité étendue, threaded interrupts généralisés, etc.), mais il ne faut pas oublier qu'il s'agit d'un projet ne bénéficiant pas d'une audience aussi large que celle du noyau Linux standard et dont

Copyright © 2015 Eyrolles.

le suivi (en termes de correctifs, par exemple) ne pourra être aussi réactif que celui du kernel officiel, bien qu'un effort notable soit visible depuis le noyau 3.0.

Bien sûr, il serait appréciable que toutes ces extensions temps réel trouvent leur place au sein du noyau classique – en étant activables ou non à la compilation du kernel – mais ça n'est pas encore le cas, essentiellement à cause de la diminution de performances brutes qu'imposent certains de ces traitements spécifiques.

Outils de mesure des performances

Il arrive un moment, lors de la conception d'un projet temps réel, où il devient difficile de savoir comment progresser plus avant, et où la mise au point du système semble relever plutôt de l'alchimie et de l'ésotérisme que d'une technique exacte et reproductible.

Les performances globales dépendent de paramètres très différents (taux de préemption des tâches prioritaires, latence des interruptions, fluctuations des timers, etc.) parfois incompatibles entre eux. De plus, d'autres contraintes spécifiques au domaine d'application peuvent se superposer à celles liées au temps réel. En environnement embarqué, on recherchera, par exemple, l'économie d'énergie électrique, la rapidité de *boot*, l'optimisation des accès à la mémoire flash, etc. Sur des machines de pilotage industriel, on devra intégrer des drivers de périphériques ou des protocoles spécifiques (bus, réseau...) provenant de fournisseurs imposant certaines versions du kernel Linux. Dans cette situation, il est nécessaire d'utiliser des tests, des *benchmarks*, pour valider l'influence positive de chaque modification dans la configuration du système.

Outre les petits programmes que nous avons développés jusqu'ici, il existe des outils classiques et libres, qui permettent de vérifier le comportement du système dans diverses situations. Nous allons en étudier rapidement quelques-uns, mais je vous conseille de passer un peu de temps à les tester et à comprendre leurs options.

Cyclictest

Cyclictest est le plus célèbre des outils de mesure pour le temps réel. Écrit à l'origine par Thomas Gleixner, il est à présent maintenu par Clark Williams dans son projet rt-tests. Ce programme mesure les fluctuations et les latences de divers paramètres.

Le téléchargement de rt-tests et sa compilation se réalisent ainsi :

```
$ git clone https://github.com/clrkwllms/rt-tests
[...]
$ cd rt-tests
$ make all
[...]
$
```

Pour l'exécution du programme, il est conseillé d'avoir les droits *root*. L'appel de `cyclictest` avec l'argument `--help` permet d'obtenir la liste des options disponibles. Voici celles que j'utilise le plus fréquemment.

Copyright © 2015 Eyrolles.

Options	Signification
`-p 99`	Utiliser un ordonnancement Fifo de priorité 99.
`-S`	Placer un thread sur chaque processeur détecté.
`-m`	Invoquer `mlockall()` pour verrouiller la mémoire virtuelle du processus en mémoire physique.
`-D 1h`	Laisser le test tourner pendant une heure. On peut utiliser les suffixes `m` (minutes), `h` (heures) voire `d` (*days*, jours).

En voici un exemple d'exécution avec un noyau Linux standard (non modifié avec le patch PREEMPT_RT). Le test dure une heure.

```
# uname  -r
3.18.11-v7+
# ./cyclictest -p 80 -m -D 1h -q
T: 0 ( 2392) P:80 I:1000 C:3600000 Min:     12 Act:   19 Avg:   20 Max:     182
```

Voici à présent un test de la même durée sur le même noyau après application du patch PREEMPT_RT :

```
# uname -r
3.18.16-rt13-cpb+
# ./cyclictest -p 80 -m -D 1h -q
T: 0 ( 2380) P:80 I:1000 C:3600000 Min:     13 Act:   20 Avg:   21 Max:      63
#
```

Dans les deux cas, une tâche (`T:0`), lancée en priorité temps réel (`P:99`), programme un timer avec un intervalle (`I:1000`) de 1 000 microsecondes, soit une milliseconde, ce qui conduit à 36 millions de déclenchements en une heure.

Dans le premier cas, les fluctuations vont de 12 à 182 microsecondes, et leur valeur moyenne est de 20 microsecondes d'écart par rapport au déclenchement désiré.

Dans le second cas, avec le patch PREEMPT_RT, la moyenne est très légèrement dégradée à 21 microsecondes (pour que le système soit plus prédictible, il y a des vérifications supplémentaires), mais on observe surtout que l'écart maximal par rapport au timer programmé est de 63 microsecondes, ce qui est sensiblement mieux que précédemment. Ceci confirme rapidement les résultats que nous avons obtenus précédemment. Bien sûr, il faudrait valider ces mesures sur des durées encore plus longues, en parallèle avec des actions perturbatrices importantes.

Hwlatdetect, Hackbench...

D'autres outils de l'ensemble rt-tests sont très utiles.

- Hwlatency permet de détecter les latences dues au matériel. Il effectue des mesures en boucle dans l'espace kernel pour vérifier si le noyau n'est pas préempté par des interruptions qu'il ne

Copyright © 2015 Eyrolles.

peut pas gérer. Par exemple, sur les architectures de type PC, les SMI *(System Managment Interrupts)* sont des interruptions qui sont gérées par le Bios sans que le noyau ne les voit.

- Hackbench effectue des mesures de commutation entre threads et processus qui communiquent par des tubes ou des sockets. Ce programme étant très intensif, il sert d'élément perturbateur efficace pour les mesures avec `cyclictest`. Dans ce cas, il convient de le faire fonctionner sous ordonnancement temps réel avec une priorité légèrement inférieure à celle de `cyclictest`.
- Pi_stress permet de vérifier l'implémentation des héritages de priorités sur les mutex, en testant différents cas d'inversion de priorité.
- …

Je conseille, lors de la mise au point d'un système temps réel, de lancer des séries de tests de ce package ou d'autres comme les *Realtime Tests* de LTP *(Linux Test Package)*. Pour cela, il est important de préparer des scripts shell permettant de relancer les tests à plusieurs reprises, sur de longues périodes, après chaque modification du kernel, en sauvegardant systématiquement les résultats dans des fichiers. On trouvera dans l'annexe A quelques pistes d'optimisation dans la configuration du noyau pour ses aspects temps réel. Ce n'est qu'en comparant les performances avec ou sans chacune des options que l'on peut optimiser le système final.

Économies d'énergie

Nous allons nous intéresser à présent à un paramètre important pour les systèmes embarqués : la consommation d'énergie électrique. Sur la plupart des systèmes interactifs, la charge du processeur varie en permanence entre des états d'intense activité (calcul, compilation, compression, encodage, etc.) et des périodes de repos durant lesquelles aucune opération n'a lieu, jusqu'à la prochaine sollicitation provenant de l'utilisateur.

En fait, sur un système de type bureautique, la charge est généralement très basse, et la puissance complète du processeur n'est nécessaire que pendant des pics d'activité ponctuels. Voici, par exemple, la charge de la station Linux sur laquelle je rédige ces lignes :

```
$ uptime
11:44:07 up 29 days, 2:26, 9 users, load average: 0.11, 0.10, 0.08
$
```

La commande `uptime` permet de voir la charge du système – le nombre moyen de tâches en exécution ou en attente d'exécution – pour la dernière minute (`0.11` ici), les cinq (`0.10`) et quinze (`0.08`) dernières minutes. Nous pouvons estimer que sur cet exemple, le processeur n'est utilisé en moyenne que 10 % du temps environ. Naturellement, il y a d'autres moments où la charge peut-être beaucoup plus élevée (compilation d'un noyau Linux, par exemple).

Un processeur qui fonctionne à plein rendement consomme de l'énergie et s'échauffe sensiblement (ce qui nécessite la mise en marche d'un ventilateur consommant à son tour de l'énergie électrique). Pour limiter la consommation électrique – améliorant ainsi l'autonomie des systèmes embarqués – et retarder le vieillissement prématuré du processeur surchauffé, plusieurs techniques sont utilisées de nos jours.

Copyright © 2015 Eyrolles.

Variation de fréquence d'horloge

Pour les processeurs récents, il est possible de contrôler la fréquence de fonctionnement du CPU (par exemple, grâce à la technologie *SpeedStep* d'Intel ou *Cool'n'Quiet* d'AMD). Lorsque la charge est faible, on se contentera d'une basse vitesse d'exécution du code. Si la charge augmente, on pourra élever (suivant divers critères) la fréquence d'interprétation des instructions machine. La consommation électrique du processeur est proportionnelle à sa fréquence. Le noyau Linux est tout à fait capable de prendre en charge cette variation de fréquence, en utilisant un algorithme dont l'heuristique (le *governor*) est configurable par l'administrateur (par le biais du pseudo-fichier `/sys/devices/system/cpu/cpuNN/cpufreq/scaling_governor`, `NN` étant le numéro de CPU). La liste des heuristiques disponibles se trouve dans le pseudo-fichier `/sys/devices/system/cpu/cpuNN/cpufreq/scaling_available_governors`. Il existe, par exemple, les heuristiques suivantes :

- `powersave` : utiliser toujours la fréquence la plus basse pour limiter la consommation (utile pour un ordinateur portable sur batterie) ;
- `performance` : utiliser toujours la fréquence la plus haute pour optimiser les temps de traitement (centres de calcul, serveurs d'applications, etc.) ;
- `ondemand` : faire varier automatiquement la fréquence en fonction de la charge système pour ajuster la puissance disponible aux demandes de l'utilisateur (poste de travail) ;
- `conservative` : comme `ondemand`, mais en minimisant les changements de fréquence ;
- `userspace` : laisser l'administrateur paramétrer la fréquence depuis l'espace utilisateur (en fixant dans `/sys/devices/system/cpu/cpuNN/cpufreq/scaling_cur_freq` une des valeurs proposées dans `/sys/devices/system/cpu/cpuNN/cpufreq/scaling_available_frequencies`).

Ainsi, des lignes de commandes telles que :

```
# cd /sys/devices/system/cpu/0/cpufreq
# echo powersave > scaling_governor
```

permettront de basculer le processeur 0 sur une fréquence réduite économisant l'énergie de la batterie.

Dans le cadre d'un système devant répondre à des contraintes temps réel, ce comportement devra bien évidemment être pris en considération. Plusieurs approches sont possibles :

- bloquer en permanence le processeur à sa fréquence la plus élevée en basculant dès le *boot* du système sur le *governor* `performance`. Cette solution est la plus coûteuse en énergie électrique (donc en autonomie sur batterie), et nous perdons l'intérêt de la variabilité de la fréquence du processeur ;
- utiliser dès le *boot* le *governor* `powersave` qui basculera le processeur sur sa fréquence la plus faible et la plus économique. Bien qu'économique en énergie et en usure du processeur, cette approche perd tout l'intérêt de la variation de fréquence, et des vitesses d'exécution plus rapides disponibles ;
- laisser le *governor* `ondemand` choisir pour nous la fréquence adéquate. Cette méthode est *a priori* séduisante, mais nous n'avons aucune garantie de la réactivité du système lors d'une

Copyright © 2015 Eyrolles.

demande importante de charge CPU. Nous allons devoir effectuer quelques expériences pour vérifier le comportement du noyau Linux avec cette heuristique.

Le programme suivant fixe le *governor* indiqué en argument sur le CPU 0, puis lance un thread temps réel sur ce CPU. Le thread va mesurer une série de durées de boucles actives.

```
exemple-governor-01.c
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

#define NB_ITERATIONS_PAR_BOUCLE  200000
#define NB_BOUCLES                10000

static struct timeval tv[NB_BOUCLES+1];

int main(int argc, char * argv[])
{
  int fd;
  int i, j;
  char filename[256];
  long long int duree;
  struct sched_param param;

  if (argc != 2) {
    fprintf(stderr, "usage: %s <governor>\n", argv[0]);
    exit(1);
  }

  sprintf(filename,
 "/sys/devices/system/cpu/cpu%d/cpufreq/scaling_governor",
         sched_getcpu());
  fd = open(filename, O_RDWR);
  if (fd < 0) {
    perror(filename);
    exit(2);
  }
  if (write(fd, argv[1], strlen(argv[1])) < 0) {
    perror(argv[1]);
    exit(2);
  }
  close (fd);

  param.sched_priority = 10;
  if (sched_setscheduler(0, SCHED_FIFO, & param) != 0) {
    perror("pthread_attr_setschedpolicy");
```

Copyright © 2015 Eyrolles.

```
    exit(2);
  }

  sleep(1);

  for (i = 0; i < NB_BOUCLES; i ++) {
    gettimeofday(& (tv[i]), NULL);
    for (j = 0; j < NB_ITERATIONS_PAR_BOUCLE; j ++)
      ;
  }
  gettimeofday(& (tv[i]), NULL);

  for (i = 0; i < NB_BOUCLES; i ++) {
    duree  = tv[i+1].tv_sec - tv[i].tv_sec;
    duree *= 1000000;
    duree += tv[i+1].tv_usec - tv[i].tv_usec;
    printf("%6lld\n", duree);
  }
  return EXIT_SUCCESS;
}
```

En fin de programme, la durée des boucles est affichée sur la sortie standard. Nous nous en servirons avec nos outils habituels.

Heuristique performance

Commençons par lancer notre programme avec le *governor* `performance`. Cela va nous permettre de calibrer le nombre d'itérations par boucle active en fonction du processeur. Disons qu'une durée de quelques centaines de microsecondes représente une valeur convenable. Le programme doit être lancé depuis l'identité *root* pour modifier le *governor* et accéder à une priorité temps réel.

Le test est réalisé sur un Raspberry Pi 2.

```
# ./exemple-governor-01 performance | ../chapitre-04/calculer-statistiques
 Nb mesures = 10000
 Minimum = 2000
 Maximum = 2030
 Moyenne = 2003
 Ecart-type = 4
#
```

Sur ce poste, la durée des boucles à fréquence maximale est de 2 003 microsecondes en moyenne.

Copyright © 2015 Eyrolles.

Heuristique powersave

Essayons à présent avec le *governor* économique afin de voir la durée des boucles avec la fréquence minimale :

```
# ./exemple-governor-01 powersave | ../chapitre-04/calculer-statistiques
 Nb mesures = 10000
 Minimum = 3000
 Maximum = 3051
 Moyenne = 3006
 Ecart-type = 8
#
```

Dans ce cas, la durée moyenne des boucles est en moyenne de 3 006 microsecondes, soit 1,5 fois la durée en mode performance. Ceci est cohérent avec les fréquences de fonctionnement de ce processeur :

```
# cd /sys/devices/system/cpu/cpu0/cpufreq/
# cat scaling_available_frequencies
600000 900000
#
```

Nous disposons donc facilement de deux modes de fonctionnement pour notre processeur : un mode économique pour les périodes où le système est au repos ou exécute des tâches de fond peu intenses, et un mode performant où le processeur fonctionne au maximum de ses possibilités. Dans certains cas, il peut être suffisant de programmer la fréquence de fonctionnement au *boot* et de la laisser inchangée pour la suite.

Toutefois, dans de nombreuses applications temps réel embarquées, il est important de bénéficier à la fois d'une consommation électrique aussi faible que possible et d'une puissance CPU importante pendant les pointes de traitement. Vérifions donc le comportement du *governor* qui effectue les basculements automatiquement.

Heuristique ondemand

Le test avec l'heuristique `ondemand` nous montre une augmentation dynamique de la fréquence afin de répondre à la demande soutenue de temps CPU :

```
# ./exemple-governor-01 ondemand | ../chapitre-04/calculer-statistiques
 Nb mesures = 10000
 Minimum = 2000
 Maximum = 3049
 Moyenne = 2022
 Ecart-type = 139
#
```

Copyright © 2015 Eyrolles.

Nous voyons bien que les premières occurrences de la boucle se déroulent à la fréquence basse du système, ce qui conduit à des durées de 3 000 microsecondes de moyenne, puis la fréquence du CPU augmente pour obtenir des boucles de 2 000 microsecondes.

Sur le Raspberry Pi 2, il n'y a que deux fréquences disponibles, aussi les variations sont-elles brutales. Sur d'autres systèmes, il peut y avoir des transitions plus douces. Par exemple, sur un PC portable, une gamme plus large de fréquences est disponible :

```
$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
2400000 2300000 2200000 2100000 2000000 1900000 1800000 1700000 1600000 1500000
1400000 1300000 1200000
```

Il existe alors une autre heuristique, *conservative*, qui retarde les transitions entre fréquences pour obtenir des changements plus rares.

Il faut être conscient que ces manipulations sont très sensibles à l'activité générale du système et à la charge du CPU concerné. Si l'on réitère cette manipulation à plusieurs reprises, les résultats peuvent beaucoup varier, notamment le délai avant que le thread décide de beaucoup augmenter la fréquence processeur en réponse à la demande de temps CPU. En général, nous trouvons une latence de plusieurs millisecondes entre le début de la phase active et le changement de fréquence. C'est habituellement trop pour une application temps réel.

Dans le cas d'un système temps réel embarqué, où nous voulons concilier l'économie de batterie avec une fréquence réduite pour les phases de repos et la puissance processeur pour répondre aux charges de calcul importantes, nous devrons adapter nous-même la fréquence en basculant d'un *governor* `powersave` à un *governor* `performance` à la demande.

Conclusion

Le noyau Linux est un projet en développement actif. Ses fonctionnalités temps réel, bien qu'elles intéressent un public assez réduit, ne font pas exception et leur amélioration est permanente. Lorsqu'on désire ajuster les performances temps réel de Linux, plusieurs solutions existent, à commencer par l'application du patch PREEMPT_RT. Dans ce cas, il est important d'utiliser des outils de mesure pour valider les améliorations obtenues. D'autres alternatives peuvent également s'offrir au concepteur de systèmes temps réel, comme Xenomai ou RTAI, que nous étudierons dans les prochains chapitres.

Points clés

- Avec le noyau Linux standard, le traitement de certaines interruptions est parfois long, au détriment des tâches utilisateur, même temps réel.
- L'évolution actuelle des gestionnaires d'interruptions vers les threaded interrupt améliorera le comportement de Linux, mais le patch PREEMPT_RT permet de bénéficier de fonctionnalités avancées.

Copyright © 2015 Eyrolles.

- Il est important de disposer de moyens permettant de mesurer les performances d'un système temps réel pendant sa mise au point. L'outil `cyclictest` et ceux qui l'accompagnent permettent de mesurer et valider les progrès durant cette phase.
- L'énergie consommée par le processeur, cruciale en environnement embarqué, est gérée par un sous-système de Linux utilisant des heuristiques configurables pour ajuster la fréquence du processeur.

Exercices

Exercice 1 (**)

Appliquez le dernier patch PREEMPT_RT sur le noyau correspondant, compilez-le pour votre machine en suivant les recommandations de l'annexe A, et démarrez votre système sur ce nouveau kernel.

Exercice 2 (**)

Reprenez le programme `exemple-timer-create-02` du chapitre 4 et faites-le fonctionner sur un système PREEMPT_RT. Observez-vous des différences notables dans les performances ?

Exercice 3 (**)

Écrivez un programme qui fasse une boucle active (limitée dans le temps) et faites-le tourner sur un noyau Linux standard avec un ordonnancement temps réel et une priorité élevée. Votre système répond-il au `ping` depuis une autre machine ? Qu'en est-il sur un système PREEMPT_RT ?

Exercice 4 (***)

Repérez l'interruption correspondant à votre interface réseau. Sur un système PREEMPT_RT, identifiez le kernel thread qui gère cette interruption. En faisant varier la priorité de ce thread, arrangez-vous pour que votre système réponde à nouveau au `ping`.

Exercice 5 (***)

Écrivez un programme qui boucle pendant 5 secondes et affiche ensuite le nombre d'itérations qu'il a réalisé. Testez-le sur un processeur avec un *governor* `performance`, `powersave` et `ondemand`, en observant les variations de puissance CPU.

Copyright © 2015 Eyrolles.

9

Extensions temps réel de Linux

Jusqu'à présent, nous avons étudié le noyau Linux standard (celui que l'on nomme *Vanilla Kernel*) distribué par Linus Torvalds, ainsi que le patch PREEMPT_RT de Thomas Gleixner qui améliore le comportement temps réel natif de Linux. Il existe également une seconde approche pour optimiser les performances et se rapprocher des systèmes temps réel stricts, s'appuyant sur un *nanokernel* plus prioritaire que Linux, comme le proposent les projets RTLinux, RTAI et Xenomai. C'est ce dernier que nous développerons plus en détail dans les chapitres à venir.

Les nanokernels temps réel

Principes

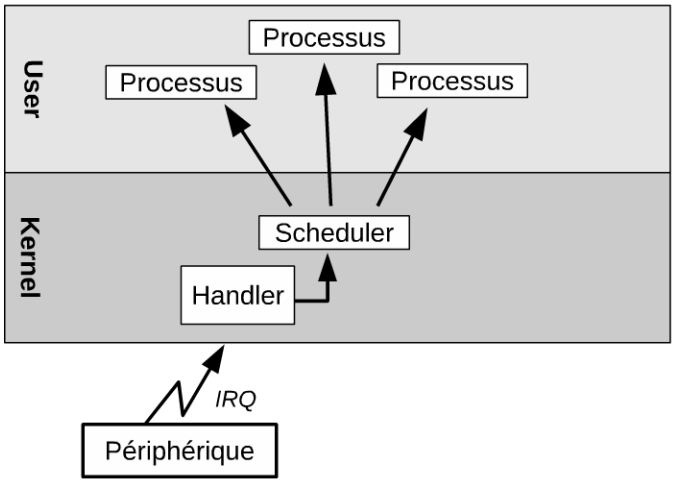
Si nous reprenons le principe de fonctionnement général d'un système Linux, nous pouvons observer son comportement sur la figure 9-1.

L'interactivité du système est organisée autour de l'arrivée d'interruptions depuis des périphériques externes (clavier, souris, disque, ports de communication, cartes d'acquisition, etc.) et des composants internes du processeur (MMU, timers matériels, etc.). Le traitement des interruptions est réalisé dans des handlers qui réveillent les tâches de l'espace utilisateur et les threads du kernel. L'appel de l'ordonnanceur – qui s'effectue à des moments variables suivant la configuration de la préemptibilité du noyau – entraîne la commutation entre les tâches en fonction de leurs priorités temps réel et temps partagé.

Copyright © 2015 Eyrolles.

Figure 9-1

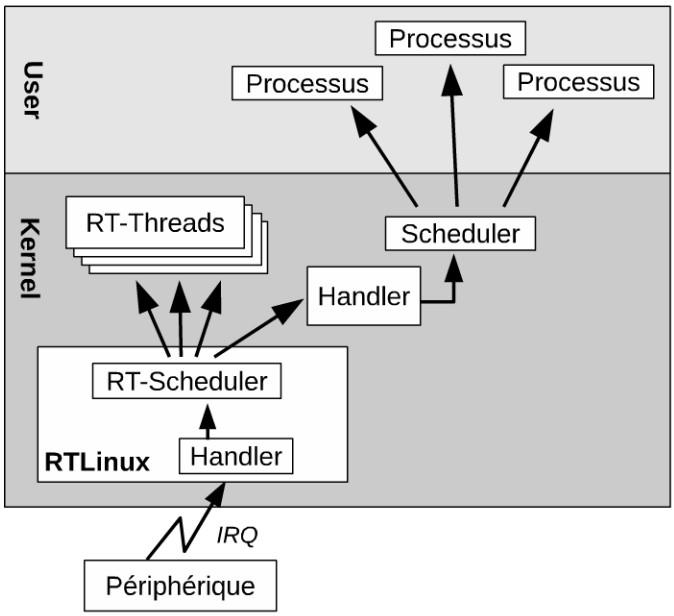
Ordonnancement dans le noyau Linux standard



L'idée d'améliorer le comportement de Linux pour supporter les contraintes du temps réel strict est apparue dans le courant des années 1990, principalement à travers le projet RTLinux de Victor Yodaiken et Michael Barabanov. Le concept est représenté sur la figure 9-2.

Figure 9-2

Ordonnancement avec RTLinux



Copyright © 2015 Eyrolles.

RTLinux

Une couche logicielle supplémentaire vient s'intercaler entre le contrôleur d'interruption et les gestionnaires du noyau Linux. En pratique, le handler invoqué par le processeur lorsque survient une interruption ne dépend plus du noyau Linux original mais d'un ensemble de routines ajoutées dans le noyau par un patch. Ce handler activera un petit ordonnanceur simple et déterministe capable de commuter entre des tâches temps réel se trouvant dans l'espace mémoire du kernel Linux (les *RT-Threads*).

Lorsque les traitements temps réel sont terminés, l'ordonnanceur RTLinux rend le contrôle au noyau Linux original, en invoquant éventuellement le handler concerné par l'interruption reçue.

> Le projet RTLinux, développé initialement à l'université du Nouveau-Mexique, est très rapidement devenu un produit commercial, supporté par la société FSMLabs, fondée par Victor Yodaiken et Michael Barabanov, et rachetée en 2007 par Wind River Systems.
>
> Plutôt bien accueilli initialement dans la communauté Linux, ce projet s'est trouvé largement critiqué lorsque Victor Yodaiken a déposé un brevet, validé en 1999 sous le numéro 59957450, pour réserver l'usage d'un « système d'exploitation temps réel exécutant des tâches temps réel et exécutant un système d'exploitation généraliste comme tâche de fond ». Ceci pour contrer la concurrence de projets commerciaux ou libres, comme RTAI dans sa version initiale.

Le principe de fonctionnement est de découpler précisément le système en tâches à contraintes temporelles fortes qui seront ordonnancées par le noyau temps réel strict, et en tâches de moindres priorités qui pourront être exécutées en temps réel souple ou temps partagé par le noyau Linux. Naturellement, les tâches peuvent communiquer entre elles, par l'intermédiaire de buffers de mémoire partagée, de Fifo temps réel, de mutex, etc.

On peut imaginer un système d'acquisition et de datation de données (opération assez courante dans les environnements industriels). Les tâches de réception et d'horodatage des données sont ordonnancées par le micronoyau temps réel. Ensuite, les données sont transmises *via* des Fifo temps réel (dans lesquelles l'écriture du côté kernel n'est pas bloquante) à des processus de l'espace utilisateur, chargés d'assurer l'enregistrement sur le disque, le traitement (compression, décryptage, etc.) des informations, la présentation graphique du résultat, la diffusion sur le réseau, etc.

De nos jours, il existe deux versions de RTLinux :

- *Open RTLinux*, version simplifiée dont l'utilisation est limitée à des projets sous licences GPL ou compatibles ;
- *Wind River Real-Time Core RTLinux*, produit commercial enrichi par un environnement de développement et du support technique, utilisable pour développer des projets propriétaires.

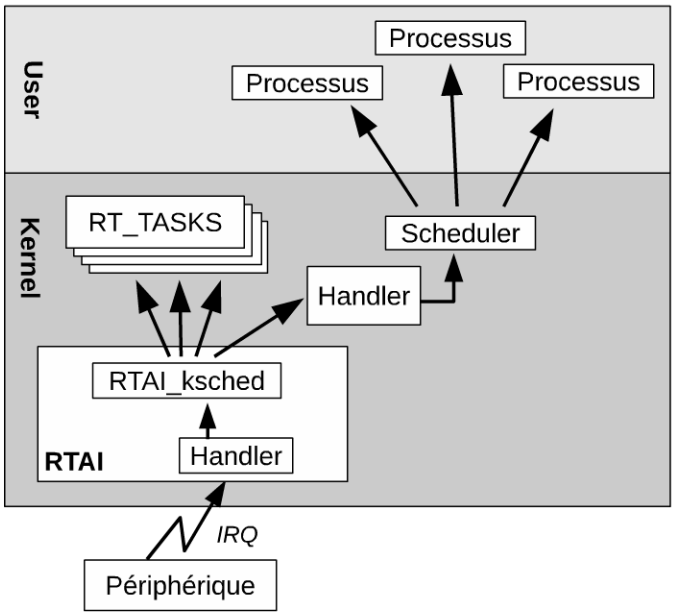
RTAI et Adeos

Face à l'orientation de plus en plus commerciale de RTLinux dans les années 1990, une équipe de l'université polytechnique de Milan, menée par Paolo Mantegazza, a mis au point le projet RTAI *(Real Time Application Interface)* utilisant le même principe que RTLinux (figure 9-3).

Copyright © 2015 Eyrolles.

Figure 9-3

RTAI première génération



Gratuit et sous licence libre (LGPL), RTAI se développa rapidement avec une interface de programmation cohérente et facile d'accès (elle était assez proche des fonctions de la norme Pthreads).

Lorsque le brevet de FSMLabs fut dévoilé, l'avenir de RTAI sembla soudainement incertain : les restrictions d'usage du modèle employé par RTAI pesaient sur les possibilités d'évolution et d'utilisation de ce système pour des projets industriels.

En 2001, Karim Yaghmour publia un article décrivant un mécanisme de gestion des interruptions, nommé Adeos *(Adaptative Domain Environment for Operating Systems)* [YAGHMOUR 2001]. L'idée sous-jacente, très proche de ce que l'on nomme aujourd'hui « virtualisation », consiste à disposer d'une couche logicielle basse qui capture les interruptions matérielles et les envoie dans un pipeline (figure 9-4). Plusieurs systèmes d'exploitation fonctionnent en parallèle sur cette machine (en se répartissant correctement les ressources matérielles). Chacun est installé dans un domaine spécifique et reçoit les interruptions en fonction de sa position sur le pipeline.

