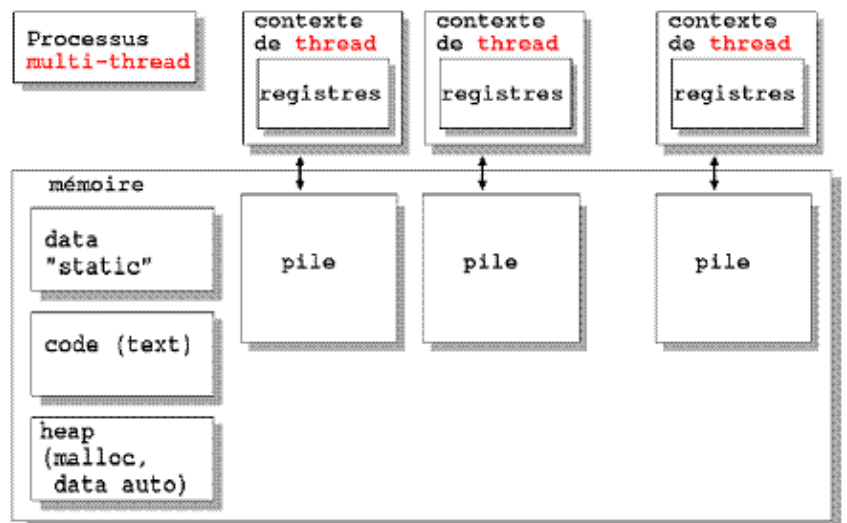


# Les threads

Version 1



Y. CHALLAL, H. BETTAHAR, M. VAYSSADE

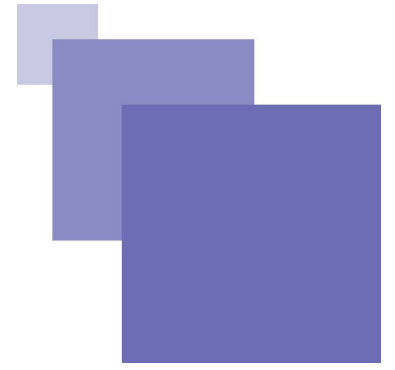


# Table des matières

<b>Objectifs</b>	<b>5</b>
<b>I - Les threads</b>	<b>7</b>
A. Concept de threads.....	<b>7</b>
B. Implémentation des threads.....	<b>10</b>
C. Création de threads.....	<b>11</b>
D. Résiliation d'une Pthread.....	<b>13</b>
E. Synchronisation de Pthreads : pthread_join.....	<b>14</b>
F. Les Pthreads et fork.....	<b>17</b>
G. Pthreads et signaux.....	<b>17</b>
H. L'exclusion mutuelle et mutex.....	<b>20</b>
I. La synchronisation et variables conditionnelles.....	<b>23</b>



# Objectifs



- Utiliser l'API thread POSIX
- Analyser les problèmes de synchronisation de threads



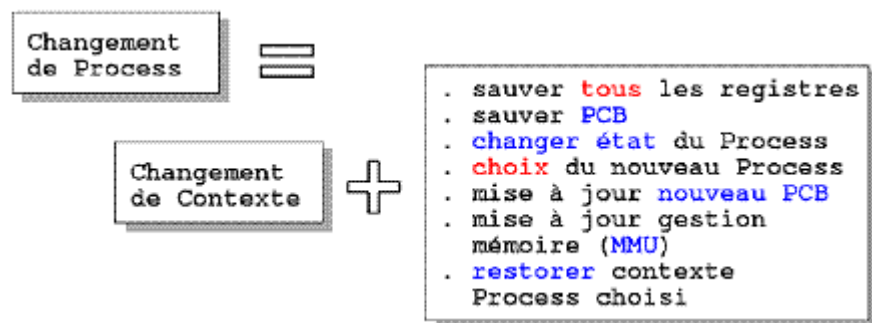
# Les threads

Concept de threads	7
Implémentation des threads	10
Création de threads	11
Résiliation d'une Pthread	13
Synchronisation de Pthreads : pthread_join	14
Les Pthreads et fork	17
Pthreads et signaux	17
L'exclusion mutuelle et mutex	20
La synchronisation et variables conditionnelles	23

## A. Concept de threads

### Processus et threads

La création et la gestion de processus est relativement coûteuse. Les changements de processus nécessitent un travail non négligeable (sauvegarde de PCB, changement d'état, choix d'un autre processus, chargement de son PCB, etc.).



*Changement de processus vs. changement de contexte*

Par ailleurs, un processus constitue un univers plutôt fermé : si partager des données est possible, les mécanismes de protection des données manipulées s'avèrent pénalisant pour certaines applications.

L'idée est donc de raffiner la notion de processus classique afin de rendre naturel le partage de données entre certaines entités relevant de la même application et accélérer les changements de contexte entre elles. Ceci est obtenu en distinguant les deux aspects couverts par la notion de processus :

- **unité d'encapsulation** : le processus constitue le conteneur pour un certain nombre de ressources (espace d'adressage, et descripteurs de

fichiers et IPC par exemple) qu'on peut voir comme une machine virtuelle.

