GESTION DES THREADS

PLAN

1. Notion de thread
2. Manipulation
3. Programmation
4. Synchronisation

NOTION DE THREAD

- Thread =
  - *processus léger*
  - *Fil d'exécution*
  - *tâche au sein d'un processus*

- Processus habituel : un seul thread

- Principe

  décomposer le programme en plusieurs tâches

1 NOTION DE THREAD

Principe

diviser le processus en plusieurs tâches

Exécution multithreads:

- **Multiples exécutions séquentielles de différentes parties d'un processus**

Processus

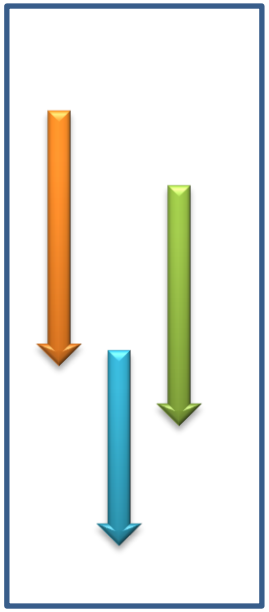
1 NOTION DE THREAD

Principe

diviser le processus en plusieurs tâches

Exécution multithreads:

- **Multiples exécutions séquentielles de différentes parties d'un processus**
  - ➢ **Exécution en parallèle**

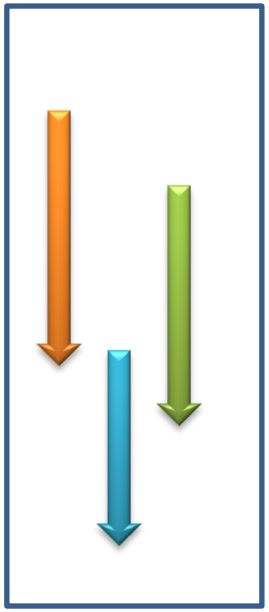


Processus

1 NOTION DE THREAD

Principe

diviser le processus en plusieurs tâches



Exécution multithreads:

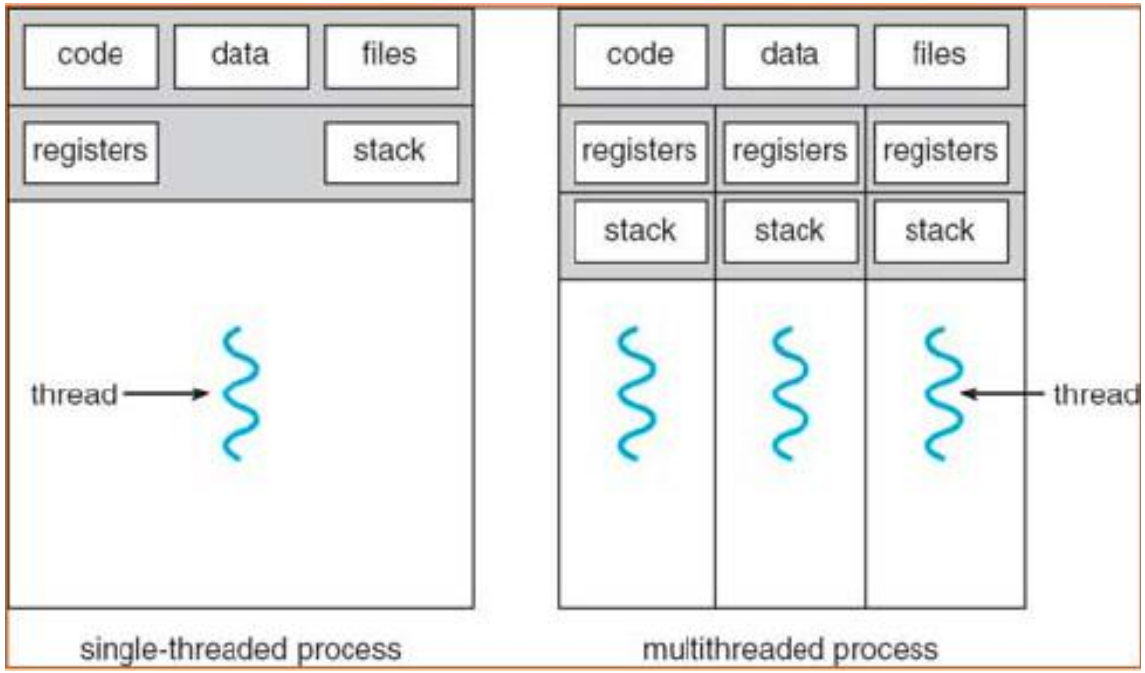
- **Multiples exécutions séquentielles de différentes parties d'un processus**
  - ➢ **Exécution en parallèle**
- **Tous les threads partagent le même espace d'adressage (virtuel) du processus**
  - ➢ **Pas de protection ou séparation mémoire entre threads!!**

