Gestion et coordination de multiples fils d'exécution.

I. Processus

A. Concurrence

Définition 1 Processus [TOR 14.2] Un processus est un programme en exécution caractérisé par son code, registres, les ressources qu'il utilise et l'ensemble de la mémoire allouée a pour exécution, la pile et le tas. Un nombre unique l'identifie: il s'agit du **pid**.

Remarque 2 Il y a bien plus de processus s'exécutant sur un ordinateur que de cœurs de calculs disponibles `ps -a -u -x`. Il faut donc gérer le partage de ces cœurs de calculs entre les processus.

Définition 3 Mémoire d'un processus [TOR 2.2.2] La mémoire d'un processus est principalement divisée en quatre parties:

- La **section texte .data** contenant le code exécutable
- La **section donnée .rodata .bss** contenant les variables globales ou statiques
- Le **tas** contenant les données allouées dynamiquement
- La **pile** contenant les variables locales et les cadres des fonctions

Définition 4 Fil d'exécution Aussi appelé **Thread**, c'est un sous-processus exécutant une partie du programme du processus père. Dans le cas de multiple fils d'exécution, la mémoire est partagée, et l'on parle dans ce cas de processus à **multiples fils d'exécution**.

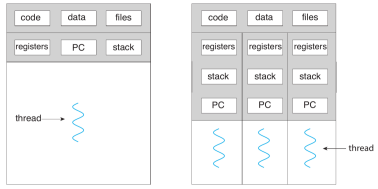


Figure 4.1 Single-threaded and multithreaded processes.

Exemple 5 C'est le cas du navigateur web qui utilise plusieurs fils d'exécutions pour la réception, l'envoie, l'affichage de la page.

Définition 6 Concurrence [TOR 14.1] Deux fils d'exécutions sont dits en concurrence lorsqu'ils sont exécutés en entrelacement, quelques étapes de l'un sont alors effectuées avant quelques étapes de l'autre.

B. Parallélisme

Définition 7 Processeur multicœur Dans le but d'améliorer les performances, plusieurs cœurs, permettant l'exécution simultanée de plusieurs fils d'exécution, peuvent être disposés sur un même puce CPU.

Définition 8 Parallélismes [TOR 14.1] Sur un système possédant plusieurs cœurs, le parallélisme entre deux processus signifie l'assignation des fils d'exécutions sur des cœurs distincts.

Définition 9 Ordonnanceur [TOR 2.3] L'ordonnanceur est un composant du noyau dont le rôle est de gérer la ressource qu'est le temps processeur en donnant/enlevant l'accès à un cœur par un fil d'exécution.

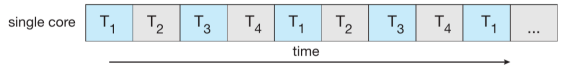


Figure 4.3 Concurrent execution on a single-core system.

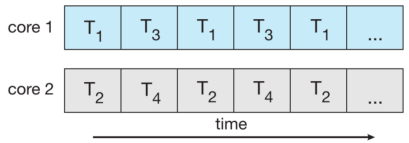


Figure 4.4 Parallel execution on a multicore system.

DEV

Exemple 10 Tri fusion Le tri fusion peut être implémenté à l'aide de multiples fils d'exécutions. Pour ce faire, diviser le tableau en deux créé un nouveau fil d'exécution et la fusion termine un ancien fil d'exécution.

Définition 11 Appel système Il s'agit d'une instruction dont le but est de permettre à un programme de passer temporairement en mode noyau dans l'objectif d'effectuer une instruction privilégiée.

DEV

Exemple 12 Invite de Commande L'explication du fonctionnement d'une invite de commande simple à l'aide d'appels systèmes.

