Table des matières

1. PROCESSUS LOURD - PROCESSUS LÉGER

1.1. Processus lourd

Un processus lourd correspond à une unité d'ordonnancement.
Au niveau du système, il est caractérisé par un PCB (Process Control Block) qui contient :

- Le contexte mémoire
- Les segments mémoire :
  - le segment de code : code du programme exécuté par le processus (peut être partagé par plusieurs processus).
  - le segment de données : zone des variables globales et statiques, ainsi que du tas, géré dynamiquement.
  - le segment de pile : zone de transmission des paramètres, des retours de fonctions et des variables locales.

- L'ensemble des variables d'environnement et de leurs valeurs
- La priorité du processus pour l'attribution de l'UC.
- Les descripteurs des fichiers utilisés dont les trois canaux standards qui sont ouverts automatiquement à la création du processus :
  - 0 : **STDIN**(par défaut : le clavier) ;
  - 1 : **STDOUT** (par défaut : l'écran) ;
  - 2 : **STDERR** (par défaut : l'écran).

Chaque processus possédant ses propres segments de données et de pile, il n'est pas possible de partager des données entre les processus. Il est nécessaire d'utiliser des outils de communication entre processus (IPC) comme les zones de mémoire partagée.

1.2. Processus légers : Threads.

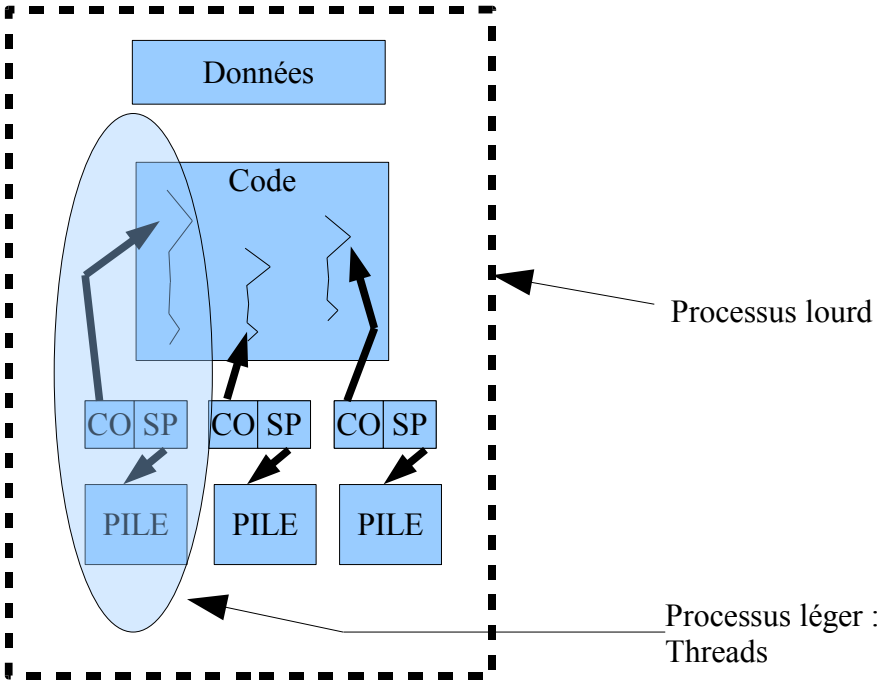
Un ou plusieurs Threads s'exécutent en concurrence dans un processus lourd.
=> un processus lourd contient au moins 1 thread (le thread main)
Les threads s'exécutant au sein d'un même processus partagent :

. le segment de code,
. le segment de données,
. le tas (heap),
. les descripteurs de fichiers ouverts,
. le répertoire de travail, userid et groupid,
. les handlers de signaux.

Chaque thread possède :

. un mini-PCB (son Compteur Ordianal :CO + quelques autres registres),
. sa pile et le Pointeur de Pile associé(SP) ,
. ses attributs d'ordonnancement (priorité, état, etc.)
. structures pour le traitement des signaux (masque et signaux pendants).

Un processus lourds hébergeant trois threads



Avantages :

Création plus rapide
Partage des ressources
Communication entre threads plus simple (variables globales)

Inconvénient :

Synchronisation et exclusion plus difficile à gérer

1.3. Thread noyau et thread utilisateur

La norme POSIX définit deux types d'implémentation des threads :

→ **Thread noyau :**

* Les threads sont des entités du système
* Le système possède un descripteur pour chaque thread.
* Permet l'utilisation des différents processeurs dans le cas des machines multiprocesseurs.

→ **Thread utilisateur :**

* L'état est maintenu en espace utilisateur. Aucune ressource du noyau n'est allouée au thread.
* Des opérations peuvent être réalisées indépendamment du système.
* Le noyau ne voit qu'une seule thread : tout appel système bloquant un thread aura pour effet de bloquer son processus et par conséquent toutes les autres threads du même processus.

1.4. Réentrance :

Un fonction réentrante est une fonction qui peut être appelée simultanément par plusieurs Threads.

=> pas de manipulation de variable globale
=> utilisation de mécanismes de synchronisation permettant de régler les conflits provoqués par des accès concurrents.

Remarques : Une librairie est dite

*** multithread-safe (MT-safe) :**
=> fonctions réentrantes vis-à-vis du parallélisme
*** async-safe :**
=> fonctions réentrantes vis-à-vis des signaux

2. LA LIBRAIRIE PTHREAD

La librairie Pthread définit un ensemble d'objets (structures) et de fonction permettant de gérer les paramètres et de contrôler le comportement des threads (créer, lancer ..) ainsi que des outils de synchronisation (condition, mutex).

En règle générale :

- Un Pthread est identifié par un ID unique
- En cas de succès une fonction renvoie 0 et une valeur différente de NULL en cas d'échec.
- Pthreads n'indiquent pas l'erreur dans *errno*. (Possibilité d'utiliser *strerror*.)
- Fichier <*pthread.h*> : déclaration des constantes et prototypes des fonctions.
- Fichier <errno.h> pour les code d'erreur retourné par les fonctions
- Faire le lien avec la bibliothèque *libpthread.a :* gcc …. –l pthread
- Directive #define _REENTRANT ou gcc … -D _REENTRANT

2.1. Nommage des objets et des fonctions

Les type correspondant aux différentes structures définies par la librairie pthread respectent la convention de nomage ci dessous :

Nom	Signification / utilisation
pthread_t	identifiant d'un *thread*
pthread_attr_t	attributs d'un *thread*
pthread_mutex_t	*mutex* (exclusion mutuelle)
pthread_mutexattr_t	attributs d'un *mutex*
pthread_cond_t	variable de condition
pthread_condattr_t	attributs d'une variable de condition
pthread_key_t	clé pour accès à une donnée globale réservée
pthread_once_t	initialisation unique

Le nom d'une fonction définie dans Pthread.h s'obtient en :

Enlevant _t au type d'objet qu'elle manipule

En ajoutant un suffixe correspondant à l'action réalisée :

_init : initialiser un objet.
_destroy: détruire un objet.
_create : créer un objet.
_getattr : obtenir l'attribut *attr* des attributs d'un objet.
_setattr : modifier l'attribut *attr* des attributs d'un objet.