Processus

1 NOTION DE THREAD

Les ressources d'un processus sont partagées par tous ses threads :

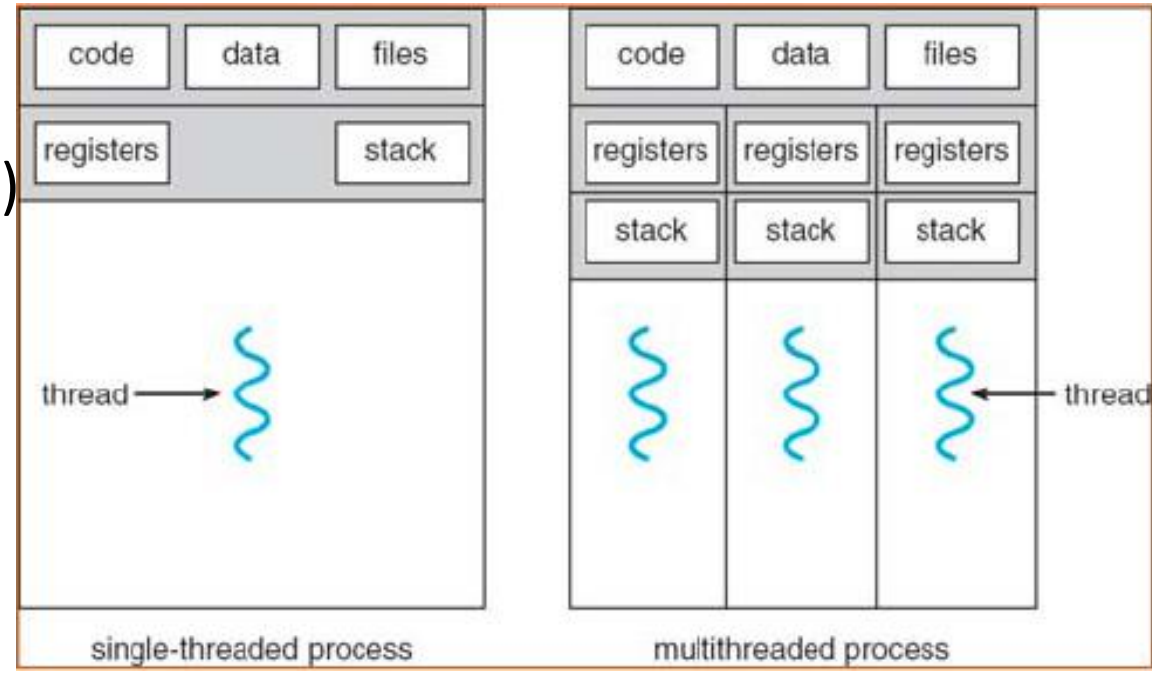
- Code
- Variables globales
- Fichiers ouverts
- Signaux
- Droits Unix
- Environnement de shell
- Répertoire de travail



