Slide 1

3 Verrous et barrières

Objectif du chapitre = expliquer comment deux mécanismes clés de synchronisation peuvent être mis en oeuvre

1. Section critique = pour assurer qu'une séquence d'actions arbitraires est exécutée de façon atomique
2. Barrière = point de synchronisation devant être atteint par un groupe de processus avant que n'importe quel processus puisse poursuivre son exécution

Mise en oeuvre typique des sections critiques = utilisation d'un cadenas qui protège l'entrée dans la section critique

Différentes mises en oeuvres des cadenas sont possibles selon le langage, la machine, etc.:

- À l'aide de l'instruction `await` (notation d'Andrews)
- À l'aide de *spinning locks* (niveau machine)

Slide 2

3.1 Le problème de la section critique

Vieux problème de la programmation concurrente (*le* problème classique)

Soit une série de processus désirant obtenir un accès *exclusif* à une section critique (e.g., pour accéder de façon exclusive à une ressource partagée) et ayant la forme suivante:

```
process CS[i = 1 to n] {
   while(true) {
     protocole d'entree;
     section critique;
     protocole de sortie;
     section non-critique;
   }
}
```

Slide 3

Tâche = définir des protocoles d'entrée et de sortie ayant les propriétés suivantes:

1. Exclusion mutuelle = au plus un processus à la fois exécute la section critique
2. Absence de *deadlock* = si 2 ou plusieurs processus tentent d'entrer dans la section critique, au moins un va réussir
3. Absence de délais inutiles = si un processus tente d'accéder à la section critique et pas les autres, rien ne devrait empêcher le premier processus d'y entrer
4. Entrée éventuelle = éventuellement, un processus qui tente d'entrer la section critique devrait y parvenir

Slide 4

Pseudo solution:

```
process CS[i = 1 to n] {
   while(true) {
     < section critique; >
     section non-critique;
   }
}
```

Évite de résoudre le problème: comment "`<...>`" est-il réalisé?

Deuxième solution possible pour 2 processus: en utilisant une instruction `await` simple: Figure 3.1 (p. 97)

Désavantage: pour k processus $\Rightarrow$ k variables `in1`, `in2`, ..., `ink`

3.2 Sections critiques: *spin locks*

Solution pour 2 processus mais n'utilisant qu'une seule variable `lock` et facilement généralisable à n processus: Figure 3.2 (p. 98)

3.2.1 *Test-and-set*

Slide 5

Comment réaliser l'instruction atomique suivante?

```
<await (!lock) lock = true>
```

Réponse: la plupart des machines modernes ont une ou des instructions spéciales permettant la mise en oeuvre d'une telle instruction, par exemple, *Test-and-set* (TS)

Informellement: `TS(lock)` = de façon atomique, la valeur de `lock` est lue et sauvée, la valeur `true` lui est affectée, puis la valeur sauvée est retournée

Plus formellement, dans la notation d'Andrews, TS peut être décrit comme suit:

```
bool TS( bool lock ) {
  < bool initial = lock; /* Sauver valeur initiale. */
    lock = true;         /* Barrer cadenas. */
    return initial;      /* Retourner valeur initiale. */
  >
}
```

Slide 6

Entrée dans un section critique avec TS et conduisant à l'utilisation d'un *spin lock*:

```
bool lock = false;

process CS[i = 1 to n] {
  while (true) {
    while (TS(lock)) skip;       /* Spinning! */
    section critique;
    lock = false;
    section non-critique;
  }
}
```

Slide 7

3.2.2 Test and Test and Set

Solution précédente avec TS inefficace dans une machine multi-processeurs si plusieurs processus tentent d'accéder à la solution critique:

- La variable `lock` est un point chaud (*hot spot*) $\Rightarrow$ contention mémoire
- Les processus écrivent constamment dans la variable `lock`, même si le cadenas est barré ($\Rightarrow$ invalidation fréquence des caches)

Solution alternative = modification du protocole d'entrée pour utiliser un style *double (multiple) checks*, qui aide à ce que la valeur retournée par TS soit `true`:

```
while (lock) skip;
while (TS(lock)) {
  while (lock) skip;
}
```

Slide 8

3.3 Sections critiques: solutions équitables

Désavantage de la solution avec *spin locks* = requiert une mise en oeuvre *fortement* équitable de l'ordonnancement (*scheduling*) des processus

Or, la plupart des solutions efficaces pour l'ordonnancement ne sont pas fortement équitables, mais seulement *faiblement* équitables

D'autres algorithmes existent pour assurer l'accès exclusif à une section critique qui ne dépendent pas de l'utilisation d'une approche fortement équitable du *scheduling*:

- Algorithme de Peterson (*tie breaker algorithm*): pas facile à généraliser pour n processus
- Algorithme du *ticket*: requiert l'utilisation d'une instruction atomique `Fetch-and-add`
- Algorithme de la boulangerie: ne nécessite aucune instruction machine spéciale (mais plus complexe que celui du *ticket*)

Slide 9

3.4 Barrières de synchronisation

Structure typique d'un programme parallèle itératif:

```
while(true) {
  co [i = 1 to n] {
     code qui realise la tache i;
  }
}
```

Désavantage: crée `n` processus à chaque itération, donc coûteux

Solution préférable = créer les `n` processus au début puis les faire synchroniser à la fin de chaque itération $\Rightarrow$ *barrière* de synchronisation

```
process Worker[i = 1 to n] {
  while (true) {
    code qui realise la tache i;
    attendre que toutes les n taches aient complete;
  }
}
```

