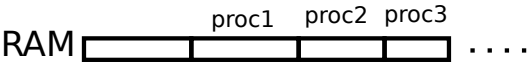
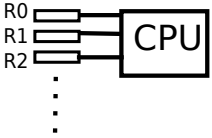
Ressource :

- wiki-du-lama
- Systèmes d'exploitation (A. Tanenbaum) chap 2 et 6

Définition : La multiprogrammation, c'est exécuter simultanément plusieurs programmes sur une même machine, un même CPU, en basculant rapidement de l'un à l'autre, créant une impression de simultanéité.





	Proc1	Proc2	Proc3
R0	...	...	...
R1	...	...	...
...	...	...	...

Processus : Un programme en cours d'exécution

2 composantes principales :

- Espace d'adressage (mémoire) : code, données, pile
- Registres : valeurs

Préemption : Action d'interrompre l'exécution d'un processus avec l'intention de la reprendre plus tard.

Rq : La coopération du processus n'est pas requise, et le processus ne "sait" pas qu'il est préempté.

Rq : On ne connait pas l'ordonnanceur. On ne peut faire aucune supposition sur ce qu'il fera.

Causes de la préemption :

1. A la demande du processus (sleep, ...)
2. Appel système (ex : utilisation d'un périphérique)
3. Déclenchement d'une interruption (par l'horloge, par un périphérique etc.)

Exemple wiki

Exemple : x = x + 1 voir figure 2 incomplète

Conclusion :

1. Prudence lorsque deux processus simultanés manipulent les même données
2. Difficulté de debuggage car les erreurs dépendent de l'ordonnanceur, qu'on ne maîtrise pas.

Condition de concurrence : C'est une situation où deux processus (ou plus) manipulent des données partagées et où le résultat final dépend de l'ordre d'exécution.

Exemple : Spool d'impression :
Un tableau contenant les documents et un compteur prochainLibre.
3 étapes pour imprimer :

- Lire ProchainLibre
- Ecrire le doc dans cette case
- Incrémenter prochainLibre

PL = 10

ProcA :
Lit 10
Ecrit le doc dans 10
ProcB :
Lit 10
Ecrit le doc dans 10
ProcA :
PL = 11
ProcB :
PL = 11

Section critique : Partie du code dans laquelle deux processus ne doivent jamais se trouver simultanément.

Deux objectifs :

1. Identifier les sections critiques
2. S'assurer qu'il n'y a jamais deux processus sur ces sections (Assurer l'exclusion mutuelle)

Problème : Si un code source manipules des variables locales dans cet ordre : Comment sont les sections critiques ?

$$\begin{array}{ccccc} L- & P- & L- & P- & L-\ldots \\ - & C- & - & C- & -\ldots? \\ - & & C & - & -\ldots? \end{array}$$

Les 4 règles d'or

1. Deux processus ne doivent jamais se trouver simultanément en section critique
2. Il ne faut jamais faire de supposition quand à la vitesse et au nombre de processus en oeuvre
3. Aucun processus s'exécutant en section critique ne doit bloquer les autres (ie : il faut se dépêcher de quitter la section critique)
4. Aucun processus ne doit attendre indéfiniment l'accès à une section critique.

Fork : Dans un systeme UNIX, les nouveaux processus sont toujours créés avec la commande fork().

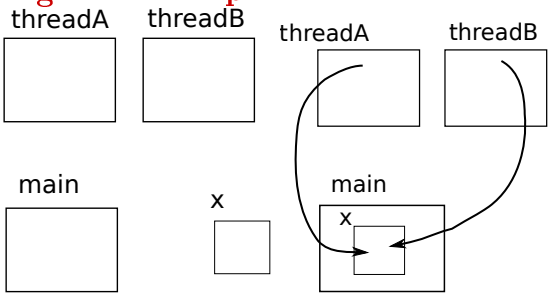
- Crée une nouvelle entrée dans la table des processus
- Copie de l'espace d'adressage
- Copie des descripteurs de fichiers
- Retourne
  - 0 pour le fils
  - un nombre positif strict pour le père

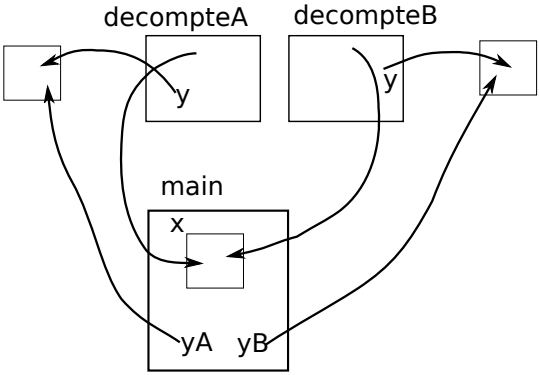
Wait : Le père devrait toujours attendre la fin de l'exécution de son fils avec la fonction waitpid, ou encore wait s'il n'a qu'un seul fils.

