TRAVAUX PRATIQUES 1
Commandes et primitives liées aux processus sous Linux

L'objectif de ce TP est de voir les principales commandes et primitives fournies par les systèmes d'exploitation de type UNIX de sorte à interagir avec un processus.
Nous rappelons qu'un processus est la représentation de la dynamique d'exécution d'un programme (ou partie d'un programme si celui-ci créé durant son exécution plusieurs autres processus).
Toutes les commandes utilisées dans ce TP sont à taper dans une fenêtre shell qui les lance et les exécute.
Une fois votre session ouverte sous un système d'exploitation de type UNIX, ouvrez un terminal pour entrer les commandes décrites dans l'énoncé. Dans la suite, le prompt du terminal sera symbolisé par « `$>` ».

Nous rappelons également que la commande **`man`** (pour *manuel*) fournit un descriptif détaillé sur l'utilisation d'une commande système passée en paramètre, et que pour sortir du manuel il faut appuyer sur la touche **q** du clavier. Il ne faut donc pas hésiter à consulter cette documentation qui est un véritable aide-mémoire.

Dans ce TP vous aurez à écrire des programmes en langage C pour utiliser les primitives systèmes. Pour cela, vous pouvez utiliser un éditeur de texte comme `kwrite` en exécutant la commande ci-dessous dans le répertoire contenant `prog.c` : `$>kwrite prog.c &`

Rappel : le symbole `&` placé en fin de commande permet de lancer l'exécution de la commande en tâche de fond, ainsi le terminal vous redonne la main pour entrer une nouvelle commande.

EXERCICE 1

Dans cet exercice nous revenons sur la visualisation d'informations liées aux processus en cours d'exécution.
Nous rappelons que le système maintient une arborescence des processus présents dans le système et chaque processus est caractérisé par plusieurs informations listées ci-dessous :

1) **utilisateur (UID) :** nom ou identifiant de l'utilisateur ayant lancé le programme,
2) **pid (Processus Identifier) :** correspond à l'entier attribué par le système à un processus pour l'identifier,
3) **ppid (Parent Processus Identifier) :** correspond à l'identifiant du processus parent ayant engendré le processus en question,
4) **état (S, STA ou STATE) :** définit par l'ordonnanceur du système, plusieurs valeurs sont possibles :
    1. S pour *Stopped* si le processus est en sommeil,
    2. R pour *Running* si le processus est en exécution ou prêt à s'exécuter,
    3. T arrêté temporairement à la demande d'un signal comme `CTRL+Z`,

4. Z processus dans l'état zombie,
5. +, l, < indications additionnels avec des états ci-dessus (associé aux processus en premier plan, possède des processus légers, haute priorité).

- **utilisation processeur (c ou %CPU) :** indique le pourcentage utilisation du processeur par rapport à la durée de vie du processus,
- **terminal (TTY) :** terminal auquel est rattaché le processus (pour affichages et contrôle …),
- **durée d'exécution (TIME) :** temps cumulé passé à exécuter le processus,
- **commande (CMD) :** correspond au nom du programme exécutable.
- **priorité d'exécution statique (PRI) :** priorité statique allant égale 0 pour les applications non temps réelle et allant de 1 à 99 pour les applications temps réelle (une valeur important implique une priorité plus forte),
- **priorité d'exécution dynamique (NI) :** priorité dynamique allant de -20 à 19 (une faible valeur implique une plus forte priorité) la priorité dynamique est utilisée entre les processus avec une priorité statique égale,
- **ordonnancement (CLS) :** classe d'ordonnancement utilisé pour le processus (toutes les classes sont préemptives), plusieurs valeurs sont possibles :

1. - ou ? : non signalé,
2. TS (SCHED_OTHER) : ordonnancement classique en temps partagé (politique tourniquet appliquée pour tous les processus de cette classe) utilisé par la plupart des processus d'application non temps réel (priorité statique égale à 0),
3. B (SCHED_BATCH) : politique similaire à SCHED_OTHER utilisée pour les applications non-interactive car de priorité toujours plus faible que SCHED_OTHER,
4. FF (SCHED_FIFO) : ordonnancement pour les applications temps réelle (priorité statique > 0) à base de tranches de temps, exécution du processus suivant la politique FIFO (premier arrivé premier servi) jusqu'à ce qu'il soit bloqué par une opération d'entrée/sortie ou préempté par un processus de priorité supérieure,
5. RR (SCHED_RR) : politique d'exécution tourniquet dans chaque file d'attente de même priorité où chaque processus s'exécute durant un quantum de temps et est ensuite placé à la fin de la liste de sa priorité (s'il a été préempté par un processus de priorité supérieure alors il terminera sa tranche de temps lorsqu'il reprendra son exécution).

