Outils de synchronisation

Module
Systèmes Communicants et Synchronisés

Master Informatique
1ère année

V. Felea & L. Philippe

Contenu

- définition, propriétés
- types de synchronisation
- types d'outils

Besoins

- concurrence des *activités* (processus/threads)
- l'ordre des accès (éventuellement, à une mémoire partagée) peut influencer le résultat final
  - données partagées – les activités concourent pour une ressource → cohérence des données
  - accès – les activités nécessitent à être exécutées dans un ordre → ordonnancement des accès
- gestion : la *synchronisation*

Définition et types de synchronisation

= ensemble de procédures mises en œuvre pour informer/garantir l'état des exécutions les unes par rapport aux autres

- types
  - *compétitive* – une ressource partagée est utilisée
  - *coopérative* – deux ou plusieurs activités asynchrones attendent l'occurrence d'un certain événement

Propriétés d'un système dynamique (évoluant dans le temps)

- liveness (vivacité) – quelque chose de bien arrive fatalement (le programme entre fatalement dans un état souhaitable)
  - un message sera délivré à son destinataire
- safety (sûreté) – quelque chose de mauvais n'arrive jamais (le programme n'entre jamais dans un état inacceptable)
  - incohérence dans les données

Propriétés de la synchronisation

= attribut qui est vrai pour toutes les traces possibles de l'exécution du programme

- starvation free – sans famine (aucun accès demandé n'attend pas à l'infini)
- deadlock free – sans interblocage
- livelock free – sans blocage actif

Famine

- un accès à des ressources est refusé constamment
- sans ces ressources, le programme ne peut pas finir
- Exemple
  - un thread peut attendre indéfiniment un accès à une ressource, car d'autres threads arrivent, demandent la ressource et l'obtiennent avant le thread en question

Famine – exemple (1)

- gestion de l'accès en écriture à un objet
- plusieurs threads d'un processus – tableau
- chaque thread marqué : en attente d'accès ou non
- thread sélectionné pour exécution : recherche du premier thread prêt en parcourant le tableau depuis le début
  - thread *n* cherche à avoir l'accès
  - tout thread < *n* accède à tour de rôle
  - pas de garanti d'accès

Famine – exemple (2)

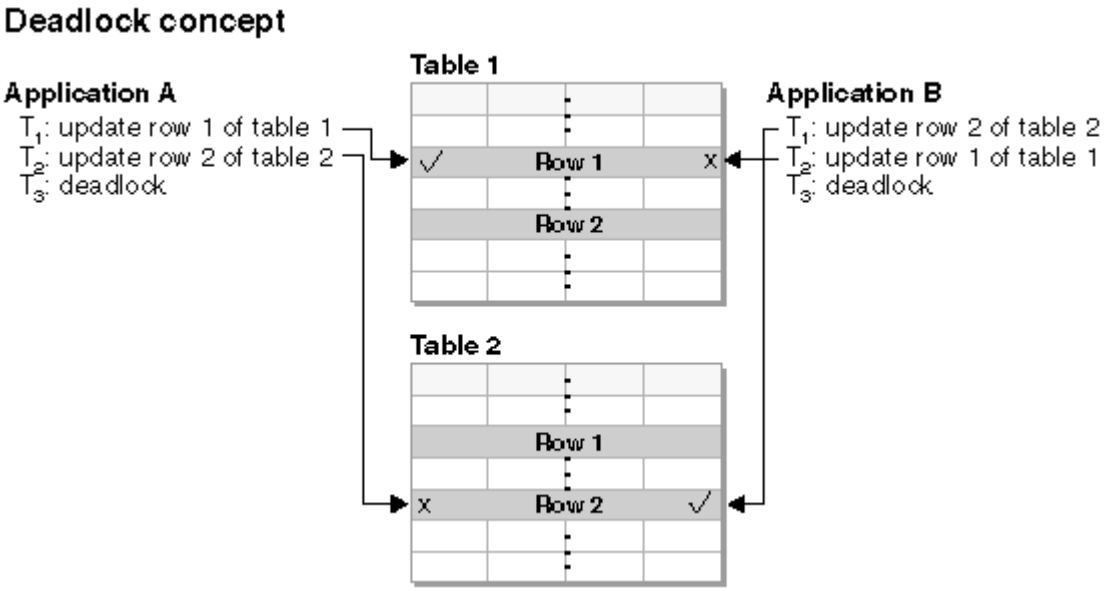
- solution algorithmique
  - changer le parcours du tableau
    - recherche aléatoire
    - point de départ aléatoire
  - créer une file d'attente
- démonstration non famine = vivacité
  - le thread obtiendra un jour l'accès
- pas de respect de l'ordre d'accès (sauf pour le cas d'une file d'attente)

