Deuxième partie

L'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

Trop de pain ? ♪♪♪

Vous

1. Arrivez à la maison
2. Constatez qu'il n'y a plus de pain
3. Allez à une boulangerie
4. Achetez du pain
5. Revenez à la maison
6. Rangez le pain

Votre colocataire

1. Arrive à la maison
2. Constate qu'il n'y a plus de pain
3. Va à une boulangerie
4. Achète du pain
5. Revient à la maison
6. Range le pain

Spécification

Propriétés de correction

- Sûreté : un seul pain est acheté
- Vivacité : s'il n'y a pas de pain, quelqu'un en achète

Que se passe-t-il si

- votre colocataire était arrivé après que vous soyez revenu de la boulangerie ?
- Vous étiez arrivé après que votre colocataire soit revenu de la boulangerie ?
- Votre colocataire attend que vous soyez là pour vérifier s'il y a du pain ?

⇒ *race condition* quand la correction dépend de l'ordonnancement des actions

Solution 1 ?

Vous (processus A)

```
A1. si (pas de pain
        && pas de note) alors
A2.   laisser une note
A3.   aller acheter du pain
A4.   enlever la note
   finsi
```

Colocataire (processus B)

```
B1. si (pas de pain)
        && pas de note) alors
B2.   laisser une note
B3.   aller acheter du pain
B4.   enlever la note
    finsi
```

⇒ deux pains possibles si entrelacement A1.B1.A2.B2...

Solution 2 ?

Vous (processus A)

```
laisser une note A
si (pas de note B) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note A
```

Colocataire (processus B)

```
laisser une note B
si (pas de note A) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note B
```

⇒ zéro pain possible

Solution 3 ?

Vous (processus A)

```
laisser une note A
tant que note B faire
   rien
fintq
si pas de pain alors
     aller acheter du pain
finsi
enlever la note A
```

Colocataire (processus B)

```
laisser une note B
si (pas de note A) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note B
```

Pas satisfaisant

Hypothèse de progression / Solution peu évidente / Asymétrique / Attente active

Interférence et isolation

```
(1) x := lire_compte(2);          (a) v := lire_compte(1);
(2) y := lire_compte(1);          (b) v := v - 100;
(3) y := y + x;                   (c) ecrire_compte(1, v);
(4) ecrire_compte(1, y);
```

- Le compte 1 est partagé 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.

Accès concurrents

Exécution concurrente ♪♪♪

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

Modification concurrente ♪♪♪

```
⟨ 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",y,x); ⟩
⇒ affiche 0 0 ou 2 1 ou 0 1 ou 2 0 !
```

Section critique

Définition

Les séquences $S_1 =$ (1) ; (2) ; (3) ; (4) et $S_2 =$ (a) ; (b) ; (c) sont des sections critiques, qui doivent chacune 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 $S_2; S_1$.
- cette équivalence peut être atteinte en contrôlant directement l'ordre d'exécution de S_1 et S_2 (exclusion mutuelle), ou en contrôlant les effets de S_1 et S_2 (contrôle de concurrence).

≪ Y a-t-il du pain ? Si non alors acheter du pain ; ranger le pain. ≫

L'exclusion mutuelle ♪♪♪

Exécution en exclusion mutuelle d'un ensemble de sections critiques

- ensemble d'activités concurrentes A_i
- variables partagées par toutes les activités
  variables privées (locales) à chaque activité
- structure des activités

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

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

Propriétés du protocole d'accès ♪♪♪

- (sûreté) à tout moment, au plus une activité est en cours d'exécution d'une section critique

$$\text{invariant } \forall i, j \in 0..N-1 : A_i.excl \wedge A_j.excl \Rightarrow i = j$$

- (progression ou vivacité globale) lorsqu'il y a (au moins) une demande, une activité qui demande à entrer sera admise

$$(\exists i \in 0..N-1 : A_i.dem) \rightsquigarrow (\exists j \in 0..N-1 : A_j.excl)$$
$$\forall i \in 0..N-1 : A_i.dem \rightsquigarrow (\exists j \in 0..N-1 : A_j.excl)$$

- (vivacité individuelle) si une activité demande à entrer, elle finira par obtenir l'accès (son attente est finie)

$$\forall i \in 0..N-1 : A_i.dem \rightsquigarrow A_i.excl$$

($p \rightsquigarrow q$: à tout moment, si p est vrai, alors q sera vrai ultérieurement)

Plan

Comment ? ♪♪♪

- Solutions logicielles utilisant de l'attente active : tester en permanence la possibilité d'entrer
- Mécanismes matériels
  - simplifiant l'attente active (instructions spécialisées)
  - évitant l'attente active (masquage des interruptions)
- Primitives du système d'exploitation/d'exécution

Forme générale

```
Variables partagées par toutes les activités
Activité Ai
    entrée
    section critique
    sortie
```


Une fausse solution ♪♪♪

Algorithme

```
occupé : shared boolean := false;

tant que occupé faire nop;
occupé ← true;
    section critique
occupé ← false;
```

(Test-and-set non atomique)

