SPP (Synchro et Prog Parallèle)

Unit 5: Beyond Locks: Semaphores and Monitors

François Taïani

Session Overview

- So far one family of synchronisation mechanisms:
  - ➔ locks (plain, reentrant, read/write)
  - ➔ 2 types of realisation: spin locks, and blocking locks
- With which we addressed two types of problem
  - ➔ competition (mutual exclusion)
  - ➔ collaboration (readers/writers, producers/consumers)
- Today: more advanced synchronisation mechanisms
  - ➔ semaphores
  - ➔ monitors

Semaphore

- Literally
  - ➔ carrying (phore) signal, signs (sema)
- XