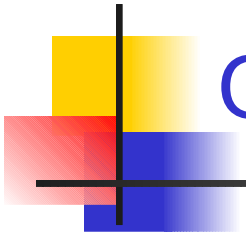




Luc Courtrai

# CONCURRENCE

**Université de Bretagne SUD**
**UFR SSI - Departement MIS**



# Plan

Notion de processus

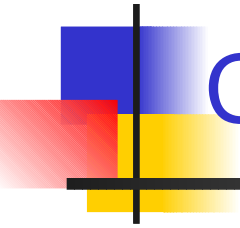
Les appels système UNIX

La synchronisation entre processus

La communication entre processus

Les threads JAVA

**Les Posix threads**



# Concurrence:  Les Posix threads

Les Posix Threads

C'est une API Standardisée
(liste de fonctions C)
à implanter
(mode USER ou NOYAU)

# Concurrence: Les Posix threads

## Création d'un Thread

NAME

pthread_create - créé un nouveau thread

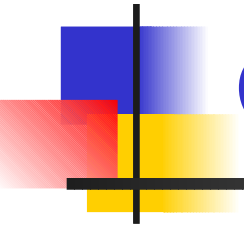
SYNOPSIS

#include <pthread.h>

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

DESCRIPTION

pthread_create créé un nouveau thread s'éxécutant concurremment avec le thread appelant. Le nouveau thread exécute la fonction start_routine en lui passant arg comme premier argument. Le nouveau thread s'achève soit explicitement en appelant pthread_exit(3), ou implicitement lorsque la fonction start_routine s'achève. Ce dernier cas est équivalent à appelér pthread_exit(3) avec la valeur renvoyée par start_routine comme code de sortie.



# Concurrence: Les Posix threads

VALEUR RENVOYÉE

En cas de succès, l'identifiant du nouveau thread est stocké à l'emplacement mémoire pointé par l'argument thread, et 0 est renvoyé. En cas d'erreur, un code d'erreur non nul est renvoyé.

ERREURS

EAGAIN pas assez de ressources système pour créer un processus pour le nouveau thread.

EAGAIN il y a déjà plus de PTHREAD_THREADS_MAX threads actifs.

# Concurrence:  Les Posix threads

EXEMPLE

```
// Thread.c
// Lancement de deux threads
//

#include<pthread.h> // API

// Code du thread
void * thcode(void *) {

  printf("numero de thread %d\n",pthread_self());

  // ....
  pthread_exit(NULL);
}
```

# Concurrence: Les Posix threads

```
  // programme principal

int main(void){
  pthread_t t_id1,t_id2; // identifieurs de threads

  // pthread_init(); en mode USER LEVEL (lance l'ordonnanceur)

  // crée et lance le premier thread (attribut par default)
  pthread_create(&t_id1, NULL, thcode, (void *)NULL);
  // crée et lance le deuxieme thread
  pthread_create(&t_id2, NULL, thcode, (void *)NULL);

  // attente les 2 threads
  pthread_join(t_id1, NULL);
  pthread_join(t_id2, NULL);
}
//  gcc –pthread exemple.c
```

# Concurrence:  Les Posix threads

EXEMPLE

Avec des paramètres

```
#include<pthread.h> // API

// Code du thread
void * thcode(int *i){ // void * startRoutine(void *)

  printf("numero de thread %d\n",pthread_self());

  // ....
  pthread_exit(NULL); // void pthread_exit(void *retval);
}
```

# Concurrence: Les Posix threads

```
  // programme principal

int main(void){
  pthread_t t_id1,t_id2; // identifieurs de threads
  int i=1,j=2;  // parametres des fonctions


  // crée et lance le premier thread (attribut par default)
  pthread_create(&t_id1, NULL, thcode, (void *)&i);
  // crée et lance le deuxieme thread
  pthread_create(&t_id2, NULL, thcode, (void *)&j);


}
//  gcc –pthread exemple.c
```

# Concurrence: Les Posix threads

```
/* Attributes for threads.  */
typedef struct
{
  int __detachstate;
  int __schedpolicy;
  struct __sched_param __schedparam;
  int __inheritsched;
  int __scope;
  size_t __guardsize;
  int __stackaddr_set;
  void *__stackaddr;
  size_t __stacksize;
} pthread_attr_t;
```

