Recueil de travaux pratiques de SE2

Lakhdar Loukil
Université Oran 1
Département d'informatique

Année universitaire: 2023-2024

Contents

1 Commandes Linux pour afficher des informations sur les processus

Le système d'exploitation Linux dispose d'un ensemble de commandes permettant de récupérer des informations sur les processus qui s'exécutent dans une machine Linux. Il existe en particulier les commandes `ps`, `pstree` et `top`. Il existe également le pseudo-système de fichiers `/proc` qui fournit une interface aux structures de données du kernel et en particulier aux processus du système.

1.1 La commande `ps`

La commande `ps` permet d'afficher des informations sur une sélection de processus actifs du système. La sélection des processus et les informations affichées dépendent des options fournies à la commande. Sans options, `ps` sélectionne et affiche les processus de l'utilisateur courant et associés au terminal dans lequel la commande est invoquée:

```
laloukil@laloukil:~$ ps
  PID TTY          TIME CMD
 2636 pts/9    00:00:00 bash
 2649 pts/9    00:00:00 ps
```

La commande `ps` possède de nombreuses options (voir le manuel de la commande pour plus de détails). Nous donnons, dans ce qui suit, quelques options utiles.

1.1.1 Liste des processus du système

Pour lister tous les processus du système, on utilise l'option `-e` ou `-A`.

```
laloukil@laloukil:~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    7 ?        00:00:01 rcu_sched
    8 ?        00:00:00 rcuos/0
    9 ?        00:00:00 rcuos/1
   10 ?        00:00:00 rcuos/2
   11 ?        00:00:00 rcuos/3
   12 ?        00:00:00 rcu_bh
.....
```

Les champs affichés pour chaque processus sont: `PID` (identifiant du processus), `TTY` (le terminal associé au processus, `?` indique que le processus n'est pas rattachée à un terminal), `TIME` (temps CPU cumulé) et `CMD` (la commande qui a créé le processus).

1.1.2 Liste complète des processus

L'option `-f` (full) permet d'afficher des champs (colonnes) supplémentaires pour un processus. Les colonnes additionnelles affichées par l'option `-f` sont `UID` (identifiant de l'utilisateur), `PPID` (identifiant du processus père), `C` (rapport du temps d'utilisation du CPU sur le temps d'exécution) et `STIME` (date de démarrage du processus).
L'option `-f` peut être combinée avec d'autres options pour afficher des colonnes additionnelles.

Exemple:

```
laloukil@laloukil:~$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 19:24 ?        00:00:01 /sbin/init
root         2     0  0 19:24 ?        00:00:00 [kthreadd]
root         3     2  0 19:24 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 19:24 ?        00:00:00 [kworker/0:0H]
root         7     2  0 19:24 ?        00:00:01 [rcu_sched]
root         8     2  0 19:24 ?        00:00:00 [rcuos/0]
root         9     2  0 19:24 ?        00:00:00 [rcuos/1]
root        10     2  0 19:24 ?        00:00:00 [rcuos/2]
root        11     2  0 19:24 ?        00:00:00 [rcuos/3]
.....
```

Remarque:
Il est parfois utile de relier en pipe la commande `ps` avec la commande `more`, `less` ou `grep` pour prendre le temps de consulter les longues listes de processus (cas des commandes `more` et `less`) ou pour filtrer le résultat de la commande `ps` et n'afficher que les informations utiles (cas de la commande `grep`).

Exemples:

1. `ps` en pipe avec `more` affiche la liste des processus par page. Pour défiler la liste par ligne (resp. par page), appuyer sur la touche *Entree* (resp. *Barre d'espacement*):

   ```
   laloukil@laloukil:~$ ps -A | more
   ```

2. `ps` en pipe avec `less` permet de défiler la liste les processus vers le haut et vers le bas à l'aide des touches *Haut* et *Bas* du clavier. Pour quitter la commande, il suffit d'appuyer sur la touche *Q*:

   ```
   laloukil@laloukil:~$ ps -A | less
   ```

3. La commande suivante permet de vérifier si le processus `firefox` est en cours d'exécution:

```
laloukil@laloukil:~$ ps -A | grep firefox
12494 ?        00:01:20 firefox
```

1.1.3 Liste des processus d'un utilisateur spécifique

Pour sélectionner les processus attachés à un utilisateur spécifique, on utilise l'option `-u` suivie du nom de l'utilisateur. Plusieurs utilisateurs peuvent être indiqués, il suffit de les séparer par une virgule.

La commande suivante permet d'afficher les processus de l'utilisateur `root`:

```
laloukil@laloukil:~$ ps -f -u root
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 10:10 ?        00:00:01 /sbin/init
root         2     0  0 10:10 ?        00:00:00 [kthreadd]
root         3     2  0 10:10 ?        00:00:00 [ksoftirqd/0]
root         4     2  0 10:10 ?        00:00:00 [kworker/0:0]
.....
```

1.1.4 Liste des processus par nom ou par PID

L'option `-C` filtre les processus par commande ou nom de processus. La commande suivante affiche tous les processus `getty`:

```
laloukil@laloukil:~$ ps -f -C getty
UID        PID  PPID  C STIME TTY          TIME CMD
root       901     1  0 10:10 tty4     00:00:00 /sbin/getty -8 38400 tty4
root       905     1  0 10:10 tty5     00:00:00 /sbin/getty -8 38400 tty5
root       911     1  0 10:10 tty2     00:00:00 /sbin/getty -8 38400 tty2
root       912     1  0 10:10 tty3     00:00:00 /sbin/getty -8 38400 tty3
root       915     1  0 10:10 tty6     00:00:00 /sbin/getty -8 38400 tty6
root      1059     1  0 10:10 tty1     00:00:00 /sbin/getty -8 38400 tty1
```

L'option `-p` filtre les processus par leur PID. La commande suivante affiche les processus de PID 3564 et 3582:

```
laloukil@laloukil:~$ ps -f -p 3564,3582
UID        PID  PPID  C STIME TTY          TIME CMD
root      3564     2  0 11:14 ?        00:00:00 [kworker/1:0]
root      3582     2  0 11:23 ?        00:00:00 [kworker/0:2]
```

1.1.5 Tri des processus par utilisation du CPU et de la mémoire

Un administrateur système a souvent besoin de connaître les processus gourmands en temps CPU et/ou en occupation de la mémoire. L'option `--sort` permet de trier la liste des processus sur un champ donné ou un paramètre particulier.

Plusieurs champs peuvent être spécifiés avec l'option `--sort`. Les champs doivent être dans ce cas séparés par une virgule. Les champs peuvent être préfixés par le signe `"-"` ou `"+"` pour un tri descendant ou ascendant respectivement.

La commande suivante trie la liste de tous les processus par ordre décroissant sur l'utilisation du CPU (colonne `%CPU`) puis par ordre décroissant sur l'occupation de la mémoire (colonne `%MEM`).

```
laloukil@laloukil:~$ ps -aux --sort=-pcpu,-pmem | less
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
laloukil  3302  2.5  7.8 1151932 313672 ?      Sl   10:55   1:42 /usr/lib/firefox/fire
laloukil  2566  1.2  4.2 1337256 167212 ?      Sl   10:16   1:21 texmaker
laloukil  1910  0.8  2.8 1542504 111644 ?      Sl   10:11   0:58 compiz
root      1115  0.8  1.5 361612 62256 tty7     Ssl+ 10:10   0:56 /usr/bin/X -core :0 -
laloukil  1741  0.5  0.3 365120 13848 ?        Ssl  10:11   0:39 /usr/bin/ibus-daemon
root        17  0.3  0.0      0     0 ?        S    10:10   0:25 [migration/0]
root        20  0.2  0.0      0     0 ?        S    10:10   0:19 [migration/1]
.....
```

1.2 La commande `pstree`

La commande `pstree` affiche les processus en exécution sous forme d'arborescence dont la racine est le processus `init` ou `systemd` pour les distributions récentes de Linux(père de tous les processus Linux).

La commande `pstree` possède de nombreuses options, nous donnons ci-dessous quelques options particulièrement intéressantes pour ce TP:

`-p` : affiche les PIDs des processus.
`-a` : affiche la ligne de commande et ses arguments.
`<pid>` : affiche affiche l'arborescence du processus <pid>.
`-s` : affiche les processus parents du processus spécifié.

```
laloukil@laloukil:~$ pstree -p
init(1)─┬─ModemManager(680)─┬─{ModemManager}(758)
        │                   └─{ModemManager}(765)
        ├─NetworkManager(769)─┬─dhclient(4008)
        │                     ├─dnsmasq(1134)
        │                     ├─{NetworkManager}(770)
        │                     ├─{NetworkManager}(773)
        │                     └─{NetworkManager}(775)
        ├─accounts-daemon(1307)─┬─{accounts-daemon}(1309)
        │                       └─{accounts-daemon}(1310)
        ├─acpid(971)
        ├─atd(1033)
        ├─avahi-daemon(736)───avahi-daemon(739)
        ├─bluetoothd(743)
        ├─colord(1598)─┬─{colord}(1691)
        │              └─{colord}(1692)
        ├─cron(999)
        ├─cups-browsed(964)
        ├─cupsd(1505)─┬─dbus(1701)
        │             └─dbus(4340)
        ├─dbus-daemon(605)
        ├─getty(881)
        ├─getty(898)
        ├─getty(907)
        ├─getty(908)
        ├─getty(912)
        ├─getty(1123)
```

```
laloukil@laloukil:~$ pstree -a
init
  ├─ModemManager
  │   └─2*[{ModemManager}]
  ├─NetworkManager
  │   ├─dhclient -d -sf /usr/lib/NetworkManager/nm-dhcp-client.action -pf /run/sendsigs.omit.d/network-manager.dhclien
  │   ├─dnsmasq --no-resolv --keep-in-foreground --no-hosts --bind-interfaces --pid-file=/run/sendsigs.omit.d/network-
  │   └─3*[{NetworkManager}]
  ├─accounts-daemon
  │   └─2*[{accounts-daemon}]
  ├─acpid -c /etc/acpi/events -s /var/run/acpid.socket
  ├─atd
  ├─avahi-daemon
  │   └─avahi-daemon
  ├─bluetoothd
  ├─colord
  │   └─2*[{colord}]
  ├─cron
  ├─cups-browsed
  ├─cupsd -f
  │   ├─dbus dbus://
  │   └─dbus dbus://
  ├─dbus-daemon --system --fork
  ├─getty -8 38400 tty4
  ├─getty -8 38400 tty5
  ├─getty -8 38400 tty2
  ├─getty -8 38400 tty3
  ├─getty -8 38400 tty6
  ├─getty -8 38400 tty1
```

Comme pour la commande `ps`, `pstree` peut être reliée en pipe avec les commandes `more`, `less` ou `grep` pour consulter ou chercher un processus dans l'arborescence.