Alternance ♪♪♪

Algorithme

```
tour : shared 0..1;

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

- note : **i** = identifiant de l'activité demandeuse
- deux activités (généralisable à plus)
- lectures et écritures atomiques
- alternance obligatoire

Priorité à l'autre demandeur ♪♪♪

Algorithme

```
demande : shared array 0..1 of boolean;

demande[i]  ← true;
tant que demande[j] faire nop;
    section critique
demande[i] ← false;
```

- **i** = identifiant de l'activité demandeuse
  j = identifiant de l'autre activité
- deux activités (non facilement généralisable)
- lectures et écritures atomiques
- risque d'attente infinie (interblocage)

Peterson 1981 ♪♪♪

Algorithme

```
demande: shared array 0..1 of boolean := [false,false];
tour : shared 0..1;

demande[i] ← true;
tour ← j;
tant que (demande[j] et tour = j) faire nop;
    section critique
demande[i] ← false;
```

- deux activités (non facilement généralisable)
- lectures et écritures atomiques
- évaluation non atomique du « et »
- vivacité individuelle

Solution pour *N* activités (Lamport 1974) ♪♪♪

L'algorithme de la boulangerie

```
int num[N];        // numéro du ticket
boolean choix[N];  // en train de déterminer le n°

choix[i] ← true;
int tour ← 0;  // local à l'activité
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) ∧ (num[k],k)≺(num[i],i) faire nop;
section critique
num[i] ← 0;
```

Instruction matérielle TestAndSet ♪♪♪

Retour sur la fausse solution avec test-and-set non atomique de la variable *occupé* (page 17).

Soit TestAndSet(x), instruction indivisible qui positionne x à vrai et renvoie l'ancienne valeur :

Définition

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

Utilisation du TestAndSet

Alors : protocole d'exclusion mutuelle :

Algorithme

```
occupé : shared 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 multiprocesseurs symétriques.

Instruction FetchAndAdd ♪♪♪

Définition

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

```
ticket : shared int := 0;
tour : shared int := 0;
montour : int;   // local à l'activité
montour ← FetchAndAdd(ticket);
tant que tour ≠ montour faire nop;
     section critique
FetchAndAdd(tour);
```

Spinlock x86 ♪♪♪

Spinlock Linux 2.6

```
; initialement Lock = 1
acquire: lock dec word [Lock]
         jns cs              ; jump if not signed
   spin: cmp dword [Lock], 0
         jle spin            ; loop if ≤ 0
         jmp acquire         ; retry entry
     cs:                     ; section critique
release: mov dword [Lock], 1
```

`lock dec` = décrémentation atomique multiprocesseur avec positionnement du bit "sign"

Masquage des interruptions ♪♪♪

Éviter la préemption du processeur par une autre activité :

Algorithme

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

- plus d'attente !
- mono-processeur seulement
- pas d'entrée-sortie, pas de défaut de page, pas de blocage dans la section critique

→ μ-système embarqué

Le système d'exploitation

1. Contrôle de la préemption
2. Contrôle de l'exécution des activités
3. « Effet de bord » d'autres primitives

Ordonnanceur avec priorités ♪♪♪

Ordonnanceur (scheduler) d'activités avec priorité fixe : l'activité de plus forte priorité s'exécute, sans préemption possible.

Algorithme

```
priorité ← priorité max // pas de préemption possible
   section critique
priorité ← priorité habituelle // avec préemption
```

- mono-processeur seulement
- les activités non concernées sont aussi pénalisées
- entrée-sortie ? mémoire virtuelle ?

→ système embarqué

Éviter l'attente active : contrôle des activités ♪♪♪

Algorithme

```
occupé : shared bool := false;
demandeurs : shared fifo;
bloc atomique
     si occupé alors
          self ← identifiant de l´ activité courante
          ajouter self dans demandeurs
          se suspendre
     sinon
          occupé ← true
     finsi
fin bloc
     section critique
bloc atomique
     si demandeurs est non vide alors
          p ← extraire premier de demandeurs
          débloquer p
     sinon
          occupé ← false
     finsi
fin bloc
```

Le système de fichiers (!)

Pour jouer : effet de bord d'une opération du système d'exploitation qui réalise une action atomique analogue au TestAndSet, basée sur l'existence et la création d'un fichier.

Algorithme

```
tant que
    open("toto", O_RDONLY | O_EXCL | O_CREAT, 0) == -1
    // échec si le fichier existe déjà; sinon il est créé
faire nop;
    section critique
unlink("toto");
```

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

La réalité

Actuellement, tout environnement d'exécution fournit un mécanisme de verrou (*lock*), avec les opérations atomiques :

- obtenir (*acquire*) : si le verrou est libre, l'attribuer à l'activité demandeuse ; sinon bloquer l'activité demandeuse
- rendre/libérer (*release*) : si au moins une activité est en attente du verrou, transférer la possession à l'un des demandeurs et le débloquer ; sinon marquer le verrou comme libre.

Algorithme

```
accès : shared lock
accès.acquire
    section critique
accès.release
```