# Concurrence: Les Posix threads

```
pthread_attr_t        pta;

  //initilialise la structure
  rc = pthread_attr_init(&pta);
```

Etat détaché ou joignable
mode DETACHED les ressources du threads sont liberées dès la fin de thread
mode JOINABLE les ressouces du threads sont liberées lorsque le thread parent effectue le pthread_join

```
rc = pthread_attr_setdetachstate(&pta, PTHREAD_CREATE_DETACHED);
                        // PTHREAD_CREATE_JOINABLE
```

# Concurrence: Les Posix threads

```
// Politique d'ordonnancement
pthread_attr_setschedpolicy(&pta,SCHED_RR);

    SCHED_OTHER  (processus  normal (temps partagé))
    SCHED_RR (temps reel, round-robin (partage entre les thread RR)
    SCHED_FIFO (temps-réél, premier dans  la  liste, premier exécuté
     //                  pas de temps partagé
```

Seul root peut créer des threads de type SCHED_RR et SCHED_FIFO

# Concurrence: Les Posix threads

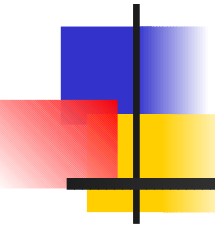
## Priorité

```
struct sched_param    param;
memset(&param, 0, sizeof(param));
param.sched_priority = XX ; //  XX < PRIORITY_MAX_NP
                        // [0 0]  pour OTHER
                        // [1 99]  pour RR
                        // [1 99]  pour FIFO
pthread_attr_setschedparam(&attr, &param);
```

Seul root peut modifier ces priorités  kernel 2.28.

# Concurrence:  Les Posix threads

```
// taille de la pile (INUTILE SUR LINUX THREAD)
size_t taille = 1000000;
pthread_attr_setstacksize(&pta,taille);
```



# Concurrence: Les Posix threads

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

pthread_join suspend l'exécution du thread appelant jusqu'à ce que le thread identifié par th achève son exécution, soit en appelant pthread_exit(3) soit après avoir été annullé.

Si thread_return ne vaut pas NULL, la valeur renvoyée par th y sera enregistrée. Cette valeur sera l'argument passé à pthread_exit(3)

# Concurrence: Les Posix threads

```
// code du thread
void * thcode(void *p){
  int ret =1;
  pthread_exit((void *) ret);
}

// programme principal
int main(void){
  pthread_t t_id1;
  pthread_create(&t_id1, NULL, thcode, (void *)NULL);
  {
    int cr;
    pthread_join(t_id1, (void **)( & cr));
    printf("code de retour %d\n",cr);
  }
}
```

# Concurrence: Les Posix threads

**int pthread_detach(pthread_t th);**

pthread_detach place le thread th dans l'état détaché. Cela garantit que les ressources mémoires consommées par th seront immédiatement libérées lorsque l'exécution de th s'achèvera. Cependant, celà empêche les autres threads de se synchroniser sur la mort de th en utilisant pthread_join.

# Concurrence: Les Posix threads

**Synchronisation : Les Verrous**

```
//Declaration d'un verrou  d'exclusion mutuelle

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
     //PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP
     //PTHREAD_ERREURCHECK_MUTEX_INITIALIZER_NP

int pthread_mutex_init(&lock,NULL);
// int   pthread_mutex_init(pthread_mutex_t   *mutex,   const
//       pthread_mutexattr_t *mutexattr);
//  attribut
//          -  rapide, (par defaut)
//          -  recursif ( il peut être verrouillé plusieurs fois par le même thread
//          -  verification d'erreur (C'est le thread qui
//                effectue le lock qui doit effectuer le unlock)
```

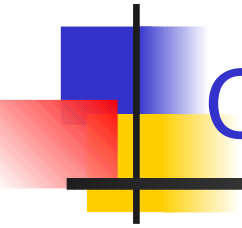
# Concurrence:  Les Posix threads

**Synchronisation : Les Verrous**

// **Exemple**
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);

