Programmation concurrente

Outils élémentaires de synchronisation

Polytech/INFO 4, 2022-2023

Fabienne Boyer
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

Plan

- **Pourquoi a t'on besoin de synchronisation**
- **Les concepts et outils de base**
  - Exclusion mutuelle
  - Section critique
  - Verrou

Le problème

threads (ou processus) concurrents

accès concurrents à des ressources partagées

incohérences potentielles

Exemple du compte bancaire

```
credit(t_acc account, float val) {account = account + val; }
        (a) load(account, ACCU)
        (b) add(ACCU, val)
        (c) store(ACCU,  account)
```

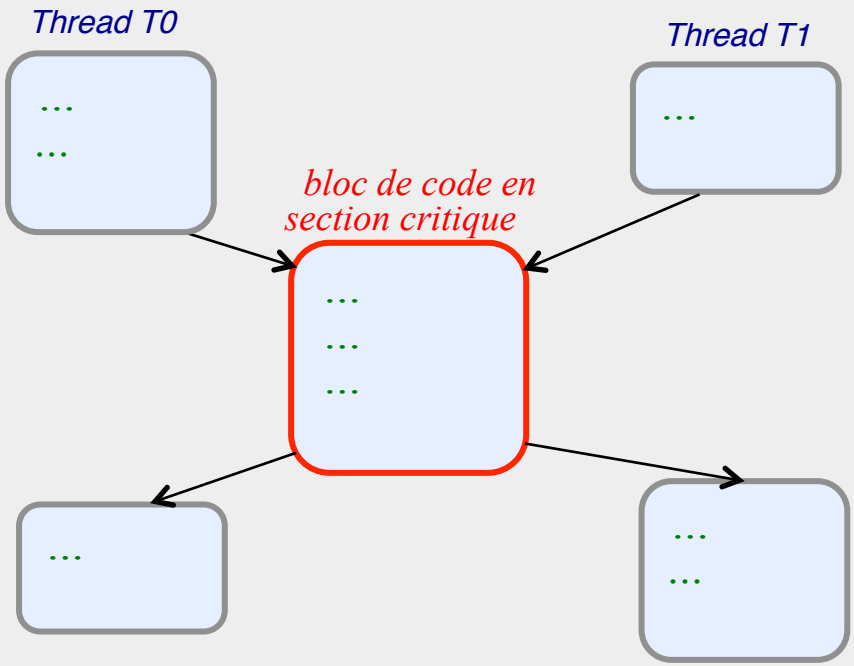
Scenario:

thread T1: credit(A, 2500) // thread T2: credit(A, 2000)

→il se peut que le compte A ne soit crédité que de 2500 ou 2000, au lieu de 4500 (vu en TD)

Concept d'exclusion mutuelle

- Un bloc de code en exclusion mutuelle garantit qu'il n'y a jamais plus d'un thread en exécution dans le bloc
- On appelle cela une section critique (Dijkstra)



→ Si T0 entre avant T1: T1 ne pourra entrer dans le bloc que lorsque T0 en sera sorti

→ Si T1 entre avant T0: T0 ne pourra entrer dans le bloc que lorsque T1 en sera sorti

Mise en œuvre des sections critiques

- **Principe de mise en oeuvre**

  <code de contrôle *avant*> *(entry-section)*

  bloc de code en exclusion mutuelle

  <code de contrôle *après*> *(exit-section)*

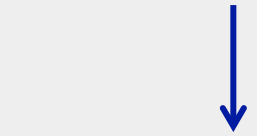
- **Propriétés à garantir**
  - Exclusion: lorsqu'un processus est dans une SC, aucun autre ne peut y être
  - Pas de blocage intempestif: si la SC est libre, un processus doit pouvoir entrer immédiatement
  - Absence de privation (famine): tout processus doit pouvoir entrer en SC en un temps fini
  - Aucune hypothèse ne doit être faite sur la politique de scheduling

Mise en œuvre logicielle des sections critiques (premier essai)

```
// shared data
int busy= false;

 // entry-section
while (busy);
busy = TRUE;

<critical section>

// exit-section
busy = FALSE;
```

a) load busy ACCU
b) cmp null
c) branch ..