L'ordonnanceur est la partie du noyau qui décide quel processus prêt va être exécuté. L'ordonnanceur de Linux propose trois grandes politiques différentes (une pour les processus classiques et deux pour les applications à vocation temps-réel).
Une valeur de priorité statique est assignée à chaque processus. L'ordonnanceur dispose d'une liste de tous les processus prêts pour chaque valeur possible de priorité statique (allant de 0 à 99). Afin de déterminer quel processus doit s'exécuter, l'ordonnanceur de Linux recherche la liste non-vide de plus haute priorité statique et prend le processus en tête de cette liste. La politique d'ordonnancement détermine pour chaque processus l'emplacement où il sera inséré dans la liste contenant les processus de même priorité statique, et comment il se déplacera dans cette liste.

La commande `ps` (pour *Processus*) permet d'obtenir les informations décrites ci-dessus pour l'ensemble des processus en cours d'exécution.

Par défaut, cette commande n'affiche que les processus associés aux applications lancées explicitement par un utilisateur.
L'option `-u utilisateur` permet d'avoir accès à la liste de tous les processus associés à l'utilisateur `utilisateur` : `$> ps -u utilisateur`
L'option `-l` permet d'afficher plus d'informations sur les processus listés par exemple pour un utilisateur donné : `$> ps -lu utilisateur`

Ci-dessous est donné un exemple d'informations obtenues à l'aide de ces paramètres :

```
F   S UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY    TIME    CMD
100 S  0   467   1    0  60   0 - 256 read_c tty5   00:00:00 mingetty
100 S  0   468   1    0  60   0 - 256 read_c tty6   00:00:00 mingetty
100 S  0   576   570  0  60   0 - 498 read_c pts/0  00:00:00 cat
100 S  0   580   578  0  70   0 - 576 wait4  pts/1  00:00:00 bash
100 S  0   581   579  0  60   0 - 77  wait4  pts/2  00:00:00 bash
000 S  0   592   581  2  61   0 - 253 down_f pts/2  00:00:01 essai
040 S  0   593   592  2  61   0 - 253 write_ pts/2  00:00:01 essai
100 R  0   599   580  0  73   0 - 652 -      pts/1  00:00:00 ps
```

Les champs `S, PID, PPID` et `CMD` codent respectivement l'état du processus, la valeur du PID et du PPID pour le processus et le nom du programme exécuté.

Plus généralement, la commande suivante permet d'afficher toutes les informations associées à tous les processus en cours d'exécution dans le système : `$> ps aux`
Afin de répondre aux questions ci-dessous, ouvrez un second terminal pour vous permettre d'afficher en parallèle de l'exécution des programmes demandés les informations fournies par la commande `ps`.

Soit le programme `exo1.c` suivant :

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

int main(void)
{
  while(1)
  {
    printf("Je suis vivant et vais me mettre en sommeil pour 2s.\n");
    sleep(2);
  }
  exit(0);
}
```

Dans le programme ci-dessus, la fonction `sleep(s)` permet de demander au système de changer l'état du processus et de le mettre en sommeil pendant `s` secondes. De plus, la fonction `exit(c)` permet de demander au système d'arrêter l'exécution du processus et d'envoyer un code de retour `c` au processus parent (nous verrons en détail cette fonction un peu plus loin).

Question 1 – Tapez, compilez et exécutez le programme `exo1.c` ci-dessus. Indiquez à l'aide de la commande `ps` les informations pour le processus associé au programme exécuté (PID, PPID, état, durée d'exécution TIME). On rappel que la commande à utiliser pour compiler un programme : `$> gcc -o exo1 exo1.c`

Question 2 – Exécutez la commande `ps -ejH` ou `ps axjf` afin d'afficher l'arborescence des processus. Indiquez le processus parent du processus lié au programme `exo1.c`.

Question 3 – Par défaut la commande `ps` n'affiche pas la classe de politique d'ordonnancement utilisée pour chaque processus, il faut indiquer les informations désirées à afficher avec l'option `-o` comme suit :