1.3 La commande `top`

La commande `top` offre une vue dynamique et en temps réel de l'activité du système. Elle affiche des informations de synthèse sur l'utilisation de la mémoire et du processeur, suivies de la liste des processus et threads actuellement gérés par le noyau Linux. Les informations sur le système, le type, l'ordre et la taille des informations sur les processus

peuvent être configurés par l'utilisateur.
La commande fournit une interface interactive pour la manipulation des processus. Elle permet de trier les processus par utilisation du CPU, utilisation de la mémoire, etc. (voir manuel de la commande pour plus de détails).

Un exemple d'informations affichées par `top` est donné ci-dessous:

```
laloukil@laloukil:~$ top

top - 11:43:54 up  1:25,  3 users,  load average: 0,24, 0,26, 0,36
Tasks: 204 total,   1 running, 203 sleeping,   0 stopped,   0 zombie
%Cpu(s):  6,5 us,  1,1 sy,  0,2 ni, 90,8 id,  1,4 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   3971652 total,  2766076 used,  1205576 free,   130468 buffers
KiB Swap:  7811068 total,        0 used,  7811068 free.  1406432 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1304 root      20   0  490400 147212 130332 S   6,5  3,7   2:34.87 Xorg
 1814 laloukil  20   0  373696  26828   3080 S   6,5  0,7   0:53.35 ibus-daemon
 1913 laloukil  20   0  205460   3452   2868 S   6,5  0,1   0:03.87 ibus-engine-sim
 4174 laloukil  20   0 1245168 340264  62584 S   6,5  8,6  10:02.43 firefox
 4957 laloukil  20   0  662428  23764  13544 S   6,5  0,6   0:16.60 gnome-terminal
 5959 laloukil  20   0   29172   1596   1096 R   6,5  0,0   0:00.01 top
    1 root      20   0   33908   3200   1476 S   0,0  0,1   0:01.92 init
    2 root      20   0       0      0      0 S   0,0  0,0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0,0  0,0   0:00.10 ksoftirqd/0
```

Les informations affichées par la commande `top` sont organisées en deux zones: *zone de synthèse* et *zone des tâches*.

La zone de synthèse donne des informations sur l'état général du système telles que le nombre total de processus, le nombre de processus en exécution, nombre de processus endormis, etc. Nous mettons l'accent sur la troisième, quatrième et cinquième ligne qui donnent des informations respectivement sur l'utilisation du CPU, de la mémoire physique et de la mémoire virtuelle.

La troisième ligne donne différents pourcentages d'utilisation du CPU. Nous expliquons ci-après la signification des différentes abréviations:

- `ut`: temps passé à exécuter des processus utilisateurs (processus de basse priorité). Sous Linux, l'échelle des priorités d'un processus varie de -20 (priorité la plus élevée) à +19 (priorité la plus basse). Le niveau de priorité par défaut d'un processus est celui de son processus parent, et vaut généralement zéro.
- `sy`: temps passé à exécuter des processus du noyau Linux.
- `ni`: temps passé à exécuter des processus utilisateurs de priorité élevée.
- `id`: temps d'inactivité du processeur.
- `wa`: temps passé à attendre des E/S.
- `hi`: temps passé à servir les interruptions matérielles.
- `si`: temps passé à servir les interruptions logicielles.
- `st`: temps volé par cette VM par l'hyperviseur.

La quatrième ligne reflète la mémoire physique. Les champs affichés sont `total`, `used` (mémoire utilisée incluant le cache disque), `free` (libre) et `buffers` (mémoire utilisée pour les E/S).

La cinquième ligne reflète la mémoire virtuelle. Le champ `cached` est la taille de la mémoire utilisée par le cache disque.

La zone des tâches affiche une liste triée des processus en cours d'exécution sur votre système. Par défaut, la liste est triée par ordre décroissant d'utilisation du processeur; la liste peut cependant être triée sur d'autres champs tels que le taux d'occupation de la mémoire, par exemple. Différents champs peuvent être affichés dans la zone des tâches; les champs par défaut sont:

- `PID`: numéro d'identification du processus.
- `USER`: nom de l'utilisateur qui exécute le processus.
- `PR`: priorité relative du processus.
- `NI`: valeur nice du processus représentant la priorité du processus. Les valeurs négatives donnent une priorité élevées aux processus alors que les valeurs positives donnent des priorités basses aux processus.
- `VIRT`: quantité totale de mémoire utilisée par le processus. Cela inclut le code, les données et les librairies partagées utilisées.
- `RES`: quantité totale la mémoire physique (résidente) utilisée par le processus.
- `S`: état du processus. Les valeurs les plus courantes sont `S` pour "Sleeping" ou `R` pour "Running".
- `%CPU`: pourcentage du temps CPU utilisé par le processus. Ceci est relatif à un seul processeur; dans un système multiprocesseur, il peut être supérieur à 100
- `%MEM`: pourcentage de RAM disponible utilisée par le processus. Ceci n'inclut pas les données qui ont été échangées sur le disque.
- `TIME+`: temps CPU total utilisé par le processus depuis le début. Ce champs compte uniquement le temps que le processus a utilisé le CPU et ne compte pas le temps d'endormissement.
- `COMMAND`: le nom du programme.

Commandes `top` les plus communément utilisées
Il existe plusieurs commandes qui peuvent être émises à `top`. La liste complète peut être consultée dans sa page de manuelle. Nous donnons ci-dessous les commandes les plus communément utilisées:

`q`	Quit.
`h ou ?`	Help.
`s`	Fixer la durée entre les mises à jour de l'affichage.
`espace`	Met à jour l'affichage.
`M`	Trier les processus par taille de mémoire (colonne `%MEM`).
`P`	Trier les processus par activité du CPU (colonne `%CPU`).
`F ou O`	Sélectionner le champs sur lequel la liste des processus sera triée.
`< >`	Déplacer le champs de trie: '<': champs gauche; '>': champs droit .
`u`	Réduire l'affichage aux processus appartenant à un utilisateur spécifique.
`k`	Tuer un processus.

1.4 Le pseudo-système de fichiers `/proc`

Le système de fichiers `/proc` est un pseudo-système de fichiers qui fournit une interface aux structures de données du noyau. Il est généralement monté dans `/proc`. Il contient des informations sur le système à l'exécution (mémoire du système, les périphériques montés, la configuration matérielle, etc.) Il est considéré comme une fenêtre sur le noyau Linux en cours d'exécution. Beaucoup d'utilitaires systèmes sont simplement des appels à des fichiers de `/proc`.

Dans ce qui suit, nous allons décrire quelques fichiers et répertoires sous `/proc`. Pour de plus amples détails, se référer aux pages du manuel de `/proc`.

- `/proc/[pid]`: il y a un sous-répertoire pour chaque processus en exécution. Le nom du sous-répertoire correspond au `PID` du processus.
- `/proc/devices`: affiche les informations sur les périphériques connectés.
- `/proc/modules`: affiche la liste des modules chargés par le système.
- `/proc/mounts`: liste des systèmes de fichiers actuellement montés par le système.
- `/proc/partitions`: contient les numéros de début et de fin, le nombre de blocs et le nom de chaque partition.
- `/proc/pci/devices`: informations sur les périphériques PCI.
- `/proc/cpuinfo`=: donne des informations le processeur (modèle, fréquence, taille du cache, nombre de coeurs, tailles des adresses, etc.)
- `/proc/meminfo`: informations sur la mémoire physique (RAM) et la mémoire d'échange (swap).
- `/proc/filesystems`: liste des systèmes de fichiers supportés par le kernel.

1.5 Exercices

1. Utiliser la commande `ps` pour afficher les processus `bash` de tous les utilisateurs.

2. Utiliser la commande `ps` pour afficher les processus de l'utilisateur `root`. Pour chaque processus, on souhaiterait voir en particulier les colonnes `PID`, `PPID` et `STIME`.

