Université de Metz

Nom :________________________ **Prénom :**________________________

Groupe :________________________

T.P. de Système d'Exploitation Unix

M.S.T. Télécom 2^e^Année

Année Universitaire 2004/2005

Y. Morère

Table des matières

1 Introduction **4**

2 Qu'est-ce que le multi-threading ? **4**

3 Le multithreading **5**
3.1 Les threads en théorie : Les processus et les threads 5
3.2 Les threads en théorie : threads versus processus 6
3.3 Les threads en théorie : implémentation des threads 6
3.3.1 Les états d'un thread 6
3.3.2 Threads liés, multiplexés et green threads 7
3.3.3 Les attributs des threads 9
3.4 Le cas de Linux 2.4 10
3.5 La librairie NTPL (Ingo Molinar et Ulrich Drepper) et les modifications apportées au noyau Linux 10
3.5.1 Les nouveautés de la NTPL (Ulrich Drepper, 2003) 11
3.5.2 Les modifications relatives dans le noyau Linux 11
3.5.3 Ce qu'il manque pour un «100% POSIX compliant» 11

4 L'API des *pthreads* (*POSIX threads*) **12**
4.1 Opérations classiques et avancées 12
4.1.1 Gestion des threads 12
4.1.2 Annulation d'un thread 13
4.1.3 Ordonnancement des threads 13
4.1.4 Réception des signaux 13
4.1.5 Duplication de processus 14
4.2 Synchronisation 14
4.2.1 Les mutex : Opérations sur un mutex 14
4.2.2 Les mutex : Gestion des mutex 14
4.2.3 Les mutex : Partage d'un mutex 14
4.2.4 Les mutex : Protocoles pour la résolution des priorités 15
4.2.5 Les mutex : Configuration de la priorité 15
4.2.6 Les variables de conditions : Gestion classique 15
4.2.7 Les variables de conditions : Utilisation d'attributs 16
4.2.8 Les variables de conditions : Le partage d'une variable de condition 16
4.3 Les données privées 16
4.4 La modification des attributs 16
4.4.1 Gestion de base 16
4.4.2 Attributs detached/joinable 16
4.4.3 Modification de la pile d'un thread 17
4.4.4 Attribut sur l'ordonnancement 17

5 Applications **17**

6 Les bibliothèques de *threads* **18**

7 Comment créer des *threads* sous LINUX ? **18**

8 Partages des données et synchronisation **20**
8.1 Les MUTEX 20
8.2 Les variables de condition 21
8.3 Les sémaphores POSIX 26

9 Mode de création des *threads* : `JOINABLE` ou `DETACHED` **28**

10 Destruction de *thread* : cancellation **30**

11 Excercice 1 **32**

12 Debug d'un programme multi-thread sous LINUX **33**

13 Conclusion et bibliographie **34**

Les buts fixés pour cette série de tps sont les suivants :
- Se familiariser avec la programmation multi-threading.
- Comprendre les notions vues en cours (exclusion mutuelle, sémaphore).
- Utiliser la language `C` en programmation système.

Évaluation et notation :
- Le ou les compte-rendus comporteront les commandes saisies, les résultats obtenus ainsi que les réponses aux questions.

1 Introduction

Ce TP est issu d'un article de Pierre Ficheux (`pficheux@com1.fr`). Il s'agit d'une introduction à la programmation multi-threads sous LINUX. Les exemples de programmation utilisent la bibliothèque LinuxThreads disponible en standard sur la majorité des distributions LINUX récentes.

2 Qu'est-ce que le multi-threading ?

Les programmeurs LINUX et plus généralement UNIX sont depuis longtemps habitués aux fonctionnalités multi-tâches de leur système préféré. Tous ceux qui se sont frottés un tant soit peu à la programmation système savent qu'il est aisé sous UNIX de créer des processus fils à partir d'un processus existant en utilisant l'appel système `fork`, comme le montre le petit exemple de code ci-dessous : **fork.c**

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

int i;

main (int ac, char **av)
{
  int pid;

  i = 1;

  if ((pid = fork()) == 0) {
    /* Dans le fils */
    printf ("Je suis le fils, pid = %d\n", getpid());
    sleep (2);
    printf ("Fin du fils, i = %d !\n", i);
    exit (0);
  }
  else if (pid > 0) {
    /* Dans le pere */
    printf ("Je suis le pere, pid = %d\n", getpid());
    sleep (1);

    /* Modifie la variable */
    i = 2;
    printf ("le pere a modifie la variable a %d\n", i);

