Exercices Corrigés

avec rappels de cours

Programmation Système

processus, tubes, signaux, threads

El Mostafa DAOUDI
Prof. Faculté des Sciences, Oujda
daoudie@yahoo.com

Impression Numérique
Oujda

Dépôt Légal : 2020 MO 1152
ISBN : 978-9920-39-371-3

Edition 2020

Ce travail est le fruit de plusieurs années d'enseignement du module "Programmation Système" sous Unix à la Faculté des Sciences d'Oujda.

اطلب التوفيق من الله.

Contenu

L'essentiel des références:

- Programmation système en C sous Linux
  2e édition
  Auteur: Christophe Blaess
  Edition: Eyrolles

- Les systèmes d'exploitations : Unix, Linux et Windows XP avec C et Java
  Auteur: Samia Bouzefrane
  Edition: Dunod

Chapitre 1 : Fonctions sur les processus

1. Introduction

- Processus (process en anglais), est un concept central dans tous systèmes d'exploitation. Plusieurs définitions:
  - C'est un programme en cours d'exécution; c'est-à-dire, un programme à l'état actif.
  - C'est l'image de l'état du processeur et de la mémoire pendant l'exécution du programme. C'est donc l'état de la machine à un instant donné.

- Un processus (processus fils) est créé par un autre processus (processus père). Le processus fils est un clone (copie conforme) du processus créateur (père). Il hérite de son père le code, les données la pile et tous les fichiers ouverts par le père. Mais attention: les variables ne sont pas partagées.

- Tous les processus ont un ancêtre unique et commun. C'est le processus « init ».

- Chaque processus est identifié par un numéro unique, le PID (Processus IDentification).
  - Le PPID (Parent PID) d'un processus est le PID de son père.
  - Le PID du processus « init » est 1.

2. Les primitifs

- La fonction «int getpid()» renvoie le PID du processus qui appelle cette fonction.
- La fonction « int getppid() » renvoie le PPID du processus père (père du processus qui appelle cette fonction).

3. Création d'un processus

Un processus (processus père) peut créer un autre processus (processus fils) grâce à l'appel système de « fork() » (sous UNIX/Linux) qui est déclaré dans <unistd.h>. Elle renvoi un entier.

Syntaxe

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

- « pid_t » est un type identique à un entier. Il est déclaré dans « /sys/types.h ».

En cas de succès, elle renvoie:

- le PID du processus fils dans le processus père.
- 0 dans le processus fils.
- -1 en cas d'échec.

4. Utilisation classique

```
# include <unistd.h>
…
if (fork() != 0) { // Si je suis le processus père
        /*Exécution du code correspondant au processus  père */
 }
 else { // Si je suis le processus fils
        /*Exécution du code correspondant au processus fils */
 }
```

- Après l'appel de la fonction « fork() », le processus père continue l'exécution de son programme.
- Le processus fils exécute le même programme que le processus père et démarre à partir de l'endroit où la fonction « fork() » a été appelée.

5. La primitive « sleep() »

Syntaxe

```
int sleep(int s )
```

- Le processus qui appelle « sleep() », est bloqué pendant « s » secondes.

6. La primitive «exit() »

La primitive «exit() » permet de terminer le processus qui l'appelle. Elle est déclarée dans le fichier « stdlib.h ».

Syntaxe:

```
void exit (int status)
```

- par convention: status=0 indique une terminaison normale sinon indique une erreur.

7. Synchronisation sur terminaison entre père et fils

Les fonctions « wait() » et « waitpid() » permettent une synchronisation sur terminaison entre le père et ses fils. Elles sont déclarées dans <sys/wait.h>.

La fonction « wait() »

L'appel de la fonction « wait(0) » (ou wait(NULL)) bloque le processus qui l'appelle en attente de la terminaison de l'un de ses processus fils.

- Si le processus qui appelle « wait(0) » n'a pas de fils, alors « wait(0) » retourne -1. Sinon, elle retourne le PID du processus fils qui vient de se terminer.
- Pour attendre la terminaison de tous les fils, il faut faire une boucle sur le nombre de fils, ou exécuter l'instruction :

```
while (wait(0)!=-1);          /* ou  while (wait(NULL)!=-1); */
```

La fonction « waitpid() »

L'appel de la fonction « waitpid(pid_t pid, 0,0) » (ou waitpid(pid_t pid,NULL, 0)) permet d'attendre un processus fils particulier ou appartenant à un groupe, où:

- « pid »: désigne le pid du processus fils qu'on attend sa terminaison.
  - Si pid >0: le processus père attend le processus fils identifié par «pid».
  - Si pid=0, le processus appelant attend la terminaison de n'importe quel processus fils appartenant au même groupe que le processus appelant.
  - Si pid =-1: le processus père attend la terminaison de n'importe quel processus fils, comme avec la fonction « wait() ».

En cas de succès, elle retourne le « pid » du processus fils attendu.

Exercice1 :

1. *Donner l'arbre de processus générés par le programme suivant :*

```
main ( ) {
    if (fork())
        if (fork())
            if (fork()==0)
                fork();
    while (wait(0)!=-1);
}
```

2. *Donner l'arbre de processus générés par le programme suivant :*

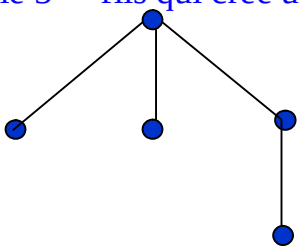
```
main ( ) {
    if (fork()) {
        if (fork()) {
            if (fork()==0)
                fork();
        }
        else
            fork();
    }
    else
        fork();
    while (wait(0)!=-1);
}
```

Réponses:

1. Le processus père crée 3fils et le troisième fils crée à son tour un processus fils.

```
main ( ) {
    if (fork())  // le processus père crée le premier fils
       if (fork())  // le processus père crée le deuxième fils
          if (fork()==0)   // fork() est exécuté par le père qui crée un 3ème fils
             fork();  // exécuté par le 3ème fils qui crée un fils.
    while (wait(0)!=-1);
}
```

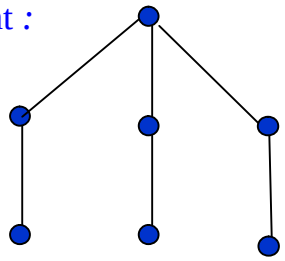
L'arbre généré est le suivant :



2. La première partie du code est identique à la question 1.

- Le « fork() » du premier « else » concerne le deuxième fils qui crée à son tour un processus fils.
- *Le* « fork() » du deuxième « else » concerne le premier fils qui crée à son tour un processus fils.

Donc l'arbre généré est le suivant *:*



Exercice 2 :

1. *Donnez l'arbre de processus généré par le programme suivant :*

```
#include<unistd.h>
main() {
   fork();
   if (fork())
      if (fork());
   sleep(10) ;
}
```

2. *Donner l'arbre de processus généré par le programme suivant :*

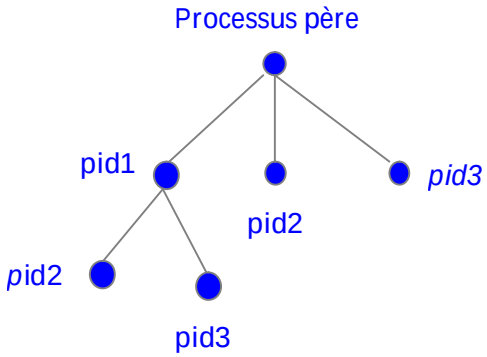
```
#include<unistd.h>
main() {
fork();
if (fork())
   if (fork());
         fork();
sleep(10);
}
```

Réponses :

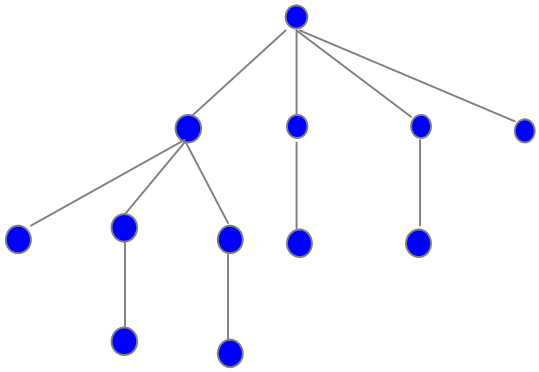
1. Pour bien comprendre, on rajoute dans le programme des identifiants pour les processus créés (appels de fork()). Soient «pid1», «pid2» et «pid3» ces identifiants.

```
…
main() {
  pid_t  pid1, pid2, pid 3;
  pid1=fork();  // le processus père crée un premier fils « pid1 »
  if ((pid2=fork()))  // exécuté par le processus père et par le fils « pid1 ».
                  // donc, chacun d'eux crée un processus fils « pid2 ».
        if ((pid3=fork()));   // exécuté par le père et par « pid1 ».
                  // donc, chacun d'eux crée un processus fils « pid3 ».
  sleep(10) ;
}
```

Donc l'arbre généré est le suivant :



2. Dans cette question on rajoute, à la fin du programme précédent, l'instruction « fork() ». Cette instruction sera exécutée par tous les processus. Donc, chaque processus crée un fils. Par conséquent l'arbre généré est le suivant :

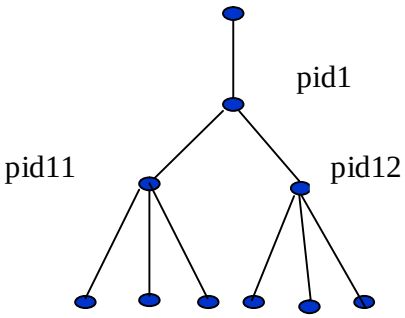


Exercice 3 :

1. *Donner l'arbre de processus générés par le programme C suivant :*

```
# include <sys/wait.h>
# include <unistd.h>
main ( ) {
  if (fork()) {
    if (fork()) {
      if (fork()) {
      }
    }
  }
  while (wait(0)!=-1);
}
```

2. *Ecrire un programme C qui permet de générer l'arbre de processus, représenté par la figure ci-dessous, où le processus père crée le processus fils « pid1 ». Le processus fils « pid1 » crée deux processus fils à savoir « pid11 » et « pid12 ». Ensuite le processus « pid11 » crée 3 processus fils et le processus « pid12 » crée à son tour trois processus fils. Chaque processus doit attendre la terminaison de ses fils avant de se terminer.*

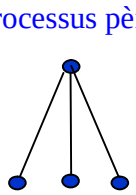


Réponses:

1. Le processus père crée 3 fils. En effet:

```
main ( ) {
  if (fork()) { // exécuté par le processus père qui crée un premier fils
    if (fork()) {  // exécuté par le processus père qui crée un deuxième fils
      if (fork()) { // exécuté par le processus père qui crée un troisième fils
```

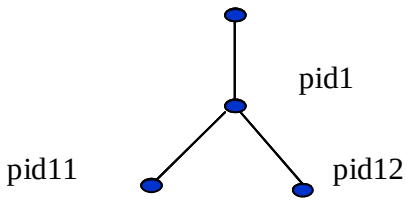
Donc l'arbre généré est le suivant:



2. Le processus père crée, tout d'abord, le processus « pid1 », ensuite, le processus « pid1 » crée les processus « pid11 » et « pid12 » en utilisant les instructions suivantes:

```
pid1=fork(); // le processus père crée le fils « pid1 ».
if (pid1==0) {   // si je suis le processus « pid1 »
   pid11=fork(); // le processus « pid1 » crée le fils « pid11 ».
   if (pid11==0) {  // si je suis le processus « pid11 »
      // code des instructions à exécuter par le processus « pid11 »
   }
   else  { // si je suis le processus « pid1 »
      pid12=fork(); // le processus « pid1 » crée le fils « pid12 ».
      if (pid12==0) {  // si je suis le processus « pid12 »
         // code des instructions à exécuter par le processus « pid12 »
      }
   }
}
```

Donc l'arbre généré par les instructions précédentes est le suivant:



Ensuite chacun des processus « pid11 » et « pid12 » génère un arbre de processus identique à celle de la question 1 donc chacun des deux processus va exécuter les mêmes instructions données dans le programme de la question 1.

```
if (fork()) {
   if (fork()) {
      if (fork()) {
```

Donc le programme C complet est le suivant:

```
….
main ( ) {
    pid1=fork(); // le processus père crée le fils « pid1 ».
    if (pid1==0) {   // si je suis le processus « pid1 »
       pid11=fork(); // se processus « pid1 » crée le fils « pid11 ».
       if (pid11==0) {  // si je suis le processus « pid11 »
          if (fork()) { // exécuté par « pid11 » qui crée un premier fils
             if (fork()) {  // exécuté par « pid11 » qui crée un deuxième fils
                if (fork()) { // exécuté par« pid11 » qui crée un troisième fils
                }
             }
          }
       }
    }
    else  { // si je suis le processus pid1
       pid12=fork(); // le processus « pid1 » crée le fils « pid12 ».
       if (pid12==0) {  // si je suis le processus « pid12 »
          if (fork()) { // exécuté par « pid12 » qui crée un premier fils
             if (fork()) {  // exécuté par « pid12 » qui crée un deuxième fils
                if (fork()) { // exécuté par « pid12 » qui crée un troisième fils
                }
             }
          }
       }
    }
    while (wait(0)!=-1);
}
```

Exercice 4:

1. *Donnez l'arbre de processus générés par le programme C suivant :*

```
…
main() {
        pid_t  pid1, pid2, pid3;
        pid1=fork();
        pid2=fork();
        if (pid2!=0) {
                pid3=fork();
        }
        while (wait(0)!=-1);
}
```

2. *Donnez l'arbre de processus générés par le programme C suivant :*

```
…
main() {
    pid_t  pid1, pid2, pid3, pid4;
    pid1=fork();
    pid2=fork();
    if (pid2!=0) {
       pid3=fork();
    }
    pid4=fork() ;
    while (wait(0)!=-1);
}
```

Réponses:

1.

- L'instruction :

  pid1=fork();

  est exécutée par le processus père qui crée le fils « pid1 »
- L'instruction :

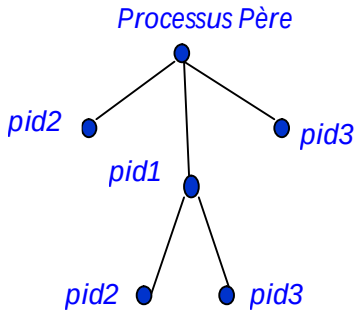
  pid2=fork();

  est exécutée par les processus père et « pid1 ». Donc chacun d'eux crée un processus fils nommé « pid2 ».

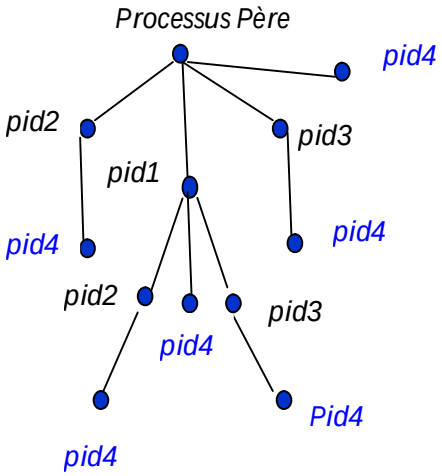
- L’instruction :

```
if (pid2!=0) { // si je ne suis pas « pid2 » (si je suis le père ou « pid1 »)
        pid3=fork(); //exécuté par le père et par « pid1 »
}
```

Donc chacun des processus père et « pid1 » crée le processus « pid3 ». Donc l'arbre généré est le suivant:

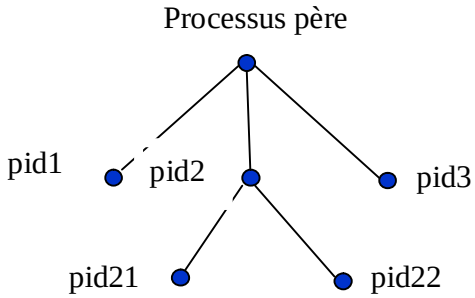


2. L’instruction « pid4=fork() ; » est exécutée par tous les processus. Donc chaque processus crée un processus fils « pid4 ». Donc l'arbre généré est le suivant:



Exercice 5 :

Ecrire un programme C qui permet de générer l'arbre de processus représenté par la figure ci-dessous où le processus père crée trois processus fils à savoir « pid1 », « pid2 » et « pid3 ». Le processus fils « pid2 » crée à son tour deux processus à savoir « pid21 » et « pid22 ». Chaque processus affiche son PID (numéro d'identification) ainsi que le PID de son père.



Réponses :

```
…
main ( ) {
    pid_t pid1, pid2, pid3, pid21, pid22;
    pid1=fork();   // tout d'abord le père crée le fils « pid1 »
    if (pid1!=0) { // si je ne suis pas « pid1 » (si je suis le processus père)
       pid2=fork();  // le processus père crée le processus fils « pid2 »
       if (pid2!=0) { // si je ne suis pas « pid2 » (si je suis le processus père)
          pid3=fork();  // le processus père crée le processus fils « pid3 »
          if (pid3!=0) { // si je ne suis pas « pid3 » (si je suis le père)
             // le processus père attend la terminaison de ses 3 fils :
             while (wait(0) !=-1)
          }
          else { // si je suis le processus « pid3 »
             // « pid3 » affiche son identifiant et celui de son père.
             printf("Processus pid3 = %d pere =  %d \n", getpid(), getppid());
          }
       }
       else {  // si je suis le processus « pid2 »
          pid21=fork(); // le processus « pid2 » crée le fils « pid21 »
```

```
            if (pid21!=0) { // si je ne suis pas « pid21 » (si je suis « pid2 »)
               pid22=fork();  // le processus « pid2 » crée le fils « pid22 »
               if (pid22!=0) { // si je ne suis pas « pid22 » (si je suis « pid2 »)
                  printf("Processus pid2= %d pere = %d \n", getpid(), getppid());
                  // Le processus «pid2» attend la terminaison de ses 2 fils
                  while (wait(0) !=-1);
               }
               else {  // si je suis le processus « pid22 »
                  // « pid22 » affiche son identifiant et celui de son père.
                  printf("pid22 = %d   mon pere =  %d \n",getpid(),getppid());
               }
            }
            else { // si je suis le processus « pid21 »
               // « pid21 » affiche son identifiant et celui de son père.
               printf("pid21 = %d   mon pere =  %d \n",getpid(),getppid());
            }
         }
      }
      else { // si je suis le processus « pid1 »
         // « pid1 » affiche son identifiant et celui de son père.
         printf("pid1 = %d   pere =  %d \n",getpid(),getppid());
      }
   }
```

Exercice 6 :

Considérons les programmes suivants :

```
#include<unistd.h>
main() {
      fork() && fork() ;
}
#include<unistd.h>
main() {
      ( fork() || fork() ) ;
}
```

```
#include<unistd.h>
main() {
        fork()&&( fork() || fork() ) ;
}
```

1. *Donner l'arbre de processus généré par l'exécution de chaque programme.*

2. *Implémenter et exécuter ces programmes pour comparer les résultats.*

Réponses :

Pour bien comprendre, on introduit un identifiant pour chaque processus créé (à chaque appel de « fork() »).

Premier programme

```
main() {
    (pid1=fork()) && (pid2=fork()) ;
}
```

- Tout d'abord, l'instruction (pid1=fork()) est exécutée par le processus père qui crée le processus fils « pid1 ».
- Ensuite, l'instruction :

```
pid2=fork();
```

n'est exécutée que si l'évaluation de la première instruction est vrai (différente de zéro) c'st à dire si (pid1 !=0). Donc l'instruction (pid2=fork()) est exécutée par le processus père qui crée le deuxième fils « pid2 ».

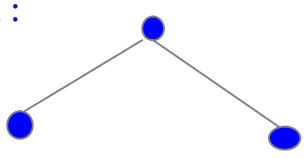
Donc

```
(pid1=fork()) && (pid2=fork()) ;
```

est équivalente à:

```
pid1=fork();
if (pid1 !=0)
   pid2=fork();
```

Donc l'arbre généré est le suivant :



Deuxième programme

```
main() {
    ( (pid1=fork()) || (pid2=fork()) ) ;
}
```

L'instruction (pid1=fork()) est initialement exécutée par le processus père qui crée le processus fils « pid1 ». Ensuite, si (pid1==0), c'est-à-dire si je suis le processus fils, alors j'exécute l'instruction (pid2=fork()) qui crée le fils « pid1 ».

Donc

```
(pid1=fork()) || (pid2=fork()) ;
```

est équivalente à:

```
pid1=fork();
if (pid1 ==0)
   pid2=fork();
```

Donc l'arbre généré est le suivant :

Troisième programme

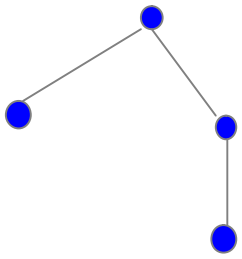
```
main() {
    (pid1=fork())&&( (pid2=fork()) || (pid3=fork()) ) ;
}
```

Tout d'abord l'instruction (pid1=fork()) est évaluée par le processus père qui crée le fils « pid1 ».

```
Si (pid1 !=0)  // si je suis le processus père
```

Le processus père exécute l'instruction (pid2=fork()) || (pid3=fork()) qui génère l'arbre précédent.

Donc l'arbre généré est le suivant :



Exercice 7 :

Ecrire un programme telle que l'arbre de processus généré par son exécution et un :

- *arbre binaire de profondeur 1*
- *arbre binaire de profondeur 2*

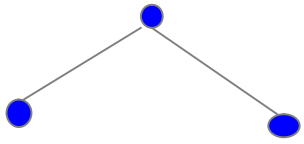
Réponses :

Arbre binaire de profondeur 1 (voir exercice précédent)

```
main() {
    fork() && fork() ;
}
```

ou

```
main() {
    if(fork()!=0) { // le père crée un premier fils (par exemple le fils droit
        fork() ; // exécuté par le père qui crée le fil gauche (2ème fils)
    }
}
```



Arbre binaire de profondeur2

```
main() {
    if(fork()!=0) { // le père crée le fils droit
        if(fork()==0) {  // le père crée le fils gauche
            // le fils gauche crée à son tour deux fils droit et gauche
            fork() && fork());
        }
    }
    else { // le fils droit
        // le fils droit crée à son tour deux fils droit et gauche
        fork() && fork());
    }
    while (wait(0)) !=0 ;
}
```

Exercice 8 :

1.*Donner l'arbre de processus générés par le programme suivant :*

```
main ( ) {
  if (fork()) {
     if (fork()) {
          if (fork()==0) {
               fork()&&fork();
          }
     }
     else {
          fork();
     }
  }
  while (wait(0)!=-1);
}
```

2.*Donner l'arbre de processus générés par le programme suivant :*

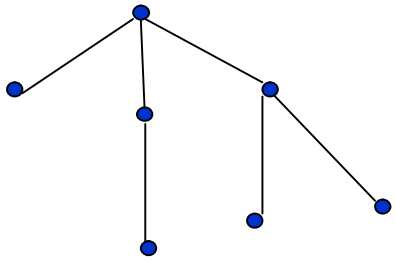
```
main ( ) {
  if (fork()) {
     if (fork()) {
          if (fork()==0) {
               fork()&&fork();
          }
     }
    else {
          fork();
    }
    fork();
  }
  while (wait(0)!=-1);
}
```

Réponses:

1.

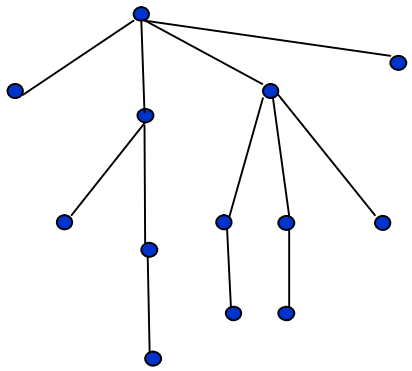
```
main ( ) {
  if (fork()) { // le père crée le premier fils
        if (fork()) { // exécuté par le père qui crée un deuxième fils
              if (fork()==0) { // fork() est exécuté par le père qui crée un troisième fils
                    fork()&&fork(); // exécuté par le troisième fils qui crée deux fils
              }
        }
        else { // exécuté par le deuxième fils
              fork();  // le deuxième fils crée un fils
        }
  }
  while (wait(0)!=-1);
}
```

L'arbre généré est le suivant:



2. La différence avec le programme de la question 1 est la dernière instruction « fork() ». Cette instruction est exécutée dans le premier bloc « if {} » qui concerne le processus père et non le premier processus fils créé. Donc cette est exécutée par tous les processus sauf par le premier fils crée. Donc, chaque processus de l'arbre précédent crée un fils sauf le premier fils.

L'arbre généré est le suivant:

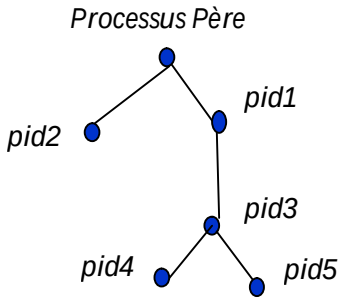


Exercice 9 :

1. *Donnez l'arbre de processus généré par le programme ci-dessous. Sur chaque nœud de l'arbre, if faut noter les identifiants des processus générés (Processus père, pid1, pid2).*

```
……
main() {
   (pid1=fork()) && (pid2=fork());
}
```

2. *Considérons l'arbre de processus représenté par la figure ci-dessous. Le processus père crée deux processus fils « pid1 » et « pid2 ». Le processus « pid1 » crée un processus fils « pid3 », ensuite le processus « pid3 » crée deux processus fils «pid4 » et « pid5 ».*



a. *Sans utiliser le résultat de la question 1, écrire un programme C qui permet de générer l'arbre de processus représenté par la figure ci-dessus.*

b. *En utilisant le résultat de la question 1, écrire un programme C qui permet de générer l'arbre de processus représenté par la figure ci-dessus.*

Réponse:

1. La première instruction:

pid1=fork())

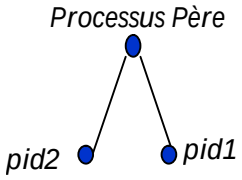
est exécutée par le processus père qui crée le processus fils « pid1 » donc la valeur de « pid1 » est différente de 0 dans le processus père et elle est égale à zéro dans le processus « pid1 ».

