Programmation multi-threads

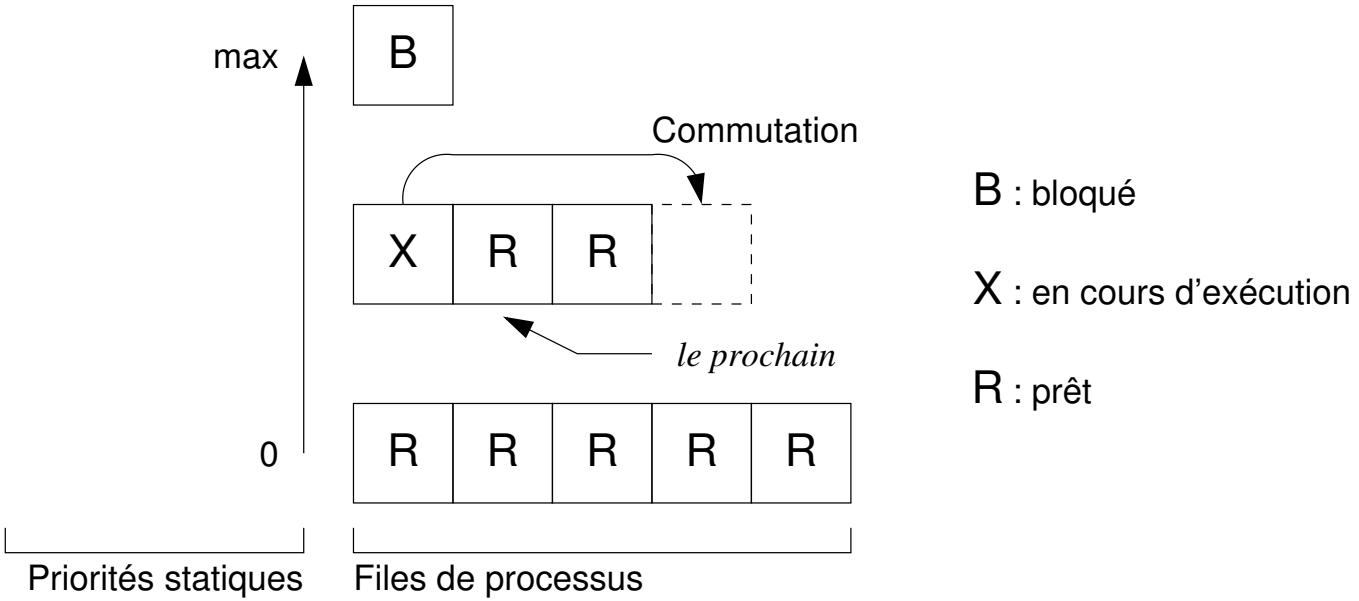
Ordonnancement
Principes
Threads Posix
Synchronisation

Fabrice Harrouet
École Nationale d'Ingénieurs de Brest
harrouet@enib.fr
http://www.enib.fr/~harrouet/

L'ordonnancement des processus

- **Contrôlé par les fonctionnalités de `sched.h`**
  - *"Comment choisir le processus à exécuter ?"*
  - *"Jusqu'à quand ?"*
  - Choix d'un mode d'ordonnancement par processus (pas pour l'ensemble)
  - Géré par files de priorités (statiques)
  - Éventuellement des prioriétés dynamiques

L'ordonnancement des processus



- **Pour qu'un processus soit exécuté**
  - Aucun processus de priorité statique $>$ ne doit être prêt
  - Il doit être le premier de sa file

L'ordonnancement des processus

- **Commutations implicites**
  - Appels systèmes bloquants
  - Préemptions par l'ordonnanceur (horloge)
  - Terminaison du processus
- **Commutations explicites**
  - Appel système `sched_yield()` (`man 2 sched_yield`)
    `int sched_yield(void);`
  - Cède le processeur à un autre processus
  - Passe à la fin de la file de sa priorité statique
  - Si aucun autre processus de priorité statique $\geq$ n'est prêt
    $\rightarrow$ continuer sans commuter
  - Retour : `0` si ok, `-1` si erreur (cause ?)

L'ordonnancement des processus

- **Trois modes d'ordonnancement** (`man 2 sched_setscheduler`)
  - Choix du mode par processus, pas pour l'ensemble
  - Choix d'une priorité statique
    (mini./maxi. dépendantes du système et du mode)
  - `SCHED_FIFO`
    - Mode à tendance *"temps-réel"*
    - Uniquement des priorités statiques $>$ `0`
    - Pas de préemption par horloge $\rightarrow$ Multi-tâches coopératif
    - Risque de bloquage (pour priorités statiques $\leq$) !
    - Nécessite des privilèges (`root`)