// SECTION CRITIQUE

pthread_mutex_unlock(&lock);



# Concurrence: Les Posix threads

## Synchronisation : Les Verrous

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

si le verrou est déjà pris pthread_mutex_trylock
rend la main immédiatement avec le code d'erreur EBUSY

**int pthread_mutex_destroy(pthread_mutex_t *mutex);**
détruit le verrou

# Concurrence:  Les Posix threads

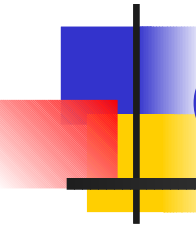
**Synchronisation : Les Sémaphores  POSIX 1003.1**

**SystemV**
**IPC tels que décrits par ipc(5), semctl(2) et semop(2).**

```
#include<semaphore.h>

semt_t sem;

extern int sem_init ((sem_t *__sem, int __pshared, unsigned int __value));
//                                          pshared = NULL  :
Linux gère  les sémaphores partagés entre plusieurs processus IPC
```



# Concurrence: Les Posix threads

**Synchronisation : Les Sémaphores**

/* Wait for SEM being posted. */
**extern int sem_wait ((sem_t *__sem));**

sem_wait suspend le thread appelant jusqu'à ce que le sémaphore pointé par sem aît un compteur non nul. Alors, le
compteur du sémaphore est atomiquement décrémenté.

/* Post SEM. */
**extern int sem_post ((sem_t *__sem));**

sem_post incrémente atomiquement le compteur du sémaphore pointé par sem. Cette fonction ne bloque jamais et peut être utilisée de manière fiable dans un gestionnaire de signaux.

# Concurrence: Les Posix threads

**Synchronisation : Les Sémaphores**

/* Test whether SEM is posted. */
**extern int sem_trywait ((sem_t *__sem));**

sem_trywait est une variante non bloquante de sem_wait. Si le sémaphore pointé par sem a une valeur non nulle, le compteur est atomiquement décrémenté et sem_trywait retourne immédiatement 0. Si le compteur du sémaphore est zéro, sem_trywait retourne immédiatement en indiquant l'erreur EAGAIN.

/* Get current value of SEM and store it in *SVAL. */
**extern int sem_getvalue ((sem_t *__sem, int *__sval));**

sem_getvaleur sauvegarde à l'emplacement pointé par sval la valeur courrante du compteur du sémaphore sem.

# Concurrence:  Les Posix threads

**Synchronisation : Les Sémaphores**

Exemple du producteur consomateur avec un tampon borné
lecture de lignes sur l'entree standard

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


#define BSIZE 5  // taille du tampon borné
#define MAXSTR   512 // taille d'une chaine

// semaphores
sem_t s_libre;// place libre
sem_t s_article; // article dans le tampon

// tampon borné partagé
char shared_buffer[BSIZE][MAXSTR];
```

# Concurrence: Les Posix threads

**Synchronisation : Les Sémaphores**

```
// code du producteur
void producter( ){
   int place=0; // indice dans le tampon
   char buf[MAXSTR]; // chaine intermédaire
   printf("Producteur %ld\n",pthread_self());
   // lecture des ligne sur l'entrée standard
   while(fgets(buf,MAXSTR,stdin) != NULL) {
        printf("Producteur :%s",buf);
        sem_wait(&s_libre);
        strcpy(shared_buffer[place],buf);
        sem_post(&s_article);
        place = (place+1)%BSIZE;
   }// fin des entrées émet un marqueur de fin
   sem_wait(&s_libre);
   strcpy(shared_buffer[place],"EOF"); // code de sortie
   sem_post(&s_article);
   pthread_exit(0);
}
```

# Concurrence: Les Posix threads

**Synchronisation : Les Sémaphores**

```
// code du consommateur
void consumer( ) {

   int place=0; // indice dans le tampon

   printf("consommateur %ld\n",pthread_self());

   while(1) {
        sem_wait(&s_article);
        printf("consommateur :%s",shared_buffer[place]);
        sem_post(&s_libre);
        if (strcmp(shared_buffer[place],"EOF")==0)    break;
        place = (place+1)%BSIZE;
   }
   printf("Sortie Consommateur \n");
   pthread_exit(0);
}
```

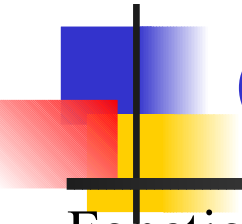
# Concurrence: Les Posix threads

**Synchronisation : Les Sémaphores**