1 NOTION DE THREAD

Un thread possède

- Identificateur de thread
- Sa propre pile d'exécution
- Pointeur d'instruction(C.O)
- Ses propres registres



NOTION DE THREAD

Données communes entre les threads

- Contenu de la mémoire (programme, tas, état des entrées/sorties)

Données privées

- Données de la pile
- Valeurs des registres
- Informations sur l'ordonnancement du thread
- Gardées dans le Thread Control Block (TCB)

Deux composantes d'un processus:

- Mémoire virtuelle (protection)
- Threads (pour exécution parallèle)

NOTION DE THREAD

Applications de la programmation multi-threadée

Systèmes embarqués

- Avions, ascenseurs, systèmes médicaux, systèmes audio/vidéo, . . .
- Un seul programme, opérations concurrentes

Systèmes d'exploitations modernes

- Opérations parallèles entre les utilisateurs
- Pas de protection nécessaire dans le noyau

Serveurs de base de données

- Accès concurrents (en parallèles) à la base
- Activités en background de maintenance de la base

Serveurs réseau

- Multiples requêtes du réseau
- Serveur de fichier, serveur web, réservation de billets,...

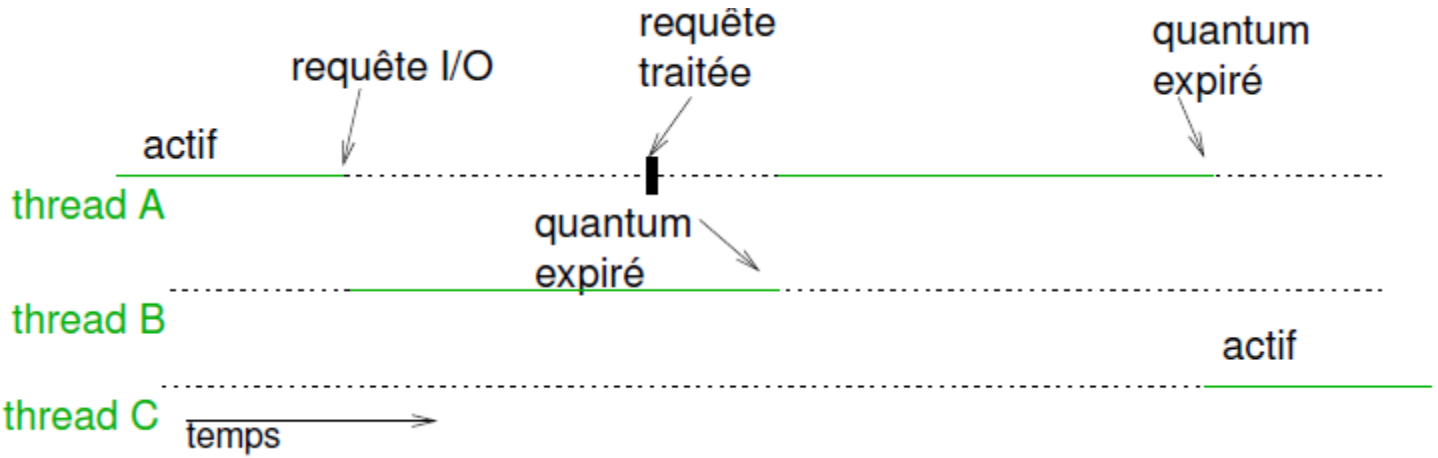
NOTION DE THREAD

- **Ordonnancement**
  - ordonnancement au sein du processus
  - Typiquement, un autre thread prend l'exécution quand :
    - le thread courant se bloque (E/S), ou
    - le thread courant a épuisé son quantum de temps

NOTION DE THREAD

- **Ordonnancement**
  - ordonnancement au sein du processus
  - Typiquement, un autre thread prend l'exécution quand :
    - le thread courant se bloque (E/S), ou
    - le thread courant a épuisé son quantum de temps

Sur une architecture monoprocesseur/monocœur :