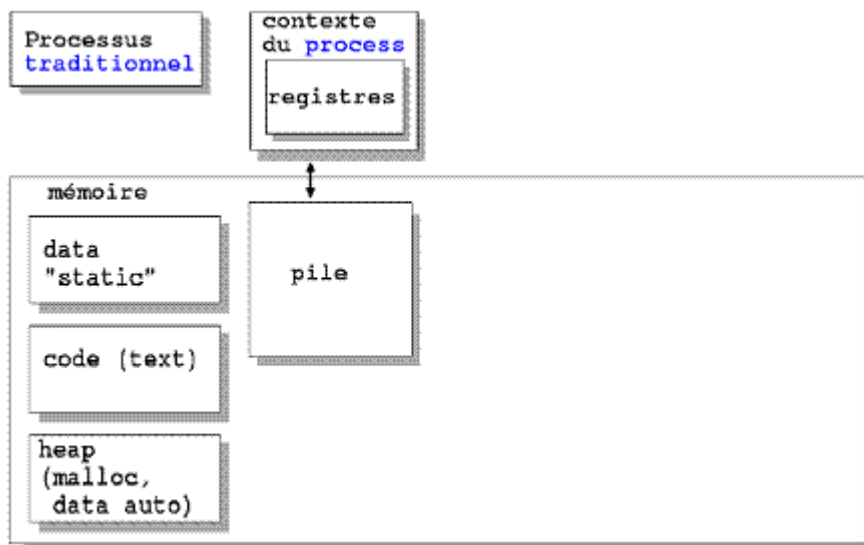
- **unité d'exécution** : un processus correspond à l'exécution d'un programme et c'est cet aspect des processus qui est pris en compte par l'ordonnanceur du système

Une telle unité d'exécution (contexte) est appelée thread. Le processus désignera l'unité d'encapsulation (allocation).

### Ressources d'un processus et ressources d'une thread

Les ressources encapsulées dans un processus sont typiquement :

- un espace d'adressage incluant les différentes régions (data, code, etc.)
- un utilisateur et un groupe propriétaire
- un répertoire de travail
- une table des descripteurs
- des handlers de signaux

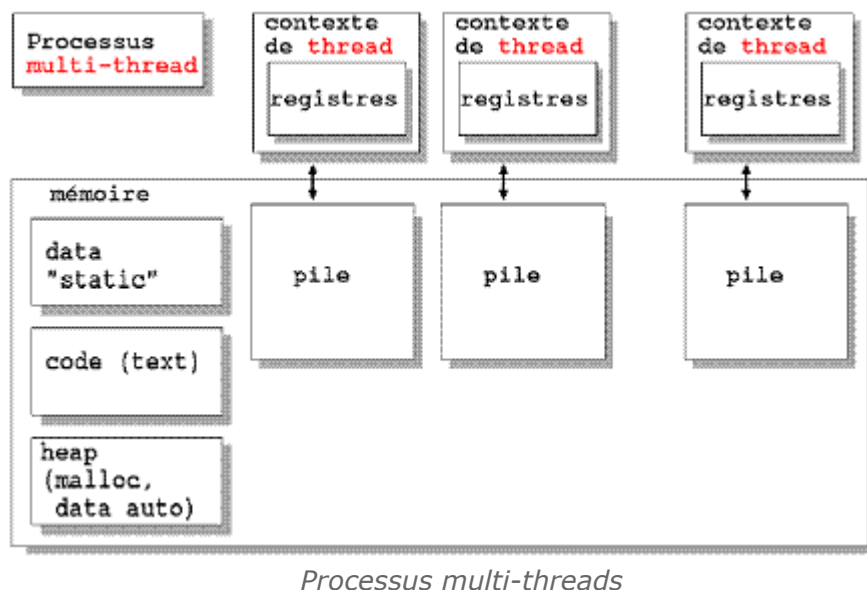


*Processus classique*

Les ressources propres à une thread sont celles qui permettent d'en contrôler l'exécution :

- un contexte d'exécution constitué d'un état des différents registres de la machine, en particulier le compteur ordinal (registre d'instruction) ;
- les attributs d'ordonnement : état, classe d'ordonnement et priorité, temps consommé ;
- la pile d'exécution
- un masque de signaux et des signaux pendants





### Avantages des threads

- Création plus rapide, avec surcôt (overhead) système plus faible que le `fork()` ;
- le changement de contexte de thread à thread est plus rapide car c'est le même espace virtuel qui est partagé entre tous les threads
- des threads multiples peuvent s'exécuter en parallèle sur une machine multiprocesseur. Le bénéfice du parallélisme est aussi apporté sur une machine monoprocesseur par recouvrement des temps d'E/S dans un thread avec des calculs dans un autre ;
- dans une application multi-threads, les échanges de données entre threads se font facilement par accès à l'espace virtuel commun ;