On peut alors imaginer faire fonctionner – dans un environnement servi en premier lieu – un petit système d'exploitation déterministe, et dans un second environnement – servi par la suite –, un système d'exploitation comme Linux, sans contrainte temporelle particulière.

Bien entendu, ce concept que l'on observe sur la figure 9-5 fournira les mêmes fonctionnalités que RTLinux. Toutefois, la présentation d'Adeos ne parle ni d'ordonnanceur temps réel, ni de système d'exploitation généraliste fonctionnant en tâche de fond. Ainsi, il n'entre pas dans les limites du brevet de FSMLabs.

Copyright © 2015 Eyrolles.

Figure 9-4

Concept du mécanisme Adeos

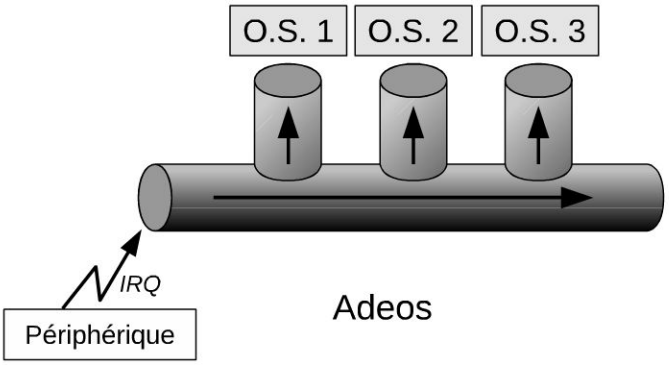
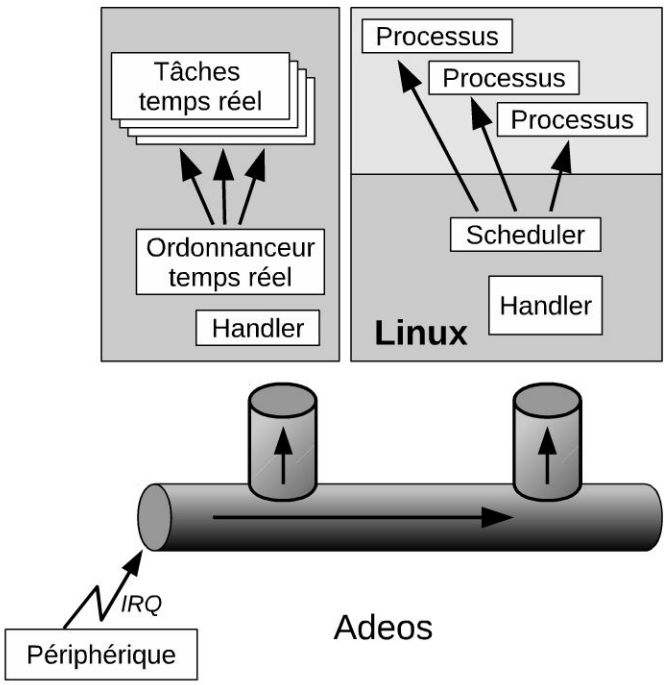


Figure 9-5

Adeos et le temps réel



L'implémentation d'Adeos fut réalisée par Philippe Gerum – talentueux développeur français –, puis il fut intégré comme couche basse de RTAI. Adeos n'existe pas en tant que système indépendant, mais vient se greffer dans le noyau Linux classique pour profiter de son initialisation, de la gestion du matériel, et intercepte les IRQ avant que les handlers de Linux ne les voient.

Le résultat est donc très proche de celui que l'on obtient avec l'implémentation de la figure 9-2. Les interruptions capturées sont transmises à un premier domaine (au sens Adeos) qui contient l'ordonnanceur de RTAI. Celui-ci active les threads temps réel qui s'exécutent dans l'espace mémoire du kernel, avec les privilèges de ce dernier (pour l'accès aux port d'entrées-sorties, par exemple).

Copyright © 2015 Eyrolles.

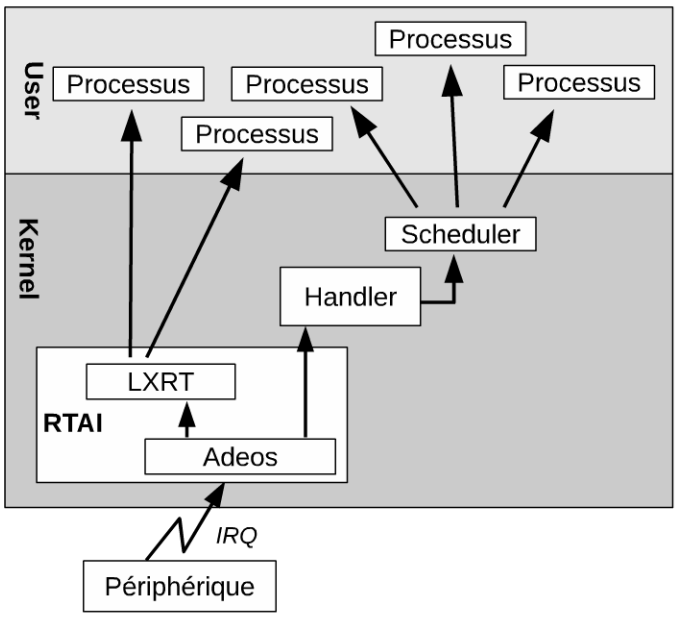
Lorsque l'ordonnanceur de RTAI a fini d'activer les tâches temps réel, il rend le contrôle à Adeos qui enverra les interruptions au domaine suivant du pipeline : le noyau Linux classique dont le scheduler ordonnancera les tâches temps réel souple et temps partagé. Le noyau Linux n'a plus la possibilité de couper les interruptions matérielles comme il le faisait auparavant (notamment dans les verrouillages de spinlocks) ; en remplacement, il verrouillera l'équivalent d'un mutex demandant temporairement à Adeos de ne plus lui transmettre d'interruptions.

Les premières versions de RTAI ne pouvaient faire fonctionner que des tâches se trouvant dans l'espace kernel de Linux. Pour cela, le code du thread temps réel était écrit sous forme de module, puis chargé dans le noyau (avec la commande `insmod`). Ceci permettait d'être sûr que le thread serait toujours chargé en mémoire lorsqu'il fallait l'activer (le code du noyau et ses données sont toujours présents en mémoire physique) et qu'il disposerait des privilèges nécessaires pour réaliser des opérations d'entrés-sorties vers les périphériques.

L'évolution suivante consista à faire fonctionner des tâches temps réel non seulement dans l'espace kernel, mais également à ordonnancer des processus de l'espace utilisateur. Pour cela, un module spécifique de RTAI, nommé LXRT, supervise des processus de l'espace utilisateur. Ces derniers peuvent demander explicitement, avec un appel système spécifique nommé `make_hard_realtime()`, à être ordonnancés par LXRT. À partir de ce moment, Adeos n'envoie plus d'interruptions au domaine Linux, jusqu'à ce que le processus invoque `make_soft_realtime()`.

Figure 9-6

Processus temps réel RTAI



Xenomai

Ce principe est également celui du projet Xenomai, créé par Philippe Gerum. Intégré initialement dans RTAI, sous le nom RTAI/Fusion, Xenomai est devenu indépendant en 2005 lorsque des divergences de vue sont apparues avec l'équipe originale de RTAI.

Copyright © 2015 Eyrolles.

Dans le premier domaine d'Adeos, est installé un ordonnanceur simple et déterministe, nommé Nucleus (qui n'a rien à voir avec le système Nucleus-RTOS de Mentor Graphics). Dans le second domaine, se trouve le noyau Linux classique. Dans un processus de l'espace utilisateur, on peut démarrer un thread reconnu par Nucleus, qui s'exécutera avec une priorité temps réel supérieure à celles des tâches du noyau Linux. Tant que le thread réclamera du temps CPU, le kernel Linux sera suspendu et ne recevra aucune interruption. Ce n'est qu'une fois tous les threads de Nucleus endormis que Linux verra arriver les interruptions survenues entre temps.

Le problème principal avec ce type de construction est le risque de voir le thread temps réel invoquer des appels système Linux et rendre le contrôle au noyau. Ce dernier serait alors susceptible de le préempter pour commuter vers une autre tâche, nous entraînant dans un cas évident d'inversion de priorité. En outre, les interruptions ne parvenant plus au kernel Linux, ce dernier n'est plus à même d'effectuer les commutations (ni de gérer correctement la mémoire virtuelle ou les entrées-sorties vers les disques, par exemple) et un blocage définitif du système est probable très rapidement. Les premières versions de RTAI/LXRT interdisaient totalement ces appels système dans la portion temps réel strict.

Avec Xenomai, une autre approche a été adoptée autorisant les processus à faire des appels système, durant lesquels ils seront temporairement ordonnancés par Linux. Pour cela, Adeos fait circuler dans son pipeline des événements autres que les interruptions, comme les invocations d'appels système ou les déclenchements du scheduler Linux. Les threads de Xenomai peuvent s'exécuter selon deux modes :

- le mode *primaire* est celui où le thread est directement sous le contrôle de Nucleus ;
- le mode *secondaire* voit le thread ordonnancé par le noyau Linux.

Les migrations d'un mode à l'autre sont automatiquement prises en charge par Adeos.

Le thread Xenomai débute son existence par un appel `rt_task_create()` suivi d'un `rt_task_start()` invoqués dans un processus Linux. L'exécution commence alors en mode primaire, et le thread restera dans ce mode tant qu'il sera actif dans l'espace utilisateur (à consommer du temps CPU). Il se trouvera en concurrence avec les autres threads Xenomai, et pourra être préempté par les handlers des interruptions gérées par celui-ci. Pendant tout ce temps, le noyau Linux reste suspendu.

Supposons à présent que notre thread effectue un appel système vers le noyau Linux (par exemple, un `write()` vers un périphérique). Le déclenchement de l'appel système se traduit par un événement Adeos qui circule dans le pipeline. Xenomai le reçoit et fait migrer le processus en mode secondaire. Le noyau Linux est alors réactivé pour répondre à l'appel système, avec le risque que celui-ci préempte la tâche pour en activer une autre plus prioritaire

> Dans les versions précédentes, les interruptions restaient toujours en attente (dans un domaine intermédiaire du pipeline qui sert de « bouclier ») et n'étaient envoyées à Linux qu'une fois le processus endormi dans l'appel système. Ceci n'est plus vrai de nos jours.

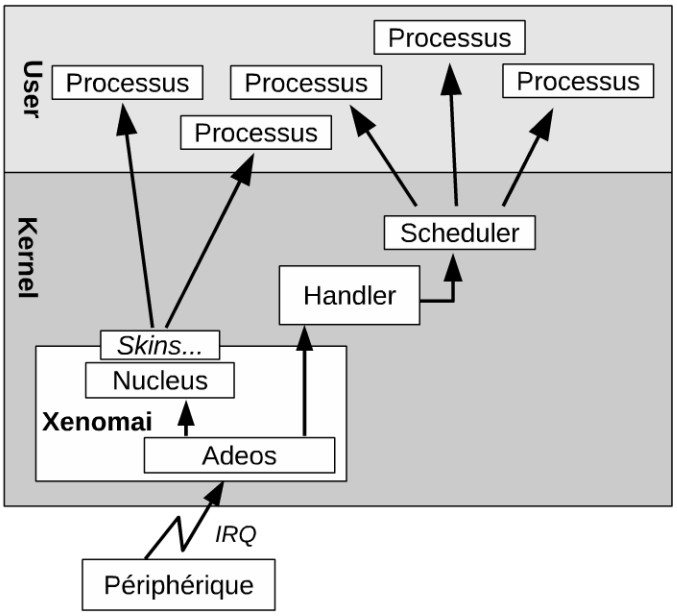
Inversement, lorsque le kernel reçoit une interruption qui entraîne le réveil du processus et son retour dans l'espace utilisateur, Xenomai le fait à nouveau migrer, cette fois vers le mode primaire.

Copyright © 2015 Eyrolles.

L'implémentation d'Adeos et Nucleus est réalisée avec un soin particulier pour garantir les temps de latence les plus faibles entre l'arrivée d'une interruption et sa prise en charge par le processus concerné.

Figure 9-7

Processus avec Xenomai



Interface de programmation

Une innovation intéressante fut la mise à disposition de *skins* (peaux) différentes pour le système. Les concepteurs ont défini une interface de programmation (API) pour Xenomai, en tenant compte de toutes les fonctionnalités nécessaires pour un système temps réel :

- gestion des tâches : création, terminaison, attente, mise en pause, modification de paramétrage, etc. ;
- synchronisation : sémaphores, mutex, événements, variables, conditions, etc. ;
- communications entre tâches : files de messages, tubes, mémoire partagée, etc. ;
- gestion du temps : activation périodique, mesure de l'heure, etc. ;
- traitement des interruptions.

Une fois cette interface – dite API *native* – développée, ils ont créé des surcouches permettant de simuler l'API d'autres systèmes temps réel en s'appuyant sur elle.

Ainsi, il devient facile de porter sous Xenomai du code développé pour d'autres environnements. Les principales skins sont les suivantes :

- VxWorks : système temps réel, propriété de Wind River Systems, l'un des acteurs majeurs dans le domaine industriel ;
- VRTX *(Versatile Realtime Executive)* : appartient à Mentor Graphics ;

Copyright © 2015 Eyrolles.

- uITRON « micro-ITRON » : version des spécifications temps réel ITRON pour les systèmes embarqués ;
- PSOS : développé initialement pas SCG *(Software Components Group)*, ce système concurrent de VxWorks a été acheté par Wind Rivers en 1999, qui a abandonné son développement ;
- Posix PSE51 : couverture minimale de la norme Posix, qui spécifie les fonctionnalités temps réel multithreads dans un seul processus et sans interactions avec le système de fichiers.

Installation de Xenomai

Modification du noyau Linux

Le support du Raspberry Pi 2 est incomplet au moment de la rédaction de ces lignes. J'ai donc choisi de mener cette expérience sur un Raspberry 1 modèle B+.

La première opération à réaliser est la compilation d'un noyau Linux modifié, dans lequel Adeos et Xenomai sont intégrés par un patch. Pour cela, nous devons commencer par télécharger les sources de Xenomai et les décompresser. Je choisis la version stable la plus récente au moment de la rédaction de ces lignes :

```
# cd /usr/src/
# wget http://xenomai.org/downloads/xenomai/stable/xenomai-2.6.4.tar.bz2
http://xenomai.org/downloads/xenomai/stable/xenomai-2.6.4.tar.bz2
[...]
100%[==============================>] 21 193 853  646K/s ds 32s
# tar xjf xenomai-2.6.4.tar.bz2
```

Dans le répertoire `ksrc/arch/` de Xenomai, se trouvent des sous-répertoires pour les différentes architectures supportées :

```
# ls xenomai-2.6.4/ksrc/arch/
arm  blackfin  generic  Makefile  nios2  powerpc  sh  x86
```

Nous allons ici installer Xenomai sur une architecture `arm` pour rester cohérent avec les tests réalisés dans les chapitres précédents. Le principe est toutefois identique pour n'importe quelle autre architecture.

Observons le contenu du répertoire `ksrc/arch/arm/patches` pour identifier les noyaux Linux qui sont supportés par cette version de Xenomai.

```
# ls xenomai-2.6.4/ksrc/arch/arm/patches/
beaglebone   ipipe-core-3.8.13-arm-4.patch   README
ipipe-core-3.10.32-arm-5.patch  mxc          zynq
ipipe-core-3.14.17-arm-4.patch               raspberry
```

Copyright © 2015 Eyrolles.

Il y a un support pour trois noyaux : 3.8.13, 3.10.32 et 3.14.17. En outre, des patches supplémentaires devront être ajoutés pour les architectures Beaglebone, Zinq ou Raspberry Pi. Voyons le contenu du répertoire spécifique à ce dernier :

```
$ ls arm/patches/raspberry/
ipipe-core-3.8.13-raspberry-post-2.patch
ipipe-core-3.8.13-raspberry-pre-2.patch
```

Il y a deux patches spécifiques à appliquer avant (`...-pre-2-...`) et après (`...-post-2-...`) le patch `ipipe-core-3.8.13` commun. Téléchargeons l'archive du noyau Raspberry Pi, comme au chapitre précédent, et extrayons la version correcte de Linux :

```
# cd /usr/src/
# git clone https://github.com/raspberrypi/linux linux-xenomai
# cd linux-xenomai
# git checkout  rpi-3.8.y
```

Créons une branche de travail pour isoler les modifications qui seront apportées par les patches à venir :

```
# git branch rpi-3.8.13-xenomai
# git checkout rpi-3.8.13-xenomai
```

J'applique le premier patch spécifique pour Raspberry Pi :

```
# patch -p1 < ../xenomai-2.6.4/ksrc/arch/arm/patches/raspberry/*-pre-*
```

Puis le patch global pour l'architecture `arm` :

```
# patch -p1 < ../xenomai-2.6.4/ksrc/arch/arm/patches/ipipe-core-3.8.13-*
```

Et enfin le dernier patch pour Raspberry Pi :

```
# patch -p1 < ../xenomai-2.6.4/ksrc/arch/arm/patches/raspberry/*-post-*
```

Les sources du noyau ont été mises à jour ; nous devons également modifier les fichiers de configuration. Pour ce faire, Xenomai nous fournit un script auquel on indique les options suivantes :

- `--arch=` : l'architecture cible ;
- `--linux=` : l'emplacement des sources de Linux à modifier.

```
# ../xenomai-2.6.4/scripts/prepare-kernel.sh   --arch=arm   --linux=.
```

Copyright © 2015 Eyrolles.

Compilation de Linux modifié

La compilation du noyau se déroule comme décrit dans l'annexe A.

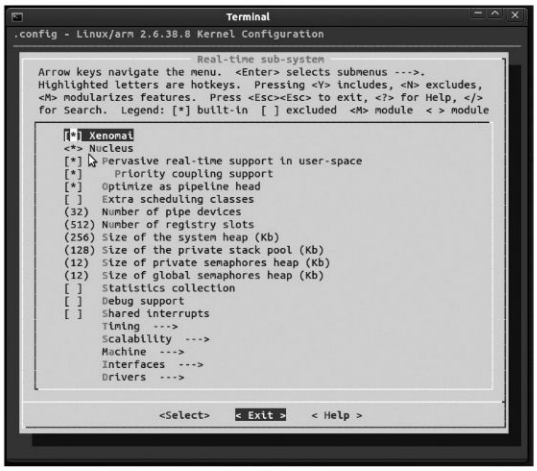
```
# zcat  /proc/config.gz  >  .config
# make  menuconfig
```

Dans le menu de configuration de Linux, nous pouvons observer une nouvelle entrée, Interrupt Pipeline, dans le menu Processor Type and Features (ou Kernel features suivant les architectures). Il s'agit de l'intégration d'Adeos, la couche bas niveau de gestion des interruptions.

Par ailleurs, un nouveau menu global de configuration est apparu, nommé Real-Time Sub-System. Il s'agit de la configuration de Xenomai en tant que module du domaine prioritaire d'Adeos.

Figure 9-8

Menu de configuration de Xenomai



Certaines options de compilation du noyau standard sont incompatibles avec un bon fonctionnement de Xenomai. Il s'agit notamment de l'APM *(Advanced Poxer Management)*, du CPU Frequency Scaling et de la gestion d'énergie par ACPI. Ces options devront être désactivées ; elles se trouvent toutes dans le menu global Power Management. Si l'une d'entre elles reste activée par erreur, un message nous en avertit dans le sous-menu Real-Time Sub-System.

Les deux premières options doivent obligatoirement être activées ; on remarquera néanmoins que le domaine Nucleus peut être compilé sous forme de module que l'on insère après le *boot*.

L'option Pervasive Real-Time Support in User-Space est nécessaire pour pouvoir gérer avec Xenomai des processus de l'espace utilisateur de Linux. En son absence, seuls des threads du kernel pourraient être placés en temps réel strict. L'option Priority Coupling Support est à présent déconseillée : elle permettait de conserver vis-à-vis de Xenomai la priorité des threads migrant en mode secondaire. Cela n'est généralement pas utile.

Comme Xenomai sera toujours le domaine le plus prioritaire du pipeline d'interruptions, des optimisations peuvent être effectuées, si l'option Optimize as Pipeline Head est activée.

Copyright © 2015 Eyrolles.

Nucleus propose des ordonnancements Fifo et Round Robin classiques, mais il permet également, si l'option Extra Scheduling Classes est activée, d'en utiliser de plus rares comme les partitions de temps ou les modes sporadiques. Nous pouvons également régler finement des paramètres du système comme les volumes mémoire réservés pour divers éléments (sémaphores, piles en espace noyau, etc.), ou activer des options de débogage ou d'instrumentation.

Les sous-menus suivants permettent d'agir sur des paramètres spécifiques de Xenomai.

Sous-menu Timing

Deux paramètres principaux sont présents dans ce sous-menu : l'un choisit le mode de fonctionnement des timers (périodiques ou apériodiques) et l'autre préconfigure la latence des interruptions. Si les divers timers employés par les threads temps réel sont tous multiples (exprimés en ticks) d'une période de base bien définie, le mode périodique est le plus adapté ; sinon, un mode apériodique peut être choisi et les durées utilisées par les timers logiciels sont précisées en nanosecondes. Quoi qu'il en soit, la gestion du timer matériel par Xenomai est réalisée par un fonctionnement apériodique. La latence (le délai entre l'arrivée d'une interruption et le réveil du thread utilisateur qui l'attend) peut également être préconfigurée dans ce sous-menu. Une valeur de zéro (par défaut) laisse toutefois le système mesurer automatiquement la latence et la rendre visible dans `/proc/xenomai/latency`.

Sous-menu Scalability

L'option principale de ce sous-menu est l'activation ou non d'un ordonnanceur en temps constant *O(1) scheduler* pour Nucleus. Rappelons que ceci n'a rien à voir avec l'ordonnanceur *O(1)* de Linux dont nous avons parlé au chapitre 3 qui était consacré aux tâches temps partagé et temps réel souple. Il s'agit ici uniquement de l'ordonnancement des threads de Xenomai. Pour un nombre réduit de tâches temps réel actives simultanément (cas le plus fréquent), le surcoût lié à la gestion d'un ordonnanceur *O(1)* ne se justifie pas vraiment.

Le même type de raisonnement s'applique à la seconde option Timer Indexing Method qui propose une organisation des timers sous forme de liste chaînée (méthode linéaire, adaptée à un faible nombre de timers), d'arbre binaire ou de table de hachage (au prix de contraintes sur les périodes de déclenchement).

Sous-menu Machine

Dépendant de l'architecture sous-jacente, ce sous-menu regroupe des paramètres permettant d'optimiser l'utilisation du matériel (coprocesseur mathématique, cache mémoire et TLB, etc.).

Sous-menu Interface

C'est ici que l'on peut choisir d'intégrer les skins qui nous intéressent pour porter des applications existantes sous Xenomai. Les interfaces applicatives sont implémentées dans l'espace utilisateur sous forme de bibliothèques, que nous allons compiler par la suite, mais un support doit être incorporé dans le noyau.

Les API proposées dans cette version sont les suivantes : Native (que nous utiliserons dans le prochain chapitre), Posix, pSOS+, uITRON, VRTX et VxWorks.

Copyright © 2015 Eyrolles.

Par ailleurs, pour pouvoir écrire dans le noyau des drivers bas niveau capables de communiquer avec les tâches Xenomai quelle que soit leur skin, il existe une interface nommée RTDM *(Real Time Driver Model).*

Les sous-menus propres à chaque API permettent de préciser des éléments de configurations spécifiques. L'option Interrupts du sous-menu Native API est désactivée par défaut et disparaîtra probablement dans la prochaine version majeure de Xenomai.

Sous-menu Drivers

Si l'interface RTDM a été activée plus haut, quelques drivers sont disponibles pour Xenomai (par exemple, pour contrôleur série ou CAN). On trouve entre autres des drivers qui permettent d'effectuer des instrumentations de latence, par exemple.

Une fois la configuration réalisée, on lance la compilation du noyau Linux comme d'habitude, voir l'annexe A pour plus de détails.

```
$ make
[…]
  Image arch/arm/boot/zImage is ready
$
```

L'installation du noyau se fait normalement avec le paramétrage d'un *bootloader* auquel on fournit l'image obtenue.

Compilation de Xenomai

Les bibliothèques de Xenomai ainsi que les programmes de tests et de mesures doivent être compilés séparément du noyau Linux. Pour cela, il faut se rendre dans le répertoire source de Xenomai.

Les options à passer au script `configure` dépendent à nouveau de l'architecture. Un fichier `README.INSTALL` est présent pour les décrire. Dans le cas d'une cross-compilation, il faudrait préciser le type de cible.

```
# cd ../xenomai-2.6.4/
# ./configure
[...]
#
```

On peut activer certaines options en fonction de l'architecture, par exemple `--enable-smp` sur les systèmes multicœurs.

Une fois la configuration obtenue, nous pouvons lancer la compilation :

```
# make
[...]
#
```

Copyright © 2015 Eyrolles.

Enfin, nous installons les bibliothèques, les fichiers d'en-tête (pour la compilation des applications), les exécutables de test, etc., dans un sous-répertoire spécifique. Par défaut, l'installation s'effectue dans `/usr/xenomai` et nécessite donc les droits *root*. Il est possible de renseigner la variable d'environnement `DESTDIR` pour installer Xenomai dans un autre répertoire (notamment lorsqu'il s'agit d'une installation pour une cible différente).

```
# make install
```

Les fichiers installés se trouvent répartis dans les sous-répertoires `bin`, `include`, `lib`, `sbin` et `share`.

Expériences avec Xenomai

Première exploration

Après un *reboot* du système sur le noyau Linux modifié, nous ne voyons pas de différence flagrante de comportement. Deux sous-répertoires sont toutefois apparus dans `/proc` :

- `/proc/ipipe` *(Interrupt Pipeline)* contient des informations concernant Adeos et ses domaines ;
- `/proc/xenomai` regroupe des pseudofichiers qui présentent les paramètres du domaine Xenomai et de Nucleus.

Examinons le contenu de `/proc/ipipe` :

```
# cd /proc/ipipe/
# ls
Linux    Xenomai  version
# cat version
4
# cat Linux
        +--- Handled
        |+-- Locked
        ||+- Virtual
 [IRQ]  |||  Handler
    0:  H..  __ipipe_do_IRQ
    1:  H..  __ipipe_do_IRQ
    2:  H..  __ipipe_do_IRQ
[...]
1023:  H..  __ipipe_do_IRQ
 1024:  H.V  __ipipe_flush_printk
 1025:  H.V  __ipipe_do_work
 1026:  H.V  rthal_apc_handler
1027:  ..V
        +--- Handled
        |+-- Locked
        ||+- Virtual
 [IRQ]  |||  Handler
```

Copyright © 2015 Eyrolles.

```
    0:  ...
    1:  ...
    2:  ...
    3:  H..  xnintr_clock_handler
    4:  ...
 [...]
 1023:  ...
 1024:  ..V
 1025:  ..V
 1026:  ..V
 1027:  H.V  xnpod_schedule_handler
#
```

Le domaine Linux est nommé `Root Domain` dans les sources de Xenomai, car il s'agit de l'hôte accueillant Adeos. Il peut recevoir toutes les interruptions réelles ou virtuelles (internes au processeur). Son identifiant (pour Adeos) est 0, et sa priorité 100.

Pour Xenomai, la priorité est maximale car il s'agit du premier domaine du pipeline.

Observons à présent les informations fournies par `/proc/xenomai` :

```
# ls /proc/xenomai/
acct  faults  interfaces  latency   rtdm   schedclasses  timebases  timerstat
apc   heap    irq         registry  sched  stat          timer      version
#
```

Le contenu de ce répertoire peut varier en fonction des options choisies dans le sous-menu realtime sub-system de Linux. Les principaux fichiers en sont :

- `acct` : état brut des tâches temps réel, que l'on retrouve formaté de manière plus lisible dans `stat` ;
- `affinity` : (absent sur le Raspberry Pi unicœur où Xenomai est compilé sans l'option `--smp`) affinité de Xenomai par rapport aux processeurs du système.
- `apc` : informations sur les *Asynchronous Procedure Call*, c'est-à-dire les traitements par Linux de la fin d'une interruption gérée par Xenomai ;
- `faults` : liste des exceptions déclenchées par Xenomai sur chaque processeur ;
- `heap` : statistiques sur l'utilisation de la mémoire par Xenomai et les composants employés par l'API (sémaphores, pile, etc.) ;
- `interfaces/` : dans ce répertoire, un pseudofichier est présent pour chaque skin de Xenomai, indiquant un compteur d'usage ;
- `irq` : liste des interruptions gérées par Xenomai avec leur nombre d'occurrences sur chaque CPU ;
- `latency` : latence mesurée en nanosecondes entre le déclenchement d'une interruption et le début du traitement dans un handler de Xenomai ;
- `rtdm/` : les fichiers contenus dans ce sous-répertoire fournissent des informations sur les drivers Xenomai employant l'interface RTDM ;

Copyright © 2015 Eyrolles.

- `sched` : liste des tâches temps réel avec leur état et les informations sur les timers qu'elles emploient ;
- `stat` : états des tâches du système, assez proche de ce que l'on trouve dans `sched`, avec en outre le nombre de *Context Switches* et de *Mode Switches* (commutation du mode primaire en secondaire et inversement) ;
- `timebases` : bases de temps utilisées par chaque skin ;
- `timer` : informations sur le timer disponible pour Xenomai ;
- `timerstat/` : statistiques sur les déclenchements des timers utilisés par les différentes skins ;
- `version` : numéro de version de Xenomai (2.6.4 dans notre cas).

