Noyau d'un système d'exploitation
INF2610

Séance de révision

Département de génie informatique et génie logiciel

POLYTECHNIQUE MONTRÉAL
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

Automne 2016

Séance de révision

Généralités

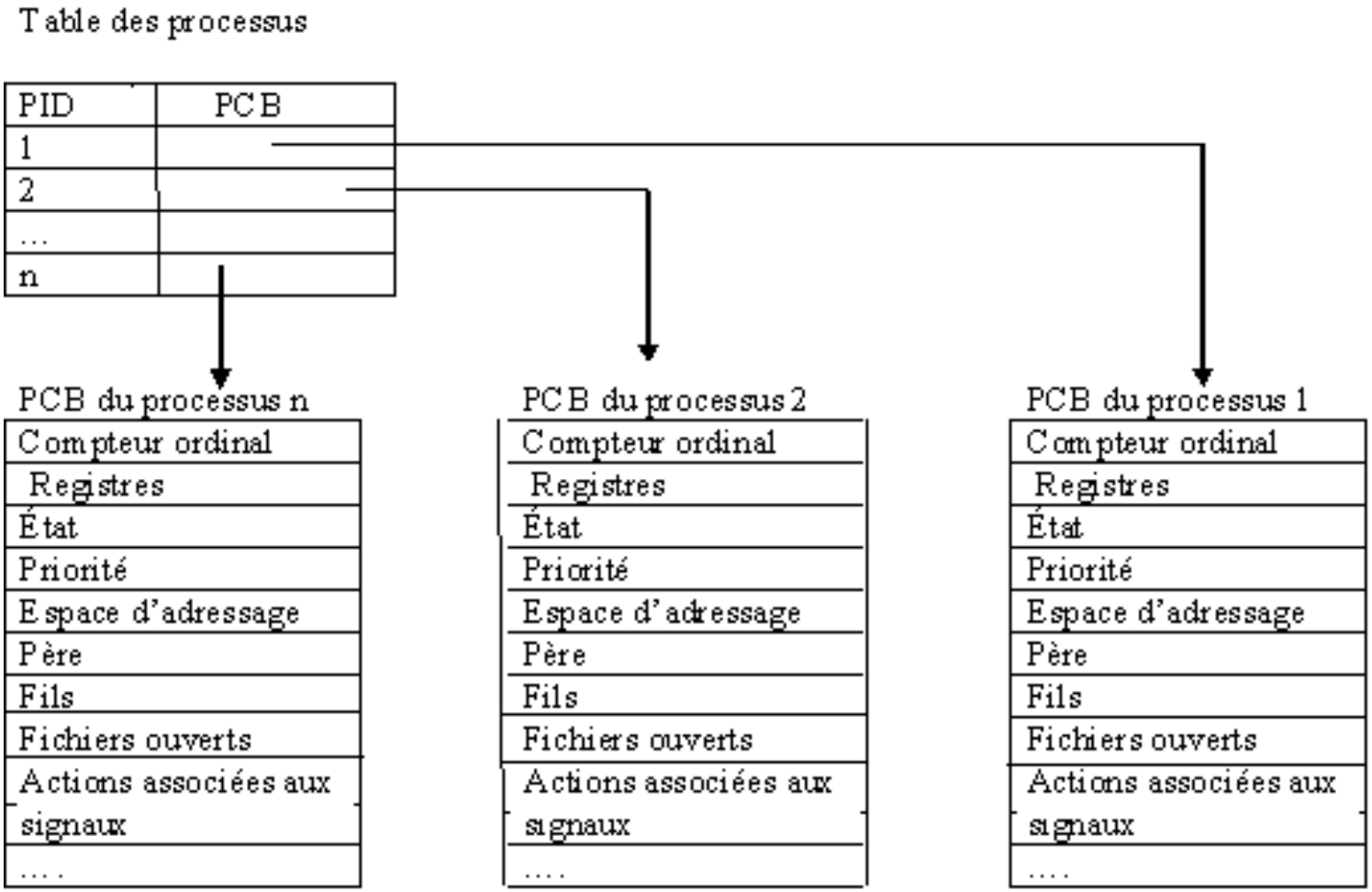
Processus & Threads

Tubes de communication & signaux

Synchronisation (sans les moniteurs)

Exercices

Processus = Programme en cours d'exécution



Processus (Linux – Unix)