### Inconvénients des threads

La plupart des inconvénients résultent du partage du même espace virtuel entre tous les threads d'un même processus :

- danger d'incohérence dans les accès aux variables communes, nécessité de programmer des sections critiques et de synchroniser les threads
- comportement plus complexe d'un processus multi-threads vis-à-vis des signaux
- comportement plus complexe vis-à-vis des appels systèmes
- certains appels n'affectent qu'un seul thread, d'autres les affectent tous ;
- nécessité d'écrire les fonctions utilisées par plusieurs threads d'une façon ré-entrante ;
- précautions supplémentaires à prendre lors de l'appel à certaines fonctions système ; par exemple, les descripteurs des fichiers ouverts sont partagés : un thread peut lire / écrire dans un fichier ou déplacer le pointeur (`seek`), sans que les autres threads ne le sachent ;
- obligation pour les programmeurs du système d'exploitation de fournir une version ré-entrante de la plupart des bibliothèques du système ;

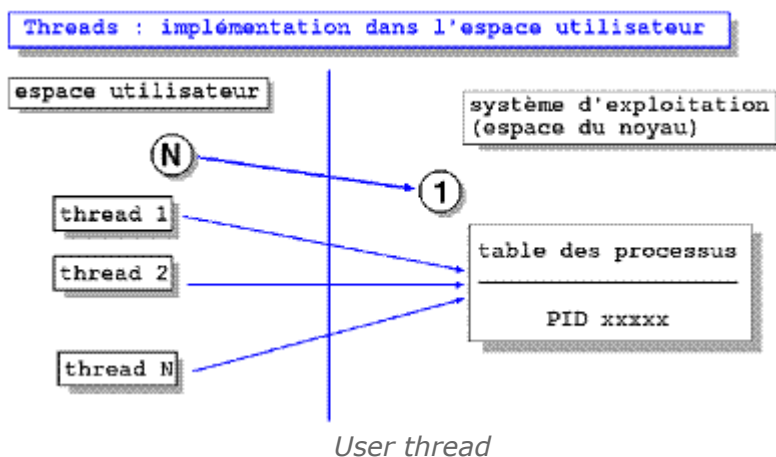
## B. Implémentation des threads

Il existe deux approches différentes :

- kernel threads
- user threads

### User threads

Une bibliothèque permet la gestion des threads en mode utilisateur. Dans ce cas, l'état d'une thread est maintenu en espace utilisateur. Aucune ressource du noyau n'est allouée à une thread. Le système considère un processus une seule thread : tout appel système bloquant aura pour effet de bloquer le processus et de le rendre non éligible : aucune thread du processus ne pourra donc poursuivre son exécution.

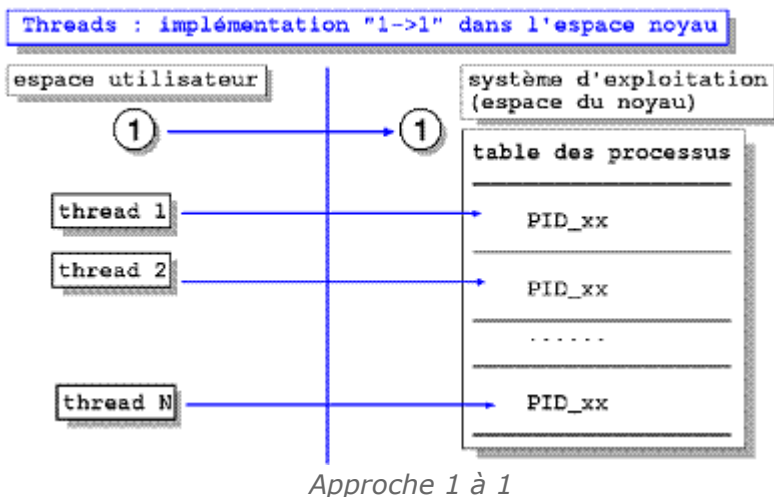


### Kernel threads

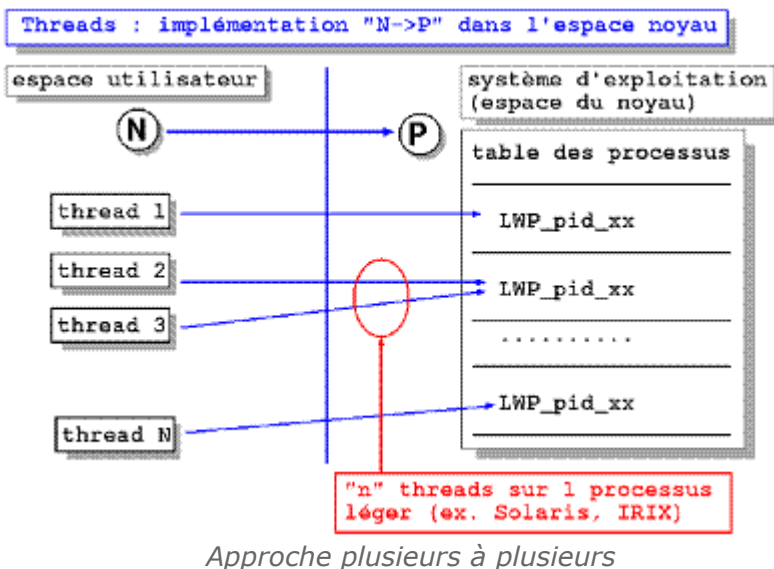
Dans ce cas, les threads sont des entités du système : le noyau possède alors un descripteur pour chaque thread et l'ordonnanceur du système travaille au niveau de ces entités. Dans le cas d'une architecture multi-processeur, cela permet une utilisation fine des différents processeurs.