La deuxième instruction:

pid2=fork();

sera exécutée seulement si la valeur de la première instruction est vrai (c'est-à-dire si (pid1 !=0) donc elle est exécutée par le processus père qui crée le deuxième fils « pid2 »

Donc l'arbre de processus générés est le suivant :



2.

2.a. Sans utiliser le résultat de la question 1

```
….
main() {
  // déclarations
  pid1=fork() ; // le processus père crée le fils « pid1 »
  if (pid1!=0) { // si je suis le processus père
     pid2=fork(); // le processus père crée le fils « pid2 »
  }
  else { // si je suis le processus « pid1 »
    pid3=fork(); // le processus « pid1 » crée le processus « pid3 »
    if (pid3==0) { // si je suis le processus « pid3 »
        pid4=fork(); // le processus « pid3 »  crée le processus « pid4 »
        if (pid4!=0) // si je sui le processus pid3
          pid5=fork(); // le processus « pid3 » crée le processus « pid5 »
    }
  }
  while(wait(0)!=-1);
}
```

2.b. Pour la création des processus « pid1 » et « pid2 » ainsi que pour la création des processus « pid4 » et « pid5 » on utilise le résultat de la question 1.

```
….
main() {
   // déclarations
   // le père crée les processus « pid1 » et « pid2 » avec l'instruction:
```

```
        (pid1=fork()) && (pid2=fork());
        // Puisque le processus « pid1 » est créé avant « pid2 » donc la valeur
        // « pid1 » est différente de 0 dans les processus père et « pid2 ». Donc
        // l'instruction: if (pid1==0) est vrai uniquement dans le processus « pid1 »
        if (pid1==0) { // si je suis le processus « pid1 ».
            pid3=fork(); // le processus « pid1 » crée le processus fils « pid3 ».
            if (pid3==0) // si je suis le processus « pid3 »
            // «pid3» crée les processus «pid4» et «pid5» avec l'instruction:
            (pid4=fork()) && (pid5=fork());
        }
        while(wait(0)!=-1);
}
```

Exercice 10 :

Donnez l'arbre de processus généré par le programme C ci-dessous. Sur chaque nœud de l'arbre, il faut noter les identifiants (processus père, pid1, pid2, pid3) des processus générés.

```
….
main() {
    pid_t  pid1, pid2, pid3;  // les identifiants des processus.
    pid1=fork();
    pid2=fork();
    if ( ((pid1==0) && (pid2!=0)) || ((pid1 !=0) && (pid2==0)) ) {
        pid3=fork();
    }
    while(wait(0) !=-1) ;
}
```

Réponses:

- L'instruction:
  pid1=fork();
  est exécuté par le processus père qui crée le fils « pid1 »

- L’instruction :

  pid2=fork();

  est exécutée par les processus père et « pid1 ». Donc chacun d’eux crée processus « pid2 » (même nom de variable mais avec des identifiants différents dans chaque processus).

- L'instruction :

  if (((pid1 ==0) && (pid2!=0)) || ((pid1 !=0) && (pid2==0))) {

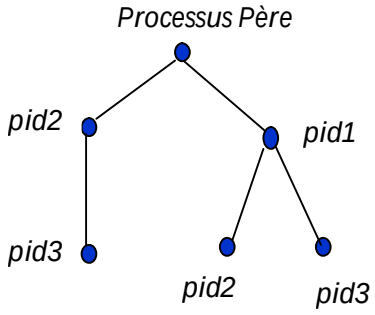
  est exécutée par chacun des processus père, « pid1 » et « pid2 ».

  Si l’évaluation de la condition, dans un processus, retourne vrai alors ce processus exécute l'instruction ci-dessous qui crée le processus «pid3».

  pid3=fork();

  - Dans le processus père : on a « pid1 !=0 » et « pid2 !=0 » donc l’expression précédente retourne faux dans le processus père.
  - Dans le processus « pid1 » : on a « pid1==0 » et « pid2 !=0 » car « pid2 » est créé par « pid1 ». Donc l’expression précédente retourne vrai dans le processus « pid1 » et par conséquent l’instruction « pid3=fork() ; » est exécutée par le processus « pid1 ».
  - Dans le processus « pid2 » : on a « pid2==0 » et « pid1 !=0 » car « pid1 » est créé avant « pid2 » et par conséquent la valeur de « pid1 » dans le processus « pid2 » est différente de 0. Donc l’expression précédente retourne vrai dans le processus « pid2 » et par conséquent l’instruction « pid3=fork() ; » est exécutée par le processus « pid2 ».

Donc l’arbre des processus générés est le suivant:



Exercice 11:

Considérons le programme suivant :

```
#include<unistd.h>
main( ) {
fork();
fork();
fork();
}
```

1. *Donner l'arbre des processus généré par ce programme.*

2. *Remplacez le premier appel à « fork() » par un appel à « execl() » qui exécute le programme lui-même. Que se passe-t-il ?*

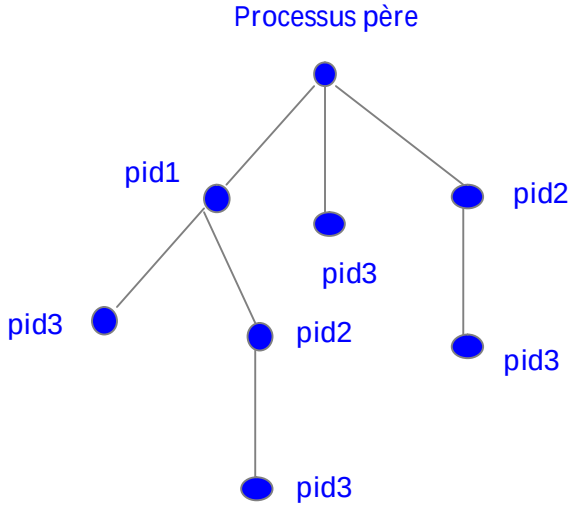
3. *Remplacez le deuxième appel à « fork() » par un appel à « execl() » qui exécute le programme lui-même. Que se passe-t-il maintenant ?*

Réponses :

1. Pour bien comprendre, on introduit dans le programme un identifiant pour chaque processus. Soient «pid1», «pid2» et «pid3» les identifiants des processus.

```
main( ) {
    pid_t pid1, pid2, pid3;
    pid1=fork();  // le processus père crée un premier fils « pid1 »
    pid2=fork(); // cette instruction est exécutée par le père et par « pid1 ».
                 // Donc, chacun d'eux crée un processus fils « pid2 ».
    pid3=fork();// cette instruction est exécutée par tous les processus créés.
                 // Donc, chacun des processus crée un processus fils « pid3 ».
    }
```

L'arbre généré et le suivant:



2. Supposons que le nom du programme est « exo2_2_fork » est qu'il est dans le répertoire « /home/mdaoudi/ ». On remplace le premier fork() par un appel de « execl() »

```
main() {
    execl("/home/mdaoudi/exo2_2_fork","exo2_2_fork",NULL);
    fork();
    fork();
}
```

Exécution : Boucle infinie puisque il y a un appel récursif du même programme sans aucune condition d'arrêt.

3. On remplace le deuxième « fork() » par un appel de « execl() ».

```
main() {
    fork();
    execl("/home/mdaoudi/exo2_2_fork","exo2_2_fork",NULL);
    fork();
}
```

Exécution : Le système se bloque car création un nombre infini de processus qui chargera le système.

Chapitre 2 : Gestion des signaux

1. Introduction

Les processus ayant le même propriétaire peuvent communiquer entre eux en utilisant les signaux. Un signal (ou interruption logicielle) est un message très court qui peut être envoyé à un processus par :

- le système d'exploitation (déroutement) : pour informer les processus utilisateurs d'un événement anormal (par exemple une division par 0) qui vient de se produire durant l'exécution d'un programme.
- un processus utilisateur.
- la frappe d'une touche du terminal (par exemple les touches CTRL-C et CTRL \ génèrent les signaux SIGINT et SIGQUIT dont l'action par défaut est de terminer le processus).

Un programme qui traite les signaux doit inclure « #include<signal.h> ».

2. Caractéristiques des signaux

Chaque signal a un nom (constante symbolique commençant par « SIG ») et un numéro allant de 1 à «NSIG» (ou «_NSIG»).

- La numérotation peut être différente selon le système.
- Le récepteur du signal ne connaît pas son émetteur.

Quelque signaux

2. SIGINT : interruption terminal
3. SIGQUIT : signal quitter du terminal
8. SIGFPE : erreur arithmétique
9. SIGKILL : terminaison impérative. Ne peut être ignoré ou intercepter
10. SIGUSR1 : signal utilisateur 1
12. SIGUSR2 : signal utilisateur 2
13. SIGPIPE : écriture dans un conduit sans lecteur disponible
14. SIGALRM : alarme horloge: expiration de time

4. Etats des signaux

- Un signal pendant (pending) est un signal en attente d'être pris en compte.
- Un signal est délivré lorsqu'il est pris en compte par le processus qui le reçoit. La prise en compte d'un signal entraîne l'exécution d'une fonction spécifique au signal (le « handler »).

5. Comportement des signaux

A chaque signal est associé un gestionnaire du signal (« handler ») par défaut appelé « SIG_DFL ». Le traitement par défaut peut être :

- ignoré en associant au signal le handler «SIG_IGN ».
- remplacé par un traitement personnalisé en associant au signal un nouveau handler.

6. Signaux particuliers

Pour tous les signaux, l'utilisateur peut remplacer le « handler » par défaut par un « handler » personnalisé, à l'exception des signaux:

- « SIGKILL » permet de tuer un processus.
- « SIGSTOP » permet de stopper un processus (stopper pour reprendre plus tard, pas arrêter).
- « SIGCONT » permet de faire reprendre l'exécution d'un processus stoppé (après un « SIGSTOP »).

7. Manipulation des signaux

La manipulation des signaux peut se faire dans un programme utilisateur, essentiellement par les deux primitives principales :

- la fonction « signal() » pour associer un nouveau handler (une action personnalisée) à un signal donné.
- la fonction « kill() » pour envoyer des signaux.

Envoi d'un signal par un processus

La primitive «int kill(pid_t pid, int sig) » permet d'envoyer un signal « sig » vers un ou plusieurs processus.

- « sig » désigne le signal à envoyer (on donne le nom ou le numéro du signal). Si « sig » est égal à 0 aucun signal n'est envoyé, mais la valeur de retour de la primitive « kill() » permet de tester l'existence ou non du processus « pid » (si kill(pid,0) retourne 0 (pas d'erreur) le processus de numéro « pid » existe).
- « pid » désigne le ou les processus destinataires (récepteurs du signal).
    - Si pid > 0, le signal est envoyé au processus d'identité « pid ».
    - Si pid=0, le signal est envoyé à tous les processus qui sont dans le même groupe que le processus émetteur (processus qui a appelé la primitive « kill() »).

- Si pid < -1, le signal est envoyé à tous les processus du groupe du processus de numéro |pid|.

- Elle renvoie 0 si le signal est envoyé et -1 en cas d'échec.

Traitement personnalisé de signal (Mise en place d'un handler)

La primitive «signal (int sig, new_handler) » installe des handlers personnalisés pour le traitement des signaux. Cette primitive fait partie du standard de C. Elle met en place le handler spécifié par « new_handler() » pour le signal « sig ».

- La fonction « new_handler » est exécutée par le processus à la délivrance (la réception) du signal. Elle reçoit le numéro du signal.
- A la fin de l'exécution de cette fonction, l'exécution du processus reprend au point où elle a été suspendue.

Attente d'un signal

La primitive « pause() » bloque (endorme: attente passive) le processus appelant jusqu'à l'arrivée d'un signal **quelconque** puis action en fonction du comportement associé au signal. Elle est déclarée dans « <unistd.h> ».

8. La fonction « raise () »

La fonction « int raise (int sig) » est équivalente à « kill(getpid(),sig) ».

9. La fonction « alarm() »

L'appel système «unsigned int alarm(unsigned int sec) » permet à un processus de gérer des temporisation (timeout). Le processus est avertit de la fin d'une temporisation par un signal « SIGALRM ».

- Pour annuler une temporisation avant qu'elle n'arrive à son terme, on appelle « alarm(0) ».
- Le traitement par défaut du signal « SIGALRM » est la terminaison du processus.

10. Signal SIGHUP

« SIGHUP =Signal Hang UP » (déconnexion de l'utilisateur): ce signal est envoyé automatiquement au processus si on ferme le terminal auquel il est attaché.

Exercice 1 :

Considérons le programme suivant:

```
#include <stdio.h>
#include <signal.h>
main(){
      while (1) {
            printf(" le processus boucle \n");
   }
}
```

1. *Que se passe-t-il si on appuie sur Ctrl-C ?*

2. *Quel signal sera envoyé au processus lorsqu'on appuie sur Ctrl-C ?*

3. *Modifier le programme afin que lorsqu'on appuie sur Ctrl-C, le processus affiche le message « Arrêt dans 10 secondes », se bloque pendant 10 secondes ensuite il se termine.*

Réponses :

1. Le programme boucle en affichant " le processus boucle ". Lorsqu'on appuie sur Ctrl-C l'affichage sera interrompu.

2. Le signal envoyé est « SIGINT »

3. **Principe :**

- on installe un nouveau comportement (handler) pour le signal « SIGINT ».
- après réception de «SIGINT», le message "Arrêt dans 10 secondes" sera affiché.
- le processus se bloque pendant 10 secondes.
- on rétablit le comportement par défaut du signal « SIGINT »
- envoie du signal « SIGINT » à lui-même pour se terminer.

Code C

```
// Le nouveau handler pour le signal « SIGINT »
void hand(int s){
      printf("Arrêt dans 10 secondes \n ");
      sleep(10);  // se bloque pendant 10 secondes
      // rétablir le comportement par défaut du signal « SIGINT »
      signal(SIGINT, SIG_DFL);
      kill(getpid(),s);  // envoie de « SIGINT » à lui-même pour se terminer
}
```

```
int main(){
        signal(SIGINT,hand);
        while (1) {
                printf("Le processus boucle \n");
        }
}
```

Exercice 2 :

Ecrire un programme qui masque les signaux «SIGINT » et « SIGQUIT » pendant 30 secondes et ensuit les rétablit. Pour connaitre les combinaisons des touches taper la commande « %stty -a ».

Réponses:

```
main( ) {
        int i,p;
        // tout d'abord on ignore le comportement par défaut des signaux
        // «SIGINT » et « SIGQUIT ».
        signal(SIGINT, SIG_IGN);  // ignorer le signal «SIGINT »
        signal(SIGQUIT,SIG_IGN);  // ignorer le signal « SIGQUIT »
        printf("Les signaux SIGINT et SIGQUIT sont ignorés \n");
        sleep(30);  // se bloquer pendant 30 secondes
        printf("Rétablissement des signaux \n");
        // on rétablit le comportement par défaut des deux signaux
        signal(SIGINT,SIG_DFL); // traitement par défaut de «SIGINT »
        signal(SIGQUIT,SIG_DFL);   // traitement par défaut « SIGQUIT »
        while(1);
}
```

Exercice 3 :

Ecrire un programme qui effectue un ping-pong des signaux entre un processus et son fils. Le signal « SIGUSR1 » est envoyé par le fils au père et « SIGUSR2 » est envoyé du père au fils. Le premier signal est envoyé par le père. On récupère le nombre de signaux sur la ligne de commande.

Réponses:

Version1 : avec risque de blocage

Dans cette solution, il ya risque de blocage: par exemple le père fait « kill() » alors que le fils n'a pas encore fait pause()

….

```
void handlerSignal( ) ;
int cptr , limite ;
pid_t pid;
main ( int argc,char **argv ) {
    if(argc==1) {
        printf (" il faut passer la valeur de limite en argument  \n" ) ;
        exit(1) ;
    }
    limite=atoi(argv[1]);
    cptr =0;
    pid=fork();   // le processus père crée le processus fils « pid »
    if(pid==0) {  // si je suis le processus fils
        // installation d'un handler pour le signal « SIGUSR2 »
        signal(SIGUSR2, handlerSignal ) ;
        printf(" Fils => gestionnaire installe, PID=%d\n" , getpid( )) ;
        while (cptr<limite) {
            pause(); // attend le signal « SIGUSR2 » du père
            cptr++;
            kill(getppid(),SIGUSR1);  // envoi de « SIGUSR1 » au père
            printf ( "cpt = %d pid= %d signal = %d envoye a %d\n", cptr,
                    getpid(), SIGUSR1,getppid());
        }
    }
    else {  // si je suis le processus père
        // installation d'un handler pour le signal « SIGUSR1 »
        signal(SIGUSR1,handlerSignal) ;
        sleep(1); // laisser le temps au fils pour installer le handler
        printf("Pere => gestionnaire installe , PID=%d \n", getpid()) ;
```

```
            while (cptr<limite){
                cptr++;
                kill(pid,SIGUSR2 );  // envoi du signal "SIGUSR2" au fils pid
                printf ( "cpt = %d  pid= %d signal = %d   envoye a %d\n",cptr,
                    getpid(), SIGUSR2,pid);
                pause();// attend le signal « SIGUSR1 » du fils
                }
        }
}
void handlerSignal(int sig){
        printf ("\t \t [%d ] gestionnaire %d => signal capte\n",getpid(), sig);
}
```

Version 2 : solution sans blocage

Pour éviter le risque de blocage (éviter qu'un processeur fasse « kill() » alors que le récepteur du signal n'a pas encore fait « pause() »), on remplace « pause() » par une attente active "tant que le signal n'est pas encore arrivé", en introduisant deux variables v1 (dans le processus père) et v2 (dans le processus fils). Ces variables changent leur valeur dès la réception d'un signal.

```
int v1=1;
int v2=1;

void hand_pere( int sig) {
        printf ("\t \t [%d ] gestionnaire %d => signal capte\n",getpid(), sig);
        v1=0; // à la réception du signal, on change la valeur de v1
}

void hand_fils( int sig) {
        printf ("\t \t [%d ] gestionnaire %d => signal capte\n",getpid(), sig);
        v2=0;  // à la réception du signal, on change la valeur de v2
}
```

```
int cptr , limite ;
pid_t pid;
main ( int argc,char **argv ) {
        …
        pid=fork();
        if(pid==0) {  // le processus fils
            // installe un nouveau handler pour le signal « SIGUSR2 »
            signal(SIGUSR2, hand_fils ); {
            printf(" Fils => gestionnaire installe, PID=%d\n" , getpid( )) ;
            while (cptr<limite) {
                while (v2) ; // attente active
                v2=1;  //rétablir la valeur de v2
                cptr++;
                kill(getppid(),SIGUSR1); // envoyer  « SIGUSR1 » au père
                printf ( "cpt = %d pid= %d signal = %d envoye a %d\n", cptr,
                        getpid(),  SIGUSR1,getppid());
            }
        }
        else {  // si je suis le processus père
            // installe un nouveau handler pour le signal « SIGUSR1 »
            signal(SIGUSR1,hand_pere) ;
            sleep(1); // laisser le temps au fils pour installer le nouveau handler
            printf("Pere => gestionnaire installe , PID=%d \n", getpid()) ;
            while (cptr<limite){
                cptr++;
                kill(pid,SIGUSR2 ); // envoyer « SIGUSR1 » au fils
                printf ( "cpt = %d  pid= %d signal = %d   envoye a %d\n",cptr,
                        getpid(), SIGUSR2,pid);
                while (v1) ;  // attente active
                v1=1;  // rétablir la valeur de v1
            }
        }
}
```

Exercice 4 :

1. *Donnez l'arbre de processus généré par le programme ci-dessous :*

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
main() {
      if (fork()==0) {
         printf("Sortie 1 \n ");
      }else {
         if (fork()!=0) {
            printf("Sortie 2 \n ");
         } else {
            if (fork()==0) {
               printf("Sortie 3 \n ");
               exit(0);
             }
            printf("Sortie 4 \n ");
          }
       }
  }
```

2. *Modifier ce programme afin que son exécution commence par l'affichage de «Sortie 3», suivi de « Sortie 4 », ensuite « Sortie 1» et en fin «Sortie 2».*

N.B : ne pas utiliser la fonction « sleep() ».

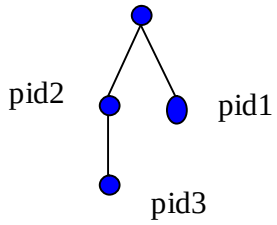
Réponses:

1. Pour bien comprendre, on introduit un identifiant pour chaque processus créé (à chaque appel de « fork() »).

```
main() {
      if ((pid1=fork())==0) {  // le processus père crée le premier fils « pid1 »
         printf("Sortie 1 \n ");   // exécuté par le processus « pid1 »
      }
      else { // si je suis le processus père
         if((pid2=fork())!=0) { // le processus père crée le deuxième fils «pid2»
            printf("Sortie 2 \n ");  // exécute par le processus père.
         }
```

```
        else { // si je suis le deuxième processus fils (le processus « pid2 »)
            if ((pid3=fork())==0) { //  « pid2 » crée un processus fils « pid3 »
                printf("Sortie 3 \n ");  // exécuté par le processus « pid3 »
                exit(0);
            }
            printf("Sortie 4 \n "); // exécuté par « pid2 »
        }
    }
}
```

Donc l'arbre généré est le suivant



2. L'ordre d'affichage est le suivant: Tout d'abord "pid3", suivi de "pid2", suivi de "pid1" et en fin le processus père.

Remarque: La solution suivante est basée sur la synchronisation sur terminaison des processus et l'envoie des signaux. On peut élaborer une autre solution basée uniquement sur les signaux.

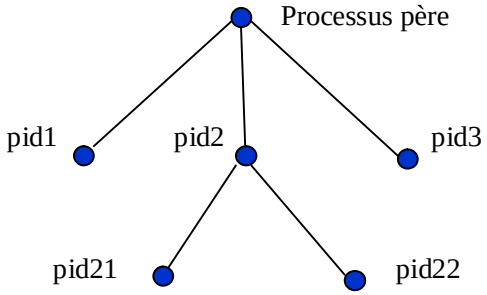
- Le processus « pid3 » affichage « Sortie 3 » et se termine.
- Le processus « pid2 » attend la terminaison de son fils « pid3 », ensuite affiche « Sortie 4 » et en fin informe le processus « pid1 » en lui envoyant le signal « SIGUSR1 ». Il faut noter que le processus « pid2 » est créé après le processus « pid1 » donc le processus « pid2 » connait l'identifiant du processus « pid1 ».
- Le processus « pid1 » installe un handler pour le signal « SIGUSR1 » enuite attend un signal (envoyé par « pid2 ») avant d'afficher « Sortie 1 ».
- Le père attend la terminaison de tous ses fils avant d'afficher « Sortie 2 ».

