Gestion des ressources

Jean-Michel Adam - Florian Rodriguez

Plan

- Spécification du problème
- Section Critique (SC)
- Exclusion Mutuelle
  - Principe
  - Propriétés
- Réalisation d’exclusion Mutuelle
  - Matérielle (Interruption, instruction TAS)
  - Logicielle (attente active, verrou, sémaphores);
- Difficultés liées au partage de ressources
  - Interblocage
  - Famine

Introduction

- Les processus partagent des ressources
- Les ressources sont des données ou des périphériques
- Des ressources sont partagées simultanément par plusieurs processus
- Des ressources sont partagées séquentiellement par plusieurs processus
- Les ressources sont réquisitionnables ou non

Ressources partageables simultanément

- fichier ou zone mémoire en lecture seule
- périphériques partageable (disques)

Ressources partageables séquentiellement

- fichier ou zone mémoire en écriture
- périphériques non partageable (imprimante)
- processeur

*-> ce sont des **ressources critiques***

Spécification du problème

Exemple: partage d'une donnée en mémoire

- Deux tâches (A et B) s’exécutent simultanément
- Elles doivent modifier la même donnée
- Chaque tâche doit donc :
  - lire cette donnée,
  - calculer sa nouvelle valeur
  - mémoriser la nouvelle valeur

-> Problème: l’exécution simultané des tâches A et B peut perturber les calculs

Exemple: partage d'une variable solde

Tâche A: dépot de chèque

- Lire la valeur de la variable solde
- Ajouter le montant du chèque (mtCh)
- Mémoriser la nouvelle valeur

```
courA <- Solde
courA + mtCh
Solde <- courA
```

Copy

Tâche B: dépot d'espèces

- Lire la valeur de la variable solde
- Ajouter le montant des espèces (mtL)
- Mémoriser la nouvelle valeur

```
courB <- Solde
courB + mtL
Solde <- courB
```

Copy

Exemple: partage d'une variable solde

Solde = X		courA		courB
X				
	courA -< Solde	X		
			courB <- Solde	X
			courB <- courB + mtL	X + mtL
X+mtL			Solde <- courB	
	courA <- courB +mtCh	X+mtCh		
X+mtCh	Solde <- courA			

Exemple: spooler d'impression

- Lorsque un processus veut imprimer un fichier, il entre son nom dans un répertoire de spoole spécial.
- Un autre processus le démon d’impression, regarde périodiquement s’il y a des fichiers à imprimer.
- notre répertoire de spoole possède un certain nombre d’entrées, numérotées 0,1,2 … etc pour accueillir les fichiers à imprimer.
- Deux variables partagées : out, qui pointe vers le prochain fichier à imprimer et in, qui pointe vers la prochaine entrée libre du répertoire

Exemple: spooler d'impression

- Processus A lit `in` et mémorise 7 dans `place_libre_suivante`
- Interruption d'horloge, Processus B lit `in` qui vaut toujours 7 et place le nom de son fichier à la séptième position du répértoire puis met à jour `in` à 8
- Processus A est relancé et poursuit son execution. Lit la variable `place_libre_suivante`, y trouve 7, place le nom du fichier à imprimer à la poition 7. Puis met 8 dans `in`.
- Le répértoire d'impression est dans un état cohérent, pas de pb.
- Le fichier B ne sera jamais imprimé.

Section critique

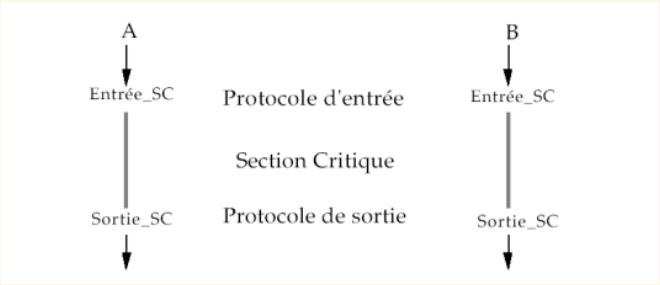
- Une **Section Critique (SC)** est un segment de code qui accède à une ressource partagée
- Une **SC** est ensemble de suites d'instructions qui peuvent produire des résultats erronés lorsqu'elles sont exécutées simultanément par des processus différents.
- L'exécution de deux **SC** appartenant à des ensembles différents et ne partagent pas de variables ne pose aucun problème.
- Pratiquement les **SC** doivent être détectées par les concepteurs de programmes.
- Dès qu'il y a des variables partagées, il y a forte chance de se retrouver en présence de **SC**.