Il existe plusieurs approches pour implémenter les threads dans ce cas :

- **L'approche 1-1** : à chaque thread utilisateur correspond une thread noyau. L'avantage de cette approche est que les threads sont traitées individuellement par le noyau (par exemple un appel système bloquant ne bloque que la thread appellante. C'est l'approche adoptées dans le noyau Linux 2.6.



- **L'approche M-P :** différentes threads utilisateurs sont multiplexées (sur un nombre inférieur ou égal) de threads noyau (appelés : light weight processes). C'est la solution adoptées dans Solaris.



## C. Création de threads

### Norme POSIX

Nous considérons l'interface proposée dans POSIX pour manipuler ce qu'il est d'usage d'appeler Pthreads.

### Identification d'une Pthread

Une Pthread est identifiée par un descripteur de type opaque pthread\_t.

### Attributs d'une pthread [jmrifflet]

Lors de la création d'une Pthread, un certain nombre de ses attributs sont transmis sous le type opaque pthread\_attr\_t.

La manipulation d'une variable de type pthread\_attr\_t se fait à travers :

- la fonction :  

```
int pthread_attr_init(pthread_attr_t *attributs) ;
```

qui permet d'affecter une valeur par défaut à chacun des attributs codés dans l'objet d'adresse spécifiée ;

- des primitives dédiées aux différents attributs permettant d'en consulter ou d'en modifier les valeurs :

Chaque attribut codé dans un objet de type `pthread_attr_t` possède un nom. Les primitives `pthread_attr_getnom` et `pthread_attr_setnom` permettent respectivement d'extraire d'une variable de type `pthread_attr_t` la valeur de l'attribut ou d'y modifier la valeur de cet attribut.



### Exemple : Manipulation de l'attribut detachstate [jmrifflet]

Les fonctions permettant d'accéder à ou de modifier l'information relative à l'attribut `detachstate` ont le prototype suivant :

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *etat) ;
```

```
int pthread_attr_setdetachstate(const pthread_attr_t *attr, int etat) ;
```

Un appel à la fonction de type `get` permet de récupérer, à l'adresse spécifiée, l'état vis-à-vis de l'attribut `detachstate` : `PTHREAD_CREATE_JOINABLE` (joignable), `PTHREAD_CREATE_DETACHED` (non joignable). Un appel à la fonction de type `set` affecte la valeur état donnée au champ codant cet attribut dans la variable d'adresse spécifiée. Les deux fonctions renvoient 0 en cas de réussite et une valeur positive correspondant à la nature de l'erreur sinon.



### Méthode : Création d'une Pthread

Un appel à la fonction

```
#include <pthread.h>
int pthread_create(pthread_t *pthread, pthread_attr_t
*attr, void *(*fonction)(void*), void *arg) ;
```

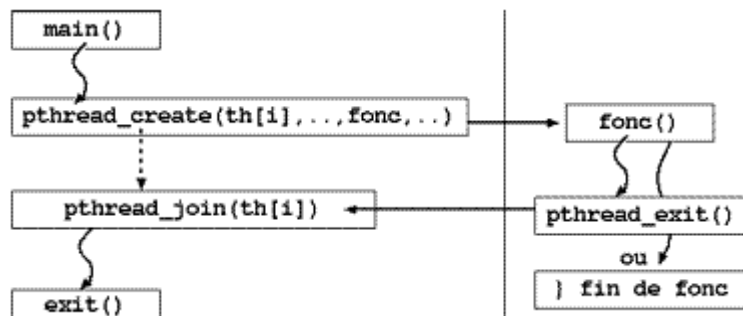
correspond à la demande de création d'une nouvelle Pthread dans le processus auquel la Pthread appelante appartient.

Les attributs de la nouvelle Pthread sont déduits de la valeur de `attr*` (la valeur `NULL` donne les valeurs par défaut aux attributs).

La nouvelle Pthread (dont l'identité est récupérée à l'adresse `pthread`) exécute la fonction spécifiée avec `arg` comme seul paramètre.

Un retour de la fonction exécutée par une Pthread provoque la terminaison de cette Pthread : un appel à la fonction `pthread_exit` est réalisé.

Si l'appel réussit, la fonction renvoie 0 au retour. En cas d'erreur, une valeur non nulle indiquant directement l'erreur est renvoyée (`ENOMEM`, `EINVAL` ou `EPERM`).



Création d'une pthread



### Méthode : Terminaison d'une Pthread [jmrifflet]

Un appel à la fonction