Programmes de tests

Xenomai est livré avec quelques applications de tests, qui se trouvent dans `/usr/xenomai/bin`, sauf si vous avez utilisé l'option `--prefix` du script `configure`.

```
# cd /usr/xenomai/bin/
# ls
arith                insn_read            wakeup-time
check-vdso           insn_write           wf_generate
clocktest            irqloop              wrap-link.sh
cmd_bits             klatency             xeno
cmd_read             latency              xeno-config
cmd_write            mutex-torture-native xeno-regression-test
cond-torture-native  mutex-torture-posix  xeno-test
cond-torture-posix   rtcanrecv            xeno-test-run
cyclictest           rtcansend            xeno-test-run-wrapper
dohell               rtdm
insn_bits            switchtest
#
```

Les bibliothèques de Xenomai se trouvent dans `/usr/xenomai/lib`. Elles sont indispensables pour lancer une tâche dans l'espace utilisateur. Parfois, il est nécessaire de remplir la variable d'environnement `LD_LIBRARY_PATH` (initialement vide) en indiquant ce chemin supplémentaire où le chargeur devra rechercher les bibliothèques dynamiques.

Ensuite, nous allons exécuter un premier test :

```
# /usr/xenomai/bin/latency
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|     -6.000|     -5.000|     13.000|       0|     0|     -6.000|     13.000
RTD|     -7.000|     -5.000|     14.000|       0|     0|     -7.000|     14.000
```

Copyright © 2015 Eyrolles.

```
[...]RTT|  00:45:10  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|     -7.000|     -5.000|     14.000|       0|     0|     -8.000|     36.000
RTD|     -7.000|     -5.000|     16.000|       0|     0|     -8.000|     36.000
RTD|     -7.000|     -5.000|     13.000|       0|     0|     -8.000|     36.000
RTD|     -7.000|     -5.000|     15.000|       0|     0|     -8.000|     36.000
(Contrôle-C)
RTS|     -7.000|     -5.000|     36.000|       0|     0|    00:45:14/00:45:14
#
```

Ce programme a tourné pendant quarante-cinq minutes sur un Raspberry Pi avec une charge moyenne (quelques opérations sur le système de fichiers et observations du comportement du programme). Un timer matériel est déclenché toutes les 100 microsecondes et l'application vérifie l'instant auquel le gestionnaire d'interruptions est invoqué. Nous voyons que sur notre système, il y a un décalage, une latence, entre l'instant attendu et le véritable déclenchement. La latence la plus faible obtenue est négative – le déclenchement a eu lieu plus tôt que prévu car Xenomai anticipe les retards –, et la pire (c'est la valeur qui importe vraiment dans un contexte temps réel strict) de 36 microsecondes.

Il s'agit d'une situation moyenne car la charge système est normale et un nombre raisonnable d'interruptions se sont déclenchées. Nous aimerions mesurer ces paramètres dans des circonstances plus extrêmes. Si nous connaissons approximativement les conditions moyennes de fonctionnement du système final (nombre de processus, débit réseau, fréquence des interruptions, etc.), nous pouvons effectuer des tests de longue durée dans un environnement beaucoup plus perturbé.

Xenomai est fourni avec un script nommé `dohell` qui place le système dans des conditions infernales... En exécutant de nombreuses commandes en parallèle, il provoque une activité importante des processus et le déclenchement de fréquentes interruptions (réseau, disque...). Il est configurable grâce à différentes options :

Options	Signification
`-b` *hackbench*	Chemin d'accès au programme de tests `hackbench`. Si l'option n'est pas fournie, ce test n'est pas lancé.
`-s` *serveur* `-p` *port*	Cible vers qui envoyer des paquets réseau avec `nc` (ou `netcat`). Par défaut, le port est 9. Si `-s` n'est pas précisé, cet outil est ignoré.
`-m` *repertoire*	Répertoire (point de montage, éventuellement) dans lequel on crée en boucle un fichier de 100 Mo remplis de zéros.
`-l` *repertoire*	Répertoire contenant les test LTP *(Linux Test Packages)* à exécuter.
duree	Durée en secondes d'exécution de `dohell`.

Utilisé par l'option `-b`, le programme `hackbench` que nous avons évoqué dans le chapitre précédent peut être trouvé sur le site d'Ingo Molnár chez Redhat : http://people.redhat.com/mingo/cfs-scheduler/tools/.

Copyright © 2015 Eyrolles.

Quant aux tests de LTP, on peut les obtenir à l'adresse suivante : http://ltp.sourceforge.net/.

Outre les commandes précédentes, le script `dohell` exécute en boucle les actions suivantes en parallèle :

- lire `/proc/interrupts` ;
- exécuter `ps` ;
- copier `/dev/zero` dans `/dev/null` ;
- descendre l'arborescence depuis la racine.

Voici un exemple d'exécution de `dohell` :

```
# ./dohell -b /tmp/hackbench -s 192.168.3.52 -m /mnt/ 1000
```

Sur le répertoire `/mnt`, un *ramdisk* de 128 Mo est monté.

Le script fonctionne pendant un peu plus d'un quart d'heure. En parallèle, nous exécutons naturellement le programme `latency`.

Une autre machine envoie des `pings` à la nôtre en mode *flood* (ininterrompu, avec l'option `-f`), déclenchant ainsi de nombreuses interruptions réseau, et on génère en outre, depuis un autre terminal, une forte activité sur un port USB avec un scanner.

```
# while true; do
scanimage -format=pnm -resolution=1200dpi > fic.png
sleep 1; done
# ./latency

[...]
RTS|     -7.000|     -4.000|     51.000|       0|     0|    00:22:02/00:22:02
#
```

Cette fois, la latence maximale, de 51 microsecondes, est sensiblement plus longue, mais nous sommes probablement proche du pire délai possible. On peut remarquer que pendant l'exécution de `dohell`, les latences moyennes mesurées pendant chaque intervalle sont également plus élevées. Lorsque `dohell` s'interrompt, les valeurs minimales et moyennes diminuent à nouveau.

D'autres outils de tests sont fournis avec Xenomai, notamment une version compilée avec l'API Posix du programme `cyclictest` que nous avons utilisé dans le chapitre précédent. En le laissant tourner pendant une vingtaine de minutes (1 000 secondes) à mesurer les fluctuations d'un timer à la milliseconde, on obtient :

```
# ./cyclictest -i 1000  -l 1000000 -p 99
[...]

T: 0 (29666) P:99 I:    1000 C: 1000000 Min:  -3 Act:   32 Avg:   11 Max:   57
#
```

Copyright © 2015 Eyrolles.

En parallèle de ce programme, `dohell` effectuait son travail de perturbation et de charge du système. La variation maximale sur cette exécution est de 57 microsecondes, ce qui est un peu meilleur que les résultats obtenus avec le patch PREEMPT_RT (63 microsecondes).

Conclusion

L'installation de Xenomai est une opération relativement simple. La phase de compilation du noyau ne diffère pas d'une adaptation classique, et pour la préparation des bibliothèques de l'espace utilisateur, les scripts de configuration déterminent les paramètres à employer en interrogeant au maximum la chaîne de compilation.

Les outils de tests permettent d'obtenir rapidement un aperçu des possibilités et des limites du système. Nous allons examiner dans le prochain chapitre l'API native proposée par Xenomai afin de développer nos propres applications.

Points clés

- Les extensions temps réel pour Linux s'appuient généralement sur une couche logicielle qui capture les interruptions et les renvoie au noyau Linux standard après avoir réalisé des opérations plus prioritaires.
- Le pipeline d'interruptions proposé par Adeos permet de s'affranchir du brevet FSM Labs et de faire fonctionner des systèmes temps réel strict sur Linux. Xenomai est l'un des projets les plus aboutis et les plus dynamiques.
- L'installation de Xenomai passe par l'application d'un patch sur les sources du noyau, sa recompilation et la génération des bibliothèques nécessaires de l'espace utilisateur.
- Les outils de tests de Xenomai permettent de mesurer la latence et la fluctuation des timers. Les résultats sont souvent sensiblement meilleurs qu'avec PREEMPT_RT.

Exercices

Exercice 1 (*)

Téléchargez la dernière version de Xenomai et vérifiez pour quelles versions du noyau Linux sont fournis les patches. Téléchargez une de ces versions et appliquez-lui le patch.

Exercice 2 (**)

Préparez la configuration du noyau modifié. Activez les options liées au temps réel. Compilez Linux et démarrez votre système sur ce kernel. Vérifiez la présence des sous-répertoires `ipipe` et `xenomai` de `/proc`.

Copyright © 2015 Eyrolles.

Exercice 3 (**)

Compilez et installez les bibliothèques de Xenomai et les outils de tests. Utilisez le programme `latency` pour vérifier la bonne compilation et installation de Xenomai.

Exercice 4 (***)

Exécutez le programme `cyclictest`, tout en faisant fonctionner `dohell` en parallèle. Invoquez ce dernier de manière à charger le système au maximum, tant en activité CPU qu'en interruptions. Laissez le système fonctionner quelques heures. Comparez la pire latence obtenue avec les résultats du chapitre précédent.

Copyright © 2015 Eyrolles.

10

Programmer avec Xenomai

Nous avons précédemment installé Xenomai et exécuté quelques programmes de tests qui nous permettent d'avoir un premier aperçu des performances de ce système. Nous allons à présent commencer à créer nos propres applications en utilisant l'API de Xenomai.

Programmation de tâches simples

Principes

Comme nous l'avons évoqué au chapitre précédent, le développement sous Xenomai, s'effectue en utilisant des processus tout à fait classiques de l'espace utilisateur, au sein desquels on démarre des threads temps réel qui sont ordonnancés par Xenomai, sauf lorsqu'ils exécutent des appels système durant lesquels c'est l'ordonnanceur Linux qui reprend le contrôle, ce que l'on nomme « mode secondaire ».

Les deux difficultés que peut rencontrer le développeur Xenomai sont :

- la connaissance de l'API temps réel : si Xenomai propose des skins permettant d'incorporer facilement du code écrit pour d'autres systèmes d'exploitation, il est néanmoins conseillé d'utiliser l'API « native » pour les développements spécifiques, car elle est en principe plus efficace ;
- la recherche des commutations du mode primaire au mode secondaire qui se produisent lors d'appels système, parfois dissimulés dans des fonctions de bibliothèque C, comme `malloc()`, qui invoque de temps à autre les appels `brk()` ou `mmap()`.

Copyright © 2015 Eyrolles.

A priori, le point le plus compliqué est le second, car il entraîne un passage d'un ordonnancement temps réel strict Xenomai (plus prioritaire que les traitements d'interruptions de Linux) à un temps réel souple Linux. Les conséquences ne sont pas directement visibles, mais la prédictibilité des temps de réponse est amoindrie. En outre, la détection de ces basculements est difficile du fait que certaines fonctions de bibliothèques n'invoquent pas immédiatement les appels système sous-jacents mais les diffèrent pour optimiser les temps de traitement en regroupant plusieurs opérations. On peut penser à `malloc()`, mais aussi à `fprintf()`, par exemple. Heureusement, il existe des outils de mise au point intégrés dans Xenomai qui nous permettront de détecter ces basculements en mode secondaire.

La documentation de l'API native de Xenomai est disponible sur le site web de Xenomai (http://www.xenomai.org), mais également dans le répertoire d'installation : `/usr/xenomai/share/doc/xenomai/html/api/`.

Il s'agit d'une documentation générée à l'aide de l'outil *Doxygen*. Je ne présenterai ici que les fonctions essentielles à l'écriture de tâches, en laissant le lecteur se reporter à ces sources pour avoir plus de précisions sur les routines utilisées.

L'écriture d'application avec Xenomai nécessite l'inclusion de certains des fichiers d'en-tête suivants.

Fichiers	Rôles
`<rtdk.h>`	Accès aux fonctions de la famille `rt_printf()`, décrites plus loin.
`<native/alarm.h>`	Programmation d'alarme pour activer un thread temps réel de manière différée et/ou périodique.
`<native/buffer.h>`	Communication Fifo par blocs mémoire entre threads ou processus.
`<native/cond.h>`	Synchronisation sur des variables conditions proches des `pthread_cond_t`.
`<native/event.h>`	Notification d'événements en utilisant des *flags* programmables.
`<native/heap.h>`	Allocation de mémoire par tas, éventuellement partagés entre processus.
`<native/intr.h>`	Gestion des interruptions depuis l'espace utilisateur.
`<native/misc.h>`	Autorisation d'accès aux ports d'entrées-sorties.
`<native/mutex.h>`	Synchronisation sur des mutex proches des `pthread_mutex_t`.
`<native/pipe.h>`	Communication par tubes nommés avec les processus de l'espace Linux.
`<native/queue.h>`	Transmission de messages sans copie (buffers partagés).
`<native/sem.h>`	Synchronisation par sémaphores.
`<native/task.h>`	Fonctions et types permettant de gérer les tâches : création, configuration, destruction, mise en sommeil, envoi de message, etc.
`<native/timer.h>`	Configuration, lecture et conversion de l'heure système.

Copyright © 2015 Eyrolles.

Initialisation du processus

Au chapitre 6, nous avons évoqué l'importance de l'utilisation de `mlockall()` afin de verrouiller en mémoire physique les pages utilisées par le processus, évitant ainsi tous les comportements imprévisibles liés à la gestion de la mémoire virtuelle.

Avec Xenomai, cet appel est indispensable, car le déclenchement d'une faute de page (lors d'un accès à une zone qui n'est pas encore disponible) entraînerait un retour en mode secondaire. Ceci se produirait de manière totalement imprévisible puisque par défaut, c'est à l'instant de la première utilisation d'une page mémoire qu'elle est attribuée par le noyau, et non lors de son allocation initiale.

Tout programme Xenomai devra donc commencer – avant de lancer ses tâches temps réel – par une invocation :

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

Si le programme temps réel doit afficher des messages sur sa sortie standard, sa sortie d'erreur ou dans un fichier de traces, il peut utiliser l'une des fonctions de la bibliothèque RTDK (intégrée dans Xenomai) :

```
int rt_printf   (const char * format...);
int rt_vprintf  (const char * format,
                 va_list arguments);

int rt_fprintf  (FILE * flux, const char * format...);
int rt_vfprintf (FILE * flux, const char * format,
                 va_list arguments);

int rt_puts (const char * chaine);

void rt_syslog  (int priorite, const char *format...);
void rt_vsyslog (int priorite, const char *format,
                 va_list args);
```

À l'usage, ces fonctions sont très proches de leurs équivalents de la libC standard. Leur fonctionnement interne est toutefois assez différent, car pour garantir un déroulement déterministe, il est hors de question de s'appuyer directement sur les appels système de Linux comme `write()`. La bibliothèque RTDK de Jan Kiszka propose donc de mémoriser les messages dans des buffers circulaires spécifiques à chaque thread. Un thread non temps réel est chargé d'effectuer l'affichage des données périodiquement (100 ms).

Ceci nécessite de démarrer le thread d'affichage et d'initialiser les buffers de stockage, ce qui s'effectue à l'aide de la fonction :

```
int rt_print_init (size_t taille, const char * nom) ;
```

La taille et le nom sont ceux du buffer à allouer.

On pourrait appeler directement `rt_print_init()`, mais il est beaucoup plus courant d'invoquer :

```
void rt_print_auto_init(int valide);
```

Copyright © 2015 Eyrolles.

qui s'assurera que, dès la première utilisation d'une des fonctions de la famille précédente, `rt_print_init()` sera correctement appelée. On ajoute donc, juste après le `mlockall()` précédemment évoqué :

```
rt_print_auto_init(1);
```

Création de tâche

La création d'une tâche peut être réalisée avec `rt_task_create()`, qui prépare toutes les structures de données nécessaires au fonctionnement d'un nouveau thread, suivi de `rt_task_start()`, qui démarre véritablement le thread suspendu.

On peut également regrouper les deux opérations en une seule avec `rt_task_spawn()`, qui crée une tâche et la laisse démarrer immédiatement.

```
int rt_task_create (RT_TASK * tache,
                    const char * nom,
                    int taille_pile,
                    int priorite,
                    int mode);

int rt_task_start (RT_TASK * tache,
                   void (* fonction) (void *arg),
                   void * arg);

int rt_task_spawn (RT_TASK * tache,
                   const char * nom,
                   int taille_pile,
                   int priorite,
                   int mode,
                   void (* fonction) (void * arg),
                   void * arg);
```

Arrêtons-nous un instant sur les paramètres de ces fonctions.

Paramètres	Signification
`tache`	Pointeur sur une structure qui sera initialisée durant l'appel à `rt_task_create()` pour représenter la tâche temps réel.
`nom`	Nom de la tâche tel qu'il apparaît dans `/proc/xenomai/sched`.
`taille_pile`	Taille de la pile. Si la valeur est nulle, une taille par défaut est affectée permettant de stocker 1 024 entiers.
`priorite`	La priorité va de 1 à 99 dans l'espace des tâches Xenomai de manière similaire aux threads temps réel souple de Linux.
`mode`	Configuration du thread temps réel. Voir page ci-contre.
`fonction`	La fonction que le thread doit exécuter.
`arg`	Argument que la fonction du thread temps réel reçoit en paramètre à son démarrage.

Copyright © 2015 Eyrolles.

Le paramètre `mode` est une combinaison par OU binaire entre les éléments suivants.

Constantes	Signification
`T_CPU(`*numero*`)`	La tâche s'exécutera sur le CPU indiqué. Le *numero* doit être strictement inférieur à `RTHAL_NR_CPUS`.
`T_FPU`	La tâche temps réel utilisera les ressources de la FPU *(Floating Point Unit)*. Il s'agit d'un attribut fixé automatiquement pour les tâches créées dans l'espace utilisateur. Pour celles créées dans le noyau, ceci permet d'indiquer à Xenomai s'il faut sauvegarder ou non les registres de la FPU lors des commutations vers cette tâche.
`T_JOINABLE`	Cet attribut indique qu'il est possible d'attendre la fin de l'exécution de la tâche. Nous allons l'utiliser ci-après.
`T_SUSP`	Le thread est créé dans un état suspendu. On l'activera ensuite avec `rt_task_resume()`.
`T_WARNSW`	Si le thread passe du mode primaire au mode secondaire, il recevra un signal `SIGXCPU`. Par défaut, ceci tue le processus, mais on peut le capturer pour analyser le problème. Nous le détaillerons ci-après.

Dans notre premier exemple, nous allons également utiliser les appels système :

```
int rt_task_join (RT_TASK * tache);
```

qui permet d'attendre la fin d'un thread temps réel, et :

```
int rt_task_sleep (RTIME duree);
```

pour endormir la tâche courante pendant une durée comptée en nanosecondes (nous reviendrons sur cet argument ultérieurement).

```
exemple-hello-01.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <rtdk.h>
#include <native/task.h>


void fonction_hello_world (void * unused)
{
  while (1) {
    rt_printf("Hello from Xenomai Realtime Space\n");
    rt_task_sleep(1000000000LL); // 1 milliard ns = 1 s.
  }
}
```

Copyright © 2015 Eyrolles.

```
int main(void)
{
  int err;
  RT_TASK task;

  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_print_auto_init(1);

  if ((err = rt_task_spawn(& task, "Hello 01",
                           0, 99, T_JOINABLE,
                       fonction_hello_world, NULL))!= 0){
    fprintf(stderr, "rt_task_spawn: %s\n",strerror(-err));
    exit(EXIT_FAILURE);
  }
  rt_task_join(& task);
  return 0;
}
```

Les fonctions de l'API Xenomai renvoient généralement zéro si tout s'est bien passé et une valeur d'erreur en cas d'échec. Elles ne remplissent pas la variable globale `errno` comme peuvent le faire d'autres fonctions de la bibliothèque C, aussi ne doit-on pas appeler directement `perror()`, mais enregistrer le code d'erreur négatif et le transmettre à `strerror()` avant de l'afficher.

Compilation et exécution

La compilation d'un programme avec l'API native de Xenomai va faire appel à un `Makefile` dans lequel on invoquera le script `xeno-config` (installé dans `/usr/xenomai/bin/`) pour obtenir les paramètres de compilation à transmettre à `gcc`. En voici un exemple minimal.

Makefile :

```
XENOCONFIG=/usr/xenomai/bin/xeno-config
CC=     $(shell $(XENOCONFIG) --cc)
CFLAGS= $(shell $(XENOCONFIG) --skin=native --cflags)
LDFLAGS=$(shell $(XENOCONFIG) --skin=native --ldflags)
LDFLAGS+=-lnative
LDLIBS=-lnative -lxenomai

all:: exemple-hello-01
clean::
    rm -f exemple-hello-01 *.o
```

Compilons et exécutons le programme.

S'il s'agit d'une cross-compilation (pour un processeur différent), le script `xeno-config` indiquera le nom du compilateur utilisé lors de la génération des bibliothèques. Toutefois, il faudra s'assurer que la variable d'environnement `PATH` contient bien le répertoire où se trouve ce dernier.

Copyright © 2015 Eyrolles.

```
$ make
gcc -I/usr/xenomai/include -D_GNU_SOURCE -D_REENTRANT -Wall
➥ -Werror-implicit-function-declaration -pipe -D__XENO__  -g  -lnative
➥ -L/usr/xenomai/lib -lxenomai -lpthread -lrt  -lnative   exemple-hello-01.c
➥ -lnative -lxenomai -o exemple-hello-01
$
```

On démarre le processus normalement. Pour qu'il puisse trouver les bibliothèques dont il a besoin, il faut configurer la variable `LD_LIBRARY_PATH` avant de lancer l'application.

```
# export LD_LIBRARY_PATH=/usr/xenomai/lib/
# ./exemple-hello-01
Hello from Xenomai Realtime Space
Hello from Xenomai Realtime Space
Hello from Xenomai Realtime Space
 (Contrôle-C)
#
```

Pendant le déroulement du thread, il est possible de l'observer ainsi :

```
# cat /proc/xenomai/sched
CPU PID  CLASS PRI  TIMEOUT   TIMEBASE  STAT NAME
  0  0    idle -1      -         master   R   ROOT/0
  1  0    idle -1      -         master   R   ROOT/1
  2  0    idle -1      -         master   R   ROOT/2
  3  0    idle -1      -         master   R   ROOT/3
  0  13159  rt  99  686ms314us  master   D   Hello_01
  0 0    idle   -1     -       master   R    ROOT/0
  1 0    idle   -1     -       master   R    ROOT/1
# cat /proc/xenomai/stat
CPU  PID  MSW  CSW        PF  STAT      %CPU  NAME
  0  0     0   29036091   0   00500080   99.9 ROOT/0
  1  0     0          8   0   00500080  100.0 ROOT/1
  2  0     0          7   0   00500080  100.0 ROOT/2
  3  0     0          0   0   00500080  100.0 ROOT/3
  0  13159 0        183   0   00300184    0.0 Hello_01
  0  0     0   41560275   0   00000000    0.0 IRQ2316: [timer]
#
```

Nous observons la présence du noyau Linux (`ROOT`) sur quatre CPU et du handler de l'interruption timer pour Xenomai. Notre tâche a été activée 183 fois (colonne *Context Switches*, `CSW`) mais n'a pas subi de commutation du mode primaire vers le secondaire (colonne *Mode Switches*, `MSW`). Ces informations peuvent être vues de manière plus synthétique avec la commande `rtps` se trouvant dans `/usr/xenomai/sbin/` :

Copyright © 2015 Eyrolles.

```
# /usr/xenomai/sbin/rtps
PID   TIME              THREAD          CMD

0     023:32:54.105,594 ROOT/0            -
0     023:34:05.627,537 ROOT/1            -
0     023:34:05.316,192 ROOT/2            -
0     023:34:06.580,449 ROOT/3            -
13159 000:00:00.003,791 Hello_01          ./exemple-hello-01
0     000:00:22.984,643 IRQ2316:[timer]   -
#
```

Processus unithread

Il est possible de basculer en temps réel un thread déjà existant dans le domaine Linux sans en créer de nouveau. Pour cela, on invoque :

```
int rt_task_shadow (RT_TASK *    tache,
                    const char * nom,
                    int          priorite,
                    int          mode);
```

En voici un exemple très simple :

```
exemple-hello-02.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <rtdk.h>
#include <native/task.h>


int main(void)
{
  int err;
  RT_TASK task;

  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_print_auto_init(1);

  if ((err = rt_task_shadow(& task, "Hello World 02",
                            99, T_JOINABLE)) != 0) {
    fprintf(stderr, "rt_task_shadow:%s\n",strerror(-err));
    exit(EXIT_FAILURE);
  }
  while (1) {
    rt_printf("Hello World 02\n");
```

Copyright © 2015 Eyrolles.

```
    rt_task_sleep(1000000000LL); // 1 sec.
  }
  return 0;
}
```

L'exécution donne un résultat comparable.

```
# ./exemple-hello-02
Hello World 02
Hello World 02
Hello World 02
Hello World 02
Hello World 02
 (Contrôle-C)
#
```

Recherche des changements de modes

Si une tâche a l'attribut `T_WARNSW` *(Warn Switch)*, elle recevra un signal `SIGXCPU` dès qu'elle passera du mode primaire au mode secondaire. Par défaut, ce signal tue le processus, et le shell affiche le message « CPU time limit exceeded » mais nous pouvons également installer un gestionnaire de signal pour diagnostiquer le problème.

Pour fixer l'attribut `T_WARNSW`, on peut l'indiquer dans le paramètre `mode` à la création du thread, ou employer la fonction `rt_task_set_mode()` dont les possibilités complètes dépassent un peu le cadre de ce livre.

Nous allons forcer un passage en mode secondaire en appelant la fonction `fprintf()` pour écrire sur la sortie standard. Toutefois, afin de voir que la détection n'est pas toujours évidente, nous écrirons un message (de 256 octets) sans retour chariot final.

> Cette expérience ne présente pas toujours les mêmes résultats, en fonction de la bibliothèque C et de la plate-forme de tests utilisées.

La fonction `fprintf()` de la bibliothèque C stocke les messages dans un buffer de sortie. Ce n'est que lors de la réception d'un retour chariot, lorsque le buffer est plein ou encore lors d'un appel `fflush()`, que l'appel système `write()` de Linux est invoqué.

Le buffer par défaut faisant 4 096 octets, ce n'est qu'à la seizième écriture que la commutation se fera. Vérifions le comportement :

```
exemple-hello-03.c  :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
```

Copyright © 2015 Eyrolles.

```
#include <rtdk.h>
#include <native/task.h>

void fonction_hello_world (void * unused)
{
  int i = 1;
  while (1) {
    rt_fprintf(stderr, "Ecriture de 256 octets : %d\n",
                        i++);
    fprintf(stdout, "0123456789ABCDEF123456789ABCDEF"
                    "0123456789ABCDEF123456789ABCDEF"
                    "0123456789ABCDEF123456789ABCDEF"
                    "0123456789ABCDEF123456789ABCDEF"
                    "0123456789ABCDEF123456789ABCDEF"
                    "0123456789ABCDEF123456789ABCDEF"
                    "0123456789ABCDEF123456789ABCDEF"
                    "0123456789ABCDEF123456789ABCDEF");
    rt_task_sleep(1000000000LL);
  }
}

int main(void)
{
  int err;
  RT_TASK task;
  [...]

  if ((err = rt_task_spawn(& task, "Hello 01",
                        0, 99, T_JOINABLE | T_WARNSW,
                     fonction_hello_world, NULL)) != 0) {
    fprintf(stderr, "rt_task_spawn: %s\n",strerror(-err));
    exit(EXIT_FAILURE);
  }
  rt_task_join(& task);
  return 0;
}
```

À l'exécution, nous voyons les messages écrits avec `rt_fprintf()` sur la sortie d'erreur, mais pas ceux sur la sortie standard, puisqu'ils sont conservés dans le buffer.

```
# ./exemple-hello-03
Ecriture de 256 octets : 1
Ecriture de 256 octets : 2
Ecriture de 256 octets : 3
[...]
Ecriture de 256 octets : 14
Ecriture de 256 octets : 15
Ecriture de 256 octets : 16
CPU time limit exceeded
#
```

Copyright © 2015 Eyrolles.

Il est bien sûr possible d'intercepter le signal `SIGXCPU` avec la fonction `sigaction()` classique, et d'installer un gestionnaire qui affichera des informations avant de terminer le processus. On pourrait ainsi identifier les points de commutations vers le mode secondaire pour essayer de les éliminer.

Alarmes et tâches périodiques

Il est souvent nécessaire dans les applications temps réel de différer un traitement, afin qu'il soit réalisé plus tard, ou de programmer une action à effectuer périodiquement. Pour obtenir ce comportement, Xenomai nous propose deux méthodes complémentaires : les réveils périodiques et les alarmes. Chaque tâche peut être régulièrement activée par un réveil personnel. Elle peut aussi recevoir les signaux d'une ou plusieurs alarmes (qui peuvent elles-mêmes être partagées entre plusieurs tâches).

Réveils périodiques

Le principe est simple : nous pouvons rendre un thread périodique, après sa création, grâce à la fonction :

```
int rt_task_set_periodic (RT_TASK * tache,
                          RTIME     debut,
                          RTIME     periode);
```

Le premier paramètre est l'identifiant de la tâche. Dans le cas où une tâche veut se rendre périodique elle-même, elle peut employer `NULL` ou `rt_task_self()` qui lui renvoie toujours son propre identifiant.

```
RT_TASK * rt_task_self (void);
```

Les deux autres paramètres s'expriment différemment suivant la valeur fournie pour l'option Base Period du sous-menu Real-Time Sub-System>Interfaces>Native API lors de la compilation du noyau. Ceci est vrai par ailleurs pour toutes les valeurs temporelles de l'API native de Xenomai :

- si la valeur est zéro, les dates de déclenchement et les durées sont mesurées en nanosecondes ;
- si la valeur n'est pas nulle, elle correspond à la durée (en microsecondes) d'un tick, et toutes les autres valeurs de temps sont exprimées en multiples de ce tick.