Par exemple :

pthread_create : crée un thread (objet *pthread_t*).
pthread_mutex_init : initialise un objet du type *pthread_mutex_t*.

3. GESTION DES THREADS

3.1. Généralités :

Un ***Thread*** :

- Est identifié par un ID unique.
- Exécute une fonction passée en paramètre lors de sa création.
- Possède des attributs.
- Peut se terminer (pthread_exit) ou être annulé par un autre thread (*pthread_cancel*).
- Peut attendre la fin d'un autre thread (pthread_join).
- Un Pthread possède son propre masque de signaux et signaux pendants.
- La création d'un processus donne lieu à la création du thread main.
- Retour de la fonction main entraîne la terminaison du processus et par conséquent de tous les threads de celui-ci.

3.2. Manipulation des attributs d'un thread

Attributs passés au moment de la création d'un thread :

- Paramètre du type pthread_attr_t
- Initialisation d'une variable du type pthread_attr_t avec les valeurs par défaut :
  int pthread_attr_init (pthread_attr_t *attrib) ;
- Chaque attribut possède un nom utilisé pour construire les prototypes de deux types fonctions :
  - pthread_attr_getnom (pthread_attr_t *attr, …)
    Extraire la valeur de l'attribut *nom* de la variable *attr*
  - pthread_attr_setnom (pthread_attr_t *attr, …)
    Modifier la valeur de l'attribut *nom* de la variable *attr*

où '*nom*' peut prendre les valeurs :

- **scope (*int*)** - thread noyau ou utilisateur
  PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS

- **stackaddr (*void **)** - adresse de la pile

- **stacksize (*size_t*)** - taille de la pile

- **detachstate (*int*)** - thread joignable ou détaché
  PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED

- **schedpolicy (*int*)** – type d'ordonnancement
  SCHED_OTHER (unix) , SCHED_FIFO (temps-réel FIFO), SCHED_RR (temps-réel round-robin)

- **schedparam (*sched_param **)** - paramètres pour l'ordonnanceur

- **inheritsched (*int*)** - ordonnancement hérité ou pas
  PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED

Exemples de fonctions :

- **Obtenir/modifier l'état de détachement d'un thread**

int pthread_attr_get**detachstate** (const pthread_attr_t *attributs,int *valeur);
int pthread_attr_set**detachstate** (const pthread_attr_t *attributs,int valeur)

- **Obtenir/modifier la taille de la pile d'un thread**

int pthread_attr_**getstacksize** (const pthread_attr_t *attributs,size_t *taille);
int pthread_attr_set**stacksize** (const pthread_attr_t *attributs,size_t taille);

- **Obtenir la taille de pile d'un thread**

pthread_attr_t attr; size_t taille;
pthread_attr_**getstacksize**(&attr, &taille);

- **Détachement d'un thread**

pthread_attr_t attr;
pthread_attr_**setdetachstate**(&attr, PTHREAD_CREATE_DETACHED);

- **Modifier la politique d'ordonnancement (temps-réel)**

pthread_attr_t attr;
pthread_attr_**setschedpolicy**(&attr,SCHED_FIFO);

3.3. Création d'un Thread :

A la création d'un thread on définit :

- Ces attributs
- La fonction qui sera exécutée
- Un paramètre passé au thread

int pthread_create(pthread_t *tid, *// [IN/OUT] ID du thread créé*
pthread_attr_t *attr, *//[IN/OUT] attribut du thread créé*
void * (*fonc) (void *), *// la fonction a exécuter*
void *arg); *// argument passé au thread*

- **attr** : si NULL, le thread est créé avec les attributs par défaut.
- code de renvoi :
  0 en cas de succès.
  En cas d'erreur une valeur non nulle indiquant l'erreur:
  EAGAIN : manque de ressource.
  EPERM : pas la permission pour le type d'ordonnancement demandé.
  EINVAL : attributs spécifiés par *attr* ne sont pas valables.

La création d'un processus (main) crée le thread principal,
L'appel à **pthread_create** crée un thread annexe

Remarque: On obtient l'ID d'un thread grace à la fonction
pthread_t pthread_self (void);

3.4. Terminaison d'un Thread

- Fin du thread principal (main) => fin de tout les threads crées dans main
- Retour de la fonction correspondante au thread ou appel à la fonction ***pthread_exit.***
  (aucun effet sur l'existence du processus ou des autres threads).
- L'appel à ***exit*** ou ***_exit*** par un thread annexe provoque la terminaison
  du processus et donc de tous les autres thread

3.5. Exemples

3.5.1. Création avec attributs standards

```
//*************************** creation_1.c*********************************//
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *test (void *arg)
{
        unsigned int i;
        printf ("Argument reçu %s, tid: %u\n" ,(char*)arg, (unsigned int)pthread_self());
        for (i=0; i <10000000; i++);
        printf ("fin thread %u\n",(unsigned int)pthread_self());
        return NULL;
}
```

```
int main (int argc, char ** argv)
{
        pthread_t tid;
        int L_ret;
        L_ret = pthread_create (&tid, NULL,test, "BONJOUR");
        if ( L_ret!= 0)
        {
                perror(" erreur pthread_create \n");
                exit (1);
        }
        sleep (3);
        printf ("fin thread main \n" );
        return EXIT_SUCCESS;
}
```

3.5.2. Terminaison d'un thread

Usage : ./terminaison <arg1> <arg2>
par exemple ./terminaison toto titi

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

#define NUM_THREADS 2

void *func_thread (void *arg)
{
        printf ("Argument reçu : %s, thread_id: %u \n", (char*)arg, (unsigned int) pthread_self());
        pthread_exit ((void*)0);
        return NULL;
}

int main (int argc, char ** argv)
{
        int i;
        pthread_t tid [NUM_THREADS];
        int L_ret;
        for (i=0; i < NUM_THREADS; i++)
        {
                L_ret = pthread_create (&(tid[i]), NULL, func_thread, argv[i+1]);
                if (L_ret != 0)
                {
                                printf ("erreur pthread_create \n");
                                exit (1);
                }
        }
        sleep (3);
        return EXIT_SUCCESS;
}
```

3.6. Thread joignable et thread détaché

On peut déterminer le comportement d'un thread en fin d'exécution de deux façon :

- Joignable (par défaut)
  Attribut : PTHREAD_CREATE_JOINABLE
  En se terminant suite à un appel à *pthread_exit,* les valeurs de son identité et de retour sont conservées jusqu'à ce qu'un autre thread en prenne connaissance (appel à *pthread_join*).
  Les ressources sont alors libérées.

- Détachée
  Attribut : PTHREAD_CREATE_DETACHED
  Lorsque le thread se termine toutes les ressources sont libérées.
  Aucun autre thread ne peut les récupérer.

3.6.1. Détaché un Thread

- **En cours d'éxécution :**
  Fonction *pthread_detach* :
  int pthread_detach(pthread_t tid);

- **Lors de sa création :** en changeant son atribut
  Exemple:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
pthread_create (tid, &attr, func, NULL);
```

3.6.2. Attendre la terminaison d'un Thread joignable

On utilise la fonction qui attend la fin du thread *tid.*

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

- *thread tid* doit appartenir au même processus que le thread appelant.
- Si le *thread tid* n'est pas encore terminé, le thread appelant sera bloqué jusqu'à ce que le *thread tid* se termine.
- Si le *thread tid* est déjà terminé, le thread appelant n'est pas bloqué.
- *Thread tid* doit être joignable.
  - Sinon la fonction renverra EINVAL.
- Un seul thread réussit l'appel.
  - Pour les autres threads, la fonction renverra la valeur ESRCH.
  - Les ressources du *thread* sont alors libérées.