On utilise une variable partagée (busy) qui indique si la SC est libre



- On va voir en TD pourquoi cette solution ne fonctionne pas
- Ce point doit être TOTALEMENT clair pour vous

Solution purement algorithmique: Algorithme de Dekker (1965)

```
// shared data
int turn = 0;
int flag[2] = {false, false};


// entry-section
flag[i] = TRUE;
while (flag[1-i]) {  // the other wants to enter
            if (turn == 1-i) {   // it isn't my turn
                        flag[i] = FALSE;
                        while (turn != i);
                        flag[i] = TRUE;
            }
}

<critical section>

// exit-section
turn = 1 – i;
flag[i] = FALSE;
```

- fonctionne pour 2 processus,
- non généralisable à *n* processus,
- non fifo (risque de famine)

Algorithme de Peterson (1981) (vu en TD)

```
// shared data
int last = 0;
int interested[2] = {FALSE, FALSE};

// entry-section
interested[i] = TRUE;
last = i;
while (last == i && interested[1-i]) ; // wait

<critical section>

// exit-section
interested[i] = FALSE;
```

fonctionne pour 2 processus, généralisable à *n* processus (algo. de Lamport)

Solution purement algorithmique: Algorithme de Lamport (vu en TD)

```
// shared data
int ticket[n] = {0,0,0,…};
int choosing[n];    // indicates if a process is currently getting a ticket

// entry-section
choosing[i] = TRUE;                          // process i is getting a ticket
ticket[i] = 1 + max(ticket[0], …, ticket[n-1]);
choosing[i] = FALSE;                         // process i has got a ticket

for (j=0; j< n; j++) {
   while (choosing[j]);                      // wait while process  j is getting a ticket
   while (ticket[j] != 0) &&                 // wait  if process j has a lower ticket
          ((ticket[j] < ticket [i]) || (ticket[j] == ticket [i]) && j<i)));
}

<critical section>

// exit-section
ticket[i] = 0;
```

→ **fonctionne pour *n* processus**

Mise en œuvre des sections critiques

- **Solutions logicielles**
  - Complexes et relativement peu efficaces
- **Solutions basées sur le matériel**
  - Masquage des IT
  - Usage de l'instruction Test&Set / Swap

Usage du masquage des interruptions

- **Principe**
  - Entry section : masquer les IT
  - Exit section : restaurer les IT
- **Problèmes**
  - Solution non utilisable en multiprocesseurs
  - Temps passé en SC non contrôlable
- **En mono-processeur, le masquage est acceptable quand la durée d'exécution du code masqué est très courte**

Instruction Test&Set

- **Instruction assembleur qui teste et modifie le contenu d'un mot mémoire de manière atomique (non interruptible)**
- **Fonctionne en multiprocesseurs**

Equivalent fonctionnel en langage C:

```
int Test&Set (int *b) {
   // set b to true, then return previous value of b
   int res = *b;
   *b = TRUE;
   return res;
}
```

Mise en œuvre de section critique avec Test&Set

```
// shared data
int busy= FALSE;


void entry_section(){
while (Test&Set (&busy));
}

void exit_section(){
busy = FALSE;
}
```

- → Garantit l'exclusion
- → Risque de privation
- → Utiliser un système de ticket pour éviter la privation

Mise en œuvre de section critique avec Test&Set

```
// shared data type
typedef struct {
    int dist_ticket;
    int clock;
    int busy;
}t_sc;


void init_section(t_sc *sc){
  sc->dist_ticket=0;
  sc->clock=0;
  sc->busy = FALSE;
}
```

```
void entry_section(t_sc *sc){
        int my_ticket;
        while (test&set(&sc->busy)));
takes a ticket:
        my_ticket = sc->dist_ticket++;
        release(&(sc->busy));
wait for my turn:
        while (my_ticket != sc->clock);

void exit_section(t_sc *sc){
        while (test&set(&sc->busy)));
        sc->clock++;
        release(&(sc->busy));
}
```

Suppose que l'instruction CMP est atomique (processor dependant)

Exemple applicatif

```
// shared data

int cpt = 0; // a shared counter
t_sc sc;     // critical section to manipulate cpt
```

```
Thread T0:

  init(&sc);

  <create threads T1 .. Tn,
   passing them &sc as argument>
```

```
Thread Tn (t_sc *sc):

    …
    entry_section(sc);
    cpt++;
    exit_section(sc);
    …
```