Dans la plupart des cas, on préfère laisser la période de base de Xenomai à zéro, et exprimer les valeurs temporelles en nanosecondes. C'est ce que nous utiliserons dans la suite de ce chapitre.

> Il est possible de modifier dynamiquement la configuration pour la base de temps de l'API native à l'aide de la fonction `rt_timer_set_mode()`, mais en pratique, ceci est rarement utilisé.

Copyright © 2015 Eyrolles.

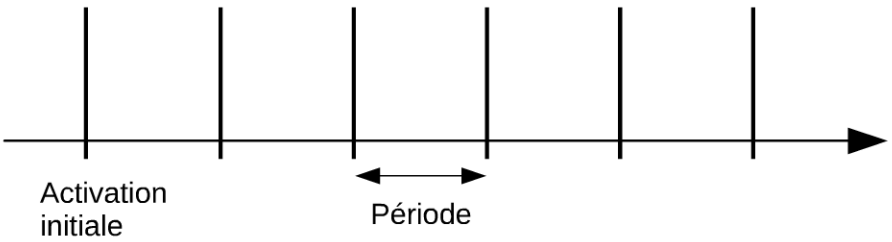
Le paramètre `debut` indique l'heure de la première activation de la tâche. Si la valeur est `TM_NOW`, l'activation est immédiate. On peut également utiliser la valeur `rt_timer_read()` qui nous renvoie la valeur du timer système, à laquelle on ajoute la durée désirée avant le premier déclenchement.

```
RTIME rt_timer_read (void);
```

Le troisième paramètre est la période de réveil du thread. La valeur `TM_INFINITE` permet d'arrêter l'activation régulière de la tâche.

Figure 10-1

Activations d'une tâche périodique



Pour être réveillée, celle-ci doit préalablement être en attente sur l'appel `rt_task_wait_period()` :

```
int rt_task_wait_period (unsigned long * depassements);
```

Cette fonction bloque la tâche appelante (sauf si la période n'a pas encore été programmée) jusqu'à la prochaine activation. Elle renvoie normalement zéro. En cas de problème (période non programmée, signal reçu, etc.), `rt_task_wait_period()` renvoie un code d'erreur négatif.

Le comportement global de la tâche périodique ressemblera donc à :

```
while (rt_task_wait_period (NULL) == 0) {
    // traitement
}
```

Si toutefois le traitement dure plus longtemps qu'une période, ou si une autre tâche plus prioritaire préempte le thread appelant, un dépassement pourra se produire. Si le paramètre de `rt_task_wait_period()` n'est pas `NULL`, le pointeur sera renseigné avec le nombre d'activations manquées. Nous examinerons ce cas plus loin.

Voici un exemple simple d'une tâche affichant l'heure toutes les secondes :

```
exemple-periodique-01.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/timer.h>
#include <rtdk.h>

void fonction_periodique (void * arg)
{
    int err;
    unsigned long depassements;
```

Copyright © 2015 Eyrolles.

```
    rt_task_set_periodic(rt_task_self(), TM_NOW,
                         1000000000); // 1. sec.
    rt_printf("[%lld] Timer programmé...\n",
                rt_timer_read());
    while ((err=rt_task_wait_period(&depassements)) == 0){
        rt_printf("[%lld]", rt_timer_read());
        if (depassements != 0)
            rt_printf(" Depassements: %lu", depassements);
        rt_printf("\n");
    }
    fprintf(stderr, "rt_task_wait_period(): %s\n",
                strerror(-err));
    exit(EXIT_FAILURE);
}


int main()
{
    RT_TASK task;

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_print_auto_init(1);

    if (rt_task_spawn(& task, NULL, 0, 99,
            T_JOINABLE, fonction_periodique, NULL) != 0) {
        fprintf(stderr, "Impossible de creer la tache\n");
        exit(EXIT_FAILURE);
    }

    rt_task_join(& task);
    return 0;
}
```

L'exécution affiche les heures en nanosecondes :

```
# ./exemple-periodique-01
[3358442575000] Timer programmé...
[3359442585000]
[3360442583000]
[3361442581000]
[3362442583000]
[3363442584000]
[3364442584000]
[3365442581000]
[3366442583000]
[3367442583000]
[3368442583000]
  (Contrôle-C)
#
```

Copyright © 2015 Eyrolles.

Le chiffre en gras évoluant sur chaque ligne représente bien les milliards de nanosecondes. Nous voyons que les microsecondes fluctuent à chaque période (ce qui est cohérent avec les résultats déjà obtenus), mais que les dizaines de microsecondes restent presque stables (système avec une charge moyenne).

Voyons à présent ce qui se passe si notre application – mal dimensionnée évidemment – doit réaliser ponctuellement un traitement plus long que la période d'activation, ou si une autre tâche la préempte pendant plus d'une période.

Pour éviter de geler totalement notre système, nous allons réaliser une boucle active plus longue que la période lors d'une activation sur quatre. La période est également réduite pour avoir une activation toutes les demi-secondes (le comportement est plus visible ainsi).

La fonction périodique est modifiée comme suit :

```
exemple-periodique-02.c :
[...]

void fonction_periodique (void * arg)
{
    int err;
    unsigned long depassements;
    int i;

    rt_task_set_periodic(rt_task_self(),
                         TM_NOW, 500000000);
    rt_printf("[%lld] Timer programmé...\n",
                       rt_timer_read());
    i = 0;
    while (1) {
        rt_task_wait_period(& depassements);
        rt_printf("[%lld]", rt_timer_read());
        if (depassements != 0)
            rt_printf(" Depassements: %lu", depassements);
        rt_printf("\n");
        i ++;
        i = i % 4;
        if (i == 0)
            rt_timer_spin(800000000);
    }
    fprintf(stderr, "rt_task_wait_period(): %s\n",
                strerror(-err));
    exit(EXIT_FAILURE);
}

[...]
```

Copyright © 2015 Eyrolles.

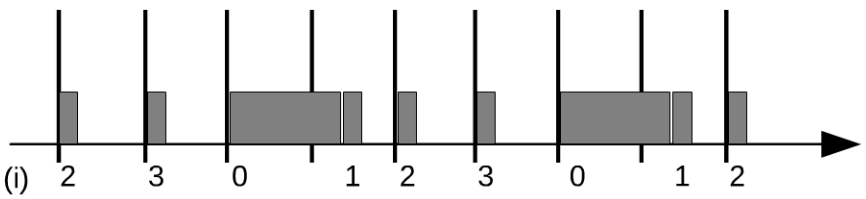
Les premiers résultats peuvent sembler un peu surprenants :

```
# ./exemple-periodique-02
[3432254806000] Timer programmé...
[3432754815000]
[3433254813000]
[3433754811000]
[3434254814000]
[3435054882000]
[3435254815000]
[3435754810000]
[3436254817000]
[3437054863000]
[3437254817000]
[3437754810000]
[3438254817000]
[3439054864000]
[3439254814000]
[3439754811000]
   (Contrôle-C)
#
```

Les chiffres en gras représentent les dixièmes de secondes. Nous remarquons qu'une période sur quatre dure 0,8 seconde tandis que les trois autres durent 0,5 seconde, ce que nous attendions. Nous pouvons toutefois observer que `rt_task_wait_period()` n'indique pas de débordement complet de période. La situation est résumée sur la figure 10-2 sur laquelle les zones grisées représentent les cycles de travail actif du thread.

Figure 10-2

Dépassement de la période d'activation



Les numéros situés sous l'axe des abscisses indiquent les valeurs de la variable `i`. L'activation de la séquence `i=1` commence immédiatement après celle `i=0`, sans attendre la période suivante. Toutefois, cette activation se produit bien dans son créneau et aucun décalage n'est indiqué. Si nous modifions comme suit notre programme pour allonger la période de travail (de boucle active dans notre cas) à 1,2 seconde :

```
exemple-periodique-03.c :
[...]
        i = i % 4;
        if (i == 0)
            rt_timer_spin(1200000000);
    }
[...]
```

Copyright © 2015 Eyrolles.

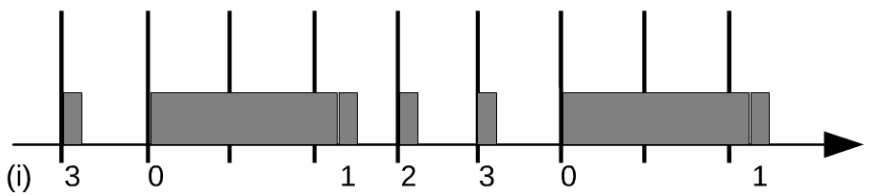
nous obtenons alors :

```
# ./exemple-periodique-03
[3583969345000] Timer programme...
[3584469354000]
[3584969345000]
[3585469350000]
[3585969345000]
[3587169406000] Depassements : 1
[3587469354000]
[3587969344000]
[3588469352000]
[3589669399000] Depassements : 1
[3589969353000]
[3590469349000]
[3590969347000]
[3592169390000] Depassements : 1
[3592469353000]
   (Contrôle-C)
#
```

À présent, `rt_task_wait_period()` nous indique bien un nombre de périodes complètes dépassées par le travail actif, ce qui est résumé sur la figure 10-3. En effet, l'activation `i=1` est décalée d'une période complète par rapport à son instant théorique, ce qui est indiqué par l'argument de `rt_task_wait_period()`.

Figure 10-3

Dépassement de plus d'une période



Nous pouvons effectuer une mesure de précision des tâches périodiques avec le programme suivant, auquel on fournit une période en microsecondes et qui nous affiche sur sa sortie standard les durées effectives entre activations.

```
exemple-periodique-04.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/timer.h>
#include <rtdk.h>

void fonction_periodique (void * arg)
```

Copyright © 2015 Eyrolles.

```
{
  RTIME precedent = 0;
  RTIME heure = 0;
  RTIME periode;
  RTIME duree;
  periode = * (RTIME *) arg;
  periode = periode * 1000; // en ns
  rt_task_set_periodic(NULL, TM_NOW, periode);

  while(1) {
    precedent = heure;
    rt_task_wait_period(NULL);
    heure = rt_timer_read();
    if (precedent == 0)
      // Ignorer le premier déclenchement
      continue;
    duree = heure - precedent;
    rt_printf("%llu\n", duree/1000);
  }
}

int main(int argc, char * argv[])
{
  RT_TASK task;
  RTIME periode;
  int err;

  if ((argc != 2)
    || (sscanf(argv[1], "%llu", & periode) != 1)) {
    fprintf(stderr, "usage: %s periode_en_us\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_print_auto_init(1);

  if ((err = rt_task_spawn(& task, NULL, 0, 99,
      T_JOINABLE, fonction_periodique, & periode)) != 0) {
    fprintf(stderr, "rt_task_spaw: %s\n", strerror(-err));
    exit(EXIT_FAILURE);
  }

  rt_task_join(& task);
  return 0;
}
```

Copyright © 2015 Eyrolles.

Effectuons un premier test avec une période d'une milliseconde, suivie d'une mesure longue durant une heure.

```
# ./exemple-periodique-04 1000
1000
1000
999
1001
999
999
1000
999
1000
1000
999
1001
998
999
1000
999
999
1000
    (Contrôle-C)
# ./exemple-periodique-04 1000 > data-1.txt
    (Contrôle-C au bout d'une heure environ)
#
```

Le premier jeu de résultats peut être analysé avec les outils développés au chapitre 4 :

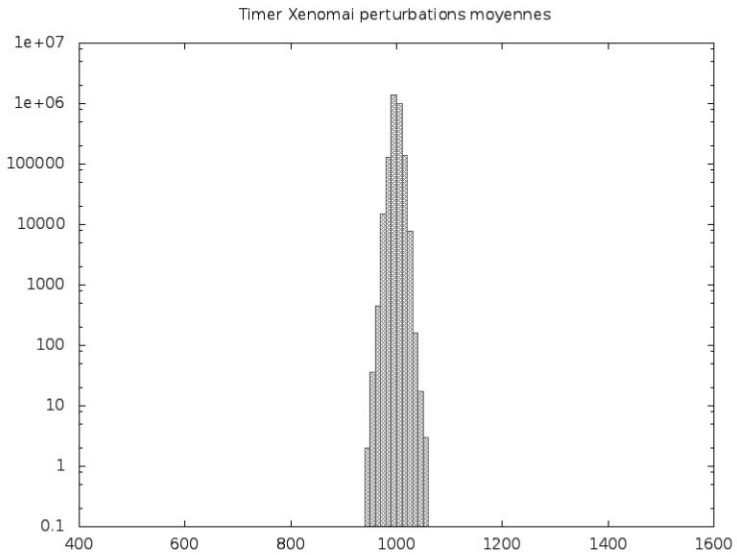
```
# ../chapitre-04/calculer-statistiques < data-1.txt
 Nb mesures = 2716761
 Minimum = 942
 Maximum = 1052
 Moyenne = 999
 Ecart-type = 4
# ../chapitre-04/calculer-histogramme 100 500 1500 < data-1.txt > histo-1.txt
# ../chapitre-04/afficher-histogramme.sh histo-1.txt "Timer Xenomai non perturbé"
```

Le résultat est visible sur la figure 10-4 : les valeurs sont équivalentes à celles que nous avions obtenues avec le temps réel de Linux (chapitre 6), mais ici, nous avons laissé le système fonctionner pendant beaucoup plus longtemps, ce qui le rend sensible à des interruptions rares.

Copyright © 2015 Eyrolles.

Figure 10-4

Timer Xenomai avec perturbations moyennes



Faisons à présent fonctionner le même programme avec de fortes perturbations engendrées par le script `dohell` dont nous avons parlé dans le précédent chapitre. Ce dernier est invoqué régulièrement afin de déclencher des pointes d'activité, tant au niveau de la charge du système que des interruptions reçues. Voici les résultats obtenus :

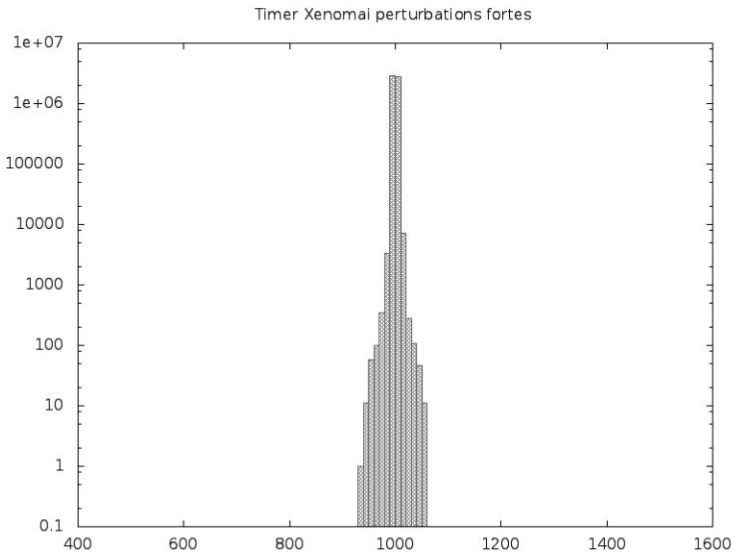
```
# ../chapitre-04/calculer-statistiques < data-2.txt
 Nb mesures = 5696977
 Minimum = 937
 Maximum = 1058
 Moyenne = 999
 Ecart-type = 2
#
```

La figure 10-5 montre peu de changements par rapport à la précédente : les valeurs étant beaucoup plus nombreuses, la courbe semble plus effilée, mais il y a en réalité peu de différences. Nous remarquons néanmoins une très bonne tenue de la valeur maximale puisqu'elle est comparable à celle obtenue avec le noyau PREEMPT_RT, alors que les expériences du chapitre 8 se déroulaient sur des durées beaucoup plus courtes.

Copyright © 2015 Eyrolles.

Figure 10-5

Timer Xenomai avec fortes perturbations



Alarmes

Les alarmes de Xenomai ont un rôle un peu différent des timers précédents, car elles sont beaucoup plus proches des systèmes de *watchdogs* (chiens de garde) ou de *timeout* (dépassement de délai).

> Un watchdog est un mécanisme courant en informatique industrielle : il s'agit d'un composant logiciel ou matériel qui est prêt à réinitialiser le système s'il détecte que l'application principale ne répond plus.

Une alarme est programmée pour se déclencher à partir d'un instant donné et se répéter éventuellement avec une période fixée. Une tâche réveillée par une alarme s'exécute avec une priorité supérieure à toutes les tâches normales de Xenomai. Elle pourra donc prendre toutes les mesures d'urgence impliquées par le déclenchement de l'alarme.

On crée l'alarme et on la détruit avec :

```
int  rt_alarm_create (RT_ALARM * alarme, const char *nom);
void rt_alarm_delete (RT_ALARM * alarme);
```

puis on la programme et l'arrête avec les fonctions :

```
int rt_alarm_start (RT_ALARM * alarm,
                    RTIME debut, RTIME periode);
int rt_alarm_stop  (RT_ALARM * alarm);
```

Copyright © 2015 Eyrolles.

Enfin, une tâche peut se mettre en attente sur l'alarme avec la fonction `rt_alarm_wait()`, qui nous rappelle le `rt_task_wait_period()` étudié précédemment.

```
int rt_alarm_wait (RT_ALARM * alarme);
```

Dans l'exemple suivant, nous allons vérifier que plusieurs tâches peuvent être simultanément en attente sur la même alarme. Nous allons également examiner leurs priorités d'exécution avant et après déclenchement de l'alarme, grâce à la fonction :

```
int rt_task_inquire (RT_TASK *tache, RT_TASK_INFO *info);
```

Celle-ci remplit une structure contenant les champs suivants, susceptibles de nous intéresser pour suivre l'activité d'une tâche.

Nom	Type	Signification
`name`	`char [ ]`	Nom de la tâche.
`status`	`unsigned`	État de la tâche. Par exemple, `T_BLOCKED` (endormie), `T_READY` (prête), `T_DELAYED` (différée), etc.
`bprio`	`int`	Priorité initiale.
`cprio`	`int`	Priorité courante (peut être différente de la priorité initiale dans les situations d'héritage de priorité).
`ctxswitches`	`int`	Nombre de changements de contexte.
`modeswitches`	`int`	Nombre de commutations vers le mode secondaire.
`exectime`	`RTIME`	Durée d'exécution en nanosecondes.
`relpoint`	`RTIME`	Heure de la prochaine activation de la tâche.

Nous pourrons ainsi visualiser la priorité du champ `cprio`.

exemple-alarme-01.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/alarm.h>
#include <native/task.h>
#include <native/timer.h>
#include <rtdk.h>

static RT_ALARM alarme;

void fonction_thread (void * arg)
{
    int err;
    int numero = (int) arg;
    RT_TASK_INFO rtinfo;
```

Copyright © 2015 Eyrolles.

```
    rt_task_inquire(NULL, & rtinfo);
    rt_printf("[%d] Priorite initiale %d\n"
              numero, rtinfo.cprio);
   while(1) {

        if ((err = rt_alarm_wait(& alarme)) != 0) {
            fprintf(stderr, "rt_alarm_wait(): %s\n",
                            strerror(-err));
            exit(EXIT_FAILURE);
        }
        rt_task_inquire(NULL, & rtinfo);
        rt_printf("[%d] priorite : %d, heure : %llu\n",
                  numero, rtinfo.cprio, rt_timer_read());
    }
}

#define NB_TACHES 5

int main(void)
{
    int err;
    int i;
    RT_TASK task[NB_TACHES];
    char nom_tache[80];

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_print_auto_init(1);

    if ((err = rt_alarm_create(& alarme, "Ex01")) != 0) {
        fprintf(stderr, "rt_alarm_create(): %s\n",
                        strerror(-err));
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < NB_TACHES-1; i++) {
        snprintf(nom_tache, 80, "Alarme-%d\n", i+1);
        if ((err = rt_task_spawn(&(task[i]), nom_tache, 0,
                         90-NB_TACHES + i, T_JOINABLE,
                    fonction_thread, (void *)(i+1)))!=0) {
            fprintf(stderr, "rt_task_spawn: %s\n",
                             strerror(-err));
            exit(EXIT_FAILURE);
        }
    }

    if ((err = rt_alarm_start(& alarme,
                TM_NOW, 1000000000)) != 0) {
        fprintf(stderr, "rt_alarm_start: %s\n",
                         strerror(-err));
        exit(EXIT_FAILURE);
    }
```

Copyright © 2015 Eyrolles.

```
    for (i = 0; i < NB_TACHES; i ++)
        rt_task_join(& task[i]);
    return 0;
}
```

Les cinq tâches sont créées avec des priorités croissantes pourtant, lorsque l'alarme se déclenche, elles sont réveillées dans l'ordre de mise en attente et s'exécutent alors avec une priorité beaucoup plus élevée (257) :

```
# ./exemple-alarme-01
[1] Priorite initiale 85
[2] Priorite initiale 86
[3] Priorite initiale 87
[4] Priorite initiale 88
[1] priorite : 257, heure : 3680733471000
[2] priorite : 257, heure : 3680733518000
[3] priorite : 257, heure : 3680733549000
[4] priorite : 257, heure : 3680733574000
[1] priorite : 257, heure : 3681733456000
[2] priorite : 257, heure : 3681733519000
[3] priorite : 257, heure : 3681733553000
[4] priorite : 257, heure : 3681733583000
[1] priorite : 257, heure : 3682733455000
[2] priorite : 257, heure : 3682733517000
[3] priorite : 257, heure : 3682733548000
[4] priorite : 257, heure : 3682733578000
[1] priorite : 257, heure : 3683733455000
[2] priorite : 257, heure : 3683733514000
[3] priorite : 257, heure : 3683733547000
[4] priorite : 257, heure : 3683733576000
[1] priorite : 257, heure : 3684733455000
[2] priorite : 257, heure : 3684733516000
[3] priorite : 257, heure : 3684733549000
[4] priorite : 257, heure : 3684733579000
   (Contrôle-C)
#
```

Ce mécanisme est donc conçu pour créer des tâches de surveillance, qui pourront se déclencher périodiquement (ou au bout d'une durée préprogrammée si l'alarme n'est pas inhibée entre temps) et agir plus prioritairement que les autres tâches de l'application temps réel.

Watchdog

On notera qu'il existe dans Xenomai un chien de garde intégré qui se déclenche si une tâche consomme du temps CPU de manière ininterrompue pendant une durée supérieure à un seuil configurable au moment de la compilation du noyau. Ce watchdog interrompra la tâche avec le signal `SIGXCPU`.

Copyright © 2015 Eyrolles.

Dans le menu Real-Time Sub-System de la configuration du kernel, l'activation de l'option Debug Support fait apparaître plusieurs nouvelles options dont :

- Watchdog Support : activer la présence d'un chien de garde logiciel intégré dans Xenomai ;
- Watchdog Timeout : durée (en secondes) maximale d'exécution ininterrompue d'une tâche avant déclenchement du signal `SIGXPCU`.

Pour vérifier si le watchdog est programmé dans le noyau, on peut rechercher les lignes suivantes dans le fichier `.config` ou dans `/proc/config.gz`, si les options Kernel .config Support et Enable Access to .config Through /proc/config.gz sont activées dans le menu General Setup comme recommandé dans l'annexe A.

```
# zcat /proc/config.gz | grep XENO_OPT_WATCH
CONFIG_XENO_OPT_WATCHDOG=y
CONFIG_XENO_OPT_WATCHDOG_TIMEOUT=4
#
```

Ici, le délai est configuré à la valeur par défaut (4 secondes). Vérifions-le avec le programme suivant qui réalise des boucles actives de durées croissantes :

```
exemple-watchdog-01.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <rtdk.h>
#include <native/task.h>
#include <native/timer.h>

int main(void)
{
    int err;
    RT_TASK task;
    int nb_secondes;
    int i;

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_print_auto_init(1);

    if ((err = rt_task_shadow(& task, "Essai Watchdog",
                              99, T_JOINABLE)) != 0) {
        fprintf(stderr, "rt_task_shadow: %s\n",
                        strerror(-err));
        exit(EXIT_FAILURE);
    }
    nb_secondes = 1;
    while (1) {
        rt_fprintf(stderr, "Boucle active de %d s.\n",
```

Copyright © 2015 Eyrolles.

```
                                nb_secondes);
            rt_task_sleep(500000000); // 0.5 s
            for (i = 0; i < nb_secondes; i ++)
                rt_timer_spin(1000000000ULL);
            nb_secondes ++;
        }
        return 0;
}
```

L'exécution confirme le délai programmé :

```
# ./exemple-watchdog-01
Boucle active de 1 s.
Boucle active de 2 s.
Boucle active de 3 s.
Boucle active de 4 s.
CPU time limit exceeded
#
```

Synchronisation des tâches

Naturellement, Xenomai offre une panoplie complète de mécanismes classiques de synchronisation – sémaphores, mutex, variables conditions, attente d'événements. Nous allons observer quelques exemples simples en rappelant qu'une description plus complète de l'API est disponible dans la documentation en ligne ou dans le répertoire d'installation de Xenomai.

Sémaphores

Xenomai propose une implémentation des sémaphores classiques de Dijkstra, ainsi que quelques extensions. Un sémaphore est un objet de synchronisation courant dans la programmation temps réel. Il est associé à un compteur initialisé avec une valeur *N* positive. Chaque fois qu'une tâche a besoin d'accéder à la ressource protégée par le sémaphore, elle appelle l'opération `P()`. Si le compteur est strictement supérieur à zéro, il est décrémenté et l'opération se termine immédiatement, sinon la tâche est endormie jusqu'à ce que le compteur redevienne positif (ou que le délai maximal d'attente soit atteint). Lorsqu'une tâche a terminé son accès à la ressource concernée, elle invoque l'opération `V()`, qui incrémente le compteur en réveillant éventuellement une tâche en attente. Nous sommes ainsi sûrs qu'il n'y aura jamais plus de *N* tâches tenant le sémaphore à un instant donné (si *N* est initialisé à `1`, le comportement est approximativement celui d'un mutex).

`P()` et `V()` viennent du hollandais Proberen (tester) et Verhogen (augmenter), langue maternelle de Dijkstra, qui a décrit en premier cette structure de synchronisation.

Copyright © 2015 Eyrolles.

Les fonctions essentielles pour manipuler les sémaphores de Xenomai sont les suivantes :

```
int rt_sem_create (RT_SEM * semaphore, const char * nom,
                   unsigned long valeur, int mode);
int rt_sem_delete (RT_SEM * semaphore);
int rt_sem_p (RT_SEM * semaphore, RTIME delai_maxi);
int rt_sem_v (RT_SEM * semapÐore);
```

Nous allons construire un petit exemple dans lequel six tâches temps réel vont tenter simultanément d'accéder à un sémaphore dont le compteur est initialisé – avec le troisième argument de `rt_sem_create()` – à la valeur 4. Chaque tâche qui obtient l'accès au sémaphore le conserve durant deux secondes, avant de le restituer et de se remettre en attente, ceci à quatre reprises. À chaque fois qu'un thread obtient ou restitue le sémaphore, il l'indique dans une table globale qu'une septième tâche vient consulter régulièrement pour afficher l'état des threads du processus.

```
exemple-semaphore-01.c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <rtdk.h>
#include <native/sem.h>
#include <native/task.h>
#include <native/timer.h>

#define NB_TACHES 6
static RT_SEM  sem;
static int tient_semaphore[NB_TACHES];

void thread (void * arg)
{
    int i;
    int num = (int) arg;
    for (i = 0; i < 4; i ++) {
        tient_semaphore[num] = 0;
        rt_sem_p(& sem, TM_INFINITE);
        tient_semaphore[num] = 1;
        sleep(2);
        rt_sem_v(& sem);
    }
    tient_semaphore[num] = -1; // fini
}

void observateur(void * unused)
{
    int i;
    rt_printf("Avec semaphore    Sans semaphore\n");
```

Copyright © 2015 Eyrolles.

```
    while (1) {
        for (i = 0; i < NB_TACHES; i ++)
            if (tient_semaphore[i] == 1)
                rt_printf("%d ", i);
        rt_printf ("          ");
        for (i = 0; i < NB_TACHES; i ++)
            if (tient_semaphore[i] == 0)
                rt_printf("%d ", i);
        rt_printf("\n");
        rt_task_sleep(1000000000);
    }
}

int main(int argc, char * argv[])
{
    int i;
    int err;
    RT_TASK task[NB_TACHES];
    RT_TASK observ;

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_print_auto_init(1);

    if ((err = rt_sem_create(& sem, "Sem-01",
                             NB_TACHES-2, S_PRIO)) != 0) {
        [...]
    }

    for (i = 0; i < NB_TACHES; i ++) {
        if ((err = rt_task_spawn(& task[i], NULL, 0, 99,
                  T_JOINABLE, thread, (void *)i)!=0)) {
            [...]
        }
    }

    if ((err = rt_task_spawn(& observ, NULL, 0, 99,
                     T_JOINABLE, observateur, NULL)!=0)){
        [...]
    }

    for (i  = 0; i < NB_TACHES; i ++)
        rt_task_join(& task[i]);
   rt_task_delete(& observ);
    rt_sem_delete(& sem);
   return EXIT_SUCCESS;
}
```

Copyright © 2015 Eyrolles.

Ce programme présente un défaut important : l'accès aux cases du tableau `tient_semaphore[]` n'est pas protégé. Il y a donc un risque que la consultation faite par le thread observateur se produise au moment exact d'une écriture par l'un des autres threads et obtienne une valeur incohérente. Dans une application réelle, il conviendrait d'utiliser un autre mécanisme de verrouillage pour protéger l'accès au tableau, comme un mutex que nous verrons plus loin. Lors de l'exécution, nous voyons bien que seules quatre tâches peuvent obtenir simultanément le sémaphore, les deux autres restant en attente.

