



Synchronisation

Pr Belkhir A.



BELKHIR ABDELKADER



07/02/202

Synchronisation

système d'interruption (préemption)

Scheduling

compétition pour le partage de ressources limitées

un système de réservation de lignes aériennes

l'agence A voit qu'il y a suffisamment de places libres

.

.

l'agence A réserve des places

l'agence B voit qu'il y a suffisamment de places libres

.

.

l'agence B réserve des places

Problème de la section critique

$\{P_1, P_2, \dots, P_n\}$

Chaque processus dispose d'un segment de code appelé "section critique" dans lequel le processus peut lire des variables communes, met à jour une table et écrit dans un fichier.

Une caractéristique importante de ce système est: quand un processus est dans sa section critique, aucun autre processus ne doit exécuter sa section critique. Ainsi, les exécutions des sections critiques sont mutuellement exclusives dans le temps.

Problème de la section critique

begin

variables communes

PARBEGIN

P0;

P1;

PAREND

end

La structure générale d'un processus est:

Repeat

code d'entrée

section critique

code de sortie

section non critique

Until false

Algorithme1:

turn=0: P0 est autorisé à exécuter sa section critique.

turn=1: P1 est autorisé à exécuter sa section critique.

On utilisera une autre variable j avec $j = 1-i$. Le corps d'un processus P_i avec $i=0,1$ est:

Repeat

 etiq: si $\text{turn} \neq i$ alors aller à étiq;

 section critique

$\text{turn} = j$;

 section non critique

Until false

Avantages :

l'exclusion mutuelle est satisfaite. Pour chaque valeur de tour , une section critique et une seule peut s'exécuter, et ce jusqu'à son terme.

l'interblocage est impossible puisque tour prend soit la valeur 1, soit la valeur 2 (Les deux processus ne peuvent pas être bloqués en même temps).

la privation est impossible: un processus ne peut empêcher l'autre d'entrer en section critique puisque tour change de valeur à la fin de chaque section critique.

Inconvénients :

P1 et P2 sont contraints de fonctionner avec la même fréquence d'entrée en section critique

Si l'exécution de P2 s'arrête, celle de P1 s'arrête aussi: le programme est bloqué. La dépendance de fonctionnement entre P1 et P2 leur confère le nom de coroutines.

Algorithme2

var flag: array[0,1] de type booléen.

Les éléments du vecteur sont initialisés à faux. Si flag[i]=vrai, alors le processus P_i est en exécution de sa section critique.

La structure d'un processus P_i est:

Repeat

 etiq: si flag[j] alors aller à etiq;

 flag[i]=vrai;

 section critique

 flag[i]=faux;

 section non critique

Until false

Avantage :

on rend moins dépendants les deux processus en attribuant une clé de section critique à chacun.

Inconvénients :

Au début c_1 et c_2 sont à 1. P1 prend connaissance de c_2 et met fin à la boucle while.

Si la commutation de temps a lieu à ce moment, c_1 ne sera pas à 0 et P2 évoluera pour mettre c_2 à 0, tout comme le fera irrémédiablement P1 pour c_1 .

La situation $c_1 = c_2 = 0$ qui en résultera fera entrer simultanément P1 et P2 en section critique : l'exclusion mutuelle ne sera pas satisfaite.

Si l'instruction $c_i = 0$ était placée avant la boucle d'attente, l'exclusion mutuelle serait satisfaite, mais on aurait cette fois interblocage.

A un moment donné, c_1 et c_2 seraient nuls simultanément P1 et P2 exécuteraient leurs boucles d'attente indéfiniment.

Algorithme3

Une troisième solution consiste à utiliser une variable qui marque son intention d'entrer en section critique.

Lorsque les deux processus veulent entrer en section critique au même moment, l'un des deux renonce temporairement.

Aucun n'entre en section critique.

Algorithme4: (solution de DEKKER)

flag[0,1] de type booléen turn: 0 ou 1.

Initialement, flag[0]= faux et flag[1]=faux et turn = 0 ou 1.

Repeat

flag[i]=vrai;

tantque flag[j]

faire si turn=j alors flag[i]=faux;

etiq: si turn=j alors aller à etiq;

flag[i]=vrai;

fsi

fait

section critique

turn=j;

section non critique

Until false

Si p_1 veut entrer en section critique ($c_1 = 0$), alors que p_2 le veut aussi ($c_2 = 0$), et que c'est le tour de p_1 ($tour = 1$), p_1 insistera ($while (c_2 == 0)$ sera actif). Dans p_2 , la même boucle aboutira à $c_2 = 1$ (renoncement temporaire de p_2).

- On démontre que cet algorithme résout l'exclusion mutuelle sans privation, sans interblocage, sans blocage du programme par arrêt d'un processus.

- Cet algorithme peut être généralisé à n processus au prix d'une très grande complexité.

-De manière générale, les algorithmes par attente active présentent un défaut commun : les boucles d'attente et le recours fréquent à la variable $tour$ gaspillent du temps UC.

-Nouvelle idée : mettre en sommeil un processus qui demande à entrer en section critique dès lors qu'un autre processus y est déjà. C'est l'attente passive.

Solutions hardwares

L'instruction TEST_and_SET

fonction

```
TEST_and_SET(bool:booléen):booléen
```

```
begin
```

```
    TEST_and_SET = bool;
```

```
    bool = vrai;
```

```
end
```

L'instruction TEST_and_SET

On utilise une variable booléenne lock initialisée à faux.

Repeat

 etiq: si TEST_and_SET(lock) alors aller à
etiq;

 section critique

 lock = faux;

 section non critique

Until false

Sémaphores

Définition:

Un sémaphore est une variable entière qui est accessible uniquement à travers deux opérations atomiques notées P et V.