Section critique: Principe

- Les SC doivent être exécutés en **Exclusion Mutuelle**:
  - une SC ne peut être commencée que si aucune autre SC du même ensemble n'est en cours d'exécution.
- Avant d'exécuter une SC, un processus doit s’assurer qu’aucun autre processus n'est en train d’exécuter une SC du même ensemble.
- Dans le cas contraire, il devra pas progresser, tant que l’autre processus n’aura pas terminé sa SC.
- Nécessité de définir un protocole d'entrée en SC et un protocole de sortie de SC.

Protocole d'entrée/sortie en SC

- **Protocole d ’entrée en SC (prologue):**
  ensemble d ’instructions qui permet cette vérification et la non progression éventuelle.
- **Protocole de sortie de SC (épilogue):**
  ensemble d’instructions qui permet à un processus ayant terminé sa SC d'avertir d'autres processus en attente que la ressource soit libre.



Structure des processus

```
Début
Section non critique
  /prologue/
  *SC*
  /épilogue/
Section non critique
Fin
```

Copy

Exclusion mutuelle

Exclusion mutuelle: propriétés

- **Exclusion mutuelle:**
  Si un processus est en SC, aucun autre processus ne peut y accéder.
- **Progression:** (pas d'interblocage)
  Si aucun processus n’est en SC et qu’un processus désire y accéder, il doit pouvoir le faire.
- **Attente limitée:** (pas de famine)
  Si un processus désire accéder à la SC, il doit pouvoir le faire dans un temps fini.

Exclusion mutuelle

- L'exclusion mutuelle n'est pas garantie si:
  - un processus peut entrer en SC alors qu'un autre s'y trouve déjà.
  - un processus désirant entrer en SC ne peut pas y entrer alors qu’il n’y a aucun processus en SC.
  - un processus désirant entrer en SC n'y entrera jamais car il sera jamais sélectionné lorsqu’il est en concurrence avec d'autres processus

Exclusion mutuelle: solutions

- Exclusion mutuelle logicielle
  - attente active (Dekker-Peterson)
  - attente passive (Dijkstra)
    - verrou
    - sémaphores
- Exclusion mutuelle matérielle
  - solution monoprocesseur: masquage d'interruption
  - solution multiprocesseur: instruction TAS (Test And Set)

Exclusion mutuelle: solutions logicielles

- Nous supposons que les instructions en language machine de base (load, store, test) sont atomiques.
  - Si deux instructions sont exécutées en concurrence, le résultat est le même que celui de leur exécution séquentielle.
  - Si un `load` et un `store` sont exécutés en concurrence, le `load` obtiendra soit l'ancienne valeur soit la nouvelle valeur, mais pas une valeur intermédiaire.

Exclusion mutuelle: solutions logicielles

- Nous décrivons seulement un processus P_i typique dont la structure générale est:

```
while(true){
  section_entree();
  region_critique();
  section_sortie();
  hors_region_critique();
}
```

Copy

- Pour deux processus nous avons P_0 et P_1 ou P_i et P_j
- i = |j-1| et j = |i-1|;

Algo 1: Alternance strict

- Les processus P_0 et P_1 partage une variable `turn` qui indique le tour du processus qui peut entrer en SC. Si `turn` vaut i, alors P_i peut entrer en SC.

```
while(true){
  while (turn != i); // attente active
  region_critique();
  turn = j;
  hors_region_critique();
}
```

Copy

- Ne satisfait pas au besoin de **progression** car si `turn` vaut j, P_i ne peut pas entrer en SC.

Algo 2: attente active

- Le pb de l'algo 1 est qu'il ne garde pas suffisamment d'infos sur l'état de chaque processus.
- Pour remédier au problème, nous pouvons remplacer la variable `turn` par un tableau de variables `flag`.

```
var flag = array[0..1] of boolean;
```

Copy

- `flag[i]` indique le désir d'entrer en SC. Si `flag[i]` vaut `true`, alors P_i désire entrer en SC.

```
while(true){
      flag[i] = true;
      while (flag[j]);
   region_critique();
      flag[i] = false;
   hors_region_critique();
}
```

Copy

- L'exclusion mutuelle est satisfaite mais le pb de **progression** persiste.
- Séquence d'execution suivante:
  T_0: P_0 fixe flag[0] = true;
  T_1: P_1 fixe flag[1] = true;

Algo 3: Dekker-Peterson

- Une combinaison des 2 algos précédents.
  - Les processus P_0 et P_1 partagent deux variables `flag` et `turn`.
  - `flag` indique le désir d'entrer en SC
  - `turn` indique le tour du processus qui peut entrer en SC.

```
while(true){
  flag[i] = true;
  turn = j;
  while ((flag[j]) && (turn == j));
  region_critique();
  turn = j;
  flag[i] = false;
  hors_region_critique();
}
```

Copy

- Satisfait aux propriétés d'**exclusion mutuelle**, de **progression** et d'**attente limitée**.