```
# ./exemple-semaphore-01
Avec semaphore    Sans semaphore
0 1 2 3           4 5
0 1 2 3           4 5
0 1 4 5           2 3
0 1 4 5           2 3
2 3 4 5           0 1
2 3 4 5           0 1
0 1 2 3           4 5
0 1 2 3           4 5
0 1 4 5           2 3
0 1 4 5           2 3
2 3 4 5
2 3 4 5
#
```

Le dernier paramètre de `rt_sem_create()` peut prendre l'une des trois valeurs suivantes, ce qui représente une extension par rapport aux sémaphores usuels.

Valeurs	Signification
`S_PRIO`	Lors d'une opération `V()`, le compteur est incrémenté et on réveille la tâche la plus prioritaire endormie dans une opération `P()` pour qu'elle puisse tester et décrémenter le compteur.
`S_FIFO`	Lors d'une opération `V()`, le compteur est incrémenté, puis on réveille la première tâche endormie, quelle que soit sa priorité.
`S_PULSE`	Lors d'une opération `V()`, on réveille la tâche la plus prioritaire. Si aucune tâche n'est en attente, le compteur n'est pas augmenté. Ainsi, le sémaphore joue un rôle de point de blocage pour chaque `P()` et de libération à chaque `V()`, ce qui rappelle le principe des variables conditions. Dans ce mode, le compteur doit obligatoirement être initialisé avec une valeur nulle (et il la conserve).

Mutex

Les mutex de Xenomai sont principalement utilisés à travers les quatre fonctions suivantes :

```
int rt_mutex_create  (RT_MUTEX * mutex, const char * nom);
int rt_mutex_delete  (RT_MUTEX * mutex);
int rt_mutex_acquire (RT_MUTEX * mutex, RTIME delai);
int rt_mutex_release (RT_MUTEX * mutex);
```

Copyright © 2015 Eyrolles.

Le nom indiqué en second paramètre de `rt_mutex_create()` apparaît sous forme de pseudofichier dans le répertoire `/proc/xenomai/registry/native/mutexes/`.

Le verrouillage du mutex est réalisé par `rt_mutex_acquire()` et sa libération avec `rt_mutex_release()`. Voyons un exemple inspiré de ceux que nous avons construits dans les chapitres 6 et 7 pour mesurer le temps de commutation entre threads.

exemple-mutex-01.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/mutex.h>
#include <native/timer.h>
#include <rtdk.h>

static RT_MUTEX mutex;

#define NB_COMMUTATIONS_MAX  1000
static int nb_commutations = 0;
static unsigned long long int avant_commut;
static unsigned long long int apres_commut;
static unsigned long long int durees[NB_COMMUTATIONS_MAX];


void thread_2 (void * unused)
{
  rt_task_sleep(5000000UL); // 5 ms

  while (nb_commutations < NB_COMMUTATIONS_MAX) {
    rt_mutex_acquire(& mutex, TM_INFINITE);
    apres_commut = rt_timer_read();
    durees[nb_commutations]= apres_commut - avant_commut;
    nb_commutations ++;
    rt_mutex_release(& mutex);
    rt_task_yield();
  }
}


void thread_1 (void * unused)
{
  rt_mutex_acquire(& mutex, TM_INFINITE);

  while (nb_commutations < NB_COMMUTATIONS_MAX) {
    rt_task_sleep(10000000UL); // 10 ms
    avant_commut = rt_timer_read();
    rt_mutex_release(& mutex);
    rt_task_yield();
```

Copyright © 2015 Eyrolles.

```
    rt_mutex_acquire(& mutex, TM_INFINITE);
  }
  rt_mutex_release(& mutex);
}

int main(void)
{
  int i;
  int err;
  RT_TASK task[2];

  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_print_auto_init(1);

  if ((err = rt_mutex_create(& mutex,"Exemple-01"))!=0){
       [...]
    }

    if ((err = rt_task_spawn(& task[0], NULL, 0, 99,
                  T_JOINABLE, thread_1, NULL) != 0)) {
       [...]
    }

    if (rt_task_spawn(& task[1], NULL, 0, 99,
                     T_JOINABLE, thread_2, NULL) != 0) {
       [...]
    }
    rt_task_join(& task[0]);
    rt_task_join(& task[1]);
    rt_mutex_delete(& mutex);

    for (i = 0; i < NB_COMMUTATIONS_MAX; i ++)
        printf("%lld\n", durees_commutations[i]/1000);
   return EXIT_SUCCESS;
}
```

Le thread 1 prend le mutex en premier, attend quelques instants et le libère afin que le thread 2 puisse le prendre. La durée de cette commutation est mesurée. Le mutex 2 relâche ensuite le mutex et le cycle reprend pendant 1 000 commutations.

Les résultats sont affichés à la fin de l'exécution. Nous pouvons les étudier avec les outils développés au chapitre 4.

```
# ./exemple-mutex-01 > data-mutex-01.txt
# ../chapitre-04/calculer-statistiques < data-mutex-01.txt
 Nb mesures = 1000
 Minimum = 2
 Maximum = 11
 Moyenne = 4
 Ecart-type = 1
#
```

Copyright © 2015 Eyrolles.

Les résultats sont comparables à ceux de Linux. La moyenne est de 4 microsecondes au lieu de 3, et le maximum de 11 microsecondes au lieu de 12.

Les mutex de Xenomai disposent automatiquement du mécanisme d'héritage de priorité (chapitre 7). Ceci permet d'éviter les situations d'inversions de priorités que l'on peut rencontrer si une tâche de haute priorité est bloquée en attente d'un mutex tenu par un thread de faible priorité préempté par une tâche de priorité intermédiaire.

Nous allons mettre ceci en évidence en reprenant le même principe que celui du programme `exemple-inversion` du chapitre 7. La tâche T1 (priorité 10) prend le mutex et démarre la tâche T3 (priorité 30) qui se met en attente sur le même mutex. Ensuite, T1 démarre une tâche T2 (priorité 20) qui consomme du temps CPU avant de se terminer.

```
exemple-mutex-02.c  :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/mutex.h>
#include <native/timer.h>
#include <rtdk.h>

static RT_MUTEX  mutex;

void thread_2(void * unused);
void thread_3(void * unused);

void thread_1(void *unused)
{
    int err;
    RT_TASK task_2, task_3;

    rt_fprintf(stderr, "T1 demarre\n");
    rt_fprintf(stderr, "T1 demande le mutex\n");
    rt_mutex_acquire(& mutex, TM_INFINITE);
    rt_fprintf(stderr, "T1 tient le mutex\n");
    rt_fprintf(stderr, "T1 demarre T3\n");
    if ((err = rt_task_spawn(& task_3, NULL, 0, 30,
                    T_JOINABLE, thread_3, NULL) != 0)) {
        fprintf(stderr, "rt_task_spawn:%s\n",
                strerror(-err));
        exit(EXIT_FAILURE);
    }
    rt_fprintf(stderr, "T1 demarre T2\n");
    if ((err = rt_task_spawn(& task_2, NULL, 0, 20,
                    T_JOINABLE, thread_2, NULL) != 0)) {
        fprintf(stderr, "rt_task_spawn:%s\n",
                strerror(-err));
        exit(EXIT_FAILURE);
```

Copyright © 2015 Eyrolles.

```
    }

    rt_fprintf(stderr, "T1 lache le mutex\n");
    rt_mutex_release(& mutex);
    rt_fprintf(stderr, "T1 attend T2 et T3\n");
    rt_task_join(& task_2);
    rt_task_join(& task_3);
    rt_fprintf(stderr, "T1 se termine\n");
}


void thread_2(void * unused)
{
    rt_fprintf(stderr, "    T2 demarre\n");
    rt_fprintf(stderr, "    T2 travaille\n");
    rt_timer_spin(3000000000LLU);
    rt_fprintf(stderr, "    T2 se termine\n");
}


void thread_3(void * unused)
{
    rt_fprintf(stderr, "        T3 demarre\n");
   rt_fprintf(stderr, "        T3 demande le mutex\n");
    rt_mutex_acquire(& mutex, TM_INFINITE);
    rt_fprintf(stderr, "        T3 tient le mutex\n");
   rt_fprintf(stderr, "        T3 travaille\n");
    rt_timer_spin(3000000000LLU);
   rt_fprintf(stderr, "        T3 lache le mutex\n");
    rt_mutex_release(& mutex);
    rt_fprintf(stderr, "        T3 se termine\n");
}


int main(int argc, char * argv [])
{
    int err;
    RT_TASK task;

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_print_auto_init(1);

    if ((err = rt_mutex_create(& mutex,"Ex-02")) != 0) {
        fprintf(stderr, "rt_mutex_create:%s\n",
                 strerror(-err));
        exit(EXIT_FAILURE);
    }
    if ((err = rt_task_spawn(& task, NULL, 0, 10,
                    T_JOINABLE, thread_1, NULL) != 0)) {
        fprintf(stderr, "rt_task_spawn:%s\n",
```

Copyright © 2015 Eyrolles.

```
                strerror(-err));
        exit(EXIT_FAILURE);
    }
    rt_task_join(& task);
    rt_mutex_delete(& mutex);
    return 0;
}
```

Dans une situation d'inversion de priorité, T2 pourrait s'exécuter en entier avant que T1 ne puisse redémarrer, libérer le mutex et laisser T3 se dérouler.

Nous voyons qu'en réalité, il n'en est rien. L'héritage de priorité propulse T1 temporairement à la priorité 30 où elle n'est pas préemptée par T2.

```
# ./exemple-mutex-02
T1 demarre
T1 demande le mutex
T1 tient le mutex
T1 demarre T3
        T3 demarre
        T3 demande le mutex
T1 demarre T2
T1 lache le mutex
        T3 tient le mutex
        T3 travaille
        T3 lache le mutex
        T3 se termine
T1 attend T2 et T3
    T2 demarre
    T2 travaille
    T2 se termine
T1 se termine
```

Conclusion

Nous avons vu quelques primitives de base proposées par Xenomai pour la gestion des tâches et leur synchronisation. L'API complète est naturellement bien plus riche, et l'on trouvera de nombreux détails et exemples dans la documentation en ligne.

Nous ne nous sommes intéressés pour le moment qu'à la programmation Xenomai (et Linux) en mode utilisateur. Il est pourtant parfois nécessaire de développer des modules de code qui viendront se loger directement dans le kernel. Ceci est utile pour gérer et traiter efficacement les interruptions, par exemple, mais également les entrées-sorties vers les périphériques. Nous allons en étudier un aperçu dans le prochain chapitre.

Copyright © 2015 Eyrolles.

Points clés

- Le développement sous Xenomai se fait de manière tout à fait classique. Les spécificités ne tiennent qu'aux options de compilation et d'édition des liens.
- La création de tâches Xenomai et leur synchronisation se programme à l'aide de fonctions relativement simples, proches de l'API Posix de gestion des threads.
- Les performances obtenues avec Xenomai sont en moyenne identiques à celles attendues avec un noyau Linux incluant le patch PREEMPT_RT. Toutefois, les résultats correspondant aux pires cas (latence maximale) sont meilleurs qu'avec le noyau PREEMPT_RT.
- Il existe une grande variabilité des performances en fonction du matériel sous-jacent et de sa configuration (*Setup BIOS*, par exemple). Nous retrouverons quelques conseils de configuration dans l'annexe A.

Exercices

Exercice 1 (*)

Écrivez un programme qui lance un thread Xenomai pour chaque CPU (ou cœur) dont vous disposez. Affectez chaque thread à un CPU en utilisant le paramètre `mode` de `rt_task_create()`. Après un petit sommeil d'une seconde, chaque thread devra se mettre en boucle active pendant une dizaine de secondes.

Vérifiez que l'interface utilisateur du système est totalement gelée pendant cette période. Vérifiez également que le noyau Linux est incapable de répondre à un `ping` depuis un autre poste.

Exercice 2 (**)

Créez une tâche périodique Xenomai qui enchaîne régulièrement des boucles actives avec `rt_timer_spin()`. Relevez l'heure avant et après la boucle active. Affichez la durée effective de chaque boucle. Quelle est la précision de ces temps d'attente ?

Exercice 3 (**)

Renouvelez l'expérience précédente en utilisant un sommeil `rt_task_sleep()` au lieu de la boucle active et vérifiez sa précision. Comparez à la valeur mesurée précédemment.

Exercice 4 (***)

Écrivez un programme qui lance une tâche périodique sur chaque CPU disponible, les périodes étant différentes. Mesurez la durée de chaque période. Quelles sont les précisions des déclenchements périodiques ?

Copyright © 2015 Eyrolles.

11

Traitement des interruptions

Jusqu'à présent, le code que nous avons étudié s'exécutait dans l'espace utilisateur, et c'est ainsi qu'il est conseillé de développer les tâches temps réel. L'isolation mémoire entre les processus et le noyau garantit la robustesse et la sécurité du système, même en cas de bogue dans une application. Toutefois, il peut s'avérer indispensable d'écrire des portions de programme qui s'exécutent dans l'espace kernel, par exemple pour gérer efficacement les interruptions. Ce chapitre constitue une brève introduction à l'écriture de code pour le noyau et ne peut se substituer à une documentation plus approfondie. Nous y étudierons le fonctionnement d'un driver, ainsi que la gestion des interruptions par Linux et par Xenomai.

Programmation d'un driver

Squelette d'un module

Le code que l'on développe pour le noyau Linux est généralement fourni sous forme de module externe (fichier `.ko`, *kernel object*), qui est inséré dynamiquement avec la commande `insmod` et éventuellement retiré par la suite avec `rmmod`. Ceci offre une certaine souplesse pour le développeur et l'administrateur du système. Le squelette général d'un module du noyau est le suivant :

```
exemple-squelette-module.c  :
#include <linux/module.h>

static int __init exemple_squelette_init (void)
{
    printk(KERN_INFO "%s: chargement du module\n",
                     THIS_MODULE->name);
    return 0;
}
```

Copyright © 2015 Eyrolles.

```
static void __exit exemple_squelette_exit (void)
{
    printk(KERN_INFO "%s: retrait du module\n",
                     THIS_MODULE->name);
}

module_init(exemple_squelette_init);
module_exit(exemple_squelette_exit);

MODULE_LICENSE("GPL");
```

Voici quelques explications concernant ce code.

- Un module du noyau ne peut pas inclure les fichiers d'en-tête système (comme `<stdio.h>`) car ils appartiennent à la bibliothèque C de l'espace utilisateur et non à l'espace kernel. Des fichiers d'en-tête spécifiques existent pour le noyau, préfixés par le chemin `<linux/>` ou `<asm/>`. L'inclusion de `<linux/module.h>` est le minimum requis, mais d'autres nous serons utiles.
- Les macros `module_init()` et `module_exit()`, que l'on place traditionnellement en fin de fichier, indiquent les noms de deux routines, qui sont appelées respectivement au chargement avec `insmod` et au retrait avec `rmmod` du module. La première est responsable de l'initialisation du module : elle doit renvoyer une valeur nulle si cette initialisation s'est bien déroulée ou un code d'erreur négatif dans le cas contraire. La seconde se charge de libérer les éventuelles ressources tenues par le module.
- Les directives `__init` et `__exit` sont spécifiques à Linux et indiquent à quel moment la fonction est utilisée, ce qui permet de la placer sur un segment de mémoire particulier. Une fois le chargement du module terminé, le noyau sait que les fonctions (et variables globales) préfixées par `__init` ne seront plus utilisées et il peut ainsi libérer la mémoire qu'elles occupent. Symétriquement, si le noyau ne permet pas le retrait des modules chargés – c'est une option de compilation du kernel – les fonctions et variables précédées par `__exit` ne sont jamais employées, et leur segment mémoire n'a pas besoin d'être chargé.
- La fonction `printk()` permet d'envoyer un message dans les traces du noyau. Elle ressemble à son homologue `printf()` de l'espace utilisateur, toutefois il est usuel de précéder le message par une constante `KERN_INFO`, `KERN_WARNING`, `KERN_DEBUG`, `KERN_ERR`, etc., en fonction de la gravité du message pour qu'il soit traité en conséquence par le système de journalisation des traces. Attention, ces constantes symboliques représentent des chaînes de caractères qui sont concaténées avec le message. Il ne faut donc pas ajouter de virgule entre le niveau de gravité et le message lui-même.

```
int printk (const char *format, ...);
```

- `THIS_MODULE` est un pointeur global qui donne accès à la structure représentant le module concerné. Ainsi, nous pouvons afficher son nom dans les traces du noyau.
- Enfin, il est nécessaire d'indiquer avec la macro `MODULE_LICENSE()` la licence d'utilisation du module. Si celle-ci n'est pas `GPL`, `GPLv2` ou quelques autres variantes compatibles avec la GPL,

Copyright © 2015 Eyrolles.

le driver est considéré comme PROPRIETARY. Il ne pourra pas être intégré statiquement dans le code du noyau et sera obligatoirement chargé comme module. Au chargement, il « tachera » le noyau (qui aura alors l'attribut *tainted*) jusqu'au *reboot*. Certaines fonctionnalités centrales du kernel ne lui seront pas accessibles.

La compilation d'un module nécessite de disposer des fichiers d'en-tête du noyau cible ainsi que de son fichier de configuration .config. Le Makefile qui permet de générer le module s'appuie également sur la chaîne de construction du noyau cible. Compilons notre module :

```
$ make
make -C /lib/modules/3.0.0-15-generic/build SUBDIRS=/exemples/chapitre-11  modules
make[1]: entrant dans le répertoire " /usr/src/linux-headers-3.0.0-15-generic "
  CC [M]  /exemples/chapitre-11/exemple-squelette-module.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /exemples/chapitre-11/exemple-squelette-module.mod.o
  LD [M]  /exemples/chapitre-11/exemple-squelette-module.ko
make[1]: quittant le répertoire " /usr/src/linux-headers-3.0.0-15-generic "
$
```

Pour pouvoir l'insérer dans notre noyau, il est évidemment nécessaire de disposer des droits *root*. Les messages envoyés dans les traces du kernel sont visibles avec la commande dmesg.

```
# insmod exemple-squelette-module.ko
# dmesg | tail
[...]
exemple_squelette_module: chargement du module
# rmmod exemple_squelette_module
# dmesg | tail
[...]
exemple_squelette_module: chargement du module
exemple_squelette_module: retrait du module
#
```

Structure d'un driver

Nous pouvons à présent commencer à envisager la construction d'un mini driver très simplifié, qui offre un point d'entrée dans /dev sous forme de fichier spécial représentant un périphérique caractère. Les données que nous écrirons dans ce fichier spécial seront stockées dans un buffer interne au noyau que nous pourrons consulter lors d'une lecture du même fichier spécial.

Étant donné la longueur du code source, et le nombre d'éléments nouveaux apportés par ce code, il sera présenté en deux parties :

exemple-read-write.c (1/2) :

```
#include <linux/version.h>
#include <linux/device.h>
```

Copyright © 2015 Eyrolles.

```
#include <linux/module.h>
#include <linux/mutex.h>
#include <linux/sched.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

static dev_t          exemple_dev;
static struct cdev    exemple_cdev;
static struct class * exemple_class = NULL;

static int exemple_read (struct file * filp,
     char * buffer, size_t length, loff_t * offset);

static int exemple_write(struct file * filp,
     const char * buffer, size_t lg, loff_t * offset);

static struct file_operations fops_exemple = {
    .owner   =  THIS_MODULE,
    .read    =  exemple_read,
    .write   =  exemple_write,
};

#define NB_MINEURS 1

static int __init exemple_init (void)
{
    int erreur;

    erreur = alloc_chrdev_region(& exemple_dev, 0,
                        NB_MINEURS, THIS_MODULE->name);
    if (erreur < 0)
        return erreur;

    exemple_class = class_create(THIS_MODULE, "exemple");
    if (IS_ERR(exemple_class)) {
        unregister_chrdev_region(exemple_dev, NB_MINEURS);
        return -EINVAL;
    }
    device_create(exemple_class, NULL, exemple_dev,
                  NULL, THIS_MODULE->name);

    cdev_init(& exemple_cdev, & fops_exemple);

    erreur = cdev_add(& exemple_cdev,
                      exemple_dev, NB_MINEURS);
    if (erreur != 0) {
        device_destroy(exemple_class, exemple_dev);
        class_destroy(exemple_class);
        unregister_chrdev_region(exemple_dev, NB_MINEURS);
```

Copyright © 2015 Eyrolles.

```
        return erreur;
    }
    return 0;
}


static void __exit exemple_exit(void)
{
    cdev_del(& exemple_cdev);
    device_destroy(exemple_class, exemple_dev);
    class_destroy(exemple_class);
    unregister_chrdev_region(exemple_dev, NB_MINEURS);
}
```

Commençons par analyser le comportement de la fonction `exemple_init()` invoquée lors du chargement du module. Elle demande tout d'abord au noyau de lui réserver une plage de périphériques caractères. Le choix du numéro majeur (qui identifie le driver) est laissé à la discrétion du kernel. Nous souhaitons obtenir `NB_MINEURS` numéros mineurs différents (qui identifient les instances du périphérique) à partir de la valeur `0`. Ici, nous ne réclamons qu'un seul numéro mineur.

Nous enregistrons ensuite avec `class_create()` notre propre classe de périphériques qui apparaîtra dans `/sys/class/`. Puis, nous y ajoutons un nouveau périphérique avec `device_create()`. L'ensemble des numéros majeur et mineur du périphérique se trouvent dans la structure `dev_t` qui a été remplie durant l'invocation de `alloc_chrdev_region()`. Ces valeurs seront consultées par le démon `udevd` lorsqu'il sera notifié de l'ajout d'un nouveau périphérique. Il pourra ainsi créer dans `/dev` le fichier spécial sur lequel nous opérerons ensuite.

La plupart des drivers courants s'enregistrent dans des classes prédéfinies. Ceci simplifie grandement l'écriture du code, puisque toutes les opérations décrites précédemment sont prises en charge automatiquement. Seule l'écriture des méthodes du driver (comme `read()` et `write()` que nous allons voir plus loin) est indispensable.

```
struct class * class_create  (struct module * module,
                              const char * nom);
void           class_destroy (struct class * classe);

struct device * device_create  (struct class * classe,
                                struct device * parent,
                                dev_t identifiant,
                                void * donnees_privees,
                                const char *nom,...)
void  device_destroy           (struct class * classe,
                                dev_t identifiant);

int  alloc_chrdev_region      (dev_t * premier_ident,
                               unsigned premier_mineur,
                               unsigned nb_mineurs,
```

Copyright © 2015 Eyrolles.

```
                                const char * nom);
void unregister_chrdev_region (dev_t premier_ident,
                                unsigned nb_mineurs);
```

La fonction `cdev_init()` permet d'associer une structure opaque `cdev` représentant un périphérique caractère générique, avec une structure `file_operations` qui contient les méthodes que nous implémentons pour ce fichier spécial. Cette structure est initialisée statiquement plus haut dans le fichier avec une syntaxe un peu particulière, grâce à laquelle nous ne fournissons que les champs qui nous intéressent. Les autres méthodes sont initialisées avec des comportements par défaut qui réussissent toujours – par exemple, pour des appels `open()` ou `close()` – ou qui échouent en indiquant qu'elles ne sont pas implémentées (pour `select()`, par exemple).

```
void cdev_init (struct cdev * periph_caractere,
                const struct file_operations * methodes);
int  cdev_add  (struct cdev * periph_caractere,
                dev_t premier_ident, unsigned nb_mineurs);
void cdev_del  (struct cdev * periph_caractere);
```

Les deux méthodes qui nous intéressent ici sont `read()` et `write()` ; elles sont décrites plus loin.

Enfin, notre fonction d'initialisation enregistre le périphérique caractère en lui attribuant les numéros majeur et mineur que nous avons obtenus au préalable. À partir de ce moment, il devient possible d'invoquer `open()` sur ce fichier spécial depuis l'espace utilisateur pour obtenir un descripteur, et d'appeler `read()` ou `write()` sur ce dernier. Ces invocations appelleront les routines de la seconde partie de notre module.

La fonction `exemple_exit()`, invoquée lors du retrait du module avec `rmmod`, effectue les opérations inverses des précédentes, dans l'ordre opposé à celui de l'initialisation.

exemple-read-write.c (2/2) :

```
#define TAILLE_DATA_EXEMPLE  4096
static char data_exemple [TAILLE_DATA_EXEMPLE];
static int  lg_data_exemple = 0;
DEFINE_MUTEX(mtx_data_exemple);


static int exemple_read (struct file * filp, char * buffer,
                         size_t lg, loff_t * offset)
{
    if (mutex_lock_interruptible(& mtx_data_exemple) != 0)
        return -ERESTARTSYS;

    if (lg > lg_data_exemple)
        lg = lg_data_exemple;

    if (lg > 0) {
        if (copy_to_user(buffer, data_exemple, lg) != 0) {
            mutex_unlock(& mtx_data_exemple);
            return -EFAULT;
```

Copyright © 2015 Eyrolles.

```
        }
        lg_data_exemple -= lg;
        if (lg_data_exemple > 0)
            memmove(data_exemple,
                    & data_exemple[lg], lg_data_exemple);
    }
    mutex_unlock(& mtx_data_exemple);
    return lg;
}


static int exemple_write (struct file * filp,
     const char * buffer, size_t lg, loff_t * offset)
{
    if (mutex_lock_interruptible(& mtx_data_exemple) != 0)
        return -ERESTARTSYS;

    if (lg > (TAILLE_DATA_EXEMPLE - lg_data_exemple))
        lg = TAILLE_DATA_EXEMPLE - lg_data_exemple;

    if (lg > 0) {
        if (copy_from_user(&data_exemple[lg_data_exemple],
                           buffer, lg) != 0) {
            mutex_unlock(& mtx_data_exemple);
            return -EFAULT;
        }
        lg_data_exemple += lg;
    }
    mutex_unlock(& mtx_data_exemple);
    return lg;
}

module_init(exemple_init);
module_exit(exemple_exit);
MODULE_LICENSE("GPL");
```

Nous voyons une variable globale `data_exemple` représentant un buffer dans lequel nous viendrons écrire les données que l'espace utilisateur nous envoie avec l'appel système `write()`, et lire les données à renvoyer lors d'un appel `read()`. Le buffer est accompagné par une variable `lg_data_exemple` indiquant le nombre de caractères qui s'y trouvent, et d'un mutex `mtx_data_exemple` pour le protéger. Ceci évite les situations de concurrence entre des appels système `read()` ou `write()` se déroulant en parallèle sur deux CPU différents ou s'entremêlant sur le même CPU avec un noyau compilé en mode préemptible.

Nous pouvons remarquer que les deux appels système verrouillent consciencieusement le mutex avant d'accéder au contenu du buffer (ou même à la variable `lg_data_exemple`) et le libèrent ensuite. La prise du mutex se fait à l'aide de la fonction `mutex_lock_interruptible` :

- si le mutex est initialement libre, l'appel revient immédiatement après l'avoir verrouillé et renvoie zéro ;

Copyright © 2015 Eyrolles.

- si le mutex est déjà bloqué au moment de `mutex_lock_interruptible()`, cette fonction endort le processus dans un sommeil interruptible. Dès que le mutex devient libre, la routine verrouille le mutex, et se termine en renvoyant zéro ;
- si le processus reçoit un signal alors qu'il est endormi en sommeil interruptible, il est immédiatement réveillé et `mutex_lock_interruptible()` se termine en renvoyant un code non nul. Par convention, l'appel système doit se terminer le plus vite possible en renvoyant l'erreur `-ERESTARTSYS`.

Lors d'un sommeil non interruptible (provoqué par `mutex_lock()`, par exemple) un processus ne peut pas être réveillé prématurément par un signal, même s'il s'agit de `SIGKILL` (de numéro `9`). La réception du signal se fera lorsque le sommeil prendra fin – sur libération du mutex.

```
int  mutex_lock_interruptible (struct mutex * mtx);
void mutex_lock               (struct mutex * mtx);
void mutex_unlock             (struct mutex * mtx);
```

Enfin, outre la gestion du buffer qui n'est pas très compliquée, nous pouvons observer dans les routines précédentes, les invocations de `copy_to_user()` et `copy_from_user()` qui permettent de copier un bloc de données entre l'espace mémoire du kernel et celui du processus appelant. Ces deux fonctions prennent en premier argument le pointeur de destination, puis celui de source et enfin le nombre d'octets à transférer. Elles renvoient le nombre d'octets qui n'ont pas pu être copiés (à cause d'une erreur de gestion mémoire dans l'espace utilisateur). La consigne, si l'une de ces fonctions renvoie une valeur non nulle, est de terminer l'appel système avec l'erreur `-EFAULT`.

```
long copy_to_user   (void __user *dest,
                     const void *source,
                     unsigned long taille) ;
long copy_from_user (void *dest,
                     const void __user * source,
                     unsigned long n)
```

Testons notre module :

```
# insmod exemple-read-write.ko
# ls /sys/class/
ata_device  bdi        dma      firmware  i2c-adapter  mem
ata_link    block      dmi      gpio      input        misc
ata_port    bluetooth  drm      graphics  leds         mmc_host
backlight   bsg        exemple  hwmon     mdio_bus     net
[...]
# ls /sys/class/exemple/
exemple_read_write
# cat /sys/class/exemple/exemple_read_write/dev
251:0
# ls -l /dev/ex*
crw------- 1 root root 251, 0    /dev/exemple_read_write
```

Copyright © 2015 Eyrolles.

```
# echo AZERTYUIOP > /dev/exemple_read_write
# cat /dev/exemple_read_write
AZERTYUIOP
# echo ABC > /dev/exemple_read_write
# echo DEF > /dev/exemple_read_write
# cat /dev/exemple_read_write
ABC
DEF
# rmmod exemple_read_write
#
```

Traitement des interruptions

Nous allons à présent écrire un petit gestionnaire d'interruptions qui prend la forme – très simplifiée – d'un système d'acquisition de données. Notre gestionnaire sera attaché à une interruption au choix ; je conseille d'employer celle du clavier, il enregistrera dans un buffer les instants auxquels les interruptions se produisent. Un driver caractère minimal permettra de venir lire ces valeurs par l'intermédiaire d'un appel système `read()`.

