Les threads POSIX

Les threads d'un processus

Dans sa forme basique, un thread est un processus, dit léger, qui s'exécute à l'intérieur d'un processus UNIX ou Linux. Rappelons qu'un processus Linux est créé par le noyau à la demande d'un processus déjà existant, via la primitive **fork(2).** C'est communément le shell qui gère votre terminal qui exécute un **fork(2)** et le shell fils résultant qui demande l'exécution, via la primitive **exec(2)** d'un programme au format « a.out ».

Un thread est une fonction écrite en C, qu'une primitive **pthread_create** transforme en une entité de programme autonome qui possède son propre espace mémoire, les variables locales de la fonction. Les variables globales du processus sont partagées par tous les threads du processus. La fonction main() joue le rôle du thread principal exécuté dès la création du processus et qui prend en charge la création des autres threads. Le programme qui suit est un exemple élémentaire de processus Linux qui crée un thread.

```
$ cat unpremierthread.c
/*   programme unpremierthread.c
     auteur Gilles GOUBET, écrit le 30 mars 2010
     Ce programme active un thread
     qui affiche un message
*/
#include <time.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
/*
  La fonction leThread contient le code du thread
*/
void * leThread(void * arg)
{
  printf("========================\n");
  printf("Execution du thread %s\n",(char *)arg);
  printf("========================\n");
  pthread_exit(NULL);
}
main()
{
pthread_t th1;  /*  Un descripteur de thread */
if (pthread_create (&th1, NULL, leThread , "Premier THREAD"))
  { perror("Creation thread probleme\n");
    exit(2);
  }
if (!pthread_join(th1,NULL))
  {
   printf("Le thread est termine\n");
   exit(0);
  }
```

```
}
```

Pour réaliser correctement l'édition de liens, il faut mentionner la bibliothèque des threads via l'option -lpthread.

```
$ cc unpremierthread.c -lpthread
$ a.out
==========================
Execution du thread Premier THREAD
==========================
Le thread est termine
```


Le fichier qu'il faut inclure pour manipuler des threads.

#include <pthread.h>

La déclaration (prototype) d'un thread

Le thread est implémenté par une fonction à laquelle on passe l'adresse d'une zone qui, si elle est différente de NULL, contient des paramètres passés au thread (*cf. pthread_create*)

void * leThread(void * arg)

pthread_create (&th1, NULL, leThread , "Premier THREAD")

Le thread peut aussi renvoyer des résultats. L'adresse renvoyée par la fonction pthread_exit peut être récupérée par la fonction **pthread_join**.

pthread_exit(NULL);

pthread_join(th1,NULL)

La création d'un thread

La création d'un thread se fait avec la fonction **pthread_create** dont le principal argument est l'adresse de la fonction qui implémente le thread.

void * leThread(void * arg)
pthread_t th1; /* Un descripteur de thread */
pthread_create (&th1, NULL, leThread , "Premier THREAD")

L'attente de la fin d'un thread

Dans le thread qui a exécuté **pthread_create** on peut attendre la fin du thread fils grâce à la fonction **pthread_join**.

```
if (!pthread_join(th1,NULL))
  {
   printf("Le thread est termine\n");
```