NOTION DE THREAD

- **Ordonnancement**

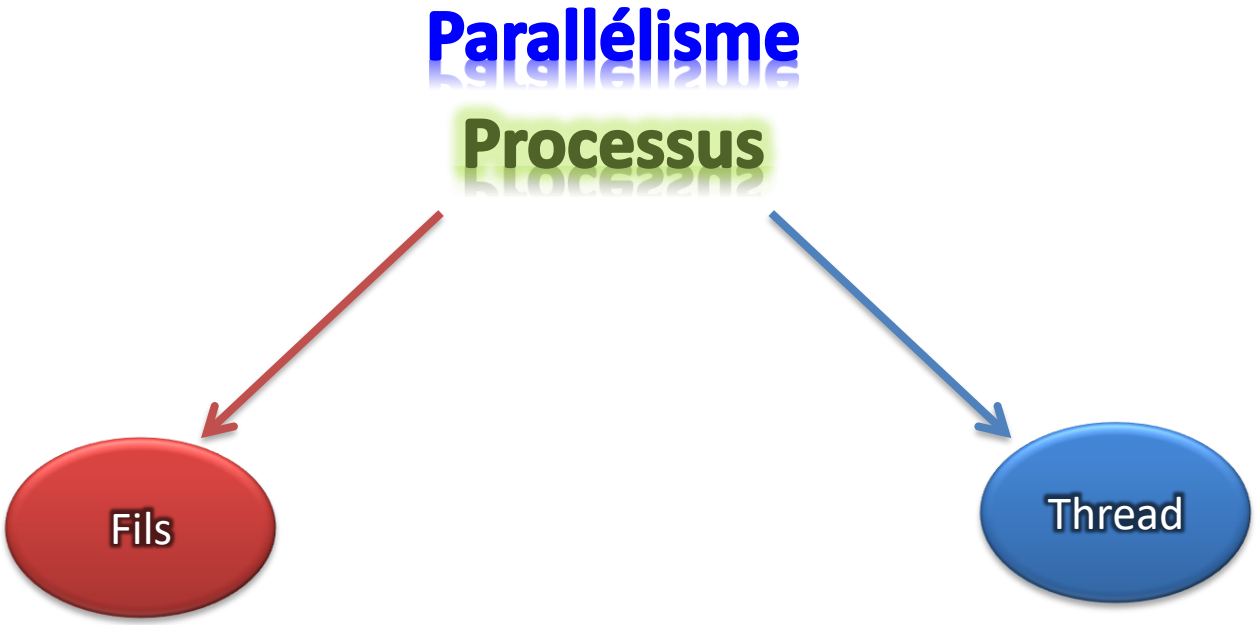
Sur une architecture multiprocesseurs/multicœurs :