- Lors du retour de la fonction pthread_join
  La valeur de terminaison de la *thread tid* est reçue dans la variable *thread_return* (pointeur).
  - Valeur transmise lors de l'appel à pthread_exit

- Si le thread a été annulé, thread_return prendra la valeur PTHREAD_ CANCEL.

- code de renvoi :
  - 0 en cas de succès.
  - valeur non nulle en cas d'échec:
    - ESRCH : thread n'existe pas.
    - EDEADLK : interblocage ou ID du thread appelant.
    - EINVAL : thread n'est pas joignable.

Exemple : le main crée 3 Threads et attend leur terminaison pour se terminer
usage : ./join <arg1> <arg2> <arg3>

```
//**********************************join.c*********************************//
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#define NUM_THREADS 3

void *func_thread (void *arg)
{
        printf ("Argument reçu %s, tid: %u\n",(char*)arg, (unsigned int)pthread_self());
        pthread_exit ((void*)0);
}

int main (int argc, char ** argv)
{
        int i,status L_ret;
        pthread_t tid [NUM_THREADS];
        for (i=0; i < NUM_THREADS; i++)
        {
                L_ret = pthread_create(&(tid[i]), NULL,func_thread, argv[i+1]);
                if (L_ret != 0)
                {
                        printf("erreur pthread_create \n"); exit (1);
                }
        }
        for (i=0; i < NUM_THREADS; i++)
        {
                L_ret = pthread_join (tid[i], (void**) &status);
                if (L_ret !=0)
                {
                        printf ("erreur pthread_join");
                        exit (1);
                }
                else
                {
                        printf ("Thread %d fini avec status :%d\n",i, status);
                }
        }
        return EXIT_SUCCESS;
}
```

3.6.3. Récupérer une valeur de retour :

Dans le programme suivant, les threads annexes retournent au thread principal le nombre de caractères de la chaîne passée en argument

usage ./join_1 <arg1> <arg2>

```
//*********************************join_1.c*******************************//
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define NUM_THREADS 2

void *func_thread (void *arg)
{
        unsigned int NB_car;
        NB_car = strlen((char*)arg);
        printf ("Argument reçu %s, tid: %u,\n",(char*)arg, (unsigned int)pthread_self());
        pthread_exit ((void*)(NB_car));
}

int main (int argc, char ** argv)
{
        int i,status;
        pthread_t tid [NUM_THREADS];
        for (i=0; i < NUM_THREADS; i++)
        {
        if (pthread_create (&(tid[i]), NULL,func_thread, argv[i+1]) != 0)
        {
                printf("pthread_create \n"); exit (1);
        }
        }
        for (i=0; i < NUM_THREADS; i++)
        {
        if (pthread_join (tid[i], (void**) &status) !=0)
        {
                printf ("pthread_join"); exit (1);
        }
        else
                printf ("Thread %d fini avec status :%u\n",i, status);
        }
        return EXIT_SUCCESS;
}
```

3.6.4. Thread joignable est détachable

Dans le programme suivant , le thread principal crée un thread annexe joignable et un thread annexe détaché.
Il attend ensuite la fin des threads annexes (join) pour récupérer la valeur de retour. Le thread détaché provoque une erreur .

```
//*******************************detach.c*******************************//
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define NUM_THREADS 2

void *func_thread (void *arg)
{
        unsigned int NB_car;
        NB_car = strlen((char*)arg);
        printf ("Argument reçu %s, tid: %u\n",(char*)arg, (unsigned int)pthread_self());
        pthread_exit ((void*)(NB_car));
}

int main (int argc, char ** argv)
{
        int i,status, L_ret;
        pthread_t tid1,tid2;
        pthread_attr_t attr;
        pthread_attr_init(&attr);

        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
        if ((pthread_create (&tid1, &attr, func_thread, argv[1])) != 0)
        {
                printf("erreur pthread_create \n");
                exit (1);
        }
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
        if ((pthread_create (&tid2, &attr, func_thread, argv[2])) != 0)
        {
                printf("erreur pthread_create \n");
                exit (1);
        }

        L_ret = pthread_join (tid1, (void**) &status);
        if (L_ret !=0)
        {
                printf ("Thread %u : erreur  %u pthread_detach \n",tid1,L_ret);
                exit (1);
        }
        else
                printf ("Thread %u fini avec status :%u\n",tid1, status);

        L_ret = pthread_join (tid2, (void**) &status);
        if (L_ret !=0)
        {
                printf ("Thread %u : erreur  %u pthread_detach \n",tid2,L_ret);
                exit (1);
        }
        else
                printf ("Thread %u fini avec status :%u\n",tid2, status);

        return EXIT_SUCCESS;
}
```

3.7. Exercice 1

Écrivez un programme qui crée deux threads annexes affichant chacun n fois un caractère c.

n et c seront passés en paramètre aux threads (dans une structure st_param que vous définirez)

Le thread principal crée les deux threds annexes joignables et attend leur terminaison

Rq : Entre deux affichages d'un caractère, faites consommer du temps processeur en faisant effectuer à votre threads quelques dizaines de millier de fois un calcul complexe (calcul du sin , multiplication, division par un nombre impaire d'un flottant ;
par exemple calc=sin (3.0*calc*calc)/5.0;)

Pour utiliser la fonction sin : incluez math.h et ajoutez -lm dans la ligne de compilation pour lier la librairie mathématique.

3.8. Exercice 2

Le programme exercice2.c permet d'afficher les entiers de 1 à 9 et leurs carrés. Un thread prend en charge l'affiche des nombres et deuxième thread calcul le carré d'un nombre et l'affiche.

Analysez le code source, compilez et testez ce programme.

Commentez son comportement.

Usage ./exercice <time_a> <time_b>
où time_a et time_b déterminent le temps consommé par chaque thread

4. EXCLUSION MUTUELLE

Nous pouvons constater que dans exercice 1 du chapitre précédent, les caractères se « mélangent » à l'écran. En effet le premier thread commence à afficher ses n caractères mais peut être préempté par le second thread qui commence alors l'affichage de siens pour à son tour être préempté …

Le fonctionnement du programme de l'exercice n'est pas satisfaisant .

