2 Processus et synchronisation

2.1 États, actions, traces, et propriétés

Slide 1

État d'un processus = valeurs des variables du programme à un certain instant, y compris les variables implicites (e.g., compteur d'instruction)

Action atomique = action qui examine ou change l'état de façon *indivisible*

Un processus exécute une séquence d'instructions. Chaque instruction est elle-même mise en oeuvre par une séquence d'une ou plusieurs *actions atomiques*

Effet de l'exécution d'un programme = *entrelacement* des diverses actions atomiques effectuées par chacun des processus

Slide 2

Trace d'un programme concurrent = séquence spécifique d'actions atomiques produite par une certain exécution d'un programme.

Chaque exécution possible produit une trace possiblement distincte.

Rôle des opérations de synchronisation = exclure certaines traces indésirables

1. Exclusion mutuelle = combiner des séquences d'actions atomiques en *sections critiques* de façon à les faire apparaître indivisibles (i.e., atomiques, donc les éléments internes ne peuvent pas être entrelacés avec d'autres séquences)
2. Synchronisation conditionnelle = retarder une action jusqu'à ce que l'état du processus satisfasse une certaine condition

Slide 3

Exemple Illustration des notions de trace, entrelacements et synchronisation: `hello_world` (Memo 39, p. 11) exécuté sur 3 processeurs

```
Processeur 0
------------
INVOKE( 0, hello_world ) /* I1 */
INVOKE( 1, hello_world ) /* I2 */
INVOKE( 2, hello_world ) /* I3 */
TERMINATE;               /* I4 */

printf( ... )            /* I5 */
TERMINATE;               /* I6 */


Processeur 1                        Processeur 2
------------                        ------------
printf( ... );    /* I7 */          printf( ... );    /* I9 */
TERMINATE;        /* I8 */          TERMINATE;        /* I10 */
```

Slide 4

Contraintes implicites de synchronisation (invocations):

```
I1-I4 < I5          (fibre non-preemptive)
I2    < I7          (invocation)
I3    < I9          (invocation)
```

Note: `Ii < Ij` $\Rightarrow$ l'instruction `Ii` doit s'exécuter *avant* `Ij`

Contraintes de synchronisation liées aux invocations de fonction: si une instruction `Ii` contient une invocation de fonction (`INVOKE`), alors la première instruction (`Ij`) de la fonction s'exécute nécessairement *après* `Ii`

Quelques traces possibles d'exécution du programme:

```
I1 - I2 - I3 - I4
I1 - I2 - I7 - I8 - I3 - I9 - I10 - I4
I1 - I2 - I7 - I3 - I8 - I4
I1 - I2 - I3 - I9 - I10 - I4
I1 - I2 - I3 - I9 - I7 - I8 - I10 - I4
. . .
```

Slide 5

Propriété d'un programme concurrent = attribut qui est vrai de *toutes* les traces possibles du programme

1. Propriété de sécurité = *nothing bad will ever happen*
2. Propriété de vivacité = *something good will eventually happen*

Exemples de propriétés

- Correctitude partielle = si le programme se termine, alors son état final sera correct (sécurité)
- Terminaison = le programme va se terminer (vivacité)
- Correctitude totale = correctitude partielle + terminaison
- Exclusion mutuelle entre processus pour accès à une section critique (sécurité)
- Un processus pourra éventuellement entrer dans une section critique (vivacité)
- Absence de *deadlock* (sécurité)

Slide 6

2.2 Parallélisation: recherche de motifs dans un fichier

Exemple (p. 45) Trouver les différentes instances d'un motif dans un fichier

```
string ligne;
lire une ligne d'entree de stdin dans ligne;
while (!EOF) {
  if (pattern est present dans ligne)
    ecrire ligne;
  lire la prochaine ligne d'entree de stdin dans ligne;
}
```

Exigence fondamentale pour qu'un programme puisse être parallélisé = doit contenir des parties *indépendantes* les unes des autres

Slide 7

Indépendendance entre processus parallèles L'*ensemble de lecture* (*read set*) d'une partie d'un programme est l'ensemble des variables lues mais non modifiées. L'*ensemble d'écriture* (*write set*) d'une partie d'un programme est l'ensemble des variables modifiées (et possiblement lues). Deux parties d'un programme sont *indépendantes* si l'intersection de leurs ensembles d'écriture est vide.