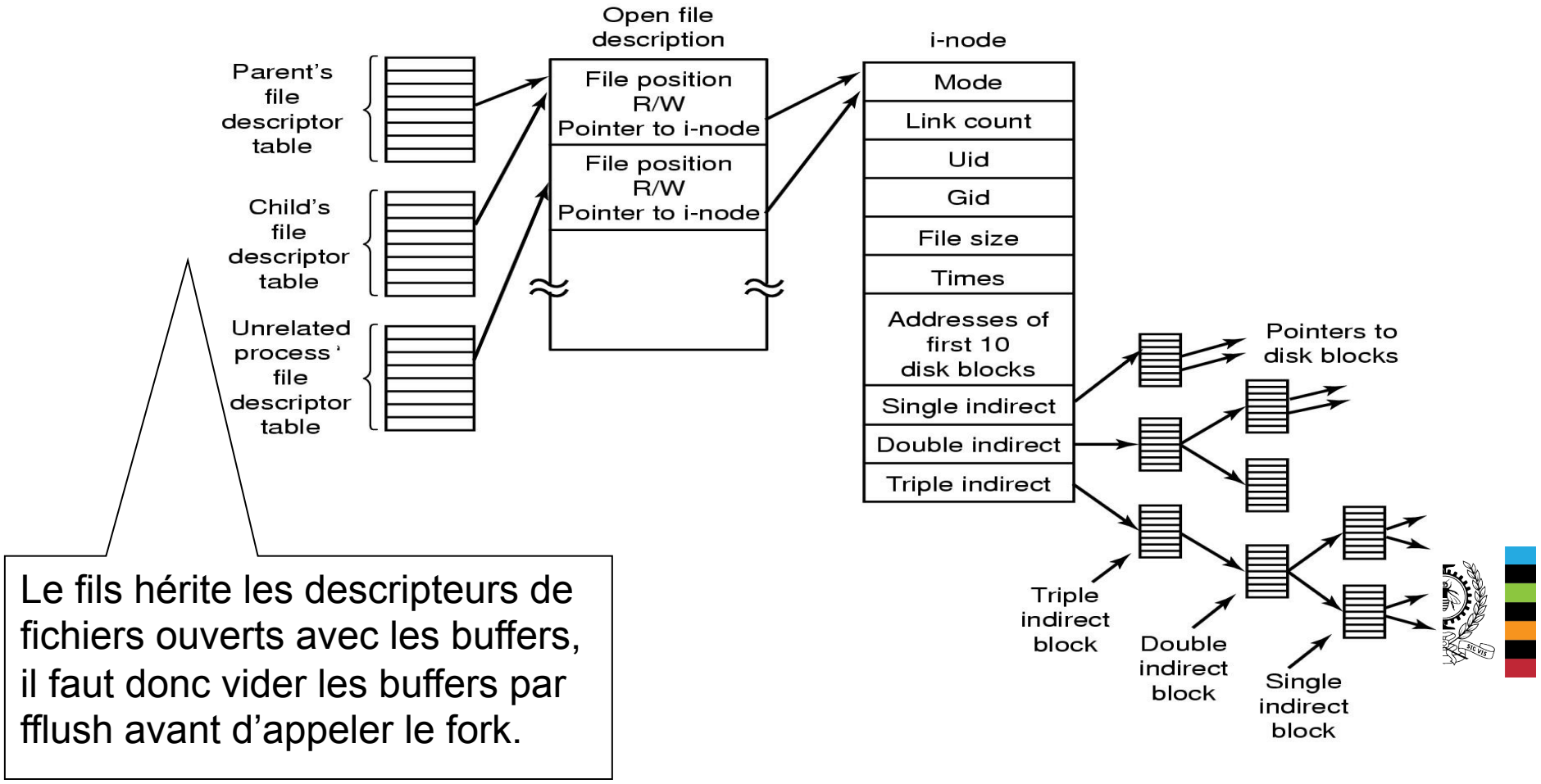
- Création par duplication (Copy-On-Write) : pid_t fork();
- Changement de code :

  int execlp(const char *file, const char *argv, …….);
- Attente ou vérification de la terminaison d'un fils

  int waitpid (pid_t pid, int * status, int options);
- Terminaison normale : void exit(int status);

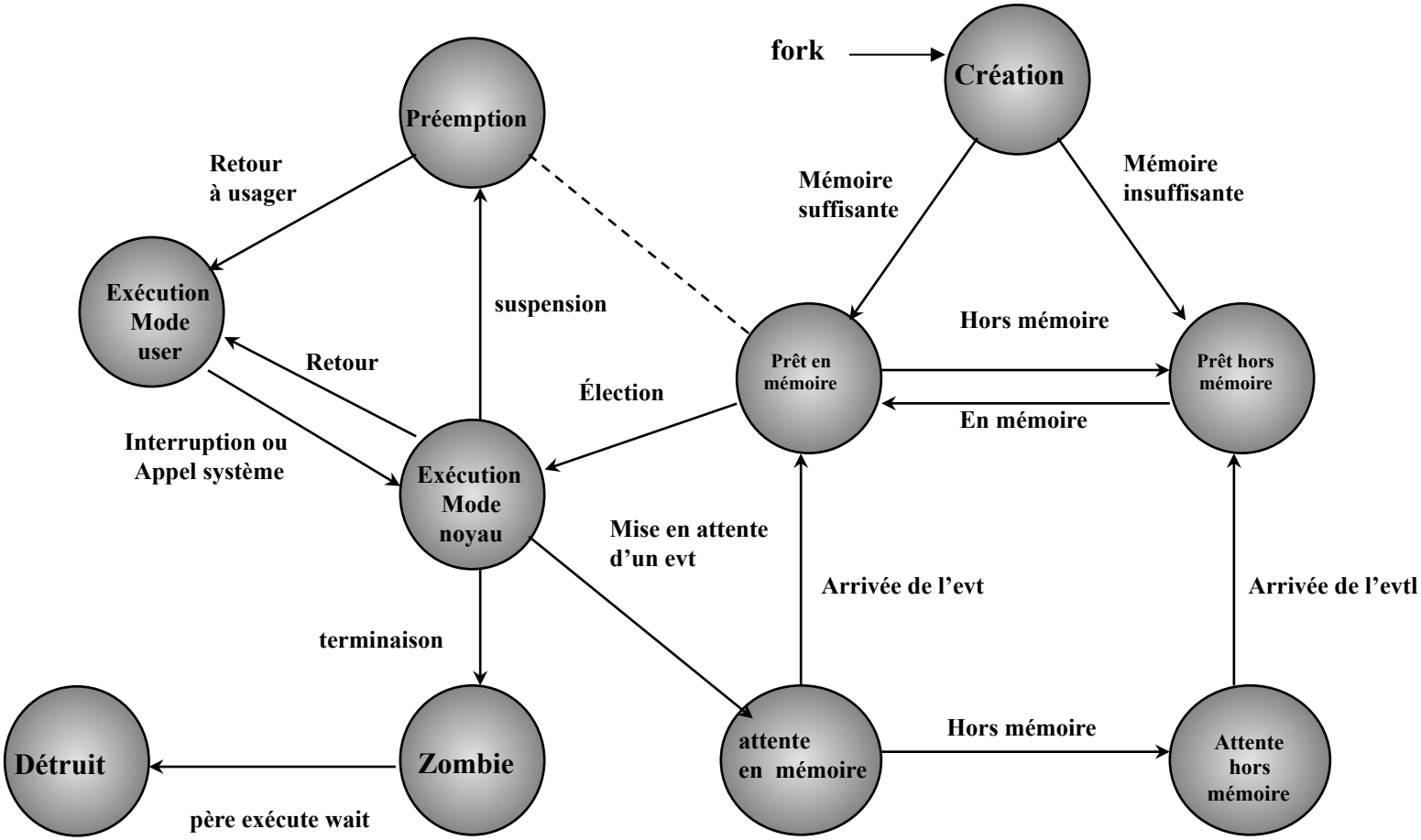
Partage de fichiers entre processus père et fils