Différentes mises en oeuvre des barrières sont possibles

Slide 10

3.4.1 Compteur partagé

Utilisation d'un compteur qui indique combien de processus sont rendus à la barrière:

```
int count = 0;

process Worker[i = 1 to n] {
  while (true) {
    code qui realise la tache i;
    < count = count + 1; >
    < await (count == n); >
  }
}
```

Désavantages:

- Contention de la mémoire pour l'accès à `count`
- `count` doit être remis à `0` après chaque itération, ce qui doit aussi être fait *avant* que les processus débutent la prochaine itération

3.4.2 Drapeaux et coordonnateurs

Solution où un processus coordonnateur s'assure que tous les processus `Worker` sont rendus à la barrière et qui envoie des signaux leur disant de poursuivre au-delà de la barrière: Figure 3.12 (p. 119)

Désavantages de cette solution:

Slide 11

- Nécessite un processus supplémentaire
- Chaque itération du coordonnateur requiert un temps proportionnel au nombre de processus à synchroniser. Or, ces derniers processus sont généralement semblables $\Rightarrow$ ils vont tous terminer en même temps $\Rightarrow$ accès à la barrière sera quasi-séquentielle (goulot d'étranglement)

Solution alternative: intégrer la tâche de coordination dans chacun des processus et les organiser à l'aide d'un arbre (binaire) de synchronisation (pp. 119–120)

3.5 Algorithmes avec parallélisme de données

Parallélisme de données $\Rightarrow$ plusieurs processus exécutent le même code mais travaillent sur des parties différentes des données

Utiles pour . . .

Slide 12

- les machines SIMD
- les machines MIMD lorsque les processus sont assez gros pour compenser pour le coût des barrières de synchronisation

3.5.1 Calcul de préfixes parallèles

Exemple = calculer, en parallèle, la somme de tous les préfixes d'un tableau

```
Entree:                          Sortie:
a = 10 20 30 40 50 60            r = 10 30 60 100 150 210
```

Donc:

$$\mathtt{r[i]} = \sum_{\mathtt{k}=0}^{\mathtt{i}} \mathtt{a[k]} \quad (\text{pour } i = 0, \ldots, n-1)$$

Slide 13

Remarque: Les préfixes parallèles peuvent être calculés pour n'importe quelle opération binaire commutative et associative, par ex., +, *, MAX

Stratégie itérative parallèle pour calculer la somme de *tous* les éléments du tableau

- On commence par calculer, en parallèle, la somme des éléments adjacents
- On combine ensuite, toujours en parallèle, la somme des éléments obtenus à l'étape précédente ... et on répète

Adaptation de cette idée pour calculer la somme de tous les préfixes:

```
sum[i] = a[i];
POUR k allant de 0 a (log n)-1 FAIRE
  sum[i] = sum[i] + sum[i-2^k]
```

Effet:

```
Valeur initial de a          1  2  3   4   5   6
sum apres distance de 1      1  3  5   7   9  11
sum apres distance de 2      1  3  6  10  14  18
sum apres distance de 4      1  3  6  10  15  21
```

Solution d'Andrews: Figure 3.17 (p. 126)

Slide 14

3.5.2 Opérations sur les listes chaînées

Voir introduction du cours: trouver le dernier élément d'une liste

3.5.3 Calculs de grilles: Itérations de Jacobi

De nombreux problèmes de calculs scientifiques peuvent être résolus à l'aide de calculs sur des grilles (matrices) où l'on converge, petit à petit, vers la bonne solution

Structure séquentielle de l'algorithme:

```
initialiser la matrice;
while( pas encore termine ) {
  calculer une nouvelle valeur pour chacun
     des points de la matrice;
  verifier la terminaison;
}
```

Note: Le calcul de la nouvelle matrice dépend généralement de la matrice actuelle

Slide 15

En termes de parallélisme de données, on obtient la structure suivante, pour une matrice $n \times n$:

```
process Grid[i = 1 to n, j = 1 to n] {
  while( pas encore termine ) {
    newgrid[i, j] = calcul de la nouvelle valeur;
    verifier la convergence (pour terminaison);
    barriere;
    grid[i, j] = newgrid[i, j];
    barriere;
   }
}
```

Cet algorithme pourrait être amélioré de plusieurs façons:

- Plutôt que de copier à chaque itération, on alterne entre deux copies
- La granularité est trop fine pour la plupart des machines existantes $\Rightarrow$ il faudrait utiliser une stratégie par blocs, où chaque bloc est attribué à un processeur

Slide 16

3.6 Calculs parallèles avec des sacs de tâches

Tâche = unité *indépendante* de travail

Sac de tâches = structure de données où une tâche prête à s'exécuter est conservée jusqu'à ce qu'un processeur la sélectionne

Code exécuté par *chacun* des processus

```
while( true ) {
  obtenir une tache du sac;
  if( il ne reste plus de tache )
    break;
  executer la tache obtenue,
     possiblement en generant de nouvelles taches et
     en les ajoutant dans le sac
 }
```

Avantages de l'approche avec sac de tâches:

- Facile à utiliser
- Indépendant du nombre de processeurs et *scalable*
- Facilite la distribution de la charge (*load balancing*)

Sac de tâches et parallélisme récursif: tâche = appel récursif à exécuter