```
main() {
    if ((pid1=fork())==0) {  // le processus père crée le premier fils « pid1 »
        // le processus « pid1 » installe un handler pour le signal « SIGUSR1 »
        signal(SIGUSR1,hand);
        pause(); // « pid1 » attend l'arrivé d'un signal
        printf("Sortie 1 \n ");   // executé par le processus fils
    }
    else { // si je suis le processus père
        if((pid2=fork())!=0) { // le processus père crée le fils « pid2 »
            while ((wait(0)!=-1));  // attend la terminaison de ses fils
            printf("Sortie 2 \n ");  // exécute par le processus père.
        }
```

```
        else { // exécuté par le processus fils « pid2 »
            if ((pid3=fork())==0) {  // exécuté par « pid2 » qui crée « pid3 »
                printf("Sortie 3 \n ");  // exécuté par « pid3 »
            }
            wait(0);  // « pid2 » attend la terminaison de son fils « pid3 »
            printf("Sortie 4 \n ");
            kill(pid1,SIGUSR1); // «pid » envoie le signal «SIGUSR1» à «pid1»
        }
    }
}
```

Exercice 5 :

1. *Ecrire un programme C qui permet de générer l'arbre de processus représenté par la figure ci-dessous où le processus père crée trois processus fils à savoir « pid1 », « pid2 » et « pid3 ». Le processus fils « pid2 » crée à son tour deux processus à savoir « pid21 » et « pid22 ». Chaque processus affiche son PID (numéro d'identification) ainsi que le PID de son père.*



2. *Supposons que l'instruction de création du processus « pid21 » est effectuée avant celle de l'instruction de création du processus « pid22 ». Modifier le programme précédent afin de s'assurer que l'affichage de « pid22 » s'effectue avant l'affichage de « pid21 » avant l'affichage de « pid2 ».*

Réponses:

1. Le processus père crée tout d'abord le fils « pid1 », instruction

```
pid1=fork();
```

ensuite il crée le fils « pid2 » et le fils « pid3 », instruction

```
if (pid1!=0) {  // si je ne suis pas «pid1» (si je suis le père)
    pid2=fork();
    if (pid2!=0) {
```

```
        pid3=fork();
    }
```

Le processus « pid2 » crée les processus « pid21 » et « pid22 » en utilisant les instructions

```
        pid21=fork();
        if (pid21!=0) {
            pid22=fork();
        }
```

Ou

```
((pid21=fork()) && (pid22=fork())
```

Programme C complet avec les fonctions d'affichage

```
…..
main ( ) {
    pid_t  pid1,pid2,pid3,pid21,pid22;
    pid1=fork();   // le processus père crée le fils « pid1 »
    if (pid1!=0) { // exécuté par le processus père
        pid2=fork();  // le processus père crée le fils « pid2 »
        if (pid2!=0) { // exécuté par le processus père
            pid3=fork();  // le processus père crée le fils « pid3 »
            if (pid3!=0) { // si je suis le processus père
                while (wait(0)!=-1);
            }
            else { // le processus « pid3 »
                printf(" pid3 = %d mon pere = %d \n",getpid(),getppid());
            }
        }
        else { // le processus « pid2 »
            pid21=fork();    // le processus « pid2 » crée le fils « pid21 »
            if (pid21!=0) { // exécuté par le processus « pid2 »
                pid22=fork();   // le processus « pid2 » crée le fils « pid22 »
                if (pid22!=0) { // exécuté par le processus « pid2 »
                    printf(" pid2 = %d   pere = %d \n",getpid(),getppid());
                    while (wait(0)!=-1);
                }
```

```
            else { // le processus « pid22 »
                printf("pid22 = %d  pere = %d \n",getpid(),getppid() );
            }
        }
        else { // exécuté par le processus « pid21 »
            printf("pid21 = %   pere = %d \n",getpid(),getppid());
        }
    }
}else { // exécuté par le processus « pid1 »
    printf("proc pid1 = %d  mon pere = %d \n",getpid(),getppid());
}
}
```

2.

- Puisque le processus « pid22 » est créé après le processus « pid21 » alors le processus « pid22 » connait l'identifiant du processus « pid21 ».
- Pour s'assurer que l'affichage de « pid22 » s'effectue avant l'affichage de «pid21», le processus « pid22 », après son affichage, il informe le processus «pid21» en lui envoyant un signal.
- Le processus « pid2 » attend la terminaison de ses deux fils avant de faire son affichage.

Programme C : partie relative au processus « pid2 », le reste est identique au programme précédent.

```
void hand(int sig) { }
main ( ) {
    … // identique au programme précédent
    else { // exécuté par le processus « pid2 »
        pid21=fork();
        if (pid21!=0) { // exécuté par le processus « pid2 »
            pid22=fork();
            if (pid22!=0) { // exécuté par le processus « pid2 »
                while (wait(0)!=-1);
                printf(" pid2 = %d   pere = %d \n",getpid(),getppid());
            }else { // exécuté par le processus « pid22 »
```

```
                printf(" pid22 = %d  pere = %d \n",getpid(),getppid());
                sleep(1); // laisser le temps à « pid21 » d'installer le handler
                kill(pid21,SIGUSR1);  // envoie de « SIGUSR1 » à « pid21 »
              }
            }else { // exécuté par le processus « pid21 »
              signal(SIGUSR1,hand);  // installe un handler pour « SIGUSR1 »
              pause();  // attente d'un signal
              printf("pid21 = %d  pere = %d \n",getpid(),getppid());
            }
          }
        …
    }
```

Exercice 6 :

1. Donner l'arbre de processus générés par le programme C suivant :

```
main ( ) {
        pid_t  pid1, pid2;
        pid1=fork();
        if ( pid1!=0 )
                pid2=fork();
        while (wait(0)!=-1);
}
```

2. Le numéro du signal « SIGUSR1 » est égal à 10. Donner les résultats possibles de l'exécution du programme C suivant :

```
main ( ) {
        pid_t  pid1, pid2;
        pid1=fork();
        if (pid1 !=0) {
                pid2=fork();
                if (pid2==0) {
                        kill(pid1, SIGUSR1);
                }
        }
```

```
        else {
                signal(SIGUSR1, hand);
                pause();
        }
        while (wait(0)!=-1);
}
 void hand(int signum) {
        printf(" signal recu :  %d \n", signum);
}
```

Réponses:

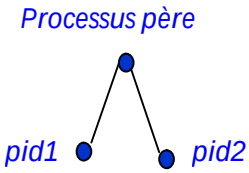
1. Le processus père crée tout d'abord le processus « pid1 », instruction:

```
pid1=fork();
```

Ensuite le processus père crée un deuxième fils « pid2 », instruction :

```
if ( pid1!=0 )  // si je suis le processus père
        pid2=fork();
```

Donc l'arbre généré est le suivant:

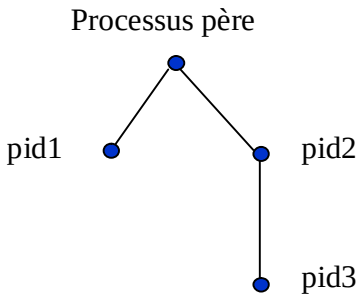


2. Résultats d'exécutions possibles

- Affiche « signal recu : 10 » et se termine normalement
- N'affiche rien et se termine normalement : en effet il se peut que le fils « pid1 » reçoit le signal avant d'installer le nouveau handler du signal « SIGUSR1 » (avant d'exécuter la fonction signal).
- Affiche « signal recu : 10 » et se bloc : en effet il se peut que le fils « pid1 » installe le nouveau handler du signal « SIGUSR1 » mais reçoit le signal avant de faire « pause() ». Dans ce cas, le nouveau handler est exécuté et ensuite on fait « pause() » après avoir reçu le signal.

Exercice 7 :

1. *Ecrire un programme C qui permet de générer l'arbre de processus représenté par la figure ci-dessous où le processus père crée deux processus fils « pid1 » et « pid2 »: il crée tout d'abord le processus « pid1 » ensuite, il crée le processus « pid2 ». Le processus « pid2 » crée le processus « pid3 ». Chaque processus doit attendre la terminaison de ses fils avant de se terminer.*



2. *Modifier le programme précédent afin que chaque processus affiche son identifiant dans l'ordre suivant :*

- *Le processus « pid3 » affiche avant le processus « pid2 ».*
- *Le processus « pid2 » affiche avant le processus père.*
- *Pas de contrainte d'affichage pour le processus « pid1 ».*

3. *Modifier le programme précédent afin que chaque processus affiche son identifiant dans l'ordre suivant :*

- *Le processus « pid3 » affiche avant le processus « pid2 ».*
- *Le processus « pid2 » affiche avant le processus « pid1 ».*
- *Le processus « pid1 » affiche avant le processus père.*

Réponses:

1.

```
….
main ( ) {
   pid_t pid1,pid2,pid3,pid4;
   pid1=fork(); // le processus père crée le fils « pid1 ».
  if (pid1!=0) { // si je suis le processus père
      pid2=fork();  // le processus père crée le fils « pid2 »
      if (pid2==0) { // si je suis le processus « pid2 »
```

```
            pid3=fork();  // le processus « pid2 » crée le fils « pid3 »
          }
        }
        // chaque processus attend la terminaison de ses fils avant de se terminer
        while (wait(0)!=-1);
      }
```

2. On propose une synchronisation basée sur la terminaison des processus:

- Le processus «pid2» attend la terminaison de son fils «pid3» avant d'afficher.
- Le processus père attend la terminaison de son fils «pid2» avant d'afficher.
- Pas de contrainte d'affichage pour le processus « pid1 ».

```
….
main ( ) {
  pid_t pid1,pid2,pid3,pid4;
  pid1=fork(); // le processus père crée le fils « pid1 »
  if (pid1!=0) { // si je suis le processus père
    pid2=fork();  // le processus père crée le fils « pid2 »
    if (pid2==0) { // si je suis le processus « pid2 »
      pid3=fork();  // le processus « pid2 » crée le fils « pid3 »
      if (pid3==0) { // si je suis le processus « pid3 »
         printf("pid3 =  %d    pere = %d   \n", getpid(),getppid());
      }
      else { // si je suis le processus « pid2 ».
       // « pid2 » attend le terminaison de son fils « pid3 » puis il affiche
        wait(0);
        printf("pid2 =  %d    pere = %d   \n", getpid(),getppid());
      }
    }
    else { // si je suis le processus père
       // le processus père attend la terminaison de « pid2 » puis il affiche
      waitpid(pid2,0,0);
      printf("processus pere =  %d    pere = %d   \n", getpid(),getppid());
    }
  }
  else { // si je suis le processus « pid1 »
     // pas de contrainte pour le fils « pid1 »
```

```
        printf("pid1 =  %d    pere = %d   \n", getpid(),getppid());
      }
    }
```

3. Ordre d'affichage:

- Le processus « pid3 » affiche avant le processus « pid2 ».
- Le processus « pid2 » affiche avant le processus « pid1 ».
- Le processus « pid1 » affiche avant le processus père.

Le processus pid2:

- attend la terminaison de son fils « pid3 »
- affiche
- envoie le signal « SIGUSR1 » à « pid1 » pour l'informer qu'il a affiché. Il faut noter que « pid2 » connait l'identifiant de « pid1 » car il est créé aprè «pid1».

Le processus pid1:

- installe un handler pour le signal « SIGUSR1 ».
- attend l'arrivé du signal. Pour éviter le blocage (il se peut que le signal arrive avant de faire « pause() ») on utilise une attente active (on utilise une variable qui change de valeur si le signal est arrivé).
- affiche

Le processus père:

- attend la terminaison de « pid1 »
- affiche

Code C

```
int t=0;
void hand(int sig) { t=1;};
main ( ) {
    ….
      else { // si je suis le processus pid2
        sleepe(1) ; // laisse le temps au processus «pid1» d'installer le handler
        wait(0);  // attend la terminaison de son fils «pid3» avant d'afficher
        printf("pid2 =  %d    pere = %d   \n", getpid(),getppid());
        // après l'affichage, il envoie un signal à « pid1 »
        kill(pid1, SIGUSR1);
      }
    }
```

```
        else { // si je suis le processus père
            // attend la terminaison de son fils « pid1 » avant d'afficher.
            waitpid(pid1,0,0);
            printf("processus pere =  %d    pere = %d   \n", getpid(),getppid());
        }
    }
    else { // le processus « pid1 »
        signal(SIGUSR1,hand); //installe le handler «hand» pour « SIGUSR1 ».
        sleep(5);
        while (t==0); // attente active à la place de pause() afin d'éviter le blocage
        printf("pid1 =  %d    pere = %d   \n", getpid(),getppid());
    }
}
```

Exercice 8:

Considérons les deux programmes ci-dessous (Programmes 1 et 2).

Programme 1

```
main() {
    pid_t  pid1, pid2, pid3;
    pid1=fork();
    if (pid1==0) {
        pid2=fork();
        if (pid2 !=0) {
            pid3=fork();
        }
    }
    }
```

Programme 2

```
main() {
    pid_t  pid1, pid2, pid3;
    pid1=fork();
    if (pid1==0) {
        pid2=fork();
        if (pid2 !=0) {
            pid3=fork();
        }
    }
pid4=fork() ;
}
```

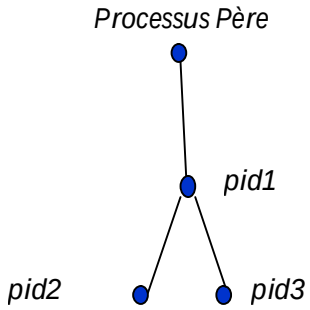
1. *Donnez l'arbre de processus généré par le programme 1. Sur chaque nœud de l'arbre, if faut noter les identifiants des processus générés (processus père, pid1, pid2, pid3).*
2. *Modifier le programme 1 afin que le processus « pid1 » se termine après ces deux fils et après l'exécution de toutes les instructions de son père.*
3. *Donnez l'arbre de processus généré par le programme 2. Sur chaque nœud de l'arbre, if faut noter les identifiants des processus générés (processus père, pid1, pid2, pid3, pid4).*

Réponses :

1.

```
        pid1=fork();   // le processus père crée le fils « pid1 »
        if (pid1==0) {    // si je suis le processus « pid1 »
                pid2=fork();   // le processus « pid1 » crée le fils « pid2 »
                if (pid2 !=0) {  // si je suis le processus « pid1 »
                        pid3=fork();   // le processus « pid1 » crée le fils « pid3 »
                }
        }
}
```

Donc l'arbre de processus générés est le suivant :



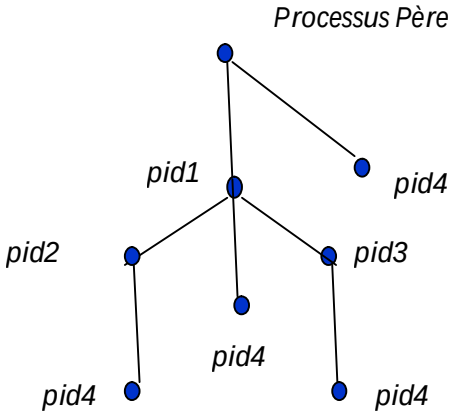
2. Principe :

- « pid1 » fait « wait(0) » pour attendre la terminaison de ses fils et attend un signal du père avant de se terminer.
- Le père, avant de se terminer, envoie un signal au processus « pid1 ».

```
…
int t=0;
void hand( int sig ){
   t=1;
}
main() {
  pid_t  pid1, pid2, pid3;
  pid1=fork(); // le processus père crée le fils « pid1 »
  if (pid1!=0) { // si je suis le processus père
     sleep(1) ; // laisser le temps à « pid1 » d'installer le handler pour « SIGUSR1 »
     kill(pid1,SIGUSR1); // le père envoie le signal « SIGUSR1 » et se termine
  }
```

```
  else { // si je suis le processus « pid1 »
    signal(SIGUSR1,hand) ;   // installe le handler « hand » pour « SIGUSR1 »
    pid2=fork(); // le processus « pid1 » crée le fils « pid2 »
    if (pid2 !=0) { // si je suis le processus « pid1 »
      pid3=fork(); // // le processus « pid1 » crée le fils « pid3 »
      if (pid3!=0) { // si je suis le processus « pid1 »
         while(wait(0) !=-1) ; // « pid1 » attend la terminaison de tous ses fils
         while(t==0) ; // « pid1 » attend l'arrivé du signal.
         // en effet lorsque le signal arrive, la valeur de t sera changée.
         // N.B. On utilise l'attente active à la place de « pause() » pour éviter le
        // blocage car il se peut que le signal arrive avant que « pid1 » fasse pause().
         printf(" pid1:  termine  :  \n" );
      }
      else {
         printf(" pid3:  termine:  \n" );
      }
    }
    else {
      printf(" pid2:  termine:  \n" );
    }
  }
}
```

3. L'arbre ci-dessous est celui généré par le programme 2. En effet, le programme 2 diffère du programme 1 par la dernière instruction (« pid4=fork(); ») qui sera exécutée par tous les processus générés par le programme 1, donc chaque processus de l'arbre précédent crée un fils d'identifiant « pid4 » (la valeur de « pid4 » est différente dans chaque processus).



Exercice 9 :

Considérons le programme C suivant:

```
…
main(){
        pid_t pid1, pid2 ;
        pid1=fork() ;
        if (pid1==0)
                pid2=fork() ;
        while (1) {
                printf(" %d  \n",getpid());
        }
}
```

1. *Donner l'arbre de processus générés par le programme C ci-dessus.*

2. *Quel est le résultat d'exécution de ce programme.*

3. *Modifier le programme C précédent afin que :*

- *Le processus « pid2 »*
  - *s'exécute pendant 5 secondes,*
  - *ensuite, il envoie le signale « SIGINT » au processus « pid1 »,*
  - *et enfin, il affiche «Arret de pid2 » et se termine.*
- *Le processus « pid1 », après réception du signal « SIGINT »,*
  - *il affiche «Arret de pid1 »,*
  - *ensuite, il envoie le signal « SIGNINT » au processus « père » et se termine* ***(ne pas utiliser la fonction « exit() »)***.
- *Le processus père, après réception du signal « SIGINT »,*
  - *il affiche « Arret du processus père » et se termine.*

Réponses :

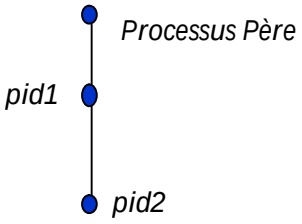
1. Le processus père crée le fils « pid1 » grâce à l'instruction :

```
pid1=fork() ;
```

Ensuite, le processus « pid1 » crée un fils « pid2 » grâce à l'instruction :

```
if (pid1==0) // si je suis le processus « pid1 »
        pid2=fork() ; // exécuté par « pid1 » qui crée « pid2 »
```

Donc l'arbre généré est le suivant



2. Le programme boucle : En effet l'instruction :

```
while (1) {
    printf(" %d  \n",getpid());
}
```

est exécutée par tous les processus générés. Chaque processus affiche indéfiniment son identifiant.

Remarque : Les affichages sont entrelacés entre les différents processus.

3. Le principe est d'utiliser une temporisation de 5 secondes.

Solution avec « SIGINT »

Processus « pid2 » :

Dans le processus « pid2 » on appelle la fonction « alarm(5) » qui bloque le processus « pid2 » pendant 5 secondes ensuite il sera averti de la fin de la temporisation (après 5 secondes) par le signal « SIGALRM ». Puisque le traitement par défaut du signal « SIGALRM » est la terminaison du processus, alors on modifie son comportement par l'installation d'un nouveau handler « fonc_hand2 () ». Une fois la temporisation est terminée, le processus « pid2 » exécute « fonc_hand2 () » qui consiste à :

- envoyer le signale « SIGINT » à son père « pid1 ».
- afficher le message « Arret de pid2 »
- puis envoyer le signal « SIGINT » à lui-même pour se terminer.

```
// handler à installer par le processus « pid2 » pour le signal « SIGALRM »
void fonc_hand2 (int s) {
    kill(getppid(), SIGINT);
    printf("Arret de pid2 \n ");
    kill(getpid(), SIGINT); // raise(SIGINT);
}
```

Processus « pid1 » :

Le processus « pid1 » installe un nouveau handler pour le signal « SIGINT ».
Après réception du signal « SIGINT » :

- il affiche le message «Arret de pid1 »,
- ensuite, il envoie le signal « SIGNINT » au processus « père »
- puis envoie le signal « SIGINT » à lui-même pour se terminer.

```
// handler à installer par le processus « pid1 » pour le signal « SIGINT »
```

```
void fonc_hand1 (int s) {
    printf("Arret de pid1 \n ");
    kill(getppid(), SIGINT);
    // Il faut réinstaller le comportement par défaut du signal « SIGINT » afin
    // que le processus « pid1 »  puisse utiliser le signal « SIGINT » pour se
    // terminer.
    signal(SIGINT,SIG_DFL);  // rétablir le comportement par défaut
    kill(getpid(),SIGINT); // raise(SIGINT);
}
```

Processus père :

Le processus père installe un nouveau handler pour le signal « SIGINT ».
Après réception du signal « SIGINT »:

- il affiche le message « Arret du processus père ».
- ensuite, il envoie le signal « SIGINT » à lui-même pour se terminer.

```
// handler à installer par le processus père pour le signal « SIGINT »
void fonc_hand (int s) {
    printf("Arret du processus père \n ");
    // Il faut réinstaller le comportement par défaut du signal « SIGINT » afin
    // que le processus père puisse l'utiliser pour se terminer.
    signal(SIGINT,SIG_DFL); // rétablir le comportement par défaut
    kill(getpid(),SIGINT); // raise(SIGINT);
}
```

Code C complet

Définition des handlers

```
void fonc_hand2 (int);
void fonc_hand1 (int);
void fonc_hand (int );
main(){
    pid_t  pid1, pid2;
    pid1=fork();  // le processus père crée le fils « pid1 »
    if (pid1==0) { // si je suis le processus « pid1 »
        pid2=fork();  // le processus « pid1 » crée le fils « pid2 »
        if (pid2==0) { // si je suis le processus « pid2 »
            // installe le handler « fonc_hand2() » pour le signal «SIGALRM»
```

```
                signal(SIGALRM, fonc_hand2);
                // Appel de « alarm(5) » pour une temporisation de 5 secondes
                alarm(5);
                while (1) {
                    printf(" %d  \n",getpid());
                }
            }
            else { // si je suis le processus « pid1 »
                // installe le handler « fonc_han1() » pour le signal «SIGINT»
                    signal(SIGINT, fonc_hand1);
                    while (1) {
                        printf(" %d  \n",getpid());
                        }
            }
        }
        else { // si je suis le processus père
            // installe le handler « fonc_han() » pour le signal «SIGINT»
                signal(SIGINT, fonc_hand);
                while (1) {
                    printf(" %d  \n",getpid());
                }
        }
    }
```

Exercice 10 :

Soient « pid1 » et « pid2 » les identifiants de deux processus fils créés par le processus père. On suppose que le code du processus « pid1 » est composés de deux parties : la première partie consiste en l'exécution de la fonction « f1() » et la deuxième partie consiste en l'exécution de la fonction « g1() ». De même on suppose que le code du processus « pid2 » est composés de deux parties : la première partie consiste en l'exécution de la fonction « f2() » et la deuxième partie consiste en l'exécution de la fonction « g2() ». On suppose que les fonctions « f1() », « g1() », « f2() » et « g2() » sont des fonctions données.

***1.** Ecrire un programme C dans lequel le processus père crée deux processus*

fils qui ont pour identifiants « pid1 » et « pid2 ». Chaque processus fils exécute son code. Le processus père doit attendre la terminaison de ses fils avant de se terminer.

2. *Modifier le programme précédent afin que chaque processus fils connaisse l'identifiant de l'autre processus fils.*

3. *Pour simplifier on suppose que chaque processus fils connait l'identifiant de l'autre processus fils. Modifier le programme de* ***la question 1.*** *afin de synchroniser l'exécution des processus « pid1 » et « pid2 » : Chaque processus ne peut commencer l'exécution de la deuxième partie de son code (exécution de « g1() » ou g2() ») qu'après la fin d'exécution de sa première partie ainsi que l'exécution de la première partie de l'autre processus (exécutions des fonctions « f1() » et f2() »).*

Réponses :

1.

