Problèmes et stratégies de cohérence et de synchronisation

I. Contexte [TOR Chap 14]

Définition 1 **Un Fil d'exécution** est une partie de programme en exécution avec ses données. Il comprend donc la pile et le compteur d'instruction. Les données et le code étant partagées entre tous les fils d'exécutions d'un même programme.

Définition 2 **Des ressources dites partagées** sont communes entres les fils. C'est sur ces données que des stratégies de synchronisation devront être appliqués.

Remarque 3 **Ordonnancement.** De nombreux fils d'éxécutions s'éxécutent en parallèle sur un ordinateur. Le système d'exploitation choisit à chaque instant quels fils a accès au processeur pour s'éxécuter.

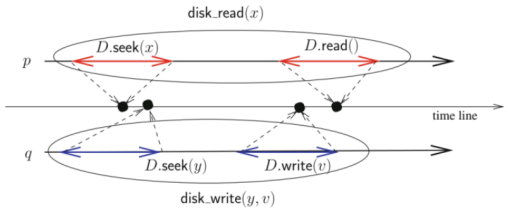
Exemple 4 **Le Tri Fusion** est un algorithme de tri parallélisable. La stratégie de synchronisation consiste ici à lancer des fils sur des parties distinctes du tableau. DEV

A. Conditions de courses (ou de compétition) [RAY 1.2.3]

Définition 5 **Conditions de courses** Ce type d'interaction arrive lorsqu'un résultat dépends de la course de vitesse entre fils d'exécution pour exécuter des instructions. Généralement, lors de l'accès à une ressource partagée.

Exemple 6 **Exemple d'entrelacement**

```
1 fonction lire_disque(pos):
2   deplacer_tete_lecture(pos)
3   r <- lire()
4   renvoie r
```



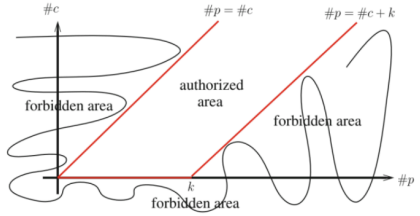
Remarque 7 **Non-déterminisme de l'exécution** Le résultat de l'entrelacement d'instructions provoque du non-déterminisme. En effet, des facteurs extérieurs peuvent influer sur des courses, comme l'ordonnanceur du noyau.

B. Problème de cohérence et de synchronisations [RAY 1.2.4]

Problème 8 **Un rendez-vous** ou barrière de synchronisation est un ensemble de « point de contrôle », un par file d'exécution. Chaque fil a le droit de dépasser ce point de contrôle dans le code quand tous les fils ont atteint leur point de contrôle.

Problème 9 **Le problème du producteur-consommateur** consistes-en le fait d'assurer que toutes les données récupérées par un « consommateur » ont été produites par un « consommateur », chaque donnée doit aussi être traitée exactement une fois.

Exemple 10 **La préservation de certains invariants** nous permet de formaliser le problème du **producteur consommateur**. Avec p, c et k respectivement le nombre d'éléments produit, consommé et la taille du buffer utilisé. On doit toujours avoir $0 \leq c \leq p \leq c + k$.



[RAY Fig 1.3]

Remarque 11 Le problème du producteur-consommateur porte sur la cohérence des données produites et consommées alors que le problème du rendez-vous porte sur la synchronisation sans données partagées.

II. Le problème de l'exclusion mutuelle [RAY 1.3]

A. Les propriétés de sûretés et de vivacité [RAY 1.3.1]

Définition 12 **Section critique** Une section critique est un ensemble de sections de code, qui, pour des raisons de cohérence, ne doivent être exécuté que par un seul fil d'exécution à la fois.

Définition 13 **Le problème de l'exclusion mutuelle** consiste à créer deux algorithmes, **l'algorithme d'entrée** et **l'algorithme de sortie**, pour acquérir (resp relâcher) une section critique et qui, lorsqu'ils encadrent une section critique, assurent sa sûreté et sa vivacité.

Définition 14 **Les propriétés de sûretés (safety)** consistent à énoncer que rien de mal ne va jamais arriver. Elles peuvent être exprimées comme des invariants.

Propriété de sûreté 15 **L'Exclusion-mutuelle** énonce une propriété de sûreté, pour une section de code fixée, qu'au plus un fil d'exécution ne peut avoir acquit la section à la fois.

Définition 16 **Les propriétés de vivacité** (liveness) consistent à énoncer que quelque chose de bien finira par arriver.

Propriété de vivacité 17 **Sans-famine** énonce que si un fil d'exécution veut acquérir une section critique, ce fil finira par l'acquérir.

Propriété de vivacité 18 **Sans-interblocage** énonce que si un fil d'exécution acquiert une section critique, alors, il finira par la relâcher.

Remarque 19 Sans-famine implique sans-interblocage.