Soit S un sémaphore:

P(S): etiq: si $S \leq 0$ alors aller à etiq;
 $S = S - 1$;

V(S): $S = S + 1$;

Sémaphores

Repeat

P(mutex);
section critique
V(mutex);
section non critique

Until false

Les sémaphores peuvent être utilisés pour résoudre des problèmes de synchronisation.

P1:

```
.  
.   
.   
S1;  
V(synch);  
.   
.
```

P2:

```
.   
.   
.   
P(synch);  
S2;  
.   
.
```

Sémaphores

Producteur

Repeat

P(N);

produit un objet et le dépose dans la zone tampon

V(O);

Until false

Consommateur

Repeat

P(O);

consomme un objet et libère un espace dans la zone
tampon

V(N);

Until false

Sémaphores et processus

```
type sémaphore = record
    valeur: entier;
    L : liste de processus
end
```

```
P(S): S.valeur = S.valeur - 1;
    si S.valeur < 0
    alors begin
        ajouter ce processus à la liste S.L
        bloquer
    end
    fsi
```

```
V(S): S.valeur = S.valeur + 1;
    Si S.valeur <= 0
    alors
        enlever un processus X de la liste S.L
        réveiller(X)
    fsi
```

Les moniteurs

- Le sémaphore permet le blocage et le réveil explicites de processus
- Le sémaphore permet d'associer un nombre d'autorisations d'accès disponibles à un objet partagé

! Il peut conduire au blocage des processus

Moniteur

Les moniteurs spécifiés par Hoare et Brinch Hansen reposent sur les principes suivants :

- Exclusion mutuelle implicite entre les méthodes d'accès =>file d'attente au module

- Conditions d'accès reposant sur des tests de variables d'état =>file d'attente par condition d'accès

Les moniteurs

- Ils simplifient la mise en place de sections critiques
- Ils sont définis par:
 - des données internes (appelées aussi variables d'état)
 - des primitives d'accès aux moniteurs (points d'entrée)
 - des primitives internes (uniquement accessibles depuis l'intérieur du moniteur)
 - une ou plusieurs files d'attentes

Structure d'un moniteur

Type $m = \textit{moniteur}$

Début

Déclaration des variables locales
(ressources partagées);

Déclaration et corps des procédures du
moniteur

(points d'entrée);

Initialisation des variables locales;

Fin

Les moniteurs

- Seul un processus (ou tâche ou thread) peut être actif à un moment donné à l'intérieur du moniteur
- La demande d'entrée dans un moniteur (ou d'exécution d'une primitive du moniteur) sera bloquante tant qu'il y aura un processus actif à l'intérieur du moniteur

⇒ L'accès à un moniteur construit donc implicitement une exclusion mutuelle

Les moniteurs

- Lorsqu'un processus actif au sein d'un moniteur ne peut progresser dans son travail (une certaine condition est fausse), il libère l'accès au moniteur avant de se bloquer.
- Lorsque des variables internes du moniteur ont changé, le moniteur doit pouvoir « réveiller » un processus bloqué.
- Pour cela, il existe deux types de primitives :
 - wait : qui libère l'accès au moniteur, puis bloque le processus appelant sur une condition
 - signal : qui réveille sur une condition un des processus en attente à l'intérieur du moniteur (un processus qui a exécuté précédemment un wait sur la même condition)

Les variables condition

- **Une variable condition** : est une variable
 - qui est définie à l'aide du type *condition*;
 - qui a un identificateur mais,
 - qui n'a **pas de valeur** (contrairement à un **sémaphore**).
- **Une condition** :
 - ne doit pas être initialisée
 - ne peut être manipulée que par les primitives Wait et Signal.
 - est représentée par une **file d'attente de processus bloqués** sur la même cause;
 - est donc assimilée à sa file d'attente.

Les variables condition

- La primitive *Wait* bloque systématiquement le processus qui l'exécute
- La primitive *Signal* réveille un processus de la file d'attente de la condition spécifiée, si cette file d'attente n'est pas vide; sinon elle ne fait absolument rien.

Les variables condition

- **Syntaxe :**

cond.Wait;

cond.Signal;

/ cond est la variable de type condition déclarée
comme variable locale */*

- **Autre syntaxe :**

Wait(cond) ;

Signal(cond);

- Un processus réveillé par *Signal* continue son *exécution* à l'instruction qui suit le *Wait* qui l'a bloqué.

Producteur-Consommateur à l'aide des moniteurs

Type ProducteurConsommateur = moniteur

{ variables locales }

Var Compte : entier ; *Plein, Vide* : **condition** ;

{ procédures accessibles aux programmes utilisateurs }

Procédure Entry Déposer(message M) ;

Début

si Compte=N alors *Plein.Wait* ;

dépôt(M);

Compte=Compte+1;

si Compte==1 alors *Vide.Signal*;

Fin

Producteur-Consommateur à l'aide des moniteurs

Procedure Entry Retirer(message M) ;

Début

si $\text{Compte}=0$ alors *Vide.Wait* ;

retrait(M);

$\text{Compte}=\text{Compte}-1$;

si $\text{Compte}==N-1$ alors *Plein.Signal*;

Fin

Début {*Initialisations*} $\text{Compte}=0$; **Fin**.

Producteur-Consommateur à l'aide des moniteurs

Processus Producteur

message M;

Début

tant que vrai faire

Produire(M);

ProducteurConsommateur.*déposer(M)*

Fin

Processus Consommateur

message M;

Début

tant que vrai faire

ProducteurConsommateur.*retirer(M)*;

Consommer(M);

Fin