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 2017

Séance de révision

Généralités

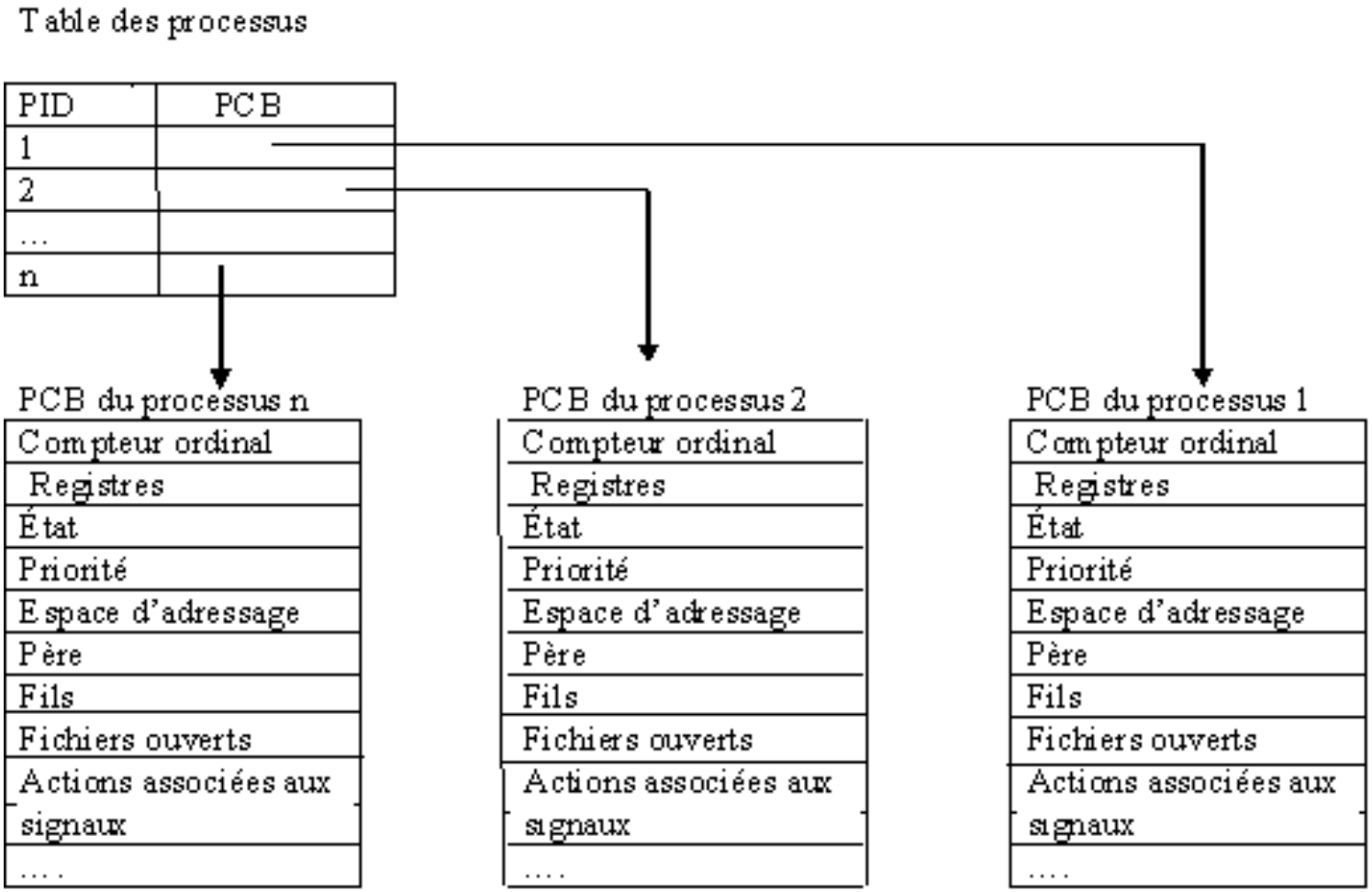
Processus & Threads

Tubes de communication & signaux

Synchronisation (sans les moniteurs)

Exercices

Processus = Programme en cours d'exécution



Processus (Linux – Unix)