Le partage des données globales

```
$ cat deuxthreadpartagedonnee.c
```

```
/*   programme deuxthreadsenpartagedonnee.c
     auteur Gilles GOUBET, écrit en mars 2010
     Ce programme crée un processus qui execute deux threads
*/
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <pthread.h>
/* Les deux threads */
pthread_t th1,th2;
/* La donnee partagee sans exclusion mutuelle */
int donnee_critique;
/* La structure pour attendre n nano secondes */
struct timespec attente = {0,} ;
/*
  La fonction laThread contient le code du module
*/
void * laThread(void * arg)
{ int i;
  char *ch;
  struct sched_param p;
  long nbNano;
  ch=(char *)arg;
  for(i=1;i<=10;i++)
   {
    if ( ch[0] == '1' )
            {
             donnee_critique++;
            }
    else
            {
             donnee_critique--;
            }
    printf("Donnee critique apres modif du thread %c :
%5d\n",ch[0],donnee_critique);
    nbNano=rand();
    attente.tv_nsec=nbNano;
    nanosleep(&attente,NULL);
   }
  return (void *)0;
}
main()
{
  srand(time(NULL));
  /* creation du premier thread */
  if ( pthread_create(&th1,NULL,laThread,"1")
   {
    printf("erreur de creation du premier thread\n");
    return 1;
   }
```

```
  /* creation du deuxième thread */
  if ( pthread_create(&th2,NULL,laThread,"2"))
   {
    printf("erreur de création du second thread\n");
    return 1;
   }
  if (!pthread_join(th1,NULL))
  {
   printf("Le thread est termine\n");
  }
  if (!pthread_join(th2,NULL))
  {
   printf("Le thread est termine\n");
  }
}
```

Les résultats

```
$ a.out
Donnee critique apres modif du thread 1 :      1
Donnee critique apres modif du thread 1 :      1
Donnee critique apres modif du thread 2 :      0
Donnee critique apres modif du thread 2 :      0
Donnee critique apres modif du thread 2 :     -1
Donnee critique apres modif du thread 1 :      0
Donnee critique apres modif du thread 1 :      1
Donnee critique apres modif du thread 1 :      2
Donnee critique apres modif du thread 1 :      3
...
Le thread est termine
Le thread est termine
```

Ce programme requiert peu de commentaires. Il crée deux threads, instances d'une même fonction. Les deux threads qui s'exécutent bien entendu en parallèle, accèdent concurremment à la même variable globale « donnee_critique » que l'une incrémente et l'autre décrémente. Nous ne nous intéressons pas pour l'instant à la fonction **nanosleep** qui permet de suspendre l'exécution d'un thread de façon beaucoup plus fine que la fonction **sleep**. L'accès à la donnée n'est pas protégée. Nous traiterons plus loin les primitives de synchronisation qui permettent de mettre en œuvre l'exclusion mutuelle.

Les attributs d'un thread

Un thread possède un ensemble d'attributs que plusieurs fonctions de l'API permettent de visualiser ou de modifier. Les attributs sont les suivants :

detachstate

Cet attribut peut avoir la valeur PTHREAD_CREATE_JOINABLE si le thread est synchronisé sur la fin de son exécution avec celui qui l'a créé ou PTHREAD_CREATE_DETACHED si le thread est détaché.

schedpolicy

Cet attribut peut avoir la valeur SCHED_OTHER, SCHED_RR ou SCHED_FIFO. La valeur par défaut, SCHED_OTHER, désigne un thread normal, temps partagé. La valeur SCHED_RR désigne un thread temps réel, l'ordonnancement est alors selon le mode "round robbin". La valeur SCHED_FIFO désigne une thread temps réel selon le mode FIFO.

schedparam

Cet attribut contient les paramètres d'ordonnancement, principalement la priorité. Sa valeur par défaut est 0. Cet attribut n'est significatif que si l'attribut schedpolicy est SCHED_RR ou SCHED_FIFO, pour le temps réel.

inheritsched

Cet attribut dit si les attributs schedpolicy et schedparam sont déterminées par le thread qui vient d'être créé ou hérités de son parent.

scope

Cet attribut précise si le thread, via sa priorité, est en compétition avec les autres processus du système (PTHREAD_SCOPE_SYSTEM) ou si elle est en compétition avec les autres threads du processus auquel elle appartient (PTHREAD_SCOPE_PROCESS).

Les primitives de gestion des attributs

La structure pthread_attr_t définit les attributs du thread.

```
struct
{
        int detachstate;
        int schedpolicy;
        struct sched_param schedparam;
        int inheritsched;
        int scope;
 } pthread_attr_t;
```


Les primitives qui manipulent cette structure sont **pthread_attr_destroy** qui détruit les attributs et

pthread_attr_init qui les initialise.