Scenario d'exécution possible avec 3 threads (T_1, T_2, T_3)

init(&l);
T_3: entry(sc) ……..
T_3: in SC……..…….....
T_1: entry(sc) – loop on while(my_ticket..)……..
T_2: entry(sc) – loop on while(my_ticket ..).……
T_3: exit(sc)……..
T_1: in SC……...
T_1: exit(sc)……
T_2: in SC…….....
T_2: exit(sc)……..

Ticket T_3	Ticket T_1	Ticket T_2	clock
			0
0			0
0			0
0	1		0
0	1	2	0
	1	2	1
	1	2	1
		2	2
		2	2
			3

Instruction Swap

- **Permutation atomique de deux mots mémoire**
- **Fonctionne en multiprocesseurs**
- **Alternative au Test&Set**

Equivalent fonctionnel en langage C:

```
void Swap(int *a, int *b) {
   int tmp = *a;
   *a = *b;
   *b = tmp;
                        }
```

Exercice
Programmer les fonctions entry_section et exit_section avec swap au lieu de test&set

Bilan sur les solutions précédentes

- **Problème des solutions précédentes**
  - Ce sont des solutions à base d'attente active
  - gaspillage de l'UC
- **Principes des solutions à base d'attente passive**
  - Suspendre un processus ou (thread) lorsque la section est verrouillée
  - Le réveiller lorsqu'elle se libère

→ Utilisation de deux primitives fournies par le noyau :

→ suspend(..) et wakeup(..)

Principes de mise en œuvre des fonctions Suspend et Wakeup

Données du noyau

```
proc_ctxt current;
proc_ctxt_list ready_queue;
```

```
void suspend( proc-ctxt_list *queue) {
   proc_ctxt *old = current;
   mask();
   put_last(queue, current);
   current= get_first(ready_queue);
   ctxt_swap(current, old);
   unmask();
 }
void wakeUp(proc_ctxt *p){
    mask();
    put_last(ready_queue, p);
    unmask();
}
```

Les verrous

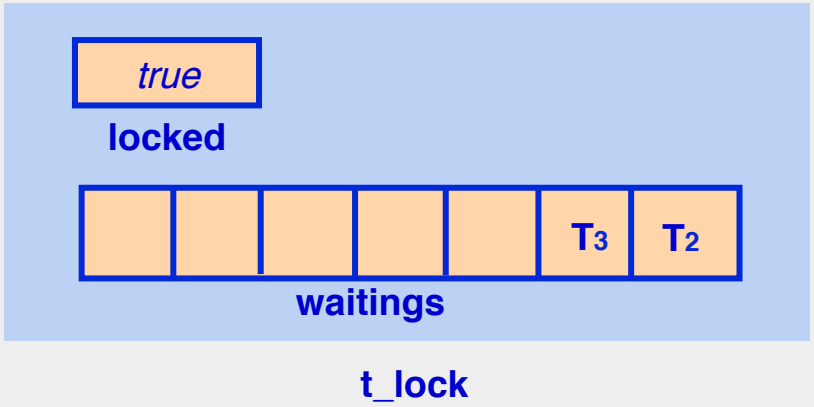
- **Outil d'exclusion mutuelle à base d'attente passive**
  - Sauf pour le cas particulier des *spin-locks*
- **Deux opérations atomiques de base**
  - lock() & unlock()
  - En général, le thread qui déverrouille doit être celui qui a verrouillé
- **Peut être ré-entrant ou pas**
  - Selon qu'un thread peut (ou pas) verrouiller un verrou qu'il possède déjà

```
// shared data
int id = 0;
t_lock l;

void init() {
    id= 0;
    init(&l);
}

int alloc_id () {
 lock(&l);
 int res = id;
 id++;
 unlock(&l);
 return res;
}
```

Exemple d'usage d'un verrou

Exemple de mise en œuvre d'un verrou

```
typedef struct {
   int locked;   // is the lock taken or free
   proc_ctxt_list waitings; // waiting queue
} t_lock;

void init_lock(t_lock *l) {
      mask();
      l->locked= FALSE;
      init_list(&(l->waitings));
      unmask();
     }
```



Exemple de mise en œuvre d'un verrou

```
void lock(t_lock *l) {
 mask();
 if (l->locked)
        suspend(&(l->waitings));
  l->locked= TRUE;
 unmask();
}
```

```
void unlock(t_lock *l) {
   proc_ctxt p;
   mask();
   l->locked= FALSE;
   if (p= get_first((&(l->waitings))))
          wakeup(p);
   unmask();
}
```

Valide ?