Interblocage (1)

= situation où deux activités ou plus sont bloquées en attente d'un événement qui doit être produit par l'autre

- Exemples
  - transferts entre comptes
  - échange entre deux objets
  - mise à jour des enregistrements dans une base de données

Interblocage – exemple



Interblocage (2)

- plusieurs ressources peuvent être impliquées
- plus il y en a, plus c'est compliqué
- apparition non déterministe
  - dépend du temps
  - bug aléatoire

Interblocage (3)

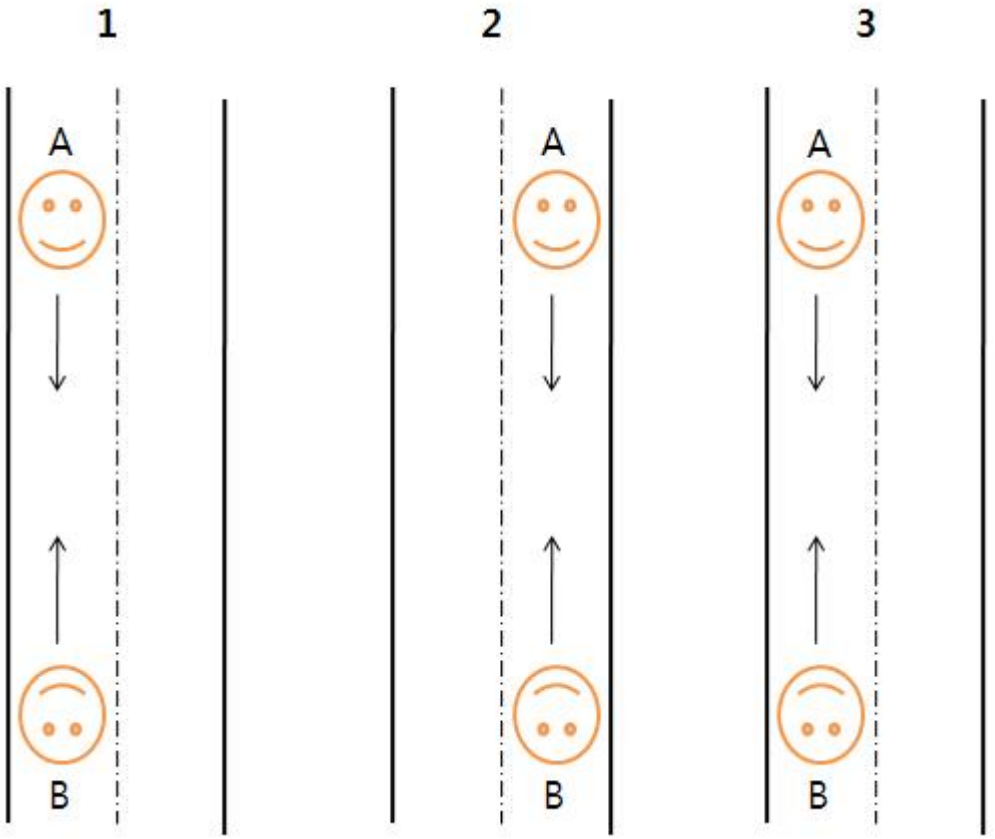
- solutions
  - toujours demander les ressources dans un même ordre
  - demander toutes les ressources simultanément
  - booléens estPriseA, estPriseB ?
- deadlock free
  - graphe des détentions et des accès aux ressources
    - sans cycles

Livelock

= les activités tournent en boucle, sans qu'il y ait de vraie activité réalisée (pas d'avancement)

- Exemple
  - le thread A agit en réponse à une action du thread B et le thread B agit en réponse à une action du thread A
  - deux personnes se cédant la place pour traverser un couloir étroit

Livelock – exemple (1)



Livelock – exemple (2)

A_0

```
prend ress0
tantque !getRess(ress1)
   libère ress0
   ... attend
   prend ress0
fait
prend ress1
```

```
prend ress1
tantque !getRess(ress0)
   libère ress1
   ... attend
   prend ress1
fait
prend ress0
```

A_1

La synchronisation = l'attente

- synchronisation compétitive : attendre qu'une activité, exécutant une section de code dite *critique*, la quitte

A_0

```
attendre avant de pouvoir
  entrer en section critique
  <section de code critique>
libérer la section critique
```

A_1

```
attendre avant de pouvoir
  entrer en section critique
  <section de code critique>
libérer la section critique
```

symétrique

- synchronisation coopérative : attendre qu'une activité rende vraie une condition de progression

A_0

```
attendre une condition
    de progression vraie

poursuivre l'exécution,
    utilisant la ressource
```

A_1

```
mettre la ressource
  dans un état
rendre la
condition de progression vraie
```

asymétrique

Attente active / attente passive

- active
  - test itératif : « dois-je attendre ? »
  - à utiliser de préférence pour des attentes courtes car la condition change rapidement
  - exemples : si la section de code critique est très courte, si l'événement qui rend la condition de progression vraie arrive très fréquemment
- passive
  - test « dois-je attendre » exécuté une seule fois, s'il rend vrai, le processus est bloqué, et attend qu'il soit débloqué ultérieurement

Attentes active / passive - exemple

- un processeur peut exécuter une seule activité à la fois
- toute activité attend jusqu'à ce que la condition « c'est mon tour » soit vraie
  - active : l'activité surveille en permanence si c'était son tour et est capable d'identifier la disponibilité du processeur
  - passive : l'activité est mise en veille, et le processeur, une fois disponible, la réveille pour l'exécuter

Outils de synchronisation

- attente active
  - solutions logicielles : variable de verrouillage, l'alternance, Peterson
  - solutions « matérielles » : Test&Set, Swap
- attente passive
  - verrous
  - sémaphores
  - moniteurs
- cas d'étude : Java

Problématique : synchronisation compétitive

- partage de ressources
  - modification de données
  - accès réservé
- zone accédée par une seule activité à la fois
- exemples
  - accès à une variable (si modification)
  - accès à un fichier (si modification)
  - accès à un périphérique

Objet critique / section critique

- objet critique = objet qui ne peut être accédé simultanément
- section critique (SC) = ensemble de suites d'instructions qui opèrent sur un ou plusieurs objets critiques et qui peuvent produire des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des activités différentes

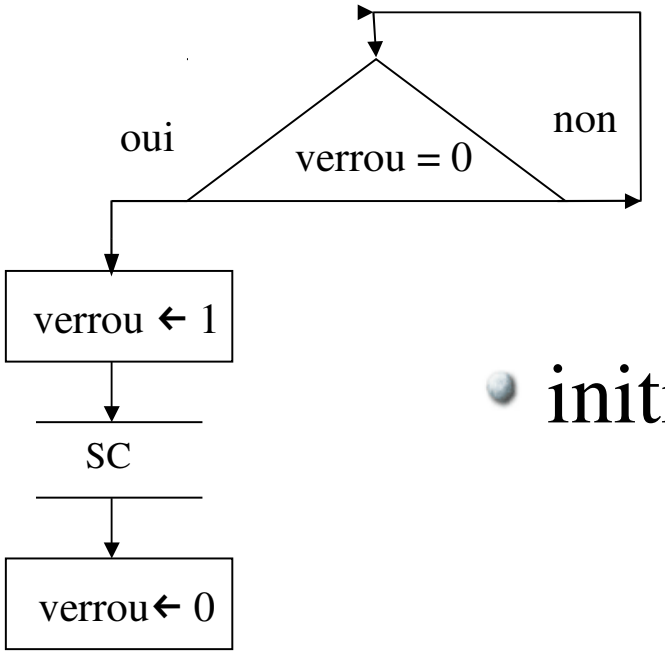
  → assurer que deux activités ne soient pas simultanément en section critique = *exclusion mutuelle*

