

SYS3
Linux temps réel

Notes provisoires du cours de système d'exploitation

Pierre BETTENS

2006-2007

Première partie
Présentation des concepts

Chapitre 1

Préalables

1.1 Présentation

"Un système temps réel est un système dont les corrections ne dépendent pas uniquement du résultat logique des algorithmes mais aussi de l'instant où ces résultats ont été produits."

Un système temps réel est un système d'informations qui respecte les contraintes de temps. Il se caractérise par les conséquences (dramatiques) pouvant survenir lorsque le système ne respecte pas les conséquences **temporelles**. Un système temps réel ne doit pas être particulièrement rapide mais bien déterministe.

Par exemple, le système de guidage d'un navire peut ne pas paraître temps réel à cause de sa faible vitesse. Néanmoins, c'est un système temps réel. On pourrait croire que les systèmes temps réel ne sont utilisés que par la NASA ... mais l'intégration actuelle des systèmes d'informations et d'électronique dans la vie de tous les jours montre le contraire.

Le lecteur intéressé par divers exemples d'applications temps réel consultera [4], il y découvrira des exemples de tests de moteurs d'avions, d'armement, de croissance de plantes, de contrôle de télescopes, de robots, ...

Aujourd'hui, les applications temps réel ne sont plus spécifiquement réalisées sur des systèmes dédiés. Elles nécessitent un système proposant d'autres applications ; un système de commande de télescope voudra diffuser des images sur le réseau. Afin de répondre à ces exigences, le système doit être **simple, prévisible et optimisé**, ce qui va à l'encontre des demandes de l'utilisateur telles que la publication d'images sur le réseau. Nous verrons dans la suite comment RTLinux résoud ce problème.

Enfin, il faudra distinguer deux types de systèmes temps réel. Le **système temps réel logique** (*Soft real time system*) et le **système temps réel physique** (*Hard realtime system*).

Dans le premier cas, un surcroît de temps dégrade les performances du système. Par exemple dans un système exécutant la décompression et l'affichage d'un fichier *mpg*, ...

Tandis que dans le second cas, il risque de coûter des vies humaines ou de graves pertes matérielles. Par exemples le système d'injection de combustible dans un avion, le système de contrôle de température dans un haut-fourneau, ...

1.2 Implication en terme d'OS

La correction sémantique de la réponse est de la responsabilité du programmeur¹, tandis que la correction temporelle dépend du système d'exploitation².

L'OS.³ doit supporter et organiser l'exécution de toutes les tâches. C'est aussi l'affaire de l'OS. de gérer toutes les interruptions. Il doit offrir :

- ↔ l'algorithme de prévision,
- ↔ les mécanismes de communication inter-processus (sémaphores, messages, etc),
 - ~ un service de boîte aux lettres, service permettant d'envoyer et de recevoir des messages entre les tâches (FIFO dans le monde unix).
 - ~ un système de sémaphores, utilisé pour le partage des ressources, la synchronisation entre les tâches, la création de section critique (opérations DOWN et UP)
- ↔ la gestion des interruptions,
- ↔ la gestion des tâches (*thread*)
 - ~ l'activation des tâches à chacune de leur période,
 - ~ la détermination et le changement d'état des tâches,
 - ~ le changement de contexte et l'élection d'une tâche prête,

Contrairement aux O.S. normaux, le but d'un Système d'Exploitation Temps Réel est de minimiser la complexité de l'OS. Il n'est pas nécessaire d'avoir un système d'exploitation qui fait beaucoup de choses, ce qui est vraiment important c'est l'exécution des tâches dans les délais impartis.

Il vaut mieux un O.S. qui prend normalement 10 unités de temps pour accomplir un changement de contexte et qui en prend 12 dans les pires cas, qu'un autre OS qui en moyenne prend 3 unités de temps mais qui peut, de temps en temps, aller jusqu'à 20.

¹Le programmeur s'efforce d'écrire un programme fournissant une solution correcte, *sémaniquement* correcte

²Le système d'exploitation fournira la solution dans les délais impartis ... sauf s'il s'écroule

³Operating System - Système d'Exploitation

Nous ne serions pas surpris de découvrir que les O.S. Temps Réel sont "plus lents" que les OS normaux. Dans certains cas, afin d'avoir des comportements prévisibles, il pourra même être nécessaire de désactiver la mémoire cache avec la perte de performances associée. La mémoire cache, les processeurs avec des unités pipelines, ... sont des ennemis de la prévisibilité et par conséquent des systèmes temps réel.

Rappels

La mémoire cache

Pour rappel, la mémoire cache accélère la communication entre le processeur et un composant servant à stocker des données (RAM, disque dur, "internet"). La caractéristique principale de cette mémoire est d'être plus rapide que l'unité de stockage concernée.

Dans le cas de la **RAM**, la mémoire cache est généralement intégrée au processeur (ou bien est de type SDRAM). Elle est plusieurs dizaines de fois plus rapide que la mémoire RAM.

Dans le cas du **disque dur**, on utilisera de la mémoire RAM comme cache pour le disque dur. On parle alors de mémoire virtuelle, ou de *swap*.

Dans le dernier cas, une partie du disque dur est utilisée pour stocker localement des pages web afin d'éviter des requêtes internet.

Le principe de la mémoire cache est simple ; recherche d'une donnée dans la mémoire cache avant la recherche en mémoire. Si l'information se trouve en mémoire cache, il y a *succès de cache* et l'information est utilisée. Dans le cas contraire, *défaut de cache*, l'information est recherchée sur le "support lent".

Unité pipeline

Le principe consiste en la mise en file d'attente (*pipelining*) des instructions dans la mémoire cache du processeur. Ainsi, pendant l'exécution d'une instruction, la suivante est déjà mise à disposition (récupération dans la mémoire cache ou RAM). On aura une économie de cycles d'horloge.

On peut résumer, si F=fetch, D=décode, E=exécute

F1 - D1 - E1 - F2 - D2 - E2 - F3 - D3 - E3 (sans pipelining)

F1 - (D1 + F2) - (E1 + D2 + F3) - (E3 + D3) - E3 (avec pipelining)

Chapitre 2

Notions de modules

2.1 Définition

Un **module** est un fragment du système d'exploitation que l'on peut insérer ou soustraire à tout moment, ce module est chargé dynamiquement. Ces modules sont généralement utilisés pour accéder au matériel (ports, mémoires, interruptions, ...).

Lorsque l'on compile ¹ un programme comprenant plusieurs fichiers source, chaque fichier est compilé séparément, dans un premier temps, pour engendrer un fichier objet *.o*, puis les objets sont liés ensemble, en résolvant toutes les références et en engendrant un fichier exécutable unique. Supposons que le fichier objet comportant la fonction `main` puisse être exécuté, et que le système d'exploitation soit capable de charger en mémoire et de procéder à l'édition de liens des autres fichiers objets quand cela s'avère nécessaire. Et bien, le noyau est capable de cela vis-à-vis de lui-même. Quand le système GNU/Linux démarre, seul l'exécutable `vmlinux` est chargé en mémoire, il contient les éléments indispensables au noyau, et plus tard, pendant l'exécution, il peut charger ou oublier de manière sélective les modules requis.

Les modules forment une fonctionnalité optionnelle du noyau Linux, qu'il faut préciser au moment de la compilation du noyau (généralement les noyaux sont compilés avec cette option).

Il est même possible de créer de nouveaux modules et de les charger sans devoir recompiler ou réamorcer le système. Quand un module est chargé, il se transforme en partie intégrante du système d'exploitation. Ce qui implique que

- ↔ il peut utiliser toutes les fonctions et accéder à toutes les variables et à toutes les structures du noyau,
- ↔ le code du module est exécuté avec le niveau maximal de privilèges du

¹Dans la suite nous ferons toujours référence au langage C, le lecteur pourra consulter [1]

processeur. Sur l'architecture i386, il est exécuté au niveau 0 (ring level 0), en conséquence de quoi il peut disposer de tout type d'accès aux entrées/sorties et exécuter des instructions privilégiées,

- ↪ la mémoire du programme et de ses données est directement réservée en mémoire physique, et il n'est pas possible de la paginer. C'est pourquoi il est impossible d'engendrer une faute de page pendant l'exécution d'un module.

Comme on peut le voir, un module chargé dynamiquement dispose déjà de certaines des caractéristiques d'un programme temps réel : il évite les retards provoqués par des fautes de pages et il peut accéder à toutes les ressources du matériel.

2.2 Mise en oeuvre

Un module est écrit à partir d'un code source écrit en C. Ce code source a les particularités suivantes :

- ↪ il ne contient pas de fonction `main` mais deux fonctions `init_module` et `cleanup_module` exécutées respectivement au chargement et au déchargement du module dans le noyau,
- ↪ il doit inclure un librairie propre ; `linux/module.h`,
- ↪ il n'est pas exécutable, nous nous intéressons uniquement au fichier objet (`.o`), qui sera intégré dans le noyau,
- ↪ il est compilé en utilisant la macro `__KERNEL__`

Une fois ce code écrit et compilé, il faudra insérer le module dans le noyau. Le lecteur s'intéressera aux commandes `lsmod`, `insmod` et `rmmod` permettant, respectivement, de lister les modules inclus dans le noyau, en ajouter et en supprimer.

2.3 Exemple Hello world

Voici un exemple de module. Pour ne pas déroger à la règle, ce module affiche *Hello world*.

Le noyau ne disposant pas de sortie standard, la fonction `printf()` n'est pas disponible, elle est remplacée par `printk()` qui envoie le message dans un tampon, lisible grâce à la commande

```
dmesg | tail -12
```

²Un `tail -f /var/log/syslog` en root dans une console permet d'avoir continuellement les messages du noyau.

```
#define MODULE
#include <linux/module.h>

/*
 * Hello world module
 */

int init_module(void) {
    printk("Module_'Hello_world'_'_inserted_\n");
    return 0;
}

void cleanup_module(void) {
    printk("Module_'Hello_world'_'_removed_\n");
}
```

Chapitre 3

Notions de threads

3.1 Définition

Un *thread* est un processus léger.

