Université Gaston Berger de Saint-Louis

Programmation concurrente

Pr. Ousmane THIARE

M2RSD
[www.ousmanethiare.com]

10 mai 2024

Contenu

Contenu

- Exécution “simultanée” de plusieurs parties d’un programme ;
- Exécution sur la même machine ou sur des machines différentes.

- Type de concurrence :
  - Pseudo-parallélisme : un seul processus s'exécute à la fois ;
  - Quasi-parallélisme : du pseudo parallélisme avec commutation à la charge du processus en exécution ;
  - Parallélisme réel : plusieurs processus en même temps.
- Distinguer entre la programmation concurrente, parallèle, et distribuée.

- Programmation parallèle :
  - Exécution d'un programme sur plusieurs unités
    - Cores, processeurs, ordinateurs
- Programmation distribuée ou répartie :
  - Répartir l'exécution d'un programme sur plusieurs machines distinctes
  - Programmation parallèle sur plusieurs machines
- Programmation parallèle sur plusieurs machines
  - Style de programmation prenant en compte les différentes unités d'exécution offertes : threads, processus, ...

Contenu

- Le parallélisme dans l'exécution de logiciels peut se produire à quatre niveaux différents
  - Niveau instructions machine : exécutant plusieurs instructions machine simultanément ;
  - Niveau instructions de code : exécutant plusieurs instructions de code source simultanément ;
  - Niveau unités : exécutant plusieurs unités (routines ou sous programmes) simultanément.
  - Niveau programmes : exécutant plusieurs programmes simultanément.

Contenu

- Le parallélisme Physique : plusieurs unités appartenant au même programme sont exécutées littéralement en parallèle sur différents processeurs ;
- Le parallélisme Logique : plusieurs unités appartenant au même programme semblent (au programmeur et à l'application) être exécutées en parallèle sur différents processeurs. En fait, l'exécution réelle des programmes à lieu de manière intercalée sur un seul processeur ;
- Pour le programmeur et le créateur de langage, les deux types de parallélisme sont les mêmes.

- Problèmes
  - Partitionnement :
    - Décomposer le programme en unités de bases (threads, processus, ...)
  - Communication :
    - Échanger les données
  - Synchronisation :
    - Ce sont les feux de signalisation pour l'exécution des tâches
    - Faire en sorte de ne pas se marcher dessus.

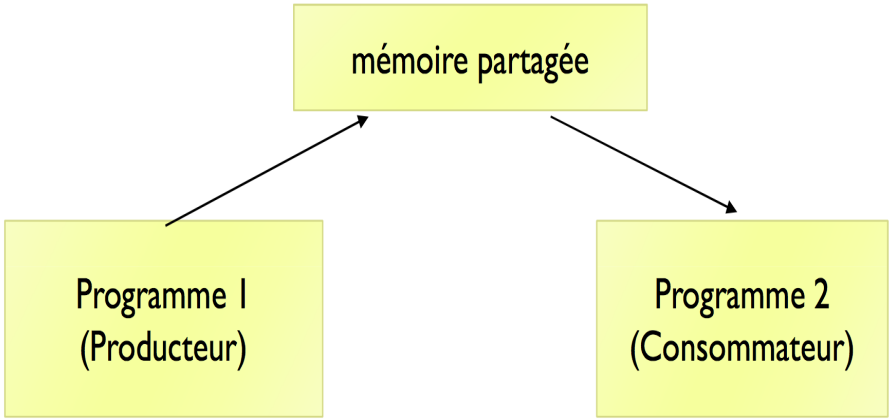
Contenu

- Mémoire partagée :
  - La communication est implicite
  - Accès mutuel aux données des autres
- Echange de messages :
  - Échanger les données
  - Chaque processus à son propre espace mémoire
  - Envoi de messages entre processus : synchrone ou asynchrone

- Types de concurrence
  - Disjointe : pas de communication entre les entités concurrentes
  - Compétitive : compétition pour l'accès à des ressources (CPU, entrées/sorties, mémoire, ...)
  - Coopérative : coopération pour atteindre un objectif commun

Contenu

Le Problème Du Producteur et du Consommateur



- Le programme 1 produit des données ; Le programme 2 utilise ces données
- La synchronisation est nécessaire :
  - L'unité du consommateur ne doit pas prendre des données si la mémoire partagée (mémoire tampon) est vide ;
  - L'unité du producteur ne peut pas placer de nouvelles données dans la mémoire partagée si elle n'est pas vide.

