2024/2025

Programmation des systèmes

Hamza EL KHOUKHI

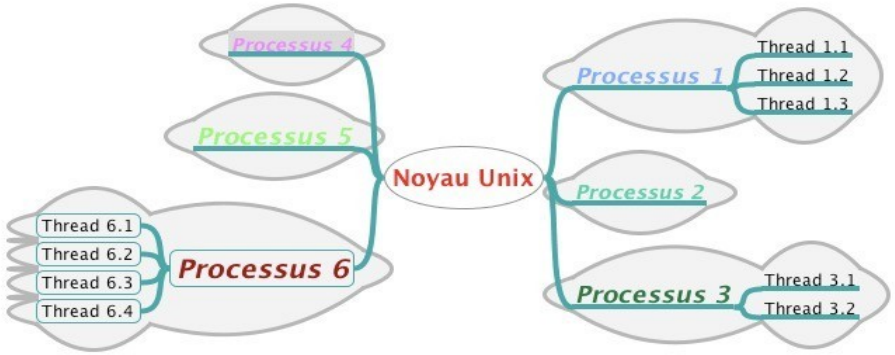
Chapitre 4:

UNIVERSITÉ TOULOUSE Jean Jaurès

Synchronisation des processus (Les threads)

Synchronisation des processus (les threads)

- Un programme se compose notamment de variables globales et de plusieurs sous- programmes dont le sous-programme principal main
- Un processus lourd se compose initialement d'un processus léger exécutant le sous- programme principal et de ressources (ex : mémoire pour variables globales)
- Le processus lourd au travers de son processus léger initial pourra créer d'autres processus légers qui exécuteront un des sous-programmes du programme
- Les processus légers (le processus initial et ceux créés ultérieurement) s'exécutent en parallèle au sein du processus lourd en **partageant les ressources**



Intérêt des threads

- Certaines applications se dupliquent entièrement au cours du traitement :
  - dans les systèmes client-serveur qui utilisent des processus, le serveur exécute un *fork()* pour traiter chacun de ses clients (un processus par client).
  - Cette duplication est souvent très coûteuse...
- Avec des threads, on peut arriver au même résultat sans gaspillage d'espace, en créant un thread par client, *mais en conservant le même espace d'adressage, de code et de données*. Mais, évidemment, si un client perturbe cet espace, cela impactera tous les clients...
- Les threads sont, par ailleurs, très bien adaptés au parallélisme. Ils peuvent s'exécuter simultanément sur des machines multiprocesseurs et multi-cœurs.

Les threads : Exemple simple

f et g mettent à jour deux parties **indépendantes** d'une liste L

```
int l[] = {1, 2, 3, 4};
void f(int *l)…
void g(int *l)…
```

- On peut paralléliser le traitement, soit :
  - en créant deux processus lourds indépendants l doit être alors copiée dans un **fichier** partagé
  - en créant deux processus légers (threads) indépendants, l est rangée en **mémoire** partagée
    - l reste directement accessible sans copie.

Méthode	Avantages	Inconvénients
Processus (fork)	Isolation mémoire, pas d'interférences	Coûteux en ressources, besoin d'IPC (pipes)
Threads (pthread)	Rapide, partage direct des données	Risque de conflits (besoin de mutex si écritures concurrentes)

Les threads : fonctionnalités

- Le module **pthread.h** fournit une interface permettant de gérer les threads, il fournit :
  - La structure **pthread_t** pour représenter un thread.
  - La fonction **pthread_create** qui permet de créer un thread et d'associer une fonction à exécuter.
  - La fonction **pthread_join** qui permet d'attendre la fin d'un thread.
  - La fonction **pthread_self** pour obtenir le thread en cours d'exécution.

Pour compiler un programme des threads :

gcc thread.c –o out -pthread

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

void *fonction_thread(void *arg) {

printf("Thread en cours d'exécution : %lu\n",
pthread_self());
return NULL;

}

int main() {

pthread_t thread1;
pthread_create(&thread1, NULL, fonction_thread, NULL);

pthread_join(thread1, NULL);
return 0;

}
```

Les threads : Terminaison

- Par défaut, un thread se termine lorsqu'il atteint la fin du bloc de sa méthode (ou le corps du programme pour le thread principal).
- Un appel à pthread_exit(NULL) dans un thread termine celui-ci, pas le programme principal (pratique peu conseillée...)

> ➢ lorsque le thread principal se termine, **tous les threads qu'il a lancé se terminent aussi, même s'ils n'ont pas terminé** : n'oubliez jamais d'appeler les méthodes `pthread_join()` des threads « fils » avant de terminer le thread principal...

Les threads : Attendre la fin d'un thread

- La méthode d'instance **pthread_join** permet au thread courant d'attendre la fin d'un autre thread. Elle est notamment très importante dans le thread principal car, lorsque celui-ci se termine, *tous les threads qu'il a créé sont supprimés*.