    sleep (3);
    printf ("Fin du pere, i = %d !\n", i);
    exit (0);
  }
  else {
    /* Erreur */
    perror ("fork");
    exit (1);
  }
}
```

Taper ou récupérer le programme `fork.c`.

Pour le compiler ,utiliser la commande `gcc`, par exemple (premier exercice) : `gcc fork.c -o exo1` ou `gcc -Wall fork.c -o exo1`. Pour plus de détail taper la commande `man gcc`.

Que se passe-t-il ? Pourquoi ? Quelles sont les limites du `fork` ?

Réponse :

la création d'un nouveau contexte est pénalisante au niveau performances. Il en est de même pour le changement de contexte (context switch), lors du passage d'un processus à un autre.
Un *thread* ressemble fortement à un processus fils classique à la différence qu'il partage beaucoup plus de données avec le processus qui l'a créé :
- Les variables globales,
- Les variables statiques locales,
- Les descripteurs de fichiers (file descriptors).

Le multi-threading est donc une technique de programmation permettant de profiter des avantages (et aussi de certaines contraintes) de l'utilisation des *threads*.

3 Le multithreading

Cette partie est tirée d'un article d'Éric Lacombe (tuxico@free.fr) du magazine Linux France Magazine du mois de Juillet/Août 2004.
Le multithreading est l'alternative à la programmation multiprocessus. Il offre un parallélisme plus léger à gérer pour le système.
Les threads créent un parallélisme intra-processus et facilitent le partage des tâches et leur communication au sein d'un processus.

3.1 Les threads en théorie : Les processus et les threads

Un thread est un fil ou flot d'exécution. Lors de la création d'un processus, un thread est créé : c'est lui qui contiendra les informations nécessaires à l'exécution du programmedont le code se situe dans l'espace d'adressage du processus.
Quand un processus n'est pas multithreadé, il n'a la possibilité que de s'exécuter séquentiellment. Il ne possède donc qu'un seul compteur ordinal , une seule pile de données temporaires, (adresse de retour, variables temporaires...) et une pile système (cas de linux) allouée par le noyau dans son espace d'adressage qui lui sert au passage de paramètres des appels systèmes effectués par le processus appelant.
Un processus contient également une section de données réservée aux variables globales ainsi qu'un *tas* (structure de données arborescente classique) utilisé pour les variables créées dynamiquement.
Tout processus contient également des informations de contrôle réunies dans une structure typiquement appelée PCB (Process Control Block). Cette zone contient l'état du processus, un compteur d'instructions, le contenu des registres de l'UC pour pouvoir les restaurer lors de l'ordonnancement. En plus de ces informatons on trouve des informations relatives aux entrées/sorties, à la gestion mémoire, à l'ordonnancement et à la comptabilistion des ressources conssomées.
Dasn un processus multithreadé, plusieurs flots d'exécution se partagent les mêmes ressources (descripteurs de fichiers, espaces d'adressage des variables...). Cependant pour permettre l'exécution simultanée de ces flots, chacun possède un identifiant unique, une pile d'exécution propore, des registres (pointeur de pile...), un état...
Chaque thread possède également des données privées (dans la norme POSIX).

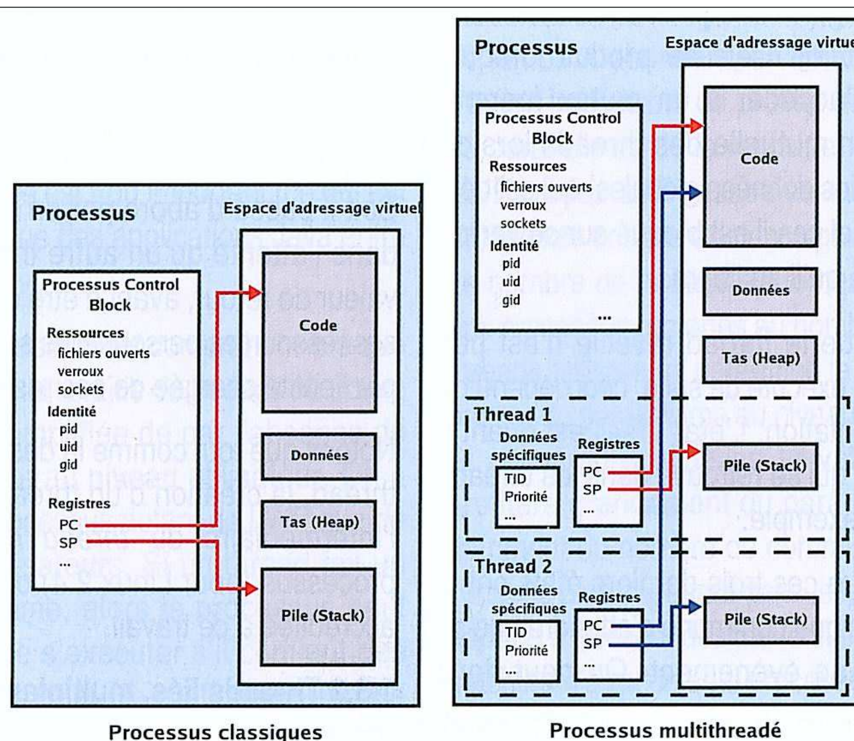


Fig. 1 – Processus classique et multithreadé

On peut résumer en disant qu'un thread est une entité d'exécution , rattachée à un processus, chargée d'exécuter une partie du processus. Ce dernier étatn vu comme un ensemble de ressources (espace d'adressage, fichiers, périphériques...) que ses threads se partagent.

3.2 Les threads en théorie : threads versus processus

Les threads apportent un certain nombres d'avantages quand il s'agit de developper des applications parallèles. Il est plus aisé etplus rapide de créer ou de détruire des threads. Sur une machine SMP (Symetric Multi Processor), plusieursthreads peivent s'exécuter en simultané sur différents processeurs. Les parties indépendantes des applications peuvent être avantageusement implémentées dasn des threads différents.
Quand un thread est bloqué, par exemple à cause d'une attente d'E/S, l'exécution peut être rtansférée à un autre thread de la même application (au lieu de passer à un autre processus), permetant de profiter pleinement du *timeslice* (tranche de temps) accordé au processus par le *scheduler* (ordonnanceur).
De plus les threads d'un même processus partageant le même espace d'adressage peuvent communiquer entre eux sans faire appel au kernel (gain de temps par rapport à la communication entre processus).
Ce dernier point fait apparaitre la nécessité de synchroniser l'accès aux ressources pour éviter la corruption des données. Il est alors nécesaires d'utiliser des verrous. Il est à noter que ces verrours sont d'autant plus efficcace qu'il ne nécessitent pas l'intervention du kernel.

3.3 Les threads en théorie : implémentation des threads

La prise en charge des threads dans le système peut varier selon l'implémentation retenue par les concepteurs . La conformité avec POSIX peut être obtenue sans qu'il en dépende de l'implémentation. Cette norme impose uniquement l'API : le systèmes est donc vu comme une boîte noire par POSIX.

3.3.1 Les états d'un thread

Quel que soit le modèle de multithreading, on retrouve toujous les mêmes états principaux pour les threads. Ils permettent, tout comme ils le font pour les processus, de renseigner l'ordonnanceur pour qu'il sache quelle tâche lancer.

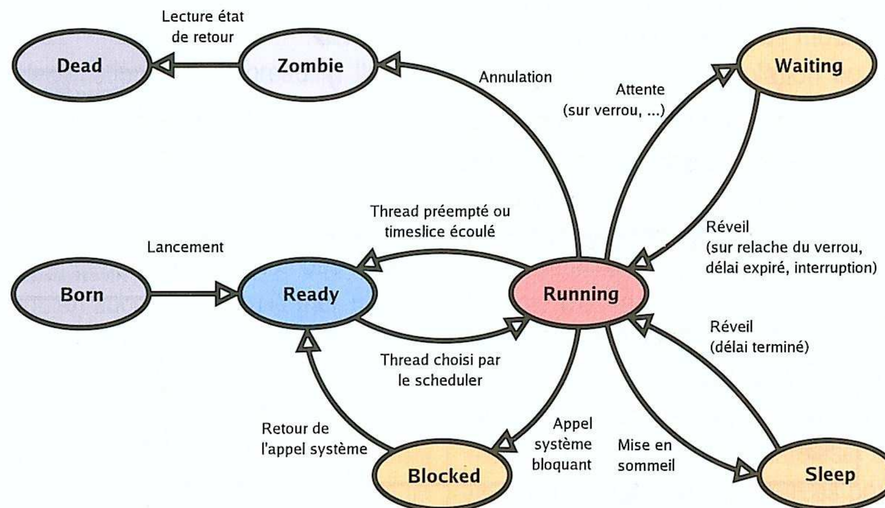


FIG. 2 – État d'un thread

On retrouve pour chaque thread trois états clés : `running`, `ready` et `blocked`. L'état `suspend` que peut avoir un processus n'existe pas (l'état `waiting` est une suspension mais n'a pas la même sémantique) ; en effet suspendre un unique thread revient à suspendre l'exécution cd tous les threads d'un même processus, puisque celui-ci se retrouve *swappé* pour laisser la place à d'autres processusen mémoire vive.

L'état `running`est attribué à un thread en cours d'exécution, c'est à dire quand il s'exécute réellemet sur le processeur.

Il faut tout de mêm faire attention, car il reta possible que l'état d'un thread soit différent de celui de son processus suivant l'implémentation multithreading retenue (cas des *green threads*).

Tout thread pouvant être choisi par l'ordonnanceur est dans l'état `ready`.

L'état `blocked` intervient lorsqu'un thread fai tun appel d'E/S, par exemple, ou un quelconque autre appel système, ou que sa rtanche de temps impartie est écoulée. Suivant le modèle utilisé, ce *timeslice* est calculé : soit au niveau de l'ordonnancement au sein des processus, auquel cas il s'agit d'une décision de l'espace utilisateur ; soit dans l'espace du noyau.

Il se peut également qu'un threadsoit dans l'état `waiting`. Cela se produit lorsqu'il essaie d'accéder à un mutex (permet l'exlusion mutuelle des threads lors de l'accès à des données globales) qui est déjà pris auquel il est bloquésur ce verrou jusqu'à ce qu'il se libère.

Notons que le thread réveillé n'est pas forcément exécuté de suite. Ceci dépend de l'implémentation. L'état `sleep` est un état qui se retrouve dans les threads Java par exemple.

Ces trois derniers états ont la caractéristique d'attendre qu'un évènement se produise.

Lorqu'un thread parvient a son terme ou lorsqu'il est tué par un autre thread, deux situations peuvent se produire. Soit ses ressources sont libérées directement par l'entité en charge (en général, il s'agit du kernel, mais dans le cas des noyaux linux 2.4, un *thread manager* associé au processus relatif au thread à supprimer se charge de la libération). Ce cas n'est possible que si le thread possède l'attribut `detached`. Soit il passe d'abord par un état *zombie*, dans l'attente qu'un autre thread lise sa valeur de retour, avant d'être détruit et que ses ressources personnelles soient libérées par l'entité chargée pour ces responsabilités.

Tout comme la destruction d'un thread, la création se fait par l'intermédiairedu *thread manager* du processus ou par l'entité accréditée à ce travail.

3.3.2 Threads liés, multiplexés et green threads

On considère généralement trois modèles de multithreading. Leurs différences sont fonctions de la prise en charge plus ou moins importante du kernel. Les threads ont toujours une existence dans l'espace utilisateur, ils sont reliés (en fonction du modèle) à des threads systèmes qui sont l'entité de base du *scheduler*.

Ce sont ces threads qui vont être multiplexés sur les différents processeurs. La liaison entre ces deux sortes de threads se fait par le biais des LWP (Light-Weight Process).

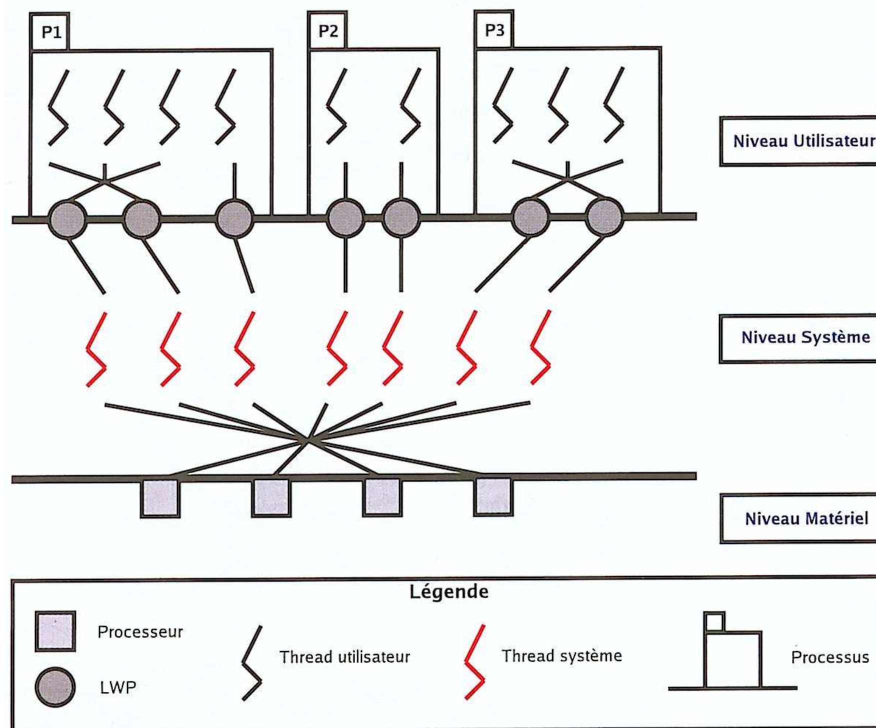


FIG. 3 – État d'un thread

Les threads liés Analysons le modèles des threads liés (figure 3 P2). C'est l'implémentation la moins complexe et s'avère être un choix judicieux lorsque l'ordonnanceur système est de complexité constante ($O(1)$). Il s'agit d'ailleurs du modèle retenu pour linux dans la série des 2.5.x et 2.6.x, l'ordonnanceur aynat été grandement modifié et etant dorénavant en ($O(1)$).
Il s'agit tout simplement d'assoscier à un thread utilisateur un thread unique système. On qualifie souvent ce modèle de 1-à-1. L'importantce d'voir un ordonnanceur de complexité constante se comprend car le nombre de threads systèmes est exactement égal au nombre de threads utilisateurs ; par conséquent, on évite l'écroulement du systèmes si le temps mis par l'ordonnanceur à faire son choix est indépendant du nombre de threads systèmes. Si ce n'est pas le cas, le systèmes des threads liés mis en place se trouve fortement affecté lorsqu'un nombre de threads créés est trop important (ce qui est caractéristique des applications Java en autres).
Dans ce modèle, il est compréhensible que la gestion de la librairie de threads se trouve fortement simplifiée de par l'absence du multiplexage au niveau utilisateur . On a donc par processus autant de LWP que de threads utilisateurs. Si un thread fait un appel système, alors le processus peut continuer de s'exécuter s'il contient des threads à l'état `ready` et que sa tranche de temps impartie n'est pas écoulée. Le cas échéant , seul le LWP associé au thread ayant effectué l'appel système est alors bloqué, laissant libre cours aux autres LWP de s'exécuter à sa place au sein du même processus. Le blocage dû aus apples systèmes se fait au niveau des threads. Notons également que les routines du kernel peuvent être multithreadées. Ce modèle est adopté par Linux 2.4 et 2.6 et Solaris 8.

Les threads multiplexés Le modèle des threads multiplexés (Figure 3 P3) est le plus compliqué à mettre en œuvre, il s'agit du modèle retenu dans Solaris 2 et Windows XP entre autres. L'ordonnancement ici se fait à deux niveaux. En effet, au niveau système, les LWP (associés au thread système) sont multiplexés et on ordonnance aussi les threads utilisateurs sur les LWP. On obtient ici une flexibilité accrue, mais on perd en efficacité. Un autre intérêt vient de la possibilté d'avoir une adaptation de la politique d'ordonancement au niveau utilisateur en fonction de l'application considérée. Les temps dus aux changements de contexte lors de l'ordonnancement des threads sont fortement réduits ; en effet , la majeure partie de la gestion peut se faire dans l'espace utilisateur (s'il n'y a pas d'ajout de LWP au processus courant), le multiplexage des threads sur les différents LWP n'est pas connu du kernel, et donc il n'y a pas de changement de contexte de sa part (pas de modification de registres...), d'où ule gain de

temps. Cependant, en contre-partie, il s'avère impossible de profiter entièrement du parallélisme de la machine, car seul le LWP ou les LWP associés aux threads multiplexés ont une existence pour le kernel. Le nombre de threads systèmes accordés aux processeurs ramenés au nombre total de threads utilisateurs, détermine la capacité à profiter du parallélisme au niveau *hardware*. Si ce nombre avoisine la valeur 1, on profitera grandement du parallélisme au détriment du nombre de commutations de contexte plus important dans le kernel. Tout dépend à quoi est destiné le système. Si la machine n'a qu'un seul processeur, alors il paraît avantageux d'opter pour peu de LWP par processus.
Dans Solaris 2 (entre autres) le kernel dispose d'un *pool* (bassin) de threads (systèmes) qu'il accorde au processus en fonction des demandes de la librairie. Si tous les LWP sont bloqués et qu'un thread est prêt à l'exécution , un autre LWP est fourni au processus pour exécuter ce thread. Les LWP inutilisés pendant un certain temps sont éliminés. La bibliothèque se charge donc d'ajuster dynamiquement le nombre de LWP fournis au processus.

Les threads liés et multiplexés Il est également possible d'avoir des processus hybrides (figure 3 P1) dans ce modèle, c'est à dire contenant à la fois des threads liés et des threads multiplexés.

Les Users Level Threads ou Green Threads Le modèle des *Users Level Threads* ou encore *Green Threads* permet une utilisation sur des systèmes ne comportant aucune fonctionnalité favorisant l'implémentation du multithreading. Ils sont implémentés uniquement dans un espace utilisateur, d'ou une portabilité accrue. Ils ne sont pas connus du noyau, leur existence est simulée uniquement dans une bibliothèque. Toutes la gestion des threads ne requiert aucun appel au kernel, d'où un gain de temps. Cependant , il s'agit là d'une maigre compensation, car il est alors impossible d'assigner à différents processeurs différents threads du même processus simultanément. En outre si un threads fait un appel système, il y a de grandes chances que la kernel passe la main à un autre processus, car il n'a pas la connaissance de la segmentation en threads du processus. Par conséquent, il modifiera l'état du processus si celui-ci effectue un appell d'E/S par exemple, et fera appel au *scheduler* pour décider à qui passer la main. Le thread aura quant à lui toujours un état `running`, son état est donc indépendant de l'état du processus. On perd ici la possibilité de profiter pleinement du *timeslice* accordé au processus.