- Le fork duplique la table des descripteurs de fichiers du processus père.

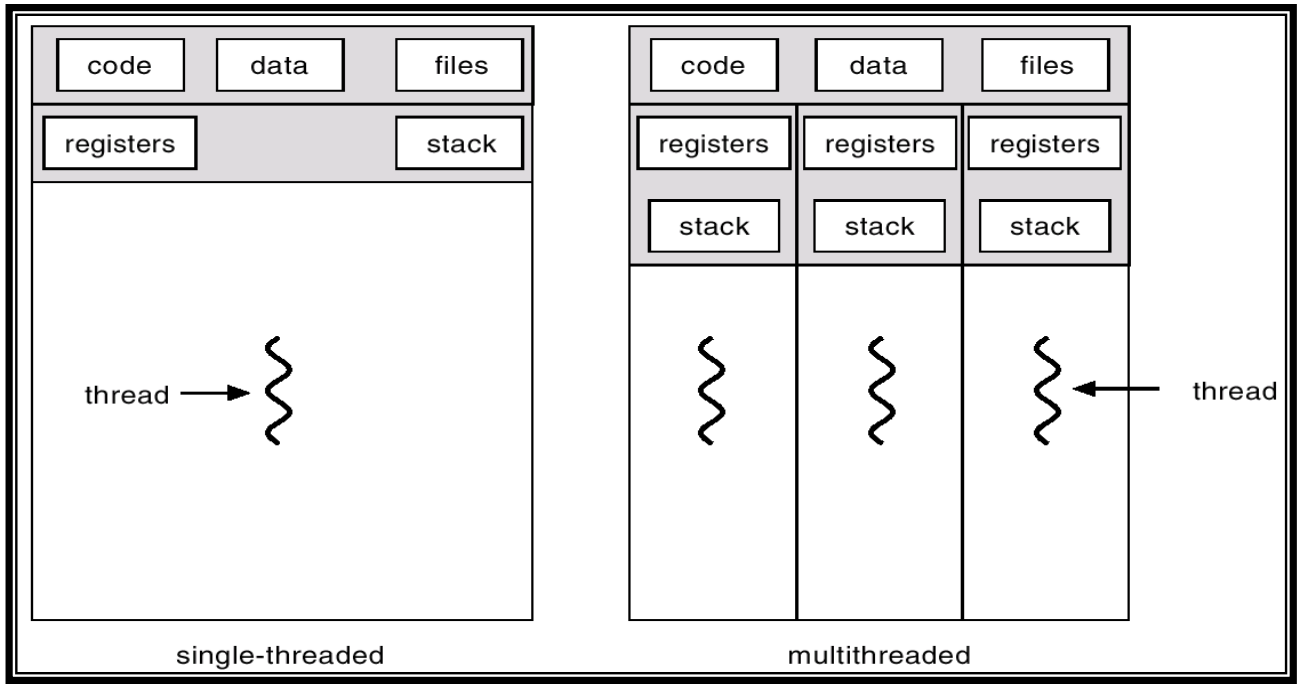


Le fils hérite les descripteurs de fichiers ouverts avec les buffers, il faut donc vider les buffers par fflush avant d'appeler le fork.