L'ordonnancement des processus

- **Trois modes d'ordonnancement**
  - `SCHED_RR` (*Round Robin*)
    - $\equiv$ `SCHED_FIFO` + préemption par une horloge
    - Mode à tendance *"temps-réel"*
    - Uniquement des priorités statiques $>$ `0`
    - Préemption par une horloge $\rightarrow$ Multi-tâches préemptif
    - Risque de bloquage (pour priorités statiques $<$) !
    - Nécessite des privilèges (`root`)

L'ordonnancement des processus

- **Trois modes d'ordonnancement**
  - `SCHED_OTHER`
    - Pour la majorité des processus
    - Optimisé pour un temps de réponse et un rendement global (pas un processus au détriment des autres)
    - Priorité statique = `0`
    - Préemption par une horloge → Multi-tâches préemptif
    - Repose sur une priorité *dynamique* dépendante du système (une quantité fixe et une autre évoluant selon les commutations)
    - Procédé spécifique à chaque système (quelquefois ≡ `SCHED_RR` avec la priorité statique `0`)

Généralités sur les threads

- **Donner plusieurs activités à un même processus**
  - Mener plusieurs traitements bloquants
  - Gagner du temps sur une machine multi-processeurs
- **Plus efficace que plusieurs processus**
  - Commutation plus efficace (*processus légers*)
  - Espace d'adressage commun (communication simplifiée)
- **Informations propres à chaque thread**
  - Pile d'exécution, valeurs des registres
- **Informations partagées au sein du processus**
  - Code, données, descripteurs de fichiers
- **Partage dépendant de l'implémentation**
  - Signaux, propriétés ...

Plusieurs implémentations

- **Espace noyau**
  - Gérés par le système (comme des processus)
  - Propriétés bien séparées
  - Permet l'utilisation de plusieurs processeurs
  - Priorités et modes d'ordonnancement des processus
  - Commutation des threads $\simeq$ commutation des processus
- **Espace utilisateur**
  - Propriétés communes car inconnu du noyau
  - Ordonnancement *"applicatif"* à l'intérieur du processus
  - Appel bloquant dans un thread
    $\rightarrow$ risque de blocage du processus
  - Utilisation d'un seul processeur
  - Commutations plus légères que pour les processus

Plusieurs implémentations

▷ **Les Pthreads**

- ◇ Norme *Posix.1c*, c'est **la** référence !
- ◇ Fonctionnalités de base portables
- ◇ Influence sur les fonctionnalités habituelles (mono-tâche)
  $\rightarrow$ très dépendant de la plate-forme

▷ **Quelques exemples**

- ◇ *Linux* : *Pthread* (noyau), *Pth* (utilisateur)
- ◇ *IRIX* : *Pthread* (utilisateur), *sproc* (noyau)
- ◇ *Solaris* : *Pthread* (noyau)
- ◇ *SunOS* : *Pthread* (utilisateur), *LWP* (noyau)
- ◇ *Windows* : threads noyau
- ◇ *Java* : ???, très variable d'une plate-forme à l'autre
- ◇ Certaines sont hybrides (noyau/utilisateur)

Précautions

- **Exécutions indépendantes**
  - Nécessité d'une synchronisation explicite
- **Espace d'adressage commun**
  - Accès concurrents à des ressources communes
    - Mécanismes d'exclusion mutuelle
  - Écriture de traitements réentrants
    - Pas de données statiques ou globales
    - Arguments supplémentaires
  - Choix de primitives système réentrantes
    - Fonction ayant l'extension `_r` (*Posix.1c*)
    - Ex : `asctime()` et `asctime_r()`

Précautions

▷ **Exemple de fonction non réentrante**

◇ Contenu de `buffer` indéterminé si invocations simultanées

```
const char * writeHexa(int i)
{
static char buffer[0x10];
sprintf(buffer,"0x%.8x",i);
return(buffer);
}
```

▷ **Version réentrante de cette fonction**

◇ Chaque invocation utilise des données différentes (transmises)

```
char * writeHexa_r(int i,char * buffer)
{
sprintf(buffer,"0x%.8x",i);
return(buffer);
}
```

Conventions sur le nomage des *Pthreads*

- `#include <pthread.h>`
- **Les types :** `pthread[_`*`objet`*`]_t`
  - *`objet`* :
    - `attr`, `mutex` ou `cond`
    - *thread* si omis
- **Les fonctions :** `pthread[_`*`objet`*`]_`*`operation`*`[_np]`
  - *`objet`* : (voir les types)
  - *`operation`* : traitement concernant le type désigné
    - `init`, `destroy` ...
    - `create`, `exit`, `join` ...
    - `lock`, `unlock`, `signal`, `broadcast` ...
  - L'extension `_np` signale un traitement non portable (spécifique à l'implémentation courante)