Notons que la commande `ps -L` permet de voir l'ensemble des LWP alloués par le kernel aux différents processus. On constate alors aisément que le nombre de threads sous linux correspond au nombre de LWP : on se trouve bien en présence d'un modèle 1-à-1.

3.3.3 Les attributs des threads

Divers attributs permeteent de modifier le comportement des threads, on se limitera ici aux attributs POSIX.
Començons par l'attribut de priorité d'un threads (pas encore supporté par le kernel 2.6.x qui manque de support approprié pour le temps réel), qui se retrouve dans les systèmes temps réel, dans lequels on attribue à des tâches critiques une priorité forte pour qu'elles puissent *préempter* les tâches de criticité moindre.
Il existe aussi des attributs modifiants le comportement de l'annulation ou de la suppression d'un thread. Nous avons vu qu'il était possible de détacher un thread du contexte, dans le sens ou sa terminaison n'entraîne pas un passage à l'état *zombie*, c'est à dire qu'aucun autre thread du même processus ne peut accéder à la valeur de retour du thread détaché. Ceci peut être intéressant lorsque des tâches n'ont pas besoin de se synchroniser à leur fin. La libération des ressources peut se faire dès la terminaison d'un thread détaché. Les deux attributs complémentaires associés sont `PTHREAD_CREATE_DETACHED` et `PTHREAD_CREATE_JOINABLE`.
Un thread peut clairement annoncer son refus d'être annulé à ses dépends (attribut `PTHREAD_CANCEL_DISABLE`) ou accepter toute demande d'annulation (`PTHREAD_CANCEL_ENABLE`). Le cas échéant, il reste possible de différer (`PTHREAD_CANCEL_DEFFERED`) cette annulation jusqu'à un point d'annulation (4 fonctions POSIX et certains appels systèmes qui peuvent être bloquant comme `read()`), évitant ainsi qu'un thread se voit tué dans une situation critique (accès en écriture à une variableglobale par exemple). Si les fonctions mises en jeu dans le corps du thread le permettent, on pourra opter pour un comportement asynchrone (`PTHREAD_CANCEL_ASYNCHRONOUS`) lors de l'annulation d'un thread, autrement dit d'une annulation immédiate.
Les autres attributs des threads correspondent à leur ordonnancement. On peut agir tout d'abord sur la politique d'ordonnancement : `SCHED_FIFO` (ordonnancement de type *premier arrivé, premier servi*),

`SCHED_RR` (*Round Robin* ou tourniquet, *i.e.* à temps partagé) et `SCHED_OTHER` (varie selon la bibliothèque de thre ads ou le noyau utilisé).

L'ordonnancement peut être ensuite effectué sur l'ensemble des threads des processus, auquel cas il a une portée système (`PTHREAD_SCOPE_SYSTEM`), ou au sein du processus (`PTHREAD_SCOPE_PROCESS`). Ceci dépend fortement de la bibliothèque de multithreading utilisée : par exemple LinuxThreads comme NTPL(Native POSIX Thread Library) ne supporte pas `PTHREAD_SCOPE_PROCESS`. En effet, il n'existe pas d'ordonnancement au niveau utilisateur. L'attribut sur la portée n'a donc de sens que sur des systèmes où le multithreading est hybride. Le cas échéant, les priorités des threads sont considérés uniquement au sein d'un processus ou au regard des priorités de tous les threads du système.

Finalement on peut choisir d'hériter de la politique d'ordonnancement du threads père (`PTHREAD_INHERIT_SCHED`) à n'importe quel moment de l'exécution et de revenir ensuite sur sa propre configuration (`PTHREAD_EXPLICIT_SCHED`).

Notons également la possibilité de modifier la taille maximale de la pile associée à un thread via l'attribut `stacksize`.

3.4 Le cas de Linux 2.4

Dans cette version, Linux ne propose pas de système de multithreading efficace. Des lacunes dans le noyau entrainaient une limitation pour l'écriture d'une bibliothèque compatible POSIX. L'appel système `clone()` servant à la création de nouveaux processus a depuis été modifiée facilitant l'implémantatin du multithreading.

Le système mis en place est un modèle 1-à-1, autrement dit les processus sont composés de threads liés uniquement. De plus, le manque de support de la part du noyau oblige à l'utilisation d'un *thread manager* au sein de chaque processus. Il a pour rôle de créer et de détruire les nouveaux threads, assure aussi l'implémentation correcte de la sémantique des signaux, ainsi que d'autres parties de gestion. Mais il ajoute une lourdeur au système, et ralentit le processus de création et de destruction.

L'absence de moyen pour la synchronisation au sein du noyau amène à l'utilisation des signaux dans l'implémentation des threads.

Ce pendant la gestion des signaux au sein de la bibliothèque de threads est fragile et non conforme POSIX. Ceci est dû à l'absence de concept de groupes de threads dans le noyau.

L'ABI (Application Binary Interface) de ELF (le format binaire couramment utilisé sous linux), n'est pas prévue pour le stockage des données privées aux threads.

Pour remédier à cela, des relations fixes entre le pointeur de pile et la position du descripteur de thread sont utilisés. A ceci s'ajoute un nombre limité de threads.

Concluons sur les problèmes de cette implémentation :

- Le *thread manager* est un goulot d'étranglement pour la création et la destruction de threads ; de plus le nettoyage du processus appartient à ce thread qui, s'il se trouve tué, ne peut plus agir.
- On ne peut pas envoyer un signal à un processuscomme un tout (chaque thread à un PID différent).
- L'utilisation des signaux comme primitives de synchronisation aboutit à de gros problèmes. Des réveil hasardeux peuvent se produire, ce qui engendre une pression supplémentaire sur le système de gestion des signaux du kernel.
- `SIGSTOP` et `SIGCONT` stoppent uniquement un thread et non l'ensemble du processus.
- Chaque thread possède un PID différent (problème avec les signaux). Une limite de (8192-1) threads sur l'IA32 (*Intel Architecture*. Le système de fichier `/proc` devient dur à utiliser, si trop de threads sont présents.
- Le manque d support du noyau empêche l'implémentation correcte des signaux. En outre, leur utilisation est une approche lourde.

3.5 La librairie NTPL (Ingo Molinar et Ulrich Drepper) et les modifications apportées au noyau Linux

Le but de cette nouvelle bibliothèque est d'assurer une compatibilité POSIX, une utilisation efficace des systèmes multiprocesseurs (SMP), des coûts de création de threads faibles, une compatibilité binaire maximale avec l'implémentation Linux Threads, un coût administratif non proportionnel au nombre de processeurs utilisés, le support NUMA (*Non-Uniform Memory Architecture*), etc...

Notons également l'intégration dans C++ pour la gestion eds exceptions où le nettoyage des objets s'apparente à la destruction de threads.

3.5.1 Les nouveautés de la NTPL (Ulrich Drepper, 2003)

Le modèle de threads choisis est du 1-à-1. La fléxibilité d'un modèle plusieur-à-plusieurs (hybride ou multiplexé) vient avec une complexité de l'implémentation du système.
De plus l'amélioration de l'ordonnanceur (complexité en temps constante) permet de rendre tout à fait efficace un système basé uniquement sur des threads liés. La gestion des signaux avec un système multiplexé aurait pu être réalisé au niveau utilisateur, cependant on aboutirait à une complexité et des retards dans la gestion des signaux.
NTPL en finit donc avec la limitation du nombre de threads, et la gestion des signaux problématiques.
Cette dernière tâche incombe au noyau, qui doit gérer la multitude de masques de signaux. Il s'ensuit quand même la suppression des retards dans la délivrance des signaux (un isgnal n'est émis qu'à un thread non bloquant).
NTPL supprime l'existence du *thread manager* (le noyau assure maintenant la désallocation des ressources des threads, ainsi qu'un support correct des signaux POSIX).
Un autre problème du *thread manager* est qu'il ralentit la création des threads. En effet, toute requête de création est traitée par lui de manière obligatoirement séquentielle, car il ne peut s'exécuter que sur un seul processeur. De plus,trop de responsabilités lui sont données, ce qui augmente de manière significative la durée d'un changement de contexte.
Les primitives de synchronisation sont désormais implémentées à l'aide de FUTEX (*Fasr User muTEX*), qui est une nouvelles fonctionnalité ajoutée au noyau (au cours de la série 2.5.x). Le mécanisme, bien que simple, permet l'implémentation de toutes ces primitives. Les threads appelants bloqués peuvent être réveillés à l'issue d'une interruption ou après un délai. En outre, la gestion de cette synchronisation peut se faire presque dans sont intégralité au niveau utilisateur, d'où un gain de rapidité. Les Futex sont faits pour fonctionner en mémoire partagée, ce qui permet la communication POSIX inter-processus.
Le problème de rendre la création de threads rapide est résolu en stockant les structures de données des threads et leurs données privées sur la pile (une modification de l'ABI de ELF est nécessaire), ainsi qu'en évitatn de libérer directement la mémoire lorsqu'un thread est tué, en vue d'une réutilisation.

3.5.2 Les modifications relatives dans le noyau Linux

Elles furent menées pour la plupart par Ingo Molnar, en vue d'assurer une interface optimale entre la bibliothèque et le kernel. Voici les modifications effectuées :

- Le support d'un nombre arbitraire de zones de données spécifiques à chaque thread pour l'IA32 et X86-64.
- L'appel système `clone` est étendu pour optimiser la création de nouveaux threads et leur destruction sans l'aide d'un autre thread. Le kernel stocke le TID (*Thread IDentifier*) d'un nouveau thread à une adresse mémoire donnée et efface son contenu lorsque le thread est tué. Ceci aide à l'implémentation d'une gestion de mémoire en espace utilisateur sans intervantion du kernel.
- La gestion des signaux POSIX pour les processus multithreadés est supportée par le noyau. Les signaux sont maintenant délivrés à un thread actif du processus, un signal fatal détruit l'intégralité du processus. Les signaux `SIGSTOP` et `SIGCONT` affectent maintenant entièrement le processus.
- L'appel système `exit()` termine désormais le thread appelant, et le nouvel appel introduit `exit_group()`, finit le processus. De plus, le temps mis par cet appel pour stopper un processus avec beaucoup de threads a été grandement réduit.
- L'appel système `exec()` utilise maintenant le même PID que le processus original, et tous les threads présents sont terminés avant que la nouvelle image du pocessus acquière le contrôle.
- Les statistiques sur l'usage des ressources affectent entièrement le processus.
- L'implémentation de `/proc` a étét améliorée pour faire face à de possibles milliers d'entrées générées. Chaque thread a un sous répertoire dans celui de son processus.
- Sur un `exit()`, les threads détachés sont supprimés via un reveil sur Futex par le noyau, ce qui correspond à un «join» du noyau (avant utilisation d'un *thread manager*).
- L'espace des PID a étét étendu à un maximum de 2 milliards sur l'IA32 et `/proc` peut contenir jusqu'à 64000 processus.

3.5.3 Ce qu'il manque pour un «100% POSIX compliant»

Malgré toutes ces améliorations, la compatibilité POSIX n'est pas totalement assurée. En effet quelques défis restent encore à relever. Voici la liste des lacunes :

- La famille des appels systèmes `setiud()` et `setgid()` doivent affecter le processus entier et non seulement le threads appelant.
- Le niveau `nice` est une composante globale du processus.
- La limite d'utilisation du CPU doit être une limite sur le temps que dépensent tous les threads d'un processus ensemble.
- Le manque de support du noyau pour le temps réel empêche NTPL de supporter certaines fonctionnalités. Bien que les appels systèmes pour modifier le scheduling soient présents, ils n'ont pas d'effet garanti car la majorité du kernel ne suit pas les règle de l'ordonnancement temps réel. Par exemple, reveiller un thread sur un Futex se fait sans regard des priorités.

4 L'API des *pthreads* (*POSIX threads*)

Les *pthreads* sont la normalisation IEEE des threads. Cette norme précise le comportement en boîte noire, autrement dit l'implémentation des fonctions peut différer d'une bibliothèque compatibles POSIX à une autre.
En ce qui concerne Linux, la gestion des signaux n'est correctement implémentée que depuis NTPL. Les priorités sur les *mutex* et *variables de conditions* ne sont pas effectives.
En effet, Linux manque encore de support pour le temps réel, par conséquent NTPL ne peut pas supporter ce qui n'est pas pris en charge par le noyau.
Notons toutefois que le partage de ces entités de synchronisation entre différents processus est géré depuis la NTPL de par l'émergence des Futex dans le noyau.

4.1 Opérations classiques et avancées

Pour la compilation d'un programme multithreadé, ajouter sur la ligne du compilateur `-D_REENTRANT` (cela permet un comportment correct de certaines macros), et sur la ligne de commande de l'éditeur de lien ajouter `-lpthread`.
Notons qu'un code est dit réentrant s'il peut être exécuté de manière simultanée par plusieurs tâches sans que ceci engendre des corruptions de données, de blocage, ou tout autre effet non désirable.
Toutes les primitives suivantes sont déclarées dans le fichier d'entête `pthread.h`

4.1.1 Gestion des threads

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

Le TID est renvoyé dans `thread`, on peut également spécifier une structure d'attributs (ou `NULL`) et la routine à exécuter peut disposer d'un argument.