```
$> ps -u utilisateur -o pid,ppid,class,ni,pri,pcpu,stat,comm
```

Utilisez la commande `ps` avec les options ci-dessus et indiquez quelle politique d'ordonnancement est utilisée pour exécuter les processus utilisateurs.

Nous rappelons que la commande `top` permet d'obtenir des informations similaires à la commande `ps`, mais contrairement à cette dernière ces informations sont mises à jour en temps réel.

Question 4 – Réitérez la procédure décrite en Question 1 mais en utilisant la commande `top`. Vous pouvez faire défiler l'affichage en utilisant les touches `<PageUp>` et `<PageDown>` (pour sortir de top vous pouvez utiliser la touche `q` ou `<CTRL>+c`).

Question 5 – A l'aide de la commande `top`, indiquez le nombre de processus en cours d'exécution, ainsi que ceux dans chacun des états : *running, sleeping, stopped* et *zombie*.
Indiquez également le pourcentage d'utilisation processeur et mémoire physique.

Le système donne la possibilité à un utilisateur de demander l'arrêt de l'exécution d'un processus qu'il a lancé. Pour cela il faut utiliser la commande `kill` en indiquant le PID du processus dont on désire demander l'arrêt au système.
L'exemple ci-dessous demande l'arrêt du processus de PID 25446 :

```
$> kill -9 25446
```

Question 6 – Lancez le programme `exo1` dans un premier terminal et utilisez la commande `kill` pour demander la terminaison du processus fils. Vous utiliserez la commande `ps` avant et après la commande `kill` dans le même terminal afin de visualiser les processus en exécution.
Est-ce que le processus associé au programme `exo1` s'est bien terminé ?

La commande `kill` a été utilisée à la question précédente afin de demander au système d'arrêter brutalement un processus en cours d'exécution. Plus généralement, la commande `kill` peut être utilisée afin d'envoyer d'autres demandes/évènements à un processus. Les systèmes de type Unix utilise le concept de signaux afin d'informer d'éléments particulier un processus. Un ensemble de 64 signaux sont définis, chacun ayant une signification et étant

codifié par un nombre entre 1 et 64 (ou un nom/macro pour une utilisation plus aisée). Les 32 premiers signaux représentent les signaux standards utilisés dans tous les systèmes Unix, et les 32 signaux suivants font référence à des événements temps réel et utilisés dans cas spécifiques.
La codification peut différer d'une version à l'autre du système. Pour afficher la liste des signaux définit sous le système utilisé vous pouvez taper la commande suivante :

```
$> kill -l
```

Cela doit vous donner un résultat proche de l'affichage ci-dessous :

