Synchronisation des Processus: Exclusion Mutuelle

N.Hameurlain
http://www.univ-pau.fr/~hameur

PLAN

- Spécification du problème
- Section Critique (SC)
- Exclusion Mutuelle
  - Principe
  - Propriétés
- Réalisation d'exclusion Mutuelle
  - Matérielle (Interruption, instruction TAS)
  - Logicielle (attente active, verrou, sémaphores);

Spécification du Problème

- Machines monoprocesseurs ou Multi-processeurs;
- Processus s'exécutent sur une machine mono/ multi Processeurs avec mémoire partagée;
- Partager des variables:
  - volontairement: coopérer pour traiter un Problème
  - involontairement: se partager des ressources;

Problème de synchronisation: Exemple

- Le partage de variables sans précaution particulière peut conduire à des résultats imprévisibles:

Si Place_dispo > 0
alors Place_dispo = Place_dispo -1 ;
répondre ('Place réservée') ;
sinon répondre ('plus de place') ;
fsi.

Hypothèses d'exécution

- Les évolutions de chaque processus sont à priori indépendantes;
- Le délai entre deux instructions d'un processus est non nul, mais fini ;
- Deux accès à une même case mémoire ne peuvent être simultanés ;
- Les registres sont sauvegardés et restaurés à chaque commutation de contexte.

Hypothèses d 'exécution: exemple

Processus1	Processus2
LOAD R1 Compteur ADD R1 1 STORE R1 Compteur	LOAD R1 Compteur SUB R1 1 STORE R1 Compteur

Sections critiques(SC): Définition

- Section Critique = ensemble de suites d'instructions qui peuvent produire des résultats imprévisibles lorsqu'elles sont exécutées « simultanément » par des processus différents.

Section Critiques (2)

- Une suite d 'instructions est éventuellement une SC relativement à d 'autres suites d 'instructions et non dans l 'absolu.
- L 'exécution de deux SC appartenant à des ensembles différents et ne partagent pas de variables ne pose aucun problème

Détermination des SC

- L 'existence implique l'utilisation de variables partagées, mais l 'inverse n 'est pas vrai;
- 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.

Exclusion Mutuelle: Principe(1)

- 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.

Exclusion Mutuelle: Principe(2)

- Dans le cas contraire, il devra pas progresser, tant que lautre 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*: ensemble d 'instructions qui permet cette vérification et la non progression éventuelle;
- *Protocole de sortie de SC*: ensemble d'instructions qui permet à un processus ayant terminé sa SC d 'avertir d 'autres processus en attente que la voie est libre

Structure des processus

Processus
Début
Section non Critique
protocole d 'entrée()
SC
protocole de sortie()
Section non critique
Fin.

Propriétés de l'exclusion Mutuelle

1. Au plus un seul processus en SC;
2. Un processus qui veut entrer en SC ne doit pas attendre qu'un autre processus passe avant lui pour avoir le droit.
3. Un processus désirant entrer en SC y entre au bout d 'un temps fini; pas de privation d'y entrer vis à vis d 'un processus

Autres propriétés de l 'exclusion Mutuelle

- La tolérance aux défaillances: si le processus en SC est détruit ou se termine anormalement, il ne faut pas qu'il bloque tout le système;
- La symétrie: Les protocoles d'E/S en SC doivent être identiques pour tous les processus et indépendants de leur nombre.

Exclusion Mutuelle

- L 'exclusion Mutuelle n'est pas garantie si:
  - a) un processus peut entrer en SC alors qu'un autre s'y trouve déjà;
  - b) un processus désirant entrer en SC doit attendre qu'un autre processus passe avant lui pour pouvoir y rentrer;
  - c) un processus désirant entrer en SC n 'y entrera jamais car il sera jamais sélectionné lorsqu'il est en concurrence avec d 'autres processus (et plus particulièrement, quand le processus qui est en SC libère la SC).

Réalisation d'exclusion Mutuelle

- Solutions logicielles : attente active (Dekker, Peterson,), attente passive (Dijkstra);
- Solutions Matérielles:
  - Monoprocesseurs: masquage d 'interruptions;
  - Multiprocesseurs: instruction indivisible, TAS.

Réalisation d'exclusion Mutuelle: solutions logicielles

- Solution naïve: résoudre le problème de partage de variables par une autre variable:

variable *occupé*
Processus pi
section non critique
Tant que (occupé) faire attente ftq //attente active//
occupé:= vrai;
SC
occupé:= faux;
section non critique

Exclusion Mutuelle: solutions matérielles sur monoprocesseur

- Solution brutale: masquage d 'interruptions
  - On empêche les commutations de processus qui pourraient violer l'exclusion Mutuelle des SC;
  - donc Seule l'interruption générée par la fin du quantum de temps nous intéresse
    - Il ne faut pas qu'un processus attende une IT de priorité inférieure à celle générée par la fin du quantum de temps à l'intérieur de SC.

