

Christophe Blaess

Solutions temps réel
sous **LINUX**

Cas pratique : le Raspberry Pi 3

3^e édition



● Éditions
EYROLLES

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 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, notamment avec la nouvelle version de Xenomai, la troisième édition a pour support d'expérimentation le nano-ordinateur Raspberry Pi 3.

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

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 • **Ordonnement temps partagé et priorités** • Temps partagé • Configuration des priorités • **Limitations de l'ordonnement temps partagé** • Mesure du temps • Tâches périodiques • Prémption 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 • Prémptibilité 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.

Expert reconnu de Linux dans l'industrie, **Christophe Blaess** conçoit et met en œuvre des systèmes embarqués industriels. Il 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...), coorganise 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 www.blaess.fr/christophe

- Téléchargez le code source des exemples
- Consultez les corrigés des exercices et de nombreux documents complémentaires
- Dialoguez avec l'auteur

Solutions temps réel sous LINUX

Dans la collection « Les guides de formation Tsoft »

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

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

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

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

Autres ouvrages sur Linux

P. Ficheux. – **Linux embarqué. Mise en place et développement.**
N° 67484, 2017, 220 pages.

K. Novak. – **Débuter avec Linux. Maîtrisez votre système aux petits oignons.**
N° 13793, 2017, 522 pages.

K. Novak. – **Administration Linux par la pratique. Les fondamentaux de l'administration système.**
N° 67738, 2019, 504 pages.

C. Blaess. – **Développement système sous Linux. Ordonnancement multitâches, gestion mémoire, communications, programmation réseau. À paraître.**
N° 67760, 5^e édition, 2019, 1068 pages.

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

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

Christophe Blaess

Solutions temps réel sous LINUX

3^e édition

● Éditions
EYROLLES

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

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.

ISBN : 978-2-212-67711-9

© Groupe Eyrolles, 2012, 2016
© Éditions Eyrolles, 2019 pour la présente édition

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 plateformes 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.

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 3 : 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.

Table des matières

Avant-propos	V
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	11
États des tâches	18
Ordonnancement	20
Préemption	21
Conclusion	21
Points clés	22
Exercices	22
Exercice 1 (*)	22
Exercice 2 (**)	22
Exercice 3 (***)	23
Exercice 4 (****)	23
CHAPITRE 2	
Interruptions, exceptions et appels système	25
Mode noyau	25
Interruptions	26
Principe	26

Entrées-sorties sans interruptions	27
Entrées-sorties avec interruptions	28
Interruptions sous Linux	29
Routage des interruptions.	30
Exceptions	32
Principe	32
Fichier core.	33
Appels système	36
Suivi d'un appel système	36
Threads du noyau	40
Conclusion	41
Points clés	41
Exercices	42
Exercice 1 (*)	42
Exercice 2 (*)	42
Exercice 3 (**).	42
Exercice 4 (**).	42
Exercice 5 (***)	42

CHAPITRE 3

Ordonnancement temps partagé et priorités	43
Temps partagé	43
Principes.	43
Ordonnanceur historique	46
Ordonnanceurs du noyau 2.6	47
Ordonnanceur CFS.	48
Groupes de processus.	49
Autres ordonnanceurs.	53
Configuration des priorités	53
Courtoisie des processus	53
Priorités entre threads	55
Conclusion	57
Points clés	57
Exercices	57
Exercice 1 (*)	57

Exercice 2 (*)	58
Exercice 3 (**)	58
Exercice 4 (***)	58
Exercice 5 (***)	58
CHAPITRE 4	
Limitations de l'ordonnancement temps partagé	59
Mesure du temps	59
Heure Unix avec gettimeofday()	60
Précision des mesures	60
Horloges Posix	62
Tâches périodiques	65
Timers Unix classiques	66
Timers Posix	67
Granularité	70
Précision	72
Préemption des tâches	81
Conclusion	84
Points clés	84
Exercices	84
Exercice 1 (*)	84
Exercice 2 (**)	84
Exercice 3 (**)	85
Exercice 4 (***)	85
Exercice 5 (***)	85
CHAPITRE 5	
Principes du temps réel	87
Définitions	87
Temps réel	87
Classes de temps réel	88
Temps réel absolu	88
Temps réel strict	89
Temps réel strict certifiable	89

Temps réel strict non certifiable	90
Temps réel souple	90
Rôles respectifs	92
Traitement direct dans le noyau	92
Traitement des interruptions	95
Temps réel sous Linux	96
Échelle des priorités	96
Configuration de l'ordonnancement	98
Processus temps réel	100
Garde-fou temps réel	102
Threads temps réel	103
Threads en Round Robin	107
Rotation sans Round Robin	111
Temps réel depuis le shell	111
Ordonnancement EDF (<i>Earliest Deadline First</i>)	113
Conclusion	119
Points clés	119
Exercices	119
Exercice 1 (*)	119
Exercice 2 (*)	120
Exercice 3 (**)	120
Exercice 4 (**)	120
Exercice 5 (***)	120