Développer avec les *Pthreads*

- **Signalement des erreurs**
  - Pour la majorité des fonctions `pthread` :
    - `errno` n'est pas utilisé
    - Retour valant `0` → ok
    - Retour non nul → code d'erreur interprété comme `errno`
- **Compilation avec la macro `_REENTRANT`**
  - `$ cc -c -D_REENTRANT prog.c`
    - Influence sur les `.h` standards
    - Implémentation différente de certains services
- **Édition de liens avec la bibliothèque `pthread`**
  - `$ cc -o prog prog.o -lpthread`

Identification de l'activité

- **Le processus dans sa globalité**
  - `#include <sys/types.h>`
    `#include <unistd.h>`
    `pid_t getpid(void);`
  - 1 PID par thread pour les threads en mode noyau !
- **Le thread courant**
  - `pthread_t pthread_self(void);`
- **Le test d'égalité**
  - `pthread_t` est un type opaque → pas de comparaison directe !
  - `int pthread_equal(pthread_t t1,pthread_t t2);`
  - Résultat non nul si `t1` ≡ `t2`, `0` sinon

Création d'un thread

- **La fonction** `pthread_create()` (`man 3 pthread_create`)
  - `int pthread_create(pthread_t * id,`
    `pthread_attr_t * attr,`
    `void * (*fct)(void *),`
    `void * fctArg);`
  - Crée un thread exécutant `fct` avec l'argument `fctArg`
  - Stocke son identifiant dans `id`
  - Le thread a les propriétés décrites par `attr` (propriétés par défaut si pointeur nul)
  - Retour : `0` si ok, non nul si erreur (`EAGAIN`)
  - Causes d'erreur :
    - `PTHREAD_THREADS_MAX` atteint
    - Plus assez de ressources système

Terminaison d'un thread

- **Fin de sa fonction**
  - Résultat transmis par la valeur de retour
- **La fonction** `pthread_exit()` (`man 3 pthread_exit`)
  - `void pthread_exit(void * result);`
  - Termine le thread courant en retournant `result`
  - Résultat lisible par `pthread_join()` (voir plus loin)
  - Appelle les traitements de `pthread_cleanup_push()` (voir plus loin)
- **Fin du programme principal**
  - Fin de `main()` (ou `exit()`) ou recouvrement (par `exec()`) → destruction de tous les threads (il reste une activité)
  - `pthread_exit()` dans `main()` → attente de tous les threads

Attente un thread

- **La fonction** `pthread_join()` (`man 3 pthread_join`)
  - `int pthread_join(pthread_t th,void ** result);`
  - Attendre la terminaison de `th` et obtenir son résultat dans `result`
  - `*result` vaut `PTHREAD_CANCELED` si `th` a été annulé (voir plus loin)
  - Les ressources de `th` sont libérées
  - Un seul `pthread_join()` sur un thread donné
  - Le thread attendu ne doit pas être détaché (voir plus loin)
  - Un thread ne peut s'attendre lui même
  - Retour : `0` si ok, non nul si erreur

Créer et attendre un thread

```
#include <pthread.h>
#include <stdio.h>
void * task(void * data)
{
int * nb=(int *)data;
fprintf(stderr,"start 1\n");
for(int i=0;i<*nb;i++) { } fprintf(stderr,"end 1\n");
return((void *)0);
}
int main(void)
{
pthread_t th;
int nb=100000000;
if(pthread_create(&th,(pthread_attr_t *)0,task,&nb))
  { fprintf(stderr,"Pb create()\n"); return(1); }
fprintf(stderr,"start 2\n");
for(int i=nb/2;i<nb;i++) { } fprintf(stderr,"end 2\n");
void * result;
if(pthread_join(th,&result))
  { fprintf(stderr,"Pb join()\n"); return(1); }
fprintf(stderr,"quit\n");
return(0);
}
```

```
$ ./prog
start 2
start 1
end 2
end 1
quit
$
```

Effet de `sched_yield()`

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
void * task(void * data)
{
for(int i=0;i<1000000;i++)
  {
  fprintf(stderr,"%d",*((int *)data));
  // sched_yield();
  }
return((void *)0);
}
int main(void)
{
int n1=1,n2=2;
pthread_t t1,t2;
if(pthread_create(&t1,(pthread_attr_t *)0,task,&n1))
  { fprintf(stderr,"Pb thread 1\n"); return(1); }
if(pthread_create(&t2,(pthread_attr_t *)0,task,&n2))
  { fprintf(stderr,"Pb thread 2\n"); return(1); }
void * result; pthread_join(t1,&result); pthread_join(t2,&result);
return(0);
}
```