Évolution de l'état d'un processus

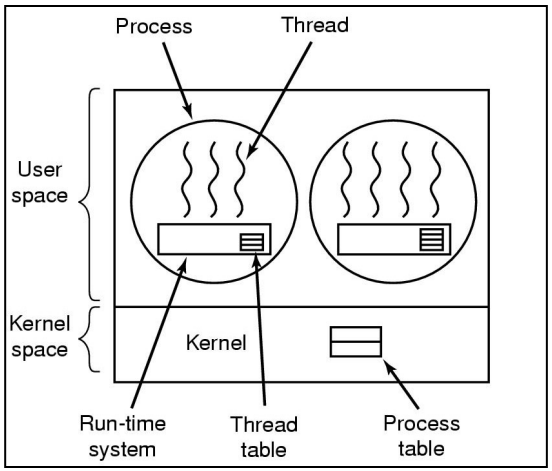


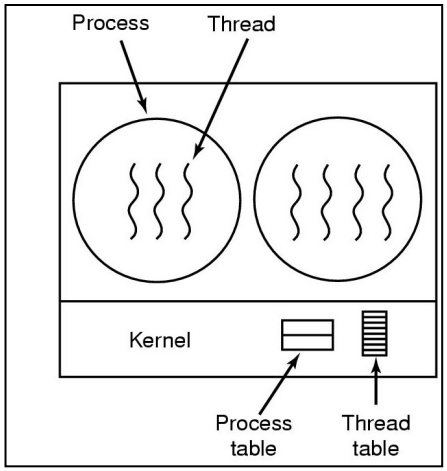
Threads



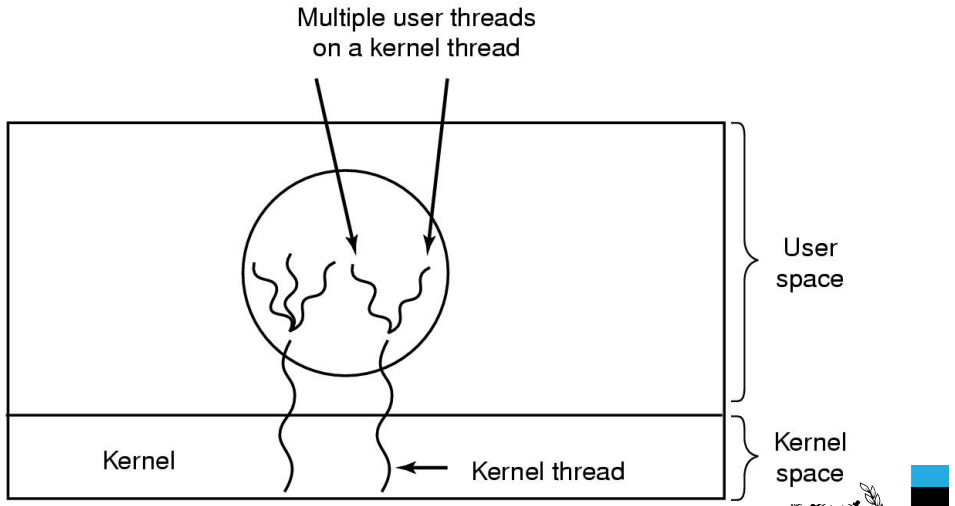
- Le multithreading permet l'exécution simultanée ou en pseudo-parallèle de plusieurs parties d'un même processus.

Implémentation des threads





- Plusieurs-à –un
- Un-à-un
- Plusirurs-à-plusieurs



Communication interprocessus : Tubes anonymes (pipes)

- Un tube de communication anonyme est créé par l'appel système:

  int pipe(int fd[2]).

  Cet appel système retourne, dans fd, deux descripteurs de fichiers :

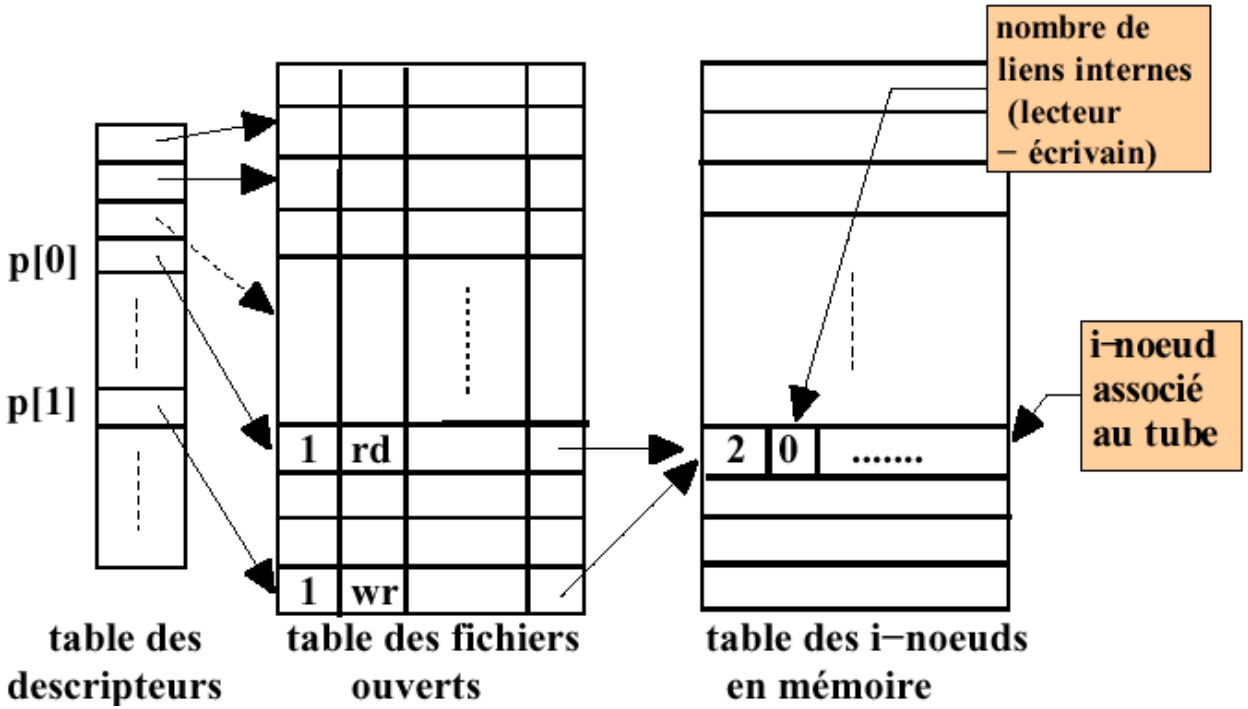
  - fd[0] contient le descripteur réservé aux lectures à partir du tube

  - fd[1] contient le descripteur réservé aux écritures dans le tube.