Exclusion mutuelle matériel: solution sur monoprocesseur

- Masquage d'interruption
  - Les interruptions sont désactivées lorsqu'un processus entre en SC.
  - Les interruptions sont réactivées lorsqu'un processus sort de SC.
  - Avantages: simple, rapide
  - Inconvénients:
    - Une SC avec while(1) bloque tout le système
    - Les sytèmes ne permettent pas à tout le monde de masquer n’importe comment les IT.
    - Ne fonctionne pas sur des multiprocesseur (ou gourmand en temps)

Exclusion mutuelle matériel: solution sur multiprocesseur

- Instruction indivisible: `Test_and_Set(TAS)`
  - Instruction atomique qui permet de consulter et de modifier un mot mémoire.
  - L'instruction TAS est exécutée en une seule opération.
- Soit C une variable globale indiquant l'état (ou l'occupation) d'une section critique, c'est-à-dire:
  - C = 1 ==> section critique occupée, et C = 0 ==> section critique libre.
  - L'algorithme de fonctionnement de TAS est le suivant:

Copy

```
function TAS(occupée:boolean) -> boolean {
  boolean tas
  tas <- occupée
  occupée <- true
  return tas
}
```

Exclusion mutuelle matériel: solution sur multiprocesseur

- Exemple d'utilisation

```
while (TAS(C));
region_critique();
C = 0;
hors_region_critique();
```

Copy

- Inconvénient: attente active

Solution à base d'attente passive

- Problème des solutions précédentes: attente active -> gaspillage de l'UC
- Principe des solutions à base d'attente passive:
  - Un processus qui ne peut pas entrer en SC doit être mis en attente.
  - Un processus en attente est réveillé par un autre processus qui sort de SC.
  - Les processus en attente sont mis en file d'attente.

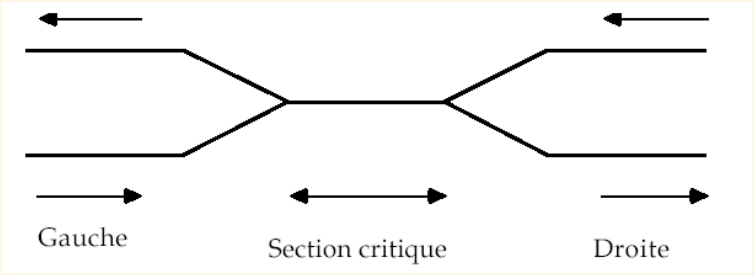
    -> utilisation de **verrous** et de **sémaphores**

Verrou

Copy

```
1  Classe Verrou {
2    privé ouvert: booléan;
3    privé attente: liste de tâches;
4
5    public void init(){
6      ouvert = true;
7      attente = [];
8    }
9
10   public void verrouiller(){
11     if (ouvert){
12       ouvert = false;
13     } else {
14       attente.ajouter(tâche_courante);
15       tâche_courante.suspendre();
16     }
17   }
18
19   public void deverrouiller(){
20     if (attente.est_vide()){
21       ouvert = true;
22     } else {
23       tâche = attente.retirer();
24       tâche.reveiller();
25     }
26   }
27 }
```

Exemple d'utilisation: voie unique avec classe Verrou




```
Tâche TrainGauche(voie: verrou){
  rouler();
      voie.verrouiller();
      traverser();
      voie.deverrouiller();
  rouler();
}
```

Copy

```
Tâche TrainDroite(voie: verrou){
  rouler();
      voie.verrouiller();
      traverser();
      voie.deverrouiller();
  rouler();
}
```

Copy

Sémaphores (Dijkstra)

- Mécanisme de synchronisation entre processus
- Un **sémaphore S** est une variable entière, manipulable par deux opérations atomiques:
  - **P** (proberen): **acquire**
  - **V** (verhogen): **release**

Sémaphores

- **acquire(S)** (P(S)):
  S <- (S - 1)
  si S < 0, alors le processus est mis en attente
  sinon le processus continue son exécution
- **release(S)** (V(S)):
  S <- (S + 1)
  si S <= 0, alors un processus en attente est réveillé

Sémaphores

- **acquire (S)** correspond à une tentative de franchissement.
  S'il n'y a pas de jeton pour la section critique alors attendre,
  sinon prendre un jeton et entrer dans la section (S),
  puis rendre son jeton à la sortie de la section critique.
- Chaque dépôt de jeton **release (S)** autorise un passage.
- Il est possible de déposer des jetons à l'avance