Exclusion mutuelle: solution brutale

- Les IT restent masquées pendant toute la SC, d'où risque de perte d'IT ou de retard de traitement.
- 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.

Exclusion Mutuelle:solution monoprocesseur(1)

Variable commune
Occupé : Booléen initialisé à faux ;
Processus Pi
Var privée déjà_occupé : booléen init. à vrai ;
Tant que déjà_occupé faire
Masquer les IT
déjà_occupé :=Occupé ;
Occupé :=Vrai ;
Démasquer les IT ;
FinTq
SC
Occupé := Faux ;
Fin.

Exclusion Mutuelle:solution monoprocesseur (2)

- Avantage: masquage des interruptions pendant la modification de *occupé*;
- Inconvénient:
  - Le masquage des IT n'est accessible qu'aux programmeurs privilégies pour des raisons de fiabilité : exemple super_ utilisateur.
  - Cette solution ne fonctionne pas sur des Multiprocesseurs.

Exclusion Mutuelle:solution Multiprocesseur (1)

- Instruction indivisible: réalisée une seule fois par le matériel:
  - Test_and_Set(TAS) : instruction indivisible de consultation et de modification d'un mot mémoire.

état *test_and_set* (var v :état)
/*état est un type contenant les valeurs occupé et libre*/
{test_and_set :=v ;
v :=occupé ;
return (test_and_set) ;
}

Exclusion Mutuelle:solution Multiprocesseur (2)

Protocole entrée{
while(test_and_set(v)=occupé) do /*attendre de façon active*/
}

Sémantique: Si v est libre, test_and_set renvoie libre, et on entre en SC avec v occupé;

Protocole sortie {v :=libre ; }

- Inconvénient: attente active.

Exclusion Mutuelle: Algorithmes de Dekker

- Solutions pour deux processus;
- Chaque processus boucle indéfiniment sur l'exécution de la section critique;
- Les Procédures *entrée* et *sortie* sont interruptibles;
- Etudier les différentes solutions de Dekker (voir TD).

Exclusion Mutuelle: Algorithmes de Peterson

- Solution symétrique pour N processus (généralisation de la solution de Dekker;
- L'interblocage est évité grâce à l'utilisation d'une variable partagée *Tour;*
  - la variable tour est utilisée de manière absolue et non relative;
- L'algortithme et généralisation pour n processus (voir TD).

Solution par attente passive: les verrous

- Le processus demandeur à entrer en SC est dans un état bloqué jusqu'à ce qu'un événement arrive (réception d 'un signal, etc..
- Le verrou est une structure composée de deux champs:
- Type Verrou = struct {état : (ouvert, fermé);
  en_attente : file de processus
  }
- Initialement le verrou est ouvert

Solution par attente passive: les verrous (2)

- Deux procédures sont définies sur les verrous

verrouiller (var v: verrou)
{
si v.état=ouvert
  alors v.état:= fermé
  sinon bloquer le proc
  dans v.en_attente
}

déverrouiller (var v: verrou)
{
si v.en_attente non vide
alors débloquer un proc. de v.en_attente
sinon v.état:=ouvert
}

les verrous : problèmes (1)

- *Verrouiller* est elle même une SC;
- *déverrouiller* doit être ininterruptible,
  - sinon incohérence si elle est interrompue entre le test et le sinon;
  - cette incohérence est passagère, car le système redevient cohérent dès qu 'un nvx proc. verrouille

les verrous : problèmes (2)

- Le verrou selon cette définition permet difficilement l 'utilisation de ressources en *m* exemplaires accessibles par *n* processus;
- un verrou est comparable à une sémaphore binaire que l 'on va définir, puisque il s'appuie sur une file pour débloquer un processus déjà bloqué.

Les sémaphores

- Introduit par Dijkstra en 1965 pour résoudre le problème d'exclusion mutuelle.
- Permettent l'utilisation de m ressources identiques (exple imprimantes) par n processus.
- Un sémaphore est une structure contenant deux champs :
  - Struct {n : entier ;
    en_attente : file de processus
    }

Sémaphores: Définition(1)

- Un sémaphore est une variable globale protégée, c'est à dire on peut y accéder qu'au moyen des trois procédures :
  - I(S, x) : initialiser le sémaphore S à une certaine valeur x;
  - P(S) ; Peut -on passer ?/peut-on continuer?
  - V(S) ; libérer?/vas y?