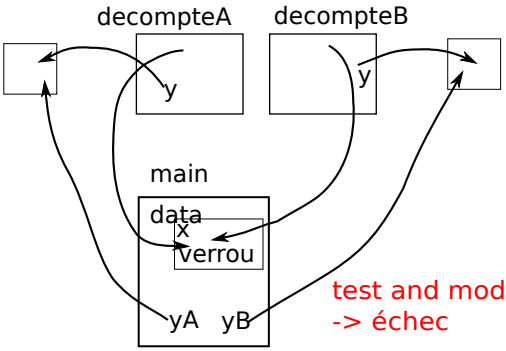
Thread : Par défaut, un processus contient un unique thread (processus = ressource + un thread)
Créer un nouveau thread, c'est créer une nouvelle pile d'exécution à l'intérieur du même processus.

Création d'un thread : On spécifie une fonction dont l'entête est **void* fct(void*)**

Figures correspondantes à la correction du td :







Solution matérielle : Le processus indique au CPU d'ignorer les interruptions.

- Avantage : Plus de préemption imposée
- Inconvénients : Cette méthode est très dangereuse, elle compromet tout le système et exige de stopper tous les CPU.

Solution logicielle 1 : L'alternance stricte Variable "verrou" et attente active. Attention : "Test and Modify" ne marche pas.

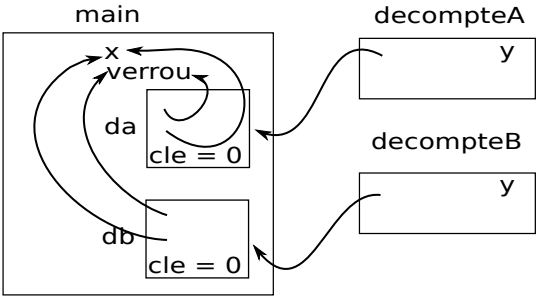
Algo A :

Tantque vrai faire
//section non critique
Tantque verrou != 0 faire
rien
Fin Tantque
verrou ← 1
Fin Tantque

Algo B :

Tantque vrai faire
//section non critique
Tantque verrou != 1 faire
rien
Fin Tantque
verrou ← 0
Fin Tantque

- Avantage : Simple, purement logiciel
- Inconvénient : Alternance stricte (pb résolu par Peterson), forte consommation CPU



Problème producteur / consommateur : On a n processus ou threads qui produisent des données. Ces données sont traitées par m autres processus ou threads. (souvent $n > 1$ et $m = 1$. Exemple : le spool d'impression).
Solution naïve :

- Données :
  - un tableau Tab de taille N. (dans lequel les producteurs écrivent et les consommateurs lisent)
  - un entier lib représentant l'indice de la première case libre pour écrire
  - un entier occ représentant l'indice de la première case occupée pour lire
  - un entier nbocc indiquant le nombre de cases occupées.
  - un entier nblib indiquant le nombre de cases libres.
- Initialisation :
  - lib et occ valent 0, nb_lib = N et nb_occ = 0.
- Producteur :

```
TantQue VRAI faire

  item <- Produire_item()

  TantQue nb_lib == 0 faire
    attendre
  FinTantQue

  nb_lib--
  Tab[lib] <- item
  lib++ (mod N)
  nb_occ++

FinTantQue
```

- Consommateur :

```
TantQue VRAI faire

  TantQue nb_occ == 0 faire
    attendre
  FinTantQue

  nb_occ--
  item <- Tab[occ]
  occ-- (mod N)
  nb_lib++
  consommer_item(item)

FinTantQue
```

Sémaphore : (Dijkstra 1965) Un entier muni de deux fonctions **POST** et **WAIT** et une file d'attente. La file d'attente contient des processus ou threads bloqués sur le sémaphore.

- POST :

```
Si il y a un processus dans la file Alors
  Debloquer le premier processus
Sinon
  Incrémenter l'entier
FinSi
```

- WAIT :

```
Si l'entier est 0 Alors
  Bloque le processus et l'ajouter dans la fill
Sinon
  Décrémenter l'entier
FinSi
```

- Contrainte : Les fonctions POST et WAIT doivent être implémentées de manière à ne jamais être interrompues. On parle d'une solution matérielle ET logicielle.