*int pthread_attr_destroy(pthread_attr_t *attr);*
*int pthread_attr_init(pthread_attr_t *attr);*

Les attributs peuvent aussi être écrits ou lus individuellement par les primitives appropriées. La forme des fonctions est toujours la même et la forme générique qui suit donne les arguments de la grande majorité des fonctions :

*int pthread_attr_set<attributduthread> (pthread_attr_t *attr, int <étatduthread>);*

*int pthread_attr_get<attributduthread> (const pthread_attr_t *attr, int *<étatduthread>);*

Vous aurez ainsi les primitives **pthread_attr_getschedpolicy** et **pthread_attr_setschedpolicy** pour l'attribut schedpolicy.

Primitives spécifiques pour l'ordonnancement des threads.

Les primitives **pthread_getschedparam** et **pthread_setschedparam** permettent de lire ou de positionner les paramètres d'ordonnancement des threads. Pour les threads temps réel de type SCHED_FIFO et SCHED_RR, le seul paramètre significatif est celui qui permet de fixer la priorité du thread, l'attribut sched_priority.

*int pthread_getschedparam(pthread_t thread, int *restrict policy,*
*struct sched_param *restrict param);*

int pthread_setschedparam(pthread_t thread, int policy,
*const struct sched_param *param);*

```
void * laThread(void * arg)
{ int i;
  char *ch;
  struct sched_param p;
  long nbNano;
  ch=(char *)arg;
  p . sched_priority = 1;
  /* Le thread positionne les parametres de son propre
  ordonnancement en s'identifiant pthread_self()
  Le type de thread est de type temps reel FIFO avec SCHED_FIFO
  Les parametres d'ordonnancement sont definis dans la
  structure p
  */
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
```


Primitives de gestion de la vie d'un thread

La primitive de création d'un thread

La primitive **pthread_create** crée un thread. Sa déclaration est la suivante :

*int pthread_create(pthread_t * thread,*
*pthread_attr_t * attr,*
*void * (*start_routine)(void *),*
*void * arg);*

La primitive **pthread_create** crée un nouveau thread, l'argument attr définit les attributs du thread (ou NULL), l'argument start_routine définit la fonction C qui correspond au code du thread. Le dernier argument définit le premier argument de la fonction start_routine. Un thread se termine en exécutant la fonction pthread_exit ou, implicitement, à la fin de la fonction.

Primitive de terminaison d'un thread.

*void pthread_exit(void *retval);*

La primitive **pthread_exit** met fin au thread. Tous les handlers qui ont été définis pour le thread par la primitive **pthread_cleanup_push** sont exécutés dans l'ordre inverse de leur insertion. Le code de retour retval peut être consulté par les autres thread grâce à la primitive **pthread_join**.

Primitive de synchronisation d'un thread appelante et appelée

*int pthread_join(pthread_t th, void **thread_return);*

La primitive **pthread_join** permet au thread appelant d'attendre la fin, par **pthread_exit** ou l'arrêt (cancel), du thread th. Le paramètre thread_return vaut NULL ou alors :

- l'argument renvoyé par th lors du thread_exit.

- PTHREAD_CANCELED si le thread s'est arrêtée sur un cancel.

Le thread th doit être joignable, c'est à dire ne pas avoir été détachée (pthread_detach).

Primitive de détachement d'une thread

int pthread_detach(pthread_t th);

La primitive **pthread_detach** place le thread dans un état détaché. un autre thread ne peut se synchroniser avec lui par **pthread_join**. Les ressources du thread sont libérées dès sa terminaison. L'état détaché peut avoir été positionné dès la création (**pthread_create**), par l'attribut detachstate.

Synthèse

Le thread est attaché à son père.

```
$ cat unpremierthread.c
#include <pthread.h>
void * leThread(void * arg)
{
...
  pthread_exit(NULL);
}
main()
{
pthread_t th1;  /*  Un descripteur de thread */
if (pthread_create (&th1, NULL, leThread , "Premier THREAD"))
  { perror("Creation thread probleme\n");
    exit(2);
  }
if (!pthread_join(th1,NULL))
  {
   printf("Le thread est termine\n");
   exit(0);
  }
```

```
}
```

Primitive de gestion de la priorité

La fonction pthread_setschedparam

La structure sched_param permet de préciser les paramètres d'ordonnancement, dont la priorité.

```
struct sched_param p;
p . sched_priority = 5;
```

La primitive **pthread_setschedparam** permet de positionner les paramètres définis dans la struct sched_param.


```
pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
```

La fonction **sched_get_priority_min** permet de connaître la priorité minimum pour une catégorie de thread, ici SCHED_FIFO.

```
printf("Priorite MIN de la tache %5d\n",sched_get_priority_min(SCHED_FIFO));
```

La fonction **sched_get_priority_max** permet de connaître la priorité minimum pour une catégorie de thread, ici SCHED_FIFO.

```
printf("Priorite MAX de la tache %5d\n",sched_get_priority_max(SCHED_FIFO));
```



```
$ cat gestionpriorite.c
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
/*
  La fonction leThread contient le code du module
*/
void * leThread(void * arg)
{
  pthread_attr_t attr;
  struct sched_param p;
  int politique;
  p . sched_priority = 5;
   /* Le thread positionne les paramètres de son propre
ordonnancement
 en s'identifiant avec la primitive pthread_self(). Le type de
thread est de type temps reel FIFO avec SCHED_FIFO
. Les paramètres d'ordonnancement sont definis dans la
structure p
   */
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
  printf("Priorite MIN de la tache
         %5d\n",sched_get_priority_min(SCHED_FIFO));
  printf("Priorite MAX de la tache
          %5d\n",sched_get_priority_max(SCHED_FIFO));
  pthread_getschedparam (pthread_self(), &politique, &p);
```

```
  printf("Priorite de la thread %5d \n",p.sched_priority);
  p . sched_priority = 50;
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
  pthread_getschedparam (pthread_self(), &politique, &p);
  printf("Priorite de la thread %5d \n",p.sched_priority);
}
main() {
/* exécuter le programme en tant que root */
        pthread_t th1;  /*  Un descripteur de thread */
       pthread_create (&th1, NULL, leThread , "Premiere
THREAD");
       if (!pthread_join(th1,NULL))
     {
       printf("La thread est terminee\n");
       exit(0);
     }
}
```

La fonction pthread_setschedprio

L'exemple qui suit illustre la modification de la priorité d'un thread grâce à la fonction **pthread_setschedprio**. Il offre aussi l'intérêt de montrer le positionnement des attributs d'un thread, préalablement à sa création, pour que le thread ne soit pas créé avec, par défaut, les mêmes attributs que son créateur.


```
$ cat gestion2priorite.c
/*   programme gestion2priorite.c
     auteur Gilles GOUBET, écrit en mars 2010
     Ce programme active un module qui crée et exécute un
     thread qui modifie sa priorité
*/
#include <time.h>
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>
/*
  La fonction leThread contient le code du module
*/
struct sched_param p;
void * leThread(void * arg)
{
   int politique;
   pthread_attr_t attr;
   printf("Priorite MIN des thread FIFO
           %5d\n",sched_get_priority_min(SCHED_FIFO));

   printf("Priorite MAX des thread FIFO
           %5d\n",sched_get_priority_max(SCHED_FIFO));
   if ( pthread_setschedprio(pthread_self(), 5))
    {  switch (errno)
       {
```

```
        case EINVAL : printf("EINVAL\n");
                      break;
        case EPERM  : printf("EPERM\n");
                      break;
        case ESRCH  : printf("ESRCH\n");
       }
      perror("Erreur : fixation priorite du thread");
    }
  pthread_getschedparam (pthread_self(), &politique, &p);
  printf("Priorite de la thread %5d \n",p.sched_priority);
   switch (politique)
     {
       case SCHED_OTHER : printf("OTHER\n");
                          break;
       case SCHED_FIFO  : printf("FIFO\n");
                          break;
       case SCHED_RR    : printf("RR\n");
                          break;
            default     : printf("Politique inconnue\n");
     }
}
main() {
/* executer le programme en tant que root */
   pthread_t th1;  /*  Un descripteur de thread */
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   if(pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED
)
     {
       perror("Erreur set attribut heritage\n");
       exit(errno);
     }
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO))
     {
       perror("Erreur set attribut politique\n");
       exit(errno);
     }
   if (pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM))
     {
       perror("Erreur set attribut scope\n");
       exit(errno);
     }
   if
   (pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE))
     {
       perror("Erreur set attribut joignable\n");
       exit(errno);
     }
   p.sched_priority=1;
   if (pthread_attr_setschedparam(&attr,&p ))
```

```
     {
       perror("Erreur set attribut schedparam\n");
       exit(errno);
     }
   if (pthread_create (&th1, &attr , leThread , "Premiere
THREAD") == EINVAL)
     { perror("Creation thread");
       printf("Erreur creation THREAD %5d\n",errno);
       exit(errno);
     }
   if (!pthread_join(th1,NULL))
     {
       printf("Le thread est termine\n");
       exit(0);
     }
}
```