```
$ ./prog # sans sched_yield()
11111111111111111111111111111111111111111
1111 ... 11122222222222222222222222222222
222222222222222222222 ... 11111111111111
11111111111111111111111111111111 ... 11
1111122222222 ... 22222222222222222222222
2222
$
$ ./prog # avec sched_yield()
12121212121212121212121212121212121212121
212121212121212121212 ... 2121212121212
1212
$
```

Annulation d'un thread

- **La fonction** `pthread_cancel()` (`man 3 pthread_cancel`)
  - `int pthread_cancel(pthread_t th);`
  - Demande au thread `th` de se terminer
  - Pas forcément pris en compte immédiatement (voir plus loin)
  - Retour : `0` si ok, non nul si erreur
- **La fonction** `pthread_testcancel()` (`man 3 pthread_testcancel`)
  - `void pthread_testcancel(void);`
  - Le thread courant teste s'il a reçu une demande d'annulation
  - Le thread se termine à ce point si le test est positif
  - Le retour du thread vaut `PTHREAD_CANCELED`
  - Il existe d'autres points d'annulation (certains appels systèmes, certaines fonctions *pthread* ... dépend de l'implémentation)

Annulation d'un thread

▷ **La fonction** `pthread_setcancelstate()`
(`man 3 pthread_setcancelstate`)

- ◇ `int pthread_setcancelstate(int state,int * oldState);`
- ◇ Change l'état d'annulation du thread courant à `state`
- ◇ Indique l'ancien état si `oldState` est non nul
- ◇ `PTHREAD_CANCEL_ENABLE` : annulations autorisées (défaut)
- ◇ `PTHREAD_CANCEL_DISABLE` : annulations ignorées
- ◇ Retour : `0` si ok, non nul si erreur

Annulation d'un thread

▷ **La fonction** `pthread_setcanceltype()`
(`man 3 pthread_setcanceltype`)

- ◇ `int pthread_setcanceltype(int type,int * oldType);`
- ◇ Change le type d'annulation du thread courant à `type`
- ◇ Indique l'ancien type si `oldType` est non nul
- ◇ `PTHREAD_CANCEL_DEFERRED` :
  annulation prise en compte aux points d'annulation (défaut)
- ◇ `PTHREAD_CANCEL_ASYNCHRONOUS` :
  annulation prise en compte immédiatement (dangereux !)
- ◇ Retour : `0` si ok, non nul si erreur

Les attributs d'un thread

- **Création des attributs** (`man 3 pthread_attr_init`)
  - `int pthread_attr_init(pthread_attr_t * attr);`
  - Initialise à leur valeur par défaut les attributs désignés par `attr`
    - `detachstate` : `PTHREAD_CREATE_JOINABLE`
    - `schedpolicy` : `SCHED_OTHER`
    - `schedparam` : `0` (priorité)
    - `inheritsched` : `PTHREAD_EXPLICIT_SCHED`
    - `scope` : `PTHREAD_SCOPE_SYSTEM` (dépend de la plate-forme)
  - Modifications éventuelles avant l'appel à `pthread_create()`
  - Retour : `0` si ok, non nul sinon (cause ?)

Les attributs d'un thread

- **Destruction des attributs** (`man 3 pthread_attr_destroy`)
  - `int pthread_attr_destroy(pthread_attr_t * attr);`
  - Libère les attributs désignés par `attr`
  - Peut avoir lieu après `pthread_create()`
  - Les valeurs sont recopiées à la création du thread
  - Retour : `0` si ok, non nul sinon (cause ?)

Les attributs d'un thread

▷ Le détachement d'un thread

(`man 3 pthread_attr_setdetachstate`)

◇ `int pthread_attr_setdetachstate(`
`pthread_attr_t * attr,int state);`

◇ Interprétation de `state` :

- ○ `PTHREAD_CREATE_JOINABLE` :
  libération des ressources après `pthread_join()`
- ○ `PTHREAD_CREATE_DETACHED` :
  libération des ressources dès la terminaison du thread

◇ Retour : `0` si ok, non nul sinon

◇ Intervention sur un thread en cours

- ○ `int pthread_detach(pthread_t th);`
- ○ Le thread désigné passe dans l'état détaché
- ○ Retour : `0` si ok, non nul sinon (inconnu ou déjà détaché)

Les attributs d'un thread

▷ **Le mode d'ordonnancement d'un thread**

(man 3 pthread_attr_setschedpolicy)
(man 3 pthread_attr_setschedparam)
(man 3 pthread_attr_setinheritsched)

◇
```
int pthread_attr_setschedpolicy(
                    pthread_attr_t * attr,
                    int policy);
```

◇
```
int pthread_attr_setschedparams(
                    pthread_attr_t * attr,
                    const struct sched_param * params);
```

◇ Voir l'ordonnancement des processus

- `schedpolicy` : `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`
- `schedparams` : champ `sched_priority`

Les attributs d'un thread

- **Le mode d'ordonnancement d'un thread**
  - `int pthread_attr_setinheritsched(pthread_attr_t * attr, int inherit);`
  - Transmission des paramètres d'ordonnancement
    - `PTHREAD_INHERIT_SCHED` : reprendre ceux du thread parent
    - `PTHREAD_EXPLICIT_SCHED` : initialisation explicite (ou valeur par défaut)
  - Concerne uniquement `schedpolicy` et `schedparams`
  - Retour : `0` si ok, non nul sinon

Les attributs d'un thread

▷ **L'implémentation du thread**

(`man 3 pthread_attr_setscope`)

◇ `int pthread_attr_setscope(`
`pthread_attr_t * attr,int scope);`

◇ Interprétation de `scope` :

○ `PTHREAD_SCOPE_SYSTEM` :
thread noyau

○ `PTHREAD_SCOPE_PROCESS` :
thread utilisateur

◇ Choix réellement possible pour les implémentations hybrides

◇ Retour : `0` si ok, non nul sinon

Attributs/Annulation d'un thread

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

void * task(void * data)
{
// pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,(int *)0);
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,(int *)0);
// pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,(int *)0);
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,(int *)0);
fprintf(stderr,"task begin\n");
int * flag=(int *)data;
for(unsigned long i=0;i<200*1000*1000;i++) {}
(*flag)++;
pthread_testcancel();
(*flag)++;
fprintf(stderr,"task end\n");
return((void *)0);
}
```