Il faudrait faire en sorte qu'un thread puisse réserver une ressource (l'affichage écran) en empêchant un autre thread qui voudrait effectuer un affichage (utiliser cette ressource) de le préempter en le mettant en sommeil jusqu'à ce que la ressource soit libérée.

On dit que la ressource devient critique ou non partageable.

Le mécanisme permettant à un thread utilisant une ressource critique d'exclure tout autre thread désirant utiliser cette ressource est nommé « exclusion mutuelle » en raccourci MUTEX (MUTual Exclusion).

Le mutex est en fait un cas particulier de sémaphore dit sémaphore booléen en ce sens qu'il ne peut prendre que deux valeurs : libre et occupé.

Un mutex () est caractérisé par
Un nom
Une variable booléenne : libre, occupé
Une file d'attente des processus (thread)
deux primitives atomiques
Lock et Unlock

4.1. Pthread et les mutex.

4.1.1. Opérations sur le mutex

- ***Création :***

  - ***Statique:***
    pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

  - ***Dynamique:***
    int pthread_mutex_init(pthread_mutex_t *m, pthread_mutex_attr *attr);

    Attributs initialisés par un appel à :
    int pthread_mutexattr_init(pthread_mutex_attr *attr);

  - ***Exemple :***
    pthread_mutex_t verrou;
    /* attributs par défaut */
    pthread_mutex_init(&verrou, NULL);

- ***Destruction :***
  int pthread_mutex_destroy (pthread_mutex_t *m)

- ***Verrouillage :***
  int pthread_mutex_lock (pthread_mutex_t *m);
  → opération bloquante si déjà le mutex est déjà verrouillé

  int pthread_mutex_trylock (pthread_mutex_t *m);
  → Renvoie EBUSY si le mutex est déjà verrouillé

- ***Déverrouillage:***
  int pthread_mutex_unlock (pthread_mutex_t *m);

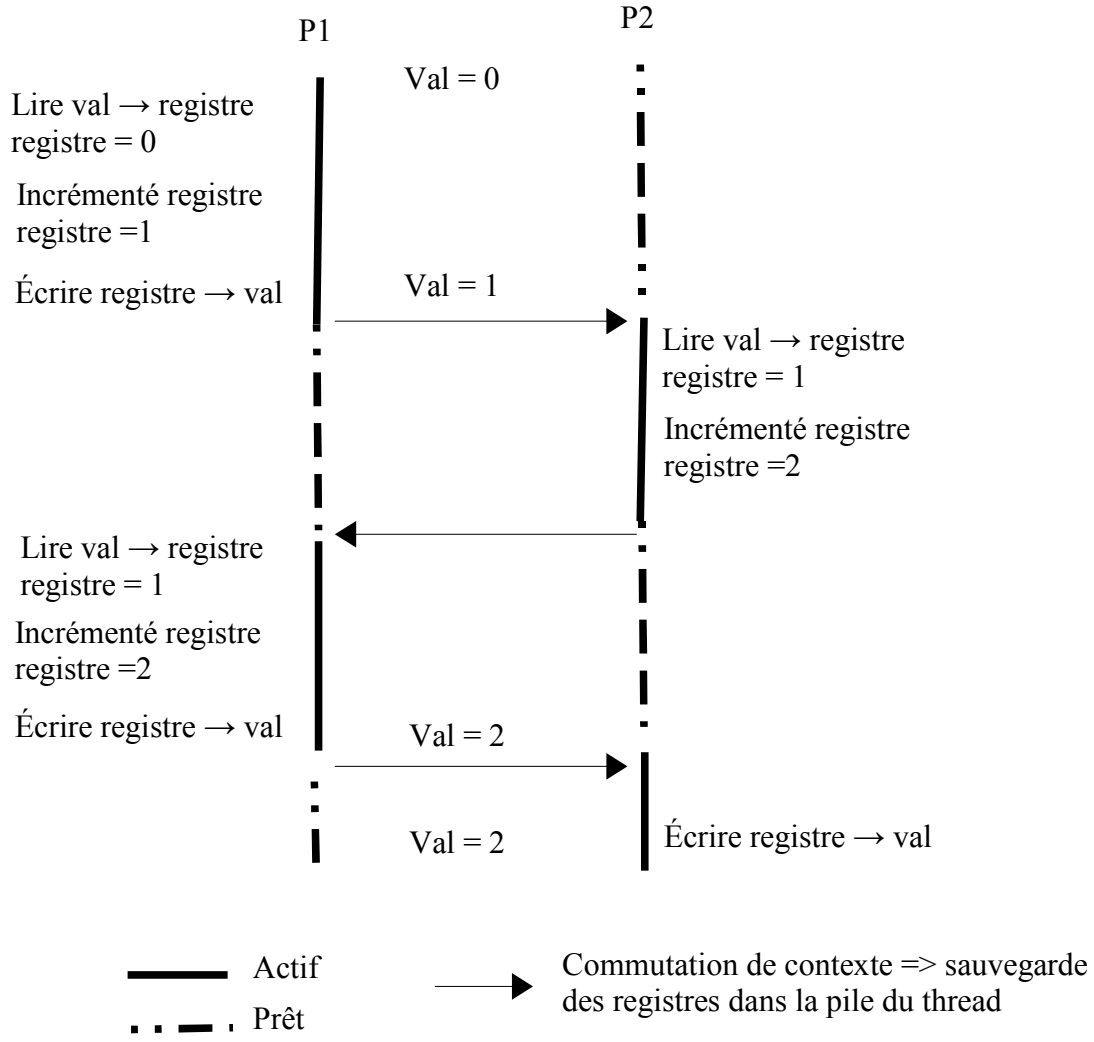
4.2. Exemples :

Le programme count.c si dessous illustre le cas typique dans lequel deux threads modifient une variable partagée (globale) .
On peut avoir le cas suivant :

val est un entier global initialisés à 0
P1 et P2 sont deux threads qui incrémente val en boucle :
l'instruction val ← val + 1 est en fait décomposée en :

* lecture de val en mémoire et stockage dans un registre
* incrémentation du registre
* écriture du registre dans val

Le déroulement du programme peut être le suivant



On remarque que la valeur de val est erronée.
L'incrémentation de val est donc une section critique qu'il faut protéger par un MUTEX.
C'est ce qu'illustre le programme count.c ci dessous.

```
//*********************************count.c*****************************//
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
        pthread_mutex_t mutex =PTHREAD_MUTEX_INITIALIZER;
        int compteur =0;

void *sum_thread (void *arg)
{
        pthread_mutex_lock (&mutex);
        compteur++;
        pthread_mutex_unlock (&mutex);
        pthread_exit ((void*)0);
}

int main (int argc, char ** argv)
{
        pthread_t tid1,tid2;
        if (pthread_create (&tid1, NULL, sum_thread,NULL) != 0)
        {
        printf("erreur pthread_create"); exit (1);
        }
        if (pthread_create (&tid2, NULL, sum_thread,NULL) != 0)
        {
        printf("erreur pthread_create"); exit (1);
        }
        pthread_mutex_lock (&mutex);
        compteur++;
        pthread_mutex_unlock (&mutex);
        pthread_join (tid1, NULL);
        pthread_join (tid2, NULL);

        printf ("compteur : %d\n", compteur);
        return EXIT_SUCCESS;
}
```