Risque de non-exclusion

	Waitings	ReadyQueue
P_1: lock *(taken)*		[⋯, P_4, ⋯]
P_2: lock	P_2	[⋯, P_4, ⋯]
P_3: lock	P_2, P_3	[⋯, P_4, ⋯]
P_1: unlock	P_3	[⋯, P_4, ⋯, P_2]
P_4: lock *(taken)*		
P_2: lock *(taken)*		

→ P_2 et P_4 obtiennent le verrou simultanément ..

→ La solution n'est pas exclusive

Exemple de mise en œuvre d'un verrou

```
void lock(t_lock *l) {
 mask();
 while (l->locked)
        suspend(&(l->waitings));
 l->locked= TRUE;
 unmask();
}
```

```
void unlock(t_lock *l) {
   proc_ctxt p;
   mask();
   l->locked= FALSE;
   if (p= get_first((&(l->waitings))))
          wakeup(p);
   unmask();
}
```

Valide ?

Risque de privation

	Waitings	ReadyQueue
P_1: lock *(taken)*		[⋯, P_4, ⋯]
P_2: lock	P_2	[⋯, P_4, ⋯]
P_3: lock	P_2, P_3	[⋯, P_4, ⋯]
P_1: unlock	P_3	[⋯, P_4, ⋯, P_2]
P_4: lock *(taken)*		
P_2: lock	P_3, P_2	

→ Non seulement P2 n'obtient pas le verrou, mais en plus il se retrouve en queue de file d'attente ..

→ Risque de privation

Exemple de mise en œuvre d'un verrou

```
void lock(t_lock *l) {
 mask();
 if (l->locked)
        suspend(&(l->waitings));
 l->locked= TRUE;
 unmask();
}
```

```
void unlock(t_lock *l) {
   proc_ctxt p;
   mask();
   l->locked= FALSE;
   if (p= get_first((&(l->waitings))))
          wakeup(p);
   else  l->locked= FALSE;
   unmask();
}
```

Valide ?

Exemple de mise en œuvre d'un verrou

```
void lock(t_lock *l) {
 mask();
 if (l->locked)
        suspend(&(l->waitings));
 l->locked= TRUE;
 unmask();
}
```

```
void unlock(t_lock *l) {
   proc_ctxt p;
   mask();
   l->locked= FALSE;
   if (p= get_first((&(l->waitings))))
          wakeup(p);
   else  l->locked = FALSE;
   unmask();
}
```

Oui cette fois c'est ok !

Verrous existants

- **Langage C**
  - pthread_mutex_t,
  - rwlock_t,
  - spinlock_t,
  - futex, ..
- **Langage C++ ou C#**
  - mutex,
  - timed-mutex, ..
- **Java**
  - interface Lock (package java.util.concurrent)
  - classes ReentrantLock, ReadWriteLock, ..

Class Lock en Java

```
// shared data
int cpt;
Lock l = ...;

int getCpt(){
   l.lock();
   try {
      cpt++;
   } finally {
    l.unlock();
  }
}
```

Résumé / outils de synchronisation de base

- **Algorithmes à base d'attente active**
  - Solutions purement algorithmiques (Peterson, Boulanger, ..)
  - Solutions utilisant l'instruction matérielle Test&Set (ou équivalent)
- **Solutions à base d'attente passive (+ performantes)**
  - Verrous
  - Moniteurs, Sémaphores, Sections critiques conditionnelles --> prochain cours
- **Ce que vous devez avoir totalement compris**
  - L'indéterminisme du à la concurrence
  - La notion de section critique
  - La notion d'attente passive
  - Les verrous (*usage et principes d'implémentation*)