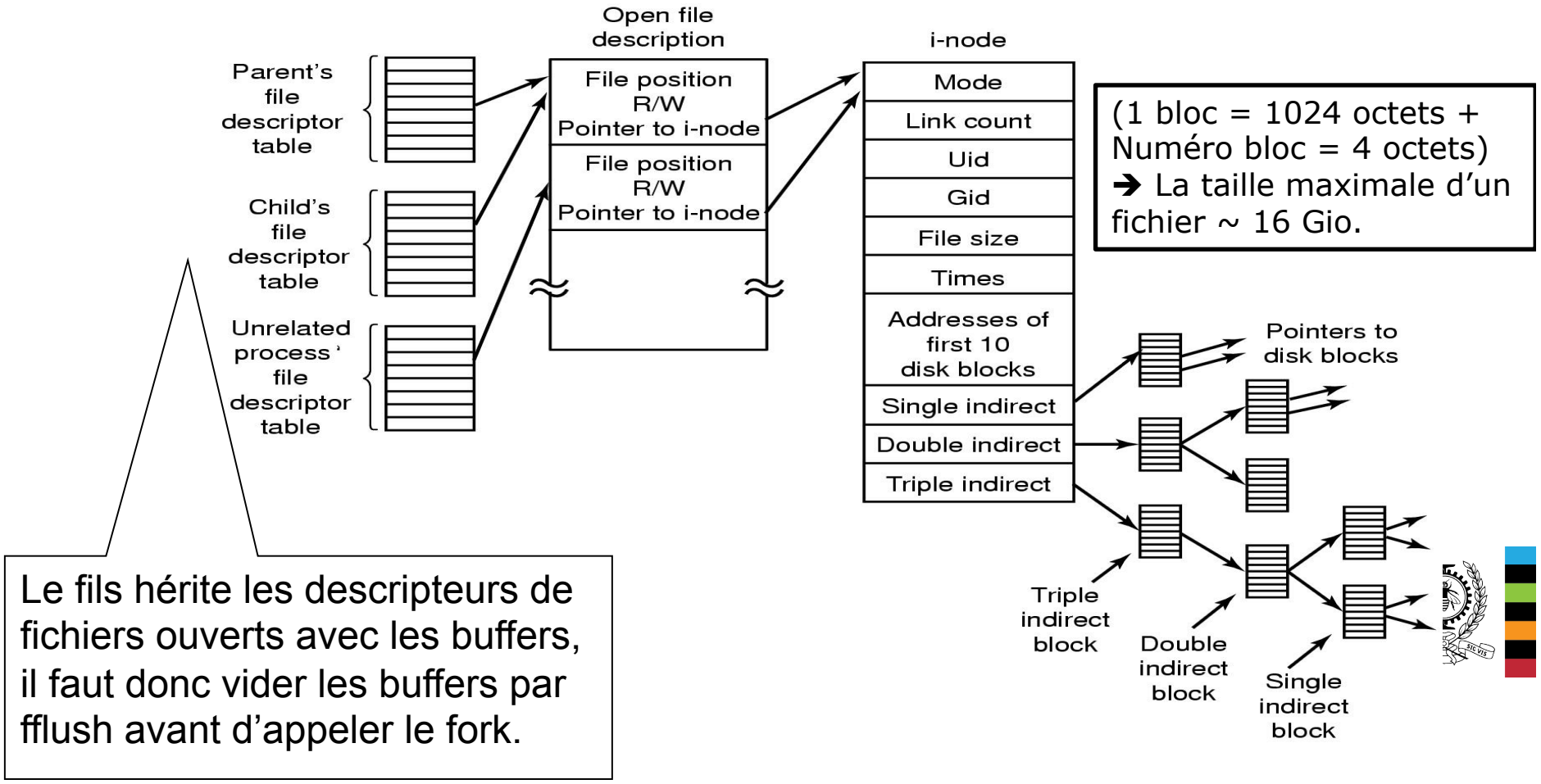
- Création par duplication (Copy-On-Write) : pid_t fork();
- Changement de code :

  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

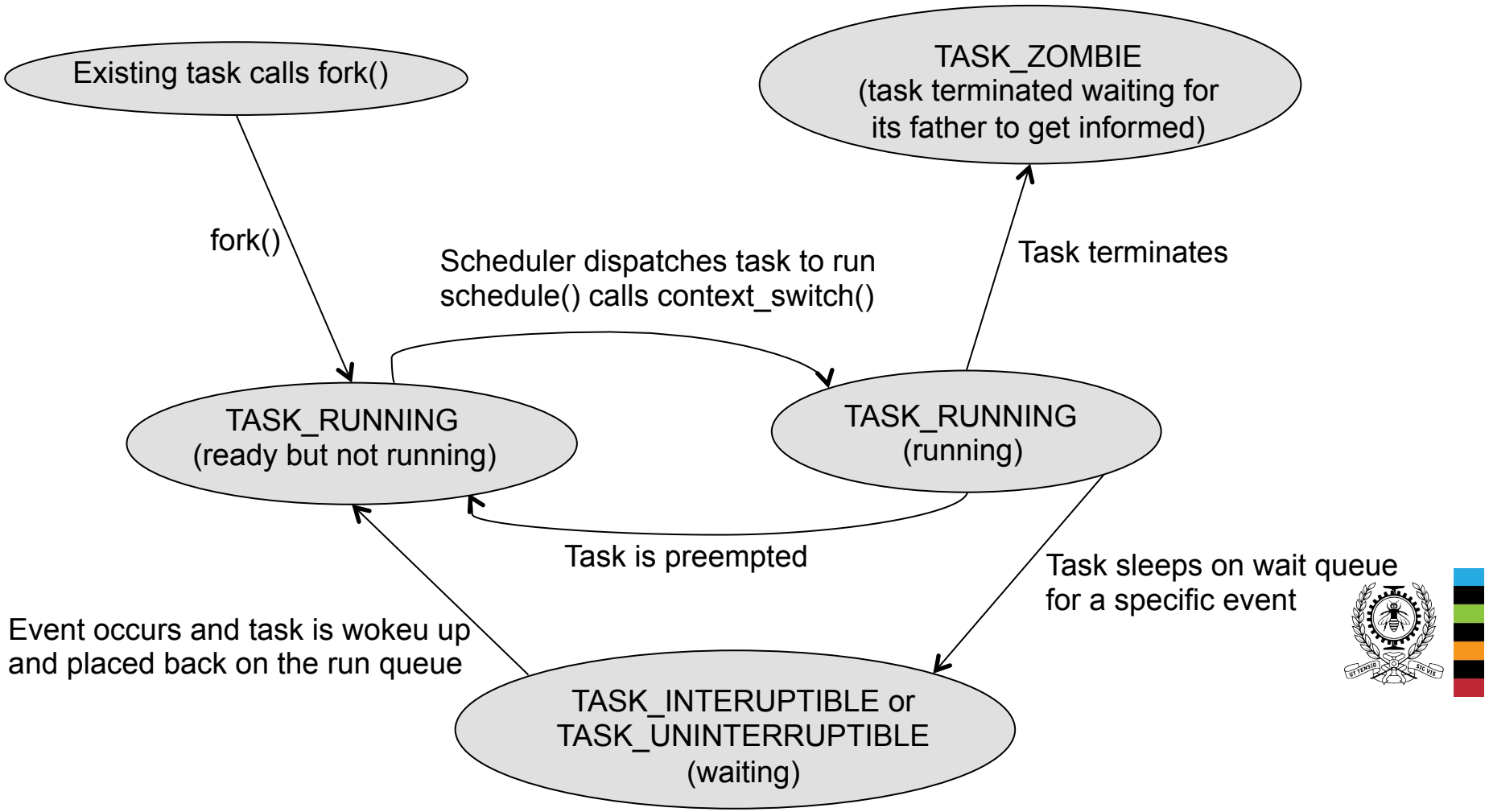
- Le fork duplique la table des descripteurs de fichiers du processus père.



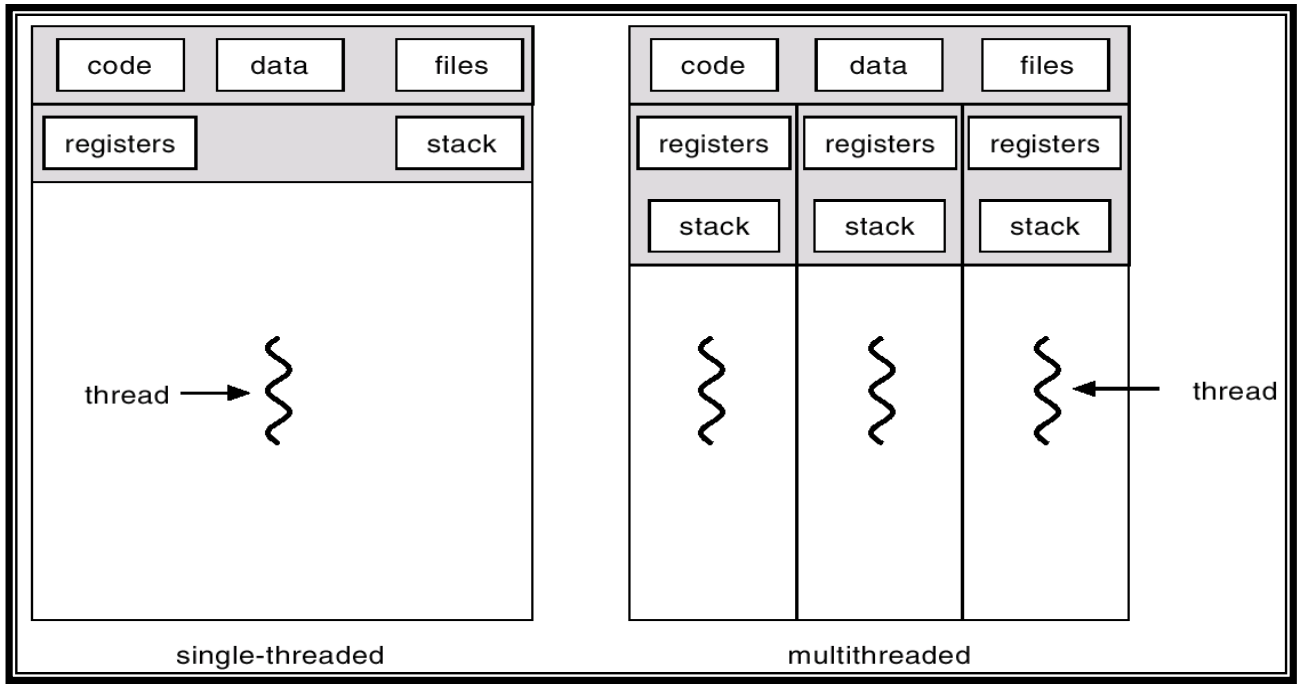
(1 bloc = 1024 octets + Numéro bloc = 4 octets)
➔ La taille maximale d'un fichier ~ 16 Gio.