```
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
9) SIGKILL      10)SIGUSR1      11)SIGSEGV      12)SIGUSR2
13)SIGPIPE      14)SIGALRM      15)SIGTERM      16)SIGSTKFLT
17)SIGCHLD      18)SIGCONT      19)SIGSTOP      20)SIGTSTP
21)SIGTTIN      22)SIGTTOU      23)SIGURG       24)SIGXCPU
25)SIGXFSZ      26)SIGVTALRM    27)SIGPROF      28)SIGWINCH
29)SIGIO        30)SIGPWR       31)SIGSYS       34)SIGRTMIN
35)SIGRTMIN+1   36)SIGRTMIN+2   37)SIGRTMIN+3   38)SIGRTMIN+4
39)SIGRTMIN+5   40)SIGRTMIN+6   41)SIGRTMIN+7   42)SIGRTMIN+8
43)SIGRTMIN+9   44)SIGRTMIN+10  45)SIGRTMIN+11  46)SIGRTMIN+12
47)SIGRTMIN+13  48)SIGRTMIN+14  49)SIGRTMIN+15  50)SIGRTMAX-14
51)SIGRTMAX-13  52)SIGRTMAX-12  53)SIGRTMAX-11  54)SIGRTMAX-10
55)SIGRTMAX-9   56)SIGRTMAX-8   57)SIGRTMAX-7   58)SIGRTMAX-6
59)SIGRTMAX-5   60)SIGRTMAX-4   61)SIGRTMAX-3   62)SIGRTMAX-2
63)SIGRTMAX-1   64)SIGRTMAX
```

Parmi ces 64 signaux, seuls les 32 premiers signaux (standards) sont utilisés couramment. Voici ci-dessous une description de certains de ceux-ci.

Numéro	Nom	Description
1	SIGHUP	Instruction (HANG UP) – Fin de session
2	SIGINT	Interruption
3	SIGQUIT	Instruction (QUIT)
4	SIGILL	Instruction illégale
5	SIGTRAP	Trace trap
6	SIGABRT (ANSI)	Instruction (ABORT)
6	SIGIOT (BSD)	IOT Trap
7	SIGBUS	Bus error
8	SIGFPE	Floating-point exception - Exception arithmétique
9	SIGKILL	Instruction (KILL) - termine le processus immédiatement
10	SIGUSR1	Signal utilisateur 1
11	SIGSEGV	Violation de mémoire
12	SIGUSR2	Signal utilisateur 2

Numéro	Nom	Description
13	SIGPIPE	Broken PIPE - Erreur PIPE sans lecteur
14	SIGALRM	Alarme horloge
15	SIGTERM	Signal de terminaison
16	SIGSTKFLT	Stack Fault
17	SIGCHLD ou SIGCLD	modification du statut d'un processus fils
18	SIGCONT	Demande de reprise du processus
19	SIGSTOP	Demande de suspension imbloquable
20	SIGTSTP	Demande de suspension depuis le clavier
21	SIGTTIN	lecture terminal en arrière-plan
22	SIGTTOU	écriture terminal en arrière-plan
23	SIGURG	évènement urgent sur socket
24	SIGXCPU	temps maximum CPU écoulé
25	SIGXFSZ	taille maximale de fichier atteinte
26	SIGVTALRM	alarme horloge virtuelle
27	SIGPROF	Profiling alarm clock
28	SIGWINCH	changement de taille de fenêtre
29	SIGPOLL (System V)	occurence d'un évènement attendu
29	SIGIO (BSD)	I/O possible actuellement
30	SIGPWR	Power failure restart
31	SIGSYS	Erreur d'appel système

Chaque processus possède un tableau associant un booléen pour chaque signal. Ce tableau est stocké dans le bloc de contrôle du processus. Le système positionne à 1 le booléen associé à un signal particulier afin d'avertir un processus de la réception du signal. Un second tableau est présent dans le bloc de contrôle afin d'indiquer quel comportement il faut adopter pour traiter le signal reçu. Lors de la réception d'un ou plusieurs signaux, le système les traitent par ordre croissant de leurs numéros.

Question 7 – Le signal `9` envoyé avec la commande `kill -9` permet de terminer brutalement (immédiatement) un processus.
A l'aide du tableau ci-dessus donnez la commande à taper afin de demander au système la terminaison d'un processus.

Nous allons nous intéresser à la prise en compte d'un signal reçu dans un programme en langage C, de sorte à faire un traitement. Si aucun traitement n'a été mis en place pour un signal, celui-ci pourra être ignoré ou conduire à la terminaison du processus suivant les version du système Linux.
Pour indiquer au système la fonction de traitement à exécuter lors de la réception d'un signal particulier par un processus, nous allons utiliser la fonction ci-dessous :

```
signal(int signum, sighandler_t handler);
```

Cette fonction prend en premier paramètre le numéro ou le nom du signal, et en second paramètre l'adresse vers la fonction de traitement à exécuter (attention : la fonction de traitement ne peut récupérer un paramètre).

Il est possible d'indiquer au système que le comportement à suivre pour le traitement d'un signal est de l'ignorer. Cela est réalisé en utilisant la macro `SIG_IGN` pour le second paramètre de la fonction `signal()`, alors que la macro `SIG_DFL` indique au système d'adopter le comportement par défaut (conduisant éventuellement à la terminaison du processus pour certains signaux).

Soit le programme `exo1bis.c` ci-dessous réalisant des écritures dans un fichier et pour lequel une fonction de traitement a été mise en place pour le signal `SIGINT` :

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

int fich;

void fonction_handler(void)
{
   printf("signal SIGINT recu\n");
   close(fich);

   exit(0);
}

int main(int argc, char **argv)
{
  int nb=0;

  // Association fonction de traitement au signal SIGINT
  signal(SIGINT, (__sighandler_t) fonction_handler);

  /*  ouverture d'un fichier */
  if((fich=open("attente.txt", O_CREAT|O_WRONLY, 0))==-1)
  {
    perror("fichier");
    exit(1);
  }

  while(1)
  {
     printf("En attente d'un signal.\n");
     sleep(2);
     nb=nb+2;
     write(fich, &nb, sizeof(nb));
  }

  exit(0);
}
```

Question 8 – Compilez et exécutez le programme ci-dessus. Une fois l'exécution lancée attendez un cours moment et appuyez sur la combinaison de touche `<CTRL>+c`.
Que constatez-vous ? Pourquoi ?