- Lorsqu'un processus est bloqué il ne consomme aucun temps CPU

- Lorsque des processus sont bloqués sur le sémaphore, son entier vaut forcément 0 (la réciproque est fausse)

Solution à l'exclusion mutuelle :

```
s : Sémaphore := 1
WAIT(s)
  [ Section critique ]
POST(s)
```

Solution du producteur / consommateur :

- Données :
  - un tableau Tab de taille N. (dans lequel les producteurs écrivent et les consommateurs lisent)
  - un entier lib représentant l'indice de la première case libre pour écrire
  - un entier occ représentant l'indice de la première case occupée pour lire

- un sémaphore nbocc qui compte les cases occupées
- un sémaphore nblib qui compte les cases libres
- un sémaphore excl pour gérer l'exclusion mutuelle

- Producteur :

```
Tant Que VRAI Faire

  item <- Produire_item()

  WAIT(nb_lib) //si nb_lib = 0 on bloque, sinon on le décrémente

  WAIT(excl) //début section critique
    Tab[lib] <- item
    lib++ (mod N)
  POST(excl) //fin section critique

  POST(nb_occ) //On débloque un consommateur

FinTantQue
```

- Consommateur :

```
Tant Que VRAI Faire

  WAIT(nb_occ)

  WAIT(excl)
    item <- Tab[occ]
    occ++ (mod N)
  POST(excl)

  POST(nb_lib)
  consommer_item(item)

FinTantQue
```

Mutex : Sémaphore binaire servant uniquement à assurer l'exclusion mutuelle. L'entier est remplacé par deux états possibles : verrouillé ou déverrouillé. Les 2 fonctions s'appellent alors LOCK et

UNLOCK

Sémaphore nommé / non nommé : Un sémaphore nommé apparait dans le système de fichier (plusieurs processus qui ne se connaissent pas peuvent l'utiliser) (c'est un fichier virtuel, les droits d'accès s'appliquent). Un sémaphore non nommé est simplement une variable de notre processus (partagée entre nos threads uniquement).

question 1

Exemple basique de fork : une variable est dupliquée du père au fils.

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

int main()
{
  int pid, x, i;
  x = 0;
  pid = fork();
  if ( pid > 0 )
    {
      for ( i=0; i<10; ++i )
        {
          ++x;
          printf( "Je suis le père et x vaut %d\n", x );
          sleep(1);
        }
      wait( NULL ); /* un père devrait toujours attendre son fils. */
    }
  else
    {
      for ( i=0; i<10; ++i )
        {
          --x;
          printf( "Je suis le fils et x vaut %d\n", x );
          sleep(1);
        }
    }
  return 0;
}
```

question 2

1. Deux threads manipulent la même variable globale

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

int x = 0;

#define THREAD_FUNC(f) (void*(*)(void*))f

void threadA( )
{
  int i;
  for ( i=0; i<10; ++i )
    {
      ++x;
      printf( "Je suis le thread A et x vaut %d\n", x );
      sleep( 1 );
    }
  pthread_exit( NULL );
}

void threadB( )
{
  int i;
  for ( i=0; i<10; ++i )
    {
      --x;
      printf( "Je suis le thread B et x vaut %d\n", x );
      sleep( 1 );
    }
  pthread_exit( NULL );
}

int main()
{
  pthread_t ta, tb;
  pthread_create( &ta, NULL, THREAD_FUNC( threadA ), NULL );
  pthread_create( &tb, NULL, THREAD_FUNC( threadB ), NULL );
  pthread_join( ta, NULL );
  pthread_join( tb, NULL );
  return 0;
}
```

2. Deux threads manipulent la même variable grâce aux pointeurs

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

#define THREAD_FUNC(f) (void*(*)(void*))f

void threadA( int * px )
{
  int i;
  for ( i=0; i<10; ++i )
    {
      ++(*px);
      printf( "Je suis le thread A et x vaut %d\n", *px );
      sleep( 1 );
    }
  pthread_exit( NULL );
}

void threadB( int * px )
{
  int i;
  for ( i=0; i<10; ++i )
    {
      --(*px);
      printf( "Je suis le thread B et x vaut %d\n", *px );
      sleep( 1 );
    }
  pthread_exit( NULL );
}

int main()
{
  int x = 0;
  pthread_t ta, tb;
  pthread_create( &ta, NULL, THREAD_FUNC( threadA ), (void*) &x );
  pthread_create( &tb, NULL, THREAD_FUNC( threadB ), (void*) &x );
  pthread_join( ta, NULL );
  pthread_join( tb, NULL );
  return 0;
}
```

question 3

1. Deux threads décrémentent un même variable

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

#define THREAD_FUNC(f) (void*(*)(void*))f

int * decompte( int * px )
{
  int y;
  int *py;
  while ( *px > 0 )
    {
      ++y;
      --(*px);
    }
  py = ( int* ) malloc( sizeof( int ) );
  *py = y;
  pthread_exit( py );
}

int main( int argc, const char ** argv )
{
  int x = atoi( argv[1] );
  int *ya, *yb;
  pthread_t ta, tb;
  (void) argc; /* ne fait rien, évite un warning si compilé avec -Wextra */
  pthread_create( &ta, NULL, THREAD_FUNC( decompte ), (void*) &x );
  pthread_create( &tb, NULL, THREAD_FUNC( decompte ), (void*) &x );
  pthread_join( ta, (void**) &ya );
  pthread_join( tb, (void**) &yb );
  printf("%d + %d = %d\n", *ya, *yb, *ya + *yb );
  free( ya );
  free( yb );
  return 0;
}
```