Sémaphores: définition (2)

- Un *sémaphore binaire* est un sémaphore dont la valeur peut prendre que deux valeurs positives possibles : en générale 1 et 0.
- Un *sémaphore de comptage* : la valeur peut prendre plus de deux valeurs positives possibles.
  - Il est utile pour allouer une ressource parmi plusieurs exemplaires identiques : la valeur est initialisée avec le nombre de ressources.

Sémaphores: Réalisations logicielles

- I(S, x) {S.n :=x ; }
- P(S) /*S.n est tjs modifié par P(S)*/
  { S.n :=S.n -1 ;
  Si S.n < 0 alors bloquer le processus en fin de S.en_attente ;}
- V(S) /*S.n est tjs modifié par V(S)*/
  { S.n :=S.n + 1 ;
  Si S.n <= 0 alors débloquer le processus en tête de S.en_attente ; }

Réalisations logicielles des primitives P et V

- Problème de l'exclusion mutuelle:
  - initialiser S à 1, et la procédure d'entrée est P(S), et la procédure de sortie est V(S)
- P et V sont des primitives plutôt que des procédures car elles sont non interruptibles
  - possible sur monoprocesseur par masquage d'Interruption.

Réalisations logicielles des primitives P et V (2)

- L'initialisation dépend du nombre de processus pouvant effectuer en même temps une même " section critique " :
  - Exemple: m, si on a m imprimantes identiques;
- Cette implémentation donne à chaque fois dans S.n le nombre de ressources libres :
- lorsque S.n est négative, sa valeur absolue donne le nombre de processus dans la file.

Sémaphores: une deuxième implantation logicielle

- Traduction directe de la spécification fonctionnelle:
  - P(S) {
    Si S.n > 0 alors S.n =S.n -1 ;
    Sinon bloquer le processus en fin de S.en_attente ;
    }
  - V(S) {
    Si S.en_attente non-vide alors débloquer le processus en tête de S.en_attente ;
    sinon S.n :=S.n +1 ;}

Sémaphore d'exclusion Mutuelle

Var mutex : sémaphore init. à 1

Processus Pi

Début

..

P(mutex)

SC

V(mutex)

...

Fin.

Sémaphores d'exclusion mutuelle: interblocage

Processus1	Processus2
P(semA) P(semB) SC V(semA) V(semB)	P(semB) P(semA) SC V(semB) V(semA)

Sémaphore de synchronisation: principe

- Un processus doit attendre un autre pour continuer (ou commencer) son exécution.

Processus1	Processus2
1 er travail V(sem)//réveil process 2	P(sem)//attente process1 2eme travail

Exemple:Producteur/ Consommateur

Solution naïve : Attente active

Déclaration communes :
Type messages : suite de caractères ;
Variables communes :
Tampon : tableau [0..N-1] de messages ;
tête, queue: entier init. à 0;

Producteur	Consommateur
Var privée Mess1 : messages ; Répéter Produire (Mess1) ; Tampon[tête] :=Mess1; tête := tête +1 ; Jusqu'à faux ;	Var privée Mess2 : messages ; Répéter Tant que tête = queue faire attente Mess2 := Tampon[queue] ; queue := queue + 1 ; Consommer (Mess2) ; Jusqu'à faux ;

Sémaphores: Producteur/consommateur

Cas :Tampon circulaire :

Déclaration communes :
Type messages : suite de caractères ;
Variables communes :
Tampon : tableau [0..N-1] de messages ;
NPLEIN : sémaphore initialisé à 0 ;
NVIDE : sémaphore initialisé à N ;

Producteur	Consommateur
Var privée Mess1 : messages ; Tête : entier init. 0 ; Répéter Produire (Mess1) ; P(NVIDE) ; Tampon[tête] :=Mess1; tête := tête +1 mod N; V(NPLEIN) ; Jusqu'à faux ;	Var privée Mess2 : messages ; Queue : entier init. 0 Répéter P(NPLEIN) ; Mess2 := Tampon[queue] ; queue := queue + 1 mod N; V(NVIDE) ; Consommer (Mess2) ; Jusqu'à faux ;

Sémaphores privés

- Problème des philosophes: solution 1
- On suppose que les philo. et les Baguettes sont numérotés de 0 à 4.

```
Var commune: Baguette: tableau [0..4] de sémaphores init. à 1;
Processus Philo (i: entier)
Répéter
    Penser;
    P(Baguette [i]);
    P(Baguette [i+1 mod 5]);
    Manger
    V(Baguette [i]);
    V(Baguette [i+1 mod 5]);
Jusqu'à fin de repas
```

Inconvénient: si chaque philo prend une baguette en même temps que les autres (P(baguette[i]), il y a blocage.

