

Programmation concurrente

Exécution “simultanée” de plusieurs parties d’un programme

Exécution sur différents *thread*, *process*, *processeur*, *machine*

quasi-parallélisme, pseudo-parallélisme, Parallélisme réel

Distinguer programmation *concurrente*, *parallèle*, et *distribuée*

Objectifs et problèmes

Objectifs:

- *Parallélisme*: Accélérer l'exécution d'une application
- *Concurrence*: Gérer plus efficacement certaines ressources
- *Distribution*: Faire collaborer plusieurs machines instables
- *Génie logiciel*: Mieux organiser le code

Problèmes:

- *Partitionnement*: Décomposer en processus
- *Communication*: Échanger les données à traiter
- *Synchronisation*: Ne pas se marcher dessus

Support matériel d'exécution concurrente

Un ensemble de processeurs liés par un système de communication

Processeurs: homogènes, mais pas forcément

Communication: cache partagée, mémoire partagée, mémoires interconnectées par un réseau spécialisé (bus, hypercube, tore, crossbar), réseau ethernet, ...

Modèles de communication

Comment les processus communiquent et se synchronisent

Mémoire partagée:

Un seul tas: accès mutuel aux données des autres

La communication est souvent implicite, liée à la synchronisation

Typiquement une machine SMP ou SMT

Passage de messages:

Chaque processus a son propre tas: les données sont exclusives

La synchronisation est souvent implicite, liée à la communication

Typiquement un super-ordinateur, ou un système distribué

L'illusion d'une mémoire partagée

Le modèle de communication ne reflète pas forcément la technique de communication sous-jacente

- Gestion par le compilateur et le système d'exécution (*partitioning*)
- Gestion par le système de pagination (*distributed shared memory*)
- Gestion par le système de mémoire cache (*directory*)

Même si il y a une mémoire physique partagée, les caches la ... cachent

Bien sûr, une mémoire peut aussi s'utiliser pour envoyer des messages

The first problems

```
incr c = swapMVar c (1 + readMVar c)
```

```
go count 0 = return ()
```

```
go count n = do incr count
```

```
              go count (n - 1)
```

```
main = do c <- newMVar
```

```
         forkIO (go c 100000)
```

```
         forkIO (go c 100000)
```

```
         print (readMVar c)
```

The first problems (2^e essai)

```
incr c = do cv <- readMVar c
          swapMVar c (1 + cv)
```

```
go count 0 = return ()
go count n = do incr count
                go count (n - 1)
```

```
main = do c <- newMVar
          forkIO (go c 100000)
          forkIO (go c 100000)
          cv <- readMVar c
          print cv
```

Le résultat n'est pas 200000:

- Interlacement des différentes opérations
- Il y a une *condition de course*
- `incr` est une *section critique*
- Il faut assurer l'*exclusion mutuelle* dans `incr` \Rightarrow ¡Synchronisation!

Le résultat n'est toujours pas 200000:

- Attendre la fin des autres processus
- ¡¡Synchronisation!!

¿Combien d'interlacements possibles?

Outils disponibles:

- Masquer les interruptions
- *Mutex*: `lock M ... unlock M`
- *Sémaphores*
- *Moniteur*
- *Dataflow*: variables vides/pleines

Comment implanter un mutex?

Attente active ou passive

Lock lock

```
lock M1
lock M2
x1 = x1 - 100
x2 = x2 + 100
unlock M2
unlock M1
```

```
lock M2
lock M1
x2 = x2 + 100
x1 = x1 - 100
unlock M1
unlock M2
```

Lock lock

lock M1	lock M2
lock M2	lock M1
$x1 = x1 - 100$	$x2 = x2 + 100$
$x2 = x2 + 100$	$x1 = x1 - 100$
unlock M2	unlock M1
unlock M1	unlock M2

¡Deadlock!

Example classique: philosophes démunis

Concurrent Haskell: dataflow

```
incr c = do cv <- takeMVar c
          putMVar c (1 + cv)
```

...

```
main = do c <- newMVar
          forkIO (go c 100000)
          forkIO (go c 100000)
          <magie>
          cv <- takeMVar c
          print cv
```

Attendre un autre processus

Compte à numéro dans une banque Suisse:

```
incr c n = do cv <- takeMVar c
            putMVar c (n + cv)
```

```
depot c n | n > 0 = incr c n
```

```
retrait c n = do << waitfor c >= n >>
                incr c (- n)
```

Comment attendre?