2. Nous essayons d'utiliser un verrou pour éviter les problèmes de concurrence (attente active)

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

#define THREAD_FUNC(f) (void*(*)(void*))f

typedef struct
{
  int x;
  int verrou;
} data;


/* La section critique est formée d'une seule ligne :
 * --(*px);
 * car c'est le seul moment où on manipule une donnée qui est commune aux deux
 * threads.
 */
int * decompte( data * pd )
{
  int y;
  int *py;
  while ( pd->x > 0 )
    {
      ++y;
      while ( pd->verrou == 1 )
        { /* ne rien faire */ }
      pd->verrou = 1;
      --(pd->x);
      pd->verrou = 0;
    }
  py = ( int* ) malloc( sizeof( int ) );
  *py = y;
  pthread_exit( py );
}

int main( int argc, const char ** argv )
{
  data d;
  int *ya, *yb;
  pthread_t ta, tb;
  (void) argc; /* ne fait rien, évite un warning si compilé avec -Wextra */
  d.x = atoi( argv[1] );
  d.verrou = 0;
  pthread_create( &ta, NULL, THREAD_FUNC( decompte ), (void*) &d );
  pthread_create( &tb, NULL, THREAD_FUNC( decompte ), (void*) &d );
  pthread_join( ta, (void**) &ya );
  pthread_join( tb, (void**) &yb );
  printf("%d + %d = %d\n", *ya, *yb, *ya + *yb );
  free( ya );
  free( yb );
  return 0;
}
```

Exercice 1 :

1. Condition de concurrence : le résultat final dépend de l'ordonnanceme
2. Section critique : portion de code où deux processus ne devraient jamais se trouver en même temps sous peine de créer une condition de concurrence.

Exercice 2

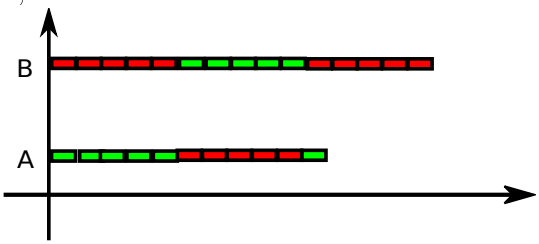
1. La création d'un thread est plus rapide que la création d'un processus.
2. Pour être le plus général possible. Pour plusieurs variables on crée une struct. (idée : créer une fonction bidon qui va répartir les champs de la struct en argument et appeler la fonction qu'on veut)
3. Il sert à enregistrer le retour d'une fonction.
4. La valeur renvoyée par pthread_join et pthread_create est 0 pour un succès, autre chose pour une erreur.

Exercice 3

1. 1,5 secondes
2. 15 + 100*75 = 7515

3. 100*15 + 10*75 = 2250

4. 1,5 secondes



Exercice 4

1. Un thread peut s'activer sur les requetes en cache pendant qu'un autre attend le disque dur

2. L'appel à pthread_create échouera si on dépasse le nombre maximal de thread autorisé

3. .

```
void server_v3()
{
  pthreadt T[NB_THREADS];
  int i;
  for (i=0 ; i<NB_THREADS; ++i)
  {
    pthread_create(T+i, NULL, (void * (*)(void*))faire_le_boulot,NULL);
  }
  for (i=0 ; i < NB_THREADS ; ++i)
  {
    pthread_join(T[i],NULL);
  }
}
void faire_le_boulot
{
  requete* req;
  page_web* page;
  while( pas_termine() )
  {
    attendre_une_requete(\& req);
    chercher_page_en_cache(\& req, \& page);
    if (page_pas_en_cache(\& page))
    {
      lire_page_sur_disque(\& req, \& page);
    }
    envoyer_requete(\& req, \& page);
  }
}
```

Exercice 5 :

1. $A1, A2, A3, A4, A5, B1, B2, B3, B4, B5, B6, B7, A6, A7$
2. La section critique est composée des lignes 4 à 6
3. Inverser les lignes 5 et 6 (la copie du document n'a pas besoin d'être en section critique.

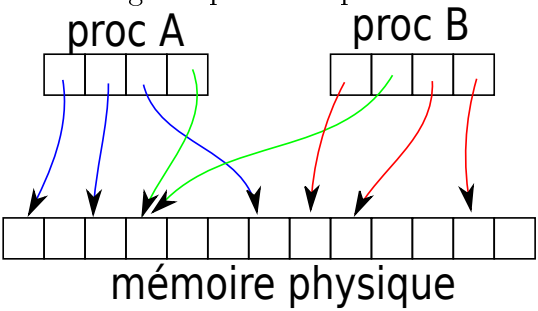
Exercice 6 :