possibilité de plusieurs threads **parallèles**

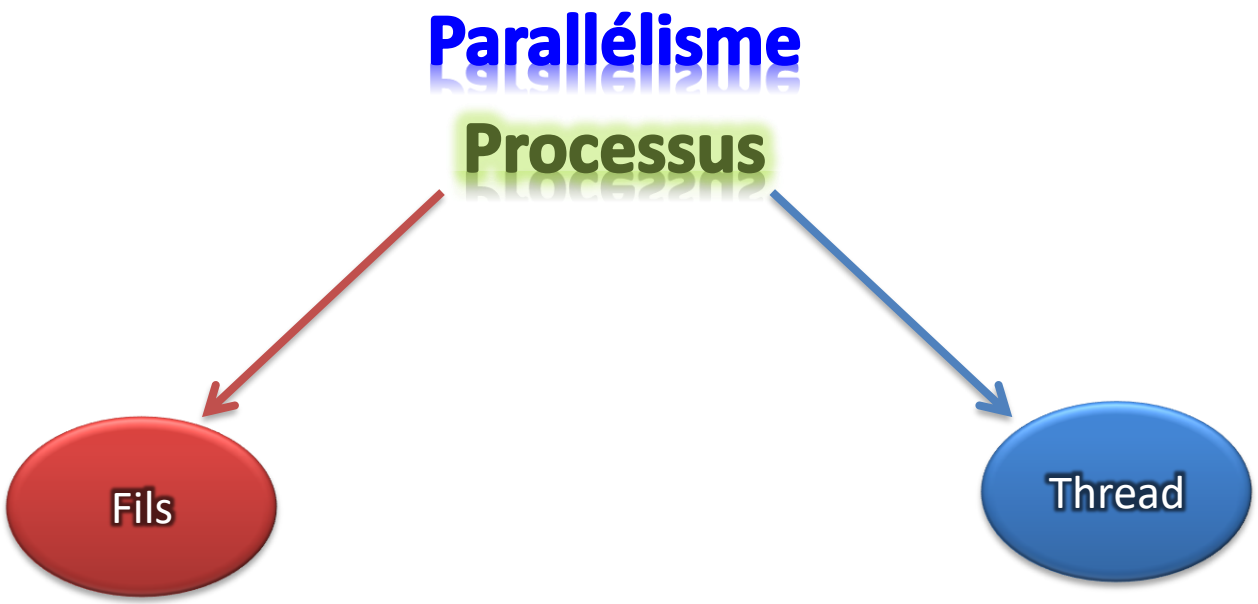
- Expression du parallélisme pour des cœurs qui partagent la mémoire
- Pour du calcul de hautes performances, on a parfois 1 cœur = 1 thread
- Les threads facilitent la programmation/communication entre les cœurs

1 NOTION DE THREAD

Ainsi on peut atteindre le :



1 NOTION DE THREAD



- Consommation de plusieurs ressources
- Communication par des mécanismes

- Moins de ressources consommées
- Communication facile à travers la mémoire commune

MANIPULATION

- pthread : threads POSIX
  - POSIX : IEEE Portable Operating System Interface, X as in UniX
  - #include <pthread.h>

MANIPULATION

- **Création**

int **pthread_create**(threadid, attr, fonc, arg);

2 MANIPULATION

- **Création**

int **pthread_create**(threadid, attr, fonc, arg);

identificateur de thread crée

MANIPULATION

- **Création**

int **pthread_create**(threadid, attr, fonc, arg);

identificateur de thread crée

la valeur d'un thread peut être obtenue au sein du thread lui-même par pthread_self()

MANIPULATION

- **Création**

int **pthread_create**(threadid, attr, fonc, arg);

attributs du thread (défaut : NULL)

- Type d'ordonnancement
- Priorité (héritée par le processus initial)
- Taille de la pile (par défaut 61440)

2 MANIPULATION

- **Création**

int **pthread_create**(threadid, attr, fonc, arg);

attributs du thread (défaut : NULL)

- Modification par :
  - pthread_attr_setsched()
  - pthread_attr_setprio ()
  - pthread_attr_setstacksize ()

MANIPULATION

- **Création**

int **pthread_create**(threadid, attr, fonc, arg);

@fonction à exécuter par le thread

MANIPULATION

- **Création**

int **pthread_create**(threadid, attr, fonc, arg);

Liste des arguments de la fonction

- **Terminaison du thread courant**

int **pthread_exit**(void * retval);

- Termine l'exécution du thread
- La valeur ***retval*** (valeur arbitraire) est la valeur de retour du thread
- Appelée implicitement si le thread termine normalement
- Ne rend jamais la main
- Ne libère pas les ressources du thread (Voir pthread_join() ou pthread_detach()

MANIPULATION

- **Récupération**

int **pthread_join**(pthread_t threadid, void * * retour);

Valeur de retour lors de la terminaison du thread

- **Récupération**

int **pthread_join**(pthread_t threadid, void * * retour);

Valeur de retour lors de la terminaison du thread

Attention à l'allocation de cette valeur

- **Récupération**

int **pthread_join**(pthread_t threadid, void * * retour);