```
 // Création des threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i + 1;
        pthread_create(&threads[i], NULL, thread_function,
&thread_ids[i]);
    }
// Attendre la fin de tous les threads créés (sauf le main)
for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
}
```

Threads : Problème

- Soit un système de réservation de billets d'avions. Le principe pourrait être :
  - Trouver une place libre;
  - NbPlacesOccupées = NbPlacesOccupées + 1;
- Ce code peut être exécuté par plusieurs agences de voyages simultanément (processus) : NbPlacesOccupées est donc partagée par tous les processus.
- Deux processus peuvent donc lire quasiment en même temps la valeur de NbPlacesOccupées et le résultat de ces deux réservations n'occupera donc qu'une seule place de plus...

Section critique

Accès à des objets partagés

- Deux processus/threads ne doivent jamais pouvoir manipuler simultanément une variable commune : il faut garantir l'accès en **exclusion mutuelle** aux variables partagées.
- On appellera **section critique** un segment de code qui manipule des variables globales.
- Pour que l'accès à ces variables se fasse en **exclusion mutuelle**, on ajoutera du code :
  - Un prologue pour attendre d'entrer en section critique.
  - Un épilogue pour signaler qu'on en est sorti
- Les threads/processus pourront ainsi arbitrer l'accès à la section critique

```
// Prologue ---> Blocage du processus/thread tant que la SC est occupée #
  Section critique
// Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
```

Section critique

Construisons une solution valide

- Contraintes
  - À tout instant, un processus au plus est engagé en SC.
  - Aucune hypothèse ne sera faite sur la vitesse des processus.
  - Tout processus peut être interrompu à tout instant.
  - Pas de famine.
  - Si la SC est libre, tout processus demandant à y entrer doit pouvoir le faire (pas de "tour de rôle").

Section critique

1ère Tentative

- Utilisation d'un booléen LIBRE initialisé à VRAI pour indiquer que la SC est libre ou non
  - LIBRE doit donc être une variable globale, partagée par tous les processus...

```
// Prologue ---> Blocage du processus/thread tant que la SC est
occupée
while (NOT LIBRE)        //  Attente active
LIBRE = FAUX             //  Entrée en SC
// Section critique
SC                       //  SC
// Épilogue ---> Le   processus/thread signale qu'il est sorti la SC
                      de
LIBRE = VRAI             // Sortie de SC
```

1ère Tentative : Analyse

- Si deux processus veulent entrer en même temps en SC et que LIBRE est à VRAI, ils vont y entrer tous les deux en même temps en mettant tous les deux LIBRE à VRAI en ressortant.
  - Supposons que P0 trouve LIBRE à VRAI et qu'une interruption survienne avant qu'il ait eu le temps de la mettre à FAUX.
  - Supposons que cette interruption donne le contrôle à P1 qui, lui aussi, veut entrer en SC : il peut le faire.
  - Quand P0 reprend le contrôle, il reprend ou il s'était arrêté : il met LIBRE à FAUX et entre en SC **Cette solution n'est pas bonne.**

On ne peut pas trouver de solution correcte avec un seul booléen. De plus, le fait que LIBRE soit une variable commune n'a fait que déplacer le problème : maintenant, c'est LIBRE qui est le sujet de l'exclusion mutuelle !

2ème Tentative

- Utilisation d'un entier TOUR donnant le numéro du processus dont c'est le tour de passer.
- On suppose que chaque processus connaît son numéro à l'aide d'une constante MOI (qui vaut 0 ou 1).
- Initialement, TOUR est initialisé à 0.