```
// programme principal
int main(){
  pthread_t prod,cons;
  // initialise les deux semaphores
  sem_init(&s_article,0,0);
  sem_init(&s_libre,0,BSIZE); // nombre de places libres
  // lancement des threads producteur et consommateur
  pthread_create(&prod, NULL, (start_routine) producter, (void *) NULL);
  pthread_create(&cons, NULL, (start_routine) consumer, (void *) NULL);

  pthread_join(prod, NULL);  // attend le producteur
  pthread_detach(prod);// libération des ressources du producteur
  pthread_join(cons, NULL); pthread_detach(cons)

  sem_destroy(&s_article); // libère les sémaphores
  sem_destroy(&s_libre);
}
```



# Concurrence: Les Posix threads

Fonctions diverses :

**int pthread_yield();**
rend la main à l'ordonnanceur pour executer d'autres tâches

**pthread_delay_np** // non portable
**unsigned int sleep (unsigned int nb_sec);**
**void usleep (unsigned long usec);**
endort un processus lourd (ou un Linux Thread)

# Concurrence: Les Posix threads

Fonctions diverses :

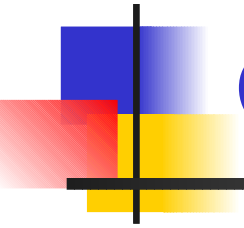
**int pthread_cancel(pthread_t thread);**
**int pthread_setcancelstate(int state, int *etat_pred);**
**void pthread_testcancel(void);**

L'annulation est le mécanisme par lequel un thread peut interrompre l'exécution d'un autre thread. Les fonctions suivantes relatives sont des points d'annulation implicite: pthread_join(3); pthread_cond_wait(3), pthread_cond_timedwait(3), pthread_testcancel(3),sem_wait(3) sigwait(3)

# Concurrence: Les Posix threads

```
void * thcode(void *p){
  // interdit les cancels
  // pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL);

  pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL);
                        // PTHREAD_CANCEL_ASYNCHRONOUS,
  while (1) {
      int i; float f=1;
      for (i=0;i<10000000;i++) f/=i;
      pthread_testcancel();
  }
  pthread_exit(&ret); // jamais execute
}
```



# Concurrence: Les Posix threads

```
// programme principal
int main(void){

   pthread_t t_id1;

   pthread_create(&t_id1, NULL, thcode, (void *)NULL);

   pthread_cancel(t_id1); // demande d'annulation


   pthread_join(t_id1, NULL);
}
```

# Concurrence: Les Posix threads

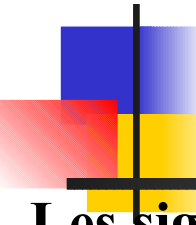
**Les signaux**

La gestion des signaux suit la nome POSIX utilisée pour les processus Lourds.

La structure sigaction spécifie la fonction (action) à lancer à la reception d'un signal.

```
struct sigaction      actions;
memset(&actions, 0, sizeof(actions));
sigemptyset(&actions.sa_mask); // vide le masque (l'ensemble)
actions.sa_flags = 0;
actions.sa_handler = fhandler; // fonction à lancer
```

**Avec** `void fhandler(int signo);`



# Concurrence: Les Posix threads

**Les signaux :**

**Arme le signal pour tous les threads de l'appli**

rc = **sigaction**(SIGUSR1,&actions,NULL);

// pour le signal SIGUSR1 on branche ``actions''

// NULL pour recupérer éventuellement l'ancienne action pour ce signal

**Envoie un signal**

int **pthread_kill**(pthread_t thread, int signo);

**Suspend le thread en attente d'un signal spécifique**

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

Cette fonction n'est pas forcément utilisée dans les applications

# Concurrence: Les Posix threads

