Programmation Concurrente

Programmation Système — R3.05

C. Raïevsky

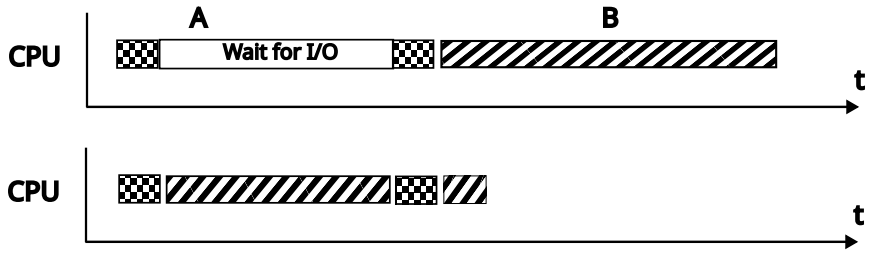
IUT A Valence
Université Grenoble Alpes

Département Informatique

BUT Informatiques 2ème année

Parallèlisme

- Utiliser les capacités de calcul parallèle
- Ne pas attendre la fin d'opérations longues (entrée/sortie, utilisateur, swapping)



Problèmes entrainés par le parallèlisme

Deux grandes classes de problèmes

Accès concurrent à des ressources partagées

- Conflit d'accès
- Incohérence des données

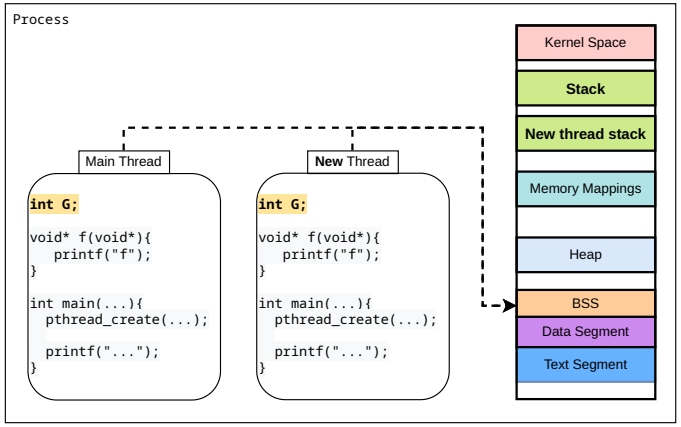
Synchronisation de flux d'exécution

- Producteurs - consommateurs
- Calculs parallèles répétés

Plan

Accès concurrent à des ressources partagées

Exemple de deux threads sur une variable globale



Accès concurrent – Exemple minimal

```
1  int G = 0;
2  void* incG() {
3     for (int i = 0; i < 500000; ++i) {   G++; }
4  }
5
6  int main(int argc, char* argv[]){
7     pthread_t th1, th2;
8
9     pthread_create(&th1, NULL, incG, NULL);
10    pthread_create(&th2, NULL, incG, NULL);
11
12    pthread_join(th1, NULL);
13    pthread_join(th2, NULL);
14
15    printf("Final value: %d\n", G);
16    return EXIT_SUCCESS;
17 }
```


Problème : Accès à une mémoire partagée

Exemple : compte en banque

Tâche A : dépot de chèques
n ← n + na

Tâche B : dépôt de liquide
n ← n + nb

Version ColdFire :

Tâche A		Tâche B	
(1a)	MOVE n,%d0	(1b)	MOVE n,%d0
(2a)	ADD na,%d0	(2b)	ADD nb,%d0
(3a)	MOVE %d0,n	(3b)	MOVE %d0,n

Exécution sans protection (monoprocesseur)

Version ColdFire :

Tâche A		Tâche B	
(1a)	MOVE n,%d0	(1b)	MOVE n,%d0
(2a)	ADD na,%d0	(2b)	ADD nb,%d0
(3a)	MOVE %d0,n	(3b)	MOVE %d0,n