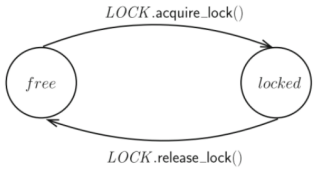
Problème 20 **Le problème de l'exclusion mutuelle** consiste, pour nos deux algorithmes, à assurer les propriétés suivantes :

- Propriété de sûreté: exclusion-mutuelle
- Propriété de vivacité: sans-famine

B. Les Verrous (Mutex) [RAY 1.3.2]

Définition 21 **Un Verrou** est un objet partagé muni des procédures acquérir `lock()` et relâcher `unlock()` qui résout le problème de l'exclusion mutuelle.

Figure 22 **Spécification séquentielle du verrou**



Implémentation C 23 [TOR Chap 14] **API pthread** (`pthread.h`)

```c
int pthread_mutex_lock(pthread_mutex_t *verrou);
int pthread_mutex_unlock(pthread_mutex_t *verrou);
```

III. Implémentations Verrous [RAY 1.3.3, 2]

A. Implémentation avec registre atomique [RAY 2.1]

Définition 24 **Une opération est dite atomique** si sont exécution apparaît comment étant faite de façon instantanée.

Définition 25 **Un registre partagé** est une donnée partagée qui peut être écrite et lu par plusieurs files d'exécutions.

Définition 26 Un registre partagé est dit **atomique** s'il les opérations de lecture et écriture sont atomiques.

Remarque 27 **L'atomicité est importante** car elle permet d'assurer qu'un raisonnement séquentiel va rester correct. Notamment, on remarque que la composition d'objet atomique est toujours atomique, ce qui simplifie les raisonnements.

DEV

Implémentation 28 **Algorithme de Péterson**

```
1 fonction lock():
2   flag[i] <- "up"
3   after_you <- i
4   attendre que (flag[1 - i] != "down")
5         ou que (after_you == i)
```

```
1 fonction unlock():
2   flag[i] <- "down"
```

Remarque 29 **L'attente active** consiste à attendre. On l'écrit par une boucle `while` sur la condition voulue, comme fait par l'algorithme de Peterson.

Remarque 30 Les variables `flag` et `after_you` sont supposés atomiques. L'algorithme de Peterson permet d'implémenter une section critique avec attente active avec cette supposition.

B. Implémentation à l'aide de primitives spécialisées [RAY 2.2]

Définition 31 **Primitives Test&Set.** Si `x` est un registre partagé initialisé à 1, les opérations atomiques de Test&Set sont `x.test&set()` qui met `x` à 0 et renvoie sa valeur précédente et `x.reset()` met 1 dans `x`.

Implémentation 32 **Test & Set sans-interblocage uniquement**

```
1 fonction lock(X):
2   r <- 1
3   tant que (r == 1) faire
```

```
1 fonction unlock(X):
2   X.reset()
```

```
4       r <- X.test&set()
```

Implémentation 33 Test&Set sans-famine. On introduit, en plus de la solution précédente, un mécanisme de tour, assurant un partage du verrou entre les fils.

Remarque 34 Il existe d'autres primitives atomiques comme Compare&Swap, Swap ou encore Fetch&Add.

C. Implémentation sans atomicité [RAY 2.3]

DEV

Implémentation 35 Algorithme de la Boulangerie de Lamport

```
1 fonction lock(me):
2   machine_ticket[me] <- up
3   ticket[me] <- 1 + max {ticket[autre]}
4   machine_ticket[me] <- down
5   pour tout autre:
6     attendre que (machine_ticket[autre] == down)
7     attendre que ((ticket[autre] == 0)
8     ou que (ticket[me], me) < (ticket[autre], autre))
```

Remarque 36 Les algorithmes précédents perdent leur correction lorsque le processeur/compilateur réordonnance les instructions.

IV. Objets concurrents avancés

A. Rendre une structure thread-safe

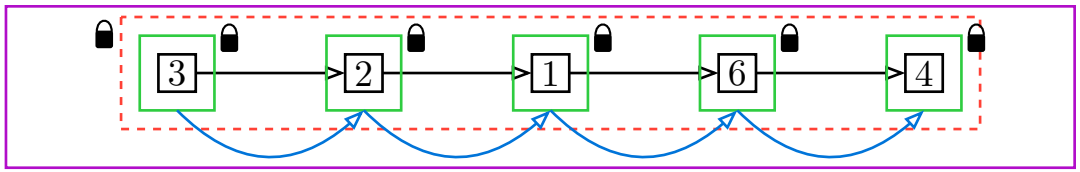
DEV

Exemple 37 Une structure de données est dit concurrence compatible (thread safe) si lorsqu'appelé dans un contexte concurrent elle conservent ses invariants.

Exemple 38 L'implémentation de `malloc` dans la `libc` est apte à la concurrence.

Exemple 39 Pour une liste chainée dans un contexte concurrent différente approche sont possibles. On utilisera un verrou global sur toute la structure, des verrous individuels pour chaque maillon ou on pourra forcer un ordre dans le verrouillage des verrous.