Attributs/Annulation d'un thread

```
                                                $ ./prog # CANCEL DISABLE
                                                task begin
                                                task end
                                                quit (flag=2)
                                                $
                                                $ ./prog # CANCEL ENABLE & DEFERRED
                                                task begin
int main(void)                                  quit (flag=1)
{                                               $
pthread_attr_t attr;                            $ ./prog # CANCEL ENABLE & ASYNCHRONOUS
pthread_t th;                                   task begin
int flag=0;                                     quit (flag=0)
int pb=0;                                       $
if(!pb) pb|=pthread_attr_init(&attr);
if(!pb) pb|=pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
if(!pb) pb|=pthread_create(&th,&attr,task,&flag);
if(!pb) pb|=pthread_attr_destroy(&attr);
for(unsigned long i=0;i<100*1000*1000;i++) {}
pthread_cancel(th);
for(unsigned long j=0;j<200*1000*1000;j++) {}
fprintf(stderr,"quit (flag=%d)\n",flag);
return(pb);
}
```

La pile de nettoyage

- **Empiler un traitement** (`man 3 pthread_cleanup_push`)
  - `int pthread_cleanup_push(void (*fptr)(void *), void * arg);`
  - Macro contenant `{`
  - Empile une fonction et son argument
  - Appel automatique (en dépilant) à la fin du thread (fin de fonction, `pthread_exit()` ou `pthread_cancel()`)
  - Permet de fermer proprement ce qui a été initialisé
- **Dépiler un traitement** (`man 3 pthread_cleanup_pop`)
  - `int pthread_cleanup_pop(int execute);`
  - Macro contenant `}`
  - Dépile une fonction et l'appelle si `execute` est non nul
  - Doit être associé à un `pthread_cleanup_push()` (même bloc)

La pile de nettoyage

```
{
// ...
void * data=malloc(0x1000);
if(data)
  {
  pthread_cleanup_push(free,data);
  FILE * input=fopen("file.txt","r");
  if(input)
    {
    pthread_cleanup_push((void (*)(void *))fclose,input);
    doSomethingWithThat(input,data);
    pthread_cleanup_pop(1); // fclose(input)
    }
  pthread_cleanup_pop(1); // free(data)
  }
// ...
}
```

Réaction aux signaux

- **Émission explicite en interne**
  - `int pthread_kill(pthread_t th,int signum);`
  - Reçu par le thread désigné
- **Émission implicite par le système** (`SIGSEGV`, `SIGBUS` ...)
  - Mode noyau : reçu par le thread en cause
  - Mode utilisateur : reçu par le processus
- **Émission explicite depuis l'extérieur**
  - Mode noyau : reçu par le thread désigné par le PID
  - Mode utilisateur : reçu par un des threads (lequel ?)
- **Appel** `pthread` **depuis un gestionnaire** $\rightarrow$ **risque de blocage !**
- $\rightarrow$ **Éviter d'utiliser les signaux avec les threads !**