Comme précédemment, nous allons procéder en deux parties pour l'étude du code source.

exemple-read-irq.c (1/2) :

```
#include <linux/interrupt.h>
#include <linux/version.h>
#include <linux/device.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/cdev.h>
#include <linux/fs.h>

#include <asm/uaccess.h>

static int numero_irq = 1;
module_param(numero_irq, int, 0444);

static dev_t          exemple_dev;
static struct cdev    exemple_cdev;
static struct class * exemple_class = NULL;

static int exemple_read (struct file * filp, char * buffer,
                         size_t length, loff_t * offset);

static irqreturn_t exemple_handler(int irq, void * ident);

static struct file_operations fops_exemple = {
    .owner   =  THIS_MODULE,
    .read    =  exemple_read,
};
```

Copyright © 2015 Eyrolles.

```
#define TAILLE_DATA_EXEMPLE  4096
static long int data_exemple [TAILLE_DATA_EXEMPLE];
static int  lg_data_exemple = 0;
static spinlock_t  spl_data_exemple;
static DECLARE_WAIT_QUEUE_HEAD(wq_data_exemple);

#define NB_MINEURS 1

static int __init exemple_init (void)
{
    int erreur;
    erreur = alloc_chrdev_region(& exemple_dev, 0,
                        NB_MINEURS, THIS_MODULE->name);
    if (erreur < 0)
        return erreur;

    exemple_class = class_create(THIS_MODULE, "exemple");
    if (IS_ERR(exemple_class)) {
        unregister_chrdev_region(exemple_dev, NB_MINEURS);
        return -EINVAL;
    }
    device_create(exemple_class, NULL, exemple_dev,
                  NULL, THIS_MODULE->name);

    spin_lock_init(& spl_data_exemple);

    cdev_init(& exemple_cdev, & fops_exemple);

    erreur = cdev_add(& exemple_cdev, exemple_dev,
                      NB_MINEURS);
    if (erreur != 0) {
        device_destroy(exemple_class, exemple_dev);
        class_destroy(exemple_class);
        unregister_chrdev_region(exemple_dev, NB_MINEURS);
        return erreur;
    }
    erreur = request_irq(numero_irq, exemple_handler,
                        IRQF_SHARED, THIS_MODULE->name,
                        THIS_MODULE->name);
    if (erreur != 0) {
        cdev_del(& exemple_cdev);
        device_destroy(exemple_class, exemple_dev);
        class_destroy(exemple_class);
        unregister_chrdev_region(exemple_dev, NB_MINEURS);
        return erreur;
    }
    return 0;
}
static void __exit exemple_exit (void)
```

Copyright © 2015 Eyrolles.

```
{
    free_irq(numero_irq, THIS_MODULE->name);
    cdev_del(& exemple_cdev);
    device_destroy(exemple_class, exemple_dev);
    class_destroy(exemple_class);
    unregister_chrdev_region(exemple_dev, NB_MINEURS);
}
```

Nous pouvons observer quelques éléments nouveaux par rapport au driver précédent. Tout d'abord, la variable globale `numero_irq` représente le numéro de l'interruption à gérer. Elle est initialisée à `1`, mais il sera possible de la modifier au chargement du module en indiquant sa valeur sur la ligne de commandes de `insmod`. C'est le rôle de la macro `module_param()`.

Il y a toujours un buffer `data_exemple`, toutefois il n'est plus protégé par un mutex. Le verrouillage d'un mutex étant susceptible d'endormir le processus, il n'est pas possible de l'invoquer depuis le contexte d'un gestionnaire d'interruptions (qui s'exécute intempestivement sans processus concerné). Aussi, une structure de données spéciale, nommée spinlock, est conçue pour assurer la synchronisation d'accès aux variables communes entre appels système et handlers d'interruption. L'initialisation correcte du spinlock est réalisée par la fonction `spin_lock_init()` dans `exemple_init()`.

```
void spin_lock_init (spinlock_t * spl);
```

Le détail du fonctionnement interne d'un spinlock dépend de la configuration du kernel pour lequel il a été compilé. Le principe est de disposer d'un verrou (rappelant un mutex) sans possibilité de mise en sommeil. Si le spinlock est déjà verrouillé, la fonction `spin_lock()` se met en attente active, bouclant jusqu'à ce qu'il soit libéré par son actuel propriétaire avec `spin_unlock()`.

```
void spin_lock    (spinlock_t * spl);
void spin_unlock  (spinlock_t * spl);
int  spin_trylock (spinlock_t * spl);
```

Ces fonctions peuvent, par exemple, permettre de se prémunir contre un accès simultané depuis deux CPU distincts. Il faut également éviter qu'un appel système qui manipule une variable globale soit interrompu pendant une modification pour laisser exécuter un handler d'interruption accédant à cette même variable. Pour cela, `spin_lock_irqsave()` et `spin_unlock_irqrestore()` vont respectivement inhiber ou activer les interruptions sur le CPU courant.

```
void spin_lock_irqsave       (spinlock_t * spl,
                              unsigned long masque);
void spin_unlock_irqsrestore (spinlock_t * spl,
                              unsigned long masque);
```

On utilisera donc les protections par `spin_lock()` et `spin_unlock()` dans les gestionnaires d'interruptions et celles par `spin_lock_irqsave()` et `spin_unlock_irqrestore()` dans les appels système.

> Le paramètre `masque` passé en argument sert à stocker et restaurer l'état des interruptions activées et celles inhibées.

Copyright © 2015 Eyrolles.

La fonction `request_irq()` nous permet d'installer un handler d'interruption. Le premier argument est celui de l'interruption à capturer. Ici, j'ai décidé d'utiliser l'interruption déclenchée lors de la pression sur une touche du clavier. Pour connaître son numéro, je vous conseille d'exécuter la commande suivante dans un terminal :

```
$ watch -n 0.1  "cat /proc/interrupts"
```

Suivant la localisation de votre système, il faudra utiliser `0.1` ou `0,1` pour demander un affichage tous les dixièmes de seconde. Vous verrez les compteurs d'interruption évoluer au gré de la vie du système et pourrez aisément déterminer quelle interruption est associée aux touches du clavier.

Le second argument de `request_irq()` est le pointeur sur la fonction de traitement de l'interruption, suivi d'un argument contenant des attributs, dont le seul qui nous intéresse ici est `IRQF_SHARED` indiquant que la ligne d'interruption est partagée par plusieurs gestionnaires simultanément. Le handler bas niveau du kernel invoquera successivement chacun d'entre eux, chacun devant renvoyer la valeur `IRQ_HANDLED` si l'interruption le concernait effectivement (par exemple, après avoir interrogé le matériel) ou `IRQ_NONE` dans le cas contraire. Enfin, on fournit le nom du driver (visible dans `/proc/interrupts`) et un pointeur qui servira d'identifiant pour le handler, que nous devrons transmettre lors de la désinstallation du gestionnaire avec `free_irq()`.

```
int request_irq (unsigned int numero_irq,
                 irq_handler_t handler,
                 unsigned long attributs,
                 const char * nom, void * ident) ;
void free_irq   (unsigned int numero_irq,
                 void * ident) ;
```

Nous remarquons également dans le code précédent la déclaration d'une *waitqueue*, une file d'attente, qui servira à bloquer le processus dans l'appel système de lecture en attendant qu'une interruption arrive. Voici la seconde partie du code de notre driver, comportant l'appel système `read()` et le handler de l'interruption capturée.

```
static int exemple_read(struct file * filp, char * buffer,
                        size_t length, loff_t * offset)
{
    char chaine[80];
    unsigned long masque;

    spin_lock_irqsave(& spl_data_exemple, masque);

    while (lg_data_exemple == 0) {
        spin_unlock_irqrestore(& spl_data_exemple, masque);
        if (wait_event_interruptible(wq_data_exemple,
                           (lg_data_exemple != 0)) != 0) {
            printk(KERN_INFO "Sortie sur signal\n");
            return -ERESTARTSYS;
        }
        spin_lock_irqsave(& spl_data_exemple, masque);
    }
```

Copyright © 2015 Eyrolles.

```
    snprintf(chaine, 80, "%ld\n", data_exemple[0]);
    if (length < (strlen(chaine)+1)) {
        spin_unlock_irqrestore(& spl_data_exemple, masque);
        return -ENOMEM;
    }

    lg_data_exemple --;
    if (lg_data_exemple > 0)
        memmove(data_exemple, & (data_exemple[1]),
                lg_data_exemple * sizeof(long int));

    spin_unlock_irqrestore(& spl_data_exemple, masque);

    if (copy_to_user(buffer, chaine,
                     strlen(chaine)+1) != 0)
        return -EFAULT;
    return strlen(chaine)+1;
}

static irqreturn_t exemple_handler(int irq, void * ident)
{
    spin_lock(& spl_data_exemple);
    if (lg_data_exemple < TAILLE_DATA_EXEMPLE) {
        data_exemple[lg_data_exemple] = jiffies;
        lg_data_exemple ++;
    }
    spin_unlock(& spl_data_exemple);
    wake_up_interruptible(& wq_data_exemple);
    return IRQ_HANDLED;
}
```

Nous pouvons observer le principe de l'appel système `read()`.

- Verrouiller le spinlock pour obtenir un accès exclusif à notre buffer.
- Si le buffer est vide, s'endormir en attente dans la waitqueue, après avoir libéré le spinlock, pour que le handler d'interruption puisse remplir le buffer et le reverrouiller après notre réveil.
- Lorsque des données se trouvent dans le buffer, les extraire et les copier vers l'espace utilisateur.
- Libérer le spinlock avant de revenir dans le processus appelant.

La mise en attente sur la waitqueue est obtenue avec `wait_event_interruptible()` qui, comme son nom l'indique, endort le processus dans un sommeil interruptible (susceptible d'être interrompu par un signal). Le premier argument est le nom de la waitqueue, le second est une condition testée par le kernel et la mise en sommeil n'a lieu que si la condition est fausse. Pour éviter tout bogue de concurrence, le test de la condition et la mise en sommeil sont atomiquement liés – ce qui implique que `wait_event_interruptible()` est une macro et non une fonction. Le réveil est réalisé par `wake_up_interruptible()`.

Copyright © 2015 Eyrolles.

```
int  wait_event_interruptible (wait_queue_head_t wq,
                               (condition));
void wake_up_interruptible    (wait_queue_head_t * wq);
```

De son côté, le handler d'interruption fonctionne ainsi.

- Verrouiller le spinlock.
- Si le buffer n'est pas plein, y ajouter une entrée contenant la valeur de `jiffies`. Cette variable globale contient le nombre de ticks système qui se sont déclenchés depuis le *boot*. Elle évolue donc en permanence et nous fournit ainsi une donnée « vivante » pour l'espace utilisateur.
- Libérer le spinlock et réveiller les éventuelles tâches endormies dans la waitqueue.

Voyons un exemple d'exécution. J'ai déterminé que sur mon poste de travail, l'interruption correspondant au clavier (connecté par USB) est l'IRQ 17.

```
# insmod exemple-read-irq.ko numero_irq=17
# cat /dev/exemple_read_irq
719048
719360
719361
719373
719374
[...]
719867
719885
  (Pression sur une touche)
720273
720294
  (Pression sur une touche)
720773
720790
  (Pression sur une touche)
721127
721140
  (Contrôle-C)
# rmmod exemple_read_irq
#
```

On remarque bien durant l'exécution du programme que notre processus appelant est endormi dans l'appel système `read()`, jusqu'à la pression sur une touche où deux interruptions se produisent (pression et relâchement). Le handler alimente alors le buffer et réveille la tâche en attente.

Cet exemple est très simple mais il correspond quand même à un schéma classique pour un driver effectuant de l'acquisition de données. La complexité que l'on rencontre dans un « vrai » driver est due essentiellement au dialogue avec le matériel (détection et négociation des plages d'entrées-sorties, programmation des transferts de données par DMA, acquittement et réactivation des interruptions, etc.).

Copyright © 2015 Eyrolles.

On notera également que si le volume de données à transférer entre l'espace utilisateur et l'espace noyau est important, on évite les appels système `read()` et `write()` qui impliquent une copie des données. Dans cette situation, on préfère employer un `mmap()` qui partage directement des pages de mémoire physique entre processus et kernel. Le périphérique peut lire ou écrire ainsi dans la mémoire projetée (par un transfert DMA) et le processus en attente – souvent dans un `ioctl()` – sera réveillé dès la fin de l'opération.

Traitement en threaded interrupt

Nous avons évoqué au chapitre 8 le fait que le projet PREEMPT_RT permettait de différer automatiquement une partie du traitement des interruptions dans des threads du kernel. Nous pouvons intégrer ceci directement dans notre driver. Il suffit de remplacer `request_irq()` par `request_threaded_irq()` qui prend un argument supplémentaire : une seconde fonction de traitement avec le même prototype que la première, qui sera exécutée dans un thread kernel spécifique.

```
int request_threaded_irq(unsigned int numero_interruption,
                         irq_handler_t handler_irq,
                         irq_handler_t fonction_thread,
                         unsigned long attributs,
                         const char *nom, void * ident);
```

La routine de service `handler_irq()` doit renvoyer la valeur `IRQ_WAKE_THREAD` pour que la seconde soit exécutée.

Nous allons modifier légèrement notre code pour intégrer ceci. Toutefois, plutôt que d'afficher l'instant de déclenchement de l'interruption (variable `jiffies`) comme nous l'avons fait précédemment, nous allons mesurer la durée s'écoulant entre le déclenchement de l'interruption et celui du thread. Pour obtenir des valeurs assez précises, nous allons lire l'heure avec la fonction `getnstimeofday()` qui nous fournit des secondes (depuis la date 01/01/1970) et un complément en nanosecondes. Il faudra alors stocker les heures dans la table `data_exemple` (dont le type sera modifié) et renvoyer leur différence lors de la lecture.

Voici les modifications apportées au programme précédent :

```
exemple-read-threadirq.c  :
[...]
#define TAILLE_DATA_EXEMPLE  4096
struct mesure {
    struct timespec handler;
    struct timespec thread;
};
static struct mesure data_exemple [TAILLE_DATA_EXEMPLE];
static int  lg_data_exemple = 0;
[...]

static int __init exemple_init (void)
{
[...]
```

Copyright © 2015 Eyrolles.

```
    erreur = request_threaded_irq(numero_irq,
                          exemple_handler, exemple_thread,
                          IRQF_SHARED, THIS_MODULE->name,
                          THIS_MODULE->name);
[...]
}

static void __exit exemple_exit (void)
{
 [...]
}


static int exemple_read(struct file * filp, char * buffer,
                        size_t length, loff_t * offset)
{
[...]
   while (lg_data_exemple == 0) {
        [...]
    }
    duree  =  data_exemple[0].thread.tv_sec  * 1000000000
           + data_exemple[0].thread.tv_nsec;
    duree -=  data_exemple[0].handler.tv_sec * 1000000000
           + data_exemple[0].handler.tv_nsec;

    lg_data_exemple --;
    if (lg_data_exemple > 0)
        memmove(data_exemple, & (data_exemple[1]),
                lg_data_exemple * sizeof(struct mesure));
    spin_unlock_irqrestore(& spl_data_exemple, masque);

    snprintf(chaine, 80, "%lld\n", duree);
    if (length < (strlen(chaine)+1)) {
        return -ENOMEM;
    }
[...]
}
```

Les deux fonctions de traitement sont les suivantes :

```
static irqreturn_t exemple_handler(int irq, void * ident)
{
  spin_lock(& spl_data_exemple);
  if (lg_data_exemple < TAILLE_DATA_EXEMPLE) {
     getnstimeofday(&(data_exemple[lg_data_exemple]
                                          .handler));
  }
  spin_unlock(& spl_data_exemple);
  return IRQ_WAKE_THREAD;
}
```

Copyright © 2015 Eyrolles.

```
static irqreturn_t exemple_thread(int irq, void * ident)
{
  unsigned long masque;
  spin_lock_irqsave(& spl_data_exemple, masque);
  if (lg_data_exemple < TAILLE_DATA_EXEMPLE) {
    getnstimeofday(&(data_exemple[lg_data_exemple]
                                          .thread));
    lg_data_exemple ++;
  }
  spin_unlock_irqrestore(& spl_data_exemple, masque);
  wake_up_interruptible(& wq_data_exemple);
  return IRQ_HANDLED;
}
```

Voici un exemple d'exécution :

```
# insmod  exemple-read-threadirq.ko  numero_irq=17
# cat  /dev/exemple_read_threadirq
12162
9940
9996
9489
8139
10564
9349
10515
8602
11763
9196
10203
9925
8977
10526
 (Contrôle-C)
# cat  /dev/exemple_read_threadirq >  data_read_threadirq.txt
 (Nombreuses pressions de touches, puis Contrôle-C)
#
```

Nous pouvons d'ailleurs observer le thread du kernel affecté au traitement de notre routine, il est visible avec `ps aux` :

```
# ps aux
USER  PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root    1 0.0  0.0 3320 1800 ?   Ss Jan27 0:02 /sbin/init
root    2 0.0  0.0    0    0 ?   S  Jan27 0:00 [kthreadd]
root    3 0.0  0.0    0    0 ?   S  Jan27 0:28 [ksoftirqd/0]
[…]
root 4123 0.0  0.0    0    0 ?   S  10:42 0:00 [irq/17-exemple_]
[...]
#
```

Copyright © 2015 Eyrolles.

Il est possible de modifier la priorité d'ordonnancement de ce thread à l'aide de la commande `chrt` que nous avons déjà utilisée.

Analysons les résultats obtenus, sans oublier que les valeurs sont en nanosecondes :

```
# ../chapitre-04/calculer-statistiques < data_read_threadirq.txt
 Nb mesures = 2632
 Minimum = 5027
 Maximum = 125940
 Moyenne = 8823
 Ecart-type = 3528
#
```

Ces mesures ont été réalisées sur un noyau vanilla, sans extensions temps réel. Elles montrent néanmoins que le traitement réalisé dans le thread est différé en moyenne de près de 9 microsecondes, avec un retard maximal de plus de 125 microsecondes. Ceci explique pourquoi cette approche n'est pas encore systématiquement appliquée sur le noyau standard : les performances moyennes sont légèrement dégradées.

En réitérant cette expérience sur un noyau avec patch PREEMPT_RT, les résultats sont un peu meilleurs :

```
# ../chapitre-04/calculer-statistiques < data_read_thread-rt.txt
 Nb mesures = 3511
 Minimum = 3944
 Maximum = 34485
 Moyenne = 8140
 Ecart-type = 1680
#
```

Le pire cas est passé de 125 microsecondes à 34 microsecondes et la moyenne est également légèrement inférieure. Si nous recommençons en augmentant la priorité du thread à la valeur Fifo 99, on obtient :

```
# ../chapitre-04/calculer-statistiques < data_read_thread-99.txt
 Nb mesures = 6394
 Minimum = 1888
 Maximum = 29928
 Moyenne = 4854
 Ecart-type = 1144
#
```

Cette fois, il s'agit de la valeur moyenne qui est sensiblement améliorée, le thread étant plus facilement sélectionné dès la fin du handler d'interruption même si un autre thread du kernel est actif.

Copyright © 2015 Eyrolles.

Nous avons vu comment traiter les interruptions depuis l'espace noyau, et communiquer les données qu'elles fournissent à l'espace utilisateur, ceci avec un noyau Linux standard ou modifié par le patch PREEMPT_RT. Il est également possible de traiter les interruptions avec Xenomai.

Interruptions avec Xenomai

La version 2.6 de Xenomai – la dernière disponible au moment de la rédaction de ces lignes – offre une possibilité qui va disparaître dans la version 3.0 à venir : la gestion des interruptions depuis l'espace utilisateur dans un processus Xenomai.

Cette fonctionnalité repose sur un objet `RT_INTR` que l'on crée avec le numéro de l'interruption à traiter, que l'on active avec `rt_intr_enable()` et sur lequel le thread vient régulièrement se mettre en attente avec `rt_intr_wait()`.

Il est d'ores et déjà déconseillé d'utiliser ce mécanisme, aussi je ne détaillerai pas le code du processus `exemple-irq-xenomai.c` qui se trouve avec les fichiers sources de ce livre.

La gestion des tâches dédiées à l'espace kernel est effectuée dans le cadre d'une skin de Xenomai, nommée RTDM *(Real Time Driver Model).*

Real Time Driver Model

Le concept RTDM, présenté dans un article de Jan Kiszka [KISZKA 2005], n'est pas uniquement attaché à Xenomai, mais c'est aujourd'hui le seul environnement qui implémente entièrement son API. Il sert d'interface entre la partie applicative (dans l'espace utilisateur) et le driver dans l'espace noyau, mais offre aussi des possibilités d'interface entre drivers.

Les périphériques gérés par RTDM peuvent être de deux types :

- périphériques nommés : ceux auxquels on accède par les appels système `open()`, `close()`, `read()`, `write()`, `ioctl()`, etc. Ils disposent d'un espace de noms indépendant de celui de Linux (qui emploie les fichiers spéciaux de `/dev`) ;
- périphériques protocoles : ces périphériques sont accessibles par `socket()` et `close()`, et implémentent la gestion de messages *via* `sendto()` et `recvfrom()`.

Nous allons voir un premier exemple – similaire au module `exemple-read-write.c` du début de ce chapitre – nous permettant d'observer les mécanismes de communication entre l'espace utilisateur et l'espace noyau.

exemple-rtdm-read-write.c (1/2) :

```
#include <linux/interrupt.h>
#include <linux/version.h>
#include <linux/device.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/cdev.h>
#include <linux/fs.h>
```

Copyright © 2015 Eyrolles.

```
#include <asm/uaccess.h>

#include <rtdm/rtdm_driver.h>

#define exemple_nom_driver "exemple_rtdm"
#define exemple_nom_device "rtdev_exemple"

static int exemple_read(struct rtdm_dev_context *ctxt,
                        rtdm_user_info_t * info,
                        void * buffer, size_t lg);

static int exemple_write(struct rtdm_dev_context *ctxt,
                        rtdm_user_info_t * info,
                        const void * buffer, size_t lg);

static int exemple_open_nonrt (struct rtdm_dev_context *c,
                               rtdm_user_info_t * info,
                               int flags);

static int exemple_close_nonrt(struct rtdm_dev_context *c,
                               rtdm_user_info_t * info);

static struct rtdm_device exemple_rtdev = {
    .struct_version = RTDM_DEVICE_STRUCT_VER,
    .device_flags   = RTDM_NAMED_DEVICE,
    .context_size   = 0,
    .device_name    = exemple_nom_device,
    .open_nrt       = exemple_open_nonrt,
    .ops = {
        .close_nrt = exemple_close_nonrt,
        .read_nrt  = exemple_read,
        .write_nrt = exemple_write,
    },
    .device_class     = RTDM_CLASS_TESTING,
    .device_sub_class = 1,
    .profile_version  = 1,
    .driver_name      = exemple_nom_driver,
    .driver_version   = RTDM_DRIVER_VER(1,0,0),
    .peripheral_name  = exemple_nom_driver,
    .provider_name    = "cpb",
    .proc_name        = exemple_nom_driver,
};

#define TAILLE_DATA_EXEMPLE  4096
static char data_exemple [TAILLE_DATA_EXEMPLE];
static int  lg_data_exemple = 0;
static rtdm_mutex_t mtx_data_exemple;

static int __init exemple_init (void)
{
```

Copyright © 2015 Eyrolles.

```
    rtdm_mutex_init(&mtx_data_exemple);
    rtdm_dev_register(& exemple_rtdev);
    return 0;
}

static void __exit exemple_exit (void)
{
    rtdm_dev_unregister(& exemple_rtdev, 0);
    rtdm_mutex_destroy(&mtx_data_exemple);
}
```

Dans cette première partie, nous pouvons observer quelques éléments nouveaux par rapport aux drivers Linux. Tout d'abord, la structure `rtdm_device` intègre tout ce qui concerne le driver, ce qui permet de l'enregistrer de manière plus simple qu'avec le noyau classique. Dans cette structure, nous voyons un champ nommé `ops` qui regroupe les méthodes – les appels système – implémentées par le driver. Chacune d'elles reçoit en argument un paramètre de type `rtdm_dev_context` qui contient les informations d'exécution temps réel, et un paramètre de type `rtdm_user_info_t` qui correspond à l'environnement du processus appelant.

Chaque méthode existe en version temps réel et en version non temps réel (avec le suffixe `_nrt`). L'implémentation de certaines méthodes (`open` et `close`, par exemple) est obligatoire en version non temps réel. Si l'implémentation temps réel n'est pas fournie, l'implémentation non temps réel peut être invoquée à sa place.

Nous voyons qu'il existe des objets de synchronisation spécifiques (ici les `rtdm_mutex`, mais on peut trouver des sémaphores, des verrous, des événements, etc.).

Voyons la seconde partie du code, avec les méthodes `read` et `write`, qui sont directement calquées sur celles définies précédemment pour le driver Linux. Nous définissons également deux méthodes `open` et `close` qui n'ont pas d'effet ici mais qui sont obligatoires pour RTDM.

```
exemple-rtdm-read-write (2/2) :

static int exemple_read(struct rtdm_dev_context * ctxt,
                        rtdm_user_info_t * info,
                        void * buffer, size_t lg)
{
    rtdm_mutex_lock(& mtx_data_exemple);

    if (lg > lg_data_exemple)
        lg = lg_data_exemple;
    if (lg > 0) {
        if (rtdm_safe_copy_to_user(info, buffer,
                                   data_exemple,lg)!= 0){
            rtdm_mutex_unlock(& mtx_data_exemple);
            return -EFAULT;
        }
```

Copyright © 2015 Eyrolles.

```
        lg_data_exemple -= lg;
        if (lg_data_exemple > 0)
            memmove(data_exemple, & data_exemple[lg],
                    lg_data_exemple);
    }
    rtdm_mutex_unlock(& mtx_data_exemple);
    return lg;
}


static int exemple_write(struct rtdm_dev_context * ctxt,
                         rtdm_user_info_t * info,
                         const void * buffer, size_t lg)
{
    rtdm_mutex_lock(& mtx_data_exemple);

    if (lg > (TAILLE_DATA_EXEMPLE - lg_data_exemple))
        lg = TAILLE_DATA_EXEMPLE - lg_data_exemple;

    if (lg > 0) {
        if (rtdm_safe_copy_from_user(info,
                          &data_exemple[lg_data_exemple],
                          buffer, lg) != 0) {
            rtdm_mutex_unlock(& mtx_data_exemple);
            return -EFAULT;
        }
        lg_data_exemple += lg;
    }
    rtdm_mutex_unlock(& mtx_data_exemple);
    return lg;
}

static int exemple_open_nonrt(struct rtdm_dev_context *c,
                              rtdm_user_info_t * info,
                              int flags)
{
    return 0;
}

static int exemple_close_nonrt(struct rtdm_dev_context *c,
                               rtdm_user_info_t * info)
{
    return 0;
}

module_init(exemple_init);
module_exit(exemple_exit);
MODULE_LICENSE("GPL");
```

Copyright © 2015 Eyrolles.

L'utilisation des `rtdm_mutex` est équivalente à celle des mutex de Linux. Nous remarquons également la présence de fonctions de copie de données depuis ou vers l'espace utilisateur. Chargeons notre module :

```
# insmod exemple-rtdm-read-write.ko
# ls /proc/xenomai/rtdm/
exemple_rtdm      fildes               named_devices   open_fildes
protocol_devices  rttest-switchtest0  rttest-timerbench0
# cat /proc/xenomai/rtdm/named_devices
Hash Name                Driver            /proc
24   rttest-timerbench0  xeno_timerbench    rttest-timerbench0
55   rttest-switchtest0  xeno_switchtest    rttest-switchtest0
74   rtdev_exemple       exemple_rtdm       exemple_rtdm
# cat /proc/xenomai/rtdm/exemple_rtdm/information
driver:      exemple_rtdm
version:     1.0.0
peripheral:  exemple_rtdm
provider:    cpb
class:       6
sub-class:   1
flags:       NAMED_DEVICE
lock count:  0
#
```

Notre driver est bien chargé. Le périphérique est nommé `rtdev_exemple`. Il faut comprendre que contrairement à Linux, RTDM n'emploie pas de fichiers spéciaux pour identifier les périphériques, ni de numéros majeur ou mineur. Il utilise simplement le nom que l'on fournit au moment de l'enregistrement du driver. Il n'est donc pas possible d'accéder aux méthodes de notre driver en employant `cat` ou `echo` en ligne de commandes. Nous devons écrire des petits programmes employant l'API utilisateur de RTDM.