```
// Définition des fonctions f1(), g1(), f2() et g2();
   main ( ) {
    pid_t pid1, pid2;
        pid1=fork(); // le processus père crée le fils « pid1 »
        if (pid1!=0) {   // si je suis le processus père
            pid2=fork();  // le processus père crée le deuxième fils « pid2 »
            if (pid2==0) {   // si je suis le processus « pid2 »
                f2();   // appel des fonctions f2() et g2()
                g2();
            }
        }
        else {  // si je suis le processus « pid1 »
            f1();   // appel des fonctions f1() et g1()
            g1();
        }
        while (wait(0)!=-1);
   }
```

2. Puisque « pid1 » est créé avant « pid2 » alors le processus « pid2 » connait l'identifiant du processus « pid1 ». Par contre « pid1 » ne connait pas l'identifiant de « pid2 », dans ce cas on utilise les pipes pour que « pid2 » envoie son identifiant au processus « pid1 »

```
…
main ( ) {
    …
        if (pid2==0) {   // si je suis le processus « pid2 »
            int nb_ecrits;
            pid_t  pid=getpid(); // dans la variable pid on sauvegarde la valeur
                                 // de « pid2 »
            close(p[0]);
            // on écrit dans le tube la valeur de « pid2 » (la variable p)
            nb_ecrits = write(p[1], &pid, 1*sizeof(int));
            f2();
            g2();
        }
    }
    else {  // si je suis le processus « pid1 »
        int nb_lus;
        pid_t pid2;
        close(p[1]);
        // « pid1 »  sauvegarde l'identifiant de « pid2 »  dans la variable pid2.
        nb_lus=read(p[0], &pid2, 1*sizeof(int));
        f1();
        g1();
        printf (" pid1=  %d   pid2 = %d \n",getpid(), pid2);
    }
    while (wait(0)!=-1);
}
```

3.

Version 1

Dans cette version on ne tient pas compte du problème de blocage. Chaque processus ne peut commencer l'exécution de sa deuxième partie qu'après la fin d'exécution des premières parties des deux processus.

Principe:

Processus pid1

- installe un handler pour le signal « SIGUSR1 »

- exécute la fonction f1()
- envoie le signal « SIGUSR2 » à « pid2 »
- attend le signal « SIGUSR1 »
- exécute la fonction g1()

Processus pid2

- installe un handler pour le signal « SIGUSR2 »
- exécute la fonction f2()
- envoie le signal « SIGUSR1 » à « pid1 »
- attend le signal « SIGUSR2 »
- exécute la fonction g2()

```
...
void hand_pid1( int sig) {   } ;
void hand_pid2( int sig) {   } ;
// Définition des fonctions f1(), g1(), f2() et g2();
main ( ) {
    pid_t pid1, pid2;
    int p[2];
    pipe(p);
    pid1=fork();  // le processus père crée le fils « pid1 »
    if (pid1!=0) {   // si je suis le processus père
        pid2=fork();  // le processus père crée le fils « pid2 »
            if (pid2==0) {   // si je suis le processus « pid2 »
            int nb_ecrits;
            // installe un handler pour le signal « SIGUSR2 »
            signal(SIGUSR2, hand_pid2 ) ;
            pid_t pid=getpid();
            close(p[0]);
            nb_ecrits = write(p[1], &pid, 1*sizeof(int));
            sleep(2);
            f2();
            // barrière de synchronisation
            kill(pid1,SIGUSR1);
            pause();
            g2();
        }
    }
```

```
        else {  // si je suis le processus « pid1 »
            int nb_lus;
            signal(SIGUSR1, hand_pid1 ) ;
            pid_t pid2;
            close(p[1]);
            nb_lus=read(p[0], &pid2, 1*sizeof(int));
            f1();
            // barrière de synchronisation
            kill(pid2,SIGUSR2);
            pause();
            g1();
        }
        while (wait(0)!=-1);
    }
```

Version 2:

Dans la version 1, il y a risque de blocage. Ilse peut que le signal arrive avant que le processeur fasse « pause(). Dans cette version on remplace pause() par une attente active. Soient v1 et V2 deux variables initialisée à 1. 0 la réception du signal, leurs valeurs deviennent nulles.

Principe:

Processus « pid1 »

- installe un handler pour le signal « SIGUSR1 ». Une fois le signal est arrivé on change la valeur d'une variable v1 (v1=0).
- exécute la fonction f1()
- envoie du signal « SIGUSR2 » à « pid2 »
- attente active: tant que le signal « SIGUSR1 » n'est pas encore arrivé (while (v1 !=0))
- exécute la fonction g1()

Processus « pid2 »

- installe un handler pour le signal « SIGUSR2 ». Une fois le signal est arrivé on change la valeur d'une variable v2 (v2=0).
- exécute la fonction f2()
- envoie du signal « SIGUSR1 » à « pid1 »
- attente active: tant que le signal « SIGUSR2 » n'est pas encore arrivé (while (v2 !=0))
- exécute la fonction g2()

…

```
int v1=1, v2=1;
void hand_pid1( int sig) {
    v1=0;
} ;
void hand_pid2( int sig) {
    v2=0;
} ;
// Définition des fonctions f1(), g1(), f2() et g2();
main ( ) {
…
        if (pid2==0) {   // si je suis le processus « pid2 »
            signal(SIGUSR2, hand_pid2 ) ;
            …..
            f2();
            // barrière de synchronisation
            kill(pid1,SIGUSR1);
            while  (v2) ; // attente active
            g2();
        }
    }
    else {  // si je suis le processus « pid1 »
            signal(SIGUSR1, hand_pid1 ) ;
            ….
            f1();
            // barrière de synchronisation
            kill(pid2,SIGUSR2);
            while (v1);  // attente active
            g1();
    }
    while (wait(0)!=-1);
}
```

Chapitre 3 : Les tubes

1. Introduction

Un tube de communication est un canal ou tuyau (en anglais pipe) dans lequel un processus peut écrire des données et un autre processus peut les lire.

- C'est un moyen de communication unidirectionnel inter-processus.
- Pour avoir une communication bidirectionnelle entre deux processus, il faut créer deux tubes et les employer dans des sens opposés.

Il existe 2 types de tubes:

- Les tubes ordinaires (anonymes): ne permettent que la communication entre processus issus de la même application (entre père et fils, ou entre frères).
- Les tubes nommés: permettent la communication entre processus qui sont issus des applications différentes.

2. Caractéristiques

- Un tube possède deux extrémités, une est utilisée pour la lecture et l'autre pour l'écriture.
- La gestion des tubes se fait en mode FIFO (First In First Out: premier entré premier sorti). Les données sont lues dans l'ordre dans lequel elles ont été écrites dans le tube.
- La lecture dans un tube est destructive. C'est-à-dire que les données écrites dans le tube sont destinées à un seul processus et ne sont donc lues qu'une seule fois. Une fois une donnée est lue elle sera supprimée.
- Plusieurs processus peuvent lire dans le même tube mais ils ne liront pas les mêmes données, car la lecture est destructive.
- Un tube a une capacité finie.

3. Les Tubes anonymes

3.1. Création d'un tube

Le tube est créé par appel de la primitive « pipe() », déclaré dans « unistd.h ». La création d'un tube correspond à celle de deux descripteurs de fichiers,

- l'un permet de lire dans le tube.
- l'autre permet d'écrire dans le tube.

Syntaxe:

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

En cas de succès, le tableau «p» est remplit par les descripteurs des 2 extrémités du tube. Par définition:

- Le descripteur d'indice 0 (p[0]) désigne la sortie du tube, il est ouvert en lecture seule.
- Le descripteur d'indice 1 (p[1]) désigne l'entrée du tube, il est ouvert en écriture seule.

3.2. Fermeture d'un descripteur

On doit fermer les descripteurs non utilisés la primitive «close(int fd)» où « fd » désigne le numéro (égal à 0 ou 1) du descripteur à fermer.

3.3. Écriture dans un tube

L'écriture dans un tube se fait grâce à l'appel de la primitive «write()» en utilisant le descripteur p[1].

Syntaxe :

```
int write (int p[1],  void *zone, int nb_car);
```

- « zone » : un pointeur sur la zone mémoire des données à écrire dans le tube.
- « nb_car »: le nombre d'octets à écrire dans le tube.

Code retour:

- en cas de succès, elle retourne le nombre d'octets effectivement écrits.
- en cas d'erreur, elle retourne -1.

Ecriture dans un tube fermé en lecture

Lorsqu'un processus tente d'écrire dans un tube sans lecteur (descripteur en lecture est fermé), alors il reçoit le signal « SIGPIPE ».

3.4. Lecture dans un tube

La lecture dans un tube se fait grâce à l'appel de la primitive « read()» en utilisant le descripteur p[0].

Syntaxe:

```
int read (int p[0],  void *zone, int nb_car);
```

- « zone » : un pointeur sur une zone mémoire dans laquelle les données, seront écrites, après lecture.
- « nb_car »: désigne le nombre d'octets que l'on souhaite lire à partir du tube.

Code retour:

- en cas de succès, elle retourne le nombre d'octets effectivement lus.
- en cas d'erreur, elle retourne -1.

4. Les tubes nommés (named pipe)

Les tubes nommés (appelés *fifo)* permettent la communication, en mode fifo, entre processus quelconques (sans lien de parenté).

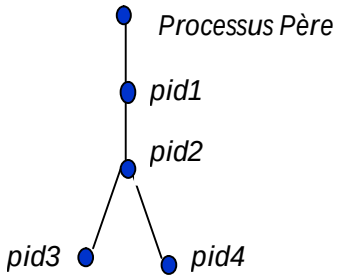
- Un tube nommé possède un nom dans le système de fichier, et apparait comme les fichiers permanents. Pour l'utiliser, il suffit qu'un processus l'appelle par son nom.
- Un tube nommé est utilisé comme un fichier normal: on peut utiliser les primitives : « open() », « close() », « read() », « write() ».
- La création d'un tube nommé, à partir d'un programme C, se fait grâce à l'appel de la primitive « mkfifo() » déclarée dans <sys/stat.h>.
- La création peut se faire sur la ligne de commande en utilisant les commandes « mkfifo » (/usr/bin/mkfifo) ou « mknod » (/bin/mknod).

Exemple:

```
% mknod  tube1  p          // crée le tube nommé "tube1" où p désigne "pipe"
% mkfifo tube2              // crée le tube nommé "tube2"
```

Exercice 1 :

1. *Ecrire un programme C qui permet de générer l'arbre de processus représenté par la figure ci-dessous. Le processus père crée le processus fils « pid1 ». Le processus fils « pid1 » crée le processus fils « pid2 ». Le processus « pid2 » crée deux processus fils: tout d'abord il crée le fils « pid3 » et ensuite il crée le processus « pid4 ».*



2. *On suppose que le processus « pid4 » affiche la lettre 'a' et « pid3 » affiche la lettre 'b'. Modifier le programme précédent afin que la lettre 'a' soit affichée avant la lettre 'b'.*
3. *Modifier le programme précédent afin que la lettre 'b' soit affichée avant la lettre 'a'.*

Réponses :

1. Le processus père crée le processus « pid1 », instruction :

```
pid1=fork();
```

Ensuite, le processus « pid1 » crée le processus « pid2 », instruction :

```
if (pid1==0)  // si je suis le processus « pid1 »
   pid2=fork(); // le processus « pid1 » crée le processus « pid2 »
```

Après sa création, le processus « pid2 » crée à son tour les processus « pid3 » et « pid4 », instructions :

```
if (pid2==0) { // si je suis le processus « pid2 »
      pid3=fork();  // le processus « pid2 » crée le processus « pid3 »
      if (pid3!=0) { // si je suis le processus « pid2 »
            pid4=fork();  // le processus « pid2 » crée « pid4 »
      }
}
```

Ou

```
if (pid2==0) {
```

```
            (pid3=fork()) && (pid4=fork());
        }
```

Le code C :

```
main ( ) {
    pid_t pid1, pid2, pid3, pid4;
    pid1=fork();     // le processus père crée le processus fils « pid1 »
    if (pid1==0) {   // si je suis le processus « pid1 »
       pid2=fork();  // exécuté par « pid1 » qui crée le processus fils « pid2 »
       if (pid2==0) { // si je suis le processus « pid2 »
            pid3=fork(); // exécuté par « pid2 » qui crée le fils « pid3 »
            if (pid3!=0) { // si je ne suis pas « pid3 » (si je suis « pid2 »)
                 pid4=fork(); // exécuté par «pid2» qui crée le fils «pid4»
            }
       }
    }
    while (wait(0)!=-1);
}
Ou
main ( ) {
       …
       if (pid2==0) { // si je suis le processus « pid2 »
            (pid3=fork() ) && (pid4=fork());
            // arbre binaire de racine « pid2 »
    }
    while (wait(0)!=-1);
}
```

2. Principe: Après l'affichage de la lettre 'a', le processus « pid4 » informe le processus « pid3 » qu'il a déjà affiché en lui envoyant un signal

Remarque : Puisque le processus « pid3 » est créé avant le processus « pid4 », donc le processus « pid4 » connait l'identifiant du processus « pid3 ».

Version 1

- Le processus « pid4 » affiche tout d'abord la lettre 'a' ensuite, il envoie le signal « SIGUSR1 » au processus « pid3 » avant de se terminer. Instructions :

```
fputc('a',stdout) ;
sleep(1);
kill(pid3,SIGUSR1);
```

- Le processus « pid3 », installe tout d'abord un nouveau handler pour le signal « SIGUSR1 », ensuite attend le signal. Une fois le signal est reçu, « pid3 » affiche la lettre 'b' et se termine. Instructions :

```
signal(SIGUSR1,hand);
pause();
fputc('b',stdout);
```

Le code C complet:

```
void hand(int signum) {   }
main ( ) {
…
                    if (pid4==0) {  // le processus « pid4 »
                       fputc('a',stdout);  //affiche la lettre 'a'
                       sleep(1); // laisser le temps à « pid3 » de faire pause()
                       // « pid4 » connait l'identifiant de « pid3 » car « pid3 »
                       // est créé avant «pid4»,
                       kill(pid3,SIGUSR1);
                     }
                }
                else { // le processus « pid3 »
                    signal(SIGUSR1,hand); // installe un handler pour SIGUSR1
                    pause();  // attend un signal
                    fputc('b',stdout);  //affiche la lettre 'b'
               }
          }
     }
     while (wait(0)!=-1);
}
```

Version 2

Dans la version précédente, il y a risque de blocage. Il se peut que le signal arrive avant de faire « pause() » (entre les appels « signal » et « pause() »).
Dans cette version on donne une solution sans blocage. On introduit une variable qui sera utilisée pour tester si le signal est déjà arrivé. Cette variable est initialisé et change de valeur lorsque le signal arrive.

```
…..
int v=1 ;  // cette variable utilisée pour vérifier si le signal est arrivé ou non
void hand(int signum) {
    v=0;  // si le signal est arrivé on modifie la valeur de la variable v.
}
main ( ) {
    …
            if (pid4==0) {  // si je suis le processus « pid 4 »
               fputc('a',stdout);   //affiche la lettre 'a'
               sleep(1);  // laisser le temps à « pid3 » de faire pause();
               // « pid4 » connait l'identifiant de « pid3 » car « pid3 »
               // est créé avant «pid4»,
               kill(pid3,SIGUSR1); // «pid4» envoie «SIGUSR1» à «pid3»
            }
        }
            else { // si je suis le processus « pid3 »
               signal(SIGUSR1,hand);
               while(v);  // attente active: tant que le signal n'est pas arrivé.
               fputc('b',stdout);  //affiche la lettre 'b'
            }
      }
    }
    while (wait(0)!=-1);
}
```

3. Puisque le processus « pid4 » est créé après le processus « pid3 » alors le processus « pid3 » ne connait pas l'identifiant du processus « pid4 ». Avant de synchroniser, le processus « pid4 » doit envoyer son identifiant au processus « pid3 » en utilisant les tubes (pipe).

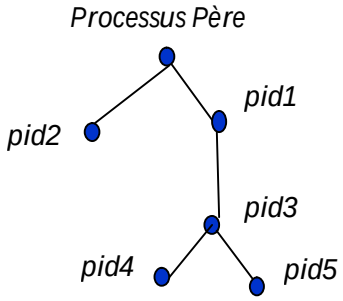
```
void hand(int signum) {    }
main ( ) {
…
                if (pid4==0) {      // si je suis le processus «pid4»
                    int nb_ecrits;
                    // dans la variable pid on sauvegarde l'identifiant de «pid4»
                    pid_t   pid=getpid();
                    signal(SIGUSR1,hand);
                    close(p[0]);
                    // « pid4 » écrit son identifiant dans le tube « p ».
                    nb_ecrits = write(p[1], &pid, 1*sizeof(int));
                    pause(); // attend le signal
                    fputc('a',stdout);
                }
            }
            else { // si je suis le processus « pid3 »
                    int nb_lus;
                    fputc('b',stdout);
                    sleep(1); // laisser le temps à «pid4» de faire «pause()»
                    close(p[1]);
                    // «pid3» lit l'identifiant de «pid4» dans le tube « p » et
                    // le sauvegarde dans la variable «pid4».
                    nb_lus=read(p[0], &pid4, 1*sizeof(int));
                    kill(pid4,SIGUSR1);
                }
            }
    }
while (wait(0)!=-1);
}
```

Remarque :

Dans la version 1, il y a risque de blocage. Il se peut que le signal arrive avant de faire « pause() » (entre les appels « signal » et « pause() »). Dans ce cas on peut appliquer une attente active (la solution de la version 2 de la question2.

Exercice 2 :

1. *Ecrire un programme C qui permet de générer l'arbre de processus représenté par la figure ci-dessous. Le processus père crée deux processus fils « pid1 » et « pid2 ». Le processus « pid1 » crée le processus « pid3 », ensuite le processus « pid3 » crée 2 processus fils «pid4 » et « pid5 ».*



2. *Modifier le programme précédent afin que « pid2 » attende la terminaison de « pid5 » avant de se terminer.*

a. *Dans le cas où le processus « pid2 » est créé avant le processus « pid1 ».*

b. *Dans le cas où le processus « pid1 » est créé avant le processus « pid2 ».*

Réponse :

1.

```
#include<stdio.h>
#include<unistd.h>
main(){
    pid_t pid1, pid2, pid3, pid4, pid5; // les identifiants des processus.
    pid1=fork(); // le processus père crée le processus fils «pid1 »
    if (pid1 !=0) { // si je suis le processus père
        pid2=fork( ) ; // le processus père crée le processus fils «pid2 »
    }
    else { // si je suis le processus «pid1 »
       pid3=fork(); // le processus «pid1 » crée le processus fils «pid3 »
       if (pid3==0) { // si je suis le processus «pid3 »
          pid4=fork() ; // le processus «pid3 » crée le processus fils «pid4 »
          if (pid4 !=0) { // si je suis le processus «pid3 »
             pid5=fork(); // le processus «pid3 » crée le processus fils «pid5 »
          }
```

```
        }
      }
      while(wait(0) !=-1) ;
}
```

2.

2.a: Puisque « pid2 » est créé avant « pid1 » dans ce cas l'identifiant du processus « pid2 » est connu dans le processus « pid1 » et puisque « pid5 » est un descendant du processus « pid1 » donc « pid5 » connait l'identifiant du processus « pid2 ».

Principe :

- « pid5 » envoie un signal à « pid2 et se termine.
- « pid2 » attend le signal avant de se terminer.

```
…
void hand( int sig ){ }
main(){
  pid_t pid1, pid2, pid3, pid4, pid5; // les identifiants des processus.
  pid2=fork();    // le processus père crée tout d'abord le processus fils «pid2 »
  if (pid2==0) {  // si je suis le processus «pid2 »
   signal(SIGUSR1,hand); // «pid2 » installe un handler pour le signal «SIGUSR1»
    pause(); // attend l'arrivé d'un signal
  }
  else {   // si je suis le processus père
   pid1=fork( ) ; // le processus père crée le processus fils «pid1 »
   if (pid1==0) { // si je suis le processus «pid1 »
     pid3=fork(); // le processus «pid1 » crée le fils «pid3 »
     if (pid3==0) { // si je suis le processus fils «pid3 »
        pid4=fork() ; // le processus «pid3 » crée le processus fils «pid4 »
        if (pid4 !=0) { // si je suis le processus «pid3 »
          pid5=fork(); // le processus «pid3 » crée le processus fils «pid5 »
          if (pid5==0) {  // si je suis le processus « pid5 »
            sleep(1);   // laisser le temps au processus pid2 de faire pause().
            kill(pid2,SIGUSR1);  // «pid5 » envoi le signal «SIGUSR1 » à «pid2 »
           }
        }
     }
   }
```

```
  }
  while(wait(0) !=-1) ;
}
```

2.b: Puisque « pid1 » est créé avant « pid2 » et que « pid5 » est un descendant du processus « pid1 », dans ce cas l'identifiant du processus « pid2 » n'est pas connu dans le processus « pid5 ».

Pour que « pid5 » puisse envoyer un signal à « pid2 », ce dernier doit envoyer son identifiant à « pid5 » en utilisant les tubes. Le processus « pid2 » écrit son identifiant dans le tube et le processus « pid5 » lit l'identifiant de « pid2 » à partir du même tube.

Principe :

- Processus « pid5 »
  - Lit l'identifiant de « pid2 » dans un tube.
  - Envoie un signal à « pid2 et se termine.
- Processus « pid2 »
  - Ecrit son identifiant dans un tube
  - Attend un signal avant de se terminer.

```
void hand( int sig ){  }
main(){
  pid_t pid1, pid2, pid3, pid4, pid5; // les identifiants des processus.
  int p[2];  // déclaration d'un tube
  int nb_ecrits, nb_lus;
  pipe(p);  // création du tube p
  pid1=fork(); // le processus père crée le processus fils «pid1 »
  if (pid1 !=0) { // si je suis le processus père
    pid2=fork( ) ; // le processus père crée le processus fils «pid2 »
    if (pid2==0) { // si je suis le processus «pid2 »
       pid_t  pid; // on sauvegarde dans la variable pid l'identifiant de «pid2 »
       signal(SIGUSR1,hand); // «pid2 » installe un handler pour «SIGUSR1»
       pid_t pid=getpid();  // «pid2 » sauvegarde son identifiant dans la variable pid
       close(p[0]);
       // «pid2 » écrit son identifiant dans le tube p
       nb_ecrits=write(p[1], &pid,1*sizeof(int));
        pause();  // «pid2 » attend un signal
    }
```

```
  }
  else { // si je suis le processus «pid1 »
    pid3=fork(); // le processus «pid1 » crée le processus fils «pid3 »
    if (pid3==0) { // si je suis le processus fils «pid3 »
      pid4=fork() ; // le processus «pid3 » crée le processus fils «pid4 »
      if (pid4 !=0) { // si je suis le processus «pid3 »
        pid5=fork(); // le processus «pid3 » crée le processus fils «pid5 »
        if (pid5==0) {   // si je suis le processus «pid5 »
              close(p[1]);
              // « pid5 » lit dans le tube p l'identifiant de « pid2 » et le
              // sauvegardé dans la variable pid
              nb_lus=read(p[0], &pid, sizeof(int));
              kill(pid,SIGUSR1);  // «pid5» envoie le signal « SIGUSR1 » à «pid2»
        }
      }
    }
  }
  while(wait(0) !=-1) ;
}
```

Exercice 3:

1. *Ecrire un programme dans lequel le processus père crée 3 processus fils :*

- *le premier processus créé, nommé «pid1», affiche le message «je suis pid1»,*
- *le deuxième processus créé, nommé «pid2», affiche le message «je suis pid2»,*
- *le troisième processus créé, nommé «pid3», affiche le message «je suis pid3».*

2. *Le but de cette question est de synchroniser l'affichage des processus «pid1» et «pid3». Modifier le programme afin que le processus «pid1» affiche son message avant l'affichage du processus «pid3».*

N.B : ne pas utiliser la fonction « sleep() ».

Réponses:

1.

```
main() {
  pid_t pid1, pid2, pid3;
  pid1=fork();  // le processus père crée le premier fils « pid1 »
```

```
    if (pid1!=0) { // si je suis le processus père
       pid2=fork();// le processus père crée le deuxième fils « pid2 »
       if (pid2!=0) { // si je suis le processus père
          pid3=fork();// le processus père crée le troisième fils « pid3 »
          if (pid3 !=0) // le processus père
               while (wait(0)!=-1); // le père attend la terminaison de tous les fils
          else { // si je suis le processus fils « pid3 »
               printf("Je suis pid3 = %d pid du père= %d \n",getpid(),getppid());
          }
        else // si je suis le processus fils « pid2 »
          printf("Je suis pid2 = %d pid du père= %d \n",getpid(),getppid());
    }
    else // si je suis le processus fils « pid1 »
         printf("Je suis pid1 = %d pid du père= %d \n",getpid(),getppid());
}
```

2. Le principe est que le processus « pid1 » affiche son message ensuite informe le processus « pid3 » de son affichage en lui envoyant un signal. Puisque le processus « pid3 » est créé après le processus « pid1 » alors le processus « pid1 » ne connait pas l'dentifiant de « pid3 ». Pour cela on utilise un tube (créé par le père) pour que le processus « pid3 » envoie son identifiant à « pid1 ».

Principe :

Le processus père :

- crée un tube
- crée les fils
- attend la terminaison de se fils

Le processus « pid3 »

- installe un handler pour le signal « SIGUSR1 »
- écrit son identifiant dans le tube
- attend le signal « SIGUSR1 » de « pid1 »
- affiche son message

Le processus « pid1 »

- lit, à partir du tube, l'identifiant du processus « pid3 »
- affiche son message
- envoie un signal à « pid3 » lui indiquant qu'il a affiché son message

Code C

```
void hand() {}
main() {
    …
        else { // si je suis le processus fils « pid3 »
            int pid;
            signal(SIGUSR1,hand); // installe un handler pour « SIGUSR1 »
            close(p[0]);
            pid=getpid();  // sauvegarde dans pid l'identifiant de « pid3 »
            // écrit dans le tube p l'identifiant de « pid3 »
            nb_ecrits= write(p[1],&pid,1*sizeof(int));
            pause(); // attend le signal « SIGUSR1 » envoyé par « pid1 »
            printf("Je suis pid3 = %d pid du père= %d \n",getpid(),getppid());
        }
      }
      else { // si je suis le processus « pid2 »
        close(p[0]);
        close(p[1]);
        printf("Je suis pid2 = %d pid du père= %d \n",getpid(),getppid());
      }
    }
    else { // si je suis le processus « pid1 »
        int pid; // id est utilisé pour sauvegarder l'identifiant de « pid3 »
        close (p[1]);
        // lit, à partir du tube p, l'identifiant de « pid3 »
        nb_lus=read(p[0],&pid,1*sizeof(int));
        printf("Je suis pid1 = %d pid du père %d \n",getpid(), getppid());
        // envoie du signal « SIGUSR1 » au processus « pid3 »
        kill(pid,SIGUSR1);
    }
}
```

Exercice 4 :

Le Le but est de faire communiquer le processus père avec ces deux processus fils.

1. *Écrire un programme dans lequel :*

a. *le processus père crée deux processus fils « pid1 » et « pid2 » et ensuite écrit dans un tube les 26 lettres de l'alphabet (chaque lettre minuscule est suivie de la même lettre majuscule : aAbBcC….).*

b. *les processus fils lisent les lettres une à une ensuite les affichent.*

2. *Modifier le programme pour synchroniser la lecture des processus fils.*

a. *Le processus fils « pid1 » lit les lettres minuscules.*

b. *Le processus fils « pid2 » lit les lettres majuscules.*

Réponses:

1. Dans cette question, aucune synchronisation entre les processus « pid1 » et « pid2 » pour la lecture dans le tube. Chaque processus lit 26 lettres mais la lecture est imprévisible.