```
void pthread_exit(void *retval);
```

Termine le thread appelant et renvoie `retval`.

```
int pthread_join(pthread_t th, void **thread_return);
```

Attend la fin d'un thread et lit sa valeur de retour.

```
int pthread_detach(pthread_t th);
```

Permet de détacher un thread, il ne peut plus communiquer sa valeur de retour à un autre thread.

```
pthread_t pthread_self(void);
```

Retourne le TID du thread appelant.

```
int pthread_equal(pthread_t thread1, pthread_t thread2);
```

Compare deux TID et renvoie une valeur non nulle s'il sont égaux.

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;

int  pthread_once(pthread_once_t  *once_control,  void  (*init_routine) (void));
```

Assure que la fonction passée en paramètre ne s'exécute qu'une seule fois. Cette routine est typiquement utilisée pour des fonctions d'initialisation comme l'allocation d'une structure globale. Parmi tous les threads appelant `pthread_once` avec la même variable de contrôle, un seul l'exécute. On déclare la variable `once_control` de manière statique.

```
int pthread_kill(pthread_t thread, int signo);
```

Envoir le signal `signo` au thread passé en argument.

```
void pthread_yield(void);
```

Le thread appelant libère le processeur sur lequel il s'exécute, il est placer dans la file `ready` (*runnable*) du scheduler qui va choisir un autre thread à exécuter (cette fonction est une extension GNU).

4.1.2 Annulation d'un thread

```
int pthread_cancel(pthread_t thread);
```

Emet une demande d'annulation au thread passé en argument.

```
void pthread_cleanup_push(void (*routine) (void *), void *arg);
void pthread_cleanup_pop(int execute);
```

Lorsque le thread reçoit une demande d'annulation, les routines empilées sont exécutées. Ces deux fonctions s'utilisent ensemble : on empile une fonction de nettoyage à l'entrée d'une section et on la dépile lorsqu'on sort. Lors d'un *pop*, on peut soit exécuter la fonction empilée (`execute=1`) soit la dépiler sans l'exécuter (`execute=0`).

```
int pthread_setcancelstate(int state, int *oldstate);
```

Le thread peut soit accepter (`PTHREAD_CANCEL_ENABLE`) ou refuser (`PTHREAD_CANCEL_DISABLE`) les demandes d'annulation de la part d'autres threads.

```
int pthread_setcanceltype(int type, int *oldtype);
```

Si on autorise l'annulation, elle peut soit prendre un effet immédiat (`PTHREAD_CANCEL_ASYNCHRONOUS`), soit lorsque l'exécution arrive à un point d'annulation (`PTHREAD_CANCEL_DEFERRED`) : il s'agit de certaines fonctions systèmes bloquantes, et quatre fonctions des pthreads.

```
void pthread_testcancel(void);
```

Teste si une damande d'annulation a été émise (point d'annulation).