```
int pthread_exit(int *statut) ;
```

termine la Pthread appelante avec une valeur de retour égale à statut. Nous verrons que cette valeur est accessible aux autres Pthreads dans le même processus par l'intermédiaire de la fonction `pthread_join`.

### Devenir d'une Pthread après un `pthread_exit` [jmrifflet]

Ceci dépend de la valeur de l'attribut `detachstate` qui peut avoir deux valeurs :

- `PTHREAD_CREATE_JOINABLE` : l'orsqu'une Pthread possédant cet attribut se termine suite à un appel de `pthread_exit`, son identité et sa valeur de retour sont conservées jusqu'à ce qu'une autre en prenne connaissance (via la fonction `pthread_join`), ce qui libérera ces dernières ressources.
- `PTHREAD_CREATE_DETACHED` : La Pthread est dit détachée. Lorsqu'elle termine, toutes les ressources qu'elle possédait sont libérées. Aucune Pthread ne peut en être informée (Les appels à `pthread_join` échoueront).



#### Remarque : Modification de l'attribut `detachstate`

Cet attribut est initialisé à `PTHREAD_CREATE_JOINABLE`. Il est possible de détacher une Pthread d'identité `tid` en appelant :

```
int pthread_detach(pthread_t tid) ;
```

## D. Résiliation d'une Pthread

### `pthread_cancel`

il est possible pour un thread d'envoyer à un autre thread un ordre de terminaison par :

```
int pthread_cancel(pthread_t tid) ;
```

Mais ceci doit être utilisé avec précaution, car si un thread se termine ainsi brutalement, sans passer par une fonction de libération de ressources, toutes les ressources qu'il possédait ne sont pas libérées automatiquement par le noyau, les laissant ainsi dans un état incohérent.

### Comportement de `pthread_cancel`

Le comportement de `pthread_cancel` dépend du type de comportement que la thread a sélectionné grâce aux deux fonctions suivantes :

- `int pthread_setcancelstate(int state, int *oldstate) ;`
  - `state=PTHREAD_CANCEL_DISABLE`, requêtes de terminaison ignorées
  - `state=PTHREAD_CANCEL_ENABLE`, autorise la terminaison
- `int pthread_setcanceltype(int type, int* oldtype) ;`
  - `type=PTHREAD_CANCEL_ASYNCHRONOUS` ; terminaison immédiate du thread cible
  - `type=PTHREAD_CANCEL_DEFFERED` ; suspendre la terminaison jusqu'au prochain point de suspension.

Les threads sont créés par défaut avec la terminaison possible et différée.

Il y'a une liste de fonctions (telle que `pause`, `msgrcv`, `pthread_cond_wait`, etc.) qui sont des points de terminaison. En particulier, la fonction `pthread_testcancel` permet au programmeur d'un thread d'insérer des points d'arrêt possibles à des endroits "non dangereux".

## E. Synchronisation de Pthreads : pthread\_join

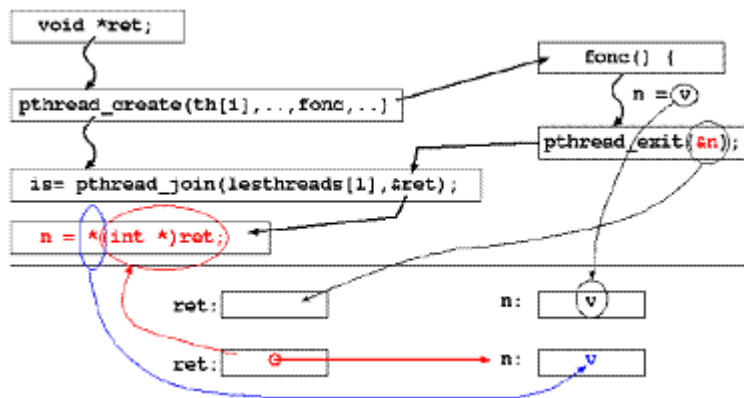
### pthread\_join

Une Pthread peut se mettre en attente de la terminaison d'une autre (appartenant au même processus) en appelant la fonction

```
int pthread_join(pthread_t tid, void *valeur) ;
```

Un tel appel bloque la Pthread appelante jusqu'à ce que la Pthread d'identité tid se termine à condition que celle-ci ne soit pas en mode détaché.

Au retour réussi d'un appel à la primitive, on récupère à l'adresse valeur l'adresse de la valeur de retour de la Pthread cible (l'adresse transmise par celle-ci lors de l'appel à pthread\_exit).



Synchronisation de threads par join



### Exemple : Deux façons de retour d'une fonction de Pthread

Une Pthread peut faire un retour de sa fonction de deux manières : pthread\_exit ou return classique

