Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Programmation Concurrente

Bakery Algorithm
Sémaphores

Jacques Supcik | 2013 | T-2a/T2d

Hes·so FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Bakery Algorithm (Leslie Lamport)

Principe

- Section critique avec plus que 2 processus.
- Principe du «ticket» dans les boulangeries (premier arrivé, premier servi).

Bakery Algorithm (Leslie Lamport)

Algorithme pour 2 processus

«Bakery Algorithm» pour 2 processus	
integer np ← 0, nq ← 0	
p	**q**
loop forever p1: non-critical section p2: np ← nq + 1 p3: await nq = 0 or np ≤ nq p4: critical section p5: np ← 0	loop forever q1: non-critical section q2: nq ← np + 1 q3: await np = 0 or nq < np q4: critical section q5: nq ← 0

- np et nq sont les numéros des tickets des deux processus.
- Une valeur de 0 indique que le processus ne veut pas entrer dans la section critique.
- En cas d'égalité, c'est le processus p qui gagne.

Bakery Algorithm (Leslie Lamport)

Algorithme pour N processus

«Bakery Algorithm» pour N processus

```
integer array[1..n] number ← [0,…,0]

     loop forever
p1:    non-critical section
p2:    number[i] ← 1 + max(number)
p3:    for all other processes j
p4:      await (number[j] = 0) or (number[i] ≪ number[j])
p5:    critical section
p6:    number[i] ← 0
```

- `i` est compris entre 1 et `N` et représente l'«ID» du processus.

for all *other* processes	**number[i] ≪ number[j]**
`for j from 1 to N` `  if j ≠ i`	`(number[i] < number[j]) or` `((number[i] = number[j]) and (i < j))`

Bakery Algorithm (Leslie Lamport)

Remarques

- Le numéro des tickets peut être incrémenté indéfiniment si des processus restent toujours dans la section critique.
- Un processus qui veut entrer dans la section critique doit demander le numéro des tickets de tous les autres processus, même si aucun autre processus désire entrer dans la région critique.
- La fonction `max(number)` doit être faite de manière atomique.

Bakery Algorithm (Leslie Lamport)

Version sans opération atomique

«Bakery Algorithm» sans opération atomique

```
boolean array[1..n] choosing ← [false,…,false]
integer array[1..n] number ← [0,…,0]
```

```
     loop forever
p1:    non-critical section
p2:    choosing[i] ← true
p3:    number[i] ← 1 + max(number)
p4:    choosing[i] ← false
p5:    for all other processes j
p6:      await choosing[j] = false
p7:      await (number[j] = 0) or (number[i] ≪ number[j])
p8:    critical section
p9:    number[i] ← 0
```

Fast Algorithm (Leslie Lamport)

- Le «bakery algorithm» doit parcourir la liste des tickets séquentiellement.
- Avec un grand nombre de processus, cet algorithme devient alors peu performant ($O(n)$).
- En 1987, Leslie Lamport publie l'article «A fast mutual exclusion algorithm» avec une complexité constante ($O(1)$).
- Nous n'étudierons pas cet algorithme dans ce cours, mais vous trouverez l'article de Leslie Lamport sur Internet:

  `http://research.microsoft.com/en-us/um/people/lamport/pubs/fast-mutex.ps`
- Cet algorithme est aussi décrit dans le chapitre 5.4 du livre «Principles of Concurrent and Distributed Programming» de M. Ben-Ari.

Sémaphores

Inventé en 1965 par Edsger W. Dijkstra

Définition

Un sémaphore **S** se compose de 2 champs:

- **V**: un entier positif.
- **L**: un ensemble de processus.