4.1.3 Ordonnancement des threads

```
int  pthread_getschedparam(pthread_t target_thread, int *policy, struct
sched_param *param);
int pthread_setschedparam(pthread_t target_thread,  int  policy,  const
struct sched_param *param);
```

La première fonction récupère la politique d'ordonnancement et la seconde l'établit. Trois politiques sont prévues : `SCHED_FIFO` (premier arrivé, premier servi), `SCHED_RR` (*round robin*, *i.e.* temps partagé, `SCHED_OTHER` (politique dependant du constructeur). L'attribut `param` contient principalement la priorité du processus.

4.1.4 Réception des signaux

Permet de positionner le masque des signaux que peuvent recevoir le thread appelant (fonctionne comme `sigprocmask()`).
Le premier argument renseigne s'il faut ajouter (`SIG_BLOCK`), soustraire (`SIG_UNBLOCK`) au masque courant ou prendre comme tel (`SIG_SETMASK`) le masque passé en second argument.

4.1.5 Duplication de processus

```
int  pthread_atfork(void  (*prepare)(void),  void (*parent)(void), void
       (*child)(void));
```

Cette fonction enregistre les trois routines passées en argument. Lors d'un appel `fork()`, avant duplication du processus, la fonction `prepare()` est appelée.
Ensuite `parent()` est exécutée dans le processus père et `child()` est exécutée dans le processus fils.
Cette fonction est utile lorsqu'est mis en jeu un verrou au sein d'un processus entre plusieurs threads, et qu'un threads appelle `fork()`.
Lors de cet appel, un processus fils est créé mais avec seulement un flot d'exécution ; celui du threads appelant .
Si ce thread (dans le contexte du processus fils) veut récupérer le verrou acquis dans le processus père par un autre thread, alors il va bloquer indéfiniment.
Pour remédier à cela `prepare` doit acquérir le verrou (mutex) en question, e,suite les deux autres routines devront libérer ce verrou dansnle contexte du père pour l'une et du fils pour l'autre.

4.2 Synchronisation

4.2.1 Les mutex : Opérations sur un mutex

```
int  pthread_mutex_init(pthread_mutex_t  *mutex,  const  pthread_mutex-
       attr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Permet de créer ou de détruire un mutex de manière dynamique. On alloue préallablement de la mémoire via `malloc` avec pour argument `sizeof(pthread_mutex_t)`. La création d'un mutex peit être paramétrée (sinon positionner le 2ème paramètre à `NULL`). Pour initialiser un mutex de manière statique on procède comme suit :

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Le première bloque jusqu'à obtenir le mutex, la seconde le relâche, et la troisième tente d'obtenir le mutex et echoue si elle ne le peut.

4.2.2 Les mutex : Gestion des mutex

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Permet la création ou la destruction d'une structure d'atrtibuts à passer lors de la création d'un mutex.

4.2.3 Les mutex : Partage d'un mutex

```
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int * pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

Le partage d'un mutex ebtre plusieurs processus se fait par l'intermédiairede ces fonctions. L'argument `pshared` peut prendre la valeur `PTHREAD_PROCESS_SHARED` pour en activer l apartage ou la valeur `PTHREAD_PROCESS_PRIVATE` pour en restreindre l'accès au processus l'ayant créé. Notons que la création d'un mutex doit se faire en mémoire partagée si l'on souhaite activer l'option `PTHREAD_PROCESS_SHARED`. Attention, sous linux cette fonctionnalité n'est disponible que depuis NTPL.

4.2.4 Les mutex : Protocoles pour la résolution des priorités

Les fonctions suivante permettent de récupérer ou d'établir le protocole utilisé pour règler les problèmes d'inversion de priorités (*inversion safe*) pouvant survenir lorsqu'un thread de priorité supérieure à un second essaie de prendre un mutex que ce dernier tient.
Dans cce cas, un thread de priorité intermédiaire peut préampter le thread de priorité basse et retarder finalement la prise du mutex par le thread de priorité haute qui est en attente sur le mutex. On risque donc un retard pour l'exécution d'une tâche critique.
Pour résoudre ce problème, un protocole simple donne temporairement une priorité au thread ayant le mutex égale à celui du thread de priorité haute (mécanisme d'héritage de la priorité). On peut également inverser les priorités des deux tâches de manière à l'aisser s'exécuter le thread de priorité «initialement» basse jusqu'à ce qu'il relâche le mutex (mécanisme d'inversion des priorités). On résout ainsi le problème du thread de priorité intermédiaire. Il s'agit ici de considération temps réel, aussi ces fonctions ne sont implémentées que pour des systèmes de ce type.

```
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int * protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

Les trois valeurs de `protocol` sont : `PTHREAD_PRIO_NONE` (pour aucun protocole), `PTREAD_PRIO_INHERIT` (protection par héritage des priorités) et `PTHREAD_PRIO_PROTECT` (protection par inversion des priorités).

4.2.5 Les mutex : Configuration de la priorité

```
int pthread_mutexattr_getprotocolprioceiling(pthread_mutexattr_t *attr,
int prioceiling, int * oldceiling);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);
```

Le valeur `prioceiling` (de 1 à 127) doit correspondre à la priorité maximale des threads pouvant obtenir le mutex. Cette valeur est ensuite utilisée par le protocole de protection d'inversion pour empêcher les problèmes.

4.2.6 Les variables de conditions : Gestion classique

```
int pthread_cond_init(pthread_cond_t    *cond,    pthread_condattr_t
       *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Permet de créer ou dsétruire une *variable de condition*. En outre, sa création peut être paramétrée (sinon positionner le 2ème argument à `NULL`). Pour initialiser une *variable de condition* de manière statique :

```
 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

Ces deux fonctions permettent la synchronisation de plusieurs threads. Détaillons leur comportement.
La première débloque le mutex passé en paramètre (il faut que le thread appelant ait initialement le mutex), puis se met en attente sur la *variable de condition*. C'est alors qu'un aure thread peut acquérir le mutex puis effectuer des modifications sur une structure d'échange par exmeple, et finalement appelle `pthread_cond_signal()`.
Le premier thread se réveillle et essaie d'obtenir le mutex (le réveil et le blocage sur le mutex se fait atomiquement dans la fonction `pthread_cond_wait()`), en vain... Il faut d'abord que l'autre thread le relâche. Notons que si plusieurs threads sont en attente sur une même *variable de condition*, alors la fonction `pthread_cond_signal()` n'en réveillera qu'un . Pour réveiller tous les threads en attente, on utilise la fonction suivante :

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Une alternative à la fonction d'attente est sa version temporisée. La fonction suivante attend jusqu'à la date précisée en troisième argument. Pour obtenir la date actuelle on peut utiliser `gettimeofday()`.

```
int   pthread_cond_timedwait(pthread_cond_t   *cond,    pthread_mutex_t
       *mutex, const struct timespec *abstime);
```

Notons qu'une demande d'annulation d'un thread en attente sur une *variable de condition* peut bloque indéfiniment. Il faut d'abord que la fonction `pthread_cond_wait()` ait récupérer le mutex (état prévisible). Cependant, il ne faut pas que le thread se termine avec le mutex bloqué. On utilise donc une fonction de nettoyage comme suit :

```
pthread_mutex_lock(&mutex);
pthread_cancel_push(pthread_mutex_unlock, (void *) &mutex);
while(condition_non_realisee)
pthread_cond_wait(&cond,&mutex);
pthread_cancel_pop(1);
```

4.2.7 Les variables de conditions : Utilisation d'attributs

```
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Permet la création ou la destruction d'une structure d'attributs à passer lors de la création d'une *variable de condition*.

4.2.8 Les variables de conditions : Le partage d'une variable de condition

```
int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *pshared);
int pthread_condattr_setpshared(const pthread_condattr_t *attr, int pshared);
```

Ces fonctions permettent respectivment de récupérer et de configurer l'état de partage d'une *variable de condition*. L'argument `pshared` peut prendre la valeur `PTREAD_PROCESS_SHARED` pour en activer la partage ou la valeur `PTREAD_PROCESS_PRIVATE` pour en restreindre l'accès au processus l'ayant créé.

4.3 Les données privées

Notons que l'utilisation dans la déclaration des variables du mot clé `__thread` permet de déclarer statiquement des données privées pour un thread C et C++. Attention tout de même, car cci n'est pas une extension officielle des langages, mais les compilateurs doivent implémenter ceci pour supporter la nouvelle ABI de l'ELF.

```
int pthread_key_create(pthread_key_t *key, void (*destr_function) (void *));
int pthread_key_delete(pthread_key_t key);
```

Permet la création et la destruction de clés pour le stockage de données spécifiques à un thread (analogue à une table d ehachage).
On peut éventuellement spécifier une fonction pour libérer la clé (lors du `delete`) en deuxième argument de la première fonction (sinon `NULL`).

```
void * pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *pointer);
```

Ces fonctionds permettent respactivement de récupérer par une variable privée (via un pointeur), ou de mémoriser la variables passée en paramètre dans une zone privée du thread appelant.

4.4 La modification des attributs

4.4.1 Gestion de base

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Permet la création ou la destruction d'une structure d'attributs à passer lors de la création d'un thread.