```c
typedef struct
{
  int n;
  int* pTrouve;
  int debut;
  int fin;
} data;

int est_premier(int n)
{
  data d[NB_THREADS];
  pthread_t T[NB_THREADS];
  int trouve = 0;
  int i;
  int rc = racine_carre(n);
  for (i = 0; i < NB_THREADS; ++i)
  {
    d[i].n = n;
    d[i].pTrouve = &trouve;
    d[i].debut = 2 + i*(rc-2)/NB_THREAD;
    d[i].fin = 1 + (i+1)*(rc-2)/NB_THREAD;
    pthread_create(&T[i],NULL,(void *(*)(void*))div_dans_inter,(void*)&d[i]);
  }
  for (i=0; i < NB_THREADS; ++i)
  {
    pthread_join(T[i],NULL);
  }
  return !trouve;
}

void div_dans_inter(data *pd)
{
  int i;
  for (i = pd -> debut; i <= pd -> fin &&(!*pd->pTrouve) ; ++i) {
    if (pd -> n % i == 0) { *pd -> pTrouve = 1; }
  }
}
}
```

Principe : Avoir des pages de mémoire inscrites dans l'espace d'adressage de plusieurs processus.



Avantage : C'est le moyen le plus rapide d'échanger des données entre deux processus.

Inconvénient : C'est aussi un très bon moyen pour créer des conditions de concurrence.

Comment ça marche ?

1. Un processus crée un fichier virtuel (shm_open)
2. On redimensionne le fichier à la taille voulue (ftruncate)
3. Projeter le fichier virtuel dans l'espace d'adressage du processus (mmap)
4. Les processus qui veulent l'utiliser ouvrent le fichier (shm_open) et le projettent dans leur espace d'adressage (mmap)
5. Les processus utilisent cette mémoire partagée (attention aux conditions de concurrence)

6. Lorsqu'un processus a terminé il ferme la projection (munmap) et le fichier (fclose)

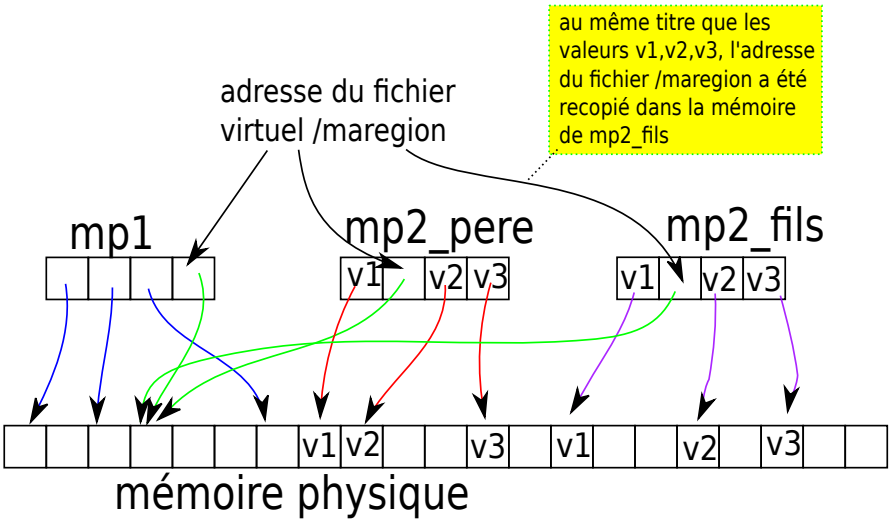
7. Lorsque tous les processus ont terminé, on détruit le fichier (shm_unlink)

Exemple prof : Le processus mp1 crée un espace partagé de la taille d'un entier et demande en boucle à l'utilisateur de changer la valeur de cet entier. Le processus mp2 utilise cet espace partagé et affiche en boucle la valeur de l'entier toutes les secondes.

Modifications apportées pour tester ce qu'il se passe lors d'un fork : Je n'ai pas modifié le fichier mp1. Par contre dans le fichier mp2 j'ai fait un fork avant d'entrer dans la boucle, puis dans la boucle, le processus indique s'il est le père ou le fils avant d'afficher la valeur de l'entier.