- Suspend l'exécution du thread appelant jusqu'à ce que le thread ***threadid*** termine.
- ****retour*** est égal à la valeur de ***retval*** donnée par le thread ***threadid*** lors de l'appel à pthread_exit()
- ****retour*** vaut PTHREAD_CANCELED si le thread ***threadid*** a été annulé
- Le thread ***threadid*** ne doit pas avoir été détaché
- Libère les ressources du thread ayant terminé

MANIPULATION

- **Destruction**

int **pthread_detach**(pthread_t threadid);

- Libère l'ensemble de ressources allouées à la fin du thread au processus initial

MANIPULATION

- **Annuler un thread depuis un autre programme**

int **pthread_cancel**(pthread_t threadid);

PROGRAMMATION

- **Compilation**

gcc prog.c – lpthread

3 MANIPULATION

- **Exemple**

```
#include <stdio.h>
#include <stdlib.h>
void * start(void * ptr)
        { int x = *(int *) ptr ;
        printf("[thread id=%d ]  ***  New thread has x =%d \n", pthread_self(), x) ;
        pthread_exit(NULL);
        }
int main()
{
        int i; int  x=3;
        pthread_t tid[5];
        void * ptr = (void *)& x;
        for(i =0; i<5;i++)
                pthread_create(&tid[i], NULL, start, ptr ) ;
        printf( "Main thread is running \n" ) ;
        for(i =0; i<5;i++)
                pthread_join(&tid[i], NULL) ;
        return EXIT_SUCCESS;
}
```

MANIPULATION

- **Exemple**

```
#include <stdio.h>
#include <stdlib.h>
void * start(void * ptr)
        { int x = *(int *) ptr ;
        printf("[thread id=%d ]  ***  New thread has x =%d \n", pthread_self(), x) ;
        pthread_exit(NULL);
        }
int main()
{
        int i; int  x=3;
        pthread_t tid[5];
        void * ptr = (void *)& x;
        for(i =0; i<5;i++)
                pthread_create(&tid[i], NULL, start, ptr ) ;
        printf( "Main thread is running \n" ) ;
        for(i =0; i<5;i++)
                pthread_join(&tid[i], NULL) ;
        return EXIT_SUCCESS;
}
```

```
[thread id=-1217299600 ]  ***  New thread has x =3
[thread id=-1228932240 ]  ***  New thread has x =3
[thread id=-1239422096 ]  ***  New thread has x =3
[thread id=-1249911952 ]  ***  New thread has x =3
[thread id=-1260401808 ]  ***  New thread has x =3
Main thread is running
```

SYNCHRONISATION

- Accès en exclusion mutuelle à une section critique
- protection de la section critique par :

  – un mutex (un verrou)

- Utilisation

```
lock(mutex)
        ... /* section critique */
unlock(mutex)
```

SYNCHRONISATION - MUTEX

- Création

```
pthread_mutex_t SB;
```

- Destruction

```
int pthread_mutex_destroy( &SB);
```

Libère les ressources allouées pour le mutex

SYNCHRONISATION - MUTEX

- initialisation

```
int pthread_mutex_init ( &SB, &att);
```

Création d'un nouveau mutex. Si ***attr*** est NULL, attributs par défaut pour le mutex

- Initialisation rapide

```
pthread_mutex_t  SB= THREAD_MUTEX_INITIALIZER;
```

SYNCHRONISATION - MUTEX

- Verrouillage

```
int pthread_mutex_lock ( &SB);
```

– Si le mutex est déjà verrouillé, cet appel bloque le thread appelant (l'endort).
– Renvoie -1 si erreur:
  - EDEADLK: un deadlock va arriver si le mutex est bloqué à cause de ce mutex.