Les systèmes Unix donnent la possibilité de changer temporairement la fonction de traitement à appeler pour traiter un signal. Pour cela il faut utiliser la fonction suivante :

```
int sigaction(int sig, const struct sigaction *p_action,  const struct sigaction *p_action_old);
```

Cette fonction prend en premier paramètre le numéro ou nom du signal et deux structures de type `sigaction` (décrite ci-dessous) permettant d'indiquer respectivement le nouveau comportement à adopter et de sauvegarder l'ancien comportement (pour pouvoir le replacer par la suite).
La structure `sigaction` est définie comme suit :

```
struct sigaction {      void ($sa_handler)() ;
                        sigset_t sa_mask ;
                        int sa_flags;}
```

Le premier champ stocke l'adresse de la fonction de traitement à exécuter, le second permet de mettre en placer un masque sur les signaux durant l'exécution de la fonction de traitement, et le dernier champ définit des options particulières pour l'exécution de la fonction de traitement. Masquer certains signaux (deuxième) permet par exemple d'éviter qu'un programme ne soit interrompu durant son exécution, sinon le résultat pourrait être incorrect (exécution atomique d'une suite d'instructions du programme).

Soit le programme `exo1ter.c` ci-dessous reprenant le programme précédent et dans lequel la fonction de traitement du signal `SIGINT` est modifié à chaque réception du signal :

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

struct sigaction action, action_old;
int fich;


void fonction_handler1(void)
{
   printf("signal SIGINT recu (fermeture du fichier)\n");

   close(fich);

   // Mise en place de la nouvelle fonction
   action.sa_handler= fonction_handler2;
   sigaction(SIGINT, &action, &action_old);
}

void fonction_handler2(void)
{
   printf("signal SIGINT recu (ouverture du fichier)\n");

   /*  ouverture d'un fichier */
   if((fich=open("attente.txt", O_CREAT|O_WRONLY, 0))==-1)
   {
      perror("fichier");
      exit(1);
   }
```

```
   // Mise en place de la fonction initiale
   sigaction(SIGINT, &action_old, &action);
}

int main(int argc, char **argv)
{
  int nb=0;

  // Association fonction de traitement au signal SIGINT
  action.sa_handler= fonction_handler1;
  sigaction(SIGINT, &action, NULL);

  /*  ouverture d'un fichier */
  if((fich=open("attente.txt", O_CREAT|O_WRONLY, 0))==-1)
  {
    perror("fichier");
    exit(1);
  }

  exit(0);
}
```

Question 9 – Compilez et exécutez le programme ci-dessus. Vérifiez qu'à chaque réception du signal `SIGINT` (appuie sur la combinaison de touche `<CTRL>+c)` le comportement est bien modifié (les fonctions `fonction_handler1()` ou `fonction_handler2()` sont appelées successivement).

EXERCICE 2

Soit le programme `exo2.c` donné ci-dessous. Ce programme créé un processus fils, de type *processus lourd*, c'est-à-dire que le nouveau processus créé est une copie parfaite du processus parent (segments de code et de données identique, mais les segments de données du processus père et fils se trouvent dans espaces mémoire distincts).
La création d'un tel type de processus est réalisée à l'aide la primitive `fork()` suivante :

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

Les fonctions getpid() et getppid() permettent à un processus d'obtenir respectivement son identifiant et celui de son processus parent :

```
pid_t getpid(void);

pid_t getppid(void);
```

Soit le programme `exo2.c` ci-dessous qui créé deux processus (parent et fils) :

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

int main(void)
{
     int pid, i;

     pid = fork();
```

```
      if(pid == 0)
      {
         for(i=0;i<10;i++)
         {printf("je suis le fils d'identifiant %d et le processus %d est
      mon père\n", getpid(), getppid());}
         exit(0);
      }
      else
      {
         while(1)
         {printf("je suis le père d'identifiant %d et le processus %d est
      mon fils\n", getpid(), pid);}
      }
      exit(0);
 }
```

Question 1 – Tapez le fichier `exo2.c` et compilez le programme `exo2.c` à l'aide de la commande suivante : `$> gcc –o exo2 exo2.c`
Exécutez le programme `exo2.c` depuis votre shell et indiquez l'évolution :

1. Quelle commande utilisez-vous pour obtenir le nombre de processus lié à `exo2` ?
2. Quelle commande tapez-vous pour détruire le fils `exo2` ? Quel est son état ?
3. Quelle commande tapez-vous pour détruire le père `exo2` ? Que peut-on constater pour le processus fils ?

Comme indiquée précédemment, la primitive `exit(c)` permet de terminer un processus et de renvoyer le code de retour `c` au parent. Ce code de retour permet au processus parent de déterminer si l'exécution de son processus fils s'est réalisée correctement. En général, un code de retour égal à 0 indique que le processus s'est exécuté correctement. Il est possible de mettre en place sa propre codification dans les programmes développé en utilisant des codes positifs afin d'identifier des exécutions incorrectes spécifiques pour les traiter dans le processus parent.

Pour permettre à un processus parent de prendre en compte la terminaison (réception du signal `SIGCHLD`) d'un (quelconque) processus, il est nécessaire d'utiliser la fonction `wait()` ci-dessous :

```
pid_t wait(int *status);
```

Cette fonction est bloquante, c'est-à-dire qu'à l'appel de cette fonction le processus est mis en sommeil (progression de l'exécution gelée) jusqu'à la réception du signal `SIGCHLD`.
Cette fonction place dans le pointeur passé en paramètre le code de retour renvoyé par le fils et retourne l'identifiant du processus fils correspondant.

La fonction `wait()` ne permet pas à un processus parent d'attendre la terminaison d'un fils en particulier, en effet le processus parent reprend l'exécution de son programme lors de la réception du signal `SIGCHLD` issu de la terminaison de n'importe lequel(s) de ses fils.
Dans le cas où l'on désire attendre la terminaison d'un fils en particulier, il faut utiliser la fonction `waitpid()` cei-dessous :

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

Cette fonction prend en premier paramètre l'identifiant du processus fils dont la terminaison doit être prise en compte et place dans le pointeur `status` en deuxièmte paramètre le code de retour renvoyé par le fils et retourne l'identifiant du processus fils correspondant. Le dernier paramètre permet d'utiliser des options particulières.

Question 2 – Le programme `exo2.c` est modifié comme ci-dessous :

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

int main(void)
{
     int pid, i;
     pid = fork();
     if(pid == 0)
     {
           for(i=0;i<10;i++)
           {printf("je suis le fils\n");}
     }
     else
     {
           printf("je suis le père\n");
           wait();
     }
     exit(0);
}
```

1. Modifiez le fichier `exo2.c` comme ci-dessus en `exo2bis.c`.
   On recompile ce programme pour générer un nouvel exécutable appelé `exo2bis` dont on lance l'exécution.
2. Quelle commande tapez-vous pour détruire le fils `exo2bis` ? Quel est son état ?

Question 3 – Modifiez le programme `exo2bis.c` de sorte à ce que le processus parent créé 3 processus fils et attende la terminaison de chacun d'eux dans l'ordre inverse de leur création. Pour cela vous aurez besoin de créer un tableau de 3 entiers afin de stocker l'identifiant de chaque fils créé avec la fonction `fork()`.

Il est possible de demander au système de remplacer tout ou partie du code à exécuter par un processus en utilisant une primitive de recouvrement `exec`. Il en existe plusieurs et elles se distinguent par la façon de récupérer les paramètres à utiliser :

- Sous forme de liste : execl, execlp, execle,
- Sous forme de tableau : execv, execvp, execve.

Par exemple, les arguments peuvent être passés sous forme de liste (**l** pour *list* en anglais), ou la variable d'environnement `PATH` est utilisée pour le chemin d'accès vers le programme exécutable à exécuter (**p** pour *path* en anglais), ou encore par modification de l'environnement (**e** pour *environment*).

Le programme `exo2bis.c` est modifié comme ci-dessous :

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

int main(void)
{
     int pid;
     pid = fork();
     if(pid == 0)
     {
          printf("je suis le fils et vais exécuter la commande ls\n");
          execlp("ls", "ls", "-l", NULL);
     }
     else
     {
          printf("je suis le père\n");
          wait();
     }
     exit(0);
}
```

Question 4 – Modifiez et exécutez le programme. Que permet de réaliser ce programme modifié.