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

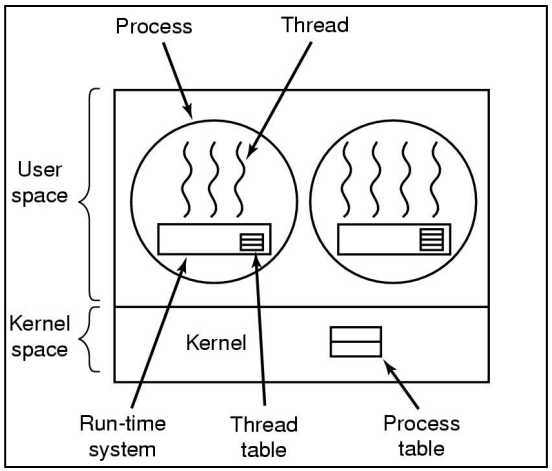


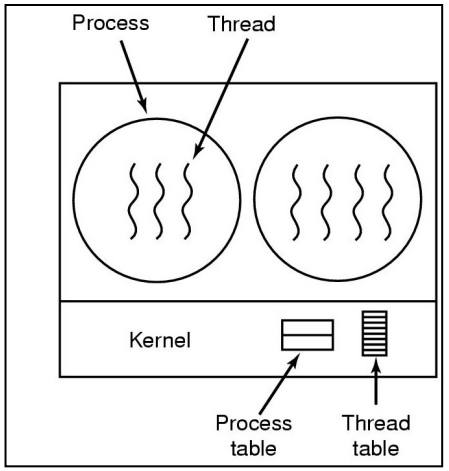
Threads



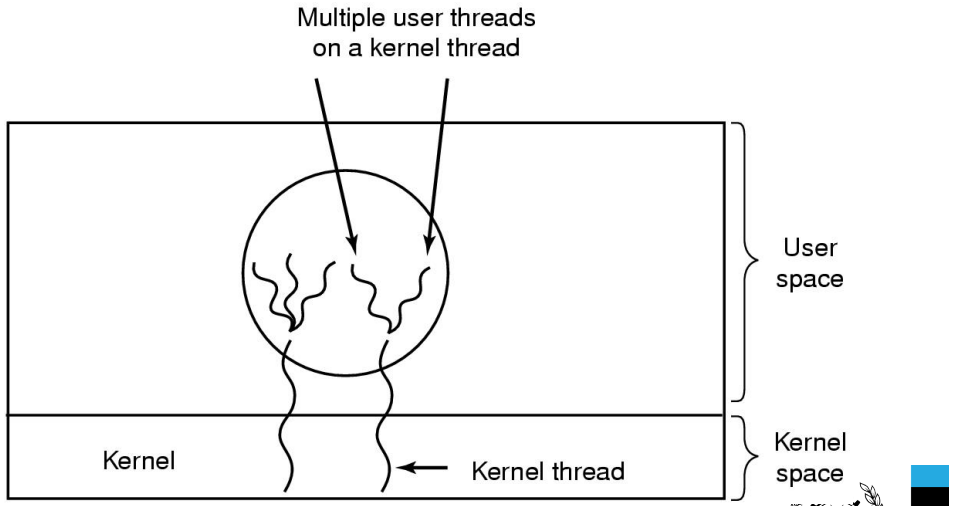
- Le multithreading permet l'exécution simultanée ou en pseudo-parallèle de plusieurs parties d'un même processus.