```
40      printf("Mémoire partagée projectée à l'adresse : %p\n", (void *) partage );
41      PID = fork();
42      while ( *partage != -1 )
43        {
44          sleep( 1 );
45          if (PID == 0)
46          {
47            printf( "FILS : Valeur de l'entier partagé : %d\n", *partage );
48          }
49          else
50          {
51            printf( "PERE : Valeur de l'entier partagé : %d\n", *partage );
52          }
53        }
```

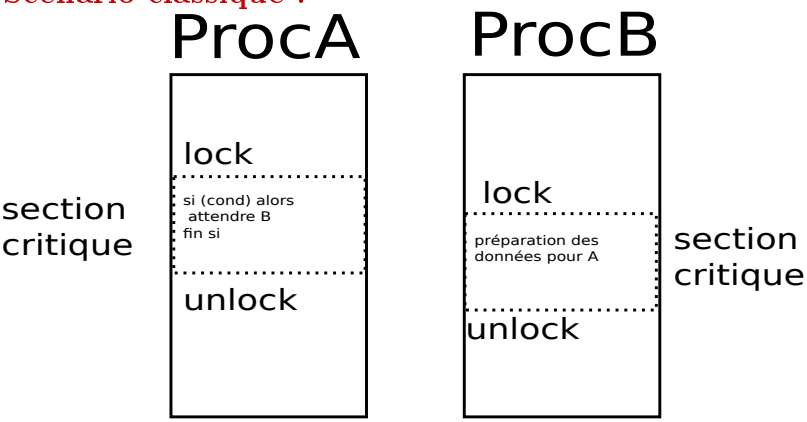
Conclusion : On pouvait naturellement imaginer que les deux processus allaient reconnaitre la valeur de l'entier partagé. En effet, la mémoire est copiée du père au fils, et l'espace mémoire mappé est donc copié avec son mapping, donc dans le fils le mapping vers le fichier virtuel existe toujours. C'est bien ce résultat attendu qui s'est produit.




```
Terminal - celine@celine-serveur: ~/Dropbox/CoursInfo/Synchronisation-622: ~
Fichier  Éditer  Affichage  Terminal  Aller  Aide
cd MemPartage/
celine@celine-serveur:~/Dropbox/CoursInfo/Synchron
ation-622/MemPartage$ ./mp1fork
Mémoire partagée projectée à l'adresse : 0xb77b800
12
11
5
2
10
-1
celine@celine-serveur:~/Dropbox/CoursInfo/Synchron
ation-622/MemPartage$
```