Dans l'exemple ci-dessous, un thread tente d'accéder à une ressource critique verrouillée par un autre thread. L'appel à pthread_mutex_trylock ne provoque pas la mise en attente du thread mais retourne EBUSY.

```
//*********************************trylockt.c*****************************//
#include <errno.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

        pthread_mutex_t mutex =PTHREAD_MUTEX_INITIALIZER;
```

→ suite trylock.c

```
void *sum_thread (void *arg)
{

        pthread_mutex_lock (&mutex);
        printf ("thread %u : j'utilise la ressource \n", (unsigned int)pthread_self());
        sleep (2);
        pthread_mutex_unlock (&mutex);
        pthread_exit ((void*)0);
}

int main (int argc, char ** argv)
{
        int L_ret;
        pthread_t tid;
        if (pthread_create (&tid, NULL, sum_thread,NULL) != 0)
        {
                printf("erreur pthread_create"); exit (1);
        }

        L_ret = pthread_mutex_trylock (&mutex);
        if(L_ret == EBUSY)
        {
                printf ("thread principal  return %d : La ressource est verrouillee \n",L_ret  );
        }
        else
        {
                printf ("thread principal : j'utilise la ressource \n");
                pthread_mutex_unlock (&mutex);
        }

        pthread_join (tid, NULL);
        return EXIT_SUCCESS;
}
```

4.3. Exercice

Modifiez l'exercice 1 du paragraphe 3.7 en faisant en sorte qu'un thread qui a commencé à afficher ses n caractères ne soit pas préempté par l'autre .

5. LES CONDITIONS

Une condition peut être utilisée par un thread quand il veut attendre qu'un événement survienne.

- Un thread se met en attente d'une condition (opération bloquante). Lorsque la condition est réalisée par un autre thread, celui-ci le signale au thread en attente qui se réveillera.
- La gestion d'une condition demande la déclaration d'une variable de type *condition* et d'une variable du type *mutex (*pour assurer la protection des opérations sur la variable *condition)*

5.1. Opérations sur les conditions :

- **Création/Initialisation (2 façons) :**
  - **Statique:**
    pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  - **Dynamique:**
    int pthread_cond_init(pthread_cond_t *cond,
    pthread_cond_attr *attr);

    Exemple :
    pthread_cond_t cond_var; /* attributs par défaut */
    pthread_cond_init (&cond_var, NULL);

- **Attente d'une condition**

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

  - **Utilisation:**
    pthread_mutex_lock(&mut_var);
    pthread_cond_wait(&cond_var, &mut_var);
    …..
    pthread_mutex_unlock(&mut_var);

  → Un thread ayant obtenu un *mutex* peut se mettre en attente sur une variable condition associée à ce *mutex*.

  - fonctionnement de pthread_cond_wait:

  → Le mutex spécifié est libéré.
  → Le thread est mis en attente sur la variable de condition *cond.*
  → Lorsque la condition est signalée par un autre thread, le *mutex* est acquis de nouveau par le thread en attente qui reprend alors son exécution.

- Notification :
  - Un thread peut signaler une condition par un appel aux fonctions :
    - **int pthread_cond_signal(pthread_cond_t *cond);**
      → réveil d'un thread en attente sur *cond*.
    - **int pthread_cond_broadcast(pthread_cond_t *cond);**
      → réveil de tous les threads en attente sur *cond*.

  Si aucun thread n'est en attente sur *cond* lors de la notification, cette notification sera perdue.

5.2. Principe général d'utilisation :

Thread B prépare des données qui sont traitées par **Thread A**

Thread principal (main) :

- Déclare et initialise une variables globale pour les données partagée nécessitant une synchronisation
- Déclare et initialise une variables globale indiquant que les données partagées sont prêtes à être traitées (afin de ne pas attendre pour rien des données prêtes)
- Déclare et initialise une condition variable de type condition (pthread_cond_t)
- Déclare et initialise le mutex associé
- Crée les threads A et B qui partagent t les données
- En fin de programme détruit le mutex et la condition

Thread A :

- Travaille jusqu'au point où la condition doit être vraie.
- Verrouille le mutex associé et test la valeur de la variable globale indiquant si le Thread-B à finit son travail sur les données partagées ou non.
  Si données non prêtes
  - Appelle `pthread_cond_wait()` pour réaliser une attente bloquante de la réalisation de la condition. (l'appel à `pthread_cond_wait()` déverrouille automatiquement et atomiquement le mutex associé de façon à ce qu'il puisse être utilisé par le Thread-B).
  - Lorsque la condition est signalée, il se réveille . Le Mutex est automatiquement et atomiquement verrouillé.

  Fin si
- Continue en exploitant les données
- Déverrouille explicitement le mutex

Thread B :

- Travaille
- Verrouille le mutex associé
- Prépare les données
- Positionne la variable indiquant que les données sont prêtes
- Signale la condition
- Déverrouille le mutex.
- Continue

5.3. Exemples

Dans le programme condition1.c ci-dessous, le thread principal crée un thread annexe et se met en attente d'une condition. Le thread annexe verrouille le mutex associé à la condition, travaille puis signale la condition ce qui réveille le thread principal qui attendait cette condition.

```
//*******************************condition1.c*****************************//
#include <errno.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mutex_fin = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_fin = PTHREAD_COND_INITIALIZER;

void *func_thread (void *arg)
{
        unsigned int tid ;
        tid =  pthread_self();
        printf ("\tthread : %u : je travaille\n", tid);
        pthread_mutex_lock (&mutex_fin);
        sleep (5);
        printf ("\tthread : %u : je signale la condition \n",tid);
        pthread_cond_signal (&cond_fin);
        pthread_mutex_unlock (&mutex_fin);
        printf ("\tthread : %u : je me termine \n", tid);
        pthread_exit ((void *)0);
}

int main (int argc, char ** argv)
{
        pthread_t tid;
        pthread_mutex_lock (&mutex_fin);
        printf ("thread principal :  je cree le thread annexe \n");
        if (pthread_create (&tid , NULL, func_thread,NULL) != 0)
        {
                printf("pthread_create erreur\n"); exit (1);
        }
        if (pthread_detach (tid) !=0 )
        {
                printf ("pthread_detach erreur"); exit (1);
        }
        printf ("thread principal :  je me met en attente de la condition \n");
        pthread_cond_wait(&cond_fin,&mutex_fin);
        printf ("thread principal :  je reprend mon travail\n");
        pthread_mutex_unlock (&mutex_fin);
        printf ("thread principal :  je me termine \n");
        return EXIT_SUCCESS;
}
```