CHAPITRE 6

Performances du temps réel souple	121
Timers temps réel	121
Précisions et fluctuations	121
Granularité des timers	124
Conclusion sur les timers	124
Temps de commutation	125
Commutation entre threads	125
Commutations entre processus	129
Comparaison processus et threads	130
Imprévisibilités dues à la mémoire virtuelle	132

Préemptibilité du noyau	134
Principes	134
Préemptibilité du noyau standard	136
Connaître la configuration d'un noyau	137
Expériences sur la préemptibilité	138
Conclusion	147
Points clés	147
Exercices	148
Exercice 1 (*)	148
Exercice 2 (**)	148
Exercice 3 (**)	148
Exercice 4 (**)	148
Exercice 5 (***)	148
CHAPITRE 7	
Problèmes temps réel classiques	149
Démarrage en Round Robin	149
Barrières Posix	151
Inversion de priorité	152
Principe	152
Héritage de priorité	156
Prise de mutex	158
Comportement en temps réel	161
Reprise de mutex en temps réel	163
Solutions	166
Appel explicite à l'ordonnanceur	168
Conclusion	169
Points clés	170
Exercices	170
Exercice 1 (*)	170
Exercice 2 (**)	170
Exercice 3 (**)	170
Exercice 4 (***)	170
Exercice 5 (***)	170

CHAPITRE 8

Limites et améliorations du temps réel Linux	171
Traitement des interruptions.	171
PREEMPT_RT	178
Threaded interrupts	183
Fully preemptible kernel.	184
Conclusion sur PREEMPT_RT	185
Outils de mesure des performances	185
Cyclicttest	185
Hwlatdetect, Hackbench...	187
Économies d'énergie	187
Variation de fréquence d'horloge	188
Heuristique performance	190
Heuristique powersave	191
Heuristique ondemand	192
Conclusion	192
Points clés	193
Exercices	193
Exercice 1 (**)	193
Exercice 2 (**)	193
Exercice 3 (**)	193
Exercice 4 (***)	193
Exercice 5 (***)	194

CHAPITRE 9

Extensions temps réel de Linux	195
Les nanokernels temps réel	195
Principes.	195
RTLinux.	197
RTAI et Adeos	197
Xenomai	200
Interface de programmation.	202
Xenomai 3	203
Installation de Xenomai	203
Modification du noyau Linux	203

Configuration de Xenomai Cobalt	206
Compilation de Xenomai	208
Expériences avec Xenomai	209
Première exploration	209
Programmes de tests	211
Conclusion	213
Points clés	213
Exercices	214
Exercice 1 (*)	214
Exercice 2 (**)	214
Exercice 3 (***)	214
Exercice 4 (****)	214
CHAPITRE 10	
Programmer avec Xenomai	215
Programmation de tâches simples	215
Principes	215
Initialisation du processus	217
Création de tâche	218
Compilation et exécution	220
Processus unithread	222
Recherche des changements de modes	223
Alarmes et tâches périodiques	226
Réveils périodiques	226
Alarmes	235
Watchdog	237
Synchronisation des tâches	239
Sémaphores	239
Mutex	242
Conclusion	247
Points clés	247
Exercices	248
Exercice 1 (*)	248
Exercice 2 (***)	248

Exercice 3 (**)	248
Exercice 4 (***)	248
CHAPITRE 11	
Traitement des interruptions	249
Programmation d'un driver	249
Squelette d'un module	249
Structure d'un driver	251
Traitement des interruptions.	255
Traitement en threaded interrupt	260
Interruptions avec Xenomai	268
Real Time Driver Model	268
Interruptions avec RTDM	274
Conclusion	276
Points clés	276
Exercices	277
Exercice 1 (*)	277
Exercice 2 (**)	277
Exercice 3 (**)	277
Exercice 4 (***)	277
CONCLUSION	
État des lieux et perspectives	279
Situation actuelle	279
Linux « vanilla »	279
Patch PREEMPT_RT.	280
Xenomai	280
Mesures	280
Perspectives	281
ANNEXE A	
Compilation d'un noyau	283
Préparation des sources	283
Configuration de la compilation	285
Principes.	285

Interfaces utilisateur	287
Options de compilation	288
Compilation et installation	289
Compilation croisée	289
Compilation native	290
ANNEXE B	
Bibliographie	293
Livres	294
Articles	294
Sites web	295
Index	297

1

Multitâche et commutation

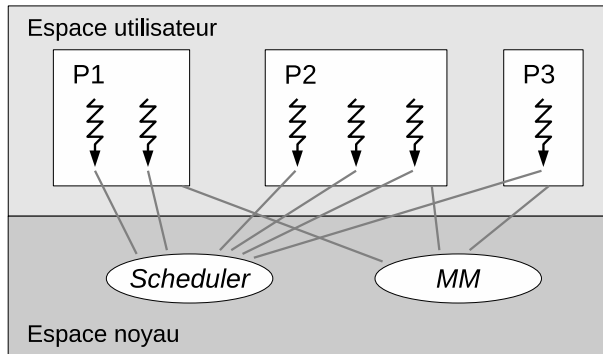
L'ordonnancement sous Linux est avant tout une affaire de gestion des tâches en attente. Sur un poste de travail courant, il tourne en permanence une bonne centaine de processus, dont la plupart sont endormis, en attente d'événements extérieurs (actions de l'utilisateur, données provenant du réseau, 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, dans lequel un ou plusieurs threads s'exécutent. La notion de threads, notamment leur normalisation par le standard Posix, a été introduit relativement tard (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.