```
#define SIGHUP        1      /* Hangup (POSIX).  */
#define SIGINT        2      /* Interrupt (ANSI).  */
#define SIGQUIT       3      /* Quit (POSIX).  */
#define SIGILL        4      /* Illegal instruction (ANSI).  */
#define SIGTRAP       5      /* Trace trap (POSIX).  */
#define SIGABRT       6      /* Abort (ANSI).  */
#define SIGIOT        6      /* IOT trap (4.2 BSD).  */
#define SIGBUS        7      /* BUS error (4.2 BSD).  */
#define SIGFPE        8      /* Floating-point exception (ANSI).  */
#define SIGKILL       9      /* Kill, unblockable (POSIX).  */
#define SIGUSR1       10     /* User-defined signal 1 (POSIX).  */
#define SIGSEGV       11     /* Segmentation violation (ANSI).  */
#define SIGUSR2       12     /* User-defined signal 2 (POSIX).  */
#define SIGPIPE       13     /* Broken pipe (POSIX).  */
#define SIGALRM       14     /* Alarm clock (POSIX).  */
#define SIGTERM       15     /* Termination (ANSI).  */
#define SIGSTKFLT     16     /* Stack fault.  */
#define SIGCLD        SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD       17     /* Child status has changed (POSIX).  */
#define SIGCONT       18     /* Continue (POSIX).  */
#define SIGSTOP       19     /* Stop, unblockable (POSIX).  */
#define SIGTSTP       20     /* Keyboard stop (POSIX).  */
#define SIGTTIN       21     /* Background read from tty (POSIX).  */
#define SIGTTOU       22     /* Background write to tty (POSIX).  */
#define SIGURG        23     /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU       24     /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ       25     /* File size limit exceeded (4.2 BSD).  */
```

# Concurrence:  Les Posix threads

**Les signaux :**

**Chaque thread peut spécifier un masque de signaux bloqués**

pthread_sigmask(int how, const sigset_t  *newmask,sigset_t *oldmask)

```
sigset_t set;
  sigemptyset(&set); // vide le masque (l'ensemble) ;
  sigaddset(&set,SIGUSR1); // ajoute le signal dans la liste des bloqués
  pthread_sigmask(SIG_SETMASK,&set,NULL); // change le masque
```

# Concurrence: Les Posix threads

**Les signaux :**

**Exemple :**

```
void *threadfunc(void *parm){
  pthread_t             self = pthread_self();
  int                   rc;
  printf("Thread %x entered\n", (unsigned)self);
  rc = sleep(30);
  if (rc != 0 && errno == EINTR) {
    printf("Thread %x got a signal delivered to it\n",
           (unsigned)self);
    return NULL;
  }
  printf("Thread 0x didn't get expected results! rc=%d, errno=%d\n",
         (unsigned)self, rc, errno);
  return NULL;
}
```

# Concurrence: Les Posix threads

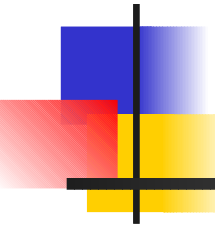
**Les signaux :**

```
int main(int argc, char **argv){
  struct sigaction        actions;
  pthread_t               threads[NUMTHREADS];
  printf("Set up the alarm handler for the process\n");
  memset(&actions, 0, sizeof(actions));
  sigemptyset(&actions.sa_mask); //vide le masque (l'ensemble)
  actions.sa_flags = 0;
  actions.sa_handler = sighand;
  //arme le signal
  rc = sigaction(SIGALRM,&actions,NULL);
  for(i=0; i<NUMTHREADS; ++i)
    rc = pthread_create(&threads[i], NULL, threadfunc, NULL);
  sleep(3);
  for(i=0; i<NUMTHREADS; ++i)
    rc = pthread_kill(threads[i], SIGALRM);
  for(i=0; i<NUMTHREADS; ++i)
    rc = pthread_join(threads[i], NULL);
}
```

# Concurrence: Les Posix threads

**Les signaux :**