II. Gestion de fil d'exécution

A. Modes d'exécutions

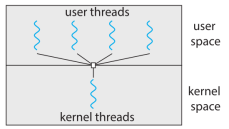
Définition 13 [OSC 21.3] L'exécution d'un programme peut s'effectuer selon deux modes possibles :

- **Utilisateur**, limité sur certaines instructions privilégiées
- **Noyau**, sans aucune restriction de permission

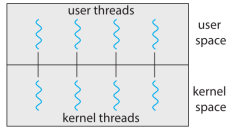
B. Modèles de multiples fils d'exécution

Remarque 14 [OSC 4.3.1] Chaque appel associe un fil d'exécution utilisateur à, au moins, un fil noyau. Il y a alors plusieurs méthodes possibles :

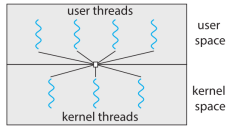
Définition 15 **Association multiple** La première méthode est d'associer plusieurs fils d'exécutions utilisateurs à un unique fil d'exécution noyau.



Définition 16 **Association individuelle** Cette deuxième méthode associe un unique fil noyau à chaque fil d'exécution utilisateur.



Définition 17 **Association hybride** Enfin, une solution hybride propose une association plus complexe où plusieurs fils d'exécutions utilisateurs sont associés à plusieurs fils d'exécutions noyaux.



Remarque 18 Chacune a ces avantages et inconvénients :

- L'association multiple est facile d'utilisation, mais peu efficace en pratique, car limitée à la concurrence.
- L'association individuelle est plus efficace, mais amène plus de problèmes de synchronisation (registres non atomiques).
- L'association hybride est flexible et dynamique et répond à tous les besoins, mais est difficile à implémenter

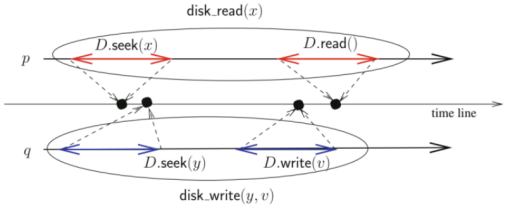
Remarque 19 En pratique, l'association individuelle est la plus couramment utilisée.

C. Problèmes en absence de coordination [RAY 1.2.3]

Définition 20 **Conditions de compétition** 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 21 **Exemple d'entrelacement**

```
fonction lire_disque(pos):          Python
  deplacer_tete_lecture(pos)
  r <- lire()
  renvoie r
```



Remarque 22 **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.

Problème 23 **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.

Définition 24 **Section critique** [RAY 1.3.1] 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 25 **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.

Propriété 26 **Famine** se produit lorsqu'un fil d'exécution qui attend une section critique attend indéfiniment.

Propriété 27 **Interblocage** se produit quand les attentes de sections critiques de plusieurs fils d'exécutions forment un cycle, bloquant tout progrès de ces derniers.

III. Primitives de synchronisation

A. Les Verrous (Mutex) [RAY 1.3.2]

Définition 28 **Un Verrou** est un objet partagé muni des procédures acquérir `lock()` et relâcher `unlock()` qui résout le problème de l'exclusion mutuelle.

1. Implémentation avec registre atomique [RAY 2.1]

Définition 29 **Une opération est dite atomique** si son exécution apparaît comment étant faite de façon instantanée.

Définition 30 **Un registre partagé** est une donnée partagée qui peut être écrite et lue par plusieurs files d'exécutions.

Définition 31 Un registre partagé est dit **atomique** si les opérations de lecture et écriture sont atomiques.

Remarque 32 **L'atomicité est importante** car elle permet d'assurer qu'un raisonnement séquentiel va rester correct. Notamment, on remarque que la composition d'objets atomiques est toujours atomique, ce qui simplifie les raisonnements.

Implémentation 33 **Algorithme de Péterson** DEV

Remarque 34 **L'attente active** consiste à attendre sans arrêter les calculs. On l'écrit par une boucle `while` sur la condition voulue, comme fait par l'algorithme de Peterson.

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

Définition 35 **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 36 **Test & Set** On peut implémenter un verrou à l'aide de ces primitives.

3. Implémentation sans atomicité [RAY 2.3]

Implémentation 37 **Algorithme de la Boulangerie de Lamport** DEV

```python
fonction lock(me):
  machine_ticket[me] <- up
  ticket[me] <- 1 + max {ticket[autre]}
  machine_ticket[me] <- down
  pour tout autre:
    attendre que (machine_ticket[autre] == down)
    attendre que ((ticket[autre] == 0)
    ou que (ticket[me], me) < (ticket[autre], autre))
```

Remarque 38 Les algorithmes précédents perdent leur correction lorsque le processeur/compilateur réordonne les instructions.