4.4.2 Attributs detached/joinable

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int   pthread_attr_getdetachstate(const   pthread_attr_t   *attr,   int *detachstate);
```

Permet de spécifier si un thread doit être détaché (`PTHREAD_CREATE_DETACHED`) ou *joinable* (`PTHREAD_CREATE_JOINABLE`), ou sinon de connaître cet état.

4.4.3 Modification de la pile d'un thread

```
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
int pthread_attr_getstacksize(const pthread_attr_t *attr, void *stacksize);
int pthread_attr_setstackaddr(const pthread_attr_t *attr, void *stackaddr);
int pthread_attr_setstacksize(const pthread_attr_t *attr, void stacksize);
```

On récupère ou on modifie soit la taille de la pile pour le thread que l'on souhaite créer, soit l'adresse de la base de la pile.

4.4.4 Attribut sur l'ordonnancement

```
int  pthread_attr_setschedparam(pthread_attr_t  *attr,  const  struct
       sched_param *param);
int  pthread_attr_getschedparam(const  pthread_attr_t  *attr,  struct
       sched_param *param);
```

L'attribut modifié ou récupéré concerne esssentiellement la priorité du thread. Le champ associé à la structure est `sched_priority`, sa valeur peut aller de 1 (thread le moins favorisé) à 127 (thread le plus favorisé).

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,  int  *policy);
```

On choisit ou on récupère ici la politique d'ordonnancement. Trois politique sont prévues : `SCHED_FIFO` (premier arrivé, premier servi), `SCHED_RR` (*round robin*, *i.e.* temps partagé), `SCHED_OTHER` (politique dépendant du constructeur).
On signale (ou on récupère) que l'ordonnancement est spécifique au thread créé (`PTHREAD_EXPLICIT_SCHED`), ou s'il doit suivre la configuration du thread créateur (`PTHREAD_INHERIT_SCHED`).

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

Ces fonctions n'ont d'intérêt que pour un multithreading hybride. Le cas échéant, l'ordonnancement peut se faire au niveau de chaque processus (`PTHREAD_SCOPE_PROCESS`), ou au niveau système (`PTHREAD_SCOPE_SYSTEM`) : la portée du scheduler est donc globale.

5 Applications

Notons les point forts d'un programme multithreadé. En effet il permet :
- Sa décomposition en tâches n'ayant pas un rapport direct de causalité. La décomposition du programme en éléments distincts facilite sa compréhension (au détriment d'une gestion plus compliquée), sa maintenabilité, sa flexibilité.
- Sa pleine utilisation du parallélisme d'une machine, ou à défaut du profit de la tranche totale de temps alloué au processus par le kernel (utilistion au mieux de la CPU).
- Le partage de la mémoire et des fichiers ouverts du processus entre ses threads permet leur communication sans recours au kernel. Les conséquences de ces rapports nous amènent trivialement à l'utilisation du multithreading dans les serveurs.

Donnons un exemplede scénario possible. Un thread R s'occupe de la reception des requêtes émises par les clients. A chacune d'entre elles un nouveau thread est créé pour la traiter, et peut même être dédié à une communication s'il y a lieu avec le client. De cette manière, le thread R peut s'occuper de nouvelles demandes.
La connexion au serveur se fait toujours par le même canal de communcation (*port*). Les clients dialoguent ensuite avec leurs threads dédiés par d'autres ports, lesquels sont discutés par les clients et le serveur avant le traitement de leur requête.
Il est aussi possible de créer un *pool* (bassin, réserve) de threads, permetttant d'anticiper sur la demande et ainsi de traiter plus rapidement les requêtes des clients. Un autre domaine dans lequel les threads sont courants est celui des interfaces graphiques. Le dialogue avec l'utilisateur se fait par un GUI (graphique user interface). Le programme associe aux actions de l'utilisateur (clic sur bouton, par exemple) une fonction appelée *callback* (fonction reflexe). Il s'agit en fait d'un thread qui est créé lorsque l'évènement

à lieu. De cette manière, l'utilisateur peut continuer à interagir avec l'application (l'interface reste active, elle n'est pas gelée par l'exécution d'une requête). Le traitement des demandes faites par l'utilisateur est donc encore une fois détaché de sa reception. Les bibliothèques graphiques GTK+ et Qt utilisent ce principe.
Le multithreadiong est donc à envisager dans tous les systèmes concurrentiels, dès lors qu'il y a partage de ressources.

6 Les bibliothèques de *threads*

De nombreux systèmes d'exploitation permettent aujourd'hui la programmation par *threads* : Solaris 5.x de SUN, Windows95/98/NT/XP et bien d'autres (dont LINUX). Dans le cas de Solaris, la bibliothèque de *threads* disponible est conforme à la norme POSIX 1003.1c ce qui assure une certaine portabilité de l'applicatif en cas de portage vers un autre système. Dans le cas des systèmes Microsoft, la bibliothèque utilisée est bien entendu non conforme à cette norme POSIX ! Il existe aujourd'hui diverses bibliothèques permettant de manipuler des *threads* sous LINUX. On dénombre deux principaux types d'implémentations de *threads* :
- Au niveau utilisateur (*user-level*). A ce moment la, la gestion des *threads* est entièrement faite dans l'espace utilisateur.
- Au niveau noyau (*kernel-level*). Dans ce cas, les *threads* sont directement gérés par le noyau.

Dans ce dernier cas, la base de l'implémentation est entre-autres l'appel système clone, également utilisé pour la création de processus classiques :

```
NAME
       clone - create a child process

SYNOPSIS
       #include <linux/sched.h>
       #include <linux/unistd.h>

       pid_t clone(void *sp, unsigned long flags)

DESCRIPTION
       clone  is  an  alternate  interface  to  fork,  with  more
       options.  fork is equivalent to clone(0, SIGCLD|COPYVM).
```

La bibliothèque LinuxThreads développée par Xavier Leroy (`Xavier.Leroy@inria.fr`) est une excellente implémentation de la norme POSIX 1003.1c. Cette bibliothèque est basée sur l'appel système clone. Je ne saurais trop vous conseiller d'utiliser ce produit, ce que nous ferons dans la suite des exemples présentés dans cet article.

7 Comment créer des *threads* sous LINUX ?

Voici un petit exemple de programme utilisant deux *threads* d'affichage : **thread1.c**

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

void *my_thread_process (void * arg)
{
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
```

```
}

main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;

  if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, my_thread_process, "2") < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }

  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

La fonction `pthread_create` permet de créer le *thread* et de l'associer à la fonction `my_thread_process`. On notera que le paramètre `void *arg` est passé au *thread* lors de sa création. Après création des deux *threads*, le programme principal attend la fin des *threads* en utilisant la fonction `pthread_join`.
Taper ou récupérer le programme `thread1.c`.
Pour le compiler ,utiliser la commande `gcc`, par exemple (premier exercice) : `gcc -D_REENTRANT -o thread1 thread1.c -lpthread`.

Que se passe-t-il ? Pourquoi ?
Réponse :

Maintenant, passer la variable `i` en variable globales. Que se passe-t-il ? Pourquoi ?
Réponse :

8 Partages des données et synchronisation

8.1 Les MUTEX

Le partage de données nécessite parfois (même souvent) d'utiliser des techniques permettant de protéger à un instant donné une variable partagée par plusieurs *threads*. Imaginons un simple tableau d'entier rempli par un *thread* (lent) et lu par un autre (plus rapide). Le *thread* de lecture doit attendre la fin du remplissage du tableau avant d'afficher sont contenu. Pour cela, on peut utiliser le système des MUTEX (MUTual EXclusion) afin de protéger le tableau pendant le temps de son remplissage.
Taper ou récupérer le programme **thread2.c**

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

static pthread_mutex_t my_mutex;
static int tab[5];

void *read_tab_process (void * arg)
{
  int i;

  pthread_mutex_lock (&my_mutex);
  for (i = 0 ; i != 5 ; i++)
    printf ("read_process, tab[%d] vaut %d\n", i, tab[i]);
  pthread_mutex_unlock (&my_mutex);
  pthread_exit (0);
}

void *write_tab_process (void * arg)
{
  int i;

  pthread_mutex_lock (&my_mutex);
  for (i = 0 ; i != 5 ; i++) {
    tab[i] = 2 * i;
    printf ("write_process, tab[%d] vaut %d\n", i, tab[i]);
    sleep (1); /* Relentit le thread d'ecriture... */
  }
  pthread_mutex_unlock (&my_mutex);
  pthread_exit (0);
}

main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;

  pthread_mutex_init (&my_mutex, NULL);

  if (pthread_create (&th1, NULL, write_tab_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, read_tab_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }
```

```
  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

Compiler le programme.
Que se passe-t-il ? Pourquoi ?
Réponse :

Maintenant, enlever les verrouillages dans les 2 *threads*. Recompiler le programme.
Que se passe-t-il ? Pourquoi ?

8.2 Les variables de condition

Une condition (variable de condition) est un mécanisme de synchronisation qui permet aux threads de suspendre leur exécution et de relacâcher les processeurs jusqu'à ce qu'un prédicat sur une donée partagée soit satisfait. Les opérations de base sont : signal de la condition (quand le prédicat est devient vrai), attente de la condition qui suspend l'exécution du thread jusqu'a ce qu'un autre thread signale la condition (la rende vrai).
Une variable de condition doit toujours être associée avec un mutex, pour éviter la «condition de course», ou un thread se prépare a attendre une variable de condition et un autre thread signale la condition juste avant le premier thread qui attend activement cette condition.
`pthread_cond_init` initialise cette variable de condition en utilisant les atrtibuts spécifiés dans `cond_attr`, ou les attributs par défaut si `cond_attr` vaut `NULL`. L'implémentation LinuxThreads ne supporte pas les attributs de conditions et le paramètre `cond_attr` est ignoré.