3. Utiliser les commandes `ps` et `head` pour afficher la liste des 5 processus les plus gourmands en CPU.

4. Par défaut, la commande `top` n'affiche pas le champ `PPID`. Comment ajouter ce champ?

5. Avec la commande `pstree`, comment afficher l'arborescence d'un processus de `pid` donné?

6. Comment connaître le nombre cœurs dont dispose le processeur de votre ordinateur?

7. Quelle commande Linux permet d'afficher la version de Linux installé sur votre ordinateur?

8. Quelle commande Linux permet d'afficher la taille de la mémoire physique de votre ordinateur?

2 Les fonctions systèmes `getpid()`, `getppid()`

Il existe dans le langage `C` deux fonctions systèmes qui retournent le PID du processus courant (processus qui appelle la fonction) et le PID du père du processus courant. Ces fonctions sont respectivement `getpid()` et `getppid()`.

Synopsis

```c
#include <sys/types.h>     // pour le type pid_t
#include <unistd.h>      //  pour les fonctions getpid() et getppid()

pid_t getpid(void);
pid_t getppid(void);
```

Exemple 2.1 (`getpid_getppid.c`)
Le programme suivant récupère le PID du processus courant ainsi que le PID de son père et les affiche à l'écran:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
   pid_t my_pid;
   pid_t my_parent_pid;
   my_pid = getpid();        // Retourne le pid du processus appelant
   my_parent_pid = getppid();  // Retourne le pid du pere du processus
     appelant
   printf("Le PID du processus appelant: %d\n", my_pid);
   printf("Le PID du pere du processus appelant: %d\n", my_parent_pid);
   sleep(100);           // Endort le processus appelant pendant 100 sec
   return 0;
}
```

3 Les fonctions systèmes `fork()`, `wait()` et `exec()`

3.1 La fonction `fork()`

La fonction `fork()` permet de cloner un processus. Elle crée un nouveau processus qui est une copie du processus qui exécute `fork()` (processus appelant). Le processus appelant est appelé *processus père*, le processus créé par le processus père est appelé *processus fils*. Au moment de l'appel de `fork()`, les espaces mémoires du père et du fils ont le même contenu. Par la suite, les écritures mémoires effectuées par l'un des processus n'affectent pas la mémoire de l'autre processus.

Le processus fils est une copie exacte du processus père sauf sur les points suivants (voir le manuel de fork() pour plus de détails):

- Le processus fils n'hérite pas des verrous mémoire du père.
- Les utilisations des ressources de processus et les compteurs de temps CPU sont remises à zéro dans le fils.
- L'ensemble des signaux en attente dans le fils est initialement vide.
- Le fils n'héritent pas des opérations d'E/S asynchrones de son père
- Le fils n'hérite pas des ajustements sémaphore de son père.
- L'enfant n'hérite pas des timers de son père.

Synopsis

```
#include <unistd.h>

pid_t fork(void);
```

En cas de succès, `fork()` retourne le PID du processus fils dans le processus père, et la valeur 0 dans le processus fils. La valeur retournée par `fork()` permet donc au programmeur de distinguer la partie de code exécutée par le processus père de celle exécutée par le processus fils.
En cas d'échec, -1 est retourné dans le père et aucun processus fils n'est créé.

Exemple 3.1 (`fork1.c`)
Dans l'exemple suivant, le processus `main` crée un processus fils, affiche `Hello`, s'endort pendant 100 secondes et se termine. Le processus fils créé par `fork()` affiche également `Hello`, s'endort pendant 100 secondes et se termine. Vous allez constater que deux `Hello` seront affichés à l'écran.

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

int main (){
   fork();
   printf("Hello du processus %d dont le pere est %d\n", getpid(), getppid
     ());
   sleep(100);
   return 0;
}
```

Exemple 3.2 (`fork2.c`)
Dans cet exemple, le processus `main` crée un processus fils, affiche son PID, s'endort pendant 600 secondes, affiche un message de fin d'exécution et se termine. En cas de succès de l'appel `fork()`, le processus fils affiche son PID, s'endort pendant 10

secondes, affiche un message de fin d'exécution et se termine. En cas d'échec de l'appel de `fork()`, un message d'erreur est affiché et le programme se termine.

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

int main (void){
  pid_t pid ;
  pid = fork();
  switch (pid) {
  case -1 :      // echec dans fork()
    fprintf(stderr,"echec du fork.\n");
    exit(1);
    break;
  case 0 :       // pid == 0: partie du code executee par le fils
    fprintf (stdout, "Fils: je demarre. Mon PID est %u.\n", getpid());
    sleep(10) ;
    fprintf (stdout, "Fils: je termine.\n");
    exit(0);
    break;
  default :      // pid > 0: partie du code executee par le processus pere
    fprintf (stdout, "Pere: je demarre. Mon PID est %u.\n", getpid());
    sleep(600) ;  //  Simuler une execution de 600 sec. */
    fprintf (stdout, "Pere: je termine.\n");
    exit (0);
    break;
  }
  return(0);
}
```

3.2 Les fonctions systèmes `wait()` et `waidpid()`

Les fonctions `wait()` et `waitpid()` permettent de suspendre le processus appelant jusqu'à ce que l'un de ses processus fils se termine.

Synopsis:

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Le processus qui appelle `wait()` est suspendu jusqu'à ce que l'un de ses fils se termine ou jusqu'à ce qu'un signal à intercepter arrive. Lorsqu'un fils se termine, le processus père est réveillé et la fonction retourne le PID de ce fils. Si, au moment de l'appel

`wait()`, un processus fils s'est déjà terminé (il est dans l'état zombie), la fonction revient immédiatement et toutes les ressources utilisées par le fils sont libérées. Si ce processus n'a pas de fils, la fonction retourne -1.

La fonction `waitpid()` suspend l'exécution du processus appelant jusqu'à ce que le processus fils de numéro `pid` (1er paramètre de `waitpid()`) se termine ou jusqu'à ce qu'un signal à intercepter arrive. Si le fils de numéro `pid` s'est déjà terminé au moment de l'appel (il est dans l'état zombie), la fonction revient immédiatement et toutes les ressources utilisées par le fils sont libérées. En cas de réussite du `wait()` et `waitpid()`, le PID du fils qui s'est terminé est renvoyé. En cas d'echec, -1 est renvoyé et `errno` contient le code d'erreur.

Le paramètre `pid` peut prendre l'une des valeurs suivantes :

- $<$ -1 : attendre la fin de n'importe quel processus fils appartenant à un groupe de processus d'ID `pid`.
- -1 : attendre la fin de n'importe quel fils. C'est le même comportement que `wait()`.
- 0 : attendre la fin de n'importe quel processus fils du même groupe que l'appelant.
- $>$ 0 : attendre la fin du processus numéro `pid`.

Si le paramètre `status`'information sur la terminaison du fils. Pour plus de détails, se référer aux pages du manuel des fonctions `wait()` et `waitpid()`.

Exemple 3.3 (`fork3.c`)
Dans cet exemple, le processus père crée un processus fils. Le processus fils affiche "Fils: je demarre. Mon PID est ...", s'endort pendant une durée aléatoire entre 0 et 20 secondes puis affiche "Fils: je termine et je sors." et sort. Le processus père affiche "Pere: J'attends la terminaison de mon fils ..." et attend que le fils termine auquel cas il affiche "Pere: Mon fils ... s'est termine. Je sors." et sort.

```
1  #include <stdio.h>
2  #include <sys/types.h>     // pour le type pid_t
3  #include <sys/wait.h>      // pour l'appel de la fonction wait()
4  #include <unistd.h>      // pour l'appel de la fonction getpid()
5  #include <stdlib.h>      // pour l'appel de la fonction exit()
6
7  int main (void){
8    pid_t pid ;
9    fprintf (stdout, "Pere: je demarre. Mon PID est %u.\n", getpid());
10   pid = fork();
11   switch (pid) {
12     case -1 :      // echec de fork()
13       fprintf(stderr,"echec du fork.\n");
14       exit(1);
```

```
15        break;
16      case 0 :       // pid == 0: partie du code executee par le fils
17        fprintf (stdout, "Fils: je demarre. Mon PID est %u.\n", getpid());
18        sleep(rand()%20) ;
19        fprintf (stdout, "Fils: je termine et je sors.\n");
20        exit(0);
21        break;
22      default :      // pid > 0: partie du code executee par le pere
23        fprintf (stdout, "Pere: J'attends la terminaison de mon fils %u\n",
       pid);
24        wait(NULL);
25        fprintf (stdout, "Pere: Mon fils %u s'est termine. Je sors.\n", pid
      );
26        exit (0);
27        break;
28    }
29    return(0);
30  }
```

3.3 Les fonctions `exec()`

Lorsqu'un processus crée un processus fils avec `fork()`, le processus fils hérite et exécute le même code que le processus père. Souvent, on a besoin de faire exécuter par le processus fils un programme différent du programme exécuté par le père. Ceci peut être réalisé avec les appels systèmes `exec()`. Lorsqu'un processus exécute `exec()`, il remplace le code courant par le code du programme dont le chemin est indiqué en premier paramètre de la fonction `exec()`.
Dans ce recueil de travaux pratiques, nous nous focalisons sur 2 fonctions: `execl()` et `execv()`.

Synopsis