- Déverrouillage

```
int pthread_mutex_unlock (&SB);
```

Réalisé par le propriétaire du mutex uniquement

SYNCHRONISATION - MUTEX

- Tentative de verrouillage

```
int pthread_mutex_trylock ( &SB);
```

- ➢ succès si le SB est libre (retourne 0), et le thread appelant devient propriétaire
- ➢ sinon retour de EBUSY

SYNCHRONISATION - MUTEX

- Exemple

```
/* Thread 1 */

pthread_mutex_lock(&m);
      v = v-1;
pthread_mutex_unlock(&m);
```

```
/* Thread 2 */

pthread_mutex_lock(&m);
    v = v*2;
pthread_mutex_unlock(&m);
```

4 SYNCHRONISATION - MONITEUR

- Conditions

- Attente d'un état (exp : tampon non vide, tampon non plein…)

- utilisée avec un mutex

SYNCHRONISATION - MONITEUR

- Conditions

- Création

```
pthread_cond_t cd;
```

- Destruction

```
pthread_cond_destroy (&cd);
```

SYNCHRONISATION - MONITEUR

- Conditions

- initialisation

```
int pthread_cond_init ( &cd, &att);
```

- Initialisation rapide

```
pthread_cond_t  cd= THREAD_COND_INITIALIZER
```

SYNCHRONISATION - MONITEUR

- Conditions

- Attente sur une condition

```
int pthread_cond_wait( &cd, &SB);
```

1. libère le SB associé avant de bloquer le thread
2. bloque le thread jusqu'à ce que la condition soit signalée
3. verrouille le SB avant de retourner suite à un signal sur la condition

SYNCHRONISATION - MONITEUR

- Conditions

- Réveiller un thread

int pthread_cond_signal(& cd);

– réveille (au moins) un thread en attente sur la condition

- Réveiller tous les threads

int pthread_cond_broadcast(&cd);

– réveille tous les threads en attente sur la condition

SYNCHRONISATION - MONITEUR

- Exemple

Exemple : Gestion de nombre de places libres dans une salle

- L'allocation de n places se fait avec la fonction **allouer** (n)
- La libération de m places se fait avec la fonction **liberer** (m)
- La variable partagée est le nombre de places libres qui est donné par **nlibre**

SYNCHRONISATION - MONITEUR

- Exemple

```
void allouer (int n) {

    if   (n > nlibre) {

  }
  nlibre = nlibre – n ;

}
```

```
void liberer (int m) {

  nlibre = nlibre + m ;


}
```

Exemple : Gestion de nombre de places libres dans une salle

- L'allocation de n places se fait avec la fonction **allouer** (n)
- La libération de m places se fait avec la fonction **liberer** (m)
- La variable partagée est le nombre de places libres qui est donné par **nlibre**

4 SYNCHRONISATION - MONITEUR

- Exemple

```
void allouer (int n) {
  pthread_mutex_lock(&mutex);
    if   (n > nlibre) {

  }
  nlibre = nlibre - n ;
  pthread_mutex_unlock (&mutex) ;
}
```

```
void liberer (int m) {
  pthread_mutex_lock (&mutex) ;
  nlibre = nlibre + m ;

  pthread_mutex_unlock (&mutex) ;
}
```

La protection de la section critique se fait avec :

- **pthread_mutex_lock** : verrouillage de la section critique
- **pthread_mutex_unlock** : déverrouillage de la section critique

SYNCHRONISATION - MONITEUR

- Exemple

```
void allouer (int n) {
  pthread_mutex_lock(&mutex);
    if   (n > nlibre) {
    pthread_cond_wait (&c, &mutex) ;
  }
  nlibre = nlibre – n ;
  pthread_mutex_unlock (&mutex) ;
}
```

```
void liberer (int m) {
  pthread_mutex_lock (&mutex) ;
  nlibre = nlibre + m ;

  pthread_mutex_unlock (&mutex) ;
}
```

Si le nombre de places disponibles n'est pas suffisant alors le thread exécutant la fonction allouer est bloqué et il est demandé de libérer la section critique pour ne pas créer des interblocages