5.4. Exercice

5.4.1. Synchronisation

Testez le programme ci dessous et complétez le afin que le calcul ne soit effectué que lorsque les données sont prêtes (tableau de valeur rempli).

```
//*******************************ex_cond1_eleve.c*******************************//

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

#define NBVAL 9
int val[NBVAL];

void tempo(char* s_time)  // pour ralentir les traitements
{
        int i;
        double d=3.0;
        int time;
        time = atoi(s_time);
        time = time *10000;
        for (i=0;i<time;i++)
        {
        d = asin(5.0* sin(i))/5.0;
        }
}

void *func_incremente (void *arg)
{
        int i;
        int time ;
        sleep(1) ; // pour simuler un travail de préparation
        val[0] = 1;
        for (i=1;i<NBVAL;i++)
        {
        val[i]= val[i-1]+1;
        tempo((char*)arg);
        }
        for (i=0;i<NBVAL;i++)
        {
        printf (" %d \n",val[i]);
        }
}

void *func_calcul (void *arg)
{
        int i;
        int time;
        for (i=0;i<NBVAL;i++)
        {
        printf ("       carre %d\n" , val[i]*val[i]);
        tempo((char*)arg);
        }
}
```

```
//*******************************ex_cond1_eleve.c      suite **********************//

int main (int argc, char ** argv)
{
        int i,status;
        int L_ret;
        pthread_t tid_a,tid_b;

        if (argc !=3)
        {
        printf (" Il faut deux parametres entiers \n");
        exit (1);
        }
        if (pthread_create(&tid_a, NULL,func_calcul, argv[2])!= 0)
        {
        printf("erreur pthread_create \n"); exit (1);
        }
        if (pthread_create(&tid_b, NULL,func_incremente, argv[1])!= 0)
        {
        printf("erreur pthread_create \n"); exit (1);
        }
        if (pthread_join (tid_a, (void**) &status) !=0)
        {
        printf ("erreur pthread_join");                 exit (1);
        }
        else
        printf ("Thread %u fini  :\n",(unsigned int)tid_a);
        if (pthread_join (tid_b, (void**) &status) !=0)
        {
        printf ("erreur pthread_join");
        exit (1);
        }
        else
        printf ("Thread %u fini  :\n",(unsigned int)tid_b);

        return EXIT_SUCCESS;
}
```

5.4.2. Rendez vous.

Écrivez un programme qui simule une course :
Le main crée NBTHREAD ″ coureurs ″ et 1 thread starter.

Les coureurs se mettent en attente de la condition start.
Le starter signale cette condition pour réveiller tous les coureurs (pthread_cond_broadcast) vous simulez le temps de course en endormant les coureurs pendant un temps aléatoire (n = rand()%4 +1; et sleep (n) ; en utilisant srand(time(NULL)); dans main pour initialiser le générateur de nombres aléatoires)

Le programme donne un résultat tel que ci dessous :

```
etudiant@Microknoppix:~/PTHREADS/EXE$ ./cond_rendezvous
main : je cree les threads coureurs
        thread coureur: 1 : j'attend le depart
        thread coureur: 2 : j'attend le depart
        thread coureur: 3 : j'attend le depart
main : je cree le thread starter
        thread starter :  a vos marques
        thread starter :  PAN !!!!!
        thread starter :  j'attends la fin de la course
        thread : 1 : je cours
        thread : 2 : je cours
        thread : 3 : je cours
        thread : 2 : je suis arrive apres 2 s
        thread : 1 : je suis arrive apres 3 s
Thread 0 fini avec status :0
Thread 1 fini avec status :0
        thread : 3 : je suis arrive apres 4 s
Thread 2 fini avec status :0
main : fin de la course
etudiant@Microknoppix:~/PTHREADS/EXE$
```

6. FONCTION D'INITIALISATION

Il arrive souvent que des opération d'initialisation doivent être effectuées avant qu'un thread puisse travailler. Chaque thread appelle donc une fonction d'initialisation. Toutefois de travaille d'initialisation ne doit être exécuté qu'une fois. Comme on ne connaît pas à priori l'ordre d'exécution des threads, la solution consistant à ne faire appeler la fonction d'initialisztion à un suel thread ne convient pas.

Pthread offre le mécanisme de **pthread_once**

Soit la fonction d'initialisation fonct_init ne devant être exécutée qu'un fois :

deux étapes :
déclarer un objet **pthread_once**

```
pthread_once_t once_block = PTHREAD_ONCE_INIT;
```

appel de la fonction par les threads :

```
pthread_once(&once_block, func_init);
```

6.1. Exemple 1

l'exemple suivant illustre l'appel à un fonction qui initialise un tableau de valeurs. Deux thread de calculs utilisent ce tableau pour faire respectivement la somme et le produit des données qu'il contient.
L'initialisation ne doit être effectuée qu'une fois par le premier thread de calcul qui est élu.

```
//*******************************fonct_init.c.c*****************************//
#include <errno.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define NB_VAL 10
int tab[NB_VAL];

pthread_mutex_t mutex_data = PTHREAD_MUTEX_INITIALIZER;
pthread_once_t once_block = PTHREAD_ONCE_INIT;

void func_init (void)
{
        int n,i ;
        printf ("thread : fonction initialisation   \n");
        pthread_mutex_lock (&mutex_data);
        printf ("fonct init: je rempli le tableau\n");
        for (i=0;i<NB_VAL;i++)
        {
        tab[i] = i+1;
        printf ("fonct init: tab[%d] = %d \n",i,tab[i]);
        }
        pthread_mutex_unlock (&mutex_data);
        printf ("donnees pretes\n");
}

void *func_calcul_somme (void *arg)
{
        int i;
        int somme=0;
        sleep (1);
        //func_init ();
        pthread_once(&once_block, func_init);
        printf ("\tje fait la somme \n");
        pthread_mutex_lock (&mutex_data);
        for (i=0;i<NB_VAL;i++)
        {
        somme += tab[i];
        printf ("\tsomme = %d\n",somme);
        }
        printf ("\t somme OK \n");
        pthread_mutex_unlock (&mutex_data);
        pthread_exit ((void *)somme);
}
```

```
void *func_calcul_produit (void *arg)
{
        int i;
        int produit=1;
        //func_init ();
        pthread_once(&once_block, func_init);
        printf ("\t\tje fait le produit \n");
        pthread_mutex_lock (&mutex_data);
        for (i=0;i<NB_VAL;i++)
        {
        produit *= tab[i];
        printf ("\t\tfonct produit : tab[%d] = %d produit %d  \n",i,tab[i],produit);
        }
        pthread_mutex_unlock (&mutex_data);
        printf ("\t\tproduit OK \n");
        pthread_exit ((void *)produit);
}
int main (int argc, char ** argv)
{
        int i,status;
        pthread_t tid_somme,tid_produit;
        printf ("main : je cree les threads de calcul \n");

        if (pthread_create (&tid_somme, NULL,func_calcul_somme,NULL) != 0)
        {               printf("erreur pthread_create somme\n"); exit (1);   }
        if (pthread_create (&tid_produit, NULL,func_calcul_produit,NULL) != 0)
        {               printf("erreur pthread_create produit\n"); exit (1);   }

        if (pthread_join (tid_somme, (void**) &status) !=0)
        {               printf ("erreur pthread_join"); exit (1);       }
        else    {               printf ("\tsomme : %d ",status);    }
        if (pthread_join (tid_produit, (void**) &status) !=0)
        {               printf ("erreur pthread_join"); exit (1);       }
        else    {               printf ("\tproduit : %d ",status);    }
        printf ("main : fin \n");
        return EXIT_SUCCESS;
`
```

7. LES SIGNAUX

Les signaux offre un outils de synchronisation pratique et puissant.