Primitives de synchronisation

L'exclusion mutuelle

Le noyau Linux propose des tableaux de sémaphores, très généraux, qui permettent de gérer tous les cas classiques de synchronisation.

Dans le cas particulier des threads, on trouve des outils spécialisés, adaptés à chaque problème de synchronisation. C'est le cas pour l'exclusion mutuelle qui consiste, rappelons-le, à garantir qu'une seule tâche, ici un thread, accède à la fois à une ressource pendant l'exécution d'une séquence d'instructions, la section critique.

```
$ cat threadmutex.c
/*   programme threadmutex.c
     auteur Gilles GOUBET, écrit en mars 2010
     Ce programme crée un processus qui exécute deux threads
     qui s'excluent mutuellement avec un sémaphore
     d'exclusion mutuelle
     Appel de pthread_mutex_init
              pthread_mutex_lock
              pthread_mutex_unlock
*/
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <pthread.h>
/* Les deux threads */
pthread_t th1,th2;
/* Le sémaphore */
pthread_mutex_t  semaphore_mutex = PTHREAD_MUTEX_INITIALIZER;
/* La donnée critique */
int donnee_critique;
/* La structure pour attendre n nano secondes */
struct timespec attente = {0,} ;
```

```
/*
  La fonction leThread contient le code du thread
*/
void * leThread(void * arg)
{ int i;
  char *ch;
  struct sched_param p;
  long nbNano;
  ch=(char *)arg;
  p . sched_priority = 1;
  /* Le thread positionne les parametres de son propre
     ordonnancement en s'identifiant pthread_self()
     Le type de thread est de type temps reel FIFO avec
     SCHED_FIFO. Les paramètres d'ordonnancement sont
     définis dans la structure p
  */
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
  for(i=1;i<=50;i++)
   {
    if ( pthread_mutex_lock( &semaphore_mutex) )
       { printf("Erreur de verrouillage\n");
         return (void *)1;
       }
    printf("debut sect. crit. thread nu %s",arg);
    if ( ch[0] == '1' )
            {
             donnee_critique++;
            }
    else
            {
             donnee_critique--;
            }
    printf("  Don. crit. %d",donnee_critique);
    nbNano=rand();
    attente.tv_nsec=nbNano*100;
    nanosleep(&attente,NULL);
    printf(" Sortie sect crit. thread nu %s\n",arg);
    if ( pthread_mutex_unlock( &semaphore_mutex) )
       { printf("Erreur de déverrouillage\n");
         return (void *)1;
        }
    nbNano=rand();
    attente.tv_nsec=nbNano;
    nanosleep(&attente,NULL);
   }

  return (void *)0;
}
main()
{
```

```
  srand(time(NULL));
  if ( pthread_mutex_init(&semaphore_mutex,NULL))
   {
    printf("erreur d'initialisation du sémaphore \n");
    return 3;
  }
  /* creation du premier thread */
  if ( pthread_create(&th1,NULL,leThread,"1"))
   {
    printf("erreur de création du premier thread\n");
    return 1;
   }
  /* creation du deuxième thread */
  if ( pthread_create(&th2,NULL,leThread,"2"))
   {
    printf("erreur de création de la seconde thread\n");
    return 1;
   }
  if (!pthread_join(th1,NULL))
  {
   printf("Le thread est terminé\n");
  }
  if (!pthread_join(th2,NULL))
  {
   printf("Le thread est terminé\n");
  }
  if ( pthread_mutex_destroy(&semaphore_mutex))
     {
      printf("erreur de destruction du sémaphore \n");
     }
}
```

