Deuxième partie

Protocoles d'exclusion mutuelle

Contenu de cette partie

- difficultés résultant d'accès concurrents à un objet partagé
- mise en œuvre de protocoles d'isolation
  - solutions synchrones (i. e. bloquantes) : attente active
    - → difficulté du raisonnement en algorithmique concurrente
    - → aides fournies au niveau matériel
  - solutions asynchrones : gestion des processus

Plan

Interférences et isolation

S_1

```
(1) x := lire(compte_2);
(2) y := lire(compte_1);
(3) y := y + x;
(4) ecrire(compte_1, y);
```

S_2

```
(a) v := lire(compte_1);
(b) v := v - 100;
(c) ecrire(compte_1, v);
```

- compte_1 et compte_2 sont partagés par les deux traitements ;
- les variables x, y et v sont locales à chacun des traitements ;
- les traitements s'exécutent en parallèle, et leurs actions peuvent être entrelacées.

(1) (2) (3) (4) (a) (b) (c) est une exécution possible, cohérente.
(1) (a) (b) (c) (2) (3) (4) " " " " "
(1) (2) (a) (3) (b) (4) (c) est une exécution possible, incohérente .

cohérence ⇐ calculs séparés ⇐ exécution séquentielle

Section critique

Définition

Les séquences $S_1 = \texttt{(1);(2);(3);(4)}$ et $S_2 = \texttt{(a);(b);(c)}$ sont des sections critiques, qui sont chacune destinées à être exécutées de manière atomique (indivisible) :

- le résultat de l'exécution concurrente de S_1 et S_2 doit être le même que celui de l'une des exécutions séquentielles $S_1; S_2$ ou bien $S_2; S_1$.
- cette équivalence peut être atteinte
  - en contrôlant directement l'ordre d'exécution de S_1 et S_2 (exclusion mutuelle),
  - ou en contrôlant les résultats (partiels ou finaux) de S_1 et S_2 (contrôle de concurrence).

Accès conflictuels : encore des exemples

Exécution concurrente

```
init x = 0;
⟨ a := x; x := a + 1 ⟩ ‖ ⟨ b := x; x := b - 1 ⟩
⇒ x = -1, 0 ou 1
```

Modifications concurrentes

⟨ x := 0x|00|01| ⟩ ‖ ⟨ x := 0x|02|00| ⟩
⇒ x = 0x0001 ou 0x0200 ou 0x0201 ou 0x0000 ou 1234 !

Cohérence mémoire

```
init x = 0 ∧ y = 0
⟨ x := 1; y := 2 ⟩ ‖ ⟨ printf("%d %d",x,y); ⟩
⇒ affiche 0 0 ou 1 2 ou 1 0 ou 0 2 ...
```


Protocoles d'exclusion mutuelle : contexte

- ensemble de processus concurrents P_i
- variables partagées par tous les processus
  variables privées (locales) à chaque processus
- structure de chacun des processus

```
cycle
      entrée   section critique   sortie
      ⋮
fincycle
```

- hypothèses :
  - vitesse d'exécution non nulle
  - section critique de durée finie

Objectif

Garantir l'exécution en exclusion mutuelle des $\neq$ sections critiques

Protocoles d'exclusion mutuelle : propriétés

- (sûreté) à tout moment, au plus un processus est en cours d'exécution d'une section critique (noté $P_k.excl$)

$$\textbf{invariant } \forall i, j \in 0..N-1 : P_i.excl \wedge P_j.excl \Rightarrow i = j$$

- (vivacité faible) lorsqu'il y a (au moins) une demande ($P_k.dem$), un processus qui demande à entrer sera admis

$$\forall i \in 0..N-1 : (P_i.dem \textbf{ leadsto } \exists j \in 0..N-1 : P_j.excl)$$

- (vivacité forte) si un processus demande à entrer, ce processus finira par obtenir l'accès (son attente est finie)

$$\forall i \in 0..N-1 : P_i.dem \textbf{ leadsto } P_i.excl$$

Plan

Mise en œuvre : moyens

- Solutions directes
  (plutôt synchronisation à grain fin)
  - solutions logicielles : lecture/écriture de variables partagées
    → attente active : tester continûment la possibilité de progresser
  - mécanismes matériels
    - simplifiant l'attente active (instructions spécialisées)
    - évitant l'attente active (masquage des interruptions)
- Recours au service de gestion des activités de l'environnement d'exécution (système d'exploitation...)