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 **privilegié**, 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.

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 execl (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>
```

```
#define LG_LINE 256

int main(void)
{
    char line[LG_LINE];

    while(1) {
        // Afficher un symbole d'invite (prompt)
        fprintf(stderr, "--> ");
        // Lire une ligne de commandes
        if (fgets(line, LG_LINE, stdin) == NULL)
            break;
        // Supprimer le retour chariot final
        line[strlen(line)-1] = '\0';
        // Lancer un processus
        if (fork() == 0) {
            // --- PROCESSUS FILS ---
            // Exécuter la commande
            execvp(line, line, NULL);
            // Message d'erreur si on échoue
            perror(line);
            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
mardi 28 août 2018; 08:22:21 (UTC +0200)
--> who
cpb      tty2          2018-08-26 08:01 (:1)
cpb      pts/2         2018-08-28 08:19 (:1)
-->
```

Toutefois, dès que l'on essaie de passer des arguments sur la ligne de commandes, la fonction `exec1p()` 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 * value);
```

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 ** value);
```

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

int main(int argc, char * argv[])
{
    pthread_t * threads = NULL;
    void * ret;
    int i;
    long n;
    // Vérification des arguments
    if (argc < 2) {
        fprintf(stderr, "usage: %s values...\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);
    }
}
```



```
fprintf(stderr, "main(): starting threads\n");
for (i = 1; i < argc; i++) {
    // Récupération de l'argument numérique
    if (sscanf(argv[i], "%ld", & n) != 1) {
        fprintf(stderr, "%s: invalid\n", argv[i]);
        exit(EXIT_FAILURE);
    }
    // Lancement du thread
    if (pthread_create(& (threads[i-1]), NULL,
        factorial, (void *) n) != 0) {
        fprintf(stderr,
            "Unable to start thread #%d\n", i);
        exit(EXIT_FAILURE);
    }
}
fprintf(stderr, "main(): threads started\n");
for (i = 1; i < argc; i++) {
    // Attente du thread
    pthread_join(threads[i-1], & ret);
    fprintf(stderr, "main(): %s! = %ld\n",
        argv[i], (long) ret);
}
fprintf(stderr, "main(): threads terminated\n");
free(threads);
return EXIT_SUCCESS;
}
```

Voici un petit exemple d'exécution :

```
$ ./exemple-threads 3 5 7
main(): starting threads
3!: computing...
7!: computing...
5!: computing...
main(): threads started
5!: computing...
7!: computing...
3!: computing...
5!: computing...
7!: computing...
main(): 3! = 6
5!: computing...
7!: computing...
7!: computing...
main(): 5! = 120
7!: computing...
main(): 7! = 5040
main(): threads terminated
$
```

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 uniprocasseur 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        : 142
model name    : Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
stepping     : 9
microcode    : 0x84
cpu MHz      : 600.060
cache size   : 4096 KB
physical id  : 0
siblings     : 4
core id      : 0
cpu cores    : 2
apicid       : 0
initial apicid : 0
fpu          : yes
fpu_exception : yes
cpuid level  : 22
wp           : yes
```

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.

```
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmpperf tsc_known_
freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm
pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ibrs ibpb stibp tpr_shadow
vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx
rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln
pts hwp hwp_notify hwp_act_window hwp_epp
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
bogomips : 5808.00
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:
[...]
processor : 1
[...]
processor : 2
[...]
processor : 3
vendor_id : GenuineIntel
cpu family : 6
model : 142
model name : Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
stepping : 9
microcode : 0x84
cpu MHz : 600.345
cache size : 4096 KB
physical id : 0
siblings : 4
core id : 1
cpu cores : 1
apicid : 3
initial apicid : 3
fpu : yes
fpu_exception : yes
cpuid level : 22
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmpperf tsc_known_
freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm
pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ibrs ibpb stibp tpr_shadow
vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx
rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln
pts hwp hwp_notify hwp_act_window hwp_epp
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
bogomips : 5808.00
```

```

cflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management :
$

```

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
$

```

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, pour connaître le nombre de CPU disponibles il est possible d'appeler la fonction :

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

On lui passera, l'argument `_SC_NPROCESSORS_ONLN`. Toutefois, 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("Nb CPU: %ld\n",
           sysconf(_SC_NPROCESSORS_ONLN));
    return EXIT_SUCCESS;
}
```

Qui nous donne :

```
$ ./exemple-sysconf
Nb 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 remarque un bref passage sur le cœur 1 pendant quelques secondes et un retour prolongé sur le cœur 2.