Réaction à `fork()`

- **Comportement normal d'un** `fork()`
  - Les données des différents threads existent dans l'enfant
  - Le processus enfant n'a qu'une activité : le thread qui l'a créé
  - *"Que deviennent ces données, les verrous ... ?"*
- **La fonction** `pthread_atfork()` (`man 3 pthread_atfork`)
  - ```
    int pthread_atfork(void (*prepare)(void),
                       void (*parent)(void),
                       void (*child)(void));
    ```
  - Enregistre des traitements à effectuer automatiquement :
    - avant le `fork()` (`prepare`)
    - après le `fork()` dans le parent (`parent`)
    - après le `fork()` dans l'enfant (`child`)
- **Éviter d'utiliser** `fork()` **avec les threads** (sauf pour `exec()`)
    ```

Synchronisation

- **Attendre la fin d'un thread**
  - `pthread_join()` (déjà vu)
- **Effectuer un traitement unique**
  - Plusieurs threads ont besoin qu'un traitement soit effectué
  - Celui-ci doit être effectué une seule fois
- **Sémaphores d'exclusion mutuelle** (verrous)
  - Limiter l'accès à une donnée
  - Un seul accès à la fois
- **Variables conditions**
  - Attendre qu'une condition soit vérifiée dans un autre thread
  - Blocage du traitement en attendant

Les traitements uniques

▷ **Synchronisation autour d'un** `pthread_once_t`
(`man 3 pthread_once`)

◇ `int pthread_once(pthread_once_t * control, void (*fptr)(void));`

◇ Avant usage, `control` doit être initialisé à `PTHREAD_ONCE_INIT`

◇ Premier accès à `control` → appel de la fonction `fptr`

◇ Accès ultérieurs à `control` → aucun effet

◇ Retour : toujours `0` !

Les traitements uniques

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

void initFunc(void)
{
fprintf(stderr,"initFunc() in %d\n",
               (int)pthread_self()); // ugly !
}

void * task(void * data)
{
fprintf(stderr,"begin task(%x) in %d\n",
               data,(int)pthread_self()); // ugly !
for(unsigned long i=0;i<50*1000*1000;i++) {}
pthread_once((pthread_once_t *)data,initFunc);
for(unsigned long j=0;j<50*1000*1000;j++) {}
fprintf(stderr,"end task(%x) in %d\n",
               data,(int)pthread_self()); // ugly !
return((void *)0);
}
```

```
$ ./prog
begin task(bffff744) in 1026
begin task(bffff748) in 2051
begin task(bffff744) in 3076
begin task(bffff748) in 4101
begin task(bffff744) in 5126
begin task(bffff748) in 6151
initFunc() in 1026
initFunc() in 2051
end task(bffff744) in 1026
end task(bffff744) in 3076
end task(bffff744) in 5126
end task(bffff748) in 4101
end task(bffff748) in 2051
end task(bffff748) in 6151
$
```

Les traitements uniques

```
int main(void)
{
pthread_once_t ctrl1=PTHREAD_ONCE_INIT;
pthread_once_t ctrl2=PTHREAD_ONCE_INIT;
pthread_t th[6];
for(int i=0;i<6;i++)
  {
  if(pthread_create(&th[i],(pthread_attr_t *)0,task,i%2 ? &ctrl1 : &ctrl2))
    { fprintf(stderr,"Pb pthread_create()\n"); return(1); }
  }
for(int j=0;j<6;j++)
  {
  void * result;
  if(pthread_join(th[j],&result))
    { fprintf(stderr,"Pb pthread_join()\n"); return(1); }
  }
return(0);
}
```

Les sémaphores d'exclusion mutuelle

- **Création d'un verrou** (`man 3 pthread_mutex_init`)
  - Type : `pthread_mutex_t`
  - Initialisation statique
    - `pthread_mutex_t myMutex=PTHREAD_MUTEX_INITIALIZER;`
  - Initialisation dynamique
    - `int pthread_mutex_init(pthread_mutex_t * mtx, pthread_mutexattr_t * attr);`
    - Généralement `attr` est nul (initialisation par défaut)
- **Destruction d'un verrou** (`man 3 pthread_mutex_destroy`)
  - `int pthread_mutex_destroy(pthread_mutex_t * mtx);`
  - Le verrou n'est plus utilisable
  - Retour : `0` si ok, non nul si erreur
  - Erreur si le verrou est monopolisé → erreur `EBUSY`

Les sémaphores d'exclusion mutuelle

- **Opération de verrouillage** (`man 3 pthread_mutex_lock`)
  - `int pthread_mutex_lock(pthread_mutex_t * mtx);`
  - Si le verrou est libre
    - il est monopolisé par le thread courant
    - le thread courant poursuit son traitement
  - Si le verrou n'est pas libre
    - le thread courant est bloqué jusqu'à la libération du verrou
  - Retour : `0` si ok, non nul si erreur

Les sémaphores d'exclusion mutuelle

- **Opération de déverrouillage** (`man 3 pthread_mutex_unlock`)
  - `int pthread_mutex_unlock(pthread_mutex_t * mtx);`
  - Libère le verrou
  - Si des threads sont bloqués en attente sur ce verrou
    - l'un d'eux est débloqué et monopolise le verrou
  - Retour : `0` si ok, non nul si erreur
- **Tentative de verrouillage** (`man 3 pthread_mutex_trylock`)
  - `int pthread_mutex_trylock(pthread_mutex_t * mtx);`
  - Si le verrou est libre
    - il est monopolisé par cet appel qui retourne `0`
  - Si le verrou n'est pas libre
    - cet appel retourne immédiatement un résultat non nul
    - il ne faut pas utiliser les données protégées par le verrou