La déclaration du sémaphore d'exclusion mutuelle

La constante PTHREAD_MUTEX_INITIALIZER permet d'initialiser un sémaphore statique de la même manière que la fonction **pthread_mutex_init** avec NULL comme second argument. L'appel à cette fonction est donc superflue dans notre programme.

C'est souvent le thread main qui initalise et détruit le sémaphore et les threads appicatives qui le verrouille et le déverrouille.

pthread_mutex_t semaphore_mutex = PTHREAD_MUTEX_INITIALIZER;

L'initalisation du sémaphore

*int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);*

Le sémaphore passé en premier argument est initialisé avec attr[1]. Cet argument est NULL quand on

1 Nous ne traitons pas les attributs d'un MUTEX, nous vous indiquons le nom de quelques primitives dont l'étude vous permettra de comprendre, le cas échéant, l'intérêt de attr.
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

utilise les valeurs par défaut. Il permet sinon de préciser des paramètres du MUTEX.

pthread_mutex_init(&semaphore_mutex,NULL)

La destruction du sémaphores

*int pthread_mutex_destroy(pthread_mutex_t *mutex);*

Le sémaphore que l'on passe en argument est détruit.

pthread_mutex_destroy(&semaphore_mutex)

Le verrouillage du sémaphore

*int pthread_mutex_lock(pthread_mutex_t *mutex);*

Le sémaphore passé en argument est verrouillé et le thread entre en section critique. Il faut noter que si le sémaphore est déjà verrouillé par un autre thread, le thread demandeur est bloqué jusqu'à ce que le sémaphore soit déverrouillé..

pthread_mutex_lock(&semaphore_mutex)

Il existe une version non bloquante de tentative de verrouillage :

*int pthread_mutex_trylock(pthread_mutex_t *mutex);*

Le déverrouillage du sémaphore

*int pthread_mutex_unlock(pthread_mutex_t *mutex);*

Le sémaphore est déverrouillé. Si un thread est alors bloqué sur une tentative de verrouillage, le sémaphore reste verrouillé et le thread préalablement bloqué entre en section critique.

pthread_mutex_unlock(&semaphore_mutex)

Le schéma producteur-consommateur

Le schéma producteur consommateur concerne deux threads. Le producteur signal une production au consommateur et le consommateur, après avoir consommé, signale au producteur qu'il peut à nouveau produire. Le schéma garantit que le producteur est bloqué tant que le consommateur n'a pas consommé et, inversement, que le consommateur est bloqué tant que le producteur n'a pas produit.

Le nombre de productions autorisées avant que le blocage intervienne est défini par la valeur initiale du sémaphore.

Le sémaphore avec compteur implémente en fait le sémaphore au sens de *Dijkstra.*

---

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *
restrict attr, int *restrict prioceiling);