```
Terminal - celine@celine-serveur: ~/Dropbox/CoursInfo/Synchronisatio
Fichier  Éditer  Affichage  Terminal  Aller  Aide
cd ../MemPartage/
celine@celine-serveur:~/Dropbox/CoursInfo/Synchro
nisation-622/MemPartage$ ./mp2fork
Mémoire partagée projectée à l'adresse : 0xb77ab0
00
PERE : Valeur de l'entier partagé : 0
FILS : Valeur de l'entier partagé : 0
PERE : Valeur de l'entier partagé : 0
FILS : Valeur de l'entier partagé : 0
PERE : Valeur de l'entier partagé : 0
FILS : Valeur de l'entier partagé : 0
PERE : Valeur de l'entier partagé : 12
FILS : Valeur de l'entier partagé : 12
PERE : Valeur de l'entier partagé : 12
FILS : Valeur de l'entier partagé : 12
PERE : Valeur de l'entier partagé : 11
FILS : Valeur de l'entier partagé : 11
PERE : Valeur de l'entier partagé : 5
FILS : Valeur de l'entier partagé : 5
PERE : Valeur de l'entier partagé : 5
FILS : Valeur de l'entier partagé : 5
PERE : Valeur de l'entier partagé : 2
FILS : Valeur de l'entier partagé : 2
PERE : Valeur de l'entier partagé : 10
FILS : Valeur de l'entier partagé : 10
PERE : Valeur de l'entier partagé : 10
FILS : Valeur de l'entier partagé : 10
PERE : Valeur de l'entier partagé : -1
FILS : Valeur de l'entier partagé : -1
celine@celine-serveur:~/Dropbox/CoursInfo/Synchro
nisation-622/MemPartage$
```

Scénario classique :



Utilisation : Une **Variable de condition** s'utilise toujours de paire avec un mutex.

3 primitives :

1. cond_wait(C: Variable de condition, M:Mutex). Doit être appelé à l'intérieur d'une section critique protégée par M. Ce que ça fait :

   (a) unlock(M)

   (b) attendre un signal sur C

   (c) lock(M)

   Intérêt de l'appel système pour faire ça : le passage de a à b est atomique, tout comme le passage entre b et c.

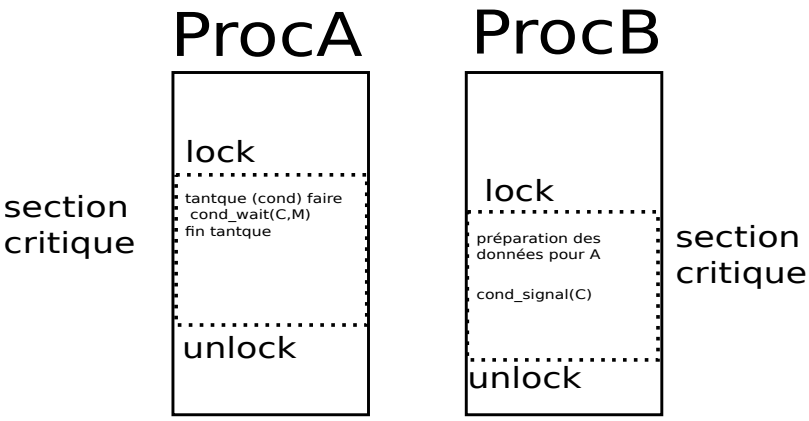
2. cond_signal(C: Variable de condition) Débloque au moins un processus bloqué sur C.

3. cond_broadcast(C: Variable de condition) Débloque tous les processus bloqués sur C.

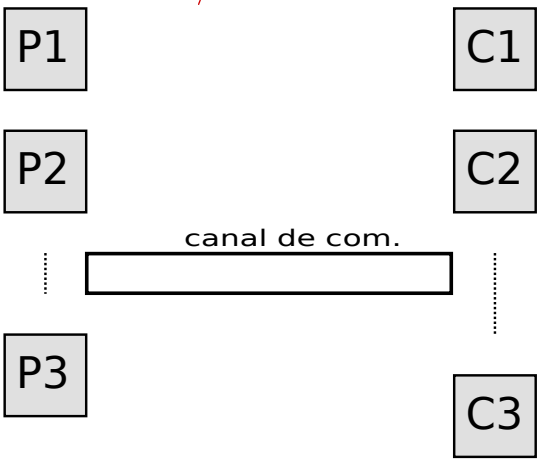
Remarque :

1. Les variables de condition sont sans mémoire. Un signal émis lorsqu'aucun processus n'est en attente est perdu.

2. Un cond_wait s'exécute toujours à l'intérieur d'un tantque.



Producteurs/consommateurs :



Outils de synchronisation :

- Sémaphore
- Mutex
- Variables de condition
- Barrières
- Tubes

Principe des barrières : Une barrière sert à bloquer des processus et à les débloquer lorsqu'un certain nombre ont atteint la barrière.

Une seule primitive : barrier_wait(B:Barriere) bloque le thread jusqu'à ce qu'un nombre suffisant d'appels à barrier_wait aient été effectués.

Descripteurs de fichier : Il s'agit d'un nombre entier utilisé ppour identifier un fichier ouvert par un processus. L'OS maintient pour chaque processus une table des descripteurs.
Par défaut, 3 descripteurs sont associés à notre programme

Descripteur	fichier
0	stdin : lecture du clavier
1	stdout : écriture dans le terminal
2	stderr : écriture dans le terminal

Puis lorsque l'utilisateur réalise la commande open, un nouveau descripteur est créé.

Manipulation :

1. Lecture : read(int fd, void* buf, size_t count). fd est le descripteur, buf est l'espace mémoire où l'OS va écrire les données lues, et count le nombre d'octets à lire. Retourne le nombre d'octets qui ont réellement été lus, et -1 si erreur.

2. Ecriture : write(int fd,const void* buf, size_t count). fd est le descripteur, buf est l'espace mémoire où l'OS va lire les données à écrire dans le fichier, et count le nombre d'octets à écrire. Retourne le nombre d'octets qui ont réellement été écrits, et -1 si erreur.