```
void sighand(int signo){
  pthread_t self = pthread_self();
  printf("Thread  %x in signal handler\n“, (unsigned) self);
  pthread_exit(1);
}
```



# Concurrence: Les Posix threads

Synchronisation : les variables conditions (moniteur):

- signaler la condition (quand le prédicat devient vrai) et attendre la condition
- suspendre la condition jusqu'à ce qu'un autre thread signale la condition

Les variables conditions doivent être protégées par un mutex

➔ **pthread_cond_init** initialise la variable condition cond.
➔ **pthread_cond_signal** relance l'un des threads attendant la variable condition cond.
➔ **pthread_cond_broadcast** relance tous les threads attendant sur la variable condition cond.
**pthread_cond_wait** déverrouille atomiquement le mutex (comme pthread_unlock_mutex) et attend que la variable condition cond soit signalée. Le mutex est recupéré à la sortie du wait

# Concurrence: Les Posix threads

PRODUCTEUR CONSOMMATEUR AVEC VARIABLES CONTITIONS

```
#define BSIZE 5  // taille du tampon borne
#define MAXSTR   512 // taille d'une chaine

pthread_mutex_t mutex =PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t plein=PTHREAD_COND_INITIALIZER;
pthread_cond_t vide=PTHREAD_COND_INITIALIZER;

int nbLigne=0; // nombre de ligne dans le tampon

// tampon bornee partage
//
char shared_buffer[BSIZE][MAXSTR];
```

# Concurrence: Les Posix threads

PRODUCTEUR CONSOMMATEUR AVEC VARIABLES CONTITIONS

```
void producter( ){
    int place=0; // indice dans le tampon
    char buf[MAXSTR]; // chaine intermedaire

    // lecture des lignes sur l'entree standard
    while(fgets(buf,MAXSTR,stdin) != NULL) {
         printf("Producteur :%s",buf);
         pthread_mutex_lock(&mutex);
         while (nbLigne == BSIZE)
                pthread_cond_wait(&plein,&mutex);
         strcpy(shared_buffer[place],buf);
         place = (place+1)%BSIZE;
         nbLigne++;
         pthread_mutex_unlock(&mutex);
         pthread_cond_signal(&vide);
    }
}
```

# Concurrence: Les Posix threads

```
void consumer( ) {
   int place=0; // indice dans le tampon
   char str[MAXSTR];
   while(1) {
      pthread_mutex_lock(&mutex);
      while (nbLigne == 0)
          pthread_cond_wait(&vide,&mutex);
     strcpy(str,shared_buffer[place]);
     place = (place+1)%BSIZE;
     nbLigne--;
     pthread_mutex_unlock(&mutex);
     pthread_cond_signal(&plein);
    printf("consommateur %s\n",str);
   }
}
```

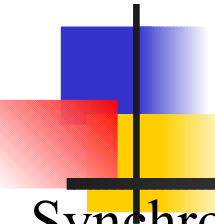
# Concurrence: Les Posix threads

Synchronisation : les variables conditions :

**Ecriture des semaphores avec le lock et les conditions**

```
typedef struct sem {
   pthread_mutex_t lock;
   pthread_cond_t  cond;
   int count;
} sem_t;

void sem_init(sem_t *s,int init){
   pthread_mutex_init(&s->lock,NULL);
   pthread_cond_init(&s->cond,NULL);
   s->count = (init <0 ? 0: init);
}
```



# Concurrence: Les Posix threads

Synchronisation : les variables conditions :

Ecriture des semaphores avec le lock et les conditions

```
void sem_p(sem_t *s){
  pthread_mutex_lock(&s->lock);
  while (s->count ==0)
     pthread_cond_wait(&s->cond,&s->lock);
  // lors de la liberation par un notify
  // il se met en concurence avec le autre thread
  // don un autre P peut passer decremente le compteur
  // et libère le verrou
  // lorsqu'il aura le verrou il faut donc qu'il reteste le compteur
  s->count --;
  pthread_mutex_unlock(&s->lock);
}
```

# Concurrence:  Les Posix threads

Synchronisation : les variables conditions  :

Ecriture des semaphores avec le lock et les conditions

```
void sem_v(sem_t *s){
  int contention;

  pthread_mutex_lock(&s->lock);
  contention = (s->count++ ==0);
  if (contention)
     pthread_cond_signal(&s->cond);
  pthread_mutex_unlock(&s->lock);
}
```