$n = X$		$\%d0_a$		$\%d0_b$
	(1a) MOVE n,%d0	X		
	Interruption ⇒		(1b) MOVE n,%d0	X
			(2b) ADD nb,%d0	$X + nb$
$X + nb$			(3b) MOVE %d0,n	
	(2a) ADD na,%d0	$X + na$		
$X + na$	(3a) MOVE %d0,n	$X + na$		

Exécution sans protection (multiprocesseur)

Version ColdFire :

Tâche A		Tâche B	
(1a)	MOVE n,%d0	(1b)	MOVE n,%d0
(2a)	ADD na,%d0	(2b)	ADD nb,%d0
(3a)	MOVE %d0,n	(3b)	MOVE %d0,n

$n = X$		$\%d0_a$		$\%d0_b$
	(1a) MOVE n,%d0	X	(1b) MOVE n,%d0	X
	(2a) ADD na,%d0	$X + na$	(2b) ADD nb,%d0	$X + nb$
$X + na$?	(3a) MOVE %d0,n	$X + na$	(3b) MOVE %d0,n	$X + nb$

Dans ce cas, la dernière tâche à s'éxécuter a le dernier mot

Dans tous les cas n $\neq$ na+nb

Solutions pour l'accès concurrent

La solution générale est :

L'exclusion mutuelle

Faire en sorte que les tâches n'accèdent pas aux ressource partagée de manière concurrente
Nécessite :

Identifier et protéger les sections critiques

Accès concurrent – Exemple minimal

```
1  int G = 0;
2  void* incG() {
3     for (int i = 0; i < 500000; ++i) {  G++; }
4  }
5
6  int main(int argc, char* argv[]){
7     pthread_t th1, th2;
8
9     pthread_create(&th1, NULL, incG, NULL);
10    pthread_create(&th2, NULL, incG, NULL);
11
12    pthread_join(th1, NULL);
13    pthread_join(th2, NULL);
14
15    printf("Final value: %d\n", G);
16    return EXIT_SUCCESS;
17 }
```


Solutions

- Attente active
- Mutexes
- Moniteurs
- Variables atomiques

Attente active

- Attente active avec des instructions classiques : solution de Dekker-Peterson
- Attente active avec l'instruction `test & set`

Dans les deux cas : gaspillage de cycles processeur

Solution de bas niveau - Les Verrous ou *Mutex* ou *Lock*

Mutex - Mutual Exclusion

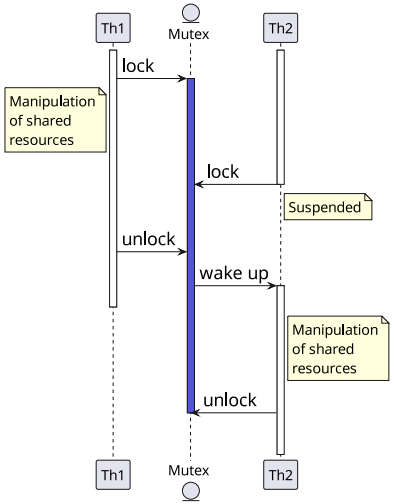
Deux opérations principales :

- Lock : "prend" le verrou
- Unlock : relâche le verrou

- Un et un seul thread peut avoir le verrou
- Les deux fonctions sont atomiques, *thread-safe*

Seul le thread qui a pris le mutex peut le libérer ! ! !

Mutex - Exemple deux threads



Les verrous POSIX – *pthread_mutex*

Mis en œuvre sous Linux avec 3 primitives principales :

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr));
```

Dans le cas d'un mutex "normal", on l'initialise avec :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Utilisation d'un mutex - Exemple

```c
int G = 0;
pthread_mutex_t verrou =
            PTHREAD_MUTEX_INITIALIZER;

void* incG() {
   for (int i = 0; i < 500000; ++i) {
      pthread_mutex_lock(&verrou);
      G++;
      pthread_mutex_unlock(&verrou);
   }
}
```

```c
int main(int argc, char* argv[]){
   pthread_t th1, th2;

   pthread_create(&th1, NULL, incG, NULL);
   pthread_create(&th2, NULL, incG, NULL);

   pthread_join(th1, NULL);
   pthread_join(th2, NULL);

   printf("Final value: %d\n", G);
   return EXIT_SUCCESS;
}
```


Moniteurs

- Assure l'exclusion mutuelle au niveau d'un objet
- Toutes les méthodes publiques prennent ce verrou

Exemple d'un compteur en C - Module `monitor`

Interface : **cpt_monitor.h**

```c
void cpt_init();

void cpt_increment();

int cpt_get();
```

cpt_monitor.c

```c
#include <pthread.h>

static int G = 0;
static pthread_mutex_t lock =
            PTHREAD_MUTEX_INITIALIZER;

int cpt_get(){
   return G;
}
```

cpt_monitor.c (suite)

```c
void cpt_init(){
   pthread_mutex_lock(&lock);
   G = 0;
   pthread_mutex_unlock(&lock);
}

void cpt_increment(){
   pthread_mutex_lock(&lock);
   G++;
   pthread_mutex_unlock(&lock);
}
```

Solution particulière, de type "singleton"

Exemple d'un compteur en C - Utilisation du module `cpt_monitor`

concurrent-cpt_monitor.c

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

#include "cpt_monitor.h"

const int REP = 500000;

void* inc() {
   for (int i = 0; i < REP ; i++) {
      cpt_increment();
   }
   return NULL;
}
```

Les `mutex` ont disparus

```c
int main(int argc, char* argv[]){

   cpt_init();

   pthread_t th1;
   pthread_t th2;

   pthread_create(&th1, NULL, inc, NULL);
   pthread_create(&th2, NULL, inc, NULL);

   pthread_join(th1, NULL);
   pthread_join(th2, NULL);

   printf("Expected value: %d\n", 2*REP);
   printf("Final value   : %d\n", cpt_get());

   return EXIT_SUCCESS;
}
```

Variables atomiques

Pour les opérations simples :

- Incrément/Décrément
- Attribution d'une valeur
- Échange de deux valeurs

Ne protège pas une section critique

Démo : atomicInt.c

Nouveaux problèmes

- Interblocage
- Famine

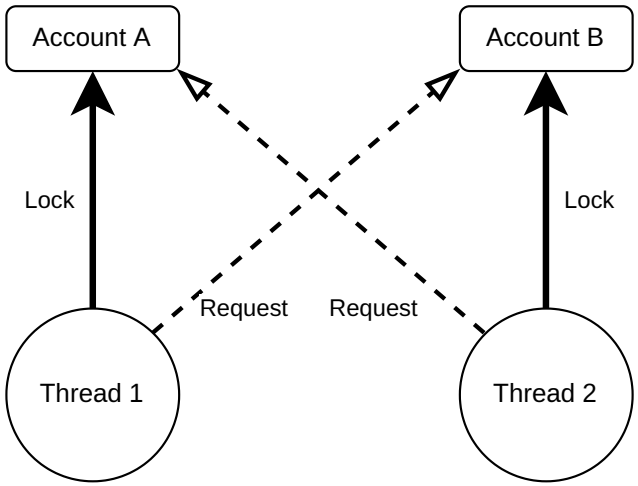
Interblocage - *Deadlock*

Blocage mutuel de plusieurs tâches

Exemple :

- Deux tâches transfèrent des montants d'un compte à l'autre
- Il faut protéger les accès concurrents aux comptes
- Démo : deadlock.c + tableau

Interblocage - *Deadlock*



Interblocage - *Deadlock*

Conditions à l'interblocage - Conditions de Coffman

1. Exclusion mutuelle
2. Détention et attente
3. Pas de préemption
4. Attente circulaire

Interblocage - *Deadlock*

Solutions

1. Briser une des conditions de Coffman
   1.1 Exclusion mutuelle
   1.2 Détention et attente
   1.3 Pas de préemption
   1.4 Attente circulaire
2. Vérifier avant chaque allocation
3. Algorithme du banquier

Mutex récursifs

- Un mutex récursif peut être pris récursivement par la même tâche
- Il doit être libéré autant de fois qu'il a été pris.