Solutions logicielles : premier essai

Algorithme

```
occupé : global boolean := false;
tant que occupé faire nop;
occupé ← true;
    section critique
occupé ← false;
```

Problème
Lecture (test) et écriture (affectation) effectuées séparément
→ invariant invalide

Solutions logicielles : alternance

Deux processus (P_0 et P_1)

Algorithme (code du processus P_i)

```
tour : global 0..1;
tant que tour ≠ i faire nop;
    section critique
tour ← i + 1 mod 2;
```

- lectures et écritures supposées atomiques
- généralisable à plus de 2 processus

Problème
alternance obligatoire

Solutions logicielles : priorité à l'autre demandeur

Deux processus (P_0 et P_1)

Algorithme (code de P_i, avec j = id. de l'autre processus)

```
demande : global array 0..1 of boolean;
demande[i] ← true;
tant que demande[j] faire nop;
    section critique
demande[i] ← false;
```

- lectures et écritures supposées atomiques
- non facilement généralisable à plus de 2 processus

Problème
risque d'attente infinie (interblocage)

Solutions logicielles : Peterson 1981 (1/2)

Deux processus (P_0 et P_1)

Algorithme (code de P_i, avec j = id. de l'autre processus)

```
demande : global array 0..1 of boolean;
tour : global 0..1;
demande[i] ← true;
tour ← j;
tant que (demande[j] et tour = j) faire nop;
    section critique
demande[i] ← false;
```

- lectures et écritures supposées atomiques
- évaluation non atomique du ≪ et ≫
- vivacité forte

Solutions logicielles : Peterson 1981 (2/2)

Exercice

L'ordre des deux premières instructions du protocole d'entrée est-il important ? Pourquoi ?

Idée de la preuve

- sûreté
  - *tour* ne peut avoir qu'une valeur (et *tour* n'est pas modifié dans la section critique)
- vivacité forte
  - si P_i attend, (*demande[j]* et *tour* = j) finit par devenir et rester faux

Solution logicielle pour n processus (Lamport 1974)

L'algorithme de la boulangerie (code du processus P_i)

```
choix : global array 0..N-1 of boolean;
num : global array 0..N-1 of integer;
tour : integer;
choix[i] ← true;
tour ← 0;
pour k de 0 à N faire tour ← max(tour,num[k]);
num[i] ← tour + 1;
choix[i] ← false;

pour k de 0 à N faire
 tant que (choix[k]) faire nop;
 tant que (num[k]≠0) et (num[k],k)≺(num[i],i) faire nop;
section critique
num[i] ← 0;
```

Remarque : autorise des lectures et écriture non atomiques

Solutions matérielles : instructions spécifiques

TestAndSet(x), instruction

- indivisible
- qui positionne x à vrai
- et renvoie l'ancienne valeur de x

Définition

```
function TestAndSet (x : in out boolean) : boolean
     declare oldx : boolean
begin
     oldx := x; x := true; //instruction atomique
     return oldx;
end TestAndSet
```

Solutions matérielles : utilisation du TestAndSet

Algorithme

```
occupé : global boolean := false;
tant que TestAndSet(occupé) faire nop;
  section critique
occupé ← false;
```

Tous les processeurs actuels possèdent une instruction analogue au TestAndSet, et adaptée aux multi-processeurs symétriques.

Question
Ce protocole est-il vivace ?