Pthread reprend les notions générales degestion des signaux vu dans le cours IPC concernant les ensembles de signaux masqués et pendant . De même la structure ***sigaction*** et la fonction de même nom permettent de configurer la réaction de vos threads à la réception de signaux.

Fonction d'usage générale

int sigemptyset (sigset_t **set*);
int sigfillset (sigset_t **set*);
int sigaddset (sigset_t **set*, int *signum*);
int sigdelset (sigset_t **set*, int *signum*);
int sigismember (const sigset_t **set*, int *signum*);

Fonctions spécifiques à pthread

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

Change le masque des signaux pour le thread appelant tel que décrit par les arguments *how* et *newmask*. Si *oldmask* ne vaut pas **NULL**, le précédent masque de signaux est sauvegardé dans l'emplacement pointé par *oldmask*.

La signification des arguments *how* et *newmask* est la même que pour **sigprocmask** .

- Si *how* vaut **SIG_SETMASK**, le masque de signal est positionné à *newmask*.
- Si *how* vaut **SIG_BLOCK**, les signaux indiqués par *newmask* sont ajoutés au masque de signaux courant (mask|=newmask)
- Si *how* vaut **SIG_UNBLOCK**, les signaux indiqués par *newmask* sont retirés du masque courrant [Ndt : mask&= !newmask].

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

Envoie le signal numéro *signo* au thread *thread*.

Le signal est reçu et géré tel que décrit dans **kill** .

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

Suspend le thread jusqu'à ce que l'un des signaux définis dans *set* soit envoyé . Le numéro du signal reçu est alors sauvegardé dans l'emplacement pointé par *sig* et la fonction rend la main. Les signaux définis dans *set* doivent être bloqués et non ignorés lorsque l'on entre dans **sigwait**. Si l'un des signaux reçus possède un gestionnaire de signal, cette fonction *n'est pas* appellée.

Si plus d'un threads utilise sigwait() pour attendre le même signal, la fonction se terminera et retournera le numéro de signal dans un seul d'entre eux. Le thread concerné ne peut pas être déterminé.

Voir le man de sigwait pour plus de détails

Il est à noté que :

- Chaque thread possède ses propres ensembles de signaux masqués et pendants
  - Les signaux pendants ne sont pas hérités.
  - Le masque d'une thread est hérité à sa création du masque du thread le créant
- Les masques de signaux sont définis par thread, mais la gestion des signaux et les gestionnaires associés, tels que mis en place par **sigaction**, sont partagés par tous les threads!

7.1. Exemple 1 :

Le main crée trois threads envoie le signal SIGALARM

```
/*************************************signal1.c*********************************/

#include <pthread.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>

#define NBR_THREAD 3

void *threadfunc(void *parm)
{
        int   ret;
        pthread_t    tid;
        tid = pthread_self();

         printf("\t\tJe suis le Thread 0x%.8x \n", tid);
         ret = sleep(8);
          if (ret != 0 )
          {
                printf("\t\tThread 0x%.8x : j'ai recu un signal au bout de  %d s\n",tid,8-ret );
                 return NULL;
          }
          printf("\t\tThread 0x%.8x  pas de signal recu dans les temps imparti\n",tid);
          return NULL;
 }

void sighand(int num_sig)
{
        pthread_t   tid = pthread_self();
         printf("\tThread 0x%.8x  je suis dans le handler de signal  %d \n",tid,num_sig);
         return;
}


int main(int argc, char **argv)
{
        int ret;
        int i;
        struct sigaction actions;
        pthread_t   threads[NBR_THREAD];

         printf("Main initialise le handler de signal \n");
        memset(&actions, 0, sizeof(actions));
        sigemptyset(&actions.sa_mask);
        actions.sa_flags = 0;
        actions.sa_handler = sighand;

         sigaction(SIGALRM,&actions,NULL);
```

signal1.c : suite

```
        for(i=0; i<NBR_THREAD; ++i)
        {
                pthread_create(&threads[i], NULL, threadfunc, NULL);
        }
        for(i=0; i<NBR_THREAD; ++i)
        {
                sleep(3);
                printf("Main  : j'envoie le sgnal SIGALRM = n° %d\n",SIGALRM);
                pthread_kill(threads[i], SIGALRM);
        }
        for(i=0; i<NBR_THREAD; ++i)
        {
                pthread_join(threads[i], NULL);
        }
        printf("Main termine\n");
        return 0;
}
```

7.2. Exemple 2 :

Le main crée trois threads détachés qui attendent le signal SIGALRM avec sigwait().
Au bout de 20 secondes, main termine les trois threads (pthread_cancel).
Lancer le programme et dans une console et envoyez sous shell le signal SIGALRM n°14 à votre process (voir les commandes ps -e et kill)
Analysez le résultat

```
/*****************************************sigwait1.c**********************************/
#include <pthread.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>

#define NBR_THREAD 3

void *threadfunc(void *parm)
{
 int   ret;
  pthread_t    tid;
  int numsig_recu;
  sigset_t newmask,signaux_attendu;
  tid = pthread_self();

  printf("\t\tJe suis le Thread 0x%.8x \n", tid);
  sigemptyset(&newmask);
  sigaddset(&newmask,SIGALRM);
  pthread_sigmask(SIG_BLOCK, &newmask, NULL);
  sigemptyset(&signaux_attendu);
  sigaddset(&signaux_attendu,SIGALRM);
   printf("\t\tJe suis le Thread 0x%.8x : j'attend le signal SIGALRM\n", tid);
           sigwait(&signaux_attendu, &numsig_recu);
   printf("\t\tJe suis le Thread 0x%.8x  j'ai recu le signal %d \n", tid,numsig_recu);

   pthread_exit ((void*)0);
}
```

suite sigwait.c

```
int main(int argc, char **argv)
{
  int ret;
  int i;
  sigset_t newmask;
  pthread_t   threads[NBR_THREAD];

  sigemptyset(&newmask);
  sigaddset(&newmask,SIGALRM);
  pthread_sigmask(SIG_SETMASK, &newmask, NULL);

  for(i=0; i<NBR_THREAD; ++i)
  {
    pthread_create(&threads[i], NULL, threadfunc, NULL);
  }
  for(i=0; i<NBR_THREAD; ++i)
  {
    pthread_detach(threads[i]);
  }

   sleep(20);
   printf ("code de retour pthread_cancel ESRCH = %d\n",ESRCH);
   for(i=0; i<NBR_THREAD; ++i)
   {
   ret =  pthread_cancel(threads[i]);
   printf (" cancel thread %d : ret = %d\n",i,ret);
   }
  printf("Main termine\n");
  return 0;
}
```