EXERCICE 3

Dans cet exercice, nous allons nous intéresser aux *processus légers* (ou *thread* en anglais). Contrairement à un processus lourd, les processus légers créés par le même processus parent partagent le même espace mémoire que celui-ci, même s'ils peuvent exécuter un segment de code différent.
Ce type de processus permet d'avoir une empreinte mémoire plus faible que l'utilisation de processus lourds (segment de données identique). De plus, comme les threads partagent le même espace mémoire il est plus facile de les faire communiquer.
En revanche, il est primordial de contrôler l'accès à la mémoire sans quoi certaines exécutions peuvent devenir incohérentes, à cause de l'accès concurrent par plusieurs processus à une même partie de la mémoire.

Comme vu en cours, la création d'un thread fils est réalisée en utilisant primitive `pthread_create()` suivante :

```
int pthread create(pthread t *thread, const pthread_attr_t *attr, void *(*routine)(void *), void *arg);
```

Cette fonction retourne en premier paramètre le numéro du thread fils créé et prend respectivement en deuxième, troisième et dernier paramètre les attribus d'exécution particulier, l'adresse de la fonction à exécuter par le thread et un pointeur vers les paramètres de la fonction à exécuter.
Un code de retour de `0` est retourné en cas de succès lors de la création, sinon une erreur s'est produite.

D'autre part, la terminaison d'un thread est réalisée à l'aide la primitive `pthread_exit()` (au lieu de exit pour un processus lourd) :

```
int pthread_exit(void *value);
```

Cette fonction retourne un code de terminaison `value` au le thread principal.

Enfin, pour prendre en compte la terminaison d'un thread la primitive `pthread_join()` est à utiliser en remplacement de la primitive `wait()` :

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

Cette fonction prend deux paramètres, le premier indique le numéro du thread pour lequel on désire prendre en compte la terminaison et le second est un pointeur vers un tableau à deux dimensions permettant d'obtenir le code de retour des thread fils.
Elle retourne `0` en cas de succès, sinon une erreur s'est produite. Comme pour la primitive `wait()`, la primitive `pthread_create()` est bloquante.

Soit le programme `exo3.c` suivant :

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

int i;

int main(void)
{
     int pid;

     i = 0;
     pid = fork();
     if (pid == 0)
     {
           i = i + 10;
           printf ("hello,  fils %d\n", i);
           i = i + 20;
           printf ("hello, fils %d\n", i);
     }
     else
     {
           i = i + 1000;
           printf ("hello, père %d\n", i);
           i = i + 2000;
           printf ("hello, père %d\n", i);
           wait();
     }
     exit(0);
}
```

Question 1 – Tapez le programme `exo3.c` ci-dessus puis compilez et exécutez le programme exécutable obtenu. Quelles traces génère l'exécution de ce programme ?

Question 2 – Peut-on obtenir une trace différente lors d'une nouvelle exécution du programme ? Testez en relançant plusieurs fois ce programme.

Le programme `exo3.c` est modifié comme suit :

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

int i;

void addition()
{
     i = i + 10;
     printf ("hello, thread fils %d\n", i);
     i = i + 20;
     printf ("hello, thread fils %d\n", i);


     pthread_exit(0);

}

int main(void)
{
     pthread_t num_thread;

     i = 0;
     if(pthread_create(&num_thread, NULL, (void *(*)())addition, NULL) ==
     -1)
          perror ("pb pthread_create\n");
     i = i + 1000;
     printf ("hello, thread principal %d\n", i);
     i = i + 2000;
     printf ("hello, thread principal %d\n", i);
     pthread_join(num_thread, NULL);
     exit(0);
}
```

Question 3 – Modifiez le fichier `exo3.c` comme ci-dessus, puis compilez en utilisant l'option `-lpthread` (comme ci-dessous) et exécutez le programme exécutable `exo3bis` obtenu.

```
$> gcc -lpthread -o exo3 exo3.c
```

Quelles traces génère l'exécution de ce programme ?

Question 4 – Peut-on obtenir une trace différente lors d'une nouvelle exécution du programme ? Testez en relançant plusieurs fois ce programme.