Implémentation des threads





- Plusieurs-à –un
- Un-à-un
- Plusieurs-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 :

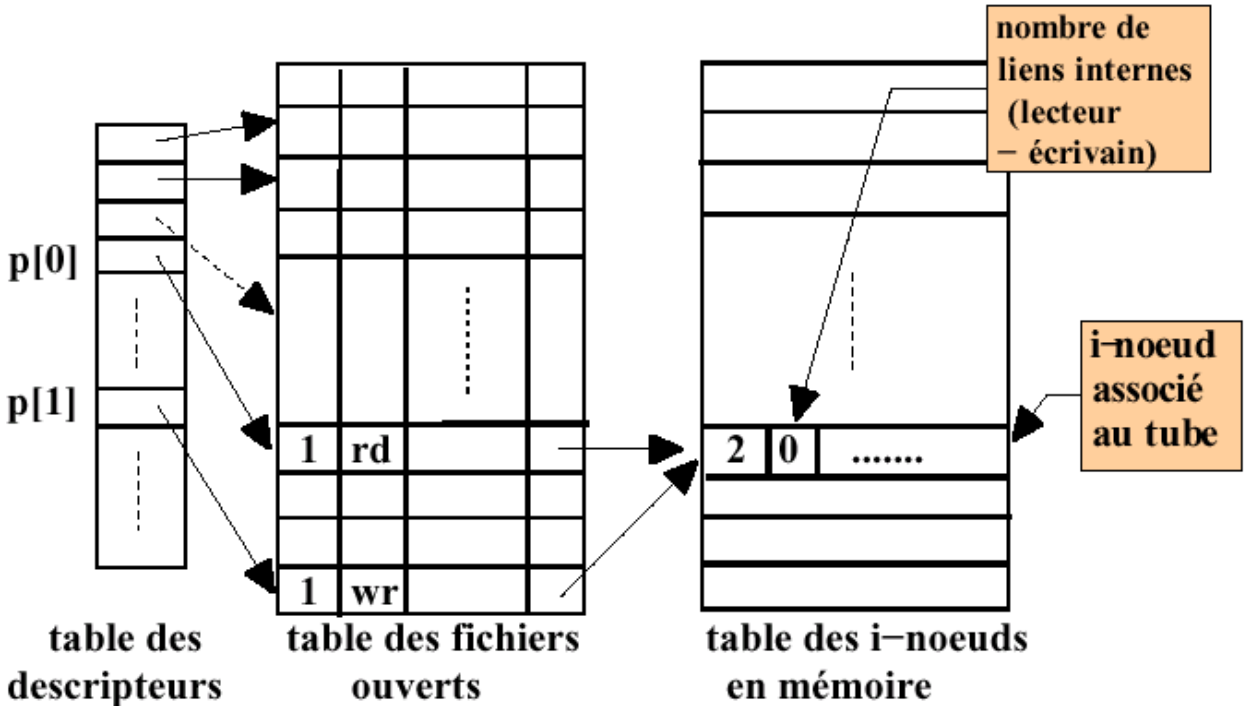
  - fd[0] contient le descripteur réservé aux lectures à partir du tube

  - 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().

- Par défaut, il y a une synchronisation sur l'ouverture d'un tube nommé. Cette synchronisation garantit juste après l'ouverture, il y a au moins un lecteur et un écrivain.

Les tubes de communication

- Chaque tube a un nombre de lecteurs (nombre de descripteurs en lecture) et un nombre d'écrivains (nombre de descripteurs en écriture).

- La fin de fichier d'un pipe (eof) est atteinte lorsque le pipe est vide et son nombre d'écrivains est 0.

- L'écriture dans un pipe rompu (0 lecteurs) par un processus génère un signal SIGPIPE qui est envoyé au processus. Le traitement par défaut de ce signal est la terminaison du processus.

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. Il sera informé par un signal envoyé par le système ou un autre processus.
- 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

- Les accès simultanés à un objet partagé peut conduire à des résultats incohérents. 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 libre.

```
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 est choisi et redevient 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)

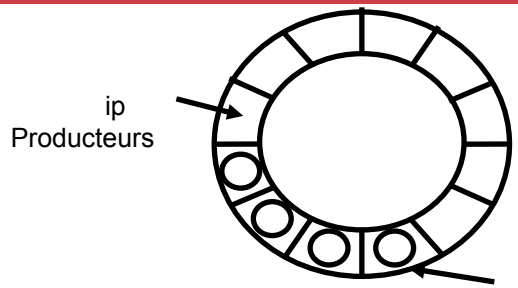
```
Semaphore S = 1 ;
```

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

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

Problème producteurs/consommateurs

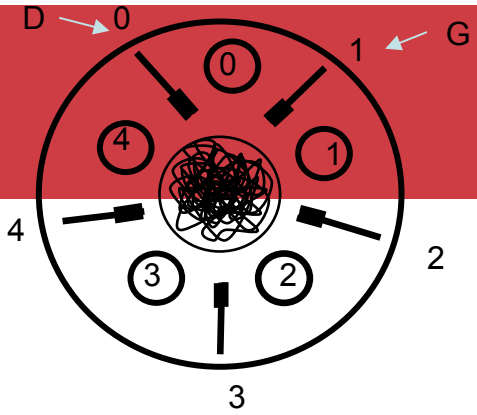
```
int tampon [N];
int ip=0,ic=0;
Semaphore libre=N, occupe=0, mutex=1;
Producteur ()
{
    while(1)
     {    P(libre) ;
          P(mutex);
          produire(tampon, ip);
          ip = Modulo (ip +1,N);
          V(mutex);
          V(occupe);
     }
}
```




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

        }
}
```