8. LES SÉMAPHORES

Ils permettent de limiter l'accès à une section critique.
Un sémaphore est caractérisé par
Une valeur entière
Une primitive P(S)
Une primitive V(S)

Les sémaphore sont implémentés dans pthread par des variables de type *sem_t*
#include <semaphore.h>

8.1. Création / Destruction :

int sem_init (sem_t *_sem_, int _partage_, unsigned int _valeur_);

- *Partage* : si valeur nulle, le sémaphore n'est partagé que par les Threads du même processus
- *Valeur* : valeur initiale du sémaphore
- *Valeur* inscrite dans un compteur qui est décrémenté à chaque foisqu'un thread rentre en section critique et incrémenté à chaque sortie.

int sem_destroy (sem_t *sem);

8.2. Entrée sortie en section critique :

int sem_wait (sem_t *sem);
Entrée en SC. Fonction bloquante.
SI la valeur du sémaphore >0
décrémente la valeur
entre en section critique
Si non
thread bloqué jusqu'à ce que la valeur redevienne >0

int sem_post (sem_t *sem);
Sortie de SC. Valeur incrémentée ; un thread en attente est réveillé.

int sem_trywait (sem_t *sem);
Fonctionnement analogue à *sem_wait* mais non bloquante.

8.3. Consulter la valeur d'un sémaphore

int sem_getvalue (sem_t *sem, int *valeur);
Renvoie la valeur du sémaphore

8.4. Exemple 1

```
//*********************************sem1.c*********************************//
// compiler avec l'option -lrt pour clock_gettime
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#define NUM_THREADS 4
sem_t sem;
void *func_thread (void *arg)
{
        unsigned int time;
        unsigned int num_thread;
        int val_sem ;
        num_thread = (unsigned int)arg;
        sem_getvalue (&sem, &val_sem);
        printf ("Thread %u val sem = %d  P(S)\n",num_thread,val_sem);
        sem_wait (&sem);
                sem_getvalue (&sem, &val_sem);
                printf ("Thread %u en Section critique  val sem = %d \n",num_thread,val_sem);
                time = rand()%4+1;
                sleep (time);
                printf ("Thread %u V(S)\n",num_thread);
        sem_post(&sem);
        printf ("Thread %u est sorti de la Section critique  val sem = %d \n",num_thread,val_sem);
        pthread_exit ( (void*)0);
}
```

Suite sem1.c

```
int main (int argc, char ** argv)
{
        int i;
        pthread_t tid [NUM_THREADS];
        srand(time(NULL));
        sem_init (&sem,0,2);
        for (i=0; i < NUM_THREADS; i++)
        {
                if (pthread_create (&(tid[i]), NULL,func_thread, (void*)i+1) != 0)
                {
                        printf ("erreur pthread_create");
                        exit (1);
                }
        }
        for (i=0; i < NUM_THREADS; i++)
        {
                if (pthread_join (tid[i], NULL) !=0)
                {
                        printf ("erreur pthread_join"); exit (1);
                }
        }
        sem_destroy (&sem);
        return 0;
}
```

8.5. producteur consommateur

À coder ..

9. STOCKAGE SPÉCIFIQUE À UN THREAD .

Une zone de stockage spécifique est utilisée vos voulez définir des zones de stockage ″ privé ″ à un thread. Toutes les fonction du thread accède à la même zone.
Chaque thread détient sa propre zone de stockage
Une zone de stockage spécifique est associé à une fonction ‘destructeur’ qui est appelé lorsque le thread à fini d’utiliser la zone de stockage spécifique.
Une zone de données privées est associée à une clé .

9.1. Création de la clé

int pthread_key_create (pthread_key_t *cle,(void*) destruc (void*));

Si *destruc* égal à NULL, l'emplacement créé n'est pas supprimé à la terminaison du *thread*.

Sinon, la fonction spécifiée dans *destruc* est appelée

9.2. Consulter une donnée privée :

void * pthread_getspecific(pthread_key_t cle);

9.3. Modifier une donnée privée :

int pthread_setspecific(pthread_key_t cle, void *valeur);

Valeur est typiquement l'adresse d'une zone allouée dynamiquement.

Dans l'exemple suivant, chaque thread possède son propre buffer de caractère dans lequel il concatène des chaînes.
Une fonction 'destructeur' libère la mémoire alloué à la fin des threads.

9.4. Exemple 1

```
/****************************************key1.c*****************************/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

/* Cle identifiant les donnees specifiques */
static pthread_key_t str_key;
/* "Once" variable assurant que la fonction d'allocation de la clé ne sera appelée qu'une fois */
static pthread_once_t str_alloc_key_once = PTHREAD_ONCE_INIT;
/* fonction qui alloue la clé appellée une seule fois*/
static void str_alloc_key (void);
/* fonction 'destructeur' appelée en fin de thread */
static void detruire_buffer (void *buffer);

static char * concatene_chaine (const char *s)
{
        char *buffer;
        pthread_once (&str_alloc_key_once, str_alloc_key);
        buffer = (char *) pthread_getspecific (str_key);
         if (buffer == NULL)
        {
                 buffer = malloc (1024);
                 if (buffer == NULL)
                         return NULL;
                buffer[0] = 0;
                pthread_setspecific (str_key, (void *) buffer);
                 printf ("Thread %x: buffer alloue a l'adresse  %p\n", pthread_self (), buffer);
        }
        strcat (buffer,s);
        sleep(1);
        return buffer;
}
```

suite key1.c

```
static void str_alloc_key (void)
{
  pthread_key_create (&str_key, detruire_buffer);
  printf ("Thread %x: cle %d allouee \n", pthread_self (), str_key);
}

static void detruire_buffer (void *buffer)
{
  printf ("Thread %x: libere le buffer a l'adresse  %p\n", pthread_self (), buffer);
  free (buffer);
}

static void *process (void *arg)
{
  char *texte;
  texte = concatene_chaine ("\tResultat du ");
  texte = concatene_chaine ((char *) arg);
  texte = concatene_chaine (" thread");
  printf ("Thread %x: \"%s\"\n", pthread_self (), texte);
  return NULL;
}

int main (int argc, char **argv)
{
  char *texte;
  pthread_t th1, th2;
            printf ("Id thread main %x\n",  pthread_self ());
  texte = concatene_chaine (" ceci est la chaine ");
  pthread_create (&th1, NULL, process, (void *) "premier");
  pthread_create (&th2, NULL, process, (void *) "deuxieme");
  texte = concatene_chaine ("de main");
  printf ("main \"%s\"\n", texte);
  pthread_join (th1, NULL);
  pthread_join (th2, NULL);
  return 0;
}
```