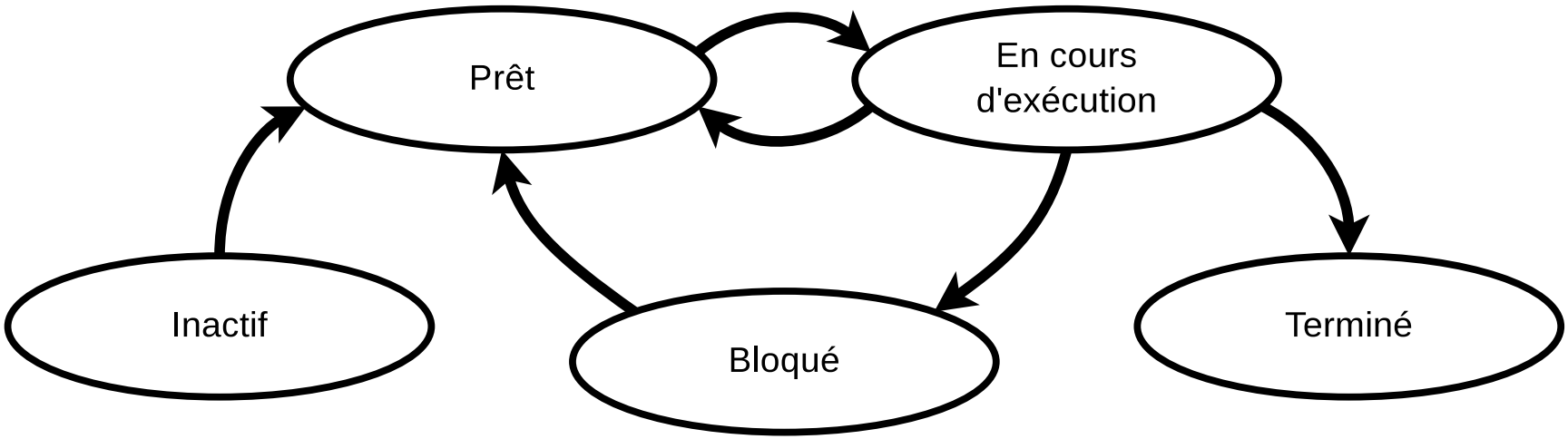
Initialisation

`Semaphore S` $\leftarrow$ `(k, {})`

(avec k $\geq$ `0`)

Sémaphores

Etat des processus (process state)



- Utilise les fonctionnalités offertes par le système d'exploitation.
- Plus besoin de faire du «busy waiting»!

Sémaphore

acquire(S)

- L'opération «acquire» est *atomique*
- «p» est le processus qui exécute l'opération

acquire(S)

```
if S.V > 0
  S.V ← S.V − 1
else
  S.L ← S.L ∪ p
  p.state ← bloqué
```

Dès que «`acquire`» est appelé avec `S.V = 0`, on dit que le processus `p` bloque sur le sémaphore `S`.

Sémaphore

release(S)

- L'opération «release» est *atomique*

release(S)

```
if S.L = {}
  S.V ← S.V + 1
else
  soit q un processus quelconque de S.L
  S.L ← S.L − {q}
  q.state ← prêt
```

Mutex

- Un «Mutex» est un sémaphore «binaire».
- $S.V \in \{ 0, 1 \}$.
- Le Mutex est initialisé avec (0, $\{\}$) ou (1, $\{\}$).

release(S) (pour un Mutex)

```
if S.V = 1
  // non défini
lse if S.L = {}
  S.V ← S.V + 1
else // comme pour le sémaphore
  soit q un processus quelconque de S.L
  S.L ← S.L − {q}
  q.state ← prêt
```

Autres noms pour l'opération «Acquire»

- Wait
- Down
- Acquire (Java)
- Proberen (test) $\rightarrow$ P
- Puis-je?
- Procure

Autres noms pour l'opération «Release»

- Signal
- Up
- Release (Java)
- Verhogen (augmenter) $\rightarrow$ V
- Vas-y!
- Vacate

Exclusion mutuelle avec un sémaphore

Algorithme pour 2 processus

Section critique avec un sémaphore	
binary semaphore S ← (1, {})	
p	**q**
loop forever p1: non-critical section p2: acquire(S) p3: critical section p4: release(S)	loop forever q1: non-critical section q2: acquire(S) q3: critical section q4: release(S)

Exclusion mutuelle avec un sémaphore

Algorithme pour N processus

- Même algorithme que pour 2 processus.
- Le choix d'un processus *quelconque* lors du «release» peut provoquer la famine (starvation) d'un processus $\rightarrow$ Remplacer le «set» $S.L$ par une «queue».

Sémaphores en Java

java.util.concurrent.Semaphore – Constructeurs

Semaphore(int permits)
Creates a Semaphore with the given number of permits and nonfair fairness setting.

Semaphore(int permits, boolean fair)
Creates a Semaphore with the given number of permits and the given fairness setting.

Sémaphores en Java

java.util.concurrent.Semaphore – acquire

`void` **`acquire`**`()`

Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.

`void` **`acquire`**`(int permits)`

Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted.

...

`boolean` **`tryAcquire`**`(long timeout, TimeUnit unit)`

Acquires a permit from this semaphore, if one becomes available within the given waiting time and the current thread has not been interrupted.

Voir `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html`.

Sémaphores en Java

java.util.concurrent.Semaphore – release

`void` **`release`**`()`
Releases a permit, returning it to the semaphore.

`void` **`release`**`(int permits)`
Releases the given number of permits, returning them to the semaphore.

...

`boolean` **`hasQueuedThreads`**`()`
Queries whether any threads are waiting to acquire.