Voici un petit programme approximativement équivalent à `cat`.

exemple-user-rtdm-read.c :

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <rtdm/rtdm.h>

#define LG_BUFFER 80

int main (int argc, char * argv[])
{
    int fd;
    int i;
    char buffer[LG_BUFFER];
```

Copyright © 2015 Eyrolles.

```
    if (argc < 2) {
        fprintf(stderr, "usage: %s rtdm_device\n",
                        argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = rt_dev_open(argv[1], O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "%s: %s\n", argv[1],
                        strerror(-fd));
        exit(EXIT_FAILURE);
    }

    while ((i = rt_dev_read(fd, buffer, LG_BUFFER)) > 0){
        buffer[i] ='\0';
        printf("%s\n", buffer);
    }
    rt_dev_close(fd);
    return EXIT_SUCCESS;
}
```

Nous voyons que les appels système `rt_dev_open()`, `rt_dev_read()` et `rt_dev_close()` sont très semblables à ceux de Linux. Nous pouvons ajouter un utilitaire d'écriture en modifiant simplement les lignes centrales du programme :

```
exemple-user-rtdm-write.c :
[...]
    fd = rt_dev_open(argv[1], O_WRONLY);
    if (fd < 0) {
        fprintf(stderr, "%s: %s\n", argv[1],
                        strerror(-fd));
        exit(EXIT_FAILURE);
    }

    for (i = 2; i < argc; i ++)
        rt_dev_write(fd, argv[i], strlen(argv[i]));

    rt_dev_close(fd);
[...]
```

Ce programme fonctionne bien entendu comme son homologue de l'espace Linux :

```
# export LD_LIBRARY_PATH=/usr/xenomai/lib/
# ./exemple-user-rtdm-write rtdev_exemple "Hello RTDM"
# ./exemple-user-rtdm-read  rtdev_exemple
Hello RTDM
#
```

Copyright © 2015 Eyrolles.

Interruptions avec RTDM

La gestion des interruptions avec RTDM se fait de manière proche de celle avec Linux. Il est possible de partager une ligne d'interruption entre deux drivers RTDM, mais il est fortement déconseillé de la partager avec un driver du kernel Linux.

Dans notre prochain exemple, nous allons utiliser une ligne d'interruption spécifique, contrôlée indépendamment des périphériques classiques du système. Pour cela, il va falloir abandonner l'architecture PC type bureautique employée jusqu'ici et se rapprocher des systèmes embarqués.

> Pendant longtemps, il était possible d'utiliser le port parallèle des PC pour effectuer des entrées/sorties facilement (lecture/écriture sur les ports 888, 889 et 890, et gestion de l'interruption 7). De nos jours, la plupart des PC n'ont plus de véritable port parallèle, certains disposent encore d'une émulation basée sur un contrôleur USB, mais elle ne convient pas pour notre exemple.

J'ai choisi d'utiliser le Raspberry Pi modèle 1 B+ sur lequel j'ai installé Xenomai dans le chapitre 9. L'un des intérêts de cette carte est la disponibilité sur un connecteur d'extension de plusieurs broches d'entrées-sorties facilement accessibles et pilotables par des ports GPIO *(General Purpose Input/Output).*

> Les GPIO sont des ports d'entrées-sorties que l'on trouve sur les microcontrôleurs et certains microprocesseurs. Chaque port peut être configuré à la demande en entrée ou en sortie au choix du programmeur (en fonction, bien sûr, du matériel externe auquel le port est relié). Sous Linux, il est très facile de manipuler depuis l'espace utilisateur les ports GPIO grâce au pseudosystème de fichiers `/sys/class/gpio/`.

Je vais donc envoyer une impulsion sur un port d'entrée afin qu'elle déclenche une interruption gérée par RTDM. Dans ce handler d'interruption, nous basculerons l'état d'un port de sortie, observable à l'oscilloscope (ou avec une simple Led pour la mise au point).

Nous allons utiliser quelques fonctions de l'API Linux pour contrôler les entrées-sorties GPIO :

```
int  gpio_request (unsigned gpio, const char * nom);
void gpio_free    (unsigned gpio);
int  gpio_direction_input (unsigned gpio);
int  gpio_direction_output (unsigned gpio, int valeur);
int  gpio_get_value (unsigned gpio);
void gpio_set_value (unsigned gpio, int valeur);
int  gpio_to_irq (unsigned int gpio)
```

La fonction `gpio_request()` réserve un numéro de GPIO pour le module, afin d'éviter les collisions entre plusieurs drivers. `gpio_direction_input()` et `gpio_direction_output()` orientent le port en entrée ou en sortie. La lecture de la valeur sur un port d'entrée se fait avec `gpio_get_value()` et l'écriture sur un port de sortie avec `gpio_set_value()`. Enfin, `gpio_to_irq()` permet de retrouver le numéro de l'interruption associée à un changement d'état sur un port d'entrée.

Copyright © 2015 Eyrolles.

Les numéros des GPIO disponibles et de ceux utilisés par les périphériques externes sont documentés sur le site raspberrypi.org . Ici, j'ai choisi en entrée la broche 11 (GPIO 17) et en sortie la broche 12 (GPIO 18). Le programme suivant installe un handler d'interruption pour RTDM.

Cette interruption sera déclenchée lorsqu'une transition montante de 0 à 3,3 V sera détectée sur la broche d'entrée. Le handler invoqué va alors basculer l'état du port de sortie.

```
exemple-rtdm-irq.c  :

#include <linux/interrupt.h>
#include <linux/version.h>
#include <linux/device.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/cdev.h>
#include <linux/gpio.h>
#include <linux/fs.h>

#include <asm/uaccess.h>
#include <rtdm/rtdm_driver.h>

static rtdm_irq_t irq_exemple;
static int exemple_handler(rtdm_irq_t * irq);

#define EXEMPLE_GPIO_IN  17
#define EXEMPLE_GPIO_OUT 18

static int __init exemple_init (void)
{
    int err;
    int num_irq = gpio_to_irq(EXEMPLE_GPIO_IN);

    if ((err = gpio_request(EXEMPLE_GPIO_IN,
                            THIS_MODULE->name)) != 0) {
        return err;
    }
    if ((err = gpio_direction_input(EXEMPLE_GPIO_IN)) !=0){
        gpio_free(EXEMPLE_GPIO_IN);
        return err;
    }
    if ((err = gpio_request(EXEMPLE_GPIO_OUT,
                            THIS_MODULE->name)) != 0) {
        gpio_free(EXEMPLE_GPIO_IN);
        return err;
    }
    if ((err = gpio_direction_output(EXEMPLE_GPIO_OUT,
                                     1)) != 0) {
        gpio_free(EXEMPLE_GPIO_OUT);
        gpio_free(EXEMPLE_GPIO_IN);
        return err;
    }
```

Copyright © 2015 Eyrolles.

```
    if ((err = rtdm_irq_request(& irq_exemple,
                    num_irq, exemple_handler,
                    0,
                    "Exemple", NULL)) != 0) {
        gpio_free(EXEMPLE_GPIO_OUT);
        gpio_free(EXEMPLE_GPIO_IN);
        return err;
    }
    irq_set_irq_type(num_irq, IRQF_TRIGGER_RISING);
    rtdm_irq_enable(& irq_exemple);
    return 0;
}


static void __exit exemple_exit (void)
{
    rtdm_irq_disable(& irq_exemple);
    rtdm_irq_free(& irq_exemple);
    gpio_free(EXEMPLE_GPIO_OUT);
    gpio_free(EXEMPLE_GPIO_IN);
}


static int exemple_handler(rtdm_irq_t * irq)
{
    static int value = 0;
    gpio_set_value(EXEMPLE_GPIO_OUT, value);
    value = 1 - value;
    return RTDM_IRQ_HANDLED;
}

module_init(exemple_init);
module_exit(exemple_exit);
MODULE_LICENSE("GPL");
```

Nous voyons que les primitives de RTDM utilisées sont très simples :

```
int rtdm_irq_request (rtdm_irq_t *  irq,
                      unsigned int numero,
                      int (* handler)(rtdm_irq_t *),
                      unsigned long flags,
                      const char * nom,
                      void * arg);
```

Le type opaque `rtdm_irq_t` du premier argument contient toutes les informations de gestion de l'interruption. Il faut le transmettre aux fonctions comme :

```
int rtdm_irq_free    (rtdm_irq_t * irq);
int rtdm_irq_enable  (rtdm_irq_t * irq);
int rtdm_irq_disable (rtdm_irq_t * irq);
```

Copyright © 2015 Eyrolles.

Après compilation, nous pouvons charger le module sur le Raspberry Pi :

```
# insmod exemple-rtdm-irq.ko
# cat /proc/interrupts
           CPU0
  3:      20768   ARMCTRL  BCM2708 Timer Tick
 32:          1   ARMCTRL  dwc_otg, dwc_otg_pcd, dwc_otg_hcd:usb1
 65:          4   ARMCTRL  ARM Mailbox IRQ
 66:          0   ARMCTRL  VCHIQ doorbell
 75:          1   ARMCTRL
 77:       9553   ARMCTRL  bcm2708_sdhci (dma)
 79:          0   ARMCTRL  bcm2708_i2c.0, bcm2708_i2c.1
 80:          0   ARMCTRL  bcm2708_spi.0
 83:       4835   ARMCTRL  uart-pl011
 84:      10565   ARMCTRL  mmc0
FIQ:              usb_fiq
Err:          0
#
```

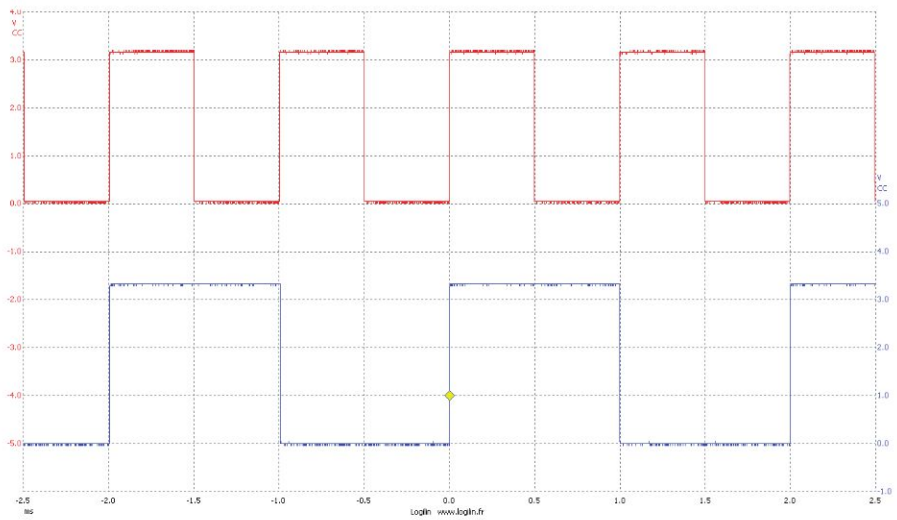
L'interruption (numéro 187) n'est pas vue par le noyau Linux. En revanche, Xenomai peut la gérer :

```
# cat /proc/xenomai/irq
  IRQ        CPU0
    3:       23269        [timer]
  187:      304556        Exemple
 1027:           0        [virtual]
#
```

Si nous connectons un oscillateur à 1 kHz sur l'entrée 11 (GPIO 17) et que nous examinons ce signal et la broche de sortie 12 (GPIO 18) sur un oscilloscope, nous observons le tracé de la figure 11-1.

Figure 11-1

Capture oscilloscope (large)



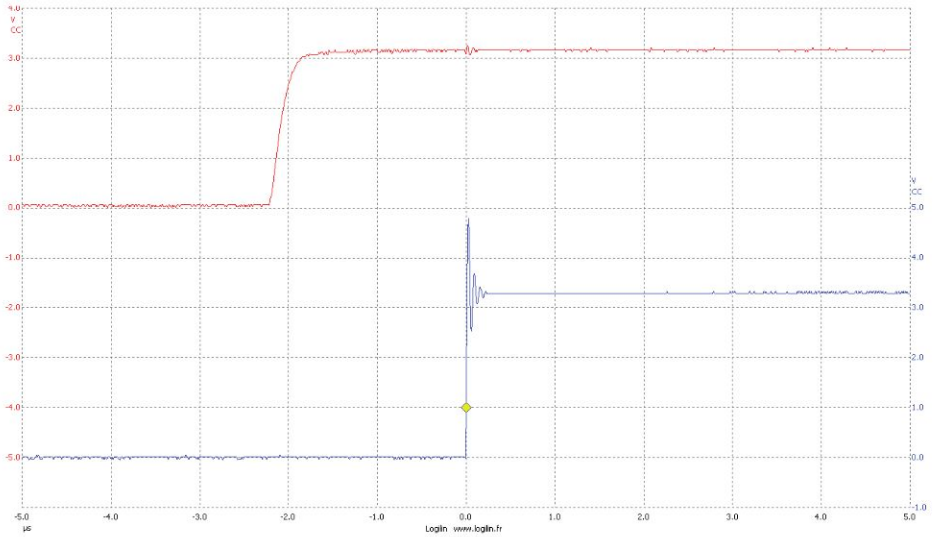
Copyright © 2015 Eyrolles.

À chaque front montant du signal supérieur (entrée), l'interruption se déclenche et le handler bascule l'état de la broche de sortie (signal inférieur). La forme du signal de sortie est perturbée par des effets capacitifs, probablement dus aux câbles que j'emploie, aux limitations de mon oscilloscope et à celles des sorties GPIO du Raspberry Pi.

Nous pouvons également zoomer sur la transition afin de mesurer le temps de latence entre le déclenchement de l'interruption et l'exécution du handler. Sur la figure 11-2, nous voyons que ce temps est d'environ 3,5 microsecondes. Il y a naturellement quelques fluctuations, mais leur amplitude est assez limitée.

Figure 11-2

Capture oscilloscope (serrée)



Dans le cas d'un système temps réel strict, il serait nécessaire de mesurer le plus long temps de latence, sous des conditions élevées de charge système et d'activité en interruption. Ceci peut se réaliser en mettant en œuvre une procédure de mesure semblable à celle que nous avons établie au chapitre 6. J'ai réalisé quelques expériences de ce type décrites sur mon blog, mais il en existe d'autres disponibles sur Internet. Je mentionnerai l'article [BROWN 2010] qui compare des tests de stabilité de timers sur plusieurs versions de Linux, PREEMPT_RT et Xenomai.

Conclusion

Dans ce chapitre, nous avons observé les rudiments de la gestion des interruptions dans le noyau, que ce soit sous Linux standard (éventuellement modifié avec PREEMPT_RT) ou en utilisant l'API RTDM pour Xenomai.

La mise au point d'un driver pour Linux n'est pas très compliquée, mais nécessite une bonne rigueur d'écriture et une certaine connaissance de l'API interne du noyau. L'emploi de RTDM permet d'utiliser une interface plus standardisée, plus uniforme et mieux documentée. Nous n'avons fait qu'effleurer le contenu de RTDM, cette API offre des fonctionnalités de gestion des timers, des tâches, des outils de synchronisation, etc., que l'on retrouvera dans la documentation disponible à l'adresse suivante : http://www.xenomai.org.

Copyright © 2015 Eyrolles.

Points clés

- La gestion des interruptions depuis l'espace utilisateur n'est pas possible sous Linux. Il est nécessaire d'écrire du code dans le noyau.
- L'écriture d'un driver s'appuie sur une interface spécifique, interne au kernel – qui est susceptible d'évoluer entre deux versions majeures de Linux – dont la documentation formelle est limitée.
- Il existe une API, nommée RTDM *(Real Time Driver Model)*, aujourd'hui intégrée principalement dans Xenomai qui permet l'écriture de drivers de façon plus standardisée que Linux, en différenciant les modes d'invocation (depuis un contexte temps réel ou non).
- La possibilité de gérer les interruptions directement depuis les threads Xenomai dans l'espace utilisateur va disparaître dans les futures versions.

Exercices

Exercice 1 (*)

Écrivez un petit module pour le noyau Linux (en vous inspirant de `exemple-read-write.c`) qui implémente une file de messages. Les messages écrits dans la file avec `write()` seront mémorisés avant d'être extraits dans l'ordre d'arrivée par des `read()`.

Exercice 2 (**)

Écrivez un handler d'interruption pour Linux que vous attachez à l'interruption souris et qui compte le nombre d'interruptions survenues. La lecture du compteur se fera par l'intermédiaire d'un appel système `read()` sur un driver minimal. L'écriture sur ce driver permettra de réinitialiser le compteur à zéro.

Exercice 3 (**)

En utilisant l'API de RTDM décrite sur http://www.xenomai.org, écrivez une tâche temps réel périodique qui lit l'heure dès son activation et calcule la durée écoulée depuis le réveil précédent. Elle effectuera des statistiques (durée minimale, maximale, moyenne) qu'elle transmettra à un processus de l'espace utilisateur *via* un appel système `rtdm_read()`.

Exercice 4 (***)

Si vous avez accès à une carte offrant des possibilités de sorties sur des ports (GPIO, par exemple), écrivez un module RTDM qui bascule périodiquement l'état d'une broche de sortie. En la connectant à un oscilloscope, vérifiez la précision du signal produit et sa bonne tenue suivant la charge en processus et en interruptions.

Copyright © 2015 Eyrolles.

Conclusion

État des lieux et perspectives

Nous avons, au gré des chapitres de ce livre, exploré différentes possibilités pour implémenter sous Linux des applications soumises à des contraintes temporelles. Il est temps de récapituler les résultats obtenus et de dresser un état des lieux.

Situation actuelle

Linux « vanilla »

Le noyau Linux standard, celui proposé directement par les distributions classiques, est surnommé vanilla. On peut en télécharger les sources sur le site http://www.kernel.org et le compiler en suivant les étapes et les conseils décrits dans l'annexe A.

Sans précision particulière, une tâche (dont nous avons étudié les caractéristiques et les états au chapitre 1) est exécutée suivant un ordonnancement temps partagé. Le but de l'ordonnanceur temps partagé (chapitre 3) est d'offrir une répartition du temps CPU la plus équitable possible entre les différentes tâches. Aussi, il ne garantit aucun délai, aucune précision temporelle. Nous avons remarqué (chapitre 4) des fluctuations sensibles dans les actions périodiques (plusieurs millisecondes) et des durées de préemption importantes même pour les tâches à qui nous donnions une priorité temps partagé élevée (à l'aide de la commande `nice`).

Il est également possible de faire exécuter une tâche sous un ordonnancement temps réel souple comme nous l'avons observé au chapitre 5. Attention, pensez dans ce cas à désactiver le « garde-fou » (`/proc/sys/kernel/sched_rt_runtime_us`) qui conserve toujours un peu de ressources CPU pour les processus temps partagé, au détriment des processus temps réel. L'ordonnancement

Copyright © 2015 Eyrolles.

temps réel s'obtient à l'aide de primitives Posix standards (`sched_setscheduler()`, etc.) ou à l'aide de la commande shell `chrt`.

Un thread temps réel est peu sensible à la charge logicielle du système, mais il est toutefois soumis à l'activité du noyau lui-même, notamment aux traitements réalisés en réponse aux interruptions et aux appels système (parfois longs) invoqués par d'autres tâches. Pour améliorer ceci, nous avons observé que la préemptibilité du kernel (option de compilation décrite dans l'annexe A) permet de commuter plus rapidement vers une tâche temps réel lorsqu'un événement externe la réveille, même si un autre processus est en train d'exécuter un appel système.

Patch PREEMPT_RT

Le projet PREEMPT_RT a pour vocation d'améliorer les performances temps réel du noyau standard. Il s'agit d'un patch que l'on applique avant compilation, comme nous l'avons vu au chapitre 8. Les avancées de PREEMPT_RT touchent de très nombreux domaines du kernel, et rendent les tâches temps réel beaucoup plus prévisibles en ce qui concerne les comportements temporels.

Un avantage de cette solution est qu'il n'est pas nécessaire de modifier le code existant, ni même de le recompiler ou d'installer de bibliothèque spécifique. Le seul fait de *booter* sur un noyau Linux modifié par le patch PREEMPT_RT confère au système une meilleure qualité du temps réel, suffisante pour de nombreux domaines d'application.

Xenomai

L'approche de Xenomai (et de quelques autres systèmes comme RT-Linux ou RTAI) repose aujourd'hui sur deux noyaux : Adeos comme base temps réel strict, et Linux pour la partie temps partagé et temps réel souple.

Les performances sont comparables et souvent meilleures que celles de PREEMPT_RT. Toutefois, l'inconvénient majeur est la nécessité d'utiliser une API spécifique (chapitre 10), même si des skins permettent d'importer du code en provenance d'autres systèmes temps réel.

Mesures

Les résultats varient très sensiblement d'une architecture à l'autre, d'une gamme de processeur à une autre, d'une carte mère à l'autre, et les valeurs mentionnées ci-après n'ont qu'un intérêt relatif, en les comparant les unes avec les autres.

Outre la fréquence du processeur et la capacité mémoire, qui influent sur les performances (rapidité d'exécution du code), de nombreux autres facteurs entrent en jeu, notamment le type de bus (PCI, PCI express, CAN, etc.) et les contrôleurs d'interruptions.

Les résultats du tableau suivant ont été relevés sur un PC d'entrée de gamme, un poste de travail bureautique. La première série de mesures a été réalisée avec une charge système (processus) soutenue mais une charge en interruption moyenne. Nous y voyons les fluctuations relevées sur une tâche périodique programmée à 1 kHz, les durées étant mesurées en microsecondes.

Copyright © 2015 Eyrolles.

Système	Période minimale	Période maximale	Période moyenne
Linux vanilla (temps partagé)	346	4234	1001
Linux vanilla (temps réel)	951	1048	999
PREEMPT_RT	948	1050	999
Xenomai	942	1057	1000

Naturellement, les processus en temps partagé sont fortement pénalisés par l'activité du système et ne sauraient nous convenir.

En ce qui concerne les tâches temps réel, les résultats sont comparables entre Linux *vanilla* et les extensions temps réel. On peut même voir une légère diminution de performances pour les pires des cas lorsqu'on utilise PREEMPT_RT ou Xenomai. Ces différences ne sont toutefois pas très représentatives.

Pour mesurer des performances temps réel, il est indispensable de maintenir le système sous une haute pression en interruption (tableau ci-dessous). Les mesures ont duré plusieurs heures, sous une charge élevée d'interruption (disque, réseau, USB, etc.).

Système	Période minimale	Période maximale	Période moyenne
Linux vanilla (temps réel)	201	8403	1000
PREEMPT_RT	319	2677	1000
Xenomai	937	1058	1000

Linux vanilla, et dans une moindre mesure PREEMPT_RT, peuvent être sujets à des fluctuations suffisamment importantes pour « rater » des déclenchements du timer. Xenomai conserve des performances constantes.

J'insiste sur le fait qu'il ne faut pas prendre ces valeurs à la lettre, mais qu'il faut absolument réaliser ses propres expérimentations, sur un système conforme à la cible visée et dans un environnement le plus proche possible de celui du fonctionnement final.

Perspectives

Le domaine du temps réel sous Linux est très actif. De nombreux acteurs, notamment industriels, s'intéressent à cet environnement pour des applications de plus en plus variées (commandes de processus, communication, vidéo, mesures et instrumentations, etc.).

Les améliorations à attendre du noyau standard sont des incorporations lentes des fonctionnalités offertes par le projet PREEMPT_RT. Pourquoi lentes ? Tout simplement parce que la plupart du temps, les progrès obtenus sur les performances dans « le pire des cas » se soldent par une légère dégradation des performances moyennes. Or, la plupart des utilisateurs de Linux ne sont – à juste titre – intéressés que par ces performances moyennes.

Après un petit creux d'activité, le projet PREEMPT_RT a repris un certain dynamisme depuis la sortie du kernel 3.0 (au moment de la rédaction de ces lignes, le dernier patch disponible

Copyright © 2015 Eyrolles.

s'applique au noyau 4.1). On peut en attendre des améliorations constantes, en espérant que les retours d'information des utilisateurs (industriels, par exemple) augmenteront afin d'aider les développeurs à progresser.

La prochaine version de Xenomai (3.0), actuellement en cours de finalisation et déjà disponible en version de test, semble très prometteuse. L'idée principale est de pouvoir utiliser l'API de Xenomai et ses skins (notamment RTDM pour écrire des drivers temps réel) tant sur un système à double kernel (Linux + Adeos) que sur le noyau PREEMPT_RT seul.

Ce rapprochement des deux principales extensions temps réel de Linux est très intéressant pour les utilisateurs désireux d'importer du code existant sur d'autres environnements, tout en bénéficiant de la facilité d'installation de PREEMPT_RT (disponible directement dans certaines distributions).

Pour terminer, j'insisterai sur l'importance pour la validation d'un système, d'une configuration soigneuse du noyau avant sa compilation, de l'exécution de tests prolongés dans des conditions de charge maximale et d'une analyse rigoureuse des résultats obtenus. Ces opérations, laborieuses mais indispensables, seront la garantie de la bonne tenue en charge de vos solutions temps réel sous Linux.

Copyright © 2015 Eyrolles.

A

Compilation d'un noyau

Nous avons parlé à plusieurs reprises de compiler un noyau Linux, voici quelques conseils et exemples de configuration.

Préparation des sources

Les sources du noyau standard (vanilla) se trouvent sur le site http://www.kernel.org/, dans différents répertoires en fonction de la version désirée. Pour télécharger le noyau Linux-4.2, par exemple, on se rendra à l'adresse suivante : https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.2.tar.xz.

On commence par décompresser l'archive obtenue dans un répertoire de travail :

```
$ tar -xJf ~/linux-4.2.tar.bz2
$ cd linux-4.2/
$
```

Une autre possibilité consiste à télécharger l'ensemble des sources de Linux depuis la version 2.6.11 et à extraire celle qui nous intéresse. On utilisera l'outil de contrôle de version Git.

```
$ git   clone  git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
  [...]
$ cd linux/
```

Copyright © 2015 Eyrolles.

Cette opération peut prendre un certain temps car il est nécessaire de télécharger environ un 1 Go. Il faut ensuite choisir la version désirée (par exemple, 4.2) en examinant celles qui sont disponibles :

```
$ git  tag -l
[...]
v4.2
v4.2-rc1
v4.2-rc2
v4.2-rc3
v4.2-rc4
v4.2-rc5
v4.2-rc6
v4.2-rc7
$ git checkout v4.2
[...]
```

Il est possible à ce moment d'appliquer un patch comme PREEMPT_RT (comme nous l'avons vu au chapitre 8) ou celui de Xenomai (étudié au chapitre 9).

Le répertoire des sources contient un ensemble de sous-répertoires qui deviennent vite familiers après quelques compilations.

```
$ ls
arch           drivers   Kbuild       mm              security
block          firmware  Kconfig      net             sound
COPYING        fs        kernel       README          tools
CREDITS        include   lib          REPORTING-BUGS  usr
crypto         init      MAINTAINERS  samples         virt
Documentation  ipc       Makefile     scripts
$
```

Le Raspberry Pi, que nous avons utilisé à maintes reprises dans les chapitres précédents, n'est pas supporté par le noyau mainline au moment de la rédaction de ces lignes. Il faut donc télécharger les sources d'un noyau légèrement modifié à partir d'un dépôt particulier :

```
$ git  clone  https://github.com/raspberrypi/linux rpi-linux
  [...]
$ cd  rpi-linux
$ git  checkout  rpi-4.1.y
```

Copyright © 2015 Eyrolles.

Configuration de la compilation

Principes

La première étape consiste à configurer la compilation, c'est-à-dire à indiquer toutes les fonctionnalités à intégrer dans le noyau voulu. L'un des principaux avantages de Linux est d'être capable de fonctionner sur un très grand nombre d'architectures et de machines différentes, et d'être adapté à des besoins spécifiques très variés. Il existe environ 15 000 options de configuration pour le noyau 4.2 !

Il est naturellement impossible de fournir toutes ces options sur une ligne de commandes d'un script `./configure`, comme on le fait souvent pour les projets GNU. La compilation du noyau Linux s'appuie sur un fichier nommé `.config` qui va contenir toutes les options choisies.

Ce fichier (qui n'est visible qu'avec l'option `-a` de `ls` car son nom commence par un point) est constitué de texte lisible, mais sa production et sa lecture sont automatisées au sein de `Makefile`. Lorsqu'on vient de décompresser un noyau vierge, il n'y a aucun fichier `.config`, toutefois certains fichiers génériques existent. Il faut employer la commande `make help` pour obtenir la liste des configurations disponibles.