```
1 #include <unistd.h>
2
3 int execl (const char* path, const char *arg0, ..., char *argn);
4 int execv (const char* path, char * const argv[]);
```

Ces fonctions renvoient –1 en cas d'échec. Les fonctions `execl()` et `execv()` sont identiques et ne diffèrent que de la façon dont les arguments sont fournis à la fonction.

Pour `execl()`, les arguments sont fournis sous forme d'une liste terminée par le pointeur `NULL`:

- `path`: chaîne de caractères indiquant le chemin absolu du nouveau programme à charger et à exécuter.

- `arg0, arg1, ..., argn`: les arguments du programme : `arg0` reprend le nom du programme. `arg1, ..., argn-1` sont les arguments du programme et `argn=NULL`.

Pour la fonction `execv()`, les arguments sont fournis dans un vecteur de chaînes de caractères dont le dernier argument est la valeur `NULL` :

- `path`: chaîne de caractères donnant le chemin absolu du nouveau programme à substituer et à exécuter.
- `argv[]`: la liste des arguments.

En cas de succès de l'appel de `execl()`(ou `execv()`), le processus charge puis exécute le code du programme dont le chemin est indiqué comme premier argument de l'appel `exec()` et les instructions qui suivent l'appel de `exec()` ne sont pas exécutées. En cas d'échec de recouvrement de code (renvoie de la valeur -1), l'exécution se poursuit à partir de l'instruction qui suit l'appel de `exec()`.

Il existe de nombreuses autres fonctions de la famille de fonctions `exec()`: `execle()`, `execve()`, `execlp()`, `execvp()`. Pour plus de détails, consulter le manuel de ces fonctions.

Exemple 3.4 (`exec1.c`)
Dans cet exemple, le processus courant est remplacée par le processus `ls`. En cas de succès de l'appel de `execl()`, c'est la commande `ls -l` qui sera exécutée. En cas d'échec, c'est le message "Erreur lors de l'exécution de ls." qui sera affiché à l'écran.

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

int main(){
   execl("/bin/ls", "ls", "-l", NULL);
   printf("Erreur lors de l'execution de ls. \n");
   return 0;
}
```

Exemple 3.5 (`exec2.c`)
Le même exemple que précédemment avec l'utilisation de la fonction `execv()`.

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

#define NMAX 5

int main () {
   char *argv [NMAX];
   argv [0] = "ls";
   argv [1] = "-l";
```

```
10    argv [2] = NULL;
11    execv ("/bin/ls", argv);
12    printf("Erreur lors de l'execution de ls. \n");
13    return 0;
14  }
```

Exemple 3.6 (`fork-exec.c`)
Dans cet exemple, le processus père crée deux processus fils. Fils 1 exécute la commande Linux `/bin/ls -la`, Fils 2 exécute la commande Linux `/bin/ps`. En cas d'échec d'exécution de la fonction `execl()`, le message `"Echec exec."` est affiché.

```
1
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <time.h>
7  #include <sys/wait.h>
8
9  int main() {
10   pid_t pid;
11   srand(time(NULL));
12
13   printf("Pere: je cree le 1er fils.\n");
14         pid = fork();
15         if (pid == 0){
16     printf("Fils 1: je vais executer la commande /bin/ls.\n");
17     sleep(rand()%5);
18     execl("/bin/ls", "-la", NULL);
19     printf("Echec exec. \n");
20   }
21
22   printf("Pere: je cree le 2eme fils.\n");
23   pid = fork();
24         if (pid == 0){
25     printf("Fils 2: je vais executer la commande /bin/ps.\n");
26     sleep(rand()%5);
27     execl("/bin/ps", "-e", NULL);
28     printf("Echec de exec.\n");
29         }
30
31   printf("Pere: J'attends la fin de mes 2 fils.\n");
32   wait(NULL);
33   wait(NULL);
34   printf("Pere: Mes 2 fils ont termines. Bye.\n");
35   return 0;
36 }
```

3.4 Exercices

1. Considérons le programme `C` suivant (`ex1-fork.c`):

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

int main (){
  pid_t pid ;
  printf("Je suis le processus %d\n", getpid());
  sleep(10);
  printf("Appel de fork.\n");
  pid = fork();
  printf("fork() retourne %d\n", pid);
  sleep(10);
  return 0;
}
```

   (a) Compilez puis exécutez le programme. Que retourne `fork()`?

   (b) Exécutez à nouveau le programme, ouvrez un second terminal et exécutez la commande `ps -a` avant l'appel de `fork()` puis après l'appel de `fork()`. Remarquez que `fork()` duplique le processus.

2. Soit le programme `C` suivant (`ex2-fork.c`):

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

int i;
int main (){
  i = 1;
  pid_t pid;
  pid = getpid();
  printf("Avant fork: Je suis le processus main, mon pid est = %d\n"
    ,pid);
  sleep(20);
  printf("Appel de fork par le processus main de PID = %d...\n",
    getpid());
  pid = fork(); // Clonage du processus
  switch(pid) {
  case -1:   // pid = -1: erreur du fork()
      fprintf(stderr,"erreur du fork.\n");
      exit(1);
      break;

  case 0:    // pid = 0 : on est dans le fils
```

```
22        fprintf(stdout,"Fork reussi... Je suis le processus fils cree
      par fork: mon PID est = %d, le PID de mon pere est = %d\n",
      getpid(), getppid());
23        sleep(30);
24        fprintf (stdout, "Fin du fils (PID=%d), i=%d...\n", getpid(),
      i);
25        exit (0);
26        break;
27
28    default:   // pid > 0: on est dans le pere
29        // Modification de la variable i par le pere
30        i = 2;
31        fprintf (stdout,"Je suis le processus pere, je viens de
      modifier la variable i, i= %d\n", i);
32        sleep (10);
33        printf ("Fin du pere (PID=%d), i = %d...\n",getpid(), i);
34        exit (0);
35        break;
36    }
37 }
```

(a) Compilez puis exécutez le programme.

(b) Quelle est la valeur de la variable `i` affichée dans le fils? Quelle est la valeur de la variable `i` affichée dans le père? Sont-elles égales? Pourquoi?

3. Écrire un programme `C` dans lequel le processus crée un fils, affiche son PID et le PID du processus fils créé, s'endort pendant 30 secondes (`sleep(30)`) et se termine (`exit(0)`). Le processus fils affiche son PID, appelle la fonction `executerQuelqueChose()`, affiche son PID et se termine.

   Le code de la fonction `executerQuelqueChose()` est donné ci-dessous (`executerQuelqueChose.c`):

```
1 #define NB_ITERS 5
2
3 void executerQuelqueChose (void){
4   for (int i = 0; i < NB_ITERS; i++){
5     printf("i = %d\n", i);
6     sleep(rand()%4);
7   }
8 }
```

(a) Exécutez le programme, ouvrez un second terminal et exécuter la commande `ps -a`. Que remarquez-vous?

(b) Mettez en commentaire l'instruction `sleep(30)` et exécuter la commande `ps -a`. Que remarquez-vous ?

4. Pour chacun des fragments de programmes `C` suivants et avant de l'exécuter, dites combien de `"Hello!"` sont affichés à l'écran et combien de processus sont créés. Dessinez l'arborescence des processus créés.

   (a) `hello1.c`

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

int main() {
  printf ("Hello!\n");
  sleep(30);
  return 0;
}
```

   (b) `hello2.c`

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

int main() {
  printf ("Hello!\n");
  fork();
  sleep(30);
  return 0;
}
```

   (c) `hello3.c`

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

int main() {
  fork();
  printf ("Hello!\n");
  sleep(30);
  return 0;
}
```

   (d) `hello4.c`

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

int main() {
  fork();
  fork();
  printf ("Hello!\n");
  sleep(30);
```

```
  return 0;
}
```

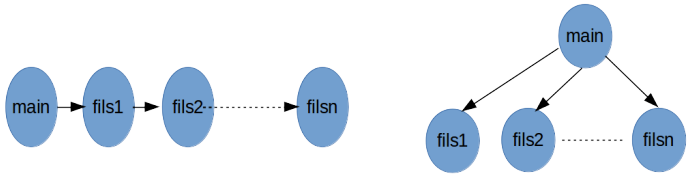
(e) `hello5.c`

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

int main() {
  pid_t pid;
  for (int i = 1; i <= 2; ++i) {
    pid = fork();
    if (pid != 0)
      printf ("Hello!\n");
  }
  sleep(30);
  return 0;
}
```