Schémas 40 Visualisation des verrous sur une liste chainée.



B. Construction d'un Sémaphore [RAY 3.2.3]

Définition 41 Sémaphore Une sémaphore S est un compteur partagé muni de la spécification suivante

- S est initialisé avec $s_0 \geq 0$
- S est toujours positif
- l'opération atomique S.down() décrémente S de 1 (bloquant tant que $S = 0$)
- l'opération atomique S.up() incrémente S de 1

Implémentation C 42 [TOR Chap 14] API PTHREAD `semaphore.h`

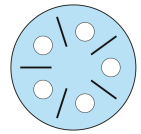
```
1 int sem_wait(sem_t *s);
2 int sem_post(sem_t* s);
```

Problème 43 [RAY 3.2.4] Lecteur-rédacteur consiste à permettre à plusieurs fils d'accéder en même temps à un fichier en lecture tout en assurant à un fil en écriture qu'il est le seul à disposer du fichier.

C. Le dinner des philosophes [TOR Chap 14]

DEV

Probleme 44 Le dinner des philosophes est un problème classique de coordination. Une assemblée de philosophe se regroupe autour d'une table pour manger et partagent leur couvert avec leur voisin. Pour manger ils doivent avoir accès aux deux couverts. L'objectif étant de les faire manger en un temps minimal.



Remarque 45 Une stratégie naïve consiste à utiliser un même mutex sur tous les couverts. Cependant qu'un seul philosophe à la fois peut alors manger au même moment.

Problèmes et stratégies de cohérence et de synchronisation

I. Contexte [TOR Chap 14]

1 Def Un Fil d'exécution
2 Def Des ressources dites partagées
3 Rem Ordonnancement.
4 Ex Le Tri Fusion

A. Conditions de courses (ou de compétition) [RAY 1.2.3]

5 Def Conditions de courses
6 Ex Exemple d'entrelacement
7 Rem Non-déterminisme de l'exécution

B. Problème de cohérence et de synchronisations [RAY 1.2.4]

8 Prob Un rendez-vous
9 Prob Le problème du producteur-consommateur
10 Ex La préservation de certains invariants
11 Rem

II. Le problème de l'exclusion mutuelle [RAY 1.3]

A. Les propriétés de sûretés et de vivacité [RAY 1.3.1]

12 Def Section critique
13 Def Le problème de l'exclusion mutuelle
14 Def Les propriétés de sûretés (safety)
15 Propriété de sûreté L'Exclusion-mutuelle
16 Def Les propriétés de vivacité
17 Propriété de vivacité Sans-famine
18 Propriété de vivacité Sans-interblocage
19 Rem
20 Prob Le problème de l'exclusion mutuelle

B. Les Verrous (Mutex) [RAY 1.3.2]

21 Def Un Verrou
22 Figure Spécification séquentielle du verrou
23 Implémentation C [TOR Chap 14] API pthread (`pthread.h`)

III. Implémentations Verrous [RAY 1.3.3, 2]

A. Implémentation avec registre atomique [RAY 2.1]

24 Def Une opération est dite atomique
25 Def Un registre partagé
26 Def
27 Rem L'atomicité est importante
28 Implem Algorithme de Péterson
29 Rem L'attente active
30 Rem

B. Implémentation à l'aide de primitives spécialisées [RAY 2.2]

31 Def Primitives Test&Set.
32 Implem Test & Set sans-interblocage uniquement
33 Implem Test&Set sans-famine.
34 Rem

C. Implémentation sans atomicité [RAY 2.3]

35 Implem Algorithme de la Boulangerie de Lamport
36 Rem

B. Construction d'un Sémaphore [RAY 3.2.3]

41 Def Sémaphore
42 Implémentation C [TOR Chap 14] API PTHREAD `semaphore.h`
43 Prob [RAY 3.2.4] Lecteur-rédacteur

IV. Objets concurrents avancés

A. Rendre une structure thread-safe [NAN]

37 Ex Une structure de données
38 Ex L'implémentation de malloc
39 Ex
40 Schémas Visualisation des verrous sur une liste chainée.

C. Le dinner des philosophes [TOR Chap 14]

44 Probleme Le dinner des philosophes
45 Rem Une stratégie naïve

Remarque

- dans le [RAY] on traduira « processus » par « fil d'exécution » d'après la définition en début de livre
- cette leçon est plus théorique que la leçon 18.
- cohérence (spatiale) mémoire != Synchronisation (Temporelle)

Le développement tri-fusion mutithread peut-être inséré dans l'introduction.

- essayer de ne pas faire un catalogue

Bibliographie

[TOR] T. Balabonski & S. Conchon & J. Filliâtre & K. Nguyen & L. Sartre, *MP2I MPI, Informatique Cours et exercices corrigés.*

[RAY] M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations.*

Alexis Hamon & Paul Adam