

Systèmes concurrents

Philippe Quéinnec

30 août 2019



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

1 Interférences entre actions

- Isolation
- L'exclusion mutuelle

2 Mise en œuvre

- Solutions logicielles
- Solutions matérielles
- Primitives du système d'exploitation
- En pratique. . .



Trop de pain ?

Vous

- Arrivez à la maison
- Constatez qu'il n'y a plus de pain
- Allez à une boulangerie
- Achetez du pain
- Revenez à la maison
- Rangez le pain

Votre colocataire

- Arrive à la maison
- Constate qu'il n'y a plus de pain
- Va à une boulangerie
- Achète du pain
- Revient à la maison
- 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)

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

Colocataire (processus B)

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

⇒ deux pains possibles



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

<p>(1) <code>x := lire_compte(2);</code> (2) <code>y := lire_compte(1);</code> (3) <code>y := y + x;</code> (4) <code>ecrire_compte(1, y);</code></p>		<p>(a) <code>v := lire_compte(1);</code> (b) <code>v := v - 100;</code> (c) <code>ecrire_compte(1, v);</code></p>
--	--	---

- 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) (a) (3) (b) (4) (c) est une exécution possible, incohérente.

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

(1) (a) (b) (c) (2) (3) (4) " " " " "



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. »

Accès concurrents

Modification concurrente

$\langle x := 0x\boxed{00}\boxed{01} \rangle \parallel \langle x := 0x\boxed{02}\boxed{00} \rangle$

$\Rightarrow x = 0x0001$ ou $0x0200$ ou $0x0201$ ou $0x0000$ ou 1234 !

Exécution concurrente

init $x = 0$;

$\langle a := x; x := a + 1 \rangle \parallel \langle b := x; x := b - 1 \rangle$

$\Rightarrow x = -1, 0$ ou 1

Cohérence mémoire

init $x = 0 \wedge y = 0$

$\langle x := 1; y := 2 \rangle \parallel \langle \text{printf}("%d %d", y, x); \rangle$

\Rightarrow affiche $0\ 0$ ou $2\ 1$ ou $0\ 1$ ou $2\ 0$!

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

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

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

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

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

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

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


Plan

1 Interférences entre actions

- Isolation
- L'exclusion mutuelle

2 Mise en œuvre

- Solutions logicielles
- Solutions matérielles
- Primitives du système d'exploitation
- En pratique...



Comment ?

- 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



Une fausse solution

Algorithme

```
occupé : global boolean := false;
```

```
tant que occupé faire nop;
```

```
occupé ← true;
```

```
    section critique
```

```
occupé ← false;
```

(Test-and-set non atomique)

Alternance

Algorithme

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

```
tour : global 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

```
boolean choix[N];  
int num[N];  
  
choix[i] ← true;  
int 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) ∧ (num[k], k) < (num[i], i) faire nop;  
  section critique  
  num[i] ← 0;
```

Instruction matérielle

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 : 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 multiprocesseurs symétriques.

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);
```


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 en SC

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

```
bloc atomique  
  si occupé alors  
    self ← identifiant de l' activité courante  
    ajouter self dans demandeurs  
    se suspendre  
  sinon  
    occupé ← true  
  fin si  
fin bloc
```

section critique

```
bloc atomique  
  si demandeurs est non vide alors  
    p ← extraire premier de demandeurs  
    débloquer p  
  sinon  
    occupé ← false  
  fin si  
fin bloc
```

Le système de fichiers (!)

Opération 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
faire nop;
    section critique
    unlink("toto");
```

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



Le système de fichiers (2)

Verrous coopératifs sur les fichiers : en lecture partagée, en écriture exclusive (appels système : lockf, 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 (l'activité est bloquée)
- portabilité aléatoire

La réalité

Actuellement, tout environnement d'exécution fournit un mécanisme analogue aux **verrous** (*locks*), 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 : global verrou // partagé
```

```
obtenir accès
```

```
    section critique
```

```
libérer accès
```