Contenu

- Nous avons deux tâches (A et B) et une variable partagée (TOTAL)
- La tâche A doit additionner 1 à TOTAL
- La tâche B doit multiplier TOTAL par 2
- Chaque tâche accomplit son opération en utilisant le processus suivant :
  - Chercher la valeur dans TOTAL
  - Effectuer l'opération arithmétique
  - Remettre la nouvelle valeur dans TOTAL
- TOTAL a une valeur originale de 3

- Sans synchronisation, 4 valeurs peuvent résulter de l'exécution des deux tâches :
  - Si A s'accomplit avant que B ne commence $\Rightarrow$ TOTAL=8
  - Si A et B cherchent leTOTAL avant que l'un ou l'autre remette la nouvelle valeur, alors :
    - Si A remet la nouvelle valeur dans TOTAL en premier $\Rightarrow$ TOTAL=6
    - Si B remet la nouvelle valeur dans TOTAL en premier $\Rightarrow$ TOTAL=4
  - Si B s'accomplit avant que A ne commence $\Rightarrow$ TOTAL=7
- Ce genre de situation s'appelle un état de course parce plusieurs tâches font la course pour utiliser les ressources partagées et le résultat dépend de l'ordre de l'exécution des tâches.

Contenu

- Une méthode générale consiste à monopoliser l'accès à une ressource
- Quand une tâche a fini d'utiliser une ressource partagée, elle doit l'abandonner de manière à ce que la ressource soit rendue disponible à d'autres tâches.

- Nous discutons maintenant de trois méthodes qui permettent l'accès mutuellement exclusif aux ressources :
  - **Les Mutex**
  - **Les Sémaphores**
  - **Les Moniteurs**

Contenu

- Un mutex est un verrou autour d'un bout de code
- Il a deux états :
  - Ouvert
  - Fermé
- Avant d'accéder à un code "critique", la tâche essaie de verrouiller le mutex (fermer)
  - S'il est déjà fermé : une autre tâche est dans la section critique, la tâche courante se met en attente
  - S'il est ouvert, la tâche courante ferme le mutex et passe à l'exécution de la section critique
- Après la fin de l'exécution de la section critique, la tâche courante ouvre le mutex et une des tâches en attente est réveillée.

Contenu

```
require 'thread'
mutex1 = Mutex.new
var = 1
a = Thread.new {
  mutex1.synchronize {
    var = var + 10
  }
}
b = Thread.new {
  mutex1.synchronize {
    var = var + 10
  }
}
```

- Problème possible avec les mutex :
  - Interblocage
- Comment se produit l'interblocage ?
  - Soient deux processus T1 et T2 et deux ressource R1 et R2
  - Un processus T1 ayant l'accès à la ressource R1
  - Un processus T2 ayant l'accès à la ressource R2
  - T1 demande l'accès à une ressource R2
  - T2 demande l'accès à une ressource R1
  - T1 et T2 sont bloqués :
    - Chacun attend l'autre pour libérer la ressource qu'il veut utiliser

x=5
y=2

```
Fermer Mutex1
Fermer Mutex2
  x = x + y
Ouvrir Mutex2
Ouvrir Mutex1
```

```
Fermer Mutex2
Fermer Mutex1
  x = x * y
Ouvrir Mutex1
Ouvrir Mutex2
```

- Introduits par Dijkstra dans les années 60
- Un sémaphore est une variable entière sur laquelle on définit deux opérations atomiques : P (wait) et V (signal)
- Un sémaphore peut être initialisé à une valeur $n \geq 0$.

- Soit un sémaphore S
- P(S) bloque le processus appelant jusqu'à ce que $S > 0$
- Une file d'attente est associée à chaque sémaphore pour contenir les processus bloqués.
- V(S) débloque le 1^{er} processus en attente s'il y en a un, sinon les signaux s'accumulent

Contenu

```
class semaphore {
      int valeur;
      listeDePcs liste;
   public:
      void P();
      void V();
}
```

```
semaphore::void P()
{  valeur--;
   if (valeur < 0)
   {    état du processus courant = bloqué;
        liste.ajoute(processus courant);
   }
}
semaphore::void V()
{  process processus;
   valeur++;
   if (valeur <= 0)
   {   processus = liste.retire();
       processus.etat = prêt;
   }
}
```

```
semaphore::void P()
{  if (valeur = 0)
   {    état du processus courant = bloqué;
        liste.ajoute(processus courant);
   }
   else  valeur--;
}
semaphore::void V()
{  process processus;
   if (!liste.vide())
   {   processus = liste.retire();
       processus.etat = prêt;
   }
   else valeur++;
}
```