```
int main() {
   int i,j, nb_lus, nb_ecrits, p[2];
   char s[52];
   pid_t  pid1,pid2;
   pipe(p);   // le processus père crée le tube p
   pid1=fork();  // le processus père crée le premier fils « pid1 »
   if (pid1!=0) { // si je suis le processus père
      pid2=fork();  // le processus père crée le deuxième fils « pid2 »
      if (pid2!=0) {   // si je suis le processus  père
            close(p[0]);
            // On sauvegarde dans le tableau s les lettres (une lettre minuscule suivie
            // d'une lettre majuscule).
            j=0;
            for(i=0;i<26;i++) {
               s[j]='a'+i;
               s[j+1]='A'+i;
               j=j+2;
            }
```

```
            // on écrit le tableau s dans le tube p
            nb_ecrits= write(p[1],s,52*sizeof(char));
            while (wait(0) != -1);  // attend la terminaison des ses fils
        }
    else {   // si je suis le processus « pid2 »
       char *buf;  // on sauvegarde dans buf les lettres lues à partir du tube p
       buf=(char *)malloc(27*sizeof(char));
       close(p[1]);
       // on lit les lettres une à une à partir du tube p
       for(i=0;i<26;i++) {
            // on lit une lettre à partir du tube p
            nb_lus=read(p[0],buf+i,sizeof(char));
        }
     }
    }
    else {  // si je suis le processus « pid1 »
       char *buf;   // on sauvegarde dans buf les lettres lues à partir du tube p
       buf=(char *)malloc(27*sizeof(char));
       close(p[1]);
       // on lit les lettres une à une à partir du tube p
       for(i=0;i<26;i++) {
            // on lit une lettre à partir du tube p
            nb_lus=read(p[0],buf+i,sizeof(char));
     }
    }
    return 0;
}
```

2. Dans la solution précédente, la lecture est imprévisible. L'objectif dans cette solution est de synchroniser l'accès au tube (la lecture sera entrelacée entre les deux processus), pour cela on utilise les signaux afin que le fils « pid1 » lit les lettres minuscules et le fils « pid2 » lit les lettres majuscules. Puisque le processus « pid2 » est créé après le processus « pid1 », donc le processus « pid1 » ne connait pas l'identifiant du processus « pid2 ». Pour cela on utilise un tube pour que « pid2 envoie son identifiant à « pid1 ».

Le processus père :

- crée les tubes p et p1
- crée les processus fils « pid1 » et « pid2 »
- écrit dans le tube p les 52 lettres (une lettre minuscule suivie d'une lettre majuscule)
- attend la terminaison de ses fils

Le processus « pid2 » :

- installe un handler pour le signal « SIGUSR1 »
- écrit son identifiant dans le tube p1.
- exécute la boucle suivante qui permet une lecture alternée avec « pid1 » :
  pour (i=0;i<26;i++) {
    attend le signal « SIGUSR1 » de « pid1 »
    lit, à partir du tube p, une lettre (la lettre majuscule)
    envoie le signal « SIGUSR1 » au processus « pid1 »

Le processus « pid1 »:

- installe un handler pour le signal « SIGUSR1 »
- lit, à partir du tube p1, l'identifiant du processus « pid2 ».
- exécute la boucle suivante qui permet une lecture alternée avec « pid2 »:
  pour (i=0;i<26;i++) {
    lit, à partir du tube p, une lettre (la lettre minuscule)
    envoie le signal « SIGUSR1 » au processus « pid2 »
    attend le signal « SIGUSR1 » de « pid2 »

Code en C

```
…
void hand(int sig) {
}
int main() {
…
    else {  // si je suis le processus « pid2 »
        char *buf;
        int pid;
        // installe un handler pour le signal SIGUSR1 « SIGUSR1 »
        signal(SIGUSR1,hand); // ou signal(SIGUSR1,SIG_IGN);
        pid=getpid();  // sauvegarde dans pid l'identifiant de « pid2 »
        // écrit dans le tube p1 l'identifiant de pid2
        nb_ecrits= write(p1[1],&pid2,sizeof(int));
```

```
            buf=(char *)malloc(27*sizeof(char));
            close(p[1]);
            // lit les lettre majuscules à partir du tube p
            for(i=0;i<26;i++) {
               // attend un signal de « pid1 »
               pause();
               // lit une lettre à partir du tube p
               nb_lus=read(p[0],buf+i,sizeof(char));
               // envoie le signal « SIGUSR1 » à « pid1 »
               kill(pid1,SIGUSR1);
            }
         }
   }
   else {  // si je suis le processus « pid1 »
      char *buf;
      int pid;  // on sauvegarde dans pid l'identifiant de « pid2 »
   // installe un handler pour le signal « SIGUSR1 »
      signal(SIGUSR1,hand);  // ou  signal(SIGUSR1,SIG_IGN);
      // lit l'identifiant de « pid2 » et le sauvegarde dans la variable pid
      nb_lus=read(p1[0],&pid,sizeof(int));
      buf=(char *)malloc(27*sizeof(char));
      close(p[1]);
            // lit les lettre minuscules à partir du tube p
      for(i=0;i<26;i++) {
            // lit une lettre à partir du tube p
            nb_lus=read(p[0],buf+i,sizeof(char));
            // envoie le signal « SIGUSR1 » à « pid2 »
            kill(pid,SIGUSR1);
            // attend un signal de « pid2 »
            pause();
     }
   }
   return 0;
}
```

Exercice 5 :

Sans utiliser la constante prédéfinie PIPE_BUF, écrire un programme qui permet de déterminer la capacité maximale d'un tube anonyme.

Réponse :

Algorithme:

Tant qu'on peut écrire dans le tube on écrit un caractère.
Lorsque le tube est plein, le programme se bloc.
….

```
main () {
char c='a';  // le caractère à écrire dans le tube.
   int i, p[2];
   pipe(p);
   printf("Ecriture dans le tube ");
   i=0;
   while(write(p[1],&c,1)==1){  // tant qu'on peut écrire dans le tube p
      i=i+1;
      printf("PIPE_BUF=%d  Le nombre d'octets écrits est %d  \n", PIPE_BUF,i);
   }
}
```

Exercice 6 :

Le but est de faire communiquer deux processus entre eux en utilisant les tubes anonymes. Écrire un programme dans lequel :

- *Le processus père crée un processus fils et lui envoie un message.*
- *Le processus fils lit le message et ensuite répond au processus père.*
- *Le processus père attend la réponse de son fils avant de se terminer.*

Réponses :

On crée deux pipes : un est utilisé pour envoyer le message du père au fils et l'autre est utilisé pour envoyer le message du fils au père.

```
…
# define MSG1 " Salam fils "  // message envoyé par le père
```

```
# define MSG2 " Salam père "   // message envoyé par le fils
main() {
    int p1[2], p2[2], nb_lus1,nb_lus2,nb_ecrits1,nb_ecrits2;
    pid_t  pid;
    char  buf1[sizeof(MSG1)], buf2[sizeof(MSG2)];
    pipe(p1);
    pipe(p2);
    pid = fork(); // le processus père crée le fils « pid »
    if (pid == 0) {  // si je suis le processus « pid »
        close(p2[0]);
        close(p1[1]);
            // lire, à partir du tube p1, le message envoyé par le père
            nb_lus1 = read(p1[0], buf1, sizeof(buf1));
            // écrire dans le tube p2, le message à envoyer au père
            nb_ecrits2= write(p2[1], MSG2, sizeof(MSG2));
    }
    else {   // si je suis le processus père
        close(p1[0]);
        close(p2[1]);
        // écrire dans le tube p1, le message à envoyer au processus « pid »
        nb_ecrits1 = write(p1[1], MSG1, sizeof(MSG1));
        // lire, à partir du tube p2, le message envoyé par le fils « pid »
        nb_lus2 = read(p2[0], buf2, sizeof(buf2));
        close(p1[1]);
        close(p2[0]);
    }
}
```

Exercice 7 :

Écrire un programme où le père et le fils communiquent par l'intermédiaire d'un tube anonyme. Le père envoie au fils N nombres entiers. Le fils calcule la somme et l'envoie au père qui l'affiche à l'écran.

Réponse :

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

```
# define N 10  // on suppose que N < la limite maximale
main () {
    int c[N];
    int i, nb_lus, nb_ecrits, somme=0;
    int p1[2], p2[2];
    pipe(p1);  // création du tube p1
    pipe(p2);  // création du  tube p2
    if (fork()!=0) {  // le père crée le fils.
        close(p1[0]);  // on ferme les descripteurs non utilisés
        close(p2[1]);
        // on itialise le tableau qui contient les entiers à envoyer
        …
        // le père écrit dans le tube p1 les N entiers.
        nb_ecrits=write(p1[1],c, N*sizeof(int)) ;
        // le père lit, à partir du tube p2, la somme calculée par le fils
        nb_lus=read(p2[0], &somme, 4);
        close(p1[1]);
        close(p2[0]);
    }
    else {  // le processus fils
        close(p1[1]); // on ferme les descripteurs non utilisés
        close(p2[0]);
        // le fils lit, à partir du tube p1, les N entiers envoyés par le père et les
        // sauvegarde dans la variable c.
        nb_lus=read(p1[0],c, N*sizeof(int));
        // le fils calcule la somme, ensuite écrit le résultat dans le tube.
        somme=0;
        for (i=0; i<N; i++)
            somme+=c[i];
        // le fils écrit dans le tube p2 la somme calculée.
        nb_ecrits=write(p2[1],&somme,4);
        close(p1[0]);
        close(p2[1]);
        }
}
```

Exercice 8 :

Le but est de faire communiquer le processus père avec ses deux processus fils « pid1 » et « pid2 » en utilisant les tubes anonymes.

1. *Écrire un programme dans lequel :*

- *Le processus père crée deux processus fils, ensuite, lit au clavier « n » entiers.*
- *Le processus père envoie à ces deux fils les « n » entiers.*
- *Le processus fils « pid1 » calcule la somme des « n » nombres entiers et affiche le résultat.*
- *Le processus fils « pid2 » calcule le produit des « n » nombres entiers et affiche le résultat.*

2. *Modifier le programme précédent afin que les processus fils fassent les calculs, ensuite ils envoient les résultats au père qui les affiche.*

Réponses:

1. Puisque la lecture dans un tube est destructive, alors on a besoin de deux tubes dans lesquels le processus père écrit les n entiers. Le premier fils lit les entiers à partir d'un tube alors que le deuxième fils lit les n entiers à partir de l'autre tube.

```
….
int tab[N];
int main() {
    int i, nb_lus, nb_ecrits, p1[2],p2[2];
    pid_t  pid1, pid2;
    pipe(p1);
    pipe(p2);
    pid1=fork();   // le processus père crée le premier fils « pid1 »
    if (pid1!=0) { // // si je suis le processus père
        pid2=fork();  // le processus père crée le deuxième fils « pid2 »
        if (pid2!=0) { // si je suis le processus père
            close(p1[0]); // on ferme les descripteurs non utilisés
            close(p2[0]);
            // le père lit N entiers à partir du clavier
            for(i=0;i<N;i++) {
                scanf("%d",&tab[i]);
            }
            // le père écrit dans le premier tube p1 les N entiers
```

```
            nb_ecrits = write(p1[1], tab, N*sizeof(int));
            // le père écrit dans le deuxième tube p2 les N entiers
            nb_ecrits = write(p2[1], tab, N*sizeof(int));
            while (wait(0) !=-1) ;
        }
        else { // si je suis le processus « pid2 »
            int p=1;
            close(p1[1]);  // on ferme les descripteurs non utilisés
            close(p1[0]);
            close(p2[1]);
            // le processus « pid2 » lit, à partir du tube p2 les N entiers
            // envoyés par le père (sont écrits par le père dans le tube p2)
            nb_lus=read(p2[0], tab, N*sizeof(int));
            // le processus « pid2 » calcule le produit des N entiers
            for(i=0;i<N;i++) {
                p=p*tab[i];
            }
        }
    }
    else { // si je suis le processus « pid1 »
        int s=0;
        close(p1[1]);  // on ferme les descripteurs non utilisés
        close(p2[1]);
        close(p2[0]);
        // le processus « pid1 » lit, à partir du tube p1 les N entiers
        // envoyés par le père (sont écrits par le père dans le tube p1)
        nb_lus=read(p1[0],tab,N*sizeof(int));
        // le processus « pid1 » calcule la somme des N entiers
        for(i=0;i<N;i++) {
            s=s+tab[i];
        }
    }
    return 0;
}
```

2. Par rapport au code précédent, il suffit de rajouter :

- Dans le code du processus père :
  - une instruction de lecture de la somme à partire du tube p1.

```
nb_lus=read(p1[0],&s, sizeof(int));
printf("reçu de pid1 %d somme= %d \n", pid1,s);
```

  - Une instruction de lecture du produit à partir du tube p2.

```
nb_lus=read(p2[0],&p, sizeof(int));
printf("reçu de pid2 %d produit= %d \n", pid2,p);
```

- Dans le processus « pid1 » :
  - Une instruction d'écriture de la somme dans le tube p1.

```
nb_ecrits= write(p1[1],&s,sizeof(int));
```

- Dans le processus « pid2 » :
  - Une instruction d'écriture du produit dans le tube p2.

```
nb_ecrits= write(p2[1],&p, sizeof(int));
```

```
int main() {
…
        if (pid2!=0) { // processus père
            int s,p;
            for(i=0;i<N;i++) {
                scanf("%d",&tab[i]);
            }
            nb_ecrits= write(p1[1],tab,N*sizeof(int));
            nb_ecrits= write(p2[1],tab,N*sizeof(int));
            // Instructions à rajouter dans le code du père
            nb_lus=read(p1[0],&s, sizeof(int));
            printf("reçu de pid1 %d somme= %d \n", pid1,s);
            nb_lus=read(p2[0],&p, sizeof(int));
            printf("reçu de pid2 %d produit= %d \n", pid2,p);
            exit(0);
        }
        else { // le processus pid2
            int p=1;
            nb_lus=read(p2[0],tab,N*sizeof(int));
            for(i=0;i<N;i++) {
                p=p*tab[i];
            }
```

```
                nb_ecrits= write(p2[1],&p, sizeof(int));
            }
        }
        else { // le processus pid1
            int s=0;
            nb_lus=read(p1[0],tab,N*sizeof(int));
            for(i=0;i<N;i++) {
                s=s+tab[i];
            }
            nb_ecrits= write(p1[1],&s,sizeof(int));
        }
        return 0;
    }
```

Exercice 9 :

1. *Ecrire un programme dans lequel le processus père crée deux processus fils qui ont pour identifiants « pid1 » et « pid2 ».*
2. *Compléter le programme précédent afin que l'identifiant de chaque processus fils soit connu par l'autre processus fils.*

*Dans la suite on souhaite faire communiquer le processus père avec ses deux processus fils en utilisant **un seul tube anonyme.** Pour simplifier, on suppose que chaque processus fils connait l'identifiant de l'autre processus fils.*

3. *Soit « n » un entier pair, écrire un programme dans lequel :*
   - *Le processus père crée deux processus fils qui ont pour identifiants « pid1 » et « pid2 ».*
   - *Les deux processus fils écrivent des nombres entiers dans le tube de manière concurrente (l'ordre d'écriture est imprévisible). Pour simplifier, on suppose que :*
     - *«pid1» écrit les entiers pairs qui sont compris entre 0 et n (0, 2,4,…, n-2).*
     - *«pid2» écrit les entiers impairs qui sont compris entre 1 et n (1,3,5,…,n-1).*
   - *Le processus père attend la terminaison de ses fils pour lire les données à partir du tube.*
4. *Ecrire le programme dans lequel on synchronise les processus fils afin d'obtenir l'ordre d'écriture suivant dans le tube : 0, 1, 2, ….,n-1. Le processus père attend la terminaison de ses fils pour lire les données à partir du tube.*

Réponses:

1. Création de deux processus par le processus père

```
…
main() {
     pid_t pid1,pid2;
     pid1=fork();  // le processus père crée un premier fils « pid1 »
     if (pid1!=0) {    //  si je suis le processus père
          pid2=fork(); // le processus père crée le deuxième processus fils «pid2».
     }
}
```

2. Puisque « pid2 » est créé après le processus « pid1 », donc l'identifiant « pid1 » est connu dans le processus fils d'identifiant « pid2 ». On doit communiquer l'identifiant « pid2 » au premier processus en utilisant un tube.

1ère solution : c'est le père qui communique l'identifiant « pid2 » au premier fils d'identifiant « pid1 ».

2ème solution : le deuxième fils qui communique son identifiant « pid2 » au premier fils d'identifiant « pid1 ».

1ère solution :

Le père communique l'identifiant « pid2 » au premier fils.

```
…..
int main() {
   int nb_lus, nb_ecrits;
   int p[2];
   pid_t pid1, pid2;
   pipe(p);   // création du tube (pipe) anonyme « p »
   pid1=fork();  //  le père crée un premier processus fils d'identifiant « pid1 ».
   if (pid1!=0) { // si je suis le processus père
      pid2=fork();  // le père crée un deuxième  fils d'identifiant « pid2 ».
      if (pid2!=0) {   // si je suis le processus père
         close(p[0]);
         // le processus père écrit l'identifiant « pid2 » dans le tube
         nb_ecrits= write(p[1],&pid2,sizeof(int));
         wait(0);
         wait(0);
       }
```

```
    }
    else {  // si je suis le processus fils d'identifiant « pid1 »
        int pid ;
        close(p[1]);
        // lit l'identifiant de « pid2 », à partir du tube p, et le sauvegarde dans la
        // variable « pid »
        nb_lus=read(p[0],&pid,sizeof(int));
    }
    return 0;
}
```

2ème solution :

le deuxième fils communique son identifiant « pid2 » au premier fils.

```
int main() {
    ….
        else {   // si je suis l processus fils d'identifiant « pid2 »
            close(p[0]) ;
            pid=getpid();  // sauvegarde son identifiant dans la variable pid
            // écrit la variable pid dans le tube p
            nb_ecrits= write(p[1],&pid,sizeof(int));
        }
    }
    else {  // si j suis le processus fils d'identifiant « pid1 »
        close(p[1]);
        // lit l'identifiant du processus « pid2 », à partir du tube p, et le
        // sauvegarde dans la variable « pid »
        nb_lus=read(p[0],&pid,sizeof(int));
    }
    return 0;
}
```

3. Les processus écrivent, dans le tube, de manière asynchrone :

- « pid1 » écrit les entiers pairs qui sont compris entre 0 et n-2.
- « pid2 » écrit les entiers impairs qui sont compris entre 1 et n-1.

Le processus père attend la terminaison de ses fils pour lire les données à partir du tube

….

```
int tab[N] ;
int main() {
   …
      if (pid2!=0) {   // si je suis le processus père
         close(p[1]);
         // le père attend la terminaison de se fils et ensuite lit les entiers à partir
         // du tube p
         while(wait(0) !=-1) ;
         nb_lus= read(p[0],tab,N*sizeof(int));
       }
    }
         else {   // si je suis le processus fils d'identifiant « pid2 »
              close(p1[0]);
              // écrire les entiers impairs (entier après entier) dans le tube p1
              i=1;
              while (i<N) {
                        nb_ecrits= write(p1[1], &i, sizeof(int));
                        i+=2;
              }
         }
    }
    else {  // si je suis le processus « pid1 »
         close(p1[0]);
         // écrire les entiers pairs (entier après entier) dans le tube p1
         i=0;
         while (i<N) {
              nb_ecrits= write(p[1], &i, sizeof(int));
              i+=2;
         }
    }
    return 0;
}
```

4. On synchronise l'écriture de deux processus fils dans le tube afin d'obtenir l'ordre d'écriture suivant dans le tube : 0, 1, 2, ….,n-1.

Principe :

Le processus père

- crée deux tubes
- crée deux processus fils « pid1 » et « pid2 ».
- écrit dans un tube l'identifiant de « pid2 » pour que « pid1 » puisse le lire
- attend la terminaison de ses deux fils
- lit les entiers à partir du tube

Le processus fils d'identifiant « pid2 »

- installe un handler pour le signal « SIGUSR1 »
- exécute une boucle pour l'écriture synchrone

  i=1;
  tant que (i<N) {
  attend un signale du « pid1 »;
  écrit l'entier i (entier impair) dans le tube
  envoie un signal à « pid1 »
  incrémenter i de 2 (i+=2);

Le processus fils d'identifiant « pid1 »

- installe un handler pour le signal « SIGUSR1 »
- lit, à partir du tube p l'identifiant de pid2
- exécute une boucle pour l'écriture synchrone

  i=0;
  tant que (i<N) {
  écrit l'entier i (entier pair) dans le tube
  envoie un signal à « pid2 »
  attend un signale du « pid2 »;
  incrémenter i de 2 (i+=2);

Code C:

```
…
void hand(int sig) {    }
main() {
…
        else { // si je suis le processus fils d'identifiant « pid2 »
            // on installe un handler pour le signal « SIGUSR1 »
            signal(SIGUSR1,hand);   // ou  signal(SIGUSR1, SIG_IGN);
            // écrire les entiers impairs (entier après entier) dans le tube p1
            i=1;
```

```
            while (i<N) {
                // attend un signal de « pid1 »
                pause();
                // écrit l'entier i dans le tube p1
                nb_ecrits= write(p1[1], &i, sizeof(int));
                // envoie le signal « SIGUSR1 » au processus « pid1 »
                kill(pid1,SIGUSR1);
                i+=2;
            }
        }
    }
    else {  // si je suis le processus fils d'identifiant « pid1 »
        // installe un handler pour le signal « SIGUSR1 »
        signal(SIGUSR1,hand);  // ou  signal(SIGUSR1,SIG_IGN);
        sleep(1);  // pour laisser le temps à « pid2 » d'installer le handler
        // lit l'identifiant du processus « pid2 », à partir du tube p
        nb_lus=read(p[0], &pid2, sizeof(int));
        // écrire les entiers pairs (entier après entier) dans le tube p1
        i=0;
        while (i<N) {
            // écrit l'entier i dans le tube p1
            nb_ecrits= write(p1[1], &i, sizeof(int));
            // envoie le signal « SIGUSR1 » au processus « pid2 »
            kill(pid2, SIGUSR1);
            // attend un signal de « pid2 »
            pause();
            i+=2;
        }
    }
}
```

Remarque :

L'inconvénient de la version précédente c'est qu'il y a risque de blocage. Pour résoudre ce problème, on remplace « pause() » par une attente active (voir les exercices sur les signaux).

Chapitre 4 : Threads Posix

1. Définitions des threads

- Un (une) thread (fil d'exécution, processus léger) est une partie du code d'un programme (une fonction), qui se déroule parallèlement à d'autres parties du programme.
- Lorsqu'on lance l'exécution d'un programme, un nouveau processus est créé. Dans ce processus, un thread est crée pour exécuter la fonction principal « main() ». Ce thread est appelé le thread principal (ou thread original).
- Un thread peut créer d'autres threads au sein du même processus; tous ces threads exécutent alors le même programme, mais chaque thread est chargé d'exécuter une partie différente (une fonction) du programme à un instant donné.
- Les threads s'exécutent en parallèle (en cas de multiprocesseur) ou de manière alternative (en cas de mono processeur). Un thread, qui termine son quantum de temps, sera interrompu pour permettre l'exécution d'un autre thread et attend son tour pour être ré-exécuté. C'est le système qui s'occupe de l'ordonnancement des threads.

2. Environnement de développement

GNU/Linux implémente l'API de threading standard POSIX (appelée aussi « pthread »). C'est une librairie qui contient les fonctions nécessaires pour la manipulation des threads.

- **Compilation**
  - Les fichiers C utilisant les threads doivent comporter la directive « include<pthread.h> ».
  - Les fonctions de « pthread » se trouvent dans la librairie « libpthread », pour la compilation d'un fichier source C utilisant les threads, il faut rajouter « -lpthread » sur la ligne de commande lors de l'édition de liens.
  - Les procédures de la librairie « pthread » renvoient 0 en cas de succès ou un code (>0) en cas d'erreur.

Exemple: Pour le programme « test_thread.c » on compile comme suit:

```
% gcc  test_thread.c  -o  test_thread  -lpthread
```

3. Création des threads

Chaque thread exécute une fonction de thread. La terminaison de cette fonction entraine la terminaison du thread. La création d'un thread est réalisée grâce à la fonction «pthread_create()».

Syntaxe:

```
int pthread_create ( pthread_t  *idthread,
                     NULL,
                     void * (*fonc) (void *),
                     void *arg );
```

- La fonction « pthread_create() » crée un nouveau thread identifié par « idthread » pour exécuter la fonction « fonc » qui est appelée avec l'argument « arg », de type « void * », qui désigne l'adresse mémoire, de(s) paramètre(s) à passer à la fonction.
- Dans la fonction « fonc » on définit le code à exécuter par le thread. Elle doit avoir la forme « void *nom_fonction(void* arg) ».

4. Variables locales et variables globales

- Les variables globales sont accessibles par tous les threads.
- Une variable locale à un thread n'est pas accessible par les autres threads.

5. Terminaison des threads

5.1. Terminaison de l'activité principal

La terminaison de l'activité initiale d'un processus (fonction «main()») entraine la terminaison du processus et par conséquent la terminaison des autres threads encore actifs dans le cadre du processus.

5.2. Terminaison par « exit() »

L'appel de « exit() » par un thread quelconque (principal ou fils) entraine la fin du processus et par conséquent la terminaison de tous les threads du processus.

5.3. Terminaison par la fonction «pthread_exit()»

La fonction «pthread_exit() » permet de terminer le thread qui l'appelle. Elle permet aussi au thread de retourner des résultats qui peuvent être récupérés par un autre thread grâce à la fonction « pthread_join() ».

Syntaxe:

void pthread_exit (void *p_status);

- «*p_status »: pointeur sur le résultat éventuellement retourné par le thread.
- Si «pthread_exit()» n'est pas donné explicitement dans la fonction thread alors « pthread_exit(NULL) » (pthread_exit(), pthread_exit(0)) est implicitement (automatiquement) exécuté pour terminer le thread.

6. Synchronisation

6.1. Introduction

Les threads au sein du même processus sont concurrents et peuvent avoir un accès simultané à des ressources partagées. Donc leur synchronisation est indispensable afin d'assurer une utilisation cohérente de ces données.

6.2. Section Critique

Une section critique est une suite d'instructions dans un programme dans laquelle se font des accès concurrents à une ressource partagée.

⇒ l'accès à cette section critique doit se faire en exclusion mutuelle.

6.3. Synchronisation sur terminaison des threads : « pthread_join() »

Un thread peut suspendre son exécution jusqu'à la terminaison d'un autre thread en appelant la fonction « pthread_join() ».

Syntaxe:

int pthread_join (pthread_t thrd, void **code_retour);

- « thrd » : désigne l'identifiant du thread qu'on attend sa terminaison.
- « code_retour »: un pointeur vers une variable qui recevra la valeur de retour du thread qui s'est terminé et qui est renvoyé par «pthread_exit()».

6.4. Mécanisme de synchronisation sur attente active

La synchronisation sur attente active est une solution algorithmique qui permet, par exemple, de bloquer l'exécution des instructions dans « thread2 » jusqu'à la fin d'exécution des instructions dans « thread1 » en utilisant une variable globale.

6.5. Les sémaphore d'exclusion mutuelle (Les Mutexs)

Les mutex, raccourcis de MUTual EXclusion locks (verrous d'exclusion mutuelle), sont des objets de type «pthread_mutex_t » qui permettent de mettre en œuvre un mécanisme de synchronisation qui résout le problème de l'exclusion mutuelle**.**

Il existe deux états pour un mutex (disponible ou verrouillé).

- Lorsqu'un mutex est verrouillé par un thread, on dit que ce thread tient le mutex. Tant que le thread tient le mutex, aucun autre thread ne peut accéder à la ressource critique.
- Un mutex ne peut être tenu que par un seul thread à la fois.
- Lorsqu'un thread demande à verrouiller un mutex déjà maintenu par un autre thread, le thread demandeur est bloqué jusqu'à ce que le mutex soit libéré.

Le principe des mutex est basé sur l'algorithme de Dijkstra

- Opération P (verrouiller l'accès).
- Accès à la ressource critique (la variable globale).
- Opération V (libérer l'accès).

Les fonctions de manipulations des mutexs:

- « pthread_mutex_init ( ) »: permet de créer le mutex (le verrou) et le mettre à l'état "unlock" (ouvert ou disponible).
- « pthread_mutex_destroy( ) »: permet de détruire le mutex.
- « pthread_mutex_lock ( ) »: tentative d'avoir le mutex. S'il est déjà pris, le thread est bloqué
- « pthread_mutex_unlock( ) » : rend le mutex et libère un thread

Création de mutex

La création de mutex (verrou) consiste à définir un objet de type « pthread_mutex_t » et de l'initialiser de deux manières.

- Initialisation statique à l'aide de la constante « PTHREAD_MUTEX_INITIALIZER »
- Initialisation par appel de la fonction « pthread_mutex_init() » déclarée dans <pthread.h>, qui permet de créer le verrou(le mutex) et le mettre en état "unlock".

Syntaxe:

int pthread_mutex_init(pthread_mutex_t *mutex_pt, pthread_mutexattr_t *attr) ;

- « mutex_pt » : pointe sur une zone réservée pour contenir le mutex créé.
- attr : ensemble d'attributs à affecter au mutex. On le met à NULL

L'opération P pour un mutex : appel blocant:

La fonction « pthread_mutex_lock() » permet, à un thread de réaliser de façon atomique, une opération P (verrouillé le mutex) par un thread. Si le mutex est déjà verrouillé (tenu par un autre thread), alors le thread qui appelle la fonction «pthread_lock() » reste bloqué jusqu'à la réalisation de l'opération V (libérer le mutex) par un autre thread.

Syntaxe:

int pthread_mutex_lock(pthread_mutex_t *mutex_pt);

- mutex_pt : pointe sur le mutex à réserver (à verrouiller)

L'opération V pour un mutex

L'appel de la fonction « pthread_mutex_unlock() » permet de libérer un mutex et de débloquer les threads en attente sur ce mutex.

Syntaxe:

int pthread_mutex_unlock(pthread_mutex_t *mutex_pt);

- mutex_pt : pointe sur le mutex à libérer

5.4. La fonction de destruction d'un mutex

On peut détruire le sémaphore par un appel à la fonction «pthread_mutex_destroy() ».

Syntaxe:

int pthread_mutex_destroy(pthread_mutex_t *mutex_pt);

- mutex_pt : pointe sur le mutex à détruire

Exercice 1:

1. *Ecrire un programme dans lequel :*
   - *Le thread principal (fonction «main()») crée un thread d'identifiant «th1».*
   - *Le thread fils «th1» lit un entier au clavier de type «int» et le sauvegarde dans une variable globale «x».*
   - *Le thread principal affiche la valeur lue par le thread «th1».*
2. *On suppose que la variable «x» est une variable locale, modifier le programme précédent afin que le thread « th1 » transmette la valeur lue au thread principal.*
3. *On suppose maintenant que le thread principal crée un deuxième thread d'identifiant «th2». Modifier le programme de la question 2 pour que le thread «th1» transmette la valeur lue au thread «th2».*
4. *Refaire la question 2 en supposant que «x» est un réel de type «double».*

Réponses :

1. Soit « x » une variable globale. On définit la fonction « lire() » qui sera exécutée par le thread « th1 » et consiste à lire la valeur de « x ». Ensuite la valeur de « x » sera affichée par le thread principal.

```
# include <pthread.h>
# include <stdio.h>
int x;   // variable globale
void* lire(void* k ) {  // définition de la fonction thread
    printf(" Entrer x : ");
    scanf("%d",&x);
}
// Programme principal
main () {
    pthread_t  th1;
    // création du thread « th1 » qui exécute la fonction « lire »
    pthread_create (&th1, NULL, lire, NULL );
    // le thread principal attend la terminaison du thread « th1 »
    pthread_join(th1,NULL); // attend la terminaison du thread « th1 »
    printf ("x=  %d  \n",x);
}
```

2. On suppose maintenant que « x » est une variable locale à la fonction « lire() ». Donc le thread « th1 » utilise la fonction « pthread_exit() » pour transmettre la valeur de « x » et le thread principal appelle « pthread_join() » pour récupérer la valeur de « x ».

```
// définition de la fonction « lire() »
    void* lire(void* k ) {
        int x;
        printf(" Entrer x : ");
        scanf("%d",&x);
        pthread_exit((void *)x);
    }
```

Pour le programme principal on propose deux versions.

- Dans la première version « x » est de type « void ».
- Dans la deuxième version « x » est de type « int »

Version 1: « x » est de type « void »

```
    # include <pthread.h>
    # include <stdio.h>
    void* lire(void* ) ;
    // Programme principal
    main () {
        void *x;
        pthread_t  th1;
        pthread_create (&th1, NULL, lire, NULL );
        pthread_join(th1,&x);
    // puisque « x »  est de type « void » il faut faire un transtypage (un cast)
    // pour transformer sa valeur vers « int »
        printf("x=  %d  \n",(int )x);
    }
```

Version 2: « x » est de type « int »

```
    # include <pthread.h>
    # include <stdio.h>
    void* lire(void* );
```

```
// Programme principal
int main () {
    int x;
    pthread_t  th1;
    pthread_create (&th1, NULL, lire, NULL );
    pthread_join(th1,(void *)&x);
    // puisque « x »  est de type « int » donc on n'a pas besoin
    // de faire un transtypage
    printf("x=  %d \n",x);
}
```

3. Le thread « th1 » utilise la fonction « pthread_exit() » pour transmettre la valeur de « x » et le thread « th2 » utilise « pthread_join() » pour récupérer la valeur de « x ». On définit la fonction « lire() » qui sera exécutée par le thread « th1 » et consiste à lire la valeur de « x » et la transmettre à l'aide de la fonction « pthread_exit() » et la fonction « afficher() » qui sera exécutée par le thread « th2 » et consiste à récupérer la valeur de « x » à l'aide de la fonction « pthread_join() ».

```
…
void* lire(void *  ); // exécuté par le thread « th1 »
void* affiche(void* k ) {  // exécuté par le thread « th2 »
    int x;
    pthread_join(th1,(void *)&x);
    printf("x=  %d  \n",x);
}
// Programme principal
int main () {
    pthread_create (&th1, NULL, lire, NULL );
    pthread_create (&th2, NULL, affiche, NULL );
    pthread_join(th2,NULL);
    return 0;
}
```

4. Dans cette question on suppose que « x » est un réel de type « double ».

```
// défintion de la function « lire() »
void* lire(void* k ) {
    double *x=(double *)malloc(sizeof(double));
    printf(" thread th: Entrer x : ");
    scanf("%lf",x);
    pthread_exit((void *)x); // on transmet la valeur de « x ».
  // ou   pthread_exit(x);
}
```

Pour le programme principal on propose deux versions.

- Dans la première version on récupère la valeur de « x » dans une variable de type « double *res »
- Dans la deuxième version on récupère la valeur de « x » dans une variable de type « void *res »

Version 1: on récupère la valeur de « x » dans une variable de type « double *res »

```
…
void* lire(void* );
// Programme principal
main () {
    double x;
    double *res=(double *)malloc(sizeof(double));
    pthread_t  th1;
    pthread_create (&th1, NULL, lire, NULL );
    // on récupère la valeur de « x » dans la variable « res ».
    pthread_join(th1,(void *)&res );
    x=*res;
    printf("thread main:  x=  %lf \n",x);
}
```

Version 2: on récupère la valeur de « x » dans une variable de type « void *res »

```
…
void* lire(void* );
// Programme principal
int main () {
    double *x=(double *)malloc(sizeof(double));
```

```
    void *res;
    pthread_t  th1;
    pthread_create (&th1, NULL, lire, NULL );
    pthread_join(th1,&res );
    x=(double *)(res);
    printf("thread main:  x=  %lf \n",*x);
}
// ou
main () {
    double x;
    void *res;
    pthread_t  th1;
    pthread_create (&th1, NULL, lire, NULL );
    pthread_join(th1,&res );
    x=*(double *)(res);
    printf("thread main:  x=  %lf \n",x);
}
```

Exercice 2:

Ecrire un programme dans lequel :
- *Le thread principal (fonction « main() ») crée deux threads.*
- *Le premier thread affiche « n » fois une lettre donnée.*
- *Le deuxième thread affiche « m » fois une autre lettre donnée.*

Les lettres à afficher ainsi que les entiers « n » et « m » sont passés en argument à la fonction de thread.

Réponses:

Les deux threads créés exécutent la même fonction thread « affiche() » mais sur des données différentes. Puisqu'on passe un seul argument à la fonction thread, alors on définit une structure (struct data) qui a deux champs (la lettre à afficher (de type char) et le nombre de fois (de type int)). Puisque l'argument passé à la fonction thread est de type « void » alors on doit convertir (faire un transtypage), dans la fonction thread, de l'argument de la fonction thread vers le type « struct data).

```
# include <pthread.h>
# include <stdio.h>
// définir une structure qui a deux champs
struct  data {
     char c;
     int nb ;
};
// on définit la fonction thread
void* affiche(void* k ) {
     struct data *p = (struct data *)k ; // convertit *k vers le type «struct data»
     int i;
     for (i = 0; i < (*p).nb ; ++i)
          fputc ((*p).c , stderr );
     /* ou
     for (i = 0; i < p-> nb ; ++i)
          fputc (p->c , stderr );
     */
}
// Programme principal
main () {
     pthread_t  thread1_id, thread2_id ;
     struct data arg1, arg2 ;
     // un exemple de données pour le thread « thread1_id »
     arg1.c = 'a';
     arg1. nb = 300;
     pthread_create (&thread1_id, NULL, &affiche, &arg1 );
     // un exemple de données « thread2_id »
     arg2.c ='b';
     arg2.nb = 400;
     pthread_create (&thread2_id , NULL, &affiche, &arg2 );
     pthread_join(thread1_id,NULL); // attendre la terminaison du 1er thread
     pthread_join(thread2_id,NULL); //attendre la terminaison du 2ème thread
}
```

Exercice 3 :

Soit «tab» un tableau de réels de taille n (on suppose que n est pair). L'objectif est de faire coopérer deux threads fils pour calculer le plus grand élément du tableau «tab». On suppose que le tableau «tab» est initialisé par le thread père avant la création des threads fils.

1. *Ecrire un programme C dans lequel :*

- *Le thread père crée deux threads fils «th1» et «th2».*
- *Le thread «th1» cherche le plus grand élément dans la première moitié du tableau «tab» puis transmet la position du plus grand élément trouvé au thread père.*
- *Le thread «th2» cherche le plus grand élément dans la deuxième moitié du tableau «tab» puis transmet la position du plus grand élément trouvé au thread père.*
- *Le thread père calcule et affiche le plus grand élément du tableau «tab».*

2. *Refaire la question **1.** en supposant que chaque thread transmet à son père le plus grand élément trouvé.*

3. *Refaire la question **1.** en supposant que chaque thread transmet à son père le plus grand élément trouvé ainsi que sa position.*

N.B. *:*

- *La première moitié concerne les éléments d'indices 0, 1,…, n/2-1.*
- *La deuxième moitié concerne les éléments d'indices n/2, n/2+1,…, n-1.*

Réponses :

1.

Version1: On définit deux fonctions threads

```
# include <pthread.h>
# include <stdio.h>
# define N 100
double tab[N] ;

void* lire1(void* k ) {  // fonction exécutée par le thread th1
    int i, pos=0;
   double max=tab[0];
   for(i=0;i<N/2;i++) {
```

```
            if (tab[i]>max) {
                    max=tab[i];
                    pos=i;
            }
        }
        pthread_exit((void *)pos);    // retourne la position du plus grand élément
    }
    void* lire2(void* k ) { // fonction exécutée par le thread th2
        int i, pos=N/2;
        double max=tab[N/2];
        for(i=N/2;i<N;i++) {
            if (tab[i]>max) {
                    max=tab[i];
                    pos=i;
            }
        }
        pthread_exit((void *)pos);   // retourne la position du plus grand élément
    }
    main () {
            int pos1, pos2;
            double max;
            pthread_t  th1, th2;
            // initialization du tableau tab
            pthread_create (&th1, NULL, lire1, NULL );
            pthread_create (&th2, NULL, lire2, NULL );
            pthread_join(th1,(void *)&pos1);
            pthread_join(th2,(void *)&pos2);
            max=tab[pos2];
            if (tab[pos1]>tab[pos2])
                    max=tab[pos1];
    }
```

Version 2: avec une seule function thread à laquelle on transmet le début de la recherche.

2. On suppose maintenant que chaque thread transmet à son père le plus grand élément trouvé au lieu de transmettre sa position.

```
…
void* lire1(void* k ) {
   int i, pos=0;
   double *x=(double *)malloc(sizeof(double));
   double max=tab[0];
   for(i=0;i<N/2;i++) {
    if (tab[i]>max) {
       max=tab[i];
       pos=i;
    }
   }
   *x=max;
   pthread_exit((void *)x);
}
void* lire2(void* k ) {
   int i, pos=N/2;
   double *x=(double *)malloc(sizeof(double));
   double max=tab[N/2];
   for(i=N/2;i<N;i++) {
    if (tab[i]>max) {
       max=tab[i];
       pos=i;
    }
  }
  *x=max;
  pthread_exit((void *)x);
}
main () {
        double max;
        double *x1=(double *)malloc(sizeof(double));
        double *x2=(double *)malloc(sizeof(double));
        void *res1,*res2;
        pthread_t  th1, th2;
```

```
        pthread_create (&th1, NULL, lire1, NULL );
        pthread_create (&th2, NULL, lire2, NULL );
        pthread_join(th1,&res1);
        pthread_join(th2,&res2);
        x1=(double *)(res1);
        x2=(double *)(res2);
        max=*x2;
        if ( *x1>*x2)
                max=*x1;
}
```

3. On suppose maintenant que chaque thread transmet à son père le plus grand élément trouvé ainsi que sa position.

```
…
struct data {
        int position;
        double sup ;
};
void* lire1(void* k ) {
        int i, pos=0;
        struct data *x=(struct data *)malloc(sizeof(struct data));
        double max=tab[0];
        for(i=0;i<N/2;i++) {
                if (tab[i]>max) {
                        max=tab[i];
                        pos=i;
                }
        }
        x->sup=max; // ou (*x).sup=max;
        x->position=pos; //ou   (*x).position=pos;
        pthread_exit((void *)x);
}
void* lire2(void* k ) {
        int i;
        struct data *x=(struct data *)malloc(sizeof(struct data));
```

```
        double max=tab[N/2];
        int pos=N/2;
        for(i=N/2;i<N;i++) {
            if (tab[i]>max) {
                max=tab[i];
                pos=i;
            }
        }
        x->sup=max; // ou  (*x).sup=max;
        x->position=pos; // ou  (*x).position=pos;
        pthread_exit((void *)x);
}
main () {
        double max;
        struct data *x1=(struct data *)malloc(sizeof(struct data));
        struct data *x2=(struct data *)malloc(sizeof(struct data));
        void *res1, *res2;
        pthread_t  th1, th2;
        // initialisation du tableau tab
        pthread_create (&th1, NULL, lire1, NULL );
        pthread_create (&th2, NULL, lire2, NULL );
        pthread_join(th1,&res1);
        pthread_join(th2,&res2);
        x1=(struct data *)(res1);
        x2=(struct data *)(res2);
        max=x2->sup;
        if (x1->sup > x2->sup)
            max=x1->sup;
}
```

Exercice 4 :

1. *Soit « tab » un tableau de réels de taille « n » et soit « x » un réel donné. Ecrire une fonction « existe(double x, int n) » en langage C qui retourne la position du réel « x » dans le tableau « tab » si elle existe, sinon elle retourne « -1 ».*

2. *Ecrire un programme C dans lequel :*

- *le thread père crée un thread fils « th » en lui passant la valeur de « x ».*
- *le thread « th » cherche si « x » existe dans le tableau « tab » puis transmet au thread père la position du réel « x » dans le tableau « tab » si elle existe, sinon il transmet « -1 ».*
- *Le thread père affiche le message « x existe dans le tableau » ou « x n'existe pas dans le tableau » suivant la valeur reçu du thread « th ».*

3. *Modifier le programme précédent afin que la recherche s'effectue dans un sous-tableau du tableau « tab » (entre les indices « debut » et « fin ») :*

- *le thread père crée un thread fils « th » en lui passant les valeurs « x », « debut » et « fin ».*
- *le thread « th » cherche si « x » existe dans le tableau « tab » entre les indices « debut » et «fin» puis transmet au thread père la position du réel « x » si elle existe, sinon il transmet « -1 ».*
- *Le thread père affiche le message « x existe dans le sous-tableau » ou « x n'existe pas dans le sous-tableau » suivant la valeur reçu du thread « th ».*

Important

Le tableau « tab » ainsi que la taille « n » sont déclarées variables globales,
« debut » et « fin » sont des variables locales déclarées dans le thread principal.

Réponses :

1. On définit la fonction « existe() » qui retourne la position du réel « x » dans le tableau « tab » s'il existe, sinon elle retourne « -1 ».

```
int existe(double x, int n) {
   int i, pos=-1;
   for(i=0;i<n;i++) {
     if (x==tab[i]) {
        pos=i;
        break ;
     }
   }
   return pos;
}
```

2. On définit la fonction « existe() » qui sera exécutée par le thread « th ». A la fonction thread on passe un argument qui contient la valeur de « x ».

On suppose que la taille du tableau est N

```
# define  N 100
double tab[N] ;
int n=N;
void* existe(void* k ) { // la fonction thread.
// puisque l'argument « k » est de type « void * » on doit faire un transtypage
// vers le type « double * »
    double *x = (double *)k ;
    int i, pos=-1;
    for(i=0;i<n; i++) {
       if (*x==tab[i]) {
          pos=i;
          break ;
       }
    }
    pthread_exit((void *)pos);  // on transmet la valeur de « pos ».
}
main() {
    void *pos;
    int j;
    double x;
    pthread_t  thread1_id ;
    // initialisation du tableau tab de taille n
    printf("Entrer  x  :  ");
    scanf("%lf",&x);
// on crée le thread « th » pour exécuter la fonction « existe() » et on lui passe
// l'argument « x ».
    pthread_create (&th, NULL, &existe, &x );
    pthread_join(th,&pos);  // le thread principal récupère la valeur de pos
    j=(int )pos;   // puisque pos est de type void * alors on fait un transtypage
    if (j==-1)
       printf(" x=  %lf  n'existe pas dans le tableau tab \n",x);
    else
```

```
        printf(" la position de x=  %lf  dans tab est   %d   \n",x,j);
    }
```

3. Dans cette question la recherche s'effectue dans un sous-tableau du tableau « tab » limité par les indices « debut » et « fin ». Donc on passe à la fonction thread un argument de type « struct data » qui a deux champs de type int pour passer les indices « debut » et « fin » et un champ de type double pour passer la valeur de « x ».

```
struct  data {
    int debut, fin ;
    double v;
};
void* existe(void* k ) {
    struct data *p = (struct data *)k ;
    int i, pos=-1;
    for(i=(*p).debut; i<(*p).fin;i++) {
       if ((*p).v==tab[i]) {
          pos=i;
          break ;
       }
    }
    pthread_exit((void *)pos);
}
main() {
    void *pos;
    int j, d, f;
    double x;
    struct data r;
    pthread_t  th ;
    printf("Entrer le debut de la recherche  :  ");
    scanf("%d", &d);
    printf("Entrer la fin de la recherche  :  ");
    scanf("%d", &f);
    printf("Entrer  x  :  ");
    scanf("%lf",&x);
    r.debut=d;
```

```
        r.fin=f;
        r.v=x;
        pthread_create (&th, NULL, &existe, &r );
        pthread_join(th,&pos);
        j=(int )pos;
        if (j==-1)
           printf(" x=  %lf  n'existe pas dans le sous-tableau tab \n",x);
        else
           printf(" la position de x=  %lf  dans tab est   %d   \n",x,j);
    }
```

Exercice 5 :

Le but est de faire coopérer deux threads pour remplir, de manière concurrente, un tableau partagé « tab » par des entiers de type « int ». L'ordre d'écriture ainsi que le nombre d'entiers écrits par chaque thread sont imprévisibles.

Règles :

- *L'écriture dans tableau s'effectue à partir de la gauche, c'est à dire à partir de l'indice 0.*
- *L'écriture s'arrête lorsqu'on atteint la taille maximale du tableau.*
- *Les deux threads ne peuvent pas accéder au même indice pour écrire.*
- *Quand un thread tient le processeur pour écrire, il commence à partir du premier indice dans lequel on n'a pas encore écrit (remplissage du tableau de la gauche vers la droite).*

Écrire un programme dans lequel :

- *le thread principal crée deux threads « th1 » et « th2 »,*
- *le thread « th1 » écrit les valeurs 2 dans le tableau « tab »,*
- *le thread « th2 » écrit les valeurs 8 dans le tableau « tab »,*
- *quand les deux threads ne peuvent plus écrire (le tableau est rempli), le thread principal affiche les éléments du tableau.*

Réponses:

On définit une seule function thread et on exploite l'argument de la function thread pour passer la valeur à afficher (2 ou 8)

```
….
#define N 4000
int tab[N];
int ind=0;
pthread_t  th1, th2;
pthread_mutex_t  mutex1,mutex2;
void *lire (void* k);
main ( ) {
    int i, k1=2, k2=8;
    pthread_mutex_init (&mutex1,NULL);
    pthread_mutex_init (&mutex2,NULL);
    pthread_create (&th1, NULL, &lire, &k1) ;
    pthread_create (&th2, NULL, &lire, &k2) ;
    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
    for (i=0;i<N;i++)
       printf("tab[%d] =  %d  \n", i, tab[i]);
}
void *lire (void* k) {
    int *p=(int *)k;
    while (1) {
          pthread_mutex_lock (&mutex1);
          if (ind<N) {
                tab[ind]=*p;
                ind++;
          }
          else {
             pthread_mutex_unlock(&mutex1);
             pthread_exit(NULL);
          }
          pthread_mutex_unlock(&mutex1);
    }
}
```

Exercice 6 :

Soit « x » une variable partagée de type « int » initialisée à 0. Le but de cet exercice est de synchroniser l'accès à la variable partagée « x » par deux threads.

1. *Dans cette question on suppose qu'aucune synchronisation n'est imposée entre les deux threads. Ecrire un programme dans lequel le thread principal crée un thread fils « th1 ».*

- *Le thread principal modifie la variable « x » en utilisant les instructions suivantes:*

```
while (1){
  x= x+1;
}
```

- *Le thread fils « th1 » modifie la variable « x » en utilisant les instructions suivantes:*

```
while (1){
  x=3x+2;
}
```

2. *Dans cette question on synchronise l'accès à la variable « x » par les deux threads. On suppose que :*

- *l'accès à la variable « x » est exclusif : si un thread accède à la variable « x » l'autre thread ne peut pas y accéder.*
- *l'accès à la variable « x » est non alterné par les deux threads: un thread peut accéder plusieurs fois successives à la variable « x ».*

a. *Modifier le programme précédent en supposant que le premier thread qui accède à la variable « x » est imprévisible (soit le thread principal soit le thread fils « th1 »).*

b. *Modifier le programme précédent en supposant que c'est le thread principal qui accède le premier à la variable « x ».*

c. *Modifier le programme précédent en supposant que c'est le thread fils « th1 » qui accède le premier à la variable « x ».*

Réponses:

1.