(f) `hello6.c`

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

int main() {
  pid_t pid;
  for (int i = 1; i <= 2; ++i) {
    pid = fork();
    if (pid == 0)
      break; // Fin de la boucle for
    else
      printf("Hello!\n");
  }
  sleep(30);
  return 0;
}
```

5. Écrire les programmes qui permettent de créer les arborescences des processus schématisées ci-dessous. La variable entière `n` (initialisée dans le programme ou lue à partir de la ligne de commande) représente le nombre de processus fils à créer.



6. Écrire un programme `C` dans lequel le processus père crée deux processus fils et s'endort pendant 10 secondes. Le 1er fils affiche les nombres paires entre 0 et 100, le second fils affiche les nombres impaires entre 0 et 100. A chaque itération, le processus fils affiche son PID, le PID de son père ainsi que le nombre.

7. Écrire un programme `C` dans lequel le processus `main` crée `N` processus fils (`N` étant initialisé dans le programme ou lue à partir de la ligne de commande) puis attend la terminaison de ses `N` processus fils avant de se terminer. A chaque terminaison d'un fils, le processus père affiche son PID (le PID du fils terminé). Pour rappel, la fonction `wait()` renvoie la valeur -1 quand il n'y a pas de processus fils en exécution.

   Chaque processus fils `Pi` affiche son PID et le PID de son père, s'endort pendant une durée aléatoire (entre 0 et 10 secondes) et se termine (`exit (i)`).

   Compiler le programme avec l'option `-Wall`.

8. Écrire un programme où le processus `main` crée 2 processus fils, l'un exécute le programme `/bin/ls` avec l'option `"-l -a"`, l'autre exécute le programme `/bin/ps` avec l'option `"-x"`, le processus `main` se termine immédiatement après la création des 2 processus. Utiliser dans un premier temps la fonction `execl()`.

9. Refaire l'exercice 8 avec le processus père qui attend la terminaison de ses 2 fils, affiche le PID du processus qui termine puis se termine.

10. Refaire l'exercice 9 en utilisant la fonction `execv()`.

4 Le multithreading

4.1 Introduction

Les systèmes d'exploitations modernes tels que Windows, Unix, Linux ou MacOS sont des systèmes multitâches. Ils peuvent exécuter plusieurs processus simultanément en partageant les ressources de l'ordinateur (CPUs, mémoire principale, canaux d'E/S, etc.) entre les différents processus. Le multitâche permet ainsi une meilleure exploitation des ressources de l'ordinateur. Quand la machine est mono-processeur (ce qui est rare de nos jours car les processeurs actuels disposent de plusieurs coeurs de calcul), celui-ci est partagé entre les différents processus et, à tout instant, un seul processus est exécuté à la fois. Dans les machines multi-processeurs, plusieurs processus peuvent être exécutés en parallèle. Chaque processus en exécution possède son propre segment de code, segment de données, ses registres, sa pile d'exécution, etc. (voir Fig. 1).

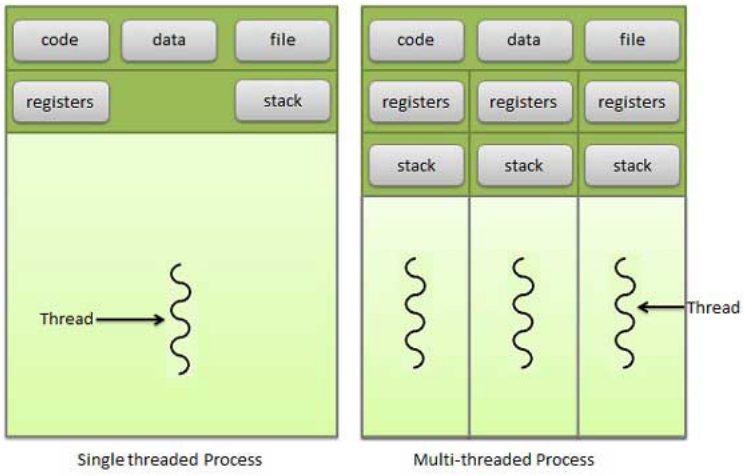


Figure 1: Processus mono-thread vs Processus multi-thread [http://www.tutorialspoint.com/operating_system/os_multi_threading.htm].

Les systèmes d'exploitation modernes permettent également le "multithreading", c'est à dire la possibilité de lancer plusieurs fils d'exécution (*threads*) concurrents au sein d'un même processus, chaque thread exécute une certaine tâche. Les threads sont parfois appelés *processus légers* (par opposition au processus qui les contient et qui est appelé *processus lourd*) car ils s'exécutent dans le contexte du processus et utilisent les ressources qu'il a allouées. Les threads d'un processus partagent l'espace d'adressage de celui-ci ce qui signifie qu'ils peuvent communiquer entre eux par le biais de cette mémoire partagée. Les threads ne partagent cependant pas les registres, le compteur de programme, la pile d'exécution du processus.

Le multithreading améliore la performance du programme en optimisant l'utilisation des ressources du système. Par exemple, quand un thread est bloqué (par exemple, en attendant l'achèvement d'une opération d'E/S), un autre thread peut utiliser le

temps CPU pour effectuer des calculs, ce qui améliore la performance globale du programme. Le multithreading permet aussi une meilleure interactivité du programme avec les utilisateurs. Par exemple, dans un traitement de texte, pendant qu'un thread fait l'impression ou l'enregistrement d'un fichier, un autre peut être utilisé pour continuer à lire les caractères tapés au clavier.

4.2 Les threads POSIX (Pthreads)

1. Création et lancement d'un thread: la fonction `pthread_create()`

```
#include <pthread.h>

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

   Permet de créer un thread et de lui associer une fonction à exécuter (`thread_fonction`). Si la fonction du thread nécessite des paramètres, ceux-ci sont passés dans le dernier argument de `pthread_create()` (voir exemples de la section **??**).

2. Joindre un thread: la fonction `pthread_join()`

```
#include <pthread.h>

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

   Permet au thread appelant de joindre (attendre la terminaison) du thread indiqué dans le premier argument de la fonction `pthread_join()`. Le second argument permet de récupérer la valeur de retour du thread.

3. Terminaison d'un thread: la fonction `pthread_exit()`

```
#include <pthread.h>

void pthread_exit(void *retval);
```

   Termine l'exécution d'un thread et envoie une valeur de retour dans la variable `*retval`.

Exemple 4.1 (`pthread1.c`)
Dans le programme suivant, le processus `main` crée et démarre `NUMTHREADS` threads et les joint avant de se terminer. Chaque thread exécute la fonction `afficheHello()` qui reçoit en argument le rang du thread et affiche la chaîne `"Hello World du thread #x!"` (`x` étant le rang du thread).

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

#define NUMTHREADS 5

void *afficheHello(void *threadid) {
  long tid;
  tid = (long)threadid;
  printf("Hello World du thread #%ld!\n", tid);
  pthread_exit(0);
}

int main (int argc, char *argv[]) {
  pthread_t threads[NUMTHREADS];
  int rc;
  long th;
  for (th = 0; th < NUMTHREADS; th++){
    printf("main: creation du thread %ld\n", th);
    rc = pthread_create(&threads[th], NULL, afficheHello, (void *)th);
    if (rc){
      printf("Erreur de creation de thread; code erreur = %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(0);
}
```

Exemple 4.2 (`pthread2.c`): Passage de paramètres à la fonction du thread
Dans cet exemple, le processus main emballe les arguments `thread_id` et `message` dans la structure `thread_args` et la passe à la fonction `afficheHello()`.

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

#define NB_THREADS  3

struct thread_args{
   int  thread_id;
   char *message;
};

void *afficheHello(void *threadarg) {
   struct thread_args *th_args;
   th_args = (struct thread_args *) threadarg;
   printf("Thread ID : %d, Message : %s\n", th_args->thread_id, th_args->
   message) ;
   pthread_exit(NULL);
}

```

```c
int main (){
   pthread_t threads[NB_THREADS];
   struct thread_args td[NB_THREADS];
   int rc;
   int i;

   for(i=0; i < NB_THREADS; i++){
      printf("main() : creation du thread %d\n", i);
      td[i].thread_id = i;
      td[i].message = "Ceci est un message";
      rc = pthread_create(&threads[i], NULL, afficheHello, (void *)&td[i
   ]);
      if (rc){
         printf("Erreur de creation du thread.\n");
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Exemple 4.3 (`pthread3.c`): Jointure du processus avec ses threads

```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){
    printf ("pthread_create error for thread 1\n");
    exit (1);
  }
  if (pthread_create (&th2, NULL, my_thread_process, "2") < 0){
    printf ("pthread_create error for thread 2\n");
    exit (1);
  }
  (void) pthread_join (th1, &ret);
  (void) pthread_join (th2, &ret);
}
```

4.3 Exercices

1. Écrire un programme `C` dans lequel le processus père (`main`) crée `NbThreads` threads (la valeur de `NbThreads` est passée en ligne de commande ou initialisée dans le programme), chaque thread exécute la fonction `fonctionThread()`. Le processus `main` attend la fin de tous ses threads avant de se terminer.

   Le code de la fonction exécutée par les threads est donné ci-dessous (`fonctionThread.c`):

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

void fonctionThread(void){
  pthread_t mon_tid;
  mon_tid = pthread_self();
  int i, nbreIters;
  nbreIters = rand()%100;
  i = 0;
  printf("Thread (0x)%x : DEBUT\n", (int)mon_tid);
  while(i < nbreIters) i++;
  printf("Thread (0x)%x : FIN\n", (int)mon_tid);
  pthread_exit(NULL);
}
```

2. **Somme parallèle des éléments d'une matrice** `(somme-parallele-elements-matrice.c)`.
   Écrire un programme où le processus `main` initialise aléatoirement une matrice réelle carrée de dimension `N` (la valeur de `N` est initialisée dans le programme) et démarre `N` threads. Le thread `i` (`i = 0, 1, .., N-1`) effectue la somme des éléments de la ligne `i` et sauvegarde le résultat dans `vect[i]` où `vect` est un vecteur partagé de taille `N`. Le processus attend la terminaison de tous les threads pour faire la somme des éléments de `vect` et afficher le résultat.

3. **Produit parallèle de deux matrices**.
   Écrire un programme multi-thread qui réalise la multiplication parallèle d'une matrice `A(n,k)` par une matrice `B(k,m)` dans une matrice `C(n,m)`. Les `n` lignes de la matrice `A` sont réparties de manière égale entre `NUMTHREADS` threads et chacun des `NUMTHREADS` calcule un bloc de `n/NUMTHREADS` lignes de la matrice produit `C`. On suppose que `n` $\gg$ `NUMTHREADS`.

5 Les mutex et les sémaphores

5.1 Les mutex

Un mutex (MUTual EXclusion) est un structure utilisée pour assurer l'exclusion mutuelle (accès non concurrent) pour l'accès aux sections critiques (parties de code qui modifient des variables partagées entre plusieurs threads). Un mutex peut être dans deux états : déverrouillé (pris par aucun thread) ou verrouillé (appartenant à un thread). Un mutex ne peut être pris que par un seul thread à la fois. Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.

1. **Initialisation d'un objet mutex**: la fonction `pthread_mutex_init()`

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *mutex_attr);
```

   Initialise le mutex pointé par `mutex` selon les attributs spécifiés par `mutex_attr`. Si `mutex_attr` vaut NULL, les paramètres par défaut sont utilisés. Une variable mutex peut être initialisée de manière statique en utilisant la constante `PTHREAD_MUTEX_INITIALIZER`.