- L'accès au tube se fait via les descripteurs. Les deux descripteurs sont ajoutés à la table des descripteurs de fichiers du processus appelant.
- Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube (duplication de la table des descripteurs de fichier).
- Lecture et écriture en utilisant read et write (bloquantes par défaut).
- Pour communiquer avec ses processus fils via des tubes de communication anonymes, le processus père doit créer les tubes de communication nécessaires avant de créer ses fils.

Communication interprocessus : Tubes anonymes (pipes)

int p[2] ;
pipe (p) ;



dup2(p[0],0);
Ou
dup2(p[1],1);

Les tubes de communication nommés

- Les tubes de communication nommés fonctionnent aussi comme des files de discipline FIFO (first in first out).

- Ils sont plus intéressants que les tubes anonymes car ils offrent, en plus, les avantages suivants :
  - Ils ont chacun un nom qui existe dans le système de fichiers (table des fichiers); Ils sont considérés comme des fichiers spéciaux ;
  - Ils peuvent être utilisés par des processus indépendants ; à condition qu'ils s'exécutent sur une même machine.
  - Ils existeront jusqu'à ce qu'ils soient supprimés explicitement ;
  - Leur capacité maximale est plus grande (40k).
  - Ils sont créés par la commande « mkfifo» ou « mknod » ou par l'appel système mknod( ) ou mkfifo().

Communication interprocessus : Signaux

- Un signal est une interruption logicielle asynchrone qui a pour but d'informer de l'arrivée d'un événement.
- Ce mécanisme de communication permet à un processus de réagir à un événement sans être obligé de tester en permanence l'arrivée.
- Un processus peut indiquer au système sa réaction à un signal qui lui est destiné :
  - ignorer le signal,
  - le prendre en compte (en exécutant le traitement spécifié par le processus),
  - exécuter le traitement par défaut ou
  - le bloquer (le différer).
- SIGKILL et SIGSTOP ne peuvent être ni ignorés ni captés.

- **Envoi d'un signal :** kill (pid, signum)**; pthread_kill(tid, signum);**
- **Prise en compte d'un signal** : sigaction (signum, action, oldaction) ; et signal(signum,action);
- **Attente d'un signal** : pause ( ); sigsuspend(mask);
- **Blocage/déblocage de signaux : sigprocmask**(how, set, oldset);

Synchronisation de processus

- Le partage d'objets sans précaution particulière peut conduire à des résultats imprévisibles. L'accès à l'objet doit se faire en **exclusion mutuelle**.
- Encadrer chaque section critique par des opérations spéciales qui visent à assurer l'utilisation exclusive des objets partagés.
- Quatre conditions sont nécessaires pour réaliser correctement une exclusion mutuelle :
  - Deux processus ne peuvent être en même temps dans leurs sections critiques.
  - Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.
  - Aucun processus suspendu en dehors de sa section critique ne doit bloquer les autres processus.
  - Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique (attente bornée).

Comment assurer l'exclusion mutuelle?

- Solution de Peterson → attente active (consommation du temps CPU) + extension complexe à plusieurs processus.
- Masquage des interruptions → dangereuse pour le système
- Instructions atomiques (TSL) → attente active + inversion des priorités => boucle infinie.
- SLEEP et WAKEUP → synchronisation des signaux
- Sémaphores (blocage / déblocage de processus, appels système)

Instructions atomiques – verrous actifs (spinlocks)

Implémentation d'un verrou actif (spin-lock) en utilisant TSL :

La boucle active se répète tant que le verrou n'est pas acquis.

```
int lock = 0;
```

```
Processus P1
while (1)
{
   while(TSL(lock)!=0);
   section_critique_P1() ;
    lock=0;
}
```

```
Processus P2
while (1)
{
   while(TSL(lock)!=0);
   section_critique_P2() ;
    lock=0;
}
```

```
int TSL(int &x)
 {
    int tmp = x;
    x =  1;
    return tmp;
}
```

Sémaphores

- Pour contrôler les accès à un objet partagé, E. W. Dijkstra (1965) avait proposé l'emploi d'un nouveau type de variables appelées sémaphores.

- Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès disponibles (jetons d'accès).

- Chaque sémaphore a au moins un nom, une valeur initiale et une file d'attente.

- Les sémaphores sont manipulés au moyen des opérations :

  - P (désigné aussi par down ou wait) et

  - V (désigné aussi par up ou signal).

Sémaphores (2)

- L'opération P(S) décrémente la valeur du sémaphore S si cette dernière est supérieure à 0. Sinon le processus appelant est mis en attente dans la file d'attente du sémaphore.
- L'opération V(S) incrémente la valeur du sémaphore S, si aucun processus n'est bloqué par l'opération P(S). Sinon, l'un d'entre-eux sera choisi et redeviendra prêt (file d'attente gérée FIFO ou parfois LIFO).
- Chacune de ces deux opérations doit être implémentée comme une opération indivisible.

Exclusion mutuelle
→ Sémaphore binaire (mutex)

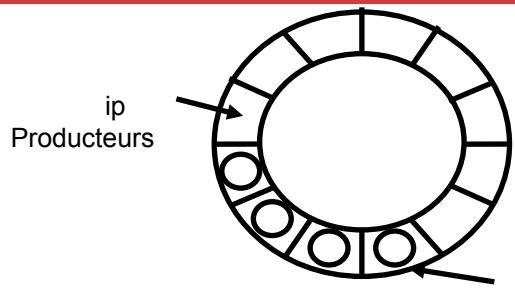
```
Semaphore S = 1 ;
```

```
Processus P1 :
{     P(S)
      Section_critique _de_P1() ;
      V(S) ;
}
```

```
Processus P2 :
{   P(S)
    Section_critique_de_P2();
    V(S) ;
}
```