Exemple (p. 46) Contient une dépendance entre la ligne traitée et la ligne lue, donc l'interférence entre les deux processus:

```
string ligne;
lire une ligne d'entree de stdin dans ligne;
while (!EOF) {
  co rechercher pattern dans ligne;
     if (pattern est present dans ligne)
      ecrire ligne;
  // lire la prochaine ligne d'entree de stdin dans ligne;
  oc;
}
```

Slide 8

Solution avec *double buffering* = utilisation d'une ligne auxiliaire et échange de rôles (pp. 46–47)

```
string ligne1, ligne2;
lire une ligne d'entree de stdin dans ligne1;
while (!EOF) {
  co rechercher pattern dans ligne1;
     if (pattern est present dans ligne1)
      ecrire ligne1;
  // lire la prochaine ligne d'entree de stdin dans ligne2;
  oc;
  ligne1 = ligne2;
}
```

Désavantage de cette solution (`co` *inside* `while`) = création de processus à chacune des itérations $\Rightarrow$ coûteux!

Solution alternative = `while` *inside* `co` (Figure 2.1, p. 48) = producteur/consommateur avec synchronisation entre les deux

Slide 9

2.3 Synchronisation: Maximum d'un tableau

Solution valide pour trouver le maximum?

```
int m = 0;
co [i = 0 to n-1]
  if (a[i] > m)
    m = a[i];
```

Que se passe-t-il si on effectue la modification suivante?

```
   <if (a[i] > m)
      m = a[i];>
```

Solution avec *double check idiom*: les comparaisons (lectures) sont faites en parallèle, mais les écritures sont faites de façon séquentielle:

```
int m = 0;
co [i = 0 to n-1]
  if (a[i] > m)
    <if (a[i] > m)
       m = a[i];>
```

Note: la partie atomique ne s'effectue que lorsqu'il y a une chance qu'on ait un nouveau maximum

Slide 10

2.4 Actions atomiques et instructions `await`

2.4.1 Atomicité de fine granularité

Action atomique de granularité fine = action réalisée directement au niveau de la machine

Note: Sur la plupart des machines modernes: les valeurs des types de base peuvent être lues/écrites de façon atomique

Note: En Java, pas atomique pour les valeurs de type `double`

Slide 11

2.4.2 Spécification d'atomicité et de synchronisation: les crochets d'atomicité et l'instruction `await`

Action atomique de grosse granularité = instruction ou séquence d'instructions qu'on désire exécuter de façon à ce que cela *semble* atomique $\Rightarrow$ utilisation de `<...>`

Instruction de synchronisation conditionnelle `await` = ne permet l'exécution (de façon atomique) d'une ou plusieurs instructions que si une condition (la *garde*) est vraie:

```
< await(B) S; >
```

Donc: combine exclusion mutuelle *et* synchronisation conditionnelle

---

Cas spéciaux sans combinaison des deux:

- `< S; >`
- `< await(B); >`

Slide 12

2.5 Synchronisation entre producteur et consommateur

Structure typique de programmes parallèles: producteur/consommateur

Exemple Trouver les instances d'un motif dans un fichier (p. 48):

- Producteur = processus qui lit une ligne
- Consommateur = processus qui analyse la ligne pour trouver le motif

Autre exemple = copier un tableau `a` d'un processus vers un

Solution avec *busy waiting* = un processus teste une condition (avec `await`) jusqu'à ce qu'elle devienne vraie

Invariant: `c <= p <= c+1`

Interprétation de certaines conditions:

`p == c`: le tampon est vide `p > c`: le tampon est plein

Slide 13

```
int buf,    # Tampon pour element a copier
    p = 0,  # Prochain element du producteur a copier
    c = 0;  # Destination dans consommateur du prochain element

process Producer {
  int a[n];             # Tableau a copier
  while (p < n) {
    <await (p == c);>   # buf est libre
    buf = a[p];
    p = p + 1;
   }
}

process Consumer {
  int b[n];
  while (c < n) {
    <await (p > c);>    # buf contient l'element a copier
    b[c] = buf;
    c = c + 1;
  }
}
```