Les mutex `pthread` ne sont pas récursifs par défaut

- Un thread qui tente de prendre un mutex qu'il a déjà se retrouve bloqué
- Pour obtenir un mutex récursif :

```
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
```


Famine - *Starvation*

- Inéquité d'accès aux ressources
- Conservation des ressources par une tâche

Plan

Problèmes de synchronisation

Synchroniser un ensemble de tâches

- Attendre le résultat d'une tâche avant de démarrer la suivante
- Réutiliser des tâches existantes
- Attendre le résultat intermédiaire d'un ensemble de tâches

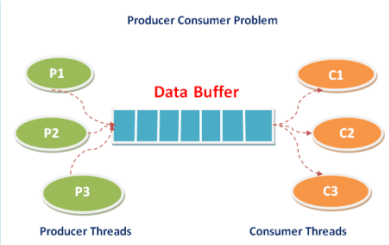
Gérer le séquencement de tâches

- Mettre en sommeil les tâches en attente de ressources
- Réveiller des tâches lorsque des données sont disponibles
- Éviter les attentes actives

Problème classique

Producteurs consommateurs

Exemple : Producteurs - Consommateurs



Producteurs

- Pose des données à traiter dans le buffer

Consommateurs

- Prend les données à traiter dans le buffer

Problèmes classiques

- Éviter les conflits d'accès au buffer
- Éviter les attentes actives sur le buffer
- Mettre en sommeil les tâches qui n'ont rien à faire
  - Un consommateur lorsque le buffer est vide
  - Un producteur lorqu'il est plein
- Ne réveiller des tâches que lorsque les ressources sont disponibles
- Maximiser le nombre de tâches actives

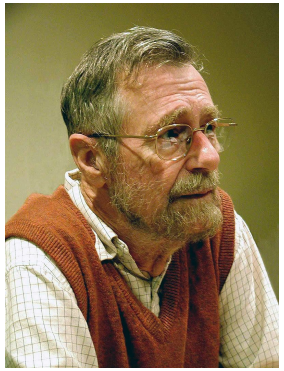
Solutions

- Semaphores
- Condition Variables

Sémaphores

Mécanisme de synchronisation

- Inventé par Edsger Dijkstra
- Permet de réguler l'accès à des ressources partagées
- Dans le cas où plusieurs ressources interchangeables sont disponibles



Sémaphore

Sémaphore :

- Un entier (S)
- Deux opérations :

Wait

- Bloquant si $S \leq 0$
- Décrémente S

Post

- Jamais bloquant
- Incrémente S

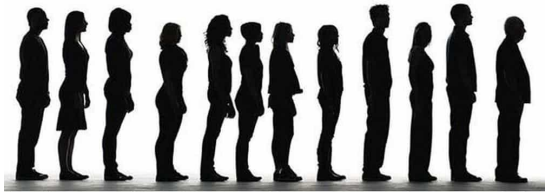
Similaire aux opérations lock et unlock d'un mutex MAIS :

- Wait pas forcément bloquant (si $S > 1$)
- Une tâche peut faire un Post sans avoir fait le Wait correspondant

Sémaphore - Analogie

Analogie : cabines d'essayage avec comptoir

- Un nombre de cabines libres : S
- Si $S > 0$: on passe, S décrémenté
- Quand on sort, S incrémenté
- Si $S \leq 0$ quand on arrive : on attend



Expérience au tableau

- Un buffer,
- deux curseur : lecture, écriture

Producteur

1. Écrit une valeur,
2. Met à jour les curseurs,

Consommateur

1. Lit (efface) la plus ancienne valeur,
2. Met à jour les curseurs,

Interruptions : que se passe-t-il ?

Quelle solution ?