Contenu

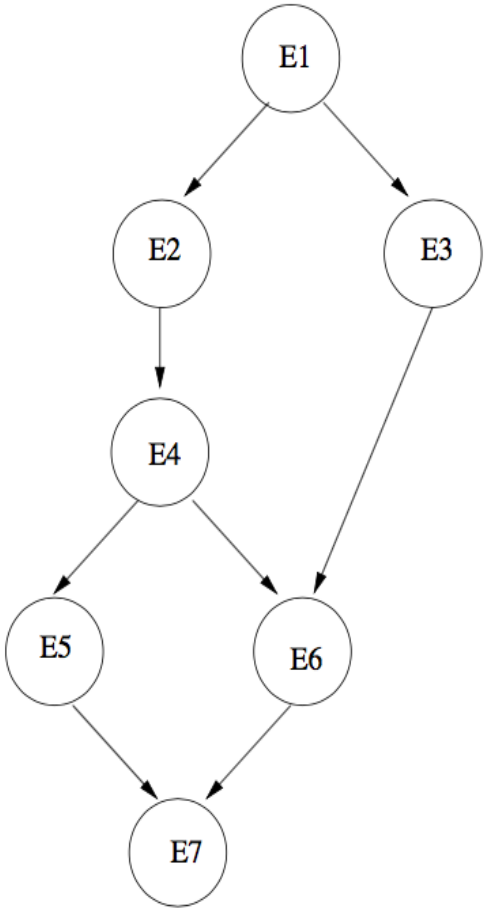
- Fil d'attente :
  - on l'implante où et comment ?
  - son implantation assure ou non l'équité...
- Atomicité :
  - les opérations P et V doivent être atomiques (sections critiques)
  - comment y parvenir (mono et multi processeur) ?

Contenu

```
semaphore mutex;

mutex.init(1);

    repeat
        P(mutex)
    .....section critique
        V(mutex)
    .....section non-critique
    forever
```

```
semaphore cond;

cond.init(0);
```

```
P1 : .                    P2 : .
     .                         .
     .                         .
     S1;                       P(cond);
     V(cond);                  S2;
     .                         .
     .                         .
     .                         .
```




```
Var a,b,c,d,e,f,g : semaphores (=0)
begin
 parbegin
   begin E1; V(a); V(b); end;
   begin P(a); E2; E4; V(c); V(d); end;
   begin P(b); E3; V(e); end;
   begin P(c); E5; V(f); end;
   begin p(d); P(e); E6; V(g); end;
   begin P(f); P(g); E7; end;
 parend;
end;
```

Contenu

- Soit un tampon contenant n espaces
- On peut initialiser un sémaphore à n pour détecter que le tampon est plein
- Utilisé dans le problème des producteurs/consommateurs

Contenu

- Assurent l'exclusion mutuelle avec facilité
- Evitent l'interblocage
- Sont-ils équitables ?

Contenu

- connaissance des compétiteurs
- opérations toujours difficiles à utiliser
- on peut oublier de mettre des éléments en section critique
- les mêmes primitives assurent l'exclusion mutuelle et la synchronisation conditionnelle
- les opérations P et V ne donnent aucune idée sur la ressource visée

Contenu

Tampon fini

- On possède un tampon contenant n éléments chacun contenant un item
- Pour la synchronisation on utilise 3 sémaphores
  - mutex (exclusion mutuelle)
  - plein et vide (synchronisation conditionnelle)

Tampon fini

```
type    item : ...

var     plein, vide, mutex : semaphore;
        tampon : array[0..n-1] of item;
        nextp, nextc : item;
```

Tampon fini

```
begin
    plein := n ; vide := 0 ; mutex := 1 ;

    parbegin
        producteur :        repeat
                                produire un "item" dans nextp ;
                                P(plein) ; P(mutex) ;
                                dépose nextp dans tampon ;
                                V(mutex) ; V(vide) ;
                            forever ;

        consommateur :      repeat
                                P(vide) ; P(mutex) ;
                                lire nextc de tampon ;
                                V(mutex) ; V(plein) ;
                                traite nextc ;
                            forever ;
    parend
end ;
```

Tampon fini

```
Program OPSYS;
var     in_mutex, out_mutex : semaphore initial (1,1);
        nun_in, num_out : semaphore initial (0,0);
        free_in, free_out : semaphore initial (n,n);
        tampon_in : array[0..n-1] of entree;
        tampon_out : array[0..n-1] of sortie;

process lecteur;
        var ligne : entree;
        loop
                lecture ligne;
                P(free_in); P(in_mutex);
                dépose ligne dans tampon_in;
                V(in_mutex); V(num_in);
        end;
end process;
```