Les variables de type `pthread_cond_t` peuvent être initialisée staiquement en utilisant la constante `PTHREAD_COND_INITIALIZER`.
`pthread_cond_signal` redémarre un des threads qui attendait le variable de condition `cond`. Si aucun thread n'attend la variable de condition, rien ne se passe. Si plusieurs threads attendent cette variable, seulement un seul est redémarré, mais on ne sais pas lequel. `pthread_cond_broadcast` redémarre tous les threads qui attendent la variable de condition.
`pthread_cond_wait` déverouille atomiquement le mutex (come `pthread_unlock_mutex`) et attends que la variable de condition `cond` soit signalée. L'exécution du thread est suspendue et ne consomme alors aucune CPU jusqu'à ce que la variable soit signalée. Le mutex doit être vérouillé par le thread appelant à l'entrée de `pthread_cond_wait`. Avant de retourner dans le thread appelant , `pthread_cond_wait` refait l'acquisition du mutex (comme dans`pthread_lock_mutex`).
Le déverouillage du mutex et l'attente sur la variable de condition sont fait atomiquement. Donc si tous les threads font toujours l'acquisition du mutex avant de signaler le variable de condition, ceci garanti que la condition ne peut pas être signalée (et donc ignorée) entre le moment ou un thread verouille le mutex et le moment ou il attends la variable de condition.
`pthread_cond_timedwait` deverouille le mutex et attend la condition atomiquement comme `pthread_cond_wait`, mais cette fonction borne la durée de l'attente. Si la condition n'a pas été signalée pendant cet intervalle de temps spécifié par `abstime`, le mutex mutex est de nouveau acquis et `pthread_cond_timedwait` retourne l'erreur `ETIMEDOUT`. Le paramètre `abstime` spécifie le temps absolu, avec la même origine que les fonctions `time(2)` et `gettimeofday(2)` : un temps absolu de 0 correspond à 00 :00 :00 GMT, January 1, 1970.
`pthread_cond_destroy` détruit une variable de condition, libère les ressources. Aucin thread ne doit attendre la variable qui va être détruite par la fonction`pthread_cond_destroy`. Dans l'implémentation LinuxThreads, aucune ressource n'est associée à la variable de condition, donc pour l'instant la fonction ne fait que vérifier qu'il n'y a aucun thread qui attende la variable.

```
int pthread_cond_init (pthread_cond_t *cond, const pthread_cond_attr *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Crée une variable condition (on ignorera l'attribut).

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

Détruit la variable condition.

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

L'activité appelante doit posséder le verrou mutex. L'activité est alors bloquée sur la variable condition après avoir libéré le verrou. L'activité reste bloquée jusqu'à ce que la variable condition soit signalée et que l'activité ait réussi à réacquérir le verrou.

```
int pthread_cond_signal (pthread_cond_t *cond);
```

Signale la variable condition : une activité bloquée sur la variable condition est réveillée. Cette activité tente alors de réacquérir le verrou correspondant à son appel de `cond_wait`. Elle sera effectivement débloquée quand elle réussira à réacquérir ce verrou. Il n'y a aucun ordre garanti pour le choix de l'activité réveillée. L'opération signal n'a aucun effet s'il n'y a aucune activité bloquée sur la variable condition (pas de mémorisation).

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Toutes les activités en attente sont réveillées, et tentent d'obtenir le verrou correspondant à leur appel de `cond_wait`.

Remarques : Les verrous sont par défaut des verrous d'exclusion mutuelle. En utilisant les attributs (non documentés ici), on peut créer un verrou dit récursif qui peut être verrouillé plusieurs fois par la même activité (le verrou doit alors être autant de fois déverrouillé avant qu'une autre activité puisse l'acquérir).
Par ailleurs, contrairement à la définition des moniteurs de Hoare, l'activité signalée n'a pas priorité sur le signaleur : le signaleur ne perd pas l'accès au moniteur s'il le possédait, et le signalé reste bloqué tant qu'il n'obtient pas le verrou. C'est pourquoi il est nécessaire d'utiliser une boucle d'attente réévaluant

la condition d'exécution. En effet, cette condition peut être invalidée entre le moment où l'activité est signalée et le moment où elle obtient effectivement le verrou, par exemple si une autre activité obtient le mutex et pénètre dans le moniteur avant l'activité signalée.
Enfin, pour des raisons d'efficacité, il est courant de faire l'appel à `cond_signal` hors de la zone lock-unlock, de sorte que l'activité signalée puisse acquérir plus facilement le verrou. Attention cependant à garantir l'atomicité des opérations du moniteur !

Exemple : Considérons deux variables partagées `x` et `y`, protégée pat le mutex `mu` et la variable de condition `cond` qui doit être signalée lorsque `x`devient supérieure à `y`.

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Attente jusqu'à ce qu'`x` soit supérieur à `y` est réalisée par :

```
pthread_mutex_lock(&mut);
while (x <= y)
{
pthread_cond_wait(&cond, &mut);
}
/* operate on x and y */
pthread_mutex_unlock(&mut);
```

Les modifications sur `x` et `y` qui pourraient faire quen`x` deviennent supérieure à `y` doivent signaler la condition si nécessaire :

```
pthread_mutex_lock(&mut);
/* modify x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

S'il peut être prouvé qu'au moins un des threads appelant doit être réveillé (par exemple, s'il y a seulement 2 threads qui communique via `x` et `y` ; `pthread_cond_signal` peut être utilisée comme une alternative plus efficace que `pthread_cond_broadcast`. Dans le doute on utilise `pthread_cond_broadcast`.
Pour attendre que `x` devienne supérieur à `y` avec un délai de 5 secondes, on fait :

```
struct timeval now;
struct timespec timeout;
int retcode;

pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
while (x <= y && retcode != ETIMEDOUT)
{
retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
}
if (retcode == ETIMEDOUT)
{
/* timeout occurred */
    }
else
{
    /* operate on x and y */
    }
pthread_mutex_unlock(&mut);
```

Exercice : Voici un exemple simple :**condition.c**

```
//Exemple de moniteur : producteur/consommateur à une case

/*Compilation
gcc -D_REENTRANT -o exemple2 exemple2.c -lpthread
*/
#include <pthread.h>
#include <string.h>

static pthread_cond_t est_libre, est_plein;
static pthread_mutex_t protect;
static char *buffer;

/* Depose le message msg (qui est dupliqué). Bloque tant que le tampon est plein. */
void deposer (char *msg)
{
    pthread_mutex_lock (&protect);
    while (buffer != NULL)
        pthread_cond_wait (&est_libre, &protect);
    /* buffer = NULL */
    buffer = strdup (msg); /* duplication de msg */
    pthread_cond_signal (&est_plein);
    pthread_mutex_unlock (&protect);
}

/* Renvoie le message en tête du tampon. Bloque tant que le tampon est vide.
 * La libération de la mémoire contenant le message est à la charge de l'appelant. */
char *retirer (void)
{
    char *result;
    pthread_mutex_lock (&protect);
    while (buffer == NULL)
        pthread_cond_wait (&est_plein, &protect);
    /* buffer != NULL */
    result = buffer;
    buffer = NULL;
    pthread_cond_signal (&est_libre);
    pthread_mutex_unlock (&protect);
    return result;
}

/* Initialise le producteur/consommateur. */
void init_prodcons (void)
{
    pthread_mutex_init (&protect, NULL);
    pthread_cond_init (&est_libre, NULL);
    pthread_cond_init (&est_plein, NULL);
    buffer = NULL;
}

/*------------------------------------------------------------------
  Fonction executee par les threads.
  ------------------------------------------------------------------*/
void Thread_ret (void)
{
  int i;
char message[25];
for (i=0;i<5;i++)
```

```
{
sprintf(message,"%d",i);
strcat(message, "--message--");
deposer(message);
printf("Depose message %d \n",i);
}                                                                                    pthread_exit(NULL);
}

void Thread_dep (void)
{
  int i;
char * message;
for (i=0;i<5;i++)
{
message = retirer();
printf("Retire message %d \n",i);
printf("Les message est : %s\n", message);
sleep(1); /*pour voir la synchronisation*/
}                                                                                    pthread_exit(NULL);
}
/*----------------------------------------------------------------*/


int main(int argc, char *argv[])
{
 pthread_t thr_main, threads[2];
int i, NbThreads;
 thr_main = pthread_self();
NbThreads = 2;

/*Initialisation du producteur/consommateur*/
init_prodcons();

 /* creation des threads  */
 pthread_create(&threads[0], NULL, (void *)Thread_dep, NULL);
printf("------ Thr %d (main) cree: %d (i = %d)\n",(int)thr_main,(int)threads[0],0);

 pthread_create(&threads[1], NULL, (void *)Thread_ret, NULL);
printf("------ Thr %d (main) cree: %d (i = %d)\n",(int)thr_main,(int)threads[1],1);

/*thread en mode join*/
pthread_join(threads[0], NULL);
       printf("------main thread %d (main) fin de : %d\n",
             (int)thr_main, (int)threads[0]);

pthread_join(threads[1], NULL);
       printf("------main thread %d (main) fin de : %d\n",
             (int)thr_main, (int)threads[1]);
printf("------main Fin de main (%d)\n", (int)thr_main);
return 0;
}
```

Compiler le programme.
Que se passe-t-il ? Pourquoi ?

Réponse :

8.3 Les sémaphores POSIX

Les sémaphores sont des compteurs pour des ressources partagées entre les threads. Les opérations de bases sur les sémaphores sont : l'incrémentation atomiquement, et attente jusqu'à ce que le compteur soit non nul et décrémentation atomiquement.