Expérience au tableau

- Un buffer,
- deux curseur : lecture, écriture
- une craie (mutex),

Producteurs

1. Prend la craie,
2. Écrit une valeur s'il y a de la place,
3. Met à jour les curseurs,
4. Pose la craie.

Consommateurs

1. Prend la craie,
2. Lit (efface) la plus ancienne valeur,
3. Met à jour les curseurs,
4. Pose la craie.

Expérience au tableau

Comment améliorer la synchronisation ?

Expérience au tableau - Sémaphores

- Un buffer,
- deux curseur : lecture, écriture
- une craie (mutex),
- Deux sémaphores : lecture et écriture

Producteurs

1. Passe le sémaphore d'écriture,
2. Prend la craie,
3. Écrit une valeur,
4. Met à jour les curseurs,
5. Pose la craie,
6. Incrémente le sémaphore de lecture.

Consommateurs

1. Passe le sémaphore de lecture,
2. Prend la craie,
3. Lit (efface) la plus ancienne valeur,
4. Met à jour les curseurs,
5. Pose la craie,
6. Incrémente le sémaphore d'écriture.

Variable de condition - *Condition Variable*

Variable particulière

- Permettant aux tâches de se mettre en attente
- Jusqu'à ce que la variable soit "signalée"
- Associée à un mutex

Trois opérations :

- Wait → Met le thread en sommeil
- Signal → Réveille un seul thread
- Broadcast → Réveille tous les threads

Variable de condition - *Condition Variable*

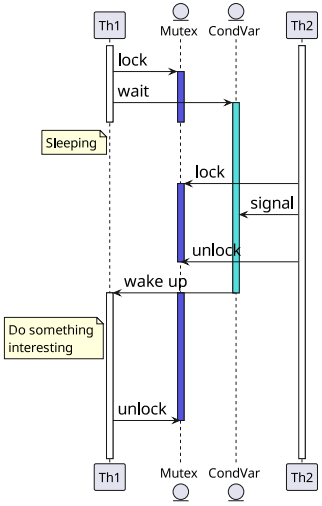
Toujours associée à un mutex

- Il faut détenir le mutex **avant wait**
- wait va relâcher le mutex en même temps qu'endormir le thread
- Lorsque le thread est réveillé, il reprend le mutex

Documentation

- `man` pthread_cond_wait
- Un document détaillée, en anglais

Variable de condition - *Condition Variable* - Exemple



Variable de condition - Utilisation typique minimale

Mise en attente d'un thread sur une variable de condition

```
1  pthread_mutex_lock(&mut);  // Always lock mutex before entering check
2
3  while (condition){
4     pthread_cond_wait(&cond_var, &mut); // Release mutex and sleep
5  }
6
7  // Just awaken, we have the mutex
8
9  /* Do something interesting */
10
11 pthread_mutex_unlock(&mut); // We need to release the mutex
```

```c
pthread_mutex_t mut =
   PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_var =
   PTHREAD_COND_INITIALIZER;
const int rep = 5;
bool go = false;
bool finish = false;

void* step(void* p) {
   while (!finish) {
      pthread_mutex_lock(&mut);
      while (!go){
         pthread_cond_wait(&cond_var, &mut);
      }
      go = false;
      pthread_mutex_unlock(&mut);
   }
   return NULL;
}
```

```c
int main(int argc, char* argv[]){
   pthread_t th;
   pthread_create(&th, NULL, step, NULL);

   for (int i = 0; i<rep; i++){
      sleep(1);
      pthread_mutex_lock(&mut);
      go = true;
      pthread_cond_signal(&cond_var);
      pthread_mutex_unlock(&mut);
   }

   pthread_mutex_lock(&mut);
   finish = true;
   go = true;
   pthread_cond_signal(&cond_var);
   pthread_mutex_unlock(&mut);

   pthread_exit(NULL);
   return EXIT_SUCCESS;
}
```


Question

Pourquoi ne pas utiliser un simple mutex à la place d'une variable de condition ?

Exemple plus complet

Le thread principal dicte le rythme à un ensemble de threads

`steps_cond_var.c`