```
int x=0;
pthread_t th1;
```

```
main ( ) {
    // le thread principal crée le thread « th1 » qui execute la fonction f1()
    pthread_create (&th1, NULL, f1, NULL) ;
    while (1){
       x=x+1;
    }
    pthread_join(th1,NULL);
}
void *f1 (void* k) { // la fonction exécutée par le thread « th1 »
    while (1) {
       x= 3*x+2;
    }
    pthread_exit(NULL);
}
```

2. On utilise les mutex pour synchroniser l'accès à la variable « x » par les deux threads en utilisant les opérations :

- P(m) pour verrouiller l'accès (tient le mutex m).
- V(m) pour libérer le mutex m.

2.a : Le premier thread qui accède à la variable « x » est imprévisible, donc les deux threads accèdent à la variable « x » de manière concurrente. Le thread qui tient le mutex c'est lui qui modifie la variable « x »

Solution théorique

Thread0 (thread maitre)	**Thread th1**

- initialise le mutex m
- ensuite, il crée le thread th1
- enfin, il exécute les instructions:
while (1){
P(m);
x=x+1;
V(m);
}
	- exécute les instructions :
	while (1){
	P(m);
	x=3x+2;
	V(m);
	}

Code en langage C

```
…
int x=0;
pthread_t  th1;
pthread_mutex_t mutex;
void *f1 (void* k);

main ( ) {
    // initialisation du mutex « mutex »
    pthread_mutex_init (&mutex,NULL);
    // creation du thread « th1 »
    pthread_create (&th1, NULL, f1, NULL) ;
    while (1){
       pthread_mutex_lock(&mutex);   // P(mutex)
       x=x+1;
       pthread_mutex_unlock(&mutex);  // V(mutex)
    }
    pthread_join(th1,NULL); // attend la terminaison de « th1 »
}

void *f1 (void* k) {  // fonction éxécutée par le thread « th1 »
    while (1) {
       pthread_mutex_lock(&mutex);   // P(mutex)
       x= 3*x+2;
       pthread_mutex_unlock(&mutex); // V(mutex)
    }
    pthread_exit(NULL);
}
```

2.b : Pour être sûr que c'est le thread principal qui accède le premier à la variable « x » alors il doit verrouiller le mutex avant de créer le thread fils.

Solution théorique

Thread0 (thread maitre)	Thread th1

- initialise le mutex m
-verrouille le mutex m (P(m)) avant de créer le thread pour commence le premier
- crée le thread th1
- exécute l'instruction
x=x+1
V(m)
- exécute les instructions:
while (1){
P(m);
x=x+1;
V(m);
}
	- exécute les instructions :
	while (1){
	P(m);
	x=3x+2;
	V(m);
	}

Code en langage C

```
…
int x=0;
pthread_t  th1;
pthread_mutex_t  mutex;
main ( ) {
    pthread_mutex_init (&mutex,NULL);
    // verrouiller le mutex
    pthread_mutex_lock(&mutex);
    pthread_create (&th1, NULL, f1, NULL) ;
    x=x+1;
    pthread_mutex_unlock(&mutex);
    while (1){
       pthread_mutex_lock(&mutex);
       x=x+1;
       pthread_mutex_unlock(&mutex);
    }
    pthread_join(th1,NULL);
}
```

```
void *f1 (void* k) {
    while (1) {
       pthread_mutex_lock(&mutex);
       x= 3*x+2;
       pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

2.c: Version1:

Pour être sûr que c'est le thread th1 qui accède le premier à la variable « x », le père doit verrouiller le mutex « m » avant la création du thread th1, ensuite attend que le mutex soit libérer par le thread th1 pour accéder à la variable « x ».

Solution théorique

Thread0 (thread maitre)	Thread th1

- verrouille le mutex m (P(m).
- crée le thread th1
- exécute les instructions:
while (1){
P(m);
x=x+1;
V(m);
}
	- exécute les instructions
	while (1){
	P(m);
	x=3x+2;
	V(m);
	}

Code en langage C

```
#include <stdio.h>
#include <pthread.h>
int x=0;
pthread_t th1;
pthread_mutex_t  m;
void *f1 (void* k);
main ( ) {
    pthread_mutex_init (&m, NULL);
    pthread_mutex_lock(&m);
    pthread_create (&th1, NULL, f1, NULL) ;
```

```
    while (1){
       pthread_mutex_lock(&m);
       x=x+1;
       pthread_mutex_unlock(&m);
    }
    pthread_join(th1,NULL);
}
void *f1 (void* k) {
    while (1) {
       pthread_mutex_lock(&m);
       x= 3*x+2;
       pthread_mutex_unlock(&m);
    }
    pthread_exit(NULL);
}
```

2.c: <u>Version2:</u> On peut aussi faire une attente active.

- Le père fait une attente active (il attend changement de la valeur de la variable globale test).
- Le thread fils accède le premier à la variable « x » et ensuite modifie la valeur de la variable globale test.

<u>Solution théorique</u>

Thread0 (thread maitre)	**Thread th1**

- soit test=0; une variable globale
- crée le thread th1
- faire une attente active
while (test==0);
- exécute les instructions:
while (1){
P(m);
x=x+1;
V(m);
}
	- Exécute les instructions
	P(m)
	x=3x+2
	test=1 ; // modifie la variable test
	V(m)
	- exécute les instructions
	while (1){
	P(m);
	x=3x+2;
	V(m);
	}

Exercice 7 :

Le but de cet exercice est de synchroniser l'accès à une variable partagée « v » par plusieurs threads. La variable « v » est modifiée par le thread principal (fonction « main() ») en utilisant les instructions suivantes :

```
i=1;
while (1){
        v+=i;
        i++ ;
}
```

N.B. Pour la synchronisation on utilise les sémaphores d'exclusion mutuelle (les Mutexs). On n'utilise pas les variables de condition.

1. *Ecrire un programme dans lequel :*

- *Le thread principal crée un thread fils « th1 », ensuite modifie la variable partagée « v » en utilisant les instructions ci-dessus.*
- *Le thread fils « th1 » affiche les différentes valeurs de la variable « v ».*

Les contraintes de synchronisation:

- *Le thread principal commence par modifier la variable « v ».*
- *Après chaque modification, le thread fils « th1 » affiche la valeur de la variable « v ».*
- *Le thread principal ne peut modifier à nouveau la variable « v » qu'après l'affichage de sa valeur par le thread fils « th1 ».*
- *Chaque valeur de « v » est affichée une seule fois par le thread « th1 ».*

2. *Dans cette question, on suppose que :*

- *Le thread principal crée deux threads fils « th1 » et « th2 », ensuite modifie la variable partagée « v » en utilisant les instructions ci-dessus.*
- *Les deux threads fils « th1 » et « th2 » affichent les différentes valeurs de la variable « v ».*

Les contraintes de synchronisation:

- *Le thread principal commence par modifier la variable « v ».*
- *Après chaque modification, les deux threads fils « th1 » et « th2 » accèdent de manière concurrente à la variable « v » pour afficher sa valeur.*
- *Le thread principal ne peut modifier à nouveau la valeur de la variable « v » qu'après l'affichage de sa valeur par ses threads fils « th1 » et « th2 ».*

- *Chaque valeur de « v » est affichée deux fois mais pas nécessairement par les deux threads. Dans le cas où le même thread affiche deux fois la même valeur de « v », l'autre thread n'affiche rien.*

3. *Refaire la question précédente en modifiant la dernière contrainte de synchronisation :*

- *Chaque valeur de « v » est affichée deux fois. Une fois par le thread « th1 » et une fois par le thread « th2 ».*

4. *Soit N une variable globale donnée, refaire les questions* ***2.*** *et* ***3.*** *en supposant que la variable « v » est modifiée par le thread principal (fonction « main() ») en utilisant les instructions suivantes:*

```
i=1;
while (i<N ){
        v+=i;
        i++ ;
```

Réponses:

1. **Principe :** lorsque le thread maitre effectue une itération, il libère le mutex pour que le thread th1 effectue une itération d'affichage. Lorsque le thread fils th1 effectue une itération d'affichage, il libère le mutex pour que le thread père passe à l'itération suivante et ainsi de suite.

Solution théorique

Thread0 (thread maitre)	Thread th1

- verrouille le mutex mutex1 : P(mutex1): pour commencer le premier
- crée le thread th1
- exécute les instructions
while (1){
P(mutex)
v+=i;
i++;
V(mutex1)
}
	- exécute les instructions :
	while (1) {
	P(mutex1);
	Affichage de v
	i++;
	V(mutex);
	}

Code en langage C

```
….
int v=0;
```

```
pthread_t th1;
pthread_mutex_t  mutex, mutex1;
void *lire1 (void* k);
main ( ) {
    int i=1;
    pthread_mutex_init (&mutex,NULL);
    pthread_mutex_init (&mutex1,NULL);
    // verrouiller le mutex «mutex1» avant de créer le thread pour être sûr que
    // c'est le thread maitre qui commence le premier
    pthread_mutex_lock (&mutex1);
    pthread_create (&th1, NULL, &lire1, NULL) ;
    while (1){
       pthread_mutex_lock (&mutex);
       v+=i;
       i++;
       pthread_mutex_unlock(&mutex1);
    }
    pthread_join(th1,NULL);
}
void *lire1 (void* k) { // exécuté par le thread
    int i=1;
    while (1) {
       pthread_mutex_lock (&mutex1);
       printf("v= %d \n",v);
       i++;
       pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

2. On définit une variable globale « nb ».

- Elle sera initialisée à zéro dans le thread maitre après chaque modification de la variable « v ».
- Elle sera incrémentée dans chaque thread après chaque affichage de « v ».

Donc, chaque thread, avant d'afficher « v » doit s'assurer qu'elle n'est pas déjà affichée 2 fois donc il doit vérifier que la valeur de « nb » est inférieure à 2. Puisque « nb » est globale, l'accès à la variable « nb » doit être protégé.

Code en langage C

```
….
int v;
int nb=0;
pthread_t th1, th2;
pthread_mutex_t mutex, mutex1;
main ( ) {
    int i=1;
    pthread_mutex_init (&mutex,NULL);
    pthread_mutex_init (&mutex1,NULL);
    pthread_mutex_lock (&mutex1);
    pthread_create (&th1, NULL, &lire1, NULL) ;
    pthread_create (&th2, NULL, &lire2, NULL) ;
    while (1){
        pthread_mutex_lock (&mutex);
        nb=0;
        v+=i;
        i++;
        pthread_mutex_unlock(&mutex1);
    }
    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
}
void *lire1 (void* k) {  // exécuté par le thread « th1 »
    while (1) {
      pthread_mutex_lock (&mutex1);
      printf("thread 1 v= %d \n",v);
      nb++;
      if (nb==2) {
          pthread_mutex_unlock(&mutex);
      }
      else
```

```
                pthread_mutex_unlock(&mutex1);
            }
            pthread_exit(NULL);
        }
        void *lire2 (void* k) {  // exécuté par le thread « th2 »
            while (1) {
               pthread_mutex_lock (&mutex1);
               printf("thread 2 v= %d \n",v);
               nb++;
               if (nb==2) {
                    pthread_mutex_unlock(&mutex);
               }
               else
                    pthread_mutex_unlock(&mutex1);
            }
            pthread_exit(NULL);
        }
```

3. Chaque valeur de « v » est affichée deux fois. Une fois par le thread « th1 » et une fois par le thread « th2 ».

Solution théorique

Thread0 (thread maitre)	Thread th1	Thread th2

- verrouiller les mutex (P(mutex1), P(mutex2)) pour commencer le premier.
- créer les thread th1 et th2
- exécuter les instructions
while (1){
P(mutex10);
P(mutex20);
v+=i;
i++;
V(mutex1)
V(mutex2);
}
	- exécuter les instructions
	while (1) {
	P(mutex1)
	Afficher v
	V(mutex10)
	}
		- exécuter les instructions
		while (1) {
		P(mutex2)
		Afficher v
		V(mutex20)
		}

Code en langage C

```
…
int v;
pthread_t th1, th2;
pthread_mutex_t mutex1,mutex2, mutex10,mutex20;
main ( ) {
    int i=1;
    // initialiser les mutex
    pthread_mutex_lock (&mutex1);
    pthread_mutex_lock (&mutex2);
    // création des threads
    while (1){
       pthread_mutex_lock (&mutex10);
       pthread_mutex_lock (&mutex20);
       v=i+100;
       i++;
       pthread_mutex_unlock(&mutex1);
       pthread_mutex_unlock(&mutex2);
    }
    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
}

void *lire1 (void* k) {
    while (1) {
          pthread_mutex_lock (&mutex1);
          printf("thread 1 v= %d \n",v);
          pthread_mutex_unlock(&mutex10);
    }
    pthread_exit(NULL);
}
void *lire2 (void* k) {
    while (1) {
          pthread_mutex_lock (&mutex2);
          printf("thread 2 v= %d \n",v);
```

```
            pthread_mutex_unlock(&mutex20);
        }
        pthread_exit(NULL);
    }
```

4. Remplacer la boucle while(1) par while (i<=N)

Exercice 8 :

Le but de cet exercice est de synchroniser l'accès aux variables partagées «v», «v1» et «v2» par plusieurs threads. On suppose que ses variables sont de type « int » et sont toutes initialisées à 0.

N.B. Pour la synchronisation on utilise les sémaphores d'exclusion mutuelle (les Mutexs). On n'utilise pas les variables de condition.

1. *Ecrire un programme dans lequel :*

- *Le thread principal crée deux threads « th1 », « th2 », ensuite modifie la variable « v » en utilisant les instructions suivantes :*

```
i=1;
while (1){
        v=v+v2+i;
        i++;
}
```

- *Le thread « th1 » modifie la variable « v1 » en utilisant les instructions suivantes:*

```
i=1;
while (1){
        v1= v+2*i;
        i++;
}
```

- *Le thread « th2 » modifie la variable « v2 » en utilisant les instructions suivantes:*

```
i=1;
while (1){
        v2=v1+3*i;
        i++;
}
```

Les contraintes de synchronisation:

- *Le thread principal commence par modifier la variable « v ».*

- *Après chaque modification de la variable « v », le thread « th1 » modifie la variable « v1 ».*
- *Après chaque modification de la variable « v1 », le thread « th2 » modifie la variable « v2 ».*
- *Après chaque modification de la variable «v2», le thread principal modifie à nouveau la variable «v».*

2. *Refaire la question précédente en supposant maintenant que le thread principal modifie la variable « v » en utilisant les instructions suivantes (pas de contrainte de dépendance entre le thread « th2 » et le thread principal) :*

```
i=1;
while (1){
        v=v+i;
        i++ ;
}
```

Réponses :

1. L'ordre d'exécution des threads est le suivant :
thread principal, suivi du thread th1, suivi du thread th2
est ainsi de suite.

Solution théorique

Thread0 (thread maitre)	Thread th1	Thread th2

- verouiller les mutex (P(mutex1) et P(mutex2)) pour que le thread maitre commence avant th1, et th1 commence avant th2
- créer les threads th1 et th2
- exécuter les instructions
while (1){
P(mutex)
Modifier v=f(v,v2)
V(mutex1)
}
	- exécuter les instructions
	while (1) {
	P(mutex1);
	Modifier v1=f1(v)
	V(mutex2)
	}
		- exécuter les instructions
		while (1) {
		P(mutex2)
		Modifier v2=f2(v1)
		V(mutex)
		}

Code en langage C

…

```
int v,v1,v2=0;
int test=0;
pthread_t th1, th2;
pthread_mutex_t mutex,mutex1,mutex2;
main ( ) {
    int i;
    // initialisation des mutex;
    pthread_mutex_lock(&mutex1); // pour que le thread0 commence avant th1
    pthread_mutex_lock(&mutex2);  // pour que th1 commence avant th2
    pthread_create (&th1, NULL, lire1, NULL) ;
    pthread_create (&th2, NULL, lire2, NULL) ;
    i=1;
    while (i<100){
        pthread_mutex_lock (&mutex);
        v=v+v2+i;
        printf("thread maitre v= %d \n",v);
        pthread_mutex_unlock(&mutex1);
        i++;
    }
    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
}
void *lire1 (void* k) {  // exécuté par le thread th1
    int i=1;
    while (1) {
        pthread_mutex_lock (&mutex1);
        printf("thread 1 v1= %d \n",v1);
        v1= v+2*i;
        pthread_mutex_unlock(&mutex2);
        i++;
    }
    pthread_exit(NULL);
}
void *lire2 (void* k) {   // éxecuté par le thread th2
    int i=1;
```

```
        while (1) {
            pthread_mutex_lock (&mutex2);
            v2=v1+3*i;
            printf("        thread 2 v2= %d \n",v2);
            pthread_mutex_unlock(&mutex);
            i++;
        }
        pthread_exit(NULL);
    }
```

2. On suppose maintenant qu'il n'y a pas de contrainte de dépendance entre le thread « th2 » et le thread principal:

Solution théorique

Thread0 (thread maitre)	**Thread th1**	**Thread th2**

- P(mutex10) et P(mutex2) pour garantir que le thread maitre commence avant le thread th1 et th1 avant th2
- créer les thread th1 et th2
- exécuter les instructions
while (1){
P(mutex)
Modifier v=f(v);
V(mutex10)
}
	- exécuter les instructions
	while (1) {
	P(mutex10)
	P(mutex12)
	Modifier v1= f1(v)
	V(mutex)
	V(mutex2)
	}
		- exécuter les instructions
		while (1) {
		P(mutex2)
		v2=f2(v1)
		V(mutex12)
		}

```
int v,v1=0,v2;
pthread_t th1, th2;
pthread_mutex_t  mutex, mutex1, mutex10, mutex12, mutex2;
main() {
    int i=1;
    // initialisation des mutex
   // verrouiller les mutex mutex10 et mutex2
    pthread_mutex_lock (&mutex10);
    pthread_mutex_lock (&mutex2);
```

```
    // Création des threads
    while (1){
        pthread_mutex_lock (&mutex);
        v=v+i;
        printf("thread maitre v= %d \n",v);
        pthread_mutex_unlock(&mutex10);
        i++;
    }
    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
}
void *lire1 (void* k) {  // exécuté par le thread th1
    int i=1;
    while (1) {
        pthread_mutex_lock (&mutex10);
        pthread_mutex_lock (&mutex12);
        v1= v+2*i;
        printf("thread 1 v1= %d \n",v1);
        pthread_mutex_unlock(&mutex);
        pthread_mutex_unlock(&mutex2);
        i++;
    }
    pthread_exit(NULL);
}
void *lire2 (void* k) {  // exécuté par le thread « th2 »
    int i=1;
    while (1) {
        pthread_mutex_lock (&mutex2);
        v2=v1+3*i;
        printf("        thread 2 v2= %d \n",v2);
        pthread_mutex_unlock(&mutex12);
        i++;
    }
    pthread_exit(NULL);
}
```

Exercice 9 :

Le but de cet exercice est de synchroniser l'accès aux variables partagées «v1» et «v2» par plusieurs threads. Les variables « v1 » et « v2 » sont modifiées par le thread principal (fonction « main() ») en utilisant les instructions suivantes:

```
i=1;
while (1){
        v1=2*i+1 ;
        v2=2*i+2 ;
        i++;
}
```

N.B. Pour la synchronisation on utilise les sémaphores d'exclusion mutuelle (les Mutexs). On n'utilise pas les variables de condition.

1. *Ecrire un programme dans lequel :*

- *Le thread principal crée deux threads «th1» et «th2», ensuite modifie les variables partagées «v1» et «v » en utilisant les instructions ci-dessus.*
- *Le thread « th1 » affiche les différentes valeurs de la variable «v1».*
- *Le thread « th2 » affiche les différentes valeurs de la variable «v2».*

Les contraintes de synchronisation:

- *Le thread principal commence par modifier les deux variables «v1» et «v2».*
- *Après chaque modification des deux variables «v1» et «v2», le thread « th1 » affiche «v1» et le thread «th2» affiche «v2». Chaque valeur est affichée une fois mais pas nécessairement dans un ordre précis.*
- *Le thread principal ne peut modifier à nouveau les valeurs des variables «v1» et «v2» qu'après l'affichage de leurs valeurs par les threads «th1» et «th2».*

2. *Refaire la question précédente en synchronisant l'affichage des deux threads fils de la manière suivante : après chaque modification des deux variables « v1 » et « v2 », le thread « th1 » accède en premier pour afficher « v1 » ensuite le thread « th2 » affiche « v2 ».*

Réponses:

1. L'affichage des valeurs v1 et v2 est imprévisible.

Solution théorique

Thread0 (thread maitre)	Thread th1	Thread th2

- Verrouiller les mutex : P(mutex1) et P(mutex2) pour garantir que le thread maitre commence avant les threads th1 et th2
- Créer les thread th1 et th2
- Exécuter les instructions
while (1){
P(mutex10);
P(mutex20);
v1=f(i)
v2=g(i)
V(mutex1);
V(mutex2);
}
	- Exécuter les instructions
	while (1) {
	P(mutex1)
	Afficher v1
	V(mutex10);
	}
		- Exécuter les instructions
		while (1) {
		P(mutex2)
		Afficher v2
		V(mutex20)
		}

Code en langage C

```
…
int v1,v2;
pthread_t th1, th2;
pthread_mutex_t mutex10,mutex20, mutex1, mutex2;

main ( ) {
    int i=1;
    // Initialiser les mutex
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    // créer les threads
    while (1){
       pthread_mutex_lock (&mutex10);
       pthread_mutex_lock (&mutex20);
       v1=2*i+1;
```

```
            v2=2*i+2;
            i++;
            pthread_mutex_unlock(&mutex1);
            pthread_mutex_unlock(&mutex2);
        }
        pthread_join(th1,NULL);
        pthread_join(th2,NULL);
}

void *lire1 (void* k) { // exécuté par le thread « th1 »
        while (1) {
            pthread_mutex_lock (&mutex1);
            printf("thread 1 v1= %d \n",v1);
            pthread_mutex_unlock (&mutex10);
        }
        pthread_exit(NULL);
}

void *lire2 (void* k) { // exécuté par le thread « th2 »
        while (1) {
            pthread_mutex_lock (&mutex2);
            printf("thread 2 v2= %d \n",v2);
            pthread_mutex_unlock (&mutex20);
        }
        pthread_exit(NULL);
}
```

2. On synchronise l'affichage des deux threads fils : à chaque itération le thread «th1» affiche « v1 » avant que le thread « th2 » affiche « v2 ».

Solution théorique :

Thread0 (thread maitre)	Thread th1	Thread th2

- Verrouiller le mutex : P(mutex1) et P(mutex2) pour garantir que le thread maitre commence avant le thread th1 et th1 commence avant le thread th2.
- Créer les thread th1 et th2
- Exécuter les instructions
while (1){
P(mutex)
v1=f(i)
v2=g(i)
i++;
V(mutex1)
}
	- Exécuter les instructions
	while (1) {
	P(mutex1)
	Afficher v1
	V(mutex2);
	}
		- Exécuter les instructions
		while (1) {
		P(mutex2)
		Afficher v2
		V(mutex)
		}

```
…
int v1,v2;
pthread_t th1, th2;
pthread_mutex_t mutex,mutex1,mutex2;
main ( ) {
    int i=1;
    // initialiser les mutex
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    // création des threads
    while (1){
       pthread_mutex_lock (&mutex);
       v1=2*i+1;
       v2=2*i+2;
       i++;
       pthread_mutex_unlock(&mutex1);
    }
```

```
    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
}
void *lire1 (void* k) { // exécuté par le thread « th1 »
    while (1) {
       pthread_mutex_lock (&mutex1);
       printf("thread 1:    v1= %d \n",v1);
       pthread_mutex_unlock(&mutex2);
    }
    pthread_exit(NULL);
}
void *lire2 (void* k) { // exécuté par le thread « th2 »
    while (1) {
       pthread_mutex_lock (&mutex2);
       printf("          thread 2:  v2= %d \n",v2);
       pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

Exercice 10:

Le but de cet exercice est de synchroniser l'accès aux variables partagées «x», «y» et «z» par trois threads. On suppose que ces variables sont de type « int » et sont toutes initialisées à 0.

N.B. Pour la synchronisation on n'utilise pas les variables de condition.

1. *Dans cette question on suppose qu'aucune synchronisation n'est imposée entre les différents threads. Ecrire un programme dans lequel le thread principal crée deux threads «th1», «th2».*

- *Le thread principal modifie la variable « x » en utilisant les instructions suivantes:*

```
while (1){
        x=x+1;
}
```

- *Le thread « th1 » modifie la variable « y » en utilisant les instructions suivantes:*

```
while (1){
        y=y+x;
}
```

- *Le thread « th2 » modifie la variable « z » en utilisant les instructions suivantes:*

```
while (1){
        z=z+2*x ;
}
```

2. *Dans cette question on synchronise l'accès aux variables partagées « x », « y » et « z » par les trois threads. Modifier le programme précédent en respectant les contraintes de synchronisation suivantes:*

- *Le thread principal commence par modifier la variable « x ».*
- *Après chaque modification de la variable « x », les threads « th1 » et « th2 » modifient les variables « y » et « z ».*
- *Après chaque modification des variables « y » et « z », le thread principal modifie à nouveau la variable « x ».*

Réponses :

1. Il n'y a pas de synchronisation entre les threads

```
#include <stdio.h>
#include <pthread.h>
int x=0, y=0, z=0;
pthread_t th1, th2;
void *f1 (void* k);
void *f2 (void* k);
main ( ) {
    pthread_create (&th1, NULL, f1, NULL) ;
    pthread_create (&th2, NULL, f2, NULL) ;
    while (1){
       x=x+1;
    }
```

```
        pthread_join(th1,NULL);
        pthread_join(th2,NULL);
    }
    void *f1 (void* k) {
        while (1) {
           y= y+x;
        }
        pthread_exit(NULL);
    }
    void *f2 (void* k) {
        while (1) {
           z= z+2*x;
        }
        pthread_exit(NULL);
    }
```

2. On synchronise l'accès aux variables partagées « x », « y » et « z ». Après chaque modification de « x » par le thread maitre, les threads « th1 » et « th2 » modifient les variables « y » et « z ».
On suppose que c'est le thread maitre qui commence le premier à modifier « x ».