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
int prioceiling);

```
$ cat producteurconsommateur.c
/*   programme producteurconsommateur.c
     auteur Gilles GOUBET, Écrit en mars 2010
     Ce programme active un module Qui exÉcute deux threads
     qui fonctionnent selon le shema producteur consommateur.
     Le premier écrit dans un caractère que lit le second.
     Deux sémaphores règlent l'accès a ce caractère.
*/
#include <time.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
pthread_t prod,cons;  /*  Deux descripteurs de thread */
/* les sémaphores */
sem_t  sem_prod,  sem_cons;
char buffer;
/*
  Le producteur
*/
void * leProducteur(void * arg)
{
  char c='A';
  struct sched_param p;
  p . sched_priority = 1;
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
  c--;
  do
    {
      sem_wait(&sem_prod);
      c++;
      printf("Caractere ecrit %c\n",c);
      buffer=c;
      sem_post(&sem_cons);
    }
  while ( c != 'Z');
  return 0;
}
/*
  Le consommateur
*/
void * leConsommateur(void * arg)
{
  struct sched_param p;
  struct timespec temps;
  char c;
  temps.tv_sec=0;
  temps.tv_nsec=3000000;
  p . sched_priority = 1;
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
  nanosleep(&temps,NULL);
```

```
  do
    {
      sem_wait(&sem_cons);
      c=buffer;
      printf("Caractere lu %c\n",c);
      sem_post(&sem_prod);
    }
  while( c != 'Z');
  return 0;
}
main() {
      int ret;
      if ( sem_init(&sem_prod,0,1))
         { printf("Erreur creation sema producteur\n");
           return 4;
         }
      if ( sem_init(&sem_cons,0,0))
         { printf("Erreur creation sema consommateur\n");
           return 4;
         }
      pthread_create (&prod, NULL, leProducteur , 0);
      pthread_create (&cons, NULL, leConsommateur , 0);
      if (!pthread_join(cons,NULL))
      {
       printf("Le thread est termine\n");
      }
      if (!pthread_join(prod,NULL))
      {
       printf("Le thread est termine\n");
      }
      sem_destroy(&sem_prod);
      sem_destroy(&sem_cons);
}
```

La déclaration du sémaphore d'exclusion mutuelle

sem_t sem_prod, sem_cons;

L'initalisation du sémaphore

#include <semaphore.h>

*int sem_init(sem_t *sem, int pshared, unsigned int value);*

La fonction **sem_init** initalise le sémaphore sem. Quand l'entier pshared vaut 0, cela veut dire que le sémaphore est partagé par les threads du process (c'est notre cas) ou par des process sinon.

L'entier value est la valeur initiale du sémaphore. Pour le producteur, cette valeur est souvent la taille du buffer dans lequel est stocké chaque production et, pour le consommateur 0.

Dans notre cas le producteur ne peut produire qu'une fois (valeur 1) et le consommateur ne peut pas consommer tant qu'une production n'a pas été réalisée (valeur 0).

sem_init(&sem_prod,0,1)

sem_init(&sem_cons,0,0)

La destruction du sémaphores

#include <semaphore.h>
*int sem_destroy(sem_t *sem);*

sem_destroy(&sem_prod);
sem_destroy(&sem_cons);

La demande de consommation ou de production

*int sem_wait(sem_t *sem);*

La fonction **sem_wait** décrémente le compteur du sémaphore. Si le compteur devient négatif, le thread demandeur est bloqué tant que le compteur ne devient pas positif.

sem_wait(&sem_prod);
sem_wait(&sem_cons);

L'activation du producteur ou du consommateur

#include <semaphore.h>
*int sem_post(sem_t *sem);*

La fonction **sem_post** incrémente le sémaphore. Si le compteur devient positif est qu'un thread est bloqué sur le sémaphore alors le thread bloqué entre en section critique.

sem_post(&sem_prod);
sem_post(&sem_cons);

Pour aller plus loin

Primitives de gestion de l'arrêt d'un autre thread.

Des primitives permettent l'arrêt d'un thread par un autre.

Le thread auquel il est demandé de s'arrêter peut :

1. ignorer la demande
2. l'exécuter immédiatement
3. la prendre en compte plus tard

Quand un thread accepte de se terminer sur demande externe, elle exécute implicitement la primitive **pthread_exit (pthread_exit(PTHREAD_CANCELED))**.

Primitives

int pthread_cancel(pthread_t thread);

La primitive pthread_cancel envoie une demande d'arrêt au thread argument.