2. **Verrouillage d'un mutex**: la fonction `pthread_mutex_lock()`

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

   `pthread_mutex_lock()` permet de verrouiller le mutex indiqué en paramètre. Si le mutex est déverrouillé, il devient verrouillé et est possédé par le thread appelant et `pthread_mutex_lock` rend la main immédiatement. Si le mutex est déjà verrouillé par un autre thread, `pthread_mutex_lock()` suspend le thread appelant jusqu'à ce que le mutex soit déverrouillé.

3. **Déverrouillage d'un mutex**: la fonction `pthread_mutex_unlock()`

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

   `pthread_mutex_unlock()` libère l'objet mutex pointé par `*mutex`. La manière dont un mutex est libéré dépend du type d'attribut du mutex. Si plusieurs threads sont bloqués sur l'objet mutex référencé par `*mutex` au moment de l'appel de `pthread_mutex_lock()`, le thread qui va acquérir le mutex dépend de l'algorithme d'ordonnancement implémenté.

4. **Comment utiliser un mutex?**
   Soit `x` une ressource (une variable, un fichier, ...) partagée entre plusieurs threads. Celle-ci peut être modifiée par plusieurs threads. `x` doit donc être protégée par un mutex en entourant tous les accès en modification de `x` par la paire d'appels `pthread_mutex_lock()` et `pthread_mutex_unlock()` comme indiqué ci-dessous:

```
int x;
pthread\_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mut); // Verrouillage du mutex
  ..................      // Section critique : partie de code qui
  ..................      // modifie la variable partagee x
pthread_mutex_unlock(&mut);// Deverrouillage du mutex
```

5.2 Exemple

Dans cet exemple, le processus `main` crée et démarre 2 threads, `thread1` et `thread2`, `thread1` (resp. `thread2`) incrémente (resp. décrémente) 100 000 fois la variable partagée `count`. Le processus `main` joint les deux threads avant de se terminer.

- **Version sans mutex** (`mutex_inc-dec-wo-mutex.c`)

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

void* incFonction();
void* decFonction();

int count = 0;     // variable partagee entre les threads

int main() {
   srand(time(0));
   int rc1, rc2;
   pthread_t thread1, thread2;

   // Creation des 2 threads
   if ((rc1 = pthread_create(&thread1, NULL, &incFonction, NULL))){
      printf("Echec de creation du Thread 1: %d\n", rc1);
   }

   if ((rc2 = pthread_create( &thread2, NULL, &decFonction, NULL))){
      printf("Echec de creation du Thread 2: %d\n", rc2);
   }

   // Attendre que les threads se terminent
   pthread_join(thread1, NULL);
```

```
26      pthread_join(thread2, NULL);
27      printf("Main --> valeur de count: %d\n", count);
28      exit(0);
29  }
30
31  void* incFonction(){
32    int i;
33    for (i = 0; i < 100000; i++) {
34       count ++;
35    }
36    pthread_exit(0);
37  }
38
39
40  void* decFonction(){
41    int i;
42    for (i = 0; i < 100000; i++) {
43       count --;
44    }
45    pthread_exit(0);
46  }
```

Exécuter plusieurs fois le programme et remarquer que la valeur finale de compteur change d'une exécution à une autre. Expliquer pourquoi.

- **Version avec mutex** (`mutex_inc-dec-with-mutex.c`)
  Dans cette version, on encadre les instructions d'incrémentation et de décrémentation de la variable partagée `count` par `pthread_mutex_lock()` et `pthread_mutex_unlock()` pour interdire la modification concurrente de cette variable par `thread1` et `thread2`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  void* incFonction();
6  void* decFonction();
7
8  pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
9  int count = 0;  // variable partagee entre les threads
10
11 int main() {
12    srand(time(0));
13    int rc1, rc2;
14    pthread_t thread1, thread2;
15
16    // Creation des 2 threads
17    if ((rc1 = pthread_create(&thread1, NULL, &incFonction, NULL))){
18       printf("Echec de creation du Thread 1: %d\n", rc1);
19    }
20
21    if ((rc2 = pthread_create( &thread2, NULL, &decFonction, NULL))){
```

```
22          printf("Echec de creation du Thread 2: %d\n", rc2);
23      }
24
25      // Attendre que les threads se terminent
26      pthread_join(thread1, NULL);
27      pthread_join(thread2, NULL);
28      printf("Main --> valeur de count: %d\n", count);
29      exit(0);
30  }
31
32  void* incFonction(){
33    int i;
34    for (i = 0; i < 100000; i++) {
35        pthread_mutex_lock( &mutex1 );
36        count ++;
37        pthread_mutex_unlock( &mutex1 );
38    }
39    pthread_exit(0);
40  }
41
42  void* decFonction(){
43    int i;
44    for (i = 0; i < 100000; i++) {
45        pthread_mutex_lock( &mutex1 );
46        count --;
47        pthread_mutex_unlock( &mutex1 );
48    }
49    pthread_exit(0);
50  }
```

Exécuter plusieurs fois le programme et remarquer que la valeur finale de `count` est tout le temps nulle.

5.3 Les sémaphores

Un sémaphore POSIX est un mécanisme de synchronisation de threads d'un processus ou de processus fils d'un processus parent. C'est une variable de type `sem_t` (défini dans `semaphore.h`) qui doit être initialisée à une valeur positive et munie de fonctions pour éventuellement suspendre (`sem_wait()`) ou activer (`sem_post()`) un thread/processus.

1. **Initialisation d'une variable sémaphore**: la fonction `sem_init()`

```
1  #include <semaphore.h>
2
3  int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Initialise le sémaphore pointé par `sem`. Le compteur associé au sémaphore est initialisé à `value`. L'argument `pshared` initialisé à 0 indique que le sémaphore est local au processus et permet de synchroniser les threads du processus uniquement.

`pshared` initialisé à une valeur non nulle indique que le sémaphore est partagé par les processus fils d'un processus.

2. **Suspension d'un thread/processus**: la fonction `sem_wait()`

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

Suspend le thread appelant jusqu'à ce que le sémaphore pointé par `sem` ait une valeur non nulle. Lorsque le compteur devient non nul, le compteur du sémaphore est atomiquement décrémenté.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

C'est une variante non bloquante de `sem_wait()`. Si le sémaphore pointé par `sem` est non nul, le compteur est décrémenté atomiquement et la fonction retourne 0. Si le compteur du sémaphore est à 0, la fonction retourne `EAGAIN`.

3. **Activation d'un thread/processus**: la fonction `sem_post()`

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Incrémente (déverrouille) le sémaphore pointé par `sem`. Si la valeur du sémaphore devient par conséquent supérieure à 0 alors un autre processus ou thread bloqué dans un `sem_wait` va être réveillé et procède à verrouiller le sémaphore.

4. **Récupération de la valeur d'un sémaphore**: la fonction `sem_getvalue()`

```
#include <semaphore.h>

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

Sauvegarde dans la variable pointée par sval la valeur courante du compteur du sémaphore sem.

5. **Destruction et libération des ressources d'un sémaphore**: la fonction `sem_destroy()`

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Détruit un sémaphore et libère toutes les ressources qu'il possède. Dans LinuxThreads, on ne peut pas associer de ressource à un sémaphore donc cette fonction ne fait que vérifier qu'aucun thread n'est bloqué sur le sémaphore.

5.4 Exemples

1. Le programme suivant crée et démarre 2 threads `th1` et `th2`. `th1` écrit dans le tableau d'entiers `tab`, `th2` lit le même tableau `tab`. Le sémaphore `S` initialisé à 0 empêche le thread `th2` de lire `tab` avant que le thread `th1` ne termine l'écriture dans `tab` (`semaphore_read-write-tab.c`).