```
void *ret;

is= pthread_create(&leasthreads[0], NULL, func_thi, (void *)0);
if (is!=0) {fprintf(stderr, "%s", strerror(is)); exit(1);}

void *func_thi( void *num) {
    int nu, n=10000;
    nu = (int) num;
    n = nu + n;
    fprintf(stdout, "thi %d %d ", nu, n);
    pthread_exit(&n);
}

is= pthread_join(leasthreads[0], &ret);
if (is!=0) {fprintf(stderr, "%s", strerror(is)); exit(1);}
if (ret != PTHREAD_CANCELED) {
    n = *(int *)ret; printf("ret de thi= %d\n", n); }
    // cast du void *ret en int *
    // déréférence du int *
```

Retour standard d'une Pthread

```

void *ret;

is= pthread_create(&lesthreads[1],NULL,func_thj,(void *)1);

void *func_thj( void *num) {
    int nu,n=10000;
    static int rr; /* pour retour thj */
    nu = (int) num;
    n = nu + n;
    fprintf(stdout,"thj %d %d ",nu,n);
    rr = n;
    return((void *)&rr);
}

is= pthread_join(lesthreads[1],&ret);
if (ret != PTHREAD_CANCELED) {
    n = (int)ret; /* si return((void *)&rr) */
    n = *(int *)ret; /* si return((void *)&rr) */
    printf("ret de thj= %d\n",n); }

```

*Retour d'une Pthread avec return*



### Remarque : Un seul pthread\_join par thread cible

Un seul Pthread peut réussir l'appel pour une Pthread donnée et en particulier récupérer la valeur de retour de cette Pthread. Pour les autres, la fonction renverra la valeur ESRCH. Si la thread a été détachée l'appel à pthread\_join échoue également en retournant EINVAL.

## F. Les Pthreads et fork

### Que duplique fork() ?

Avec l'ajout du multi-thread dans un système d'exploitation, se pose aux concepteurs la nécessité de décider quoi faire du fork :

- dupliquer tous les threads du processus
- dupliquer le processus avec un seul thread, celui qui a exécuté le fork().

Après certaines expériences, la première solution a été abandonnée vus qu'elle provoque des interblocage, et donc seule la Pthread appelante est dupliquée dans le nouveau processus.

### Comportement de exec

Si une Pthread recouvre le code et les données du processus par un appel à l'une des primitives exec, toutes les autres Pthreads du processus se terminent. La Pthread appelante, quant à elle, exécute la fonction main du programme chargé.

## G. Pthreads et signaux

### Pthreads et signaux

Les signaux internes générés par l'exécution d'un thread (e.g. SIGSEGV) sont délivrés à la thread qui a provoqué le signal.

Mais les signaux externes (envoyés au processus par kill(pid, sig)) sont, selon la norme POSIX reçus par l'ensemble du processus. Un tel signal sera pris en compte par une thread du processus recevant le signal parmi celles qui sont bloquées dans un appel à sigwait et qui ne masquent pas le signal : en effet, chaque Pthread possède son propre masque de signaux (primitive pthread\_sigmask). Par ailleurs les dispositions prises par un processus en terme de handler (au travers de la

primitive sigaction) sont partagées par toutes les Pthreads du processus.

### Envoie d'un signal à une Pthread

La primitive pthread\_kill permet d'envoyer un signal à une autre thread :

```
int pthread_kill(pthread_t Pthread, int signal) ;
```

C'est une requête d'envoi du signal spécifié à la Pthread.

### Masque de signaux d'une Pthread

Chaque Pthread possède son propre masque de signaux et hérite à sa création du masque de la Pthread la créant. Un appel à la primitive :

```
int pthread_sigmask(int comment, sigset_t *pEns, *sigset_t *pEnsAncien) ;
```

permet de consulter ou de modifier le masque de signaux de la Pthread appelante.



### Exemple : Comment contrôler les signaux dans un processus multi-thread

Si on ne veut pas que les signaux externes perturbent le fonctionnement du programme, on va procéder de la façon suivante :

- dans tous les threads, sauf un, bloquer tous les signaux
- dans un thread, dédié à la gestion des signaux :
  - bloquer tous les signaux sauf les signaux acceptés ;
  - se mettre en attente des signaux dans sigwait() ;
  - à la sortie de sigwait() ; traiter le signal arrivé et reboucler sur sigwait() ;

Ainsi, le thread qui traite les signaux est toujours en attente, mais ne fait pas le traitement dans une fonction de capture. Il peut donc utiliser toutes les fonctions pthread pour interagir avec les autres threads.