Les sémaphores d'exclusion mutuelle

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

void * task(void * data)
{
pthread_mutex_t * mtx=(pthread_mutex_t *)data;
fprintf(stderr,"begin task() in %d\n",
               (int)pthread_self()); // ugly cast !
for(int n=0;n<2;n++)
  {
  pthread_mutex_lock(mtx); // begin critical section
  fprintf(stderr,"  thread %d -->\n",
               (int)pthread_self()); // ugly cast !
  for(unsigned long i=0;i<10*1000*1000;i++) {}
  fprintf(stderr,"  thread %d <--\n",
               (int)pthread_self()); // ugly cast !
  pthread_mutex_unlock(mtx); // end critical section
  }
fprintf(stderr,"end task() in %d\n",
               (int)pthread_self()); // ugly cast !
return((void *)0);
}
```

```
$ ./prog
begin task() in 1026
  thread 1026 -->
begin task() in 2051
begin task() in 3076
  thread 1026 <--
  thread 2051 -->
  thread 2051 <--
  thread 3076 -->
  thread 3076 <--
  thread 1026 -->
  thread 1026 <--
  thread 3076 -->
end task() in 1026
  thread 3076 <--
  thread 2051 -->
  thread 2051 <--
end task() in 2051
end task() in 3076
$
```

Les sémaphores d'exclusion mutuelle

```
int main(void)
{
pthread_mutex_t mtx=PTHREAD_MUTEX_INITIALIZER;
pthread_t th[3];
int i;
for(i=0;i<3;i++)
  {
  if(pthread_create(&th[i],(pthread_attr_t *)0,task,&mtx))
    { fprintf(stderr,"Pb create\n"); return(1); }
  }
void * result;
for(i=0;i<3;i++)
  {
  if(pthread_join(th[i],&result))
    { fprintf(stderr,"Pb join\n"); return(1); }
  }
if(pthread_mutex_destroy(&mtx))
  { fprintf(stderr,"Pb mutex destroy\n"); return(1); }
return(0);
}
```

Les variables conditions

- **Création d'une condition** (`man 3 pthread_cond_init`)
  - Type : `pthread_cond_t` (doit être associé à un `mutex`)
  - Initialisation statique
    - `pthread_cond_t myCond=PTHREAD_COND_INITIALIZER;`
  - Initialisation dynamique
    - `int pthread_cond_init(pthread_cond_t * cond, pthread_condattr_t * attr);`
    - Généralement `attr` est nul (initialisation par défaut)
- **Destruction d'une condition** (`man 3 pthread_cond_destroy`)
  - `int pthread_cond_destroy(pthread_cond_t * cond);`
  - La condition n'est plus utilisable
  - Retour : `0` si ok, non nul si erreur
  - Erreur si la condition est utilisée → erreur `EBUSY`

Les variables conditions

▷ **Attente d'une condition** (`man 3 pthread_cond_wait`)

◇ `int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mtx);`

◇ Bloque le thread courant
→ débloqué quand une modification est signalée sur `cond`
→ évite l'attente active

◇ `cond` n'a aucune valeur logique ! (sert à la synchronisation)

◇ `mtx` doit être monopolisé avant et libéré après

◇ Interruptible par les signaux → relance

◇ Démarche usuelle :

```
pthread_mutex_lock(&mtx);
while(!conditionIsSatisfied())
  pthread_cond_wait(&cond,&mtx);
pthread_mutex_unlock(&mtx);
```

Les variables conditions

▷ **Attente temporisée d'une condition**

(`man 3 pthread_cond_timedwait`)

◇ `int pthread_cond_timedwait(pthread_cond_t * cond, pthread_mutex_t * mtx, const struct timespec * date);`

◇ Même principe que `pthread_cond_wait()`

◇ Renvoie `ETIMEDOUT` si `date` est dépassée

◇ `date` est une limite, pas un délai !

- Prendre la date courante (`time()`, `gettimeofday()`)
- Ajouter un délai (structure `timespec` de `nanosleep()`)
- Basé sur le temps universel (pas de fuseau horaire)
- Voir le cours sur la mesure du temps

Les variables conditions

▷ **Signaler une condition** (`man 3 pthread_cond_broadcast`)

◇ `int pthread_cond_broadcast(pthread_cond_t * cond);`

◇ Débloque tous les threads attendant la condition `cond`

◇ Le verrou associé à `cond` doit être monopolisé avant et libéré après

◇ Démarche usuelle :

```
pthread_mutex_lock(&mtx);
/* make this condition become true */
pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mtx);
```

◇ `int pthread_cond_signal(pthread_cond_t * cond);`

◇ Ne débloque qu'un thread parmi ceux qui attendent `cond`

◇ Un peu plus efficace que `pthread_cond_broadcast`

◇ Bien moins général (incohérence si plusieurs threads en attente !)

Les variables conditions

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

pthread_mutex_t mtx;
pthread_cond_t cond;
unsigned long x,y;

void * task(void *)
{
pthread_mutex_lock(&mtx);
while(x<=y)
  {
  fprintf(stderr,"until now %lu <= %lu\n",x,y);
  pthread_cond_wait(&cond,&mtx);
  }
pthread_mutex_unlock(&mtx);
fprintf(stderr,"and then %lu > %lu\n",x,y);
return((void *)0);
}
```

```
$ ./prog
..until now 115245 <= 10000000
.........................................
.........................................
.....and then 10000001 > 10000000
.........................................
.........................................
.....$
$
```