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

#define N 5
int tab[N];
sem_t S;

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

  void *lireTab (void *);
  void *ecrireTab (void *);
  srand(time(NULL));

  // Initialisation de la variable semaphore S
  if (sem_init(&S,0,0)) {
    perror("seminit");
    exit(EXIT_FAILURE);
  }

  if (pthread_create (&th1, NULL, ecrireTab, NULL) < 0) {
    perror("Thread ecrire (pthread_create)");
    exit (-1);
  }

  if (pthread_create (&th2, NULL, lireTab, NULL) < 0) {
    perror("Thread lire (pthread_create)");
    exit (-1);
  }

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

  // Destruction du semaphore S et liberation des ressources
  sem_destroy(&S);
```

```
40
41    return 0;
42  }
43
44  void *lireTab (void * arg){
45    sem_wait(&S);
46    for (int i = 0 ; i != N ; i++)
47      printf ("Thread lecture: tab[%d] vaut %d\n", i, tab[i]);
48    pthread_exit (0);
49  }
50
51  void *ecrireTab (void * arg){
52    for (int i = 0 ; i != N ; i++) {
53      tab[i] = 2 * i;
54      printf ("Thread ecriture: tab[%d] vaut %d\n", i, tab[i]);
55      sleep(rand()%2); // Simule un calcul complexe
56    }
57    sem_post(&S);
58    pthread_exit (0);
59  }
```

2. Producteur-consommateur (`semaphore_prod-cons.c`)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <time.h>
5  #include <pthread.h>
6  #include <semaphore.h>
7
8  #define SIZE 5    // taille du buffer
9
10 int buffer[SIZE];   // buffer des produits (valeurs)
11 int tail, head;   // indices de tete et de queue du buffer
12
13 // semaphores representant respectivement le nombre de places libres
       et le nombre de produits
14 sem_t empty, products;
15
16 void *producer(void *arg) { // fonction producteur
17   int valeur;
18   do { // produire une valeur (produit) jusqu'a ce que la valeur est
       0
19     int valeur = rand() % 100; // generer une valeur aleatoire entre
       0 et 99
20     printf("produit > %d\n", valeur);
21     sem_wait(&empty); // attendre une place libre dans le buffer
22     buffer[tail] = valeur; // inserer une nouvelle valeur dans le
      buffer
```

```
23      tail = (tail + 1) % SIZE;
24      sem_post(&products); // incrementer le nombre de produits dans
        le buffer
25      sleep(rand() % 5); // marquer une pose de duree aleatoire (entre
        0 et 4s) apres la production
26    }
27    while(valeur);
28    pthread_exit(NULL); // terminer le thread producteur
29  }
30
31  void *consumer(void *arg) {//fonction consommateur
32    int valeur;
33    do {// consomme une valeur (produit) jusqu'a ce que la valeur
        consommee est 0
34      sem_wait(&products); // attendre pour des produits dans le
        buffer
35      valeur = buffer[head]; // prendre la valeur d'indice head dans
        le buffer
36      head = (head + 1) % SIZE;
37      printf("\tconsomme < %d\n", valeur);
38      sem_post(&empty); // incrementer le nombre de places libres dans
        le buffer
39      sleep(rand() % 5); // simuler la consommation du produit
40    } while(valeur);
41    pthread_exit(NULL); // terminer le thread consommateur
42  }
43
44  int main(int argc, char *argv[]) {
45    // declaration des threads producteur et consommateur
46    pthread_t prod_t, cons_t;
47    srand(time(0)); // initialiser le generateur de nombres aleatoires
48    // initialiser les semaphores
49    sem_init(&empty, 0, SIZE);
50    sem_init(&products, 0, 0);
51    // creer les threads producteur et consommateur
52    pthread_create(&prod_t, NULL, producer, NULL);
53    pthread_create(&cons_t, NULL, consumer, NULL);
54    // attendre la terminaison des threads producteur et consommateur
55    pthread_join(prod_t, NULL);
56    pthread_join(cons_t, NULL);
57    // detruire les semaphores
58    sem_destroy(&empty);
59    sem_destroy(&products);
60    return 0;
61  }
```

3. Problème des lecteurs-rédacteurs: priorité des lecteurs sur les rédacteurs (`semaphore_readers-writers.c`)

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

//* Priorite des lecteurs sur les redacteurs */

sem_t mutex,writeblock;
int data = 0,rcount = 0;

void *reader(void *arg)
{
  long f;
  f = ((long)arg);
  sem_wait(&mutex);
  rcount = rcount + 1;
  if(rcount==1)
   sem_wait(&writeblock);
  sem_post(&mutex);
  printf("Donnee lue par le lecteur%ld is %d\n",f,data);
  sleep(1);
  sem_wait(&mutex);
  rcount = rcount - 1;
  if(rcount==0)
   sem_post(&writeblock);
  sem_post(&mutex);
  pthread_exit(0);
}

void *writer(void *arg)
{
  long f;
  f = ((long) arg);
  sem_wait(&writeblock);
  data++;
  printf("Donnee ecrite par le redacteur%ld is %d\n",f,data);
  sleep(1);
  sem_post(&writeblock);
  pthread_exit(0);
}

int main()
{
  long i;
  pthread_t rtid[5],wtid[5];
  sem_init(&mutex,0,1);
  sem_init(&writeblock,0,1);
  for(i=0;i<=2;i++)
  {
    pthread_create(&wtid[i],NULL,writer,(void *)i);
    pthread_create(&rtid[i],NULL,reader,(void *)i);
  }
  for(i=0;i<=2;i++)
  {
```

```
55      pthread_join(wtid[i],NULL);
56      pthread_join(rtid[i],NULL);
57    }
58    return 0;
59  }
```

5.5 Exercices

1. Écrire un programme multi-thread qui compte le nombre d'occurrences d'un élément dans un tableau d'entiers de grande taille. Le tableau de taille `vectSize` est divisé en blocs de tailles égales, chacun des `nbThreads` threads lancés compte le nombre d'occurrences de l'élément dans le bloc qui lui est affecté et l'ajoute à la variable partagée `nbOccurrences` initialisée à `0`.

2. La fonction suivante calcule séquentiellement le produit scalaire du vecteur `u` par le vecteur `v` de taille `n` et stocke le résultat dans la variable globale `result` initialisée à `0`.

   `(prod-scal-vects.c)`

   ```
   1  double result = 0.0;
   2
   3  void produit_scalaire_seq(double v[], double u[], int n) {
   4    for (int i = 0; i < n; i++) {
   5      result += v[i] * u[i];
   6    }
   7  }
   8
   ```

   Écrire un programme multi-thread qui calcule le produit scalaire de manière parallèle. Le programme démarre `nbThreads` threads, chacun des `nbThreads` threads calcule un bloc du produit scalaire et stocke le résultat partiel dans une variable partagée `result` initialisée à 0. Le résultat final est affiché par le thread `main`.

3. Dans l'exemple 5.4.1, l'écriture de tous les éléments de `tab` est suivie de la lecture de tous les éléments de `tab`. Utiliser les sémaphores pour avoir l'écriture de l'élément `tab[i]` suivie immédiatement de la lecture de l'élément `tab[i]` pour `i` de `0` à `n-1`.

4. Exécuter plusieurs fois le programme suivant (`un-deux-fin.c`) :

   ```
   1 #include <stdlib.h>
   2 #include <stdio.h>
   3 #include <unistd.h>
   ```

```c
#include <string.h>
#include <pthread.h>

void mes1(void){
  sleep(rand()%5);
  printf("Un, ");
  pthread_exit(0);
}

void mes2(void){
  sleep(rand()%5);
  printf("Deux, ");
  pthread_exit(0);
}

int main(){
  pthread_t th1, th2;
  srand(time(NULL));

  if(pthread_create(&th1, NULL, (void*) mes1, NULL)){
    fprintf(stderr,"thread1");
    exit(EXIT_FAILURE);
  }

  if(pthread_create(&th2, NULL, (void*) mes2, NULL)){
    fprintf(stderr,"thread2");
    exit(EXIT_FAILURE);
  }
  pthread_join(th1,NULL);
  pthread_join(th2,NULL);
  printf("Partez!\n");
  exit(EXIT_SUCCESS);
}
```

(a) Quelles sont toutes les sorties possibles du programme?

(b) Utiliser les sémaphores de façon à obtenir à chaque exécution la sortie `Un, Deux, Partez!`

5. Écrire un programme qui organise un rendez-vous de `N` threads.

6. Exécuter plusieurs fois le programme `C` suivant (`one-two-three-viva-Algerie.c`):

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>
#include <pthread.h>

```

```
8  void func1(void){
9    for (int i = 0; i < 4; i++){
10     sleep(rand()%5);
11     printf("One, ");
12   }
13   pthread_exit(0);
14 }
15
16 void func2(void){
17   for (int i = 0; i < 4; i++){
18     sleep(rand()%5);
19     printf("Two, ");
20   }
21   pthread_exit(0);
22 }
23
24 void func3(void){
25   for (int i = 0; i < 4; i++){
26     sleep(rand()%5);
27     printf("Three, ");
28   }
29   pthread_exit(0);
30 }
31
32 void func4(void){
33   for (int i = 0; i < 4; i++){
34     sleep(rand()%5);
35     printf("Viva l'Algerie.\n");
36   }
37   pthread_exit(0);
38 }
39
40 int main(){
41   srand(time(NULL));
42   pthread_t th1, th2, th3, th4;
43
44   if(pthread_create(&th1, NULL, (void*) func1, NULL)){
45     fprintf(stderr,"thread1");
46     exit(EXIT_FAILURE);
47   }
48
49   if(pthread_create(&th2, NULL, (void*) func2, NULL)){
50     fprintf(stderr,"thread2");
51     exit(EXIT_FAILURE);
52   }
53
54   if(pthread_create(&th3, NULL, (void*) func3, NULL)){
55     fprintf(stderr,"thread3");
56     exit(EXIT_FAILURE);
57   }
58
59   if(pthread_create(&th4, NULL, (void*) func4, NULL)){
60     fprintf(stderr,"thread4");
61     exit(EXIT_FAILURE);
```

```
62    }
63    pthread_join(th1,NULL);
64    pthread_join(th2,NULL);
65    pthread_join(th3,NULL);
66    pthread_join(th4,NULL);
67    pthread_exit(0);
68  }
```

Utiliser les sémaphores pour que nous ayons toujours la sortie suivante : One, Two, Three, Viva l'Algérie.

7. Soient 3 threads `T1`, `T2` et `T3` qui exécutent respectivement les fonctions `affiche1()`, `affiche2()` et `affiche3()` données ci-dessous (`affiche-1-2-3.c`):

```
1  void *affiche1() {
2    for(int i = 0; i < 10; i++) {
3      printf("1");
4    }
5    pthread_exit(0);
6  }
7
8  void *affiche2() {
9    for(int i = 0; i < 10; i++) {
10     printf("2");
11   }
12   pthread_exit(0);
13 }
14
15 void *affiche3() {
16   for(int i = 0; i < 10; i++) {
17     printf("3");
18   }
19   pthread_exit(0);
20 }
```

Utiliser les sémaphores pour réaliser les synchronisations suivantes:

(a) Obtenir l'affichage suivant : 123 123 123 ...