Tube : Fichier virtuel implémentant un canal de communication unidirectionnel. Il a deux extrémités : une en lecture et une en écriture. On manipule les tubes à l'aide de descripteurs de

fichiers associés aux extrémités.

Fonctionnement :

1. Un read effectué sur un tube vide bloque le thread appelant jusqu'à ce qu'un autre thread y écrive des données
2. Un write effectué sur un tube plein bloque le thread appelant jusqu'à ce qu'un autre thread y lise les données, libérant ainsi de l'espace.
3. L'OS garantit que read et write sont atomiques sur les tubes, à condition que la taille des données ne dépasse pas la taille du tube (constante PIPE_BUF définie dans limits.h) .
4. Pour lire ou écrire dans un tube, il faut que les deux extrémités aient été ouvertes. Un tube dont une des extrémités est fermée est dit cassé.
   (a) Ecrire dans un tube cassé provoque une erreur
   (b) Lire dans un tube vide et cassé ne lit rein et retourne 0.

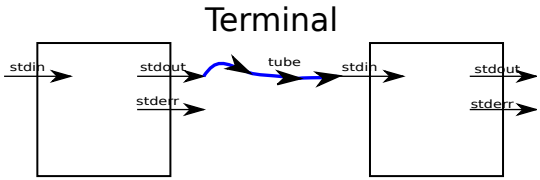
Nommés / non nommés : Un tube nommé apparait dans le système de fichier, contrairement au non nommé.

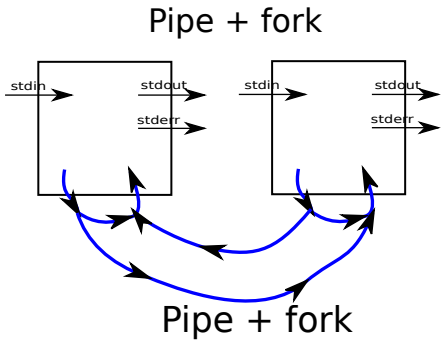
1. Création de Non nommés :
   (a) Dans le terminal en séparant deux programmes par le symbole pipe |

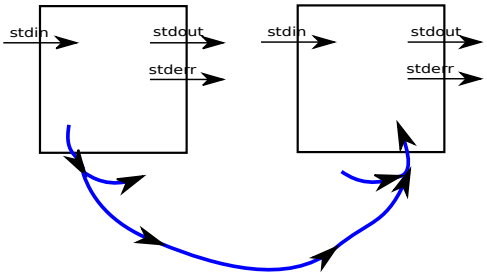
(b) Par la commande int pipe(int df[2]) (df[0] = descripteur du fichier en lecture, df[1] = descripteur du fichier en écriture)

2. Création d'un tube nommé

(a) mkfifo nomFichier







Resource : Objet physique ou virtuel pouvant être alloué à un seul processus à la fois.

Resources retirables : Peut être retirée au processus, utilisé par un autre, puis rendu au premier sans compromettre son exécution (exemples : CPU, RAM).

Resources non retirables : Ne peut être retirée au processus que lorsqu'il en aura décidé.

3 étapes pour accéder à une resource non retirable :

1. Solliciter la resource
2. Utiliser la resource
3. Libérer la resource

Famine : Situation où un processus demande l'accès à une ressource mais ne l'obtient jamais car elle est toujours attribuée à d'autres processus.

Solution : Le principe FIFO permet d'éviter la famine mais n'est pas toujours souhaitable.

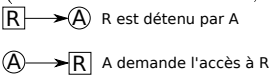
Interblocage : Situation où un ensemble de processus attendent un événement que seul un des processus de l'ensemble peut provo-

quer.

Exemple :

Proc A	Proc B
Lock(M1)	Lock(M2)
Lock(M2)	Lock(M1)
...	...
Unlock(M2)	Unlock(M1)
Unlock(M1)	Unlock(M2)

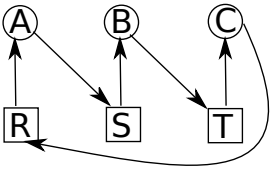
Modélisation du problème : Modèle de Holt (1972) On trace le graphe des attentes qui contient deux types de sommets (carré = ressource, rond = processus)



Exemple :

A	B	C
D(R)	D(S)	D(T)
D(S)	D(T)	D(R)
L(R)	L(S)	L(T)
L(S)	L(T)	L(R)

Si l'ordonnancement est A,B,C,A,C on obtient :



Stratégie face aux interblocages

1. Les ignorer

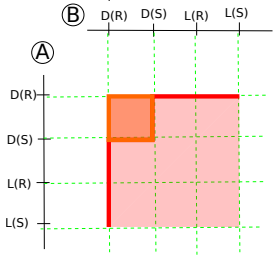
2. Les détecter et y remédier

    - Mettre à jour dynamiquement le graphe des attentes lancer périodiquement un algo de détection des cycles. En cas d'interblocage on tue un des processus et on libère toutes ses ressources.

3. Détecter dynamiquement si il est sûr d'allouer une ressource

    -

A	B
D(R)	D(S)
D(S)	D(R)
L(R)	L(S)
L(S)	L(R)



    On appelle état sûr un état à partir duquel on peut exécuter tous les processus un après l'autre dans un certain ordre sans qu'aucun ne soit bloqué. Un interblocage survient toujours à partir d'un état non sûr. Pour déterminer les états sûrs, il faut connaitre à l'avance le code que vont exécuter les programmes.

4. Imposer des contraintes pour s'assurer qu'un interblocage est impossible

    - Pas d'exclusion mutuelle

- Ordonne les ressources