```
// Prologue ---> Blocage du processus/thread tant que la SC est occupée
while (TOUR != MOI)  // Attente
TOUR = MOI           // active
                        Entrée en SC
// Section critique
SC                   // SC
/* Épilogue ---> Le processus/thread signale qu'il est sorti de la*/
SC TOUR = 1-MOI      // Sortie de SC
```

2ème Tentative : Analyse

- Si P0 et P1 sont tous les deux en SC :
  - Si P0 est entré, c'est qu'il a trouvé TOUR à 0.
  - Si P1 est entré, c'est qu'il a trouvé TOUR à 1.
  - Ce qui est absurde...

- Exclusion mutuelle GAGNEE mais alternance imposée entre P0 et P1

  ☹ Si la SC est libre, tout processus demandant à y entrer doit pouvoir le faire (pas de "tour de rôle"). → Contraintes non respectées

Section critique

3ème Tentative

- On utilise un tableau REQUETE[2] qui désigne les demandes (initialisé à FAUX).

```
// Prologue ---> Blocage du processus/thread tant que la SC est occupée
REQUETE[MOI] = VRAI
while (REQUETE[TOI]) // Attente active
# Section critique
SC                   // SC
// Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
REQUETE [MOI] = FAUX   // Sortie de SC
```

Section critique

3ème Tentative : Analyse

☺Exclusion mutuelle ok

☹Mais Si deux processus positionnent en même temps leur REQUEST[MOI] et qu'ils testent le REQUEST[TOI], chacun boucle indéfiniment. On a donc une **situation d'interblocage**

Algorithme de Peterson (1981)

- Mélange des tentatives précédentes qui utilise comme variables globales
  - Un tableau de booléens (initialisé à FAUX)
  - et un entier

```
# Prologue ---> Blocage du processus/thread tant que la SC est occupée
REQUETE[MOI] = VRAI
TOUR = 1-MOI
while (REQUETE[TOI] et TOUR ==(1-MOI))    # Attente active
# Section critique
SC                    # SC
# Épilogue ---> Le processus/thread signale qu'il est sorti de la SC
REQUETE [MOI] = FAUX   # Sortie de SC
```

Les sémaphores de Dijkstra

- Un sémaphore est un objet abstrait sur lequel on peut réaliser deux opérations atomiques
  - P(Sem) (prolagen (essayer de diminuer))
  - V(Sem) (verhogen (augmenter))
- P est l'opération susceptible de bloquer, V n'est jamais bloquant mais débloque un processus bloqué par P.
- Un sémaphore étant un objet commun à plusieurs processus, toutes les opérations P et V agissant sur un même processus doivent se faire en exclusion mutuelle.
- À un sémaphore S est associé, de façon interne :
  - un compteur cpt, qui est un entier représentant la valeur du sémaphore
  - une file d'attente f, initialement vide.

Les sémaphores : Fonctionnement

- P(Sem)

```
S.cpt = S.cpt - 1;
if (S.cpt < 0) {
   Insérer le processus exécutant P dans S.f ;
   Bloquer ce processus ;
}
```

- V(Sem)

```
S.cpt = S.cpt + 1;
if (S.cpt <= 0) { /* il y a des processus bloqués */
   Extraire le processus en tête de S.f;
   Débloquer ce processus ;
}
```

Les sémaphores : Remarques

- Le nombre de processus bloqués par un sémaphore S est égal à S.cpt
- Un processus bloqué par P(S) devra attendre qu'un autre processus exécute V(S).
- Un sémaphore permet donc de contrôler le nombre de processus autorisés à entrer dans une section critique.
- Un *MUTEX* est un sémaphore initialisé à 1, qui permet donc d'assurer l'exclusion mutuelle.

Les sémaphores

En C, les sémaphores sont gérés avec la bibliothèque **semaphore.h**. Ils permettent d'assurer l'exclusion mutuelle entre threads.

On utilise deux opérations classiques :
P() (Proberen) → sem_wait() en C
V() (Verhogen) → sem_post() en C

Le constructeur prend une valeur initiale pour le compteur. Par défaut, cette valeur est 1 (ce qui convient donc pour un mutex).