```
char *ch="AMa";
int ajout = 0;
main()
{ int is, i;
  pthread_addr_t val=0;
  init(); /* init et créer les threads */

  for(i=0; i<nth; i++) /* attendre fin des threads */
  {
    is = pthread_join( threads[i], &val); }
  printf("\n");
}

int init() {
  int is, i;
  /* créer le thread "signaux" */
  is= pthread_create(&thsig, pthread_attr_default,
    (pthread_startroutine_t)th_sig, NULL);
  ifer (is,"err. création thread");
  /* créer les threads */
  for(i=0; i<nth; i++)
  { printf("création thread %d ",i);
    is= pthread_create(&threads[i],pthread_attr_default,
      (pthread_startroutine_t)th_fonc,
      (pthread_addr_t)i );
    ifer (is,"err. création thread");
  }
  printf("=>\n");
}
```

Exemple gestion des signaux (fonction main)





en appelant `pthread_mutex_lock`, soit en mode non bloquant en appelant `pthread_mutex_trylock` :

```
int pthread_mutex_lock(pthread_mutex_t *mutex) ;
int pthread_mutex_trylock(pthread_mutex_t *mutex) ;
```

### Déverrouillage d'un mutex

Un mutex peut être libéré par la Pthread qui l'a verrouillé en appelant la fonction :

```
int pthread_mutex_unlock(pthread_mutex_t* mutex) ;
```



### Exemple : Producteurs / consommateur et Pthreads

Considérons le programme suivant où deux producteurs incrémentent deux variables partagées et le consommateur les affiche à l'écran. La séquence d'incrémentation des deux variables `ia` et `ja` est une section critique et doit être exécutée en exclusion mutuelle. Cet exemple illustre une exécution erronée :

```
int ia=0,ja=0;

void *prod1(void *arg) {
    while(true) {
        ia++; ja++; sleep(3); }
}
void *prod2(void *arg) {
    while(true) {
        ia++; sleep(2); ja++; }
}
void *cnsmr(void *arg) {
    while(true) {
        sleep(1);
        cout <<" " << ia <<" " << ja <<" " << endl; }
}

int main() {
    pthread_t pth1, pth2, cth;
    void *rval;
    int r;
    r=pthread_create(&pth1,0,prod1,0);
    r=pthread_create(&pth2,0,prod2,0);
    r=pthread_create(&cth,0,cnsmr,0);
    sleep(10); // run for 10 seconds
}
```

*Producteurs / consommateur et Pthreads*

Une exécution de ce code peut donner lieu à une désynchronisation de `ia` et `ja`.

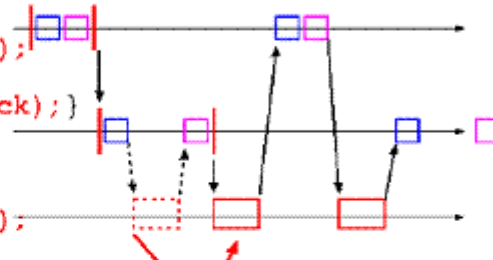
On résout le problème en protégeant la section critique avec un mutex, comme suit :

```

#include <pthread.h>
pthread_mutex_t lck;
int ia=0,ja=0;
void *prod1(void *arg) {
    while(true) {
        pthread_mutex_lock(&lck);
        ia++; ja++; sleep(2);
        pthread_mutex_unlock(&lck);
    }
}
void *prod2(void *arg) {
    while(true) {
        pthread_mutex_lock(&lck);
        ia++; sleep(1); ja++;
        pthread_mutex_unlock(&lck);
    }
}
void *cnsmr(void *arg) {
    while(true) {
        sleep(1);
        cout <<" "<< ia <<" "<< ja <<" "<< endl;
        pthread_mutex_unlock(&lck);
    }
}
int main() {
    .... pthread_mutex_init(&lck, 0); ....
}

```

Producteurs / consommateur et mutex



## I. La synchronisation et variables conditionnelles

### Exclusion mutuelle et synchronisation

L'exclusion mutuelle empêche les autres processus / threads de modifier ou toucher en même temps à une variable.

La synchronisation a pour rôle de savoir si un autre processus / thread a fait quelque chose (terminé un calcul, produit un item, modifié une structure, etc.).

Pour assurer la synchronisation, on utilise dans le cadre des threads POSIX, les variables conditionnelles : `pthread_cond_wait`, et `pthread_cond_signal` :

- `pthread_cond_wait` : met le thread en attente de la réalisation de la condition ;
- `pthread_cond_signal` : réveille un thread en attente dans la queue associé à la condition.



### Méthode : Condition et mutex

Etant donné une variable `var`, on souhaite attendre que cette variable change d'état et satisfasse une condition particulière avant de poursuivre l'exécution.

On associe "logiquement" à cette condition réelle d'une part d'une part un mutex `var_mutex` et d'autre part une variable de condition `var_cond`. Le mutex est utilisé pour assurer la protection des opérations sur la variable `var` elle-même et la variable de condition permet d'en transmettre les changements d'état.

Une utilisation type d'une variable de condition est la suivante, du côté où l'on attend qu'une certaine propriété d'une variable `var` (ici de type `int`) soit vérifiée :

```

pthread_cond_t var_cond ;
pthread_mutex_t var_mutex ;
int var ;
...
pthread_mutex_lock(&var_mutex) ; //verrouillage du mutex

```