Problème producteurs/consommateurs

```
int tampon [N];
int ip=0,ic=0;
Semaphore libre=N, occupe=0, mutex=1;
Producteur ()
{
    while(1)
     {    P(libre) ;
          P(mutex);
          produire(tampon, ip);
          ip = Mod(ip +1,N);
          V(mutex);
          V(occupe);
     }
}
```




```
Consommateur ()
 {
          while(1)
          {    P(occupe);
               P(mutex);
               consommer(tampon,ic);
               ic= Mod(ic+1, N);
               V(mutex);
               V(libre);

          }
}
```

Exercices

Processus et Threads : Exercice 1 - chapitre 2

Donnez l'arborescence de processus créés par ce programme ainsi que l'ordre de l'affichage des messages.

```
// chaine_fork_wait.c
int main()
{   int i, n=3;
      pid_t pid_fils;
      for(i=1; i<n;i++)
      {   fils_pid = fork();
          if (fils_pid > 0)
          {  wait(NULL);
               break;
          }
      }
      printf("Processus %d de pere %d\n", getpid(), getppid());
      return 0;
}
```

Processus et Threads : Exercice 2 - chapitre 2

Donnez, sous forme d'un arbre, les différents ordres possibles d'affichage de messages (chaque chemin de l'arbre correspond à un ordre possible).

```
int main()
{   printf("message0\n");
      if (fork())
      {   printf("message1\n");
            if (fork())
                  printf("message2\n");
            else exit(0);
      } else printf("message3\n");
      return 0;
}
```

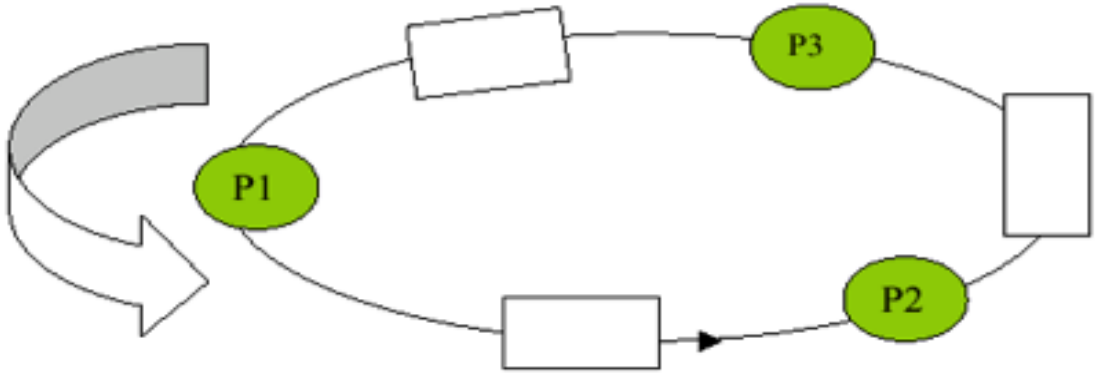
Ajouter une ligne de code pour forcer l'ordre d'affichage suivant :
message0; message3; message1;message2

Tubes de communication : Exercice 1 – chapitre 4

INF3600: Systèmes d'exploitation Contrôle périodique Hiver 2004

Question 4 (4 pts) : Communication interprocessus

On veut établir, en utilisant les tubes anonymes (pipes), une communication de type anneau unidirectionnel entre trois processus fils. Pour ce faire, la sortie standard de l'un doit être redirigée vers l'entrée standard d'un autre, selon le schéma suivant :



Complétez le programme suivant en y ajoutant le code permettant de réaliser les redirections nécessaires à la création d'un tel anneau.

Tubes de communication : Exercice 1– chapitre 4

```
int main ()
{
        /*1*/



    if (fork())  // création du premier processus
    {
        if(fork())
        {
            /*2*/
          if(fork())
            {       /*3*/

                    while (wait(NULL)>0);

                    /*4*/
          } else
          {     // processus P3
                    /*5*/
```

Tubes de communication : Exercice 1– chapitre 4

```
                execlp("program3", "program3",NULL);

                /*6*/
        }

    } else
    {     // processus  P2

          /*7*/




        execlp("program2", "program2",NULL);

          /*8*/
    }
} else
```

Tubes de communication : Exercice 1– chapitre 4

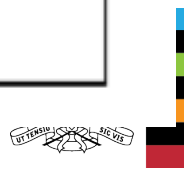
```
{   //processus P1

        /*9*/




        execlp("program1","program1", NULL);

        /*10*/
}
    /*11*/
}
```



Tubes de communication : Exercice 2 – chapitre 4

Considérez le programme suivant qui a en entrée trois paramètres : deux fichiers exécutables et un nom de fichier. Ce programme crée deux processus pour exécuter les deux fichiers exécutables.

Complétez le code de manière à exécuter, l'un après l'autre, les deux fichiers exécutables et à rediriger les sorties standards des deux exécutables vers le fichier spécifié comme troisième paramètre. On récupérera ainsi dans ce fichier les résultats du premier exécutable suivis de ceux du deuxième.

```
int main(int argc, char* argv[])
{      /*0*/
       if (fork()==0)
       {     /*1*/
              execvp(argv[1], &argv[1]);
              /*2*/
       }
       /*3*/
       if (fork()==0)
       {     /*4*/
              execvp(argv[2], &argv[2]);
              /*5*/
       }
       /*6*/
}
```

Signaux :
Exercice 3 – chapitre 4

Le signal SIGCHLD est un signal qui est automatiquement envoyé par le fils à son père lorsque le fils se termine (par un exit, un return, ou autre). **Ajoutez une fonction** et **le code nécessaire** pour que le père n'attende pas son fils de façon bloquante et que le fils ne devienne pas zombie.

```
/*0*/
int main(int argc, char *argv[])
{           /*1*/
            if (!fork())
            {           /*2*/
                        for (int i = 0 ; i <10 ; i++) ; //simule un petit calcul
                        /*3*/
                        exit(1) ;
                        /*4*/
            }
            /*5*/
            while(1) ; //Simule un calcul infini
            /*6*/
}
```

CP- Hiver 2016
Question 1 : (questions à choix multiples)

1) Un processus *zombie* est un processus qui :
a) a perdu son père et n'a plus de père.
b) a terminé son exécution en erreur.
c) a terminé son exécution et attend la prise en compte de cette terminaison par son père.
d) a perdu son père et a été adopté par le processus *init*.
e) aucune de ces réponses.