```
$ ls  -a  .config
ls: impossible d'accéder à .config: Aucun fichier ou dossier de ce type
$ make  help
Cleaning targets:
clean     Remove most generated files but keep the config and
          enough build support to build external modules
mrproper  Remove all generated files+config+various backup files
[...]
i386_defconfig      Build for i386
x86_64_defconfig    Build for x86_64
[...]
$
```

Ici, les deux configurations par défaut sont pour des architectures x86 32 et 64 bits, car il s'agit de la plate-forme sur laquelle la commande `make` a été exécutée.

Nous pouvons préciser une architecture différente avec l'option `ARCH=` de `make`. Nous considérerons ci-après le cas d'une compilation croisée pour une cible ARM.

> Attention à bien utiliser l'option `ARCH=` sur toutes les commandes `make` que nous emploierons par la suite. Il faut également être attentif à la casse : `ARCH` est en majuscules et l'architecture proprement dite en minuscules.

Copyright © 2015 Eyrolles.

Les architectures disponibles sont visibles dans le sous-répertoire `arch/`. Voici, par exemple, quelques configurations par défaut pour plates-formes Arm (il y en a 113 sur cette version de Linux) :

```
$ make ARCH=arm help
  [...]
  nuc910_defconfig          - Build for nuc910
  nuc950_defconfig          - Build for nuc950
  nuc960_defconfig          - Build for nuc960
  omap1_defconfig           - Build for omap1
  omap2plus_defconfig       - Build for omap2plus
  orion5x_defconfig         - Build for orion5x
  palmz72_defconfig         - Build for palmz72
  pcm027_defconfig          - Build for pcm027
  pleb_defconfig            - Build for pleb
  prima2_defconfig          - Build for prima2
[...]
$
```

Il est donc possible de générer un fichier de configuration de départ facilement :

```
$ make  ARCH=arm  omap2plus_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
# configuration written to .config
$
```

Pour le Raspberry Pi, la configuration du noyau spécifique s'obtient avec :

```
$ make  ARCH=arm  bcmrpi_defconfig
```

Copyright © 2015 Eyrolles.

Interfaces utilisateur

Ensuite, il faudra affiner les options de configuration. Pour cela, il existe quatre interfaces utilisateur qui vont éditer ce fichier `.config`, en le lisant au démarrage et en le réécrivant en sortie :

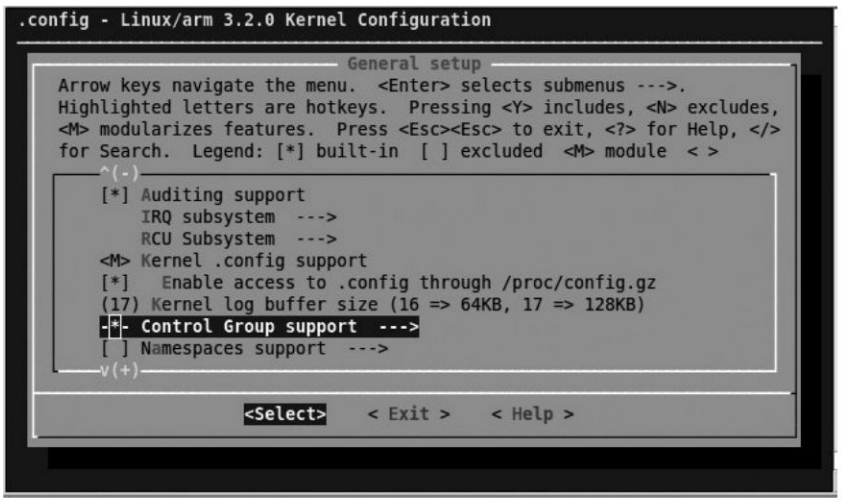
- `make ARCH=arm nconfig` : configuration utilisant une interface minimale sur console texte basée sur la bibliothèque nCurses ;
- `make ARCH=arm menuconfig` : assez proche de la configuration précédente, avec des menus en mode semi-graphique ;
- `make ARCH=arm gconfig` : interface graphique s'appuyant sur GTK+ ;
- `make ARCH=arm xconfig` : interface utilisant la bibliothèque graphique Qt.

On peut choisir n'importe quelle méthode, elles offrent les mêmes possibilités. Les deux premières permettent une navigation au clavier plus rapide qu'à la souris lorsque l'on connaît l'organisation des menus de configuration. Les deux autres offrent une vision plus globale de la configuration par des menus déroulants et une ergonomie améliorée, notamment par l'affichage permanent de l'aide concernant l'option sélectionnée.

Prenons l'exemple de `menuconfig`, choix le plus fréquent parmi les hackers de Linux.

Figure A-1

Make menuconfig



Chaque option peut être activée ou non (en appuyant sur la barre d'espace du clavier). Par exemple, sur la figure A-1, l'option Auditing support est activée et Namespace support est désactivée.

En outre, certaines options peuvent être compilées sous forme de modules comme c'est le cas ici pour Kernel .config support. Le code sera alors compilé sous forme de fichier objet indépendant du noyau, que l'on pourra insérer et retirer dynamiquement.

La configuration est une étape longue et un peu fastidieuse, d'autant qu'il est généralement nécessaire de la répéter plusieurs fois, en ajustant les paramètres après test du système.

Copyright © 2015 Eyrolles.

Options de compilation

Voici quelques options de compilation concernant le comportement temps réel. Certaines options dépendent d'autres sélections et ne sont pas toujours visibles. En outre, leurs noms peuvent varier légèrement en fonction de l'architecture choisie.

Menu General Setup

- System V IPC et Posix Message Queues : ces deux options concernent des mécanismes importants de communication entre processus qui sont généralement indispensables. Activées.
- Auditing support : ajoute du code d'instrumentation. Désactivée.
- Sous-menu Control Group Support : la plupart des options permettent de gérer finement le partage des ressources disponibles entre groupes de processus. Activées.
- Automatic process group scheduling : permet la répartition automatique des processus issus du même terminal dans un seul groupe pour un ordonnancement temps partagé (chapitre 3). Activée.
- Optimize for size : généralement utile sur les systèmes embarqués où la taille mémoire est limitée. Activée.
- Enable futex support : propose des mutex rapides et efficaces (chapitre 7). Activée.
- Embedded System : cette option donne accès à des paramètres avancés (dans le menu Configure standard kernel features) pour ajuster la compilation sur système embarqué. Activée.
- Profiling support, Kprobes, GCOV-based kernel profiling : toutes ces options ajoutent du code d'instrumentation dans le noyau. Désactivées.

Menu Processor type and feature

Ce menu varie suivant les architectures : il est parfois scindé en deux menus System type et Kernel features, parfois en Platform support et Kernel options. On y retrouve généralement les options suivantes.

- High Resolution Timer support : cette option s'appuie sur le matériel pour proposer des timers à plus de 1 kHz. Elle n'est pas disponible sur toutes les architectures mais quand elle est présente, il faut en profiter. Activée.
- Timer frequency : ce paramètre influe sur la granularité des timers logiciels de basse fréquence (voir `setitimer()` au chapitre 4). On utilise souvent une fréquence de tick à 300 Hz (pour les applications vidéo, par exemple, car cela offre des timers logiciels de bonne qualité à 50 Hz ou 60 Hz). Ce choix n'influe pas sur les timers de haute fréquence précédents.
- Tickless System : option qui demande réflexion. Le fait de conserver un tick régulier (à la fréquence indiquée précédemment) oblige le processeur à rester actif même lorsque le système est au repos, ce qui consomme de l'énergie inutilement. On aurait donc tendance sur un système embarqué autonome à activer l'option Tickless – qui élimine les ticks et économise la batterie. Toutefois, pour des systèmes temps réel, l'endormissement du processeur dans un état approfondi est pénalisant, notamment lors du réveil sur interruption.

Copyright © 2015 Eyrolles.

J'aurais donc tendance à désactiver cette option pour améliorer les performances temps réel, tout en sachant qu'elle a un coût énergétique non négligeable nécessitant de l'activer dans le cas d'un système embarqué sur batterie.

- Preemption Model : on choisira ici la proposition la plus préemptible, à savoir Preemptible Kernel pour un noyau vanilla ou Fully Preemptible Kernel après application du patch PREEMPT_RT.

Menu Power management

Les options de ce menu sont rarement compatibles avec le temps réel. J'ai généralement tendance à toutes les désactiver, surtout les variations de fréquence CPU.

Certaines options de ce menu sont également rédhibitoires pour Xenomai.

Autres menus

Les choix de configuration dans les autres menus dépendent du système pour lequel on réalise la compilation (notamment du matériel présent pour le choix des drivers) et des applications désirées (pour les protocoles réseau, par exemple).

Je conseille toutefois de désactiver les options des menus Kernel hacking et Virtualization car elles impliquent souvent une surcharge du code compilé.

Compilation et installation

Compilation croisée

La compilation pour une cible différente se fait en utilisant une chaîne de compilation croisée, c'est-à-dire un ensemble d'outils qui fonctionnent sur la machine de développement et fournissent du code prêt à s'exécuter sur la machine cible. En voici un aperçu :

```
$ ls /opt/arm-linux/usr/bin/
arm-linux-addr2line
arm-linux-ar
arm-linux-as
arm-linux-c++
[...]
arm-linux-gcc
arm-linux-gcc-4.3.6
arm-linux-gcov
arm-linux-gdb
arm-linux-gprof
arm-linux-ld
[...]
$
```

Copyright © 2015 Eyrolles.

Sur ce poste, la chaîne de compilation se trouve dans `/opt/arm-linux/usr/bin`. Tous les utilitaires GNU classiques sont préfixés par `arm-linux-` pour éviter les confusions.

La compilation du noyau Linux se fera ainsi :

```
$ make ARCH=arm CROSS_COMPILE=/opt/arm-linux/usr/bin/arm-linux-
[...]
```

Il est indispensable d'indiquer l'architecture cible. Par ailleurs, on précise dans la variable `CROSS_COMPILE` un préfixe qui sera ajouté par `make` avant le nom de chaque utilitaire invoqué (par exemple, `gcc`). Ce préfixe doit donc se terminer par un tiret (afin d'obtenir, par exemple, `.../arm-linux-gcc`).

On pourra ajouter d'autres éléments sur la ligne de commandes `make`.

- L'option `-j` *n* permet de compiler en lançant *n* jobs en parallèle en permanence. Ceci réduit le temps de compilation si l'on dispose de *n* CPU (ou *n* cœurs).
- Le type d'image désirée peut être indiqué en fin de ligne, par exemple, `uImage` si la plate-forme cible utilise le *bootloader* U-Boot.

La durée de la compilation varie en fonction du processeur et du contenu (le nombre d'options choisies lors de la configuration), de quelques dizaines de secondes à plus d'une heure !

En fin de compilation, on obtient une image dans le répertoire `arch/<architecture>/boot` qu'il faudra transférer sur la plate-forme cible. De plus, si des modules ont été compilés, on les regroupera dans un répertoire (par exemple, `tmp_mod`) dont on transférera le contenu à la racine du système de fichiers de la cible :

```
$ make  INSTALL_MOD_PATH=tmp_mod/  modules_install
```

Il faudra bien entendu configurer le *bootloader* pour qu'il sélectionne l'image produite.

Compilation native

Lors d'une compilation native, c'est-à-dire sur la plate-forme même où le noyau sera installé, on se contente de lancer :

```
$ make
```

On installera ensuite le noyau compilé et les modules avec les commandes :

```
$ make modules_install
```

Copyright © 2015 Eyrolles.

puis :

```
$ make install
```

Cette dernière commande copie le nouveau noyau dans `/boot/`, recherche les fichiers de configuration de Grub et ajoute une entrée pour la nouvelle image (en laissant les précédentes accessibles au cas où notre nouveau noyau refuserait de démarrer).

Toutefois, sur les distributions Debian et Ubuntu, on procède habituellement d'une manière légèrement différente en générant tout d'abord un package contenant le nouveau noyau, qu'on installe ensuite avec les outils d'administration système standards :

```
$ make-kpkg --initrd buildpackage
[...]
$ cd ..
$ dpkg -i *deb
[...]
```

Il existe de nombreuses documentations complémentaires sur Internet, sans oublier le répertoire `Documentation/` des sources du noyau qui contient des informations très intéressantes.

Copyright © 2015 Eyrolles.

Copyright © 2015 Eyrolles.

B

Bibliographie

Sur le blog de l'auteur, à l'adresse http://www.blaess.fr/christophe/articles, on trouvera, entre autres, les articles suivants :

- 7 mai 2012 : *Mesure de précision des timers RTDM*
- 10 mai 2012 : *GPIO, Pandaboard et temps réel*
- 27 août 2012 : *Xenomai sur Raspberry Pi*
- 26 novembre 2012 : *Les GPIO du Raspberry Pi*
- 15 février 2013 : *Raspberry Pi : interruptions GPIO avec RTDM*
- 28 décembre 2011 : *Xenomai 2.6.0 sur Ubuntu 11.10*
- 13 mars 2013 : *Mesures de latence d'interruptions à l'aide d'un STM32.*
- etc.

Livres

- [BLAESS 2011] *Développement système sous Linux, 3e édition* de Christophe Blaess, éditions Eyrolles (2011).
- [BONNET 1999] *Introduction aux systèmes temps réel* de Christian Bonnet et Isabelle Demeure, Hermes Science (1999).
- [BOVET 2006] *Understanding the Linux Kernel, 3rd Edition* de Daniel P. Bovet et Marco Cesati, O'Reilly (2006).
- [CORBET 2005] *Linux Device Drivers, 3rd Edition* de Jonathan Corbet, Alessandro Rubini et Greg Kroah-Hartman, O'Reilly (2005).

Copyright © 2015 Eyrolles.

- [COTTET 2000] *Ordonnancement temps réel, cours et exercices corrigés* de Francis Cottet, Joëlle Delacroix, Claude Kaiser et Zoubir Mammeri, Hermes Science (2000).
- [DORSEUIL 1991] *Le temps réel en milieu industriel* de Alain Dorseuil et Pascal Pillot, Dunod (1991).
- [FICHEUX 2011] *Linux embarqué* de Pierre Ficheux et Eric Bénard, Eyrolles (2012).
- [HOLLABAUGH 2002] *Embedded Linux, Hardware, Software, and Interfacing* de Craig Hollabaugh, Addison Wesley (2002).
- [KERRISK 2010] *The Linux Programming Interface, A Linux and UNIX System Programming Handbook* de Michael Kerrisk, No Starch Press (2010).
- [LOVE 2004] *Linux Kernel Development* de Robert Love, Developer's Library (2004).
- [TANENBAUM 1987] *Operating Systems, Design and Implementation* de Andrew S. Tannenbaum et Albert S. Woodhull, Prentice Hall (1987).
- [TORVALDS 2001] *Il était une fois Linux* de Linus Torvalds et David Diamond, OEM (2001).
- [YAGHMOUR 2003] *Building Embedded Linux Systems* de Karim Yaghmour, O'Reilly (2003).

Articles

- [BROWN 2010] *How Fast Is Fast Enough? Choosing Between Xenomai and Linux for Real-Time Applications. In Twelfth OSADL Real-Time Linux Workshop* de Dr. Jeremy H. Brown et Brad Martin, octobre 2010. http://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf.
- [GERUM 2005] *Life with Adeos* de Philippe Gerum. http://www.xenomai.org/documentation/branches/v2.3.x/pdf/Life-with-Adeos-rev-B.pdf
- [KISZKA 2005] *The Real-Time Driver Model and First Application* de Jan Kiszka. http://svn.gna.org/svn/xenomai/tags/v2.4.0/doc/nodist/pdf/RTDM-and-Applications.pdf
- [REEVES 1997] *What Really Happened on Mars?* de Glenn E. Reeves. http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/ Authoritative_Account.html
- [YAGHMOUR 2001] *Adaptative Domain Environment for Operating Systems* de Karim Yaghmour. http://www.opersys.com/ftp/pub/Adeos/adeos.pdf
- [YODAIKEN 2004] *Against Priority Inheritance* de Victor Yodaiken. http://www.yodaiken.com/papers/inherit.pdf

Sites web

http://christophe.blaess.fr : site de l'auteur, codes sources des programmes du livre, articles…

http://www.kernel.org : *The Linux Kernel Archives*. Le dépôt officiel des sources du noyau Linux (et de PREEMPT_RT).

http://rt.wiki.kernel.org : *Real-Time Linux Wiki*. Le site de PREEMPT_RT.

http://www.xenomai.org : site officiel de Xenomai (documentation, téléchargements, etc.).

Copyright © 2015 Eyrolles.

http://ltp.sourceforge.net : *Linux Test Project*. De très nombreux outils de tests, pas seulement dans le domaine temps réel.

http://www.osadl.org : *Open Source Automation Development Lab*. De nombreux projets et articles liés à Linux temps réel.

http://www.rtai.org : *Real Time Application Interface*. Le projet RTAI, initialement proche de Xenomai, mais moins actif de nos jours.

Copyright © 2015 Eyrolles.

Copyright © 2015 Eyrolles.

Index

Symboles

$$ (variable) 111
__exit 248
__init 248
/proc 28
/proc/config.gz 135
/proc/cpuinfo 8, 9
/proc/interrupts 28, 210, 258
/proc/ipipe 206
/proc/irq/ 29
/proc/sys/kernel/sched_rt_period_us 100
/proc/sys/kernel/sched_rt_runtime_us 100
/proc/xenomai 206, 207
/proc/xenomai/irq 274
_SC_NPROCESSORS_ONLN 10
/sys/class/ 251, 254
/sys/class/gpio/ 271
/usr/xenomai 206, 208

A

Adeos 196, 197, 199-201, 203, 206, 207
Affinité 13, 14, 49, 207
alarm() 164
alloc_chrdev_region() 250, 251, 256
APIC 24, 28
APM 203
Appel système 2, 18, 24, 34, 42, 46, 253, 265
Arm 60

B

Barabanov, Michael 194, 195
Barrière 149
BFS 51
Busybox 110

C

cdev_add() 250, 252, 256
cdev_del() 251, 252
cdev_init() 250, 252, 256
CFS 46, 47
chrt 110, 119, 141, 264
class_create() 250, 251, 256
class_destroy() 251
clock_getres() 61
clock_gettime() 61
clockid_t 61, 66
CLOCK_MONOTONIC 61, 62
CLOCK_MONOTONIC_RAW 61, 62
CLOCK_PROCESS_CPUTIME_ID 61
CLOCK_REALTIME 61, 62
clock_settime() 61
CLOCK_THREAD_CPUTIME_ID 61
Commutation 1, 241
Complexité 45, 46, 47
Conservative 187
Consommation 26, 69, 186
copy_from_user() 253, 254
copy_to_user() 252, 254, 259
core 31
CPU 31
CPU_CLR() 13
CPU_ISSET() 13
CPU_SET() 13
cpu_set_t 13
CPU_ZERO() 13
Cyclictest 184, 210

Copyright © 2015 Eyrolles.

D

DECLARE_WAIT_QUEUE_HEAD() 256
DEFINE_MUTEX() 252
device_create() 250, 251, 256
device_destroy() 251
dev_t 250
dmesg 249
dohell 209, 210, 231
Down 18
Driver 91, 249

E

errno 218
Ethernet 175
Exception 2, 24, 30, 31, 130, 131
exec() 3
execl() 3
execle() 3
execlp() 3, 5
execv() 3
execve() 3, 130
execvp() 3
exit() 3

F

Fifo 95, 106, 172, 204
Fluctuation 120
fork() 3, 131
fprintf() 221
FPU 31
free_irq() 257, 258
Futex 158

G

Galbraith, Mike 48
GDB 33
Gerum, Philippe 197, 198
getnstimeofday() 261, 263
getpriority() 53
gettimeofday() 48, 58, 62, 80
GkrellM 10, 100
Gleixner, Thomas 178, 179, 183, 184
GlibC 34, 158
gnuplot 76
Governor 187, 188, 189, 190
GPIO 271, 274
gpio_direction_input() 271
gpio_direction_output() 271
gpio_free() 271
gpio_get_value() 271
gpio_request() 271
gpio_set_value() 271
gpio_to_irq() 271
GPL 91, 248

H

Hackbench 186, 209
handler 27, 28, 255, 258, 272
High Resolution Timers Support 123
Histogramme 74
Horloge 61
HR-Timers 70
Hwlatency 185
Hyperthreading 8

I

Igep 60
Igep v.2 60
init 37
insmod 91, 141, 247, 249
Interruption 23, 24, 26, 28, 30, 34, 46, 91, 93, 171, 174, 176, 178, 193, 195, 255, 257, 271
  Pipeline d'~ 196, 199
IRQ 24, 27, 28, 176, 197, 255
irqbalance 29
IRQF_SHARED 256, 258
IRQ_HANDLED 258, 259
IRQ_NONE 181, 258
irqreturn_t 259
IRQ_WAKE_THREAD 181, 261, 262
ITIMER_PROF 65
ITIMER_REAL 65
ITIMER_VIRTUAL 65

Copyright © 2015 Eyrolles.

J

jiffies 69
Jitter 70

K

Kiszka, Jan 215, 265
Kolivas, Con 51

L

latency 210
LD_LIBRARY_PATH 208, 219, 270
lscpu 9
LTP 210
LXRT 198

M

Makefile 218
Mantegazza, Paolo 195
Mars Pathfinder 153
MCL_CURRENT 131
MCL_FUTURE 131
Mémoire 128, 130
Migration 11
mlockall() 131, 215
mmap() 92, 261
MMU 2, 31, 129-131
Mode
  Primaire 199, 213
  Secondaire 199, 213
Module 247, 248
module_exit() 248
module_init() 248
MODULE_LICENSE() 248
module_param() 255, 257
Molnár, Ingo 45, 46, 51, 178
Multicœur 8, 28
Multiprocesseur 8, 28
munlockall() 131
Mutex 124, 150, 154, 156, 240, 253
mutex_lock() 254
mutex_lock_interruptible() 252-254
mutex_unlock() 253, 254

N

Nanokernel 193
nice 51, 52
Noyau 2, 132, 135
  Compilation 69
  Espace ~ 2, 198, 203, 247
  Mode ~ 23, 27
  Thread du ~ 38
NPTL 53, 158
Nucleus 199, 200, 203, 204, 206

O

Ondemand 187, 190
Ordonnanceur 1, 10, 19, 41, 43-47, 94, 193, 204

P

patch 180
PCI 28
Performance 187, 189
Ping 172, 174, 181, 210
PIP 154
Pi_stress 186
PLC 26
Posix 1, 5, 14, 61, 103, 149, 201, 204
posix_spawn() 3
Powersave 187, 190
Prédictibilité 87
Préemptibilité 132, 134, 180, 183
Préemption 20, 43, 47, 79, 177
Preemption Model 134
PREEMPT_RT 144, 178, 179, 181, 182, 185
printk() 248
Priorité
  Héritage de ~ 154, 243
  Inversion de ~ 150, 178, 243
  Temps partagé 51, 53
  Temps réel 94
Processeur 24, 25, 27, 49
  Arm 60
  Sommeil du ~ 27
Processus 1, 17, 31, 94, 98, 127, 128
  Groupe de ~ 47

Copyright © 2015 Eyrolles.

ps 17, 38, 210
PSOS 201, 204
pthread_attr_destroy() 102
pthread_attr_getaffinity_np() 15
pthread_attr_getinheritsched() 102
pthread_attr_getschedparam() 101
pthread_attr_getschedpolicy() 102
pthread_attr_getscope() 103
pthread_attr_init() 102
pthread_attr_setaffinity_np() 15
pthread_attr_setinheritsched() 103
pthread_attr_setschedparam() 102
pthread_attr_setschedpolicy() 102, 107
pthread_attr_setscope() 103
pthread_attr_t 5, 15
pthread_atttr_setschedpolicy() 102
pthread_barrier_init() 150
pthread_barrier_wait() 150
pthread_create() 5, 15, 102
pthread_exit() 5
PTHREAD_EXPLICIT_SCHED 103
pthread_getaffinity_np() 14
pthread_getschedparam() 101
PTHREAD_INHERIT_SCHED 103
pthread_join() 6
pthread_mutexattr_init() 155
pthread_mutexattr_setprotocol() 155
pthread_mutex_lock() 152, 158, 162
pthread_mutex_trylock() 158
pthread_mutex_unlock() 158, 162
PTHREAD_PRIO_INHERIT 155
PTHREAD_SCOPE_PROCESS 103
PTHREAD_SCOPE_SYSTEM 103
pthread_setaffinity_np() 14
pthread_setschedparam() 101
pthread_t 5

R

Raspberry Pi 136
read() 253
request_irq() 256, 258
request_threaded_irq() 261, 262
Ritchie, Dennis 41
rmmod 141, 247, 249
root 61, 100, 189
Round Robin 95, 106, 147, 149, 163, 204
RS-232 136
RSDL 51
RTAI 195, 196, 198
RT_ALARM 232
rt_alarm_create() 232, 234
rt_alarm_delete() 232
rt_alarm_start() 232
rt_alarm_stop() 232
rt_alarm_wait() 233
rt_dev_close() 270
rt_dev_open() 270
rt_dev_read() 270
rt_dev_write() 270
RTDK 215
RTDM 205, 265, 269, 271
rtdm_dev_register() 267
rtdm_dev_unregister() 267
rtdm_irq_disable() 273
rtdm_irq_enable() 273
rtdm_irq_free() 273
rtdm_irq_request() 273
rtdm_irq_t 273
rtdm_mutex 267, 269
rtdm_mutex_destroy() 267
rtdm_mutex_init() 267
rtdm_mutex_lock() 267
rtdm_mutex_t 266
rtdm_mutex_unlock() 267
rtdm_safe_copy_from_user() 268
rtdm_safe_copy_to_user() 267
rtdm_user_info_t 266, 267
rt_fprintf() 215, 222
RT_INTR 265
rt_intr_enable() 265
rt_intr_wait() 265
RTLinux 178, 194, 196
RT_MUTEX 240
rt_mutex_acquire() 240, 242
rt_mutex_create() 240, 242
rt_mutex_delete() 240
rt_mutex_release() 240, 241
rt_print_auto_init() 215
rt_printf() 215

Copyright © 2015 Eyrolles.

rt_print_init() 215
rtps 219
rt_puts() 215
RT_SEM 238
rt_sem_create() 238, 240
rt_sem_delete() 238
rt_sem_p() 238
rt_sem_v() 238
rt_syslog() 215
RT_TASK 216
rt_task_create() 216
rt_task_inquire() 233
rt_task_join() 217
rt_task_self() 223
rt_task_set_periodic() 223
rt_task_shadow() 220
rt_task_sleep() 217
rt_task_spawn() 216
rt_task_start() 216
rt_task_wait_period() 224, 226-228
rt_timer_read() 224
rt_vfprintf() 215
rt_vprintf() 215
rt_vsyslog() 215
Runnable 19, 41, 44, 45, 94, 133
Running 18, 19, 42, 133

S

SCHED_BATCH 96
SCHED_FIFO 96, 109
sched_getaffinity() 13
sched_getcpu() 11
sched_getparam() 96
sched_get_priority_max() 97
sched_get_priority_min() 97
sched_getscheduler() 96
SCHED_IDLE 96
SCHED_OTHER 96
sched_priority 96
SCHED_RR 96, 107, 109
sched_rr_get_interval() 109, 112
sched_setaffinity() 13
sched_setparam() 96
sched_setscheduler() 96
schedule() 44, 45
Scheduler 1, 18, 41, 47
sched_yield() 109, 167
Scrutation 25
Sémaphore 237
setitimer() 64
setpriority() 53
Shell 3, 32, 49, 52, 110
SIGALRM 65, 98
SIGCONT 18
SIGEV_NONE 66
sigev_notify 66
SIGEV_SIGNAL 66
SIGEV_THREAD 66
SIGFPE 31
SIGILL 31
SIGKILL 18
Signal 24, 31
SIGPROF 65
SIGSEGV 31, 32
SIGSTOP 18
SIGVTALRM 65
SIGXCPU 221
sleep() 157
Sleeping 18
SMI 186
SMP 8
smp_affinity 29
spin_lock() 257, 259
Spinlock 144, 178, 257
spin_lock_init() 256, 257
spin_lock_irqsave() 257, 258
spinlock_t 256, 257
spin_trylock() 257
spin_unlock() 257, 259
spin_unlock_irqrestore() 258
spin_unlock_irqsrestore() 257
struct cdev 250, 255
struct class 250, 255
struct file_operations 250, 255
struct itimerspec 67
struct itimerval 64
struct rtdm_dev_context 266, 267
struct rtdm_device 266, 267
struct sched_param 96

Copyright © 2015 Eyrolles.

struct sigevent 66
struct timeval 58
Swap 130
sysconf() 10
SYSENTER 37
SYSEXIT 37
system() 3

T

Tasklet 176, 181
taskset 12, 110, 119, 141
Temps partagé 149
Temps réel 44
  Définition 85
  Souple 88
  Strict 87
THIS_MODULE 248
Thomson, Ken 41
Thread 1, 5, 14, 17, 18, 53, 101, 123, 128, 147, 150, 213
  Commutation de ~ 42
thread_barrier_t 149
Threaded Interrupt 178, 181, 182, 261
Tick 44, 69, 204
tickless 70
time() 48, 57
Timer 43, 46, 69, 204, 224
  Fluctuation 120
  Granularité 68, 122
  Posix 66
  Temps réel 119
  Unix 64
timer_create() 66
timer_settime() 67
Timeslice 43
time_t 57
Torvalds, Linus 48, 179
Traced 18
Tranche de temps 43, 46, 69

U

uClibC 34
uITRON 201, 204
ulimit 32
uname 135
Unix 41, 52, 64
unregister_chrdev_region() 251, 252
uptime 186
Userspace 187
usleep() 157, 165
Utilisateur
  Espace ~ 2, 93, 198, 203, 247
  Mode ~ 23, 27

V

Vaddagiri, Srivatsa 47
Virtual Runtime 46
VRTX 200, 204
VxWorks 200, 204

W

wait_event_interruptible() 258, 260
waitpid() 3
Waitqueue 258, 259
wake_up_interruptible() 259, 260
Watchdog 232, 235
Williams, Clark 184
Workqueue 176, 181
write() 34, 35, 253

X

xeno-config 218
Xenomai 198, 199, 201-208, 213, 265
  Skin 200, 207

Y

Yaghmour, Karim 196
Yodaiken, Victor 156, 194, 195

Z

Zombie 18