SC – spécification du comportement

- deux activités ne peuvent être simultanément en section critique → *exclusion mutuelle*
- aucune activité, suspendue en dehors d'une section critique, ne doit bloquer les autres activités → *progression*
- aucune activité ne doit attendre trop longtemps avant d'entrer en section critique → *temps d'attente borné*

- aucune hypothèse ne doit être faite sur les vitesses relatives des activités, leurs priorités et le nombre de processeurs

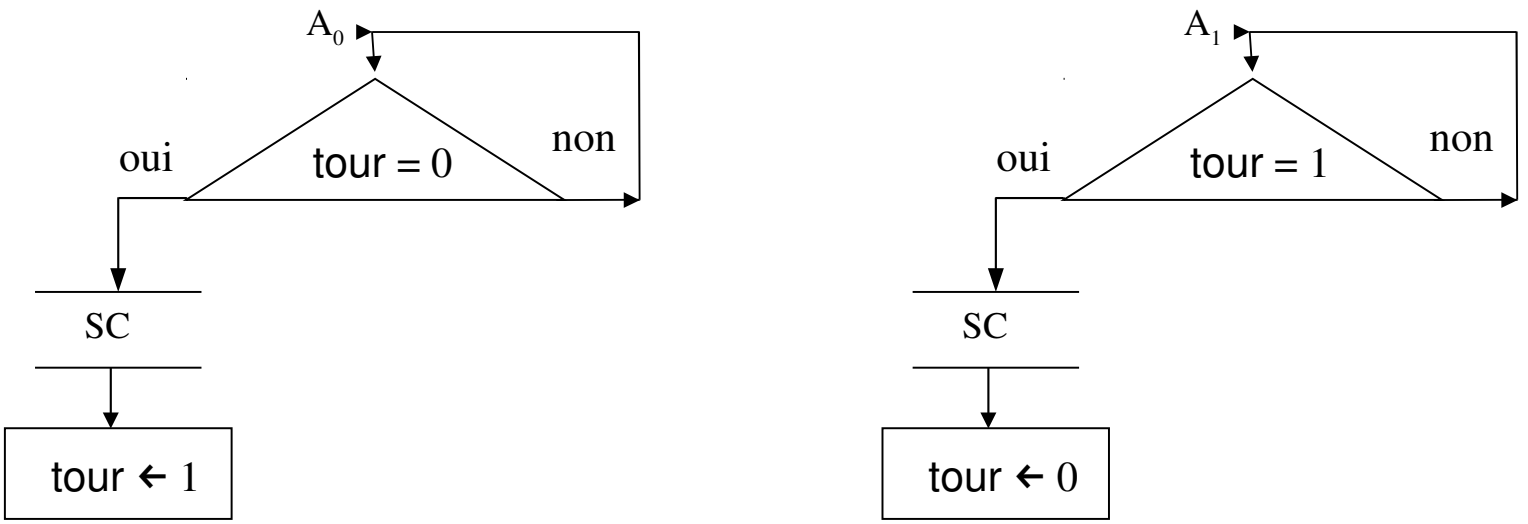
Variables de verrouillage



- initialement, verrou = 0

- problème : solution qui n'assure pas l'exclusion mutuelle (2 activités peuvent être en même temps dans la SC)