Pour rappel, un *processus*

- ↪ possède un espace mémoire (éventuellement virtuel) pouvant contenir une image du programme,
- ↪ contrôle certaines ressources (fichiers, périphériques d'E/S, ...),
- ↪ est un "chemin d'exécution" à travers un programme (c'est un programme en cours d'exécution),
- ↪ est dans un certain état d'exécution et possède une certaine priorité.

On peut distinguer deux caractéristiques différentes, la première liée aux deux premiers items - la **possession de ressources** (*resource ownership*) et la seconde liée aux deux derniers items - la **répartition des ressources** (*dispatching*).

La **possession de ressources** est généralement attribuée au **processus** qui a un espace d'adressage, un accès au processeur, des fichiers, des I/O, ... Le **partage de ressources** fait, quant à lui, référence aux *threads*. Un *thread* a un état d'exécution (*wait, run, ready*), sauve son état lorsqu'il est en attente, possède une pile, a accès à la mémoire et aux ressources des autres *threads*.

Travailler en utilisant les *threads* apporte certains avantages ;

- ↪ gain de temps lors de la création d'un *thread* puisqu'il utilise le même espace d'adressage,
- ↪ gain de temps pour terminer un *thread*,
- ↪ gain de temps pour *switcher* (changement de contexte) d'un *thread* à l'autre,
- ↪ gain de temps lors de la communication entre *threads* (puisque'ils partagent leur espace d'adressage).

Il existe deux types de *threads*, les **threads systèmes** et les **threads utilisateurs**. Les premiers utilisent la politique d'ordonnement du système qui

leur attribue donc un *pid* et "s'en occupe". Les seconds sont intégralement gérés par le programmeur, c'est lui qui devra gérer l'ordonnement des tâches ... à l'aide de la bibliothèque qui va bien. Dans ce cas de figure, l'OS n'est pas au courant de l'existence de plusieurs *threads*, ce qui implique qu'un appel système de l'un d'entre eux, bloque le processus les contenant ... et ce, même si le *thread* en état *running* continue à croire qu'il l'est.

3.2 Mise en oeuvre

Un *thread* est écrit à partir d'un code source C. Ce code a les particularités suivantes :

- ↪ il doit inclure la librairie `pthread.h`,
 - ↪ et lier cette librairie à la compilation `-lpthread`,
 - ↪ pour créer un nouveau *thread*, on utilise la fonction `pthread_create()`, avec les paramètres précisant le type d'ordonnement (temps réel ou non), l'identifiant du *thread* et sa fonction principale (`start_routine`) ainsi que ses paramètres,
 - ↪ pour supprimer le *thread*, on utilisera la fonction `pthread_cancel()`
- La signature de la fonction `pthread_create()` est la suivante ;

```
#include <pthread.h>

int pthread_create(pthread_t * thread,
pthread_attr_t * attr,
void * (* start_routine) (void *),
void * arg);
```

Le *thread* est créé en utilisant les attributs spécifiés dans `attr`. Si l'attribut est `NULL`, les attributs par défaut sont utilisés. Les valeurs par défaut précisent, par exemple, que le *thread* ne sera pas temps réel.

Pour plus de détails, quant aux attributs, consultez les fonctions POSIX suivantes (man `pthread_attr_init`):

- ↪ `pthread_attr_init()`,
- ↪ `pthread_attr_setshdparam()`,
- ↪ `pthread_attr_getshdparam()`,
- ↪ `pthread_attr_getcpu_np()`,
- ↪ `pthread_attr_setcpu_np()`,

3.3 Exemple Hello world

Voici un exemple d'utilisation des *threads*. Pour, à nouveau, ne pas déroger à la règle ce programme crée deux *threads* affichant tous deux *Hello world*.

```
#include <pthread.h>

/*
 * hello.c
 *
 * Hello world example.
 */

/*
 * run
 *
 * method executed when thread create, this method say "Hello"
 */
void* run (void* arg) {
    int id = (int) arg ;

    printf("My_id_is_%d, I say 'Hello'\n", id) ;
    pthread_exit(0);
} // end - run

/*
 * main method
 *
 * I create two thread and I wait they finish to say Hello
 */
int main ( int argc, char* argv[] ) {
    // Create 2 threads
    pthread_t task1, task2 ;

    pthread_create (&task1, NULL, run, (void*) 0 ) ;
    pthread_create (&task2, NULL, run, (void*) 1 ) ;
    pthread_join(task1, NULL) ;
    pthread_join(task2, NULL) ;

    exit(0);
} // end - main
```

Pour compiler cet exemple, il ne faut pas oublier de lier la librairie *pthread*, soit la commande :

```
gcc hello.c -o hello -lpthread
```

3.4 Synchronisation entre *threads*

Lorsque plusieurs *threads* s'exécutent, ils doivent pouvoir se synchroniser. Il existe différentes manières d'arriver à ce résultat ; les *sémaphores*, les *variables de conditions* et les "*mutex*" (sémaphore d'exclusion mutuelle).

3.4.1 Sémaphores

Un *sémaphore* est une variable protégée représentant un nombre de ressources disponibles. Lorsque qu'un processus veut utiliser une ressource, il "consulte" le sémaphore afin de savoir si la dite ressource est disponible. Si oui, il l'utilise, sinon, il attend.

Ce concept a été introduit par Edsger Dijkstra dans les années soixante, il est basé sur le principe des sémaphores des frères Chappe (XVIII siècle)¹. L'idée de base est "*la personne qui lève le bras ou pas*". Supposons avoir une série de trains devant passer sur une seule voie. Le sémaphore sera levé lorsque un train est engagé sur la voie, baissé sinon. Dès qu'un train veut s'engager, il regarde² l'état du sémaphore. S'il est baissé, il peut s'engager et demander la levée du sémaphore afin de rester seul sur la voie. Lorsqu'il est passé, il demande d'abaisser le sémaphore pour laisser la voie libre aux autres.

Informatiquement parlant, il faudra être sûr que la demande d'un sémaphore soit une opération **indivisible**. En effet, il serait malheureux qu'entre la demande d'un sémaphore et son obtention, la main soit passée à un autre processus. Ce principe de sémaphore garantit cette indivisibilité³.

Les sémaphores ne préservent pas d'un *deadlock* (interblocages), c'est au programmeur d'éviter de demander à deux processus de s'attendre mutuellement, ce qui paraît absurde de prime abord ne l'est peut-être pas dans la pratique.

¹Wikipedia.fr est notre amie.

²Ou du moins le chauffeur

³C'est pourquoi nous ne mettons pas en oeuvre les sémaphores à l'aide de "simples" variables.

Les trois opérations sur un sémaphore sont

- ↪ **Init**,
- ↪ **P** pour *Probeer*⁴, *up* et
- ↪ **V** pour *Verhoog*, *down*

L'opération **Init** n'est utilisée qu'une seule fois afin d'initialiser le sémaphore. L'API POSIX fournit la fonction `sem_init` qui permet d'initialiser un sémaphore, le type associé est `sem_t`.

Synopsis de la fonction `sem_init`

```
int sem_init(sem_t *sem, int pshared, unsigned int valeur);
```

où `sem` est un pointeur vers le sémaphore, `pshared` indique si le sémaphore est local au processus courant (`pshared=0`, seule valeur supportée par la bibliothèque `linux`) ou partagé entre plusieurs processus, `valeur`, la valeur de départ du sémaphore, soit le nombre de ressources disponibles.

Nous aurons par exemple,

```
#include <semaphore.h>

sem_t sid;          // semaphore id
sem_init(&sid, 0, 3)
```

L'opération **V** décrémente le sémaphore s'il est non nul ou place le *thread* en attente sinon. L'API POSIX fournit la fonction `sem_wait` dont voici le synopsis

```
int sem_wait(sem_t * sem);
```

L'opération **P** incrémente le sémaphore et signale donc la présence d'une ressource supplémentaire. L'API POSIX fournit la fonction `sem_post` qui ne bloque jamais. Voici son synopsis

```
int sem_post(sem_t * sem);
```

Le lecteur intéressé n'hésitera pas à lire les pages de **man** expliquant ces fonctions. Il pourra aussi compléter sa lecture avec les pages de manuel des fonctions

- ↪ `sem_open()`,
- ↪ `sem_close()`,
- ↪ `sem_unlink()`,
- ↪ `sem_destroy()`,
- ↪ `sem_getvalue()` et
- ↪ `sem_trywait()`

⁴Dijkstra est hollandais.

3.4.2 Sémaphore d'exclusion mutuelle, *mutex*

Un *sémaphore d'exclusion mutuelle*⁵ est un sémaphore ne contrôlant qu'une seule ressource, sa valeur est donc binaire. Un *mutex* permet donc d'assurer qu'un seul *thread* exécute une section critique de code.

Comme pour les sémaphores (puisque s'en est un), le *mutex* ne nous met pas à l'abri des *deadlocks*. Le lecteur avide d'histoire se renseignera sur "*Mars pathfinder*", un petit robot utilisé sur Mars resté coincé à cause d'un *deadlock* sur un *mutex*.

Hormis les phases d'initialisation et de destruction d'un *mutex*; les opérations sur celui-ci sont soit demander le *mutex* (*lock*), soit le relâcher (*unlock*).

L'API POSIX fournit les fonctions ;`pthread_mutex_init`,`pthread_mutex_lock`,`pthread_mutex_unlock`,`pthread_mutex_unlock`, ...