Solution théorique

Thread0 (thread maitre)	**Thread th1**	**Thread th2**

- verrouille les mutex mutex1et mutex2 pour garantir que le thread maitre commence avant les threads th1 et th2
- crée les thread th1 et th2
- exécute les instructions
while (1){
P(mutex01)
P(mutex02)
x=x+1
V(mutex1)
V(mutex2)
}
	- exécute les instructions
	while (1) {
	P(mutex1)
	y= y+x
	V(mutex01)
	}
		- exécute les instructions
		while (1) {
		P(mutex2)
		z= z+2*x;
		V(mutex02)
		}

Code en langage C

```
…
pthread_mutex_t  mutex1, mutex2, mutex01, mutex02;
main ( ) {
    // initialiser les mutex
    // verrouiller les mutex  mutex1 et mutex2
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    // créer les threads
    while (1){
       pthread_mutex_lock(&mutex01);
       pthread_mutex_lock(&mutex02);
       x=x+1;
       pthread_mutex_unlock(&mutex1);
       pthread_mutex_unlock(&mutex2);
    }
    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
}
void *f1 (void* k) {  // exécuté par le thread « th1 »
    while (1) {
       pthread_mutex_lock(&mutex1);
       y= y+x;
       pthread_mutex_unlock(&mutex01);
    }
    pthread_exit(NULL);
}
void *f2 (void* k) {  // exécuté par le thread « th2 »
    while (1) {
       pthread_mutex_lock(&mutex2);
       z= z+2*x;
       pthread_mutex_unlock(&mutex02);
    }
    pthread_exit(NULL);
}
```

Exercice 11 :

Le but de cet exercice est de synchroniser l'accès aux variables partagées « v », « v1 », « v2 » et « v3 » par plusieurs threads. On suppose que ces variables sont de type « int » et sont modifiées par les codes C suivants où f(), f1(), f2() et f3() sont des fonctions données.

Code0 : *while (1){* *v=f(v2,v3);* *}*	***Code2:*** *while (1){* *v2=f2(v);* *}*	***Code4:*** *while (1){* *v1=f1(v);* *v3=f3(v1);* *}*
Code1: *while (1){* *v1=f1(v);* *}*	***Code3:*** *while (1){* *v3=f3(v1)* *}*	***Code5:*** *while (1){* *v2=f2(v);* *v3=f3(v1);* *}*

Les contraintes de synchronisation pour chaque itération:

- *Après chaque modification de la variable « v », les variables « v1 » et « v2 » peuvent être modifiées simultanément.*
- *Après chaque modification de la variable « v1 », on modifie la variable « v3 »*
- *Après chaque modification des variables « v2 » et « v3 » on modifie à nouveau la variable « v ».*

On suppose qu'on commence par modifier la variable « v ».

1.*Ecrire un programme C dans lequel :*

- *Le thread principal crée trois threads « th1 », « th2 » et « th3 », ensuite modifie la variable « v » en utilisant le code « Code0 ».*
- *Le thread « th1 » modifie la variables « v1 » en utilisant le code « Code1 ».*
- *Le thread « th2 » modifie la variables « v2 » en utilisant le code « Code2 ».*
- *Le thread « th3 » modifie la variable « v3 » en utilisant le code « Code3 ».*

2.*Modifier le programme précédent en supposant maintenant que :*

- *Le thread principal crée deux threads « th1 » et « th2 », ensuite modifie la variable « v » en utilisant le code « Code0 »*

- *Le thread « th1 » modifie les variables «v1» et «v3» en utilisant le code « Code4 ».*
- *Le thread « th2 » modifie la variable « v2 » en utilisant le code « Code2 ».*

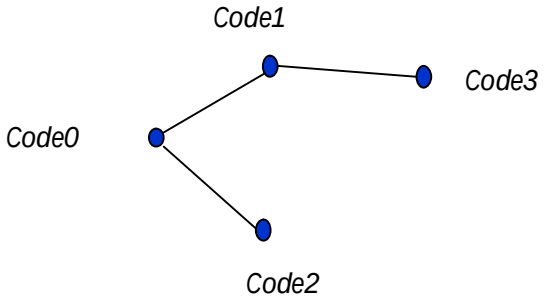
3.*Modifier le programme précédent en supposant maintenant que :*

- *Le thread principal crée deux threads « th1 » et « th2 », ensuite modifie la variable « v » en utilisant le code « Code0 ».*
- *Le thread « th1 » modifie la variable « v1 » en utilisant le code « Code1 ».*
- *Le thread « th2 » modifie les variables «v2» et «v3» en utilisant le code « Code5 ».*

N.B. Pour la synchronisation on utilise les sémaphores d'exclusion mutuelle (les Mutexs). On n'utilise pas les variables de condition.

Réponses :

1. D'après les contraintes de synchronisation entre les threads, le graphe de dépendance des threads, à chaque itération, peut être schématisé comme suit:



Principe:

- le père exécute une itération du «Code0», ensuite libère deux mutex (m1 et m2) pour que les deux threads th1 et th2 exécutent respectivement une itération «Code1» et une itération du «Code2».
- après libération du mutex m2, le thread th2 exécute une itération du « Code2 » et ensuite libère le mutex m02 pour que le père puisse passer à l'itération suivante.
- après libération du mutex m1, le thread th1 exécute une itération du « Code1 » et ensuite libère le mutex m13 pour que le thread 3 puisse exécuter une itération du «Code3».
- après libération du mutex m13, le thread th3 exécute une itération du « Code3 » et ensuite libère le mutex m03 pour que le père puisse passe à l'itération suivante.

- après libération des mutex m02 (libéré par th2) et m03 (libéré par th3), le thread père passe à l'itération suivante.

Le thread père th :

- verrouille les mutex m1 et m2 (P(m1) et P(m2)), avant la création des thread th1 et th2, pour commencer le premier à modifier la variable « v » avant que les threads « th1 » et « th2 » modifient les variables « v1 » et « v2 ».
- verrouille le mutex m13 (P(m13)) avant la création du thread th3 pour éviter que le thread th3 commence par modifier la variable « v3 » avant que le thread th1 modifie la variable « v1 » car « v3 dépend de « v1 ».
- création de trois threads th1, th2 et th3.

Les threads exécutent les instructions suivantes:

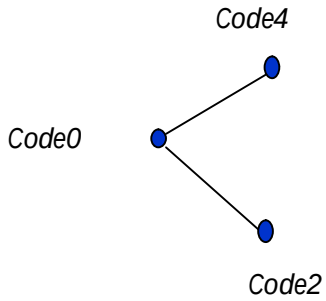
Thread père	**th1**	**th2**	**th3**
while (1) {	while(1) {	while(1) {	while (1) {
P(m02)	p(m1)	p(m2)	p(m13)
P(m03)			
v=f(v2,v3);	v1=f1(v);	v2=f2(v);	v3=f3(v1);
V(m1)	V(m13)	V(m02) ;	V(m03)
V(m2)			
}	}	}	}

Code en langage C

```
#include <stdio.h>
#include <pthread.h>
int v=0, v1=0, v2=0, v3=0;
pthread_t th1, th2, th3;
pthread_mutex_t  m1, m2, m13, m02, m03;
// déclaration des fonctions threads
main ( ) {
        // initialisation des mutex
        // verrouiller les mutex m1, m2 et m13
        pthread_mutex_lock(&m1);
        pthread_mutex_lock(&m2);
        pthread_mutex_lock(&m13);
```

```
        // création des threads
        while (1){
            pthread_mutex_lock(&m03);
            pthread_mutex_lock(&m02);
            v=f(v2,v3);
            pthread_mutex_unlock(&m1);
            pthread_mutex_unlock(&m2);
        }
        // attend la terminaison des threads
}
void *g1 (void* k) {  // exécuté par le thread « th1 »
        while (1) {
                pthread_mutex_lock(&m1);
                v1=f1(v);
                pthread_mutex_unlock(&m13);
        }
        pthread_exit(NULL);
}
void *g2 (void* k) {  // exécuté par le thread « th2 »
        while (1) {
            pthread_mutex_lock(&m2);
            v2=f2(v);
            pthread_mutex_unlock(&m02);
        }
        pthread_exit(NULL);
}
void *g3 (void* k) {  // exécuté par le thread « th3 »
        while (1) {
            pthread_mutex_lock(&m13);
            v3=f3(v1);
            pthread_mutex_unlock(&m03);
        }
        pthread_exit(NULL);
}
```

2. D'après les contraintes de synchronisation entre les threads, le graphe de dépendance des threads, à chaque itération, peut être schématisé comme suit:



Principe:

- le père exécute une itération du « Code0 », ensuite libère deux mutex (m1 et m2) pour que les deux threads th1 et th2 exécutent respectivement une itération du « Code1 » et une itération du « Code4 ».
- après libération du mutex m2, le thread th2 exécute une itération du « Code2 » et ensuite libère le mutex m02 pour que le père puisse passer à l'itération suivante.
- après libération du mutex m1, le thread th1 exécute une itération du « Code4 » et ensuite libère le mutex m02 pour que le thread père puisse passer à l'itération suivante.
- après libération des mutex m01 (libéré par th1) et m02 (libéré par th2), le thread père passe à l'itération suivante.

Le thread père th :

- verrouille les mutex m1 et m2 (P(m1) et P(m2)), avant la création des thread th1 et th2, pour commencer le premier à modifier la variable « v » avant que les threads « th1 » et « th2 » modifient les variables « v1 », « v2 » et « v3 ».
- création de trois threads th1, th2 et th3.

Les threads exécutent les instructions suivantes:

th	**th1**	**th2**
while (1) {	while(1) {	while(1) {
P(m01)	p(m1)	p(m2)
P(m02)		
v=f(v2,v3);	v1=f1(v);	v2=f2(v);
	v3=f3(v1);	
V(m1)	V(m01)	V(m02) ;
V(m2)		
}	}	}

Code C

```
…
pthread_mutex_t  m1, m2, m01, m02;
main ( ) {
        // initialisation des mutex
        // verrouiller les mutex m1 et m2
        pthread_mutex_lock(&m1);
        pthread_mutex_lock(&m2);
        // création des threads th1 et th2
          while (1){
                pthread_mutex_lock(&m01);
                pthread_mutex_lock(&m02);
                v=f(v2,v3);
                pthread_mutex_unlock(&m1);
                pthread_mutex_unlock(&m2);
        }
        // attendre la terminaison des threads
}
void *g1 (void* k) {
        while (1) {
                pthread_mutex_lock(&m1);
                v1=f1(v);
                v3=f3(v1);
                pthread_mutex_unlock(&m01);
        }
        pthread_exit(NULL);
}
void *g2 (void* k) {
        while (1) {
                pthread_mutex_lock(&m2);
                v2=f2(v);
                pthread_mutex_unlock(&m02);
        }
        pthread_exit(NULL);
}
```

3. D'après les contraintes de synchronisation entre les threads, on remarque que le « Code1 » dépend du « Code0 ». Le « Code5 » contient deux parties : une dépend du « Code0 » et l'autre du « Code1 ». Donc le graphe de dépendance des threads, à chaque itération, peut être schématisé comme suit:

Principe:

- le père exécute une itération du « Code0 », ensuite libère deux mutex (m1 et m2) pour que les deux threads th1 et th2 exécutent respectivement une itération du « Code1 » et la première partie d'une itération du « Code5 ».
- après libération du mutex m1, le thread th1 exécute une itération du « Code1 » et ensuite libère le mutex m12 pour que le thread2 puisse modifier la deuxième partie d'une itération du « Code5 ».
- après libération du mutex m2, le thread th2 exécute la première partie d'une itération du « Code5 » et après la libération du mutex m12, il exécute la deuxième partie et ensuite libère le mutex m pour que le thread père puisse passer à l'itération suivante.
- après libération des mutex m (libéré par th2), le thread père passe à l'itération suivante.

Le thread père th :

- vérouille les mutex m1 et m2 (P(m1) et P(m2)), avant la création des thread th1 et th2, pour commencer le premier à modifier la variable « v » avant que les threads « th1 » et « th2 » modifient les variables « v1 », « v2 » et « v3 ».
- vérouille le mutex m12 (P(m12)), avant la création du thread th2 pour éviter que le thread th2 commence par modifier la variable « v3 » avant que le thread th1 modifie la variable « v1 » car « v3 dépend de « v1 » (on a v3=f3(v1)).
- création de trois threads th1 et th2.

Les threads exécutent les instructions suivantes:

th

```
while (1) {
    P(m )
    v=f(v2,v3);


    V(m1)
    V(m2)
}
```

th1

```
while(1) {
        p(m1)
        v1=f1(v);

        V(m12)
}
```

th2

```
while(1) {
        p(m2)
        v2=f2(v);
        P(m12)
        v3=f3(v1);
        V(m) ;
}
```

Code en langage C

```
…
main ( ) {
        // initialisation des mutex
        // verrouiller les mutex
        pthread_mutex_lock(&m1);
        pthread_mutex_lock(&m2);
        pthread_mutex_lock(&m12);
        // création des thread th1 et th2
        while (1){
                pthread_mutex_lock(&m02);
                v=f(v2,v3);
                pthread_mutex_unlock(&m1);
                pthread_mutex_unlock(&m2);
        }
        // attendre la terminaison des threads
}
void *g1 (void* k) {
        while (1) {
                pthread_mutex_lock(&1);
                v1=f1(v);
                pthread_mutex_unlock(&m12);
        }
        pthread_exit(NULL);
}
void *g2 (void* k) {
        while (1) {
                pthread_mutex_lock(&m2);
                v2=f2(v);
                pthread_mutex_lock(&m12);
                v3=f3(v1);
                pthread_mutex_unlock(&m02);
        }
        pthread_exit(NULL);
}
```

Exercice 12 :

Le but de cet exercice est de synchroniser l'accès aux variables partagées « v », « v1 », « v2 » et « v3 » par plusieurs threads. On suppose que ces variables sont de type « int » et sont modifiées par les codes C suivants où f(), f1(), f2(), f3(), g1(), g2() et g3() sont des fonctions données.

Code0 : *while (1){* *v=f(v1,v2,v3);* *}*	***Code2****:* *while (1){* *v2=f2(v);* *}*	***Code4****:* *while (1){* *v=g1(v3);* *}*	***Code6****:* *while (1){* *v=g3(v1,v2* *);* *}*
Code1*:* *while (1){* *v1=f1(v);* *}*	***Code3****:* *while (1){* *v3=f3(v);* *}*	***Code5****:* *while (1){* *v3=g2(v1,v2* *);* *}*	

1. *Ecrire un programme C dans lequel :*

- *Le thread principal crée trois threads « th1 », « th2 » et « th3 », ensuite modifie la variable «v» en utilisant le code « Code0 ».*
- *Le thread « th1 » modifie la variable « v1 » en utilisant le code « Code1 ».*
- *Le thread « th2 » modifie la variable « v2 » en utilisant le code « Code2 ».*
- *Le thread « th3 » modifie la variable « v2 » en utilisant le code « Code3 ».*

Pour chaque itération, les contraintes de synchronisation suivantes doivent être respectées:

- *On commence par modifier la variable « v ».*
- *Après chaque modification de la variable « v », on modifie les variables « v1 », « v2 et « v3 ».*
- *Après chaque modification des variables « v1 », « v2 » et « v3 », on modifie à nouveau la variable « v ».*

2. *Modifier le programme de la question précédente en supposant maintenant qu'on commence par modifier les variables « v1 », « v2 » et « v3 ».*

3. *Modifier le programme de la question 1 en supposant maintenant que :*

- *Le thread principal modifie la variable « v » en utilisant le code « Code4 ».*

- *Le thread « th3 » modifie la variable « v3 » en utilisant le code « Code5 ».*

Pour chaque itération, les contraintes de synchronisation suivantes doivent être respectées:

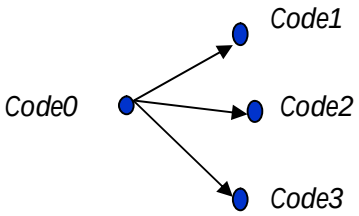
- *On commence par modifier la variable « v ».*
- *Après chaque modification de la variable « v », on modifie les variables « v1 » et « v2 ».*
- *Après modification des variables « v1 » et « v2 », on modifie la variable « v3 ».*
- *Après modification de la variable « v3 », on modifie à nouveau la variable « v ».*

4. *Modifier le programme de la question 3 en supposant maintenant qu'on commence par modifier la variable « v3 ».*

N.B. ***Pour la synchronisation on utilise les sémaphores d'exclusion mutuelle (les Mutexs). On n'utilise pas les variables de condition.***

Réponses:

1. D'après les contraintes de synchronisation entre les threads, on remarque que le « Code1 », « Code2 » et « Code3 » dépendent du « Code1 » et que les codes « Code1 », « Code2 » et « Code3 » sont indépendants. Donc le graphe de dépendance des threads, à chaque itération, peut être schématisé comme suit:



Dans cette question, on suppose que c'est le thread principal qui commence le premier à modifier la variable « v ».

Principe:

- le père exécute une itération du « Code0 », ensuite libère les mutex (m1, m2 et m3) pour que les threads th1, th2 et th3 exécutent respectivement une itération du « Code1 », une itération du « Code2 » et une itération du « Code3 ».
- après libération du mutex m1, le thread th1 exécute une itération du « Code1 » et ensuite libère le mutex m01 pour que le thread père puisse passer à l'itération suivante.
- après libération du mutex m2, le thread th2 exécute une itération du « Code2 » et ensuite libère le mutex m02 pour que le thread père puisse passer à l'itération suivante.
- après libération du mutex m3, le thread th3 exécute une itération du « Code3 » et ensuite libère le mutex m03 pour que le thread père puisse passer à l'itération suivante.
- après libération des mutex m01, m02 et m03, le thread père passe à l'itération suivante.

Le thread père th :

- verrouille les mutex m1, m2 et m3 (P(m1), P(m2) et P(m3)), avant la création des thread th1, th2 et th3, pour commencer le premier à modifier la variable « v » avant que les threads « th1 », « th2 » et « th3 » modifient les variables « v1 », « v2 » et « v3 ».
- création des threads th1, th2 et th3

Les threads exécutent les instructions suivantes:

th	**th1**	**th2**	**th3**
while (1) {	while(1) {	while(1) {	while (1) {
p(m01)	p(m1)	p(m2)	P(m3)
p(m02)			
p(m03)			
v=f(v1,v2,v3);	v1= f1(v);	v2=f2(v);	v3=f3(v);
V(m1)	V(m01) ;	V(m02) ;	V(m03) ;
V(m2)			
V(m3)			
}	}	}	}

Programme en langage C :

```
#include <stdio.h>
#include <pthread.h>
pthread_t th1, th2, th3;
pthread_mutex_t  m1, m2, m3, m01, m02, m03;
// déclaration des fonctions threads
main ( ) {
        // initialisation des mutex m1, m2, m3, m01, m02 et m03
        // verrouiller les mutex m1, m2 et m3
        pthread_mutex_lock(&m1);
        pthread_mutex_lock(&m2);
        pthread_mutex_lock(&m3);
        // création des threads th1, th2 et th3
        while (1){
                pthread_mutex_lock(&m01);
                pthread_mutex_lock(&m02);
                pthread_mutex_lock(&m03);
                v=f(v1,v2,v3);
                pthread_mutex_unlock(&m1);
                pthread_mutex_unlock(&m2);
                pthread_mutex_unlock(&m3);
        }
        // attendre la terminaison des threads
}
void *g1 (void* k) {
        while (1) {
                pthread_mutex_lock(&m1);
                v1=f1(v);
                pthread_mutex_unlock(&m01);
        }
        pthread_exit(NULL);
}
void *g2 (void* k) {
        while (1) {
                pthread_mutex_lock(&m2);
```

```
            v2=f2(v);
            pthread_mutex_unlock(&m02);
        }
        pthread_exit(NULL);
}
void *g3 (void* k) {
        while (1) {
            pthread_mutex_lock(&m3);
            v3=f3(v);
            pthread_mutex_unlock(&m03);
        }
        pthread_exit(NULL);
}
```

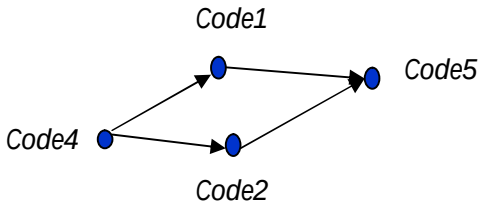
2. Pour que les threads th1, th2 et th3 commencent avant le thread père, le père doit verrouiller les mutex m01, m02 et m03 avant la création des threads th1, th2 et th3 au lieu de verrouiller les mutex m1, m2 et m3 dans la solution de la question 1.

Programme C

```
…
main ( ) {
        …
        // la partie à modifier para rapport au programme précédent
        pthread_mutex_lock(&m01);
        pthread_mutex_lock(&m02);
        pthread_mutex_lock(&m03);
        …
        // le reste ne change pas
```

3. D'après les contraintes de synchronisation entre les threads, on remarque que :
le « Code1 » et « Code2 » dépendent du « Code4 »
le « Code5 » dépend des codes « Code1 » et « Code2 »
le « Code4 » dépend du « Code5 »

Donc le graphe de dépendance des threads, à chaque itération, peut être schématisé comme suit:



Principe:

- le père exécute une itération du « Code4 », ensuite libère les mutex (m1, m2) pour que les threads th1, th2 exécutent respectivement une itération du « Code1 » et une itération du « Code2 ».
- après libération du mutex m1, le thread th1 exécute une itération du « Code1 » et ensuite libère le mutex m13 pour que le thread th3 exécute le code « Code5 ».
- après libération du mutex m2, le thread th2 exécute une itération du « Code2 » et ensuite libère le mutex m23 pour que le thread th3 exécute une itération « Code5 ».
- après libération des mutex m13 et m23, le thread th3 exécute une itération du « Code5 » et ensuite libère le mutex m pour que le thread père puisse passer à l'itération suivante.
- après libération du mutex m, le thread père passe à l'itération suivante.

- Pour que le père commence le premier, il doit verrouiller les mutex m1, m2. avant la création des threads th1 et th2.
- pour garantir que « th3 » commence après les threads th1 et th2, le père doit aussi verrouiller les mutex m13, m23 avant la création du thread th3**.**

Le thread père th :

- verrouille les mutex m1, m2, m13 et m23 (P(m1), P(m2), P(m13) et P¨(m23)).
- crée trois threads th1, th2 et th3

Les threads exécutent les instructions suivantes:

th	th1	th2	th3
while (1) {	while(1) {	while(1) {	while (1) {
p(m)	p(m1)	p(m2)	P(m13)
			P(m23)
v=g1(v3);	v1= f1(v);	v2=f2(v);	v3=g(v1,v2) ;
V(m1)	V(m13) ;	V(m23) ;	V(m) ;
V(m2)			
}	}	}	}

Code en langage C

```
…
main ( ) {
        // initialisation des mutex m, m1, m2, m13 et m23
        // verrouiller les mutex m1, m2, m13 et m23
        pthread_mutex_lock(&m1);
        pthread_mutex_lock(&m2);
        pthread_mutex_lock(&m13);
        pthread_mutex_lock(&m23);
        // création des threads
        while (1){
                pthread_mutex_lock(&m);
                v=f(v3);
                pthread_mutex_unlock(&m1);
                pthread_mutex_unlock(&m2);
        }
        // attend la terminaison des threads
}
void *g1 (void* k) {
        while (1) {
                pthread_mutex_lock(&m1);
                v1=f1(v);
                pthread_mutex_unlock(&m13);
        }
        pthread_exit(NULL);
}
```

```
void *g2 (void* k) {
        while (1) {
                pthread_mutex_lock(&m2);
                v2=f2(v);
                pthread_mutex_unlock(&m23);
        }
        pthread_exit(NULL);
}
void *g3 (void* k) {
        while (1) {
                pthread_mutex_lock(&m13);
                pthread_mutex_lock(&m23);
                v3=f3(v1,v2);
                pthread_mutex_unlock(&m);
        }
        pthread_exit(NULL);
}
```

4. Pour que le thread th3 commence le premier, le père doit verrouiller :

- Le mutex m1 (P(m1)) avant la création du thread « th1 » pour garantir que « th3 » commence avant « th1 ».
- Le mutex m2 (P(m2)) avant la création du thread « th2 » pour garantir que « th3 » commence avant « th2 ».
- Le mutex m (P(m)) avant la boucle "while" pour garantir que « th3 » commence avant le thread principal (thread « th »).

Programme en langage C

```
main ( ) {
        …
        // la partie à modifier para rapport au programme précédent
        pthread_mutex_lock(&m);
        pthread_mutex_lock(&m1);
        pthread_mutex_lock(&m2);
        …
        // le reste ne change pas
```