Attente active

```
retrait c n = do cv <- readMVar c
                if cv >= n
                  then incr c (- n)
                  else do yield
                          retrait c n
```

Problèmes:

- Gaspillage de CPU
- Condition de course

Attente active correcte

```
retrait c n = do cv <- takeMVar c
               if cv >= n
                 then putMVar c (cv - n)
                 else do putMVar c cv
                        yield
                        retrait c n
```

C'est quand même encore de l'attente active

Attente active: avec mutex

```
depot c n | n > 0 = do lock M
                    c = c + n
                    unlock M
```

```
retrait c n = do lock M
                if c >= n
                then do c = c - n
                       unlock M
                else do unlock M
                       retrait c n
```

Attente avec moniteur

Variable de condition C

```
depot c n | n > 0 = do lock M
                    c = c + n
                    signal C
                    unlock M
```

```
retrait c n = do lock M
                if c >= n
                then do c = c - n
                       unlock M
                else do release&wait M C
                       retrait c n
```

Attente avec moniteur

Imbrication

```
depot c n | n > 0 = do lock M
                    c = c + n
                    signal C
                    unlock M
```

```
retrait c n = do lock M
                while c < n
                    release&wait M C
                c = c - n
                unlock M
```

Problèmes pour le compilateur

Certaines optimisations invalidées par les courses

```
lock L1
```

```
write B = 1
```

```
t1 = read A
```

```
write B = 2
```

```
unlock L1
```

```
lock L2
```

```
write A = 1
```

```
t2 = read B
```

```
write A = 2
```

```
unlock L2
```

Conditions pour être “sans course”

Une course est un accès à une variable pendant qu'elle est modifiée

Pas de course pour les variables non partagées

Pas de course si chaque accès est protégé par un mutex

Attention à utiliser le même mutex

Chaque mutex protège un ensemble particulier de variables

Pas de course \nrightarrow correct

Composition difficile

On veut déplacer de l'argent entre 2 comptes

```
retrait c1 n
```

```
depot c2 n
```

Le moniteur protège *retrait* et *depot*

On aimerait maintenir l'état intermédiaire caché

Il faut exporter le lock M, etc...

Les mutex sont un outil pour obtenir l'atomicité

Pas de course \nrightarrow atomique

Monad STM en Haskell:

```
atomically :: STM a -> IO a
```

```
newTVar :: a -> STM (TVar a)
```

```
readTVar :: TVar a -> STM a
```

```
write TVar :: TVar a -> a -> STM ()
```

Le retour du compte suisse

```
depot c n | n > 0
  = atomically (do cv <- readTVar c
                  writeTVar c (cv + n))

retrait c n
  = atomically
    (do cv <- readTVar c
        if cv < n
          then retry
          else writeTVar c (cv - n))
```

L'opération *retry* :: *STM* α correspond à *release&wait*

Composer les deux

```
depot c n | n > 0 = do cv <- readTVar c
                    writeTVar c (cv + n)
```

```
retrait c n = do cv <- readTVar c
               if cv < n
               then retry
               else writeTVar c (cv - n)
```

on peut alors faire

```
atomically (do { retrait c1 n; depot c2 n })
```

Exporter le lock M sans risque de deadlock ni d'oubli de lock

Comment ne pas attendre

Autre composition: $orElse :: STM \alpha \rightarrow STM \alpha \rightarrow STM \alpha$

```
retrait' :: TVar Int -> Int -> STM Bool
retrait' c n = do {retrait c n; return True}
               `orElse` return False
```

Ou encore

```
retraitRiche :: [TVar Int] -> Int -> STM ()
retraitRiche [] _ = retry
retraitRiche (c:cs) n
  = retrait c n
    `orElse` retraitRiche cs n
```

Comment ça marche

`retry` attend un `unlock` parmi ceux utilisés

Quels mutex utiliser:

- Un gros mutex
- Un mutex par variable
- Demande au programmeur
- Analyse par le compilateur pour inférer des mutex
- Synchronisation optimiste

Synchronisation optimiste

Dans le cas idéal, les conflits sont rares

`lock` est une opération coûteuse, et conservatrice

On peut à la place, détecter les conflits post-facto

Synchronisation optimiste

Dans le cas idéal, les conflits sont rares

`lock` est une opération coûteuse, et conservatrice

On peut à la place, détecter les conflits post-facto

- Chaque lecture et écriture note l'état de la variable
- On garde aussi un *log* des variables accédées
- À la fin d'une transaction, on vérifie si l'état a changé
- Si oui *rollback*, sinon *commit*

Erreurs impossibles

La synchronisation optimiste introduit de nouvelles erreurs

```
atomic {  
    scale = a+b;  
    c = c / scale;  
}
```

```
atomic {  
    a -= n;  
    b += n;  
}
```

Erreurs impossibles

La synchronisation optimiste introduit de nouvelles erreurs

```
atomic {  
    scale = a+b;  
    c = c / scale;  
}
```

```
atomic {  
    a -= n;  
    b += n;  
}
```

! si $n = a + b$, alors *scale* peut être 0 !