Les synopsis des fonctions sont les suivants ;

```
#include <pthread.h>

int pthread_mutex_init(
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

La fonction `pthread_mutex_init` permet d'initialiser un *mutex* avec les valeurs contenues dans `mutexattr`. Si cette valeur vaut `NULL`, le *mutex* est initialisé avec les valeurs par défaut.

La fonction `pthread_mutex_lock` verrouille le *mutex* s'il est libre. Si le *mutex* est déjà verrouillé, la fonction bloque le processus qui se met en attente sur une liste d'attente. Au retour, le processus devient propriétaire d'un *mutex*, il le verrouille et prend la main.

La fonction `pthread_mutex_trylock` est la version non bloquante de `pthread_mutex_lock`. Son utilité se montre afin de mettre en place des mécanismes évitant les *deadlocks*.

La fonction `pthread_mutex_destroy` détruit le *mutex* s'il n'est pas verrouillé. Suivant l'implémentation `linux` des recommandations POSIX, aucune ressource ne peut être allouée à un *mutex*. Cette fonction ne fait donc que vérifier que le *mutex* n'est pas verrouillé.

D'autres fonctions sont associées aux *mutex*, notamment celles permettant de fixer ses attributs. Le lecteur peut consulter les pages de **man** à ce sujet ;

⁵Dans la suite nous dirons toujours *mutex*

- ↪ pthread_mutexattr_init,
- ↪ pthread_mutexattr_destroy et
- ↪ pthread_mutexattr_setpshared

3.4.3 Conditions variables

Les variables de conditions permettent la synchronisation de *threads* en se basant sur la valeur d'un attribut d'une variable partagée. Les **conditions** sont toujours utilisées en conjonction avec un *mutex* qui protège la variable de condition ... comme toute variable partagée.

Avec ces *conditions*, un *thread* peut aisément bloquer jusqu'à ce qu'une condition soit vérifiée. La condition est testée sous la protection d'un *mutex*. Si la condition est **fausse**, le *mutex* est relâché et le *thread* bloque sur la condition ... en attendant qu'elle change. Lorsqu'un autre *thread* change la condition (sous la protection du *mutex*) cela a pour effet de débloquent un, ou plusieurs *threads* en attente. Ils peuvent alors reprendre le *mutex* et tester la condition à nouveau.

Les opérations sur une *variable de condition*, outre les opérations d'initialisation et de destruction, sont signaler la condition et attendre sur une condition (bloquer le *thread* jusqu'à ce qu'un autre *thread* signale la condition).

L'API fournit les fonctions ; pthread_cond_init,

pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait et pthread_cond_destroy dont voici les synopses,

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(
    pthread_cond_t *cond,
    pthread_condattr_t *cond_attr);
int pthread_cond_wait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_timedwait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct time-spec *abstime);
int pthread_cond_destroy(pthread_cond_t *cond);
```

La fonction pthread_cond_init permet d'initialiser une condition. La condition pointée par cond est initialisée en utilisant les valeurs par défaut si cond_attr vaut null ou celles positionnées par la fonction pthread_condattr_init sinon.

La fonction pthread_cond_wait permet de relâcher le *mutex* et de bloquer sur la condition jusqu'à ce qu'un autre *thread* modifie la condition. Le *mutex*, *mutex* est relâché tandis que la condition cond est bloquée.

Lorsqu'un *thread* modifie la condition, il envoie un signal aux autres *threads* via une des fonctions pthread_cond_signal ou pthread_cond_broadcast. Lorsqu'un *thread* bloqué sur une condition reçoit un tel signal, il essaie d'acquiescer le *mutex* afin de pouvoir retester la condition. La fonction pthread_cond_signal permet de relancer un *thread* en attente sans savoir lequel. Tandis que la fonction pthread_cond_broadcast relance tous les *threads*.

La fonction pthread_cond_timedwait agit comme la fonction wait à ceci près qu'elle bloquera sur la condition jusqu'à ce que celle-ci soit vérifiée ou qu'un quota de temps soit écoulé.

La fonction pthread_cond_destroy permet de détruire le *mutex*.

Il doit être clair que toutes ces opérations sont liées atomiquement. L'API garantit par exemple que le déverrouillage du *mutex* et l'attente sur une condition sont liés. Si l'on croit qu'un *thread* pourrait ne pas recevoir un signal car ce signal lui serait envoyé entre le moment où il déverrouille son *mutex* et le moment où il se met en attente sur sa condition, c'est faux ... car ces deux opérations sont liées atomiquement.

3.5 Communication entre threads

La communication entre *threads* est le moyen de transférer de l'information d'un *thread* à un autre. Le concept de communication entre *threads* recoupe habituellement les notions de **signaux**, **pipe** et **fifo** (encore appelés *named pipe*).

Nous ne traiterons pas ici des **signaux**, ce sujet ayant été abordé dans le cours de système d'exploitation de deuxième (SYS2). Le lecteur désireux d'avoir un rappel peut simplement consulter les pages **man** des fonctions, signal et kill⁶.

3.5.1 Pipe

Un *pipe* est un canal de communication entre processus. En vulgarisant le principe, nous pouvons le comparer à un "tuyau" (ou tube si l'on fait une traduction simple). Un processus envoie de l'information d'un côté et l'autre processus la récupère de l'autre. Tout utilisateur linux connaît la "commande" associée, j'ai nommé |. Pipe s'utilise tous les jours, par exemple ;

ls | more

⁶C'est plutôt man 2 signal et man 2 kill qu'il faudra faire dans ce cas

La fonction `pipe` de la librairie `unistd.h` permet de créer un tel "tube" sur base de deux *file descriptor* (descripteurs de fichiers). Le synopsis de la fonction est le suivant;

```
#include <unistd.h>

int pipe(int filedes[2]);
```

`filedes` représente les deux descripteurs de fichiers associés au *pipe*. Pour utiliser un *pipe*, il faut être attentif au **sens de parcours** dans le tube. En effet, un *pipe* est unidirectionnel. `filedes[0]` permet de lire dans le tube tandis que `filedes[1]` permet d'écrire. Un moyen mnémotechnique consiste à faire l'analogie avec *stdout* et *stdin*. L'entrée standard (*file descriptor*=0) est consacrée à la lecture au clavier. La sortie standard (*file descriptor*=1) est, quant à elle, consacrée à l'écriture à l'écran.

Dans le cas particulier des *threads*, la fonction `main` créera un *pipe* qui sera partagé par les threads. Ceux-ci l'utiliseront à loisir.

Par exemple,

```
/*
 * Illustration de l'utilisation d'un pipe
 */
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>

pthread_t tecrivain ; // thread ecrivain
pthread_t tlecteur ; // thread lecteur

int filedes[2] ; // file descriptor

void* tecrivain_run (void* param) ;
void* tlecteur_run (void* param) ;

/*
 * Creation de deux threads partageant un êmme pipe
 */
int main() {
    // Creation du pipe
    if ( pipe(filedes) == -1 ) {
        fprintf(stderr, "Erreur_de_êcracion_de_pipe" ) ;
        exit(-1) ;
    }
}
```

```
/* lancement des threads
pthread_create(&tecrivain, NULL, tecrivain_run, 0) ;
pthread_create(&tlecteur, NULL, tlecteur_run, 0) ;

pthread_join(tecrivain, NULL) ;
pthread_join(tlecteur, NULL) ;
exit(0) ;
}

/*
 * Thread tecrivain, affiche et ecrit dans le pipe
 */
void* tecrivain_run (void* param) {
    char* s = "Hello_world" ;
    printf("êcrivain_êcrit_ %s_dans_le_pipe\n", s) ;
    write(filedes[1], &s, sizeof(s) ) ;
    return ;
}

/*
 * Thread tlecteur, lit dans le pipe et affiche
 */
void* tlecteur_run (void* param) {
    int i ;
    char* s ;
    read(filedes[0], &s, 11 ) ;
    printf("tlecteur_lit_ %s_dans_le_pipe\n", s) ;
    return ;
}
```

3.5.2 *Fifo* ou pipe nommé

Le principe d'un *fifo* est semblable au *pipe* si ce n'est que le second n'est pas anonyme, il est nommé. Les seules différences vont donc résider dans la manière de l'utiliser.

Un *fifo* est un fichier spécial. Il fait partie du *file system*, plusieurs processus peuvent l'ouvrir en lecture et/ou en écriture. Il est spécial en ce sens que lors d'un accès à ce *fifo*, le noyau ne passe pas par le *file system* pour faire transiter l'information, cela se fait en interne dans le noyau. Ce fichier spécial est un fichier sans contenu, son utilité est d'avoir un nom, qu'il pourra être ouvert

et/ou fermé grâce aux appels systèmes standards.

Un *fifo* peut être créé de plusieurs manières. Il est possible d'utiliser une des commandes, `mkfifo` ou `mknod`. Voici leur synopsis donné par **man** (`man mkfifo`, `man mknod`).

```
mkfifo [options] fichier...

Options POSIX : [-m mode] [--]
Options GNU (versions courtes) : [-m mode] [--help]
                                [--version] [--]

mknod [options] nom {bc} énumro_majeur énumro_mineur
mknod [ options ] nom p
```

La commande `mknod` étant plus générale, elle est simplement donnée à titre d'information.