Tampon fini

```
Process traitement ;
        var    ligne : entree ;    resultat : sortie ;
        loop
                P(num_in) ; P(in_mutex) ;
                lecture ligne de tampon_in ;
                V(in_mutex) ; V(free_in) ;
                traitement de ligne et génération de resultat ;
                P(free_out) ; P(out_mutex) ;
                dépose resultat dans tampon_out ;
                V(out_mutex) ; V(num_out) ;
        end ;
end process ;

Process imprimante ;
        var resultat : sortie ;
        loop
                P(num_out) ; P(out_mutex) ;
                lecture resultat de tampon_out ;
                V(out_mutex) ; V(free_out) ;
                impression de resultat ;
        end ;
end process ;
```

Lecteurs/écrivains

- Un objet peut être partagé par plusieurs processus
- Certains peuvent faire des lectures et d'autres des mises à jour
- Cette distinction est importante :
  - lectures simultanées seulement $\rightarrow$ pas de problème.
  - écritures simultanées $\rightarrow$ risque d'incohérences
  - lectures et écriture simultanées $\rightarrow$ risque d'incohérences

Lecteurs/écrivains

Exemple

- Compte en banque A contient \$500
- Transaction B ajoute \$10 sur A
- Transaction C ajout \$1000 sur A
- Séquence :
  - B lit A (\$500)
  - C lit A (\$500)
  - C ajoute \$1000
  - C écrit A (\$1500)
  - B ajoute \$10
  - B écrit A (\$510) $\Rightarrow$ A contient à la fin \$510

Lecteurs/écrivains

Solutions :

- Permet plusieurs lecteurs simultanées
- Accès exclusif aux écrivains
- Variations :
  - on ne fait attendre aucun lecteur
  - on ne fait attendre aucun écrivain
  - autres ? ? ?

Lecteurs/écrivains

Exemple de solution

- 2 sémaphores : mutex (1) et wrt (1)
- 1 entier (nblecteur)

Lecteurs/écrivains

Exemple de solution

```
Lecteur : P(mutex)
              nblecteur++
              if (nblecteur=1) P(wrt)
           V(mutex)
          .... lecture.....
           P(mutex)
              nblecteur--
              if (nblecteur=0) V(wrt)
           V(mutex)

Écrivain: P(wrt)
         .... écriture .....
          V(wrt)
```

Philosophes

- Introduit par Dijkstra (1965)
- 5 philosophes passent leur vie à penser et manger
- Ils partagent une table circulaire, 5 chaises, 5 plats de riz et 5 baguettes
- Pour manger il doit prendre 2 baguettes (les plus rapprochées)
- Si une des baguettes n'est pas disponible, il attend

Philosophes

```
Var baguette : array[0..4] of semaphore;
Procedure phil(i:integer)
begin
   repeat
      P(baguette[i])
      P(baguette[i+1mod5])
     ...mange ...
      V(baguette[i]
      V(baguette[i+1mod5])
     ... pense ...
  forever
end
begin
   baguette[0..4] :=1
   cobegin
      phil(0); phil(1); phil(2); phil(3); phil(4)
   coend
end
```

- Un **sémaphore** est une structure de données se composant d'un nombre entier et d'une file d'attente qui stocke des descripteurs de tâches
- Un sémaphore est une généralisation des mutex :
  - Un mutex est donc un sémaphore avec un nombre entier initialisé à 1 (sémaphore binaire)

- Le compteur d'un sémaphore permet d'indiquer le nombre de ressources encore disponibles ou le nombre de processus qui peuvent entrer en section critique
  - Dans les mutex, un seul processus peut entrer en section critique
- Il y a deux opérations liées à un sémaphore : attendre et libérer

Contenu

Attendre(Sem)

- Si compteur de Sem > 0
  - Décrémente le compteur de Sem (une ressource en moins)
  - Exécuter la section critique
- Sinon
  - Mettre l'appelant dans la file d'attente de Sem

Libérer(Sem)

- Incrémenter le compteur de Sem (libérer une place pour l'accès à la ressource)
- Si la file de Sem n'est pas vide
  - Réveiller un processus en attente dans la file d'attente de Sem

Le problème du producteur consommateur résolu avec les sémaphores

Sémaphores : articledisponible, placedisponible;
articledisponible.compteur = 0
placedisponible.compteur = BUFLEN

```
Producteur do
  loop
   Produire : valeur;
   Attendre(placedisponible);
   DEPOSER(valeur);
   Libérer(articledisponible);
  end loop
end producteur
```

```
Consommateur do
  loop
   Attendre(articledisponible);
   Récupérer (valeur);
   Libérer(placedisponible);
   Consommer(valeur);
  end loop
end consommateur
```