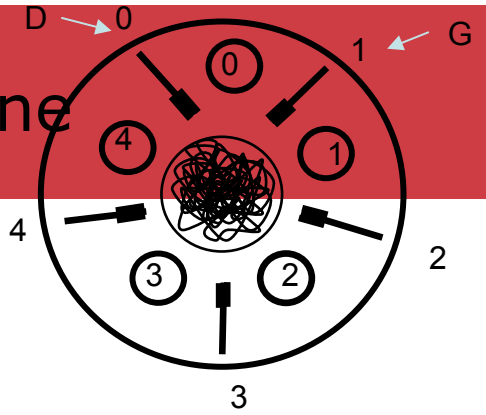
Problème des philosophes




```
void * philosophe(void * num) {
    int i = *(int *) num, nb = 2;
    while (nb) {
        sleep(1);           // penser
        sem_wait(&mutex);   // essayer de prendre les fourchettes pour manger
        if (fourch[G] == libre && fourch[D] == libre) {
            fourch[G] = occupe;
            fourch[D] = occupe;
            sem_post(&mutex);
            nb--;
            printf("philosophe[%d] mange\n", i);
            sleep(1);           // manger
            printf("philosophe[%d] a fini de manger\n", i);
            sem_wait(&mutex);   // libérer les fourchettes
            fourch[G] = libre;
            fourch[D] = libre;
            sem_post(&mutex);
        } else  sem_post(&mutex);
    }
}
```

Problème de famine

Problème des philosophes (7) : Famine




```
#define N 5         // nombre de philosophes
#define G (i+1)%N   // fourchette gauche du philosophe i
#define D i         // fourchette droite du philosophe i

sem_t f[N]; // à initialiser à 1 dans la fonction main

void * philosophe(void * num) {
    int i = *(int *) num, nb = 2;
    while (nb) {
        sleep(1);           // penser
        if (G < D) {  sem_wait(&f[G]); sem_wait(&f[D]);}
        else {  sem_wait(&f[D]); sem_wait(&f[G]); }
            nb--;
            printf("philosophe[%d] mange\n", i);
            sleep(1);           // manger
            printf("philosophe[%d] a fini de manger\n", i);
            sem_post(&f[G]); sem_post(&f[D])
    }
}
```

Demander une à une les deux fourchettes (en commençant par la plus petite) avec attente qu'elles se libèrent.

Problème des Lecteurs / Rédacteurs

Solution :

```
Semaphore tour=1, mutex=1, redact =1;
int Nbl=0;
Redacteur () {
  while(1) {
    P(tour);   // attendre son tour
    P(redact); // assurer l'accès
               // exclusif à la base
    V(tour);
    Ecriture();
    V(redact);
  }
}
```

```
Lecteur () {
  while(1) {
    P(tour);  // attendre son tour
    P(mutex) ; // assurer l'accès
               // exclusif à Nbl
    Nbl++ ;
    if (Nbl==1) P(redact);
    V(mutex);
    V(tour);
    Lecture();
    P(mutex);
    Nbl--;
    if(Nbl==0) V(redact);
    V(mutex);
  }
}
```

Exercices

Répondez aux questions à choix multiples en sélectionnant une ou plusieurs réponses. Les questions font référence aux systèmes d'exploitation de la famille UNIX et considèrent les options par défaut des appels système.

QCM – Communication interprocessus

1) On veut faire communiquer deux threads Posix d'un même processus via un tube anonyme(pipe). Le thread lecteur lit, caractère par caractère, du tube jusqu'à ce qu'il rencontre une fin de fichier. Le thread écrivain dépose dans le tube, caractère par caractère, le contenu d'un fichier.

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.

Le thread lecteur du pipe doit fermer le descripteur d'écriture du pipe, :
a) avant la première lecture du pipe.
b) entre le début et la fin du thread (peu importe l'endroit).
c) nulle part.
d) aucune de ces réponses.