*int pthread_setcancelstate(int state, int *oldstate);*

La primitive change l'état du thread appelant pour signifier s'il accepte ou ignore les demandes d'arrêt (PTHREAD_CANCEL_ENABLE ou PTHREAD_CANCEL_DISABLE). Le champ oldstate peut être NULL.

*int pthread_setcanceltype(int type, int *oldtype);*

La primitive change le comportement du thread appelant s'il reçoit une demande d'arrêt. Le type PTHREAD_CANCEL_ASYNCHRONOUS provoque l'arrêt immédiat alors que PTHREAD_CANCEL_DEFERRED diffère l'arrêt jusqu'au prochain point d'arrêt. Un thread est créé avec les attributs PTHREAD_CANCEL_ENABLE et PTHREAD_CANCEL_DEFERRED. Les demandes d'arrêts différées sont exécutées lors de la demande d'exécution de primitives POSIX. Les fonctions POSIX qui suivent sont des points d'arrêts possibles quand elles sont exécutées :

- pthread_join
- pthread_cond_wait
- pthread_cond_timedwait
- pthread_testcancel
- sem_wait
- sigwait

Autres primitives

Primitive pthread_atfork

*int pthread_atfork(void (*prepare)(void), void (*parent)(void), void (*child)(void));*

Cette primitive permet de faire exécuter des fonctions quand la primitive **fork(2)** est exécutée.

Le gestionnaire prépare est exécuté par le processus père avant la création du process fils.

Le gestionnaire parent est exécuté par le processus père avant le retour du fork.

Le gestionnaire child est exécuté par le processus fils avant le retour du fork.

Lors d'un fork, les sémaphores sont copiés dans le processus fils, tels qu'ils sont. Seul le thread courant est actif dans le processus fils. Les gestionnaires permettent de rétablir l'état correct des sémaphores dans le processus fils.

Les gestionnaires peuvent être positionnés à NULL.

Primitive pthread_once

*int pthread_once(pthread_once_t *once_control, void (*init_routine) (void));*

La primitive **pthread_once** permet de s'assurer qu'une zone de code a bien été exécutée une fois (celle contenant l'appel à pthread_once). La variable once_control a été initialisée statiquement à

PTHREAD_ONCE_INIT Lors du premier appel à pthread_once, la valeur de once_control est modifiée pour se souvenir que le code a été exécuté et la fonction init_routine est exécutée.

Primitives de gestion des signaux

Primitive pthread_sigmak

#include <signal.h>
*int pthread_sigmask(int how, const sigset_t *newmask, sigset_t *oldmask);*

Le champ how dit si les signaux de newmask constituent le nouveau masque (SIG_SETMASK) ou viennent s'ajouter aux signaux déjà masqués (SIG_BLOCK). pour une description plus complète, consultez la gestion des signaux POSIX.

Primitive pthread_kill

int pthread_kill(pthread_t thread, int signo);

La primitive permet d'envoyer un signal à un thread.

Primitive sigwait

*int sigwait(const sigset_t *set, int *sig);*

La primitive suspend l'exécution du thread jusqu'à réception de l'un des signaux défini par set. le signa reçu est mémorisé dans sig.

Primitives de définition de condition

Une condition permet à un thread de suspendre son exécution et ne plus utiliser le processeur jusqu'à ce qu'un prédicat (condition) soit réalisé sur une variable partagée. Les primitives sont les suivantes :

Primitive pthread_cond_init

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

*int pthread_cond_init(pthread_cond_t *cond,*
*pthread_con- dattr_t *cond_attr);*

Primitive pthread_cond_signal

*int pthread_cond_signal(pthread_cond_t *cond);*

Primitive pthread_cond_broadcast

*int pthread_cond_broadcast(pthread_cond_t *cond);*

Primitive pthread_cond_wait

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

Primitive pthread_cond_timedwait

*int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);*

Primitive pthread_cond_destroy

*int pthread_cond_destroy(pthread_cond_t *cond);*