Il est également possible de créer un *fifo* grâce aux fonctions fournies soit par le système (`mknod`), soit par les bibliothèques `sys/types.h`, `sys/stat.h` (`mkfifo`). **man 3 mkfifo** nous donne le synopsis de la fonction `mkfifo`. Vous pouvez consulter celui de la fonction `mknod` *via* un `man 2 mknod`⁷.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo ( const char *pathname, mode_t mode);
```

où `mode & ~umask`⁸ positionne les droits d'accès au *fifo*.

Lorsque le *fifo* est créé, il reste à ouvrir le fichier spécial dont le nom est pointé par *pathname* et recevoir un descripteur de fichier. Nous utilisons les fonctions habituelles, `open` et `close`.

3.6 Problèmes classiques

La programmation *multi-threads* entraîne son lot de problèmes bien connus concernant la communication et la synchronisation de processus (et/ou de *threads*). Rappelons ici 3 problèmes fréquemment débattus (tous les détails se trouvent dans [3]).

⁷Nous constatons alors que `mknod` est un appel système, tandis que `mkfifo` dépend d'une bibliothèque.

⁸L' `umask` du processus

3.6.1 Le problème des philosophes

Ce problème illustre les difficultés rencontrées lors d'un accès à un nombre limité de ressource. Il s'énonce comme suit :

Cinq philosophes se retrouvent assis autour d'une table ronde afin de manger un plat de spaghettis. Ils disposent chacun d'une assiette de spaghettis et d'une fourchette comme illustré sur la figure 3.1. Les spaghettis étant particulièrement huileux et donc glissants, ils ne peuvent manger avec une seule fourchette, ils en ont besoin de deux.

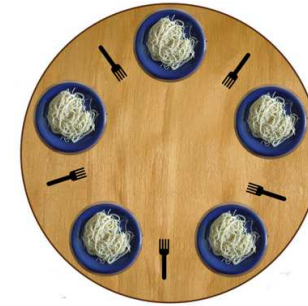


FIG. 3.1 – Source : <http://wikipedia.fr>

*Comme chacun sait, un philosophe passe son temps à **penser** et à **manger**. Lorsqu'il a faim, il essaie de se saisir des deux fourchettes qui entourent son assiette. S'il y arrive, il mange un certain temps puis repose ses fourchettes et se remet à penser. S'il n'y arrive pas, il attend.*

Ce problème est typique car il faut que l'action "prendre la fourchette gauche et la fourchette droite" soit protégée. En effet, si tous les philosophes prennent leur fourchette gauche en même temps, nous sommes dans une situation de famine ... c'est-à-dire une situation dans laquelle l'action est devenue répétitive et inutile.

Si nous voulons implémenter la solution proposée par Andrew Tanenbaum ([3] pp64-70), nous devons utiliser

- ↪ un tableau `etat` mémorisant l'activité des philosophes. Un philosophe MANGE, PENSE ...ou a FAIM,
- ↪ des macros GAUCHE et DROITE afin de connaître les voisins de chaque philosophe,
- ↪ un tableau de sémaphores permettant de mettre en attente un philosophe qui a faim et

↪ un *thread* par philosophe.

Ainsi, un philosophe (son *thread*) ne fera que penser (pendant un certain temps), prendre les fourchettes (ou bloquer si elle ne sont pas libres), manger, et reposer les fourchettes.

3.6.2 Le problème des lecteurs / rédacteurs

Ce problème ([2]) illustre, quant à lui, les accès à une base de données.

Nous distinguons ici les lecteurs des rédacteurs. Nous acceptons que plusieurs lecteurs accèdent à la base de données mais lorsqu'un rédacteur veut la modifier, il doit être le seul à y accéder. PJ Courtois [2] propose deux solutions pour résoudre le problème. L'une donnant priorité aux lecteurs l'autre aux rédacteurs.

Si l'on donne priorité aux lecteurs, un rédacteur doit attendre que plus aucun lecteur n'accède à la base de données avant d'y accéder. PJ Courtois utilise

- ↪ un sémaphore d'accès à la base de données, positionné par un rédacteur, le premier lecteur ou le dernier,
- ↪ une variable comptant le nombre de lecteurs accédant à la base de données,
- ↪ un *mutex* pour protéger cette variable.

3.6.3 Le problème du coiffeur endormi

Ce problème (comme décrit dans [3] p69) a pour cadre un salon de coiffure avec

- ↪ un coiffeur,
- ↪ un fauteuil de coiffure et
- ↪ n chaises pour les clients qui attendent.

Lorsqu'il n'y a pas de client, le coiffeur dort dans son fauteuil. Le premier client réveille le coiffeur. Les clients suivants s'assoient s'il y a de la place sinon, ils resortent.

La solution de A.Tanenbaum utilise

- ↪ un sémaphore *clients* pour compter le nombre de clients en attente,
- ↪ un sémaphore *coiffeur* pour compter le nombre de coiffeurs libres (0 ou 1),
- ↪ un *mutex* utilisé pour gérer la section critique et
- ↪ une variable *attente* pour compter le nombre de clients en attente.

Nous avons besoin d'une variable *attente*, "copie" du sémaphore car nous n'avons pas accès à la valeur d'un sémaphore et le client entrant doit savoir s'il s'assied ($attente \leq nombre_hises$) ou s'il sort (sinon).

Le **coiffeur** répète toujours la même chose, il attend un client, lorsqu'un client arrive il met à jour le nombre de clients (dans une section critique) et coiffe.

Le **client** quant à lui ne fait qu'une seule fois; entrer dans le salon (dans une section critique, regarder s'il y a de la place et si oui attendre ou réveiller le coiffeur). Si le client attend, il peut s'endormir jusqu'à ce que le coiffeur soit disponible. Lorsque sa coupe est terminée, il sort.

Chapitre 4

Programmation temps réel, *RTLinux*

4.1 Présentation

RTLinux est un système d'exploitation temps réel destiné aux applications ayant de réelles, sérieuses et non négociables *deadlines*. RTLinux se décline en plusieurs versions ; **RTLinuxPro** et **RTLinuxFree**. Nous nous intéresserons à la version gratuite. Même si elle offre certaines limitations, elle sera suffisante dans le cadre de notre laboratoire.

Comme nous l'avons dit précédemment certains aspects des systèmes d'exploitation ne sont pas compatibles avec les contraintes temps réel. Pour proposer un OS temps réel, on aurait pu enlever du noyau Linux tous ses aspects imprévisibles (algorithme de gestion des tâches, drivers, appels systèmes "ininterrompables", mémoire virtuelle, ...). Il est facile de se rendre compte que cette approche n'est pas la bonne ...

RTLinux résoud le problème d'une façon radicalement différente. Au lieu de modifier le noyau de Linux (voir 4.1), pour le rendre prévisible, il construit directement sur le processeur (i386) un petit noyau – indépendant de celui de Linux – avec un gestionnaire de tâches. Le noyau de Linux tourne au dessus de lui partageant le processeur avec les autres tâches temps réel. Linux partage alors le noyau avec d'autres tâches (voir 4.1). Plus précisément, *Linux est une tâche en arrière plan et ne tourne que lorsqu'aucune autre tâche Temps Réel n'est active.*

Il est même encore plus surprenant de savoir qu'il est possible d'installer ou de désinstaller le gestionnaire de tâches dynamiquement, car il est compilé comme un module. L'aspect temps réel du noyau rlinux est un module que l'on peut -ou pas- charger dans le noyau.

Le code du noyau Linux (comme tout O.S.) désactive généralement les interruptions comme moyen de synchronisation ou pour implanter des sections critiques (cfr cours SYS2). S'il y a une interruption de l'horloge pendant que Li-

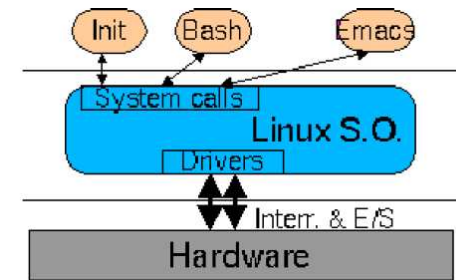


FIG. 4.1 – Linux (sans RTLinux)

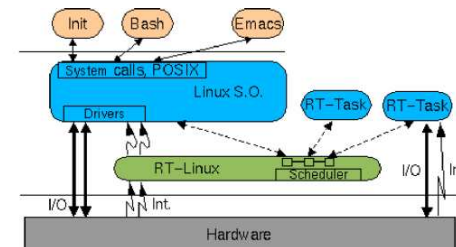


FIG. 4.2 – RTLinux

nux désactive les interruptions, elle reste bloquée et il y a par conséquent une grande imprécision temporelle. RTLinux implante une solution élégante afin d'éviter ces désagréments; tous les appels à `cli`, `sti` (*man* est ton ami) et `iret`¹ sont remplacés par `S_CLI`, `S_STI` et `S_IRET` qui les émulent. De la sorte, Linux ne peut jamais désactiver les interruptions et le noyau RTLinux garde le contrôle du système.

Le gestionnaire de tâches par défaut qui est livré avec RTLinux est préemptif², à priorités fixes et considère la tâche Linux comme celle de plus basse priorité. Si les tâches Temps Réel consomment tout le temps du processeur, alors la tâche Linux n'aura plus de temps CPU affecté et pourra donner l'impression que le système s'est arrêté. Avec RT-Linux nous n'avons pas seulement un Système Temps Réel mais aussi un O.S. classique. On peut surfer sur le web en même temps que l'on interroge et commande un système physique.

4.2 Structure de base d'un programme temps réel

Un programme RTLinux, comme nous l'avons déjà dit, n'est pas un programme "standalone", ce sont des modules qui seront chargés dans le noyau. Au chargement du module, la fonction `init_module` lancera un (ou plusieurs) *thread*. La particularité de ce *thread* est qu'il sera "temps réel", c'est-à-dire que la politique d'ordonnancement associée au *thread* ne sera pas "normale" (`SCHED_OTHER`) mais temps réel (`SCHED_RR` pour round robbin ou `SCHED_FIFO` pour fifo).

Un programme aura donc la structure suivante

```
#include ...

// declaration des differents threads
pthread_t thread;

void* start_routine(void* arg) {
    // definition de l'attribut du thread

    //lancement de son execution
    pthread_make_periodic_np(...)