QCM – Communication interprocessus

2) Laquelle des confirmations suivantes est valide pour un tube nommé créé ?
a) Le tube apparait dans le système de fichier.
b) Le tube permet exclusivement la communication entre un père et son fils
c) Le tube doit être créé dans le père avant la création du fils.
d) Le tube doit être ouvert avant d'y accéder.
e) aucune des réponses ci-dessus.

3) 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.

QCM - Synchronisation

4) La fonction kill permet uniquement:
a) d'envoyer un signal d'arrêt à un processus.
b) d'envoyer un signal à un ou plusieurs processus.
c) de tuer un processus fils.
d) de débloquer un processus fils.
e) aucune des réponses ci-dessus.

5) Considérez la solution au problème d'exclusion mutuelle qui se base sur l'instruction « TSL ».
Cette solution pose un problème de boucle infinie. Peut-on avoir le même problème dans le cas d'un ordonnancement circulaire sans priorité ?
a) Oui.
b) Non.

QCM - Synchronisation

```
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 = Modulo (ip +1,N);
             V(mutex);
              V(occupe);
      }
}
```

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

          }
}
```

6) La permutation (en rouge) :

a) n'affecte pas le comportement des producteurs et des consommateurs .
b) peut mener vers un interblocage.
c) va imposer une alternance entre les productions et les consommations.
d) va forcément mener vers un interblocage.
e) aucune des réponses ci-dessus.

QCM - Synchronisation

7) Considérez les processus P1, P2, P3 et P4, et les sémaphores S1 et S2. Les processus sont lancés en concurrence. Indiquez les ordres d'exécution des opérations atomiques a, b, c, d, e et f qui ne sont pas réalisables.

Semaphore S1=2, S2=0 ;			
P1	P2	P3	P4
P(S1) ; a; b; V(S2) ;	P(S1) ; c; V(S2) ;	P(S2) ; P(S2) ; d; e; V(S2) ;	P(S2) ; f ; V(S1) ; V(S1) ;

a) a; b; c; f; d; e;

b) a; c; b; d; e; f;

c) a; b; c; d; e; f;

d) a; b; f; c; d; e;

Exercice 1 : Processus (CP – Aut 2013)

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.

Exercice 1 : Processus (CP – Aut 2013)

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*/  int i;
             for(i=0; i<5; i++)
                 if(fork()==0)  {gen_alea(1,20); exit(0);}
             exit(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.**

Exercice 1 : Processus (CP – Aut 2013)

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*/
    int i, v;
    for(i=0; i<5; i++)
      if(fork()==0)  {v=gen_alea(1,20); exit(v);}
    while((i=wait(&v))>0)
      printf( ”valeur %d de %d \n”, WEXITSTATUS(v),i);
     exit(0);
}
```

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.

Oui.

Exercice 2 : Processus & redirection des E/S standards (CP- Hiv 2016)

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.

Exercice 2 : Processus & redirection des E/S standards (CP- Hiv 2016)

```
int v=10;
int main()
{ int i;
  v=v+1;
  int fd = open("data",  O_CREAT|O_WRONLY); // pour 2)
  for(i=0; i<2; i++)
  {   if (fork() == 0)   // F1 ou F2
      { if (fork()==0)   // F11 ou F21
          {    dup2(fd,1); close(fd); // pour 2)
               printf("proc. %d, de père %d, v=%d \n",getpid(), getppid(), v);
               exit(0);
          }
          v=v+1;
       close(fd); // pour 2)
       wait(NULL) ; exit(0);
      }
  }
  close(fd); // pour 2)
  wait(NULL); wait(NULL);exit(0);
}
```

Remarque : Si l'ouverture du fichier data est réalisée dans F11 et F21, il y aurait deux pointeurs de fichiers pour data au lieu d'un seul pointeur. Dans ce cas, pour récupérer les deux affichages, il suffit d'ajouter l'option O_APPEND à open. Ainsi, avant chaque « write », le pointeur de fichier sera positionné à la fin du fichier.