L'alternance



- tour
  - mémorise le tour de l'activité qui doit entrer en SC
  - initialisée au début à 0
- problème : solution non valable s'il y a une grande différence de vitesses entre les activités (activité bloquée par une autre qui n'exécute pas sa SC)

Peterson (1)

- principe (1981)
  - variable qui indique à qui le tour
  - des drapeaux qui indiquent qui veut entrer en SC
- pas d'alternance stricte
- solution à deux activités
  - généralisable à *n* activités
- hypothèse
  - opérations atomiques : load, store

Peterson (2)

- données partagées

  int turn ← 0

  int interested[2] ← {faux, faux}

A_i

interested[i] ← vrai

autre ← 1-i

turn ← i

tantque turn = i et interested[autre] fait // attente en cas de conflit

  SC

interested[i] ← faux

Peterson – preuve (1)

- exclusion mutuelle

  si l'activité A_0 est en SC

  interested[0]=vrai

  turn=0

  si l'activité A_1 est en SC

  interested[1]=vrai

  turn=1

- progression

  = si aucune activité n'exécute sa SC et une/des activités souhaitent entrer en SC, la décision n'est pas remise indéfiniment

  - A_0 bloqué : interested[1]=vrai et turn = 0
  - A_1 pas prêt : interested[1] = faux $\rightarrow$ A_0 débloqué
  - A_1 prêt : interested[1] = vrai $\rightarrow$ A_0/A_1 débloqué en fonction de qui fait affecte en dernier turn

Peterson – preuve (2)

- temps d'attente borné

  = après qu'une activité *A* ait fait une demande d'entrer en SC, le nombre de fois que les autres activités sont autorisées d'entrer dans leur SC avant que *A* le fasse, est borné

  - A_0 voudrait entrer en SC : interested[0] = vrai mais bloqué : interested[1]=vrai et turn = 0
  - A_1 dans sa SC
  - A_1 peut entrer à nouveau dans sa SC avant A_0 ?
    - A_1 entre si interested[0] = faux ou turn = 0
    - mais interested[0] = vrai (attente) et turn = 1 (souhaite entrer) $\rightarrow$ A_1 bloqué (et A_0 débloqué)

Attente active : variables de vérouillage / l'alternance

- opérations non *atomiques*
  - exécution des opérations de manière indivisible
  - si cond alors modifVar fsi
- combiner ces approches avec d'autres outils, matériels

Matériel de synchronisation

- des instructions matérielles spéciales qui permettent de tester et modifier le contenu d'un mot de manière atomique
- Test&Set : teste un mot mémoire et lui affecte une valeur, Swap : échange de deux mots mémoire
  - opérations atomiques
  - code assembleur

Test&Set

```
boolean TestAndSet (boolean *target){
    boolean rv = *target;
    *target = true;
    return rv;
}
```

- exclusion mutuelle avec Test&Set
  - variable partagée `boolean lock = false;`

```
do {
    while (TestAndSet(&lock)) ;
      SC
    lock = false ;
      hors SC
} while (true) ;
```

Swap

```
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

- exclusion mutuelle avec Swap
  - variable partagée `boolean lock = false;`

```
do{
    key = true;
    while (key)
        Swap(&lock, &key);
      SC
    lock = false;
      hors SC
} while (true);
```

Attente active : matériel de synchronisation

- boucle infinie pour une activité en attente active
  - inversion des priorités
- exemple
  - deux activités, A_0 et A_1, telles que A_0 est prioritaire, partagent un objet
  - règle d'ordonnancement : A_0 est exécutée dès qu'elle passe en état prêt
  - pendant que A_0 réalise une E/S, A_1 entre en section critique
  - ensuite, A_0 devient prêt lorsque A_1 est toujours en section critique
  - A_1 est donc suspendue au profit de A_0. A_0 effectue une attente active et A_1 ne peut pas être réélue (car A_0 est prioritaire)
  - A_1 ne peut sortir de SC et A_0 boucle indéfiniment

Attente active : inconvénients

consommation CPU !

une activité en attente active peut y rester en permanence !

inversion des priorités !