Les variables conditions

```
int main(void)
{
pthread_t th;
void * result;
unsigned long t;
x=0; y=10000000;
pthread_mutex_init(&mtx,(pthread_mutexattr_t *)0);
pthread_cond_init(&cond,(pthread_condattr_t *)0);
if(pthread_create(&th,(pthread_attr_t *)0,task,(void *)0))
  { fprintf(stderr,"Pb create\n"); return(1); }
for(t=0;t<20000000;t++)
  {
  if(!(t%100000)) fputc('.',stderr);
  pthread_mutex_lock(&mtx);
  if((x=t)>y) pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&mtx);
  }
if(pthread_join(th,&result))    { fprintf(stderr,"Pb join\n"); return(1); }
if(pthread_cond_destroy(&cond)) { fprintf(stderr,"Pb cond destroy\n"); return(1); }
if(pthread_mutex_destroy(&mtx)) { fprintf(stderr,"Pb mtx destroy\n"); return(1); }
return(0);
}
```

Les données privées

- **Principe**
  - Créer une clef unique accessible par tous les threads
  - Chaque thread associe ses propres données à cette clef
  - Ces données sont tout à fait indépendantes d'un thread à un autre
  - Type de la clef : `pthread_key_t`
- **Création de la clef** (`man 3 pthread_key_create`)
  - ```
    int pthread_key_create(pthread_key_t * key,
                           void (*destrFnct)(void *));
    ```
  - Initialise la clef désignée par `key`
  - Si `destrFnct` est non nul → fonction de destruction des données
    - À la terminaison ou à l'annulation de chaque thread
  - Retour : `0` si ok, non nul si erreur
  - Erreur si nombre de clef maxi atteint (`PTHREAD_KEYS_MAX`)
    ```

Les données privées

- **Destruction de la clef** (`man 3 pthread_key_delete`)
  - `int pthread_key_delete(pthread_key_t key);`
  - La clef devient inutilisable (n'appelle pas la fonction de destruction)
  - Retour : `0` si ok, non nul si erreur
- **Écriture d'une donnée** (`man 3 pthread_key_setspecific`)
  - `int pthread_key_setspecific(pthread_key_t key, const void * data);`
  - Associe la donnée `data` à la clef `key` pour le thread courant
  - Retour : `0` si ok, non nul si erreur
- **Lecture d'une donnée** (`man 3 pthread_key_getspecific`)
  - `void * pthread_key_getspecific(pthread_key_t key);`
  - Retourne la donnée associée à `key` et au thread courant
  - Retourne un pointeur nul en cas d'erreur

Les données privées

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

pthread_key_t key;

void destroyFunc(void * data)
{
fprintf(stderr,"destroying [%s]\n",
               (const char *)data);
}

void * task(void * data)
{
if(pthread_setspecific(key,((int)data)%2 ? "data is odd" : "data is even"))
  { fprintf(stderr,"Pb set specific\n"); return((void *)0); }
for(int i=0;i<100*1000*1000;i++);
fprintf(stderr,"stored in thread %d : %s\n",
               (int)pthread_self(), // ugly cast !
               (const char *)pthread_getspecific(key));
return((void *)0);
}
```

```
$ ./prog
stored in thread 1026 : data is odd
destroying [data is odd]
stored in thread 2051 : data is even
destroying [data is even]
stored in main thread : first thing stored
$
```

Les données privées

```
int main(void)
{
if(pthread_key_create(&key,destroyFunc))
  { fprintf(stderr,"Pb key create\n"); return(1); }
if(pthread_setspecific(key,"first thing stored"))
  { fprintf(stderr,"Pb set specific\n"); return(1); }
pthread_t th1,th2;
if(pthread_create(&th1,(pthread_attr_t *)0,task,(void *)1))
  { fprintf(stderr,"Pb create\n"); return(1); }
if(pthread_create(&th2,(pthread_attr_t *)0,task,(void *)2))
  { fprintf(stderr,"Pb create\n"); return(1); }
void * result;
if(pthread_join(th1,&result))
  { fprintf(stderr,"Pb join\n"); return(1); }
if(pthread_join(th2,&result))
  { fprintf(stderr,"Pb join\n"); return(1); }
fprintf(stderr,"stored in main thread : %s\n",
               (const char *)pthread_getspecific(key));
if(pthread_key_delete(key))
  { fprintf(stderr,"Pb key destroy\n"); return(1); }
return(0);
}
```