Solution: prendre un philo. droitier: solution non symétrique.

Sémaphores privés

- Autre solution: ajouter un sémaphore table init. à 4.

```
Var commune: Baguette: tableau [0..4] de sémaphores init. à 1;
             table: sém. Init à 4;
Processus Philo (i: entier)
Répéter
    Penser;
P(table)
    P(Baguette [i]);
    P(Baguette [i+1 mod 5]);
    Manger
    V(Baguette [i]);
    V(Baguette [i+1 mod 5]);
V(table)
Jusqu'à fin de repas
```

- Inconvénient: sol. ne permet pas un degrés de // maxi. car au pire cas il n' y a qu'un seul philo. qui mange à la fois.

Sémaphores privés

- **Idée de base**: Le philo. i peut manger que si certaines conditions sont vraies, et lui même est affamé et ses deux voisins (i-1 et i+1) ne mangent pas;
- Cette condition de « non suspension » peut être exprimée en utilisant un ensemble de variable d' état sur le système considéré;
- Une condition de « non suspension » n'est attendue que par un seul (chaque) philo. Il suffit d'associer à cette condition (et donc à ce philo) une sémaphore qui sera qualifiée de privée.

Sémaphores privés: schéma général

- L'état global du système est matérialisé par un ens. de variables d'état;
- Un processus Pi doit attendre qu'une condition Ci, soit vérifiée avant de continuer et il est le seul à pouvoir l'attendre;
- Une sémaphore sempriv (i), **initialisé à 0,**est associée à Ci. Seul Pi peut exécuter p(sempriv(i));
- Différents processus peuvent utiliser différentes conditions portant sur le même ensemble de variables d'état;
- Les processus qui consultent ou modifient les var doivent le faire en EM (utilisant une sémaphore Mutex);
- Donc Pi ne peut faire P(sempriv(i)) entre P(Mutex) et V(Mutex).

Sémaphores privés: schéma général

- Vérification/attente de la condition Ci par Pi:

```
P(mutex)
        /*consultation et vérification des variables d'état*/
        Si Ci alors V(sempriv(i))
        fsi;
V(mutex)
P(sempriv(i))
```

- Modification des variables d'état (par Pi ou autre processus

```
P(mutex);
        /*consultation et vérification des variables d'état*/
        Si Pj est bloqué et Cj  alors
                Mise à jour des variables d 'état
                V(sempriv(j));
        fsi;
V(mutex);
```


Sémaphores privés: exple. des philos

```
Var commune: sempriv: tableau [0..4] de sémaphores init. à 0;
             mutex: sém. Init à 1;
             etat-ph: tableau [0..4] de {penser, demande, manger} init. à penser;
Processus Philo (i: entier)
Répéter
P(mutex);
    etat-ph[i]= demande;
    Si (etat-ph[i-1 mod 5] <> manger) et (etat-ph[i+1 mod 5] <> manger)
                alors V(sempriv[i]); etat-ph[i]= manger; fsi
V(mutex);

P(sempriv[i]);
/*Manger*/
P(mutex);
    etat-ph[i]=penser;
    Si (etat-ph[i-1 mod 5] = demande) et (etat-ph[i-2 mod 5] <> manger)
                alors V(sempriv[i-1 mod 5]); etat-ph[i-1 mod 5]= manger; fsi
    Si (etat-ph[i+1 mod 5] = demande) et (etat-ph[i+2 mod 5] <> manger)
                alors V(sempriv[i+1 mod 5]); etat-ph[i+1 mod 5]= manger; fsi

V(mutex);

Jusqu'à fin de repas
```

Sémaphores collectifs

- L'état global du système est matérialisé par un ens. de variables d'état;
- {Pi} un ensemble de processus qui, pendant leur exécution, doivent attendre qu'une condition Ci soit vérifiée avant de continuer;
- Une sémaphore collective i, init à 0, est associé à Ci;
- Différents processus peuvent attendre différentes conditions portant sur le même ensemble de variable d 'état
- Les processus qui consultent ou modifient les var doivent le faire en EM (utilisant une sémaphore Mutex). Donc un élément Pi ne peut faire P(collectif(i)) entre P(Mutex) et V(Mutex).

Super-sémaphores: Exercice (Td)