    //action
}
```

¹Pour rappel, ce sont des appels assembleur qui modifient l'état des interruptions

²C'est-à-dire qu'il peut être ôté sans rendre le système instable

```
int init_module(void ) {
    return pthread_create(...);
}

int cleanup_module(void ) {
    pthread_delete_np(...)
}
```

4.3 Exemple *Hello world*

Voici un exemple de programme temps réel. Il est basé sur l'exemple `hello` de la distribution RTLinux ... et comme son nom l'indique, il dit *bonjour*.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t thread; // id du thread

void * start_routine(void *arg)
{
    struct sched_param p;

    p.sched_priority = 1;
    pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

    pthread_make_periodic_np (
        pthread_self(),
        gethrtime(),
        500000000);

    while (1) {
        pthread_wait_np ();
        rtl_printf("Hello_(my_arg_is_%x)\n",
            (unsigned) arg);
    }
    return 0;
}
```

```

int init_module(void) {
    return pthread_create (&thread, NULL, start_routine, 0);
}

void cleanup_module(void) {
    pthread_delete_np (thread);
}

```

La fonction **init_module** (exécutée lors du chargement du module dans le noyau RTLinux) crée une *thread* dont le code se trouve dans la fonction `start_routine` qui reçoit comme argument : 0.

La fonction **start_routine** est propre au *thread*,

- ↪ La section initialisation demande au scheduler une priorité de 1 par l'appel à `p.shed_priority`.
- ↪ La fonction `pthread_setchedparm` définit le comportement du scheduler pour toutes les exécutions du thread, dans ce cas un ordonnancement "fifo",
- ↪ L'appel à la fonction `pthread_make_periodic_np()` demande au scheduler d'exécuter périodiquement - à une fréquence de 2Hz (500 microsecondes) - le *thread*.
- ↪ La boucle `while`
 - ˘ commence par un appel à la fonction `pthread_wait_np()` qui bloque toutes les exécutions futures jusqu'au prochain appel du *scheduler*. Lorsque le *thread* est appelé à nouveau, il s'exécute jusqu'à ce qu'il rencontre à nouveau un appel à la fonction `pthread_wait_np()`.
 - ˘ la fonction `rtl_printf()` n'appelle aucun commentaire.

La fonction **cleanup_module()** permet de fermer proprement le *thread*. L'appel à la fonction `pthread_join()` désalloue les ressources précédemment allouées au *thread* tandis que la fonction `pthread_cancel()` annule le thread. En lieu et place de ces deux fonctions, on peut faire appel - suivant la version de RTLinux - à la fonction `pthread_delete_np()`.

4.4 Communication entre tâches, les *fifos* temps réels

La communication entre tâches temps réel se fait au moyen des pipes nommés temps réel (*realtime fifos*). Le concept est tout à fait semblable à celui de *named pipe*, pipes nommés.

Un *fifo* temps réel est un fichier spécial dans lequel un *thread* temps réel peut lire ou écrire. Généralement, ils correspondent aux fichiers `/dev/rtfi`.

Un *fifo* temps réel est créé grâce à la commande `rtf_create`. Il est utilisé grâce à des fonctions spécifiques à la librairie `linux/rtf.h`. Le synopsis des fonctions est le suivant

```

#include <rtl_fifo.h>

int rtf_create(unsigned int fifo, int size);
int rtf_create_handler(unsigned int fifo, int (* handler)());
int rtf_get(unsigned int fifo, char * buf, int count);
int rtf_put(unsigned int fifo, char * buf, int count);
int rtf_destroy(unsigned int fifo);

```

rtl_create(unsigned int fifo, int taille) : crée une file de type *fifo* avec un tampon de taille *taille*. À partir de ce moment, et jusqu'à sa destruction, le périphérique auquel on accède par le fichier `/dev/rtf[fifo]` existe et peut être utilisé.

rtl_destroy(unsigned int fifo) : la file de type *fifo* correspondante est détruite et sa mémoire est ré-allouée.

rtl_fifo_put(fifo, char *tampon, int compte) : tente d'écrire *compte* octets à partir du tampon *tampon*. S'il ne reste pas suffisamment d'espace dans le tampon de la file de type *fifo*, cette fonction renvoie -1.

rtl_fifo_get(fifo, char *tampon, compte) : tente de lire *compte* à partir de la file de type *fifo*, et s'il n'y reste pas suffisamment de données cette fonction renvoie -1.

Deuxième partie

Manipulations au laboratoire

Chapitre 5

Mise en place de l'environnement de travail

5.1 Préalable

Cette section présente les diverses choses à faire (ou à vérifier) afin d'avoir à disposition une machine permettant la compilation. Elle introduit aussi les concepts de base de linux et de la distribution que nous utilisons au laboratoire. Nous supposons dans la suite que le lecteur a une certaine familiarité avec linux ¹.

Nous nous plaçons dans les conditions du laboratoire. La distribution utilisée est une **Debian sarge** mais la majorité de ce que nous dirons ici (excepté tout ce qui concerne le système de packages) peut être transposée à d'autres distributions.

Dans la suite, nous ne parlerons plus des **opérations élémentaires et obligatoires à faire en entrant dans le labo**. Mais qu'est-ce donc ?

Contrôles Au laboratoire, nous sommes plusieurs à utiliser les machines, il est donc opportun de faire une série de petits contrôles afin de s'assurer que la machine est *en état*. Plic ploc, je contrôlerais

↪ l'intégrité de la machine ²,

↪ la connectique,

Le câble réseau est-il correctement branché aux deux extrémités ? N'y a-t-il pas de **boucles** sur le réseau ?

↪ la configuration du réseau,

¹Le lecteur, toujours lui, prendra bonne note du flou de cette phrase .. une certaine familiarité ?

²Là, il n'y a pas de problème.

Le réseau est-il correctement configuré ? Il faut configurer la carte réseau sur laquelle vous avez connecté le câble réseau³, lui donner la bonne adresse et le bon masque (*via* la commande `ifconfig`) en fonction de la machine (voir *topologie du local*), configurer la route par défaut (*via* la commande `route`) et tester le réseau *via* les commandes (au choix) `ping`, `traceroute`, ..

↪ la présence des paquets dont j'ai besoin (ceci peut se faire au fur et à mesure)

5.2 Le système de packages Debian

Chaque distribution possède un système de gestion de *packages*. Ce système permet à l'administrateur du système d'installer des logiciels. Il gère, normalement, les dépendances. En effet, un programme donné peut avoir besoin d'autres programmes ou de certaines bibliothèques afin de pouvoir fonctionner convenablement. Il me semble normal que le système de gestion des paquets gère les "dépendances". C'est-à-dire qu'il installe tout ce qu'il faut pour qu'un logiciel donné fonctionne.

Suivant les distributions, les commandes permettant de gérer ses paquets sont différentes. Nous nous intéressons au système Debian. Quel que soit le système de gestion, il doit être possible d'installer un paquet mais aussi de savoir faire des recherches dans une base afin de **trouver** le paquet adéquat.

Debian propose deux manières de faire non graphique. Il existe bien entendu divers outils graphiques qui dépendent cette fois de la distribution et de l'environnement X choisi. Nous ne les détaillons pas ici.

La première et la plus ancienne est basée sur les commandes `apt-get` et `apt-cache`, elles s'utilisent comme suit,

↪ `$ apt-cache search motClé`

Permet de chercher dans la base de données des paquets installables le logiciel qui va bien.

↪ `# apt-get install nomPaquet`

Installe le logiciel et les bibliothèques et/ou autres paquets éventuels.

↪ `$ apt-cache show nomPaquet`

Donne des infos sur un paquet.

↪ `# apt-get update`

Permet de mettre la base de données des paquets disponibles à jour (à faire régulièrement).

³Ça a l'air bête ainsi mais ...

La seconde se base sur la commande `aptitude`. Si elle n'est pas présente, vous pouvez l'installer (`apt-get install aptitude`).

↪ `aptitude search motClé`

↪ `aptitude install nomPaquet`

↪ `aptitude show nomPaquet`

↪ `aptitude update`

↪ `aptitude upgrade`

5.3 gcc and cie

5.3.1 gcc

Pour pouvoir compiler ... il faut un compilateur. La manière la plus simple d'installer un compilateur ... et ce qu'il faut comme bibliothèques est de recourir à **tasksel**⁴. Lancer `tasksel` et cocher 'developpement'.

Avant ça, vous pouvez vérifier votre système

↪ en essayant la commande `gcc`,

↪ en compilant un *Hello world*,

↪ en regardant la version de `gcc` (c'est un lien soft, il suffit de regarder vers quoi il pointe).

5.3.2 make et son Makefile

Linux fournit une commande `make` permettant de faciliter les tâches de compilation. Cette commande est orientée langage C mais peut servir à tout. Le principe est simple.

`make` se base généralement sur un fichier (devant se trouver dans le répertoire courant). Ce fichier est constitué de *cibles* et de *tâches* associées. A chaque cible, une série d'actions est associée. Nous pourrions écrire par exemple

```
$ cat Makefile
```

```
un:
    gcc un.c -o un
```

et un `make` dans une console compilerait notre programme. Si je veux qu'à chaque modification de mon fichier source et uniquement lorsque mon fichier source est modifié, `make` recompile mon programme, j'écris alors.

⁴C'est à nouveau propre à Debian mais bon

```
$ cat Makefile
```

```
un: un.c
    gcc un.c -o un
```

ainsi `make` sait qu'il doit d'abord résoudre la cible `un.c` avant d'exécuter ses actions.

Il peut, bien évidemment, définir plusieurs cibles. Et demander de résoudre des cibles avant la sienne. Par exemple,

```
$ cat Makefile
```

```
un: un.c
    gcc un.c -o un

run: un
    ./un
```

Lorsque je fais un `make run`, la cible `un` sera d'abord résolue et le programme compilé s'il ne l'était pas et ensuite la cible `run` exécutée ...ainsi que le programme dans ce cas.