```
sem_wait(&semaphore); // P() → Prend le sémaphore (bloque si déjà pris)

// SECTION CRITIQUE
printf("Thread %d entre dans la section critique.\n", id);

sleep(1); // Simulation d'un travail long

printf("Thread %d quitte la section critique.\n", id);

sem_post(&semaphore); // V() → Libère le sémaphore
```

Les sémaphores

Réservation Avion : Solution

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_THREADS 10
#define NUM_INCREMENTS 100000

sem_t mutex; // Déclaration du sémaphore
int nb_places_occupees = 0; // Variable partagée

void *inc(void *arg) {

for (int i = 0; i < NUM_INCREMENTS; i++) {
sem_wait(&mutex); // Verrouillage
nb_places_occupees++; //  Incrémentation protégée
sem_post(&mutex); //  Déverrouillage

}
return NULL;

}
```

Valeur initiale de mutex : 1 (libre).
Quand un thread entre : mutex = 0 (occupé).
Quand un thread sort : mutex = 1 (libéré).

```c
int main() {
pthread_t threads[NUM_THREADS];

sem_init(&mutex, 0, 1); // Initialisation du sémaphore (1 = Mutex)

// Création des threads

for (int i = 0; i < NUM_THREADS; i++) {

pthread_create(&threads[i], NULL, inc, NULL);

}

// Attente de la fin des threads

for (int i = 0; i < NUM_THREADS; i++) {
pthread_join(threads[i], NULL);
}

sem_destroy(&mutex); // Libération des ressources

printf("Nbre de places occupées = %d\n", nb_places_occupees);
return 0;
}
```

Les sémaphores

- Le sémaphore est principalement utilisé pour limiter l'accès à une ressource dont les capacités sont limitées (autoriser un nombre maximum d'accès)
  - Dans la réservation de places d'avions, l'accès à la variable **nb_place_occupees** est limitée à une seule personne -> la classe **pthread_mutex_t** suffit amplement.
- Un sémaphore ou un verrou doivent être vus comme un moyen de contrôler l'accès à une zone de code ce n'est pas un moyen de communiquer entre thread...
- Pour synchroniser plusieurs threads entre eux, il existe d'autres mécanismes : les **variables conditions** ou les **files synchronisées**.

Les sémaphores

Réservation Avion : Solution exemple de pthread_mutex

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_THREADS 10
#define NUM_INCREMENTS 100000

pthread_mutex_t mutex; // Déclaration du mutex
int nb_places_occupees = 0; // Variable partagée

void *inc(void *arg) {

for (int i = 0; i < NUM_INCREMENTS; i++) {

pthread_mutex_lock(&mutex); //  Verrouillage du mutex

nb_places_occupees++; //  Incrémentation protégée

pthread_mutex_unlock(&mutex); //  Déverrouillage du
mutex
}
return NULL; }
```

Valeur initiale de mutex : 1 (libre).
Quand un thread entre : mutex = 0 (occupé).
Quand un thread sort : mutex = 1 (libéré).

```c
int main() {
pthread_t threads[NUM_THREADS];

pthread_mutex_init(&mutex, NULL); // Initialisation du mutex


// Création des threads

for (int i = 0; i < NUM_THREADS; i++) {

pthread_create(&threads[i], NULL, inc, NULL);

}

// Attente de la fin des threads

for (int i = 0; i < NUM_THREADS; i++) {
pthread_join(threads[i], NULL);
}

pthread_mutex_destroy(&mutex); // Destruction du mutex

printf("Nbre de places occupées = %d\n", nb_places_occupees);
return 0;
}
```

Les sémaphores

Exemple:

Donner un exemple d'échange Ping-Pong entre deux threads en C en utilisant des sémaphores pour assurer leur synchronisation ?

Les sémaphores

Exemple de ping pong

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t sem_ping, sem_pong; // Déclaration des sémaphores

void *ping(void *arg) { for (int i = 0; i < 5; i++) {

sem_wait(&sem_ping); // Attendre son tour

printf("Ping ...\n");
sleep(1);
sem_post(&sem_pong); // Débloquer Pong
}
return NULL; }

void *pong(void *arg) {

for (int i = 0; i < 5; i++) {

sem_wait(&sem_pong); // Attendre son tour
printf("Pong!\n");
sleep(1);
sem_post(&sem_ping); // Débloquer Ping }

return NULL; }
```

```c
int main() {

pthread_t t1, t2;

sem_init(&sem_ping, 0, 1); // Ping commence (1 = débloqué)

sem_init(&sem_pong, 0, 0); // Pong attend (0 = bloqué)

pthread_create(&t1, NULL, ping, NULL);
pthread_create(&t2, NULL, pong, NULL);

pthread_join(t1, NULL);
pthread_join(t2, NULL);

sem_destroy(&sem_ping);
sem_destroy(&sem_pong);

return 0; }
```