```
while( !condition(var)){
    /* si la condition n'est pas remplie */
    /* attendre qu'un changement d'état soit signalé */
    pthread_cond_wait(&var_cond, &var_mutex) ;
}
/* exploiter la variable */
....
/* déverrouiller le mutex */
pthread_mutex_unlock(&var_mutex) ;
...
```

Ce qui suit est le code utilisé par une Pthread modifiant l'état de la variable contrôlée et signalant ce changement d'état :

```
...
/* changement de valeur de var sous protection du mutex */
pthread_mutex_lock(&var_mutex) ;
var=nouvelle_valeur ;
pthread_mutex_unlock(&var_mutex) ;
/* on peut éventuellement tester si la condition sur var
est vraie */
pthread_cond_signal(&var_cond) ;
...
```



**Attention : Sortie de l'attente d'une condition et exclusion mutuelle**

Le point important dans le canevas ci-dessus est que lors de l'attente d'une condition (par un appel pthread\_cond\_wait), la Pthread appelante libère le sémaphore associé : elle l'acquerra de nouveau automatiquement (et de manière exclusive) lors de sa sortie de cet appel.

**Initialisation d'une condition**

Une variable de condition de type pthread\_cond\_t peut être initialisé en appelant la fonction :

```
int pthread_cond_init(pthread_cond_t* cond,
pthread_condattr_t* attr) ;
```

dans laquelle une valeur NULL de attr donne à la variable cond des attributs par défaut.

**Attente de condition**

Une Pthread ayant verrouillé le mutex peut se mettre en attente sur une condition par un appel à la fonction

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
*mutex) ;
```

Les effets d'un tel appel sont les suivants :

- le mutex spécifié est libéré ;
- la Pthread est mise en attente sur la variable de condition cond ;
- lorsque la condition est signalée par une autre Pthread, le mutex désigné est acquis de nouveau par la Pthread en attente qui reprend son exécution.

**Notification**

Une Pthread peut notifier une condition par un appel à l'une des deux fonctions :

```
int pthread_cond_signal(pthread_cond_t *cond) ;
```

```
int pthread_cond_broadcast(pthread_cond_t *cond) ;
```

Un appel à la fonction `pthread_cond_signal` permet le réveil d'au moins l'une des pthreads en attente sur `cond` (s'il en existe) alors qu'un appel à la primitive `pthread_cond_broadcast` les réveille toutes.



### Attention : Variable de condition sans état

Les variables de condition sont sans état : si aucune Pthread n'est en attente sur `cond` lors de la notification, cette notification est perdue.



### Exemple : Producteurs / consommateur et variables conditionnelles

Considérons le problème de producteurs / consommateur où un producteur écrit dans un buffer de taille `BUFFER_SIZE` tant qu'il n'est pas plein et un consommateur lit à partir du même buffer tant qu'il n'est pas vide. En utilisant les variables conditionnelles, le problème peut être résolu comme suit :

```
// safecond.cpp
#include <iostream.h>
#include <unistd.h>
#include <pthread.h>
#include <cstdlib>
#define BUFFER_SIZE 4
int buffer[BUFFER_SIZE];
int readpos=0, writepos=0, count=0;
pthread_cond_t notempty;
pthread_cond_t notfull;
pthread_mutex_t lck;

void produce(int &x) { sleep(1); x=x+1;
  cout << "produce:" << x << " count=" << count << "\n";
}
void consume(int x) { sleep(2);
  cout << "consume:" << x << " count=" << count << "\n";
}
int main() { pthread_t pth, cth;
  void * retval;
  pthread_cond_init(&notempty, 0);
  pthread_cond_init(&notfull, 0);
  pthread_mutex_init(&lck, 0);
  readpos=0; writepos=0; count=0;
  pthread_create(&pth,0,prod,0);
  pthread_create(&cth,0,cnsmr,0);
  sleep(15);
}
```

*Producteur / consommateur et variables conditionnelles (main)*

```

void *prod(void *data) { int item=0;
while(true) {
produce(item);
pthread_mutex_lock(&lck);
while(count == BUFFER_SIZE) {
    cout << "producer bloqué" << endl;
    pthread_cond_wait(&notfull, &lck);
}
buffer[writepos] = item;
count++; writepos++;
if(writepos>=BUFFER_SIZE) writepos=0;
pthread_cond_signal(&notempty);
pthread_mutex_unlock(&lck);
}
}
void *cnsmr(void *data) { int item;
while (true) {
pthread_mutex_lock(&lck);
while(count == 0) {
pthread_cond_wait(&notempty, &lck); }
item = buffer[readpos];
count--; readpos++;
if(readpos>=BUFFER_SIZE) readpos=0;
if (count == BUFFER_SIZE-1) {
    cout << "producer libéré" << endl;
    pthread_cond_signal(&notfull);
}
pthread_mutex_unlock(&lck);
consume(item);
}
}

```

*Fonctions producteur et consommateur*