2) Si chaque processus père attend la fin de tous ses fils avant de se terminer, il n'y aurait aucun processus :
a) zombie dans le système.
b) qui bloque son père.
c) adopté par le processus *init*.
d) bloqué par son père.
e) aucune de ces réponses.

CP- Hiver 2016
Question 1 : (questions à choix multiples)

3) Quels sont les éléments partagés par l'ensemble des threads d'un processus ? Ils partagent :
a) l'espace d'adressage.
b) la table de descripteurs de fichiers.
c) le compteur ordinal.
d) la pile d'exécution.
e) aucun des éléments ci-dessus.

4) On veut faire communiquer deux threads *utilisateur* d'un même processus via un tube anonyme (*pipe*). À quel niveau doit-on créer le tube anonyme ? Le tube anonyme doit être créé :
a) avant la création du premier thread.
b) après la création du premier thread et avant la création du second thread.
c) après la création du second thread.
d) dans chacun des deux threads.
e) aucune de ces réponses car les threads ne pourront pas communiquer en utilisant les fonctions « *read* » et « *write* ».

CP- Hiver 2016
Question 1 : (questions à choix multiples)

5) On veut faire communiquer deux threads utilisateur d'un même processus via un tube nommé. À quel niveau doit-on ouvrir le tube nommé ? Le tube nommé doit être ouvert :
a) avant la création du premier thread.
b) dans le premier thread.
c) dans le second thread.
d) dans chacun des deux threads.
e) aucune de ces réponses car les threads ne pourront pas ouvrir le tube.

6) Si un processus appelle la fonction « *execlp* », quels sont les éléments qu'il va préserver ? Il préserve :
a) l'espace d'adressage.
b) la table des descripteurs de fichiers.
c) les liens père-fils avec les autres processus.
d) la pile d'exécution.
e) le *PID*.
f) aucun des éléments ci-dessus.

CP- Hiver 2016
Question 1 : (questions à choix multiples)

7) Le masque des signaux d'un processus indique quels signaux, à destination du processus, à :
a) ignorer.
b) capter.
c) traiter en priorité en appliquant le traitement par défaut.
d) conserver pour les traiter ultérieurement.
e) aucune des réponses ci-dessus.

8) Le nombre de processus créés par l'instruction « *while (pid=fork()) if (n >= 5) break ; else n=n+1;*», où *n* est un entier initialisé à *0*, est :
a) *4.*
b) *5.*
c) *6*.
d) *>10*.
e) aucune des réponses ci-dessus.

CP- Hiver 2016
Question 1 : (questions à choix multiples)

9) Considérez le programme suivant :

```
bool Continuer = true;
void action (int sig)
{   printf("reception de %d\n", sig);
    if (sig==SIGUSR1) Continuer =false;
}
int main ()
{   signal(SIGUSR1, action);
    signal(SIGINT, action);
    while (Continuer)
   {    kill(getpid(), SIGINT);
        sleep(1);
        kill(getpid(), SIGUSR2);
        sleep(1);
        kill(getpid(), SIGUSR1);
    }
    exit(0);
 }
```

Le processus va être terminé par :
a) le signal *SIGUSR2.*
b) le signal *SIGINT*.
c) le signal *SIGUSR1*.
d) l'instruction « *exit(0)* ».
e) aucune des réponses ci-dessus.

CP- Hiver 2016
Question 1 : (questions à choix multiples)

10) Considérez la commande « *cat* | *sort* > *data* ». La sortie standard du processus exécutant *cat* est :

a) le fichier *data*.
b) le pipe.
c) l'écran.
d) le clavier.
e) aucune des réponses ci-dessus.

CP- Hiver 2016
Question 2 : Processus & redirection des E/S standards

1) Complétez le code ci-dessous pour qu'il réalise le traitement suivant : le processus principal incrémente la variable *v*, crée deux processus fils *F1* et *F2*, puis se met en attente de la fin de ses fils. Chaque fils *Fi*, pour i=*1,2*, crée un fils *Fi1*, incrémente la variable *v*, puis se met en attente de la fin de son fils. Enfin, chaque petit fils *Fi1*, pour *i=1,2*, affiche à l'écran son numéro, celui de son père et la valeur de *v*, puis se termine. Indiquez la valeur de *v* affichée à l'écran par chaque petit fils.

```
…..
int v=10 ;
int main()
{      int i ;
       v=v+1 ;
    …..
    for(i=0 ; i<2 ; i++)
   {  ….  }
…..
}
```

2) Complétez le code en 1) afin que les affichages à l'écran (réalisées par les petits fils) soient redirigées vers un même fichier nommé data, créé par le programme.