(b) Obtenir l'affichage suivant : 1 suivi de 2 et 3 dans n'importe quel ordre (exemple: 132 132 123 132 ...).

(c) Obtenir l'affichage suivant : 1 suivi de 2 ou 3 exclusivement (exemple:12 12 13 12 13 ...).

6 Les variables de condition

6.1 Introduction

Les variables de condition fournissent un autre type de synchronisation entre threads d'un processus.

La synchronisation par variables de condition permet de suspendre l'exécution d'un thread si une certaine condition n'est pas vérifiée (par exemple, on souhaite suspendre l'exécution du thread père jusqu'à ce que son fils termine son exécution). Dans ce cas, le thread suspendu attend dans la file d'attente associée à la variable condition et libère le processeur. Dès que la condition est vérifiée, le thread suspendu sera activé par un autre thread par envoie d'un signal.

Dans la norme `POSIX`, les variables de condition sont des variables de type `pthread_cond_t`. Avant son utilisation dans un programme, une variable de condition doit être initialisée. Il existe deux manières de le faire:

6.2 Initialisation d'une variable de condition

1. La fonction `pthread_cond_init()`

```
#include <pthread.h>

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

   `pthread_cond_init()` initialise la variable de condition `cond`, en utilisant les attributs spécifiés par `cond_attr`, ou les attributs par défaut si `cond_attr` vaut `NULL`.

2. Utilisation de la constante `PTHREAD_COND_INITIALIZER`

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

6.3 Opérations sur une variable de condition

Deux opérations sont associées à une variable de condition: `pthread_cond_wait()` et `pthread_cond_signal()`.
`pthread_cond_wait()` est exécutée par un thread quand il doit suspendre son exécution et se mettre dans la file d'attente associée à la variable de condition.

```c
#include <pthread.h>

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

L'appel de `pthread_cond_wait()` déverrouille atomiquement `mutex` (qui doit auparavant avoir été verrouillé), ajoute le thread appelant à la file d'attente de la variable condition `cond` et attend que la variable condition `cond` soit signalée. L'exécution du thread est suspendue et ne consomme pas de temps CPU jusqu'à ce que la variable de condition soit signalée. Lorsque le thread se réveille et que l'appel retourne, le thread verrouille le mutex à nouveau.

Une variable condition est associée à un mutex pour éviter la situation de compétition (race condition) créée par un thread qui se prépare à attendre et un autre qui doit signaler la condition avant que le premier thread attende effectivement sur cette condition ce qui peut conduire à une situation d'interblocage (deadlock). Le thread sera en attente perpétuelle d'un signal qui n'arrivera peut être jamais.

```c
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

La fonction `pthread_cond_signal()` réveille un des threads en attente sur la condition `cond`. Si aucun thread n'est en attente `cond`, l'appel n'aura aucun effet et le signal ne servira pas à réveiller de futurs threads qui seront en attente sur `cond`.

```c
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

`pthread_cond_broadcast()` réveille tous les threads en attente sur la condition `cond`. Là aussi, si aucun thread n'est en attente sur `cond`, rien ne vas se passer et le signal ne servira pas à réveiller de futurs threads qui seront en attente sur cette condition.

```c
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

Avant de terminer un programme, il ne faut pas oublier de détruire toutes les variables condition utilisées dans le programme et libérer les ressources utilisées par celles-ci. Ceci est réalisé par l'appel de la fonction `pthread_cond_destroy()`. Avant de détruire une variable condition, il faut s'assurer qu'aucun thread n'est en attente sur cette condition.

La forme typique d'utilisation des variables de condition est:

```
// Examiner la condition en toute securite et empecher les autres
// threads de la modifier
pthread_mutex_lock (&mutex);
while ( une certaine condition est fausse )
  pthread_cond_wait (&cond, &mutex);

// Faire ce qu'il y a lieu de faire quand la condition devient vraie
faire_quelquechose();
pthread_mutex_unlock (&mutex);
```

De l'autre coté, un thread qui signale une variable de condition, ressemble en général à ceci:

```
// S'assurer d'abord qu'on a un acc s exclusif a tout ce qui comprend la
    variable de condition
pthread_mutex_lock (&mutex);

alterer_condition();

// Reveiller au moins un thread (s'il y en a) qui attend sur la condition
pthread_cond_signal (&cond);

// Permettre aux autres threads de proceder
pthread_mutex_unlock (&mutex)
```

6.4 Exemple (`varcond1.c`)

Dans le programme suivant, le thread principal crée trois threads, `threads[0]`, `threads[1]` et `threads[2]`. `threads[0]` et `threads[1]` incrémentent la variable partagée `"count"`, `threads[2]` surveille la valeur de `"count"`. Lorsque `"count"` atteint `COUNT_LIMIT`, le thread en attente est signalé par l'un des deux threads qui incrémente. Le thread en attente se réveille et modifie la valeur de `count`. Le programme continue jusqu'à ce que les threads qui incrémentent atteignent `TCOUNT`. Le programme principal joint les 3 threads et affiche la valeur finale de `count`.

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

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;

pthread_mutex_t count_mutex;
pthread_cond_t count_seuil_cond;

void *inc_count(void *t) {
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    // Tester la valeur de count and signaler le thread en attente si la condition est remplie.
    // Noter que ceci se produit quand mutex est verrouille.

    if (count == COUNT_LIMIT) {
      printf("thread %ld (inc_count()): count = %d  Seuil atteint. ", my_id, count);
      pthread_cond_signal(&count_seuil_cond);
      printf("Signal envoye.\n");
      }
    printf("thread %ld (inc_count()): count = %d, Deverouillage de mutex\n", my_id, count);
    pthread_mutex_unlock(&count_mutex);

    // Simuler l'execution d'une certaine tache
    sleep(rand()%4);
    }
  pthread_exit(NULL);
}

void *surveille_count(void *t) {
  long my_id = (long)t;

  printf("thread %ld (surveille_count()): Debut surveillance de count: \n", my_id);

  // Verouillage de mutex et attente si la condition est non remplie (count < COUNT_LIMIT.
  // Noter que pthread_cond_wait() va automatiquement et atomiquement deverrouiller mutex pendant
  // qu'il attend. Noter egalement que si COUNT_LIMIT est atteinte avant l'appel de
  // pthread_cond_wait(), la boucle est ignoree pour empecher
```

```
    pthread_cond_wait() de ne jamais retourner.
49
50    pthread_mutex_lock(&count_mutex);
51    while (count < COUNT_LIMIT) {
52      printf("thread %ld (surveille_count()): Count= %d. Suspension de l'
      execution et attente ...\n", my_id,count);
53      pthread_cond_wait(&count_seuil_cond, &count_mutex);
54      printf("thread %ld (surveille_count()): Signal recu. Count= %d\n",
      my_id,count);
55      printf("thread %ld (surveille_count()): Maj de la valeur de count...\
      n", my_id);
56      count += 125;
57      printf("thread %ld (surveille_count()): maintenant, count  = %d.\n",
      my_id, count);
58      }
59    printf("thread %ld (surveille_count()): Deverouilage de mutex.\n",
      my_id);
60    pthread_mutex_unlock(&count_mutex);
61    pthread_exit(NULL);
62  }
63
64  int main(int argc, char *argv[]) {
65    int i;
66    long t1=1, t2=2, t3=3;
67    pthread_t threads[3];
68    pthread_attr_t attr;
69    srand(time(NULL));
70
71    // Initialisation des objets mutex et la variable condition
72    pthread_mutex_init(&count_mutex, NULL);
73    pthread_cond_init (&count_seuil_cond, NULL);
74
75    // Creation des threads dans un etat joignable
76    pthread_attr_init(&attr);
77    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
78    pthread_create(&threads[0], NULL, surveille_count, (void *)t1);
79    pthread_create(&threads[1], NULL, inc_count, (void *)t2);
80    pthread_create(&threads[2], NULL, inc_count, (void *)t3);
81
82    // Attendre que tous les threads terminent.
83    for (i = 0; i < NUM_THREADS; i++) {
84      pthread_join(threads[i], NULL);
85    }
86    printf ("Main(): Jointure des %d threads. Valeur finale de count = %d.
      Termine.\n",
87            NUM_THREADS, count);
88
89    // Nettoyer et sortir
90    pthread_attr_destroy(&attr);
91    pthread_mutex_destroy(&count_mutex);
92    pthread_cond_destroy(&count_seuil_cond);
93    pthread_exit (NULL);
94  }
```

6.5 Exercices

1. Rdv de threads
2. Lecteurs/rédacteurs
3. Producteurs/consommateurs

7 Références bibliographiques

References

[ubu,] http://manpages.ubuntu.com/manpages.

[tut,] http://www.tutorialspoint.com/operating_system/os_multi_threading.htm.

[Barney, 2015] Barney, B. (2015). Posix threads programming.

[Blaess, 2005] Blaess, C. (2005). *Programmation système en C sous Linux. Signaux, processus, threads, IPC et sockets*. Eyrolles.

[Silberschatz et al., 2004] Silberschatz, A., Galvin, P. B., and Gagne, G. (2004). *Operating System Concepts*. John Wiley & Sons.