4. Implémentation via Bibliothèques [TOR Chap 14]

Implémentation C 39 **API PTHREAD d'exécution (`pthread.h`)**

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

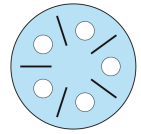
Implémentation OCaml 40 **Le module `Mutex` :**

```ocaml
val Mutex.create : unit -> Mutex.t
val Mutex.lock : Mutex.t -> unit
val Mutex.unlock : Mutex.t -> unit
```

Remarque 41 **Implémentation sans attente active** Les implémentations des bibliothèques utilisent le noyau afin de réveiller des fils d'exécution au relâchement du verrou, sans attente active.

B. Construction d'un Sémaphore [RAY 3.2.3]

Probleme 42 **Le dîner des philosophes** [TOR Chap 14] 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.



Définition 43 **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

Résolution 44 **Problème du dîner des philosophes** à l'aide des opérations S.down() et S.up() des sémaphores.

Résolution 45 **Problème du producteur-consommateur**

Gestion et coordination de multiples fils d'exécution.

I. Processus

A. Concurrence

1 Def Processus[TOR 14.2]
2 Rem
3 Def Mémoire d'un processus[TOR 2.2.2]
4 Def Fil d'exécution
5 Ex
6 Def Concurrence [TOR 14.1]

B. Parallélisme

7 Def Processeur multicœur
8 Def Parallélismes [TOR 14.1]
9 Def Ordonnanceur [TOR 2.3]
10 Ex Tri fusion
11 Def Appel système
12 Ex Invite de Commande

II. Gestion de fil d'exécution

A. Modes d'exécutions

13 Def [OSC 21.3]

B. Modèles de multiples fils d'exécution

14 Rem [OSC 4.3.1]
15 Def Association multiple
16 Def Association individuelle
17 Def Association hybride
18 Rem
19 Rem

C. Problèmes en absence de coordination [RAY 1.2.3]

20 Def Conditions de compétition
21 Ex Exemple d'entrelacement
22 Rem Non-déterminisme de l'exécution
23 Prob Le problème du producteur-consommateur
24 Def Section critique [RAY 1.3.1]
25 Def Le problème de l'exclusion mutuelle
26 Prop Famine
27 Prop Interblocage

III. Primitives de synchronisation

A. Les Verrous (Mutex) [RAY 1.3.2]

28 Def Un Verrou
29 Def Une opération est dite atomique
30 Def Un registre partagé
31 Def
32 Rem L'atomicité est importante
33 Implem Algorithme de Péterson
34 Rem L'attente active
35 Def Primitives Test&Set.
36 Implem Test & Set
37 Implem Algorithme de la Boulangerie de Lamport
38 Rem
39 Implémentation C API PTHREAD d'exécution (`pthread.h`)
40 Implémentation OCaml Le module `Mutex` :
41 Rem Implémentation sans attente active

B. Construction d'un Sémaphore [RAY 3.2.3]

42 Probleme Le dîner des philosophes [TOR Chap 14]
43 Def Sémaphore
44 Résolution Problème du dîner des philosophes
45 Résolution Problème du producteur-consommateur

Au programme

Programme 46

- Prepa :
  - Notions de processus
  - Notions de fils d'éxécution
  - Les concepts sont illustrés sur des schémas de synchronisation classiques : rendez-vous, producteur-consommateur. Les étudiants sont également sensibilisés au non-déterminisme et aux problèmes d'interblocage et d'équité d'accès, illustrables sur le problème classique du dîner des philosophes.
- Complémentaire : Concurrence : modèles de cohérence (forte, faible, PRAM et au relâchement) et d'équité. Construction des mutex et sémaphores à partir des instructions atomiques test and set. Schéma lecteurs rédacteurs.

Bibliographie

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

[OSC] A. Silberschatz & P. B. Galvin & G. Gagne, *Silberschatz's Operating System Concepts, Global Edition.*

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

Adrien Decosse & Alexis Hamon