Il subsiste un intérêt supplémentaire dans la commande `make`. Celle-ci ayant été écrite afin de compiler des programmes C, elle le fait sans l'intervention d'un `Makefile`. En effet un simple `make foo` aura pour effet d'exécuter la commande `gcc foo.c -o foo ...` normalement. De plus, `make` permet de définir des variables dans son fichier de configuration. Je peux donc définir n'importe quelle variable, par exemple `PGM=un` et écrire dans ma cible `gcc $(PGM).c -o $(PGM).`

Mieux encore je peux définir des variables particulières comprises par `gcc`. Par exemple les variables `CC` précisant le compilateur utilisé et `CFLAGS` les paramètres passés à la commandes. Nous verrons plus loin que nous aurons besoin de lier la librairie `pthread` lors de l'édition des liens de nos programmes. Nous devrions entrer une commande du type

```
$ gcc foo.c -o foo -pthread
```

En lieu et place, nous pouvons écrire un `Makefile` qui sera valable pour plusieurs programmes et pas uniquement pour le programme `foo`.

```
$ cat Makefile
```

```
CC=gcc
CFLAGS=-pthread
```

Pour compiler, il suffira d'écrire

```
$ make foo
```

`make` offre encore moult possibilités mais les bases sont là. Par exemple, j'utilise `make` pour la rédaction de ces notes avec `LATEX`. J'ai diverses cibles suivant que je veux générer un pdf, imprimer ou ...

5.4 root, le superuser

Certaines actions doivent être exécutées avec les privilèges de `root` ... mais pas toutes. Afin d'éviter certaines erreurs, vous travaillerez toujours comme simple utilisateur, appelons-le `user` et vous ne deviendrez `root` que lorsque c'est nécessaire.

Lancez donc votre environnement X en tant que `user`, et travaillez dans **des** consoles en tant que `user`. Gardez toujours (dans un autre écran virtuel) une ou plusieurs consoles `root`.

Dans la suite lorsqu'une commande commence par `#` c'est qu'elle doit être exécutée par `root` et par `$` lorsqu'elle peut être exécutée par `user`.

Chapitre 6

Notions de modules

6.1 Exemple *Hello world*

Comme vu précédemment, pour écrire un premier module, écrivez le code suivant. Il est inutile d'être *root* pour ce faire.

```
$ cat hello.c
```

```
#define MODULE
#include <linux/module.h>

/*
 * Hello world module
 */

int init_module(void) {
    printk("Module_'Hello_world'_'_inserted_'_\\n");
    return 0;
}

void cleanup_module(void) {
    printk("Module_'Hello_world'_'_removed_'_\\n");
}
```

6.2 Compilation et insertion pas à pas

Pour compiler votre programme, utilisez les paramètres suivants :

```
$ gcc -O2 -Wall -D__KERNEL__ -c hello.c
```

Un `man gcc` nous apprend :

- ↪ le commutateur `-O2` fait partie de la famille `-O`, `-O1`, `-O2`, `-O3` demandant à `gcc` d'optimiser le fichier compilé,
- ↪ le commutateur `-Wall` veut dire *Warning all*,
- ↪ le commutateur `-D` définit une macro,
- ↪ le commutateur `-c` indique au programme `gcc` qu'il doit s'arrêter après avoir engendré le fichier objet, et ne pas effectuer la phase d'édition de liens. Le résultat final est un fichier appelé `hello.o`,
- ↪ (si `gcc` ne trouve pas les headers), le commutateur `-I` indique à `gcc` qu'il doit ajouter ce chemin aux chemins par défaut de recherche des fichiers à inclure, si votre système est correctement configuré, cela n'est pas utile.

Le noyau ne dispose pas de sortie standard, aussi ne pouvons-nous pas utiliser la fonction `printf()` ... à sa place, le noyau propose sa propre version de cette fonction, appelée `printk()`, qui fonctionne presque comme la première, à ceci près qu'elle envoie son résultat à un tampon circulaire du noyau. C'est dans ce tampon que tous les messages du système aboutissent, et ce sont en réalité les messages que l'on peut observer en amorçant le système. À tout instant, on peut examiner le contenu du tampon en utilisant la commande `dmesg` ou plus directement, en examinant le fichier `/proc/kmsg` mais dans ce dernier cas, il faut être *root*.

L'insertion du module ne se fait pas *via* la commande `modprobe` (probablement car le module ne se trouve pas dans `/lib/modules/'uname -r'`) mais bien *via* la commande `insmod`, comme suit

```
# insmod hello.o
```

Le module *hello* est inséré dans le noyau, nous savons que la fonction `init_modules` a été exécutée. Allons voir le résultat dans `/proc/kmsg` avec

```
$ dmesg | tail -1
```

```
Module inserted
```

Vérifions que le module est bien inséré dans le noyau avec

```
$ lsmod
```

Module	Size	Used by
...		
hello	1234	0
...		

Et enfin, ôtons le module du noyau grâce à la commande `rmmmod` et vérifions la présence du message associé à la fonction `cleanup_module`,

```
# rmmmod hello
# dmesg | tail 2
Module inserted
Module removed
```

Chapitre 7

Notions de threads

7.1 Exemple *Hello world*

Reprenons l'exemple *Hello world* de la section 3.3 et voyons comment le mettre en oeuvre.

```
$ cat hello.c
```

```
#include <pthread.h>

/*
 * hello.c
 *
 * Hello world example.
 */

/*
 * run
 *
 * method executed when thread create, this method say "Hello"
 */
void* run (void* arg) {
    int id = (int) arg ;

    printf("My_id_is_%d, I say 'Hello'\n", id) ;
    pthread_exit(0);
} // end - run

/*
 * main method
```

```
*
 * I create two thread and I wait they finish to say Hello
 */
int main ( int argc, char* argv[] ) {
    // Create 2 threads
    pthread_t task1, task2 ;

    pthread_create (&task1, NULL, run, (void*) 0 ) ;
    pthread_create (&task2, NULL, run, (void*) 1 ) ;
    pthread_join(task1, NULL) ;
    pthread_join(task2, NULL) ;

    exit(0);
} // end - main
```

Au préalable plaçons-nous dans un répertoire dans lequel nous allons écrire tous nos programmes "thread" et écrivons un petit Makefile.

```
$ cat Makefile
```

```
CC=gcc
CFLAGS=-pthread
```

Le travail étant préparé, tout devient simple. Il suffit d'écrire

```
$ make hello
./hello
```

7.2 Exemples d'utilisation d'un *pipe* et *mutex*

Voici deux exemples, l'un de mise en oeuvre d'un *pipe* comme moyen de communication entre processus et l'autre d'un *mutex* comme moyen de synchronisation. Les deux exemples n'appellent pas de commentaires.

```
$ cat pipe.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>

/*
 * pipe
 */
```

```

* Example of using pipe for communication
* system between process
* I can doing the same thing with named
* pipe (FIFO) ... look mkfifo
*/

int filedes[2];          // File descriptor
                        // 0 = stdin -> read
                        // 1 = stdout -> write

void *read_char() ;
void *check_hit() ;

/*
* Create two thread, on for write, one for read
*/
int main() {
    int rc ;              // return code
    pthread_t tid1, tid2; // thread id

    pipe(filedes);
    /* Create thread for reading characters. */
    rc = pthread_create(&tid1, NULL, read_char, NULL);
    /* Create thread for checking hitting of any
    * keyboard key. */
    rc = pthread_create(&tid2, NULL, check_hit, NULL);

    if (rc == 0) while(1);
    exit(0);
}

/*
* First thread, read
*/
void *read_char() {
    char c = '_'; // character to read

    printf("\nEntering_routine_to_read_character".
           "..._for_quit");
    while ( c!='Q' ) {
        /* Get a character in 'c' except '\n'. */
        printf( "\nReader:_");
        do {

```

```

        c = getchar();
    } while (c == '\n');
    write(filedes[1], &c, 1);
}

void *check_hit()
{
    char c; // caracter read in pipe
    printf("\nEntering_routine_to_check_hit....");
    while(1) {
        read(filedes[0], &c, 1);
        if ( isalnum(c) && c!='Q' ) {
            printf( "\nWriter:_The_key_hit_is_%c\n", c);
            sleep(2) ;
        } else exit(0);
    }
}

```

```

$ make pipe
$ cat mutex.c

```

```

#include <pthread.h>
#include <stdlib.h>
#include <sched.h>
#include "couleurs.h"

/*
* mutex
*
* Example of use of mutex
* I have 100 EUR on my bank account.
*
* To see utility of mutex decomment line 53 and 60
*/

#define N_THREAD 5
#define N_ALEATOIRE 1+(int) (10.0*rand()/(RAND_MAX+1.0))

const char* COULEUR[] = {RED, BLUE, YELLOW, GREEN, LIGHT_BLUE};

pthread_t tids[N_THREAD]; // array of thread
pthread_mutex_t mid; // mutex
int balance; // balance of bank account

```

```

void* get_cash(void*);

int main() {
    int i; // loop indice

    // initialisation rand
    srand(5);

    balance = 100;
    printf("Balance: %d\n", balance);

    pthread_mutex_init(&mid, NULL);
    for (i=0; i<N_THREAD; i++) {
        pthread_create(&tids[i], NULL,
            get_cash, (void*)i);
    } // end - for

    for (i=0; i<N_THREAD; i++)
        pthread_join(tids[i], NULL);
    exit(0);
} // end - main

void* get_cash(void* j) {
    int amount; // amount of money
    int id = (int) j;
    int balance_tmp; // use to "facilitate" scheduling

    while (balance > 0) {
        amount = N_ALEATOIRE;
        //pthread_mutex_lock(&mid);
        balance_tmp = balance;
        printf("%sThread %d, get %d EUR.\n",
            COULEUR[id], id, amount, DEFAULT);
        balance_tmp -= amount;
        sched_yield(); // yield execution to another thread
        balance = balance_tmp;
        printf("%sThread %d, balance: %d\n",
            COULEUR[id], id, balance, DEFAULT);
        //pthread_mutex_unlock(&mid);
        sched_yield(); // yield execution to another thread
    }
}

```

```

} // end - for

} // end - get_cash

$ make mutex

```

7.3 Le problème des philosophes

Comme nous l'avons dit à la section 3.6.1 le problème des philosophes se résoud grâce aux sémaphores et aux *mutex*. Voici quelques éléments de solutions.