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

`sem_init` initialise l'objet sémaphore pointé par `sem`. Le compteur associé au sémaphore est fixé initialement à`value`. L'argument `pshared` indique si le sémaphore est local au processus courant (`pshared` à 0) ou s'il est partagé par plusieurs processus (`pshared` différent de 0). LinuxThreads ne supporte pas pour l'instant les sémaphores partagés, donc `sem_init` retourne toujours avec l'erreur `ENOSYS` si `pshared` est différent de 0.

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

`sem_wait` suspend le thread appelant jusqu'à ce que le sémaphore pointé par `sem` ait un compteur non nul. Alors le compteur du sémaphore est atomiquement décrémenté.

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

`sem_trywait` est la variante non bloquante de `sem_wait`. Si le sémaphore pointé par `sem` possède un compteur non nul, le compteur est décrémenté atomiquement et `sem_trywait` retourne immédiatement 0. Si le sémaphore possède un compteur nul, `sem_trywait` retourne immédiatement l'erreur `EAGAIN`.

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

`sem_post` incrémente atomiquement le compteur du sémaphore pointé par `sem`. Cette fonction n'est jamais bloquante et peut être utilisée de manière sure par des gestionnaires de signaux asynchrones.

```
int sem_getvalue(sem_t * sem, int * sval);
```

`sem_getvalue` stocke à l'endroit pointé par `sval` le compteur courant du sémaphore `sem`.

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

`sem_destroy` détruit un objet sémaphore, en libérant les ressources qu'il possédait. Aucun threads ne doit attendre le sémaphore lorsque celui-ci est détruit. Dans l'implémentation LinuxThreads, aucune ressources n'est associée à l'objet sémaphore, donc, pour l'instant, la fonction `sem_destroy` ne fait rien à part tester si aucun thread n'attend après l'objet sémaphore.

L'utilisation de sémaphores permet aussi la synchronisation entre plusieurs *threads*. Voici un exemple simple :**thread3.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
static sem_t my_sem;
int the_end;
void *thread1_process (void * arg)
{
  while (!the_end) {
    printf ("Je t'attend !\n");
    sem_wait (&my_sem);
  }
  printf ("OK, je sors !\n");
  pthread_exit (0);
}
void *thread2_process (void * arg)
{
  register int i;
  for (i = 0 ; i < 5 ; i++) {
    printf ("J'arrive %d !\n", i);
    sem_post (&my_sem);
    sleep (1);
  }
  the_end = 1;
  sem_post (&my_sem); /* Pour debloquer le dernier sem_wait */
  pthread_exit (0);
}
main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;
  sem_init (&my_sem, 0, 0);
  if (pthread_create (&th1, NULL, thread1_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
  if (pthread_create (&th2, NULL, thread2_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }
  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

Compiler le programme.
Que se passe-t-il ? Pourquoi ?
Réponse :

9 Mode de création des *threads* : `JOINABLE` ou `DETACHED`

Dans les exemples précédents, les *threads* sont créés en mode `JOINABLE`, c'est à dire que le processus qui a créé le *thread* attend la fin de celui-ci en restant bloqué sur l'appel à `pthread_join`. Lorsque le *thread* se termine, les resources mémoire du *thread* sont libérées grâce à l'appel à `pthread_join`. Si cet appel n'est pas effectué, la mémoire n'est pas libérée et il s'en suit une fuite de mémoire. Pour éviter un appel systématique à `pthread_join` (qui peut parfois être contraignant dans certaines applications), on peut créer le *thread* en mode `DETACHED`. Dans ce cas la, la mémoire sera correctement libérée à la fin du *thread*.
Pour cela il suffit d'ajouter le code suivant :

```
pthread_attr_t thread_attr;
if (pthread_attr_init (&thread_attr) != 0) {
  fprintf (stderr, "pthread_attr_init error");
  exit (1);
}
if (pthread_attr_setdetachstate (&thread_attr, PTHREAD_CREATE_DETACHED) != 0) {
  fprintf (stderr, "pthread_attr_setdetachstate error");
  exit (1);
}
```

puis de créer les *threads* avec des appels du type :

```
if (pthread_create (&th1, &thread_attr, thread1_process, NULL) < 0) {
  fprintf (stderr, "pthread_create error for thread 1\n");
  exit (1);
}
```

Écrire le programme en reprenant **thread3.c**, et le compiler.
Que se passe-t-il ? Pourquoi ?
Réponse :

10 Destruction de *thread* : cancellation

Les exemples ci-dessus utilisaient la fonction `pthread_exit` pour la destruction d'un *thread* (en fait le *thread* se détruisait tout seul). Il existe un mécanisme dans lequel un *thread* peut en détruire en autre à condition que ce dernier ait validé cette possibilité. Le comportement par défaut est de type décalé (`deferred`). Lorsqu'on envoie une requête de destruction d'un *thread*, celle-ci n'est exécutée que lorsque ce *thread* passe par un cancellation point comme par exemple l'appel à la fonction `pthread_testcancel`. Voici un petit exemple : **thread4.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *my_thread_process (void * arg)
{
  int i;
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
    pthread_testcancel (); /*ICI, permis de tuer = license to kill*/
  }
}
main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;
  if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
  sleep (2);
  if (pthread_cancel (th1) != 0) {
    fprintf (stderr, "pthread_cancel error for thread 1\n");
    exit (1);
  }
  (void)pthread_join (th1, &ret);
}
```

Compiler le programme.
Que se passe-t-il ? Pourquoi ?
Réponse :

Pour éviter l'utilisation des cancellation points, on peut indiquer que la destruction est en mode asynchrone en modifiant le code du *thread* de la manière suivante :

```
void *my_thread_process (void * arg)
{
  int i;
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
}
```

Écrire le programme en reprenant **thread4.c**, et le compiler.
Que se passe-t-il ? Pourquoi ?
Réponse :

11 Excercice 1

Dans cet exercice, trois *threads* seront créés, et ils incrémenteront chacun une variable globale différente. La valeur de chacune de ces variables est affichée par le programme initial.

Réponse :

12 Debug d'un programme multi-thread sous LINUX

Les dernières versions de `gdb` et de la `glibc` permettent de debugger un programme LINUX utilisant du multi-threading. Plus d'infos sont disponibles sur la page WWW de la bibliothèque LinuxThreads (voir bibliographie). Voici un petit exemple de session gdb sur le programme d'exemple `thread1`. La compilation s'effectue de la manière suivante : `gcc -ggdb -D_REENTRANT -o thread1 thread1.c`

```
(gdb) b main
Breakpoint 1 at 0x8048622: file thread1.c, line 22.
```

On a posé un point d'arrêt dans le programme principal avant la création des *threads*.

```
(gdb) n
[New Thread 25572]
[New Thread 25571]
[New Thread 25573]
Thread 1: 0
Thread 1: 1
Thread 1: 2
Thread 1: 3
Thread 1: 4
(gdb) info threads
  3 Thread 25573  0x4007a921 in __libc_nanosleep ()
* 2 Thread 25571  main (ac=1, av=0xbffffca0) at thread1.c:27
  1 Thread 25572  0x4008b2de in __select ()
```

L'action sur next exécute la fonction `pthread_create` qui provoque la création du thread 1. La commande info *threads* permet de connaître la liste des tous les *threads* associés à l'exécution du programme. Le *thread* courant est indiqué par l'étoile, le numéro du *thread* est indiqué en deuxième colonne (ici 1, 2, 3).

```
(gdb) thread 1
[Switching to Thread 25654]
#0  0x4008b2de in __select ()
```

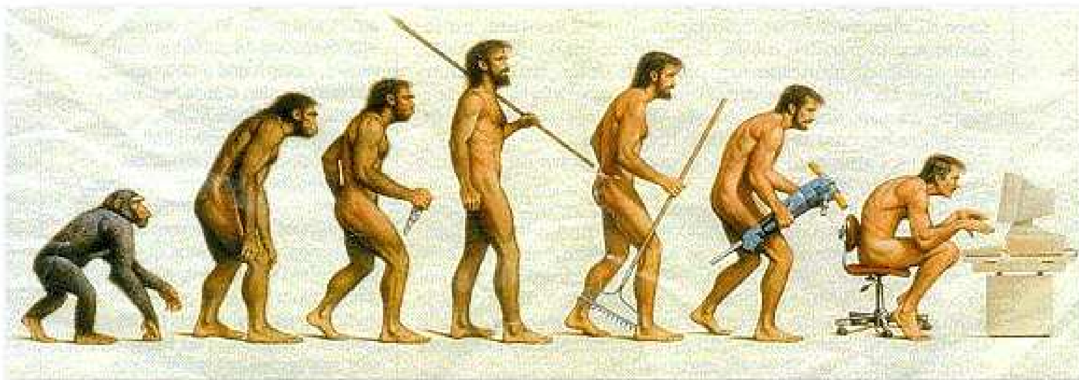
On peut passer d'un *thread* à l'autre en utilisant la commande *thread* numéro-du-thread.

13 Conclusion et bibliographie

L'utilisation du multi-threading permet de faciliter la programmation d'un grand nombres d'applications de type serveur ou multimédia, tout en améliorant les fonctionnalités du programme par rapport à une solution classique basée sur l'utilisation des créations de processus (`fork`). Les pointeurs suivants vous seront utiles si vous vous lancez dans le multi-threading :

– La bibliothèque LinuxThreads sur http://pauillac.inria.fr/~xleroy/linuxthreads
– Le site "Programming POSIX *threads*" sur http://www.humanfactor.com/pthreads
– Si vous voulez porter vos applicatifs sur Win32, la bibliothèque "POSIX Threads (pthreads) for Win32" sur http://sourceware.cygnus.com/pthreads-win32
– La FAQ du groupe de discussion `comp.programming.threads` sur http://www.serpentine.com/~bos/threads-faq

Vous en avez maintenant terminé avec Unix...



Réflexion sur l'évolution humaine...