Solutions matérielles : utilisation de FetchAndAdd

Définition

```
function FetchAndAdd (x : in out int) : int
     declare oldx : int
begin
     oldx := x; x := oldx + 1;
     return oldx;
end FetchAndAdd
```

```
ticket : global int := 0;
tour : global int := 0;
montour : local int;
montour ← FetchAndAdd(ticket);
tant que tour ≠ montour faire nop;
     section critique
FetchAndAdd(tour);
```

Question : ce protocole est-il vivace ?

Solution matérielle sans matériel : utilisation du système de fichiers

Les primitives du noyau Unix permettant la création conditionnelle de fichiers peuvent être utilisées comme des opérations atomiques analogues au TestAndSet.

Algorithme

```
tant que
     open("toto", O_RDONLY | O_EXCL | O_CREAT, 0) == -1
faire nop;
     section critique
unlink("toto");
```

- ne nécessite pas de mémoire partagée
- atomicité assurée par le noyau d'exécution

Peut-on se passer d'attente active ?

Les solutions précédentes sont correctes,
mais présentent un inconvénient sérieux

Attente active

Un processus demandant la section critique et la trouvant occupée doit tester en permanence la possibilité d'entrer en section critique

→ monopolisation ≪ inutile ≫ du (temps) processeur

Piste d'amélioration

Éviter qu'un processus devant attendre en entrée de la section critique répète ces tests ≪ inutiles ≫...

Éviter l'attente active : recours à la gestion des activités

Algorithme

(<< B >> indique que le bloc d'instructions B doit être exécuté en exclusion mutuelle)

```
occupé : global bool := false;
demandeurs : global fifo;
```

```
<< si occupé alors
      self ← identifiant du processus courant
      ajouter self dans demandeurs
      se suspendre
   sinon
      occupé ← true
   finsi >>
    section critique
<< si demandeurs est non vide alors
      p ← extraire premier de demandeurs
      débloquer p
   sinon
      occupé ← false
   finsi >>
```

- accès aux variables globales (*demandeurs*, *occupé*) en exclusion mutuelle
- cette exclusion mutuelle est réalisée par attente active (acceptable, car sections critiques courtes)

Solution matérielle : masquage des interruptions

Idée : réserver le processeur au processus en section critique

Algorithme

```
masquer les interruptions
  section critique
démasquer les interruptions
```

Limite importante :

ne fonctionne que sur les mono-processeurs
→ pas d'entrée-sortie, pas de défaut de page en SC
→ micro-systèmes embarqués

Éviter l'attente active : utilisation des primitives de verrouillage de fichiers

Verrous

- exclusifs (appel système lockf)
- ou coopératifs sur les fichiers : en lecture partagée, en écriture exclusive (appels système : flock, fcntl)

Algorithme

```
fd = open ("toto", O_RDWR);
lockf (fd, F_LOCK, 0); // verrouillage exclusif
    section critique
lockf (fd, F_ULOCK, 0); // déverrouillage
```

- attente passive (le processus est bloqué)
- portabilité aléatoire

En pratique...

- L'attente active ne peut être éliminée, mais est à limiter le plus possible sur un monoprocesseur ; elle peut être utile pour la synchronisation à grain fin sur les multiprocesseurs
- La plupart des environnements d'exécution offrent un service analogue aux verrous, avec les opérations atomiques :
  - obtenir (acquire) : si le verrou est libre, l'attribuer au processus demandeur ; sinon bloquer le processus demandeur
  - libérer (release) : si au moins un processus est en attente du verrou, transférer la possession à l'un des demandeurs et le débloquer ; sinon marquer le verrou comme libre.

Algorithme

```
accès : global verrou // partagé
obtenir accès
     section critique
libérer accès
```


Exercice : exclusion mutuelle vivace avec TestAndSet

Algorithme

```
occupé : global booléen := faux;
attend : global tableau 0..N-1 de booléen := faux;

//Code du processus i
j : local 0..N-1;
attend[i] := vrai;
tant que (attend[i] et TestAndSet(occupé)) faire nop;
     section critique
attend[i] := faux;
j := i+1 % N;
tant que (i ≠ j et non attend[j]) faire j := j+1 % N;
si j = i alors occupé := faux;
          sinon attend[j] := faux ;
```