- Nous définissons un *thread* par philosophe, la fonction principale aura donc un tableau de 5 *threads*.
- Chaque philosophe à un état, MANGE, PENSE, FAIM. L'état FAIM étant ajouté pour traiter le philosophe bloqué. Nous avons donc également un tableau d'états.
- Nous définissons deux macros GAUCHE et DROITE permettant de connaître l'*id* du voisins, du style $(id+1)\%5$.
- Un *mutex* protégera l'accès au tableau d'état des philosophes.
- Dès qu'un philosophe essaie de saisir deux fourchettes, il positionne son sémaphore. Au retour, il teste son sémaphore. S'il n'a pas les ressources, il se met en attente ... jusqu'à ce qu'un autre philosophe vienne le débloquent.

Voici un extrait de code ... la fonction principale d'un *thread* philosophe, qui peut vous servir d'exemple.

```

void* philosophe_run(void* param) {
    int id = (int) param;

    printf("Le philosophe %d arrive à table\n", id);
    sched_yield(); // rend la main pour le thread suivant
    while (1) {
        penser(id); // le philosophe pense
        prendre_fourchette(id);
        // le philosophe prend 2 fourchettes ou bloque
        manger(id); // le philosophe mange
        poser_fourchette(id); // ... et repose ses fourchettes
    } // end - while
} // end - philosophe_run

```

Après écriture, la compilation ne posera pas de problème, un `make philosophe` devra faire l'affaire.

Chapitre 8

Programmation temps réel, *RTLinux*

8.1 Exemple *Hello world*

Reprenons l'exemple *Hello world* de la section 4.3 et tâchons de le compiler et de l'exécuter. Pour ce faire, nous devons le compiler avec les FLAGS "qui vont bien" et insérer le module dans le noyau. Par facilité, un fichier `rtl.mk` est fourni avec RTLinux (il faudra le localiser) et l'insérer dans notre Makefile ... qui est généralement fournit dans les exemples.

Etape par étape, on fera. Dans un premier temps, créons le Makefile dans le même répertoire que le fichier `hello.c`. Voici un Makefile minimum ...

```
$ cat Makefile
```

```
CC=gcc
include rtl.mk

all: hello.o
```

Après avoir copié (faire un lien soft suffit) le fichier `rtl.mk`, on compile le fichier `hello.c` afin d'obtenir un fichier objet (`hello.o`), comme ceci :

```
$ make
```

Il reste à charger le module dans le noyau. Plusieurs manières de faire sont possibles. On peut le faire "à la main" à l'aide des commandes `insmod` et `rmmmod` ou utiliser le script fournit avec RTLinux `rtlinux`.

rtlinux start hello Charge le module dans le noyau,

rtlinux status hello Vérifie le status du module,

rtlinux help Aide sommaire, pour plus d'aide `man rtlinux`,

rtlinux stop hello Stoppe le program en le déchargeant du noyau.

Lorsque le module est chargé, vous verrez apparaître les messages directement dans la console ou alors en utilisant la commande `dmesg`.

A ce stade, nous avons "écrit", compilé et exécuté notre premier programme RTLinux!

8.2 Exemple d'utilisation d'un *mutex*

Cet exemple est issu de la distribution RTLinux et illustre l'utilisation d'un *mutex*. Trois *threads* sont créés et essaient d'accéder à un *mutex*. Ces *threads* attendent ensuite quelque ms avant de relacher le *mutex*.

Que se passe t'il au niveau des intervalles de temps ?

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

#define NTASKS 3
static pthread_t threads[NTASKS];

static pthread_mutex_t mutex /* = PTHREAD_MUTEX_INITIALIZER */;

static void * start_routine(void *arg)
{
    int ret;
    hrtime_t t;
    hrtime_t t2;
    int sleep = 500000000;
    int nthread = (int) arg;

    struct sched_param p;
    p.sched_priority = 1;
    pthread_setschedparam(pthread_self(),
        SCHED_FIFO, &p);

    rtl_printf("thread_%d_starts_on_CPU%d\n",
        nthread, rtl_getcpuid());
    ret = pthread_mutex_trylock(&mutex);
    rtl_printf("thread_%d:pthread_mutex_trylock_returned_%d\n",
        nthread, ret);

    if (ret != 0) {
        rtl_printf("thread_%d:about_to_pthread_mutex_lock\n",
```

```

        nthread);
    t = gethrtime();
    ret = pthread_mutex_lock (&mutex);
    t2 = gethrtime();
    rtl_printf("thread_%d:pthread_mutex_lock_\
    returned_%d(%d\ns_elapsed)\n",
    nthread, ret, (unsigned) (t2 - t));
}
rtl_printf("thread_%d:is_about_to_sleep_for_%d\ns\n",
    nthread, sleep);
nanosleep (hrt2ts(sleep), NULL);

ret = pthread_mutex_unlock (&mutex);

rtl_printf("thread_%d:pthread_mutex_unlock_returned_%d\n",
    nthread, ret);

return (void *) 35 + nthread;
}

int init_module(void)
{
    int ret;
    int i;
    pthread_attr_t attr;

    pthread_mutex_init (&mutex, 0);

    rtl_printf("RTLinux_mutex_test_starts_on_CPU%d\n",
        rtl_getcpuid());
    pthread_attr_init (&attr);
    for (i = 0; i < NTASKS; i++) {
        /* try to run one thread on another CPU */
        if (i == 1 && rtl_cpu_exists(!rtl_getcpuid())) {
            pthread_attr_setcpu_np(&attr, !rtl_getcpuid());
        }

        ret = pthread_create (&threads[i], &attr,
            start_routine, (void *) i);
        if (ret) {
            rtl_printf("failed_to_create_a_thread\n");

```

```

        return ret;
    }
}

return 0;
}

void cleanup_module(void)
{
    void *retval;
    int i;
    for (i = 0; i < 3; i++) {
        pthread_join (threads[i], &retval);
        rtl_printf("pthread_join_on_thread_%d_returned_%d\n",
            i, (int) retval);
    }
    pthread_mutex_destroy (&mutex);
}

```

Troisième partie

Annexes

Annexe A

Copyright

Copyright © 2002-2006
Pierre BETTENS

La reproduction exacte et la distribution intégrale de ce document, incluant ce copyright et cette notice, est permise sur n'importe quel support d'archivage, sans l'autorisation de l'auteur.

L'auteur apprécie d'être informé de la diffusion de ce document.

*Verbatim copying and distribution of this entire document, including this copyright and permission notice, is permitted in any medium, without author's consent.
The author would appreciate being notified when you diffuse this document.*

E-mail : pbettens@heb.be

Annexe B

Ecriture en couleur

Il est possible d'écrire en couleur dans un terminal linux. Pour ce faire, il suffit d'utiliser les séquences d'échappement qui vont bien.

Par exemple

```
...
printf("\033[31m_Red_\033[0m") ;
printf("\033[31,21m_Underline_red_\033[0m") ;
printf("\033[31,1m_Bold_red_\033[0m") ;
```

Vous pouvez utiliser un fichier de macros afin de faciliter cette écriture.

```
$ cat couleur.h
```

```
/*
 * couleur.h
 *
 * Description : Utiliation des ésquences
 * échappement pour colorier le texte dans une
 * console.
 * Auteur : PbT
 * Sur base d'un script csh étrouv au hasard du net
 * Since : 19/2/04
 */

const char DEFAULT[] = "\033[0m" ;
const char RED[] = "\033[31m" ;
const char GREEN[] = "\033[32m" ;
const char YELLOW[] = "\033[33m" ;
const char BLUE[] = "\033[34m" ;
const char MAGENTA[] = "\033[35m" ;
const char LIGHT_BLUE[] = "\033[36m" ;
```

```
const char BOLD_RED[] = "\033[31;1m" ;
const char BOLD_GREEN[] = "\033[32;1m" ;
const char BOLD_YELLOW[] = "\033[33;1m" ;
const char BOLD_BLUE[] = "\033[34;1m" ;
const char BOLD_MAGENTA[] = "\033[35;1m" ;
const char BOLD_LIGHT_BLUE[] = "\033[36;1m" ;

const char UNDERLINE_RED[] = "\033[31;21m" ;
const char UNDERLINE_GREEN[] = "\033[32;21m" ;
const char UNDERLINE_YELLOW[] = "\033[33;21m" ;
const char UNDERLINE_BLUE[] = "\033[34;21m" ;
const char UNDERLINE_MAGENTA[] = "\033[35;21m" ;
const char UNDERLINE_LIGHT_BLUE[] = "\033[36;21m" ;
```

Annexe C

Open RTLinux installation instructions

Ce document est fournit avec la distribution RTLinux. Il se trouve dans le répertoire doc et se nomme INSTALLATION.

RTLinux installation instructions
version: 0.2.0 (August 4, 2003)
AUTHOR: Artemiy I. Pavlov aka Artemio <artemio at artemio.net>

This document describes how to compile and install RTLinux/Free, a hard-realtime extension to the Linux kernel.

C.1 Préalables

C.1.1 Distribution

You may copy, modify and redistribute this document under the terms and conditions of the GNU General Public Licence version 2 or later.

C.1.2 Requirements

Installing RTLinux requires that you know the basics of compiling a Linux kernel. This does not mean that you have to be a kernel guru, but you must know what to do if something goes wrong.

C.2 Installation instructions

This section describes compiling, installing and running RTLinux

- ↪ Check your gcc version. Do not use gcc 2.96: my experience shows that gcc version 2.96 is ok, but only if you are on a uni-processor kernel. I managed to run RTLinux on a SMP kernel only with kgcc (gcc 2.91). In most cases, gcc 2.91, 2.95 and 3.x will work for sure.
- ↪ Get a clean Linux kernel from kernel.org. Also, please make sure that there is a proper rtlinux patch for the kernel you plan to use. RTLinux patches usually are supplied in RTLinux source tree, but patches that were not included can be found here: <ftp://ftp.rtlinux.at/pub/rtlinux/contrib/hofra> or at other rtlinux mirrors.
- ↪ Get an RTLinux package. We will suppose further that you have rtlinux-3.2-pre3. If you want to have a latest CVS version, simply say:

```
# cvs -d :pserver:anoncvs@canals.disca.upv.es:/var/cvs login
password: anoncvs
# cvs -d :pserver:anoncvs@canals.disca.upv.es:/var/cvs co rtlinux-3.2-pre3
```

- ↪ Unpack the Linux kernel sources in /usr/src/linux. If you unpack the source tree to some other directory, say linux-2.4.18, make sure to make a symlink to it:

```
# cd /usr/src
# ln -s linux-2.4.18 linux
```

- ↪ Unpack RTLinux in /usr/src/rtlinux-3.2-pre3. In the rtlinux sources tree, directory "patches", find the patch for the kernel version you will be compiling.
- ↪ Put the RTLinux patch in /usr/src directory (name the file "rtlinux-patch" for further convenience).
- ↪ Change to Linux kernel sources directory:

```
# cd /usr/src/linux
```

- ↪ Patch the Linux kernel with rtlinux patch:

```
# patch -p1 < ../rtlinux-patch
```

- ↪ Configure your kernel:

```
# make menuconfig/config/xconfig
```

- ↪ Configure your kernel to support your CPU type exactly. This means, if you have a Pentium 4 or a Xeon, specify Pentium 4, not Pentium III or whatever. Enable SMP if you are on a multi-processor machine. Add your hardware drivers and other options as needed. There are several notes for "General" section, however:

- ~ Sometimes switching MTRR helps if you fail to run rtplinux
- ~ Suggested that you disable APM BIOS support
- ↔ Notes for using kgcc (gcc 2.91): in the toplevel makefile (/usr/src/linux/Makefile), change the entry:


```
CC = \$(CROSS\_COMPILE)gcc \\  
to: \\  
CC = kgcc \\  
# make dep
```
- ↔ Check dependencies:


```
# make dep
```
- ↔ Make a compressed kernel image:


```
# make bzImage
```
- ↔ Compile and install modules:


```
# make modules \\  
# make modules\_install
```
- ↔ Install the new kernel:


```
# cp arch/i386/boot/bzImage /boot/rtlinux
```
- ↔ Edit /etc/lilo.conf to have a new entry:


```
image=/boot/rtlinux label=rtlinux root=/dev/sda1 (or /dev/hda3,  
etc., wherever your root partition is) read-only
```

Be sure to leave your old kernel here as well in case this new kernel doesn't boot or doesn't work properly.
- ↔ Update the bootloader:


```
# lilo
```
- ↔ Reboot the system.


```
# reboot \\  
or \\  
# shutdown -r now
```
- ↔ At boot prompt, choose "rtlinux". If it doesn't boot with the new kernel, reconfigure and recompile the linux kernel again, superceeding configuration with "make clean" command.
- ↔ Change to RTLinux sources directory:


```
# cd /usr/src/rtlinux-3.2-pre3
```
- ↔ Configure RTLinux:


```
# make menuconfig/config/xconfig
```

You can leave the defaults here.

- ↔ Check rtl.config file in the rtplinux source directory. Make sure it has:


```
CC="kgcc" \\  
instead of: \\  
CC="gcc"
```

if you are aware of using gcc 2.96 as was written in the beginning.
- ↔ Check dependencies (optional, usual "make" will perform this anyway if you skip this):


```
# make dep
```
- ↔ Make RTLinux kernel and modules:


```
# make
```
- ↔ Make devices and install RTLinux:


```
# make install
```
- ↔ Test RTLinux and it's modules:


```
# make regression
```

This will hang your machine if your Linux kernel or RTLinux was misconfigured and/or miscompiled. All the tests should output [OK].
- ↔ Start RTLinux:


```
# rtplinux start
```

This will output something like this, depending on your rtplinux configuration:

```
(+)rtl.o \\  
(+)rtl\_fifo.o  \\  
(+)mbuff.o\  
...
```

with (+) meaning a successfully loaded module and (-) stating the reverse.
- ↔ Check rtplinux status:


```
# rtplinux status
```

This will show the loaded and unloaded (+/-) modules just as "rtlinux start" did.
- ↔ Stopping rtplinux:


```
# rtplinux stop
```

This will unload all rtplinux modules:

```
(-)rtl.o  
(-)rtl\_fifo.o  
(-)mbuff.o  
...
```

C.2.1 Further information

There is an "RTLinux-HOWTO" that can be found at <http://tldp.org>. It has a good introduction to writing your own rlinux modules.

If you want to discuss RTLinux-related questions, you can join the RTLinux mailing list. To subscribe, visit this page: <http://www2.fsmlabs.com/mailman/listinfo.cgi/rtl>.

C.2.2 Thanks

Thanks very much to everybody at rlinux mailing list, especially to Hofrat, Linus, Nils and Zwane, and also big thanks and respect to all at the linux kernel mailing list

Annexe D

Planning du laboratoires

Voici le planning des laboratoires. Comme tout planning, il est susceptible d'être modifié.

Une version "évolutive" ce trouve sur le wiki (<http://wiki.namok.be>)

Planning

- ↔ Semaine 1 : Familiarisation avec l'environnement du laboratoire, exposé oral, rappel `makefile`
- ↔ Semaine 2 : Notion de module, concepts et mise en oeuvre
- ↔ Semaine 3-4-5 : Notions de threads, concepts et mise en oeuvre
 - ˘ Exemple simple
 - ˘ Mise en oeuvre d'un mutex
 - ˘ Mise en oeuvre des sémaphores
 - ˘ Le problème des philosophes
- ↔ Semaines 6-7 : Installation de rlinux
- ↔ Semaines 8 : Cours suspendus
- ↔ Semaines 10 : Installation de rlinux
- ↔ Semaine 11,12,13 : Tests, compréhension et présentation des exemples

Références bibliographiques

- [1] Brian W.KERNIGHAN et Denis M. RITCHIE. *Le langage C, norme Ansi*. Masson, Bld Saint-Germain, 120, 75280 Paris, 2^e édition, 1997. ISBN : 2 225 83035 5.
- [2] F Heymans et DL Parnas PJ Courtois. *Concurrent control with Readers ans Writers*. Commun of the ACM, vol10, octobre 1971.
- [3] Andrew TANENBAUM. *Systèmes d'exploitation. Systèmes centralisés. Suystèmes distribués*. Dunod / Prentice Hall, 3^{ème} édition, 1994. ISBN : 2 10 004554 7.

Webographie

- [4] FSMLabs. Rt linux, case studies. <http://www.fsmlabs.com>.
- [5] Dave MARSHALL. Programming in c. unix system calls and subroutines in c. <http://www.cs.cf.ac.uk/Dave/C>.
- [6] Ismael RIPPOLL. Linux temps réel (rt- linux). <http://www.linuxfocus.org/Francais/May1998/article44.html>.
- [7] The community. Wikipédia, l'encyclopédie libre. <http://www.wikipedia.fr>.
- [8] Pierre BETTENS. Site perso à destination des étudiants de l'esi, 2000. <http://esi.namok.be>.
- [9] Pierre BETTENS. Wiki des sites namok.be, 2003. <http://wiki.namok.be>.

Table des matières

I	Présentation des concepts	1
1	Préalables	2
1.1	Présentation	2
1.2	Implication en terme d'OS	3
2	Notions de modules	5
2.1	Définition	5
2.2	Mise en oeuvre	6
2.3	Exemple <code>Hello world</code>	6
3	Notions de threads	8
3.1	Définition	8
3.2	Mise en oeuvre	9
3.3	Exemple <code>Hello world</code>	10
3.4	Synchronisation entre <i>threads</i>	11
3.4.1	Sémaphores	11
3.4.2	Sémaphore d'exclusion mutuelle, <i>mutex</i>	13
3.4.3	<i>Conditions variables</i>	14
3.5	Communication entre <i>threads</i>	15
3.5.1	Pipe	15
3.5.2	<i>Fifo</i> ou pipe nommé	17
3.6	Problèmes classiques	18
3.6.1	Le problème des philosophes	19
3.6.2	Le problème des lecteurs / rédacteurs	20
3.6.3	Le problème du coiffeur endormi	20
4	Programmation temps réel, <i>RTLinux</i>	22
4.1	Présentation	22
4.2	Structure de base d'un programme temps réel	24
4.3	Exemple <code>Hello world</code>	25
4.4	Communication entre tâches, les <i>fifos</i> temps réels	26

II	Manipulations au laboratoire	28
5	Mise en place de l'environnement de travail	29
5.1	Préalable	29
5.2	Le système de packages Debian	30
5.3	<code>gcc</code> and cie	31
5.3.1	<code>gcc</code>	31
5.3.2	<code>make</code> et son <code>Makefile</code>	31
5.4	<code>root</code> , le <i>superuser</i>	33
6	Notions de modules	34
6.1	Exemple <code>Hello world</code>	34
6.2	Compilation et insertion pas à pas	34
7	Notions de threads	36
7.1	Exemple <code>Hello world</code>	36
7.2	Exemples d'utilisation d'un <i>pipe</i> et <i>mutex</i>	37
7.3	Le problème des philosophes	41
8	Programmation temps réel, <i>RTLinux</i>	42
8.1	Exemple <code>Hello world</code>	42
8.2	Exemple d'utilisation d'un <i>mutex</i>	43
III	Annexes	46
A	Copyright	47
B	Ecriture en couleur	48
C	Open <i>RTLinux</i> installation intructions	50
C.1	Préalables	50
C.1.1	Distribution	50
C.1.2	Requirements	50
C.2	Installation instructions	51
C.2.1	Further information	54
C.2.2	Thanks	54
D	Planning du laboratoires	55