Exercice 3 : Synchronisation (CP – Aut 2013)

Considérez les deux processus A et B suivants :

Semaphore S=1;	

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

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

a; b;c;d; a;d;b;c; d;a;b;c;

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

a; car les deux processus vont être bloqués par P(S);

Exercice 3 : Synchronisation (CP – Aut 2013)

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 }
```

Semaphore acces=N;

```
void get_data( )  {      P(acces);
                         // corps de get_data
                         V(acces);
                   }
```

Exercice 4 : Tubes de communication

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.

Exercice 4 : Tubes de communication

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 ?

Non, si la fonction Lire a inséré dans le pipe plus de 512 caractères, seuls les 512 premiers seront récupérés dans la variable message.

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

Oui, si la fonction Lire remplit le pipe mais n'a pas fini d'écrire dans le pipe. L'appel "write(1,c,1)" va bloquer le processus pour toujours.
Oui, , si la fonction Lire a inséré moins de 512 caractères dans le pipe puis se termine. L'appel "read(fd[0],message,512)" va bloquer le processus pour toujours. Pour éviter ce problème, il suffit de d'ajouter "close(1);" juste avant "read(fd[0],message, 512)".

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.

Exercice 4 : Tubes de communication

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];
    int sauv_1= dup(1);
    pipe (fd);
    dup2 (fd[1], 1);
    close (fd[1]);
     Lire ();
     read (fd[0], message, 512);
    close (fd[0]);
    write(sauv_1,message, 512);
     return 0;
}
```

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.

Exercice 5 : Synchronisation (hiv 2017)

- Considérez le pseudo-code de la solution au problème des producteurs / consommateurs :

```
const int N = 10;
int tampon [N];
int ip=0,ic=0;
Semaphore libre=N, occupe=0, mutex=1;

Producteur (int nump)
{ while(1)
  { P(libre);
    P(mutex);
    tampon[ip] = nump;
    ip = Modulo (ip +1,N);
    V(mutex);
    V(occupe);
  }
}
```

```
Consommateur (int numc)
 { int item;
   while(1)
   {   P(occupe) ;
       P(mutex);
       item = tampon[ic] ;
       ic= Modulo (ic+1, N);
       V(mutex);
       V(libre);
   }
 }
```

Exercice 5 : Synchronisation (hiv 2017)

Supposez qu'il y a exactement 3 producteurs (numérotés de 0 à 2) et 3 consommateurs (numérotés 0 à 2). On veut que les producteurs produisent tour à tour (producteur 0, producteur 1, producteur 2, producteur 0, etc.) et que les consommateurs consomment aussi tour à tour (consommateur 0, consommateur 1, consommateur 2, consommateur 0, etc.).

1) Modifiez le pseudo-code précédent de manière à satisfaire les requis précédents (productions tour à tour et consommations tour à tour). Pour la synchronisation, vous devez vous limiter à l'utilisation de sémaphores.
Peut-on éliminer le sémaphore mutex ? Justifiez votre réponse.

2) On veut maintenant que chaque producteur produise deux items à chaque tour. Modifiez le pseudo-code obtenu en 1 de manière à satisfaire ce requis supplémentaire. Pour la synchronisation, vous devez vous limiter à l'utilisation de sémaphores.

Exercice 5 : Synchronisation (hiv 2017)

```
const int N = 10;
int tampon [N];
int ip=0,ic=0;
Semaphore libre=N, occupe=0, mutex=1;
Semaphore prod[3] ={1,0,0}, cons[3]={1,0,0}

Producteur (int pid)
{ while(1)
   {  P(prod[pid]) ;
      P(libre);
      P(libre); // pour 2
      P(mutex);
      tampon[ip] = pid;
       ip = Modulo (ip +1,N);
       tampon[ip] = pid; // pour 2
       ip = Modulo (ip +1,N); // pour 2
       V(mutex);
       V(occupe);
       V(occupe); // pour 2
       V(prod[Modulo (pid+1,3)]) ;
    }
}
```

```
Consommateur (int cid)
  {  int item;
     while(1)
      {  P(cons[cid]) ;
         P(occupe) ;
         P(mutex);
          item = tampon[ic] ;
          ic= Modulo (ic+1, N);
          V(mutex);
          V(libre);
          V(cons[Modulo (cid+1,3)]) ;
      }
  }
```

Le sémaphore mutex n'est plus nécessaire car il n'y a ni d'accès concurrents à ip ni d'accès concurrents à ic.

Exercice 6 : Synchronisation des processus (CP- Hiv 2016)

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 :

```
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.

Exercice 6 : Synchronisation des processus (CP- Hiv 2016)

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.

La fonction get_free est safe-thread. Les deux autres ne le sont pas car ils accèdent en concurrence en lecture et écriture à la variable partagée free.

```
Semaphore mutex=1;
bool book ( )
{  P(mutex); if (free ==0)  { V(mutex); return false;   }
    free = free – 1; V(mutex); return true;
 }
bool cancel ( )
{P(mutex);  if (free<N)
      {  free = free + 1; V(mutex);  return true;}
     V(mutex); return false;
 }
```

Exercice 6 : Synchronisation des processus (CP- Hiv 2016)

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)*.

Semaphore libre= N, mutex=1;

```
bool book ( )
{    P(libre); P(mutex);
     free = free – 1; V(mutex);  return true;
 }

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