Classe Sémaphore

Copy

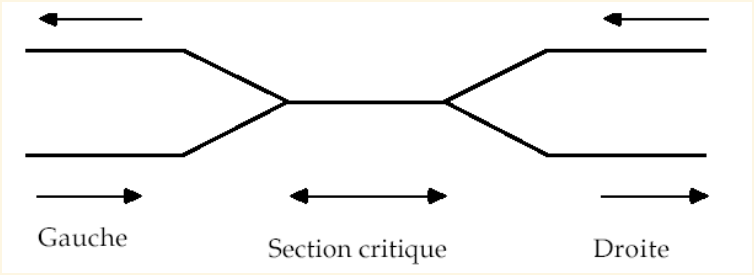
```
Classe Semaphore {
  privé jetons: entier;
  privé attente: liste de tâches;

  public void init(jetons: entier){
    jetons = jetons;
    attente = [];
  }

  public void acquire(){
    jetons--;
    if (jetons < 0){
      attente.ajouter(tâche_courante);
      tâche_courante.suspendre();
    }
  }

  public void release(){
    jetons++;
    if (jetons <= 0){
      tâche = attente.retirer();
      tâche.reveiller();
    }
  }
}
```

Exemple d'utilisation: voie unique avec classe Semaphore




```
Tâche TrainGauche(voie: Semaphore){
  rouler();
      voie.acquire();
      traverser();
      voie.release();
  rouler();
}
```

Copy

```
Tâche TrainDroite(voie: Semaphore){
  rouler();
      voie.acquire();
      traverser();
      voie.release();
  rouler();
}
```

Copy

Difficultés liées au partage de ressources

- Interblocage: exemple avec des sémaphores

```
Semaphore S1(1);
Semaphore S2(1);
```

Copy

```
Tâche A(){
  S1.acquire(); // A1
  S2.acquire(); // A2
  S1.release();
  S2.release();
}
```

Copy

```
Tâche B(){
  S2.acquire(); // B1
  S1.acquire(); // B2
  S2.release();
  S1.release();
}
```

Copy

- L'ordre d'exécution A1, B1, A2, B2 peut provoquer un interblocage: (*deadlock*)

Interblocage

- Un interblocage est une situation dans laquelle deux ou plusieurs processus sont incapables de continuer leur exécution car ils attendent des ressources qui sont détenues par d'autres processus.
- Les conditions nécessaires à l'interblocage sont:
  - **Exclusion mutuelle:** au moins une ressource doit être partagée en mode exclusif.
  - **Attente circulaire:** chaque processus attend une ressource détenue par un autre processus.
  - **Non préemption:** une ressource ne peut être libérée que volontairement par le processus qui la détient.
  - **Attente de ressources supplémentaires:** un processus peut attendre une ressource supplémentaire tout en conservant les ressources déjà acquises.
- Solutions:
  - **Prévention:** éviter les conditions nécessaires à l'interblocage. (supprimer une des conditions)
  - **Détection:** détecter l'interblocage et le résoudre.
  - **Récupération:** résoudre l'interblocage après sa détection.

Prévention à priori

- Méthode de l'allocation globale des ressources à une tâche (cond. attente de ressources supplémentaires)
  - immobilisation non productive de ressources
  - perte d'une bonne partie du parallélisme
- Méthode de l'allocation ordonnée des ressources (cond. attente circulaire)
  - ordonner les ressources et les acquérir dans le même ordre
  - nécessite de connaître à l'avance les ressources nécessaires
- Algorithme du banquier (Dijkstra)
  - chaque processus doit déclarer à l'avance le nombre maximal de ressources qu'il peut demander
  - le système doit vérifier que la demande ne met pas le système dans un état d'interblocage
  - notion d'état fiable/sain: un état est fiable si toutes les demandes futures peuvent être satisfaites

Détection / Récupération

- Algorithme de test d'état sain
- Récupération: reprendre l'exécution dans un état fiable, deux méthodes principales:
    1. Détruire les tâches en interblocage l'une après l'autre, jusqu'à ce que l'interblocage soit résolu.
    2. Sauvegarder périodiquement l'état du système et le restaurer dans un état antérieur à l'interblocage.

En pratique

- Les systèmes d'exploitation modernes utilisent une combinaison de prévention, de détection et de récupération.
- Tous les appels système susceptibles de bloquer une tâche sont implémentés avec des délais d'attente au delà desquels l'appel retourne une erreur.