CP- Hiver 2016
Question 3 : (Synchronisation des processus)

Considérez un système de réservation de billets d'un spectacle. Ce système de réservation est composé de trois fonctions : *int get_free()*, *bool book()* et *bool cancel()*. La fonction *get_free* permet de récupérer le nombre de places encore disponibles pour le spectacle. La fonction *book* permet de réserver une place pour le spectacle. La fonction *cancel* permet d'annuler une réservation pour le spectacle. On vous fournit le code suivant de ces trois fonctions :

```
// variables globales
int free=N ;
int  get_free () { return free; }

bool book ( )
{   if (free ==0) return false;
    free = free – 1; return true;
 }

bool cancel ( )
{     if (free<N)
      {  free = free + 1; return true;}
     return false;
 }
```

La gestion de la réservation de billets pour le spectacle est prise en charge par un processus. Toutes les demandes des utilisateurs concernant ce spectacle sont dirigées vers ce processus. Ce processus crée un thread pour chaque demande reçue. Ce thread se charge de traiter la demande en faisant appel aux fonctions *get_free, book* ou/et *cancel*.

Un code (ou une fonction) est dit « *safe-thread* » s'il est capable de fonctionner correctement lorsqu'il est exécuté simultanément par plusieurs threads d'un même processus.

CP- Hiver 2016
Question 3 : (Synchronisation des processus)

1) Indiquez pour chacune des fonctions *get_free, book* et *cancel* si elle est « safe-thread » ? Si vous répondez non, expliquez pourquoi par un exemple puis corrigez son code, en utilisant les sémaphores (utilisez le type *Semaphore* et les primitives *P* et *V)*. Si vous répondez oui, expliquez pourquoi en montrant que l'exécution concurrente de la fonction préserve les variables partagées dans un état cohérent.

2) On veut maintenant mettre en attente, en utilisant des sémaphores, toute demande de réservation qui ne peut être satisfaite, jusqu'à ce qu'une place se libère. Donnez le code des trois fonctions qui tient compte de cette directive et qui est aussi « safe-thread » (utilisez le type *Semaphore* et les primitives *P* et *V)*.

Processus et threads (CP-Automne 2013)

1) Un processus PP crée trois processus fils F1, F2 et F3, attend la fin de ses fils, affiche la valeur d'un compteur défini dans sa zone de données globales, puis se termine. Chaque processus fils incrémente le compteur (compteur=compteur+10), affiche sa valeur puis se termine. Le compteur est initialisé lors de sa définition à 100.

a) Donnez la valeur (ou les valeurs possibles) affichée(s) à l'écran par les processus PP, F1, F2 et F3.

b) Supposez que le compteur est déclaré et initialisé à l'intérieur de la fonction « main », avant la création du premier fils. Donnez la valeur (ou les valeurs possibles) affichée(s) à l'écran par les processus PP, F1, F2 et F3.

c) Supposez que PP crée deux threads Th1 et Th2 (au lieu de 3 processus), attend la fin des threads puis affiche la valeur du compteur. Donnez la valeur (ou les valeurs possibles) affichée(s) à l'écran par le processus PP.

Processus et threads

2) Quel est le nombre de processus créés par la séquence d'instructions suivante (supposez que les appels système ne retournent pas d'erreur) :

fork(); fork(); fork(); execvp(″/bin/ls.exe", "ls.exe") ; fork(); fork(); fork();

Donnez l'arborescence des processus créés.

Processus et threads (CP – Automne 2013)

3) Considérez le code suivant :

```
int gen_alea( int LOW, int HIGH ) //génère aléatoirement un nombre entre LOW
      et HIGH
{       srand((unsigned int) clock());
        return rand() % (HIGH - LOW + 1) + LOW;
}
int main ( )
{    /*0*/

}
```

a) Complétez le code de la fonction « main » pour créer 5 processus fils. Chaque processus fils exécute la fonction « gen_alea(1,20) » puis se termine. Le processus père se termine après création de tous ses fils. **Vous ne devez pas traiter les cas d'erreur.**

b) Est-il possible au processus principal de récupérer le nombre aléatoire généré par chacun de ses fils ? Si oui, complétez/modifiez le code de la fonction « main » afin de récupérer et d'afficher ces nombres à l'écran.

Tubes de communication

1) Considérez le code suivant :

```
void Lire ()
{  char* c;
    while (read(0,c,1) >0)
        write(1,c,1);
}
int main()
{ char message [512];
    int fd[2];
    pipe (fd);
    dup2 (fd[1], 1);
    close (fd[1]);
     Lire ();
     read (fd[0], message, 512);
    close (fd[0]);
     return 0;
}
```

1. Le processus pourra-t-il récupérer dans « message » tout le message lu par la fonction Lire ?

2. Est-ce que le processus peut bloquer indéfiniment sur la lecture ou l'écriture du tube ?

3. On veut afficher à l'écran, après fermeture du tube (c-à-d après l'instruction close(fd[0])), le message lu du tube. Complétez le code pour satisfaire cette directive.

Synchronisation (CP – Automne 2013)

1) Considérez les deux processus A et B suivants :

Semaphore S=1;

Process A ()	Process B ()
{	{
a;	P(S);
P(S);	d;
b;	V(S);
c;	
V(S)	
}	}

a) Donnez les différents ordres d'exécution des instructions atomiques a, b, c et d des processus A et B.

b) Supposez que le sémaphore S est initialisé à 0. Donnez les différents ordres d'exécution des instructions atomiques des processus A et B.

Synchronisation (CP – Automne 2013)

2) Plusieurs threads d'un même processus partagent une fonction de lecture *get_data()* qui peut s'exécuter en concurrence. Expliquez comment utiliser les sémaphores pour limiter à N le nombre maximal de threads exécutant en concurrence cette fonction de lecture.

```
void get_data( )  {   // corps de get_data }
```