SYNCHRONISATION - MONITEUR

- Exemple

```
void allouer (int n) {
  pthread_mutex_lock(&mutex);
    if   (n > nlibre) {
    pthread_cond_wait (&c, &mutex) ;
  }
  nlibre = nlibre – n ;
  pthread_mutex_unlock (&mutex) ;
}
```

```
void liberer (int m) {
  pthread_mutex_lock (&mutex) ;
  nlibre = nlibre + m ;
  pthread_cond_broadcast (&c) ;
  pthread_mutex_unlock (&mutex) ;
}
```

Une fois que le thread exécutant liberer a mis à jour la valeur de nlibre alors il est demandé de **diffuser** cette information et ce en réveillant tous les threads qui ont été bloqués sur allouer puisqu'il n'est pas sûr que la demande de la tête de la file peut être honorée suite à la libération des places

SYNCHRONISATION - MONITEUR

- Exemple

```
void allouer (int n) {
  pthread_mutex_lock(&mutex);
  while (n > nlibre) {
    pthread_cond_wait (&c, &mutex) ;
  }
  nlibre = nlibre - n ;
  pthread_mutex_unlock (&mutex) ;
}
```

```
void liberer (int m) {
  pthread_mutex_lock (&mutex) ;
  nlibre = nlibre + m ;
  pthread_cond_broadcast (&c) ;
  pthread_mutex_unlock (&mutex) ;
}
```

En recevant le signal de déblocage, le thread exécutant allouer doit de nouveau retester la possibilité de l'allocation des places

4 SYNCHRONISATION - SÉMAPHORE

- sémaphore (à compteur)

- Sémaphore généralisent les verrous
- Inventés par Dijkstra à la fin des années 1960
- Principale primitive de synchronisation à l'origine dans UNIX
- Définition: un sémaphore est un entier non négatif qui a deux opérations :
  - P(): opération atomique qui attend que le sémaphore soit positif et le décrémente de 1. Proberen (tester en néerlandais) = puis-je ?
  - V(): opération atomique qui incrémente le sémaphore de 1, et réveille les threads/processus bloquant sur P(). Verhogen (incrémente en néerlandais) = vas-y

4 SYNCHRONISATION - SÉMAPHORE

- sémaphore (à compteur)

- Sémaphore =
  - une valeur et une
  - file d'attente

SYNCHRONISATION - SÉMAPHORE

- sémaphore (à compteur)

- Création

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

pshared : sémaphore local au processus (0) ou partagé avec d'autres processus

value : valeur initiale

- Destruction

```
int sem_destroy( sem_t * sem);
```

4 SYNCHRONISATION - SÉMAPHORE

- sémaphore (à compteur)

- Notification d'un sémaphore

  int sem_post(& sem); //V

  incrémente le compteur de manière atomique

  débloque

- Attente sur un sémaphore

  int sem_wait(&sem); //P

  bloque tant que compteur < 0

  décrémente alors le sémaphore de manière atomique

4 SYNCHRONISATION - SÉMAPHORE

- sémaphore (à compteur)

- Tentative de décrémentation

```
int sem_trywait(&sem);
```

variante non-bloquante de sem_wait()

- compteur > 0 : décrémente et retourne 0
- compteur = zéro : retourne immédiatement -1 avec l'erreur EAGAIN

4 SYNCHRONISATION - SÉMAPHORE

- sémaphore (à compteur)

Problème Producteur/consommateur :

- Un producteur écrit des données dans un buffer partagé (taille limitée)
- Un consommateur les enlève du buffer

→ Besoin d'une synchronisation entre le consommateur et le producteur

- Le producteur doit attendre si le buffer est plein
- Le consommateur doit attendre si le buffer est vide

❖ Buffer = ressource partagée modifiée.

→ Besoin section critique

Exemple :

- séquence de pipe : cmd1 | cmd2 | cmd3
- filtres sur des images