- Les moniteurs est un mécanisme de haut niveau de synchronisation
- Dans les moniteurs, les structures de données partagées avec leurs opérations sont encapsulées
- Fonctionnement :
  - Un seul processus peut être actif à un moment donné à l'intérieur du moniteur
  - Toutes les autres demandes d'exécution de primitives du moniteur sont bloquées tant qu'il y aura un processus actif à l'intérieur du moniteur
  - Cela produit une exclusion mutuelle garantie

- Quel est la différence entre les moniteurs et les sémaphores ?
  - Un processus en section critique ne peut se bloquer qu'une fois la ressource est libérée
  - Le moniteur se charge de réveiller un processus en attente d'une ressource : ce n'est plus les processus qui s'occupent de gérer la file d'attente
  - Dans le cas des moniteurs on dispose de deux primitives : Attendre et Signaler

Contenu

- Les threads partagent souvent les même variables
- Un thread peut être bloqué :
  - Une application se subdivise en processus et un processus peut être composé de plusieurs threads
  - Les processus sont généralement créés lors de la conception de l'architecture alors que les threads sont créés lors de la phase de programmation/exécution.

Contenu

- Il existe différents modèles de thread
- Chaque modèle a ses avantages et inconvénients
- Chaque langage de programmation supporte un ou plusieurs modèles de thread.

- Il existe différents modèles de thread
- Chaque langage de programmation supporte un ou plusieurs modèles de thread
- Chaque modèle a ses avantages et inconvénients.

Modèles existants :

- Modèle 1 :1
- Modèle 1 :N
- Modèle M :N.

Modèle 1 :1

- Dans ce modèle un thread utilisateur (du programme) est associé à un thread système
- C'est le modèle le plus répondu (surtout sous linux)
- On l'appelle souvent "threads natifs"

Modèle 1 :1

- Avantages :
  - Les threads peuvent être exécutés dans différentes CPUs
    - Vrai parallélisme
  - Les threads ne se bloquent pas mutuellement
  - Le scheduler système s'occupe de la gestion de chaque thread

Modèle 1 :1

- Inconvénients :
  - Difficultés de mise en place et de configuration
    - À chaque fois qu'un thread utilisateur est créé, il faudra notifier le système (OS) pour créer un état système à ce thread
  - Avec beaucoup de threads, le noyau de Linux bug
  - Limitation du nombre de threads qui peuvent être créés

Modèle 1 :N

- Dans ce modèle un thread système est associé à plusieurs thread utilisateur (du programme)
- On les appelles les "thread verts" (green threads)
  - Car ils sont légers, économique ... et écologique

Modèle 1 :N

- Avantages :
  - La création, exécution et arrêt de threads est facile et n'est pas couteuse
  - Beaucoup de threads peuvent être créés
    - Des dizaines de milliers et plus

Modèle 1 :N

Inconvénients :

- Le scheduler système n'a pas connaissance des threads en exécution
  - Exemple
    - le scheduler voit un thread, alors que côté utilisateur il y a y thread s'exécutant dans ce même thread
- Les opérations d'entrée/sortie peuvent bloquer tous les autres threads utilisateur qui s'exécutent à l'intérieur du même thread système

Modèle 1 :N

Inconvénients :

- Tous les threads utilisateur s'exécutent sur la même CPU
  - Pas de vrai parallélisme

Modèle M :N

- Dans ce modèle M threads système sont associés à plusieurs thread utilisateur (du programme)
- C'est un modèle hybride
  - Combinant les deux modèles précédents

Modèle M :N

- Avantages :
  - Les threads peuvent être exécutés dans différentes CPUs
  - Seulement quelques threads peuvent être bloqués par des appels système (I/O)
  - La création, exécution et arrêt de threads est facile et n'est pas couteuse
  - Beaucoup de threads peuvent être créés
    - Des dizaines de milliers et plus

Modèle M :N

- Inconvénients :
  - Le scheduler système et le scheduler utilisateur ont besoin de collaboration étroitement
  - Les threads utilisateur associés à un même thread système se bloquent si un d'eux effectue des opérations d'entrée/sortie (I/O)
  - Difficulté à écrire, maintenir et déboguer du code basé sur ce modèle

- Erlang
  - M :N
- Java,Ruby1.9,Python
  - 1 :1
- Anciennes version de Java et Ruby
  - 1 :N
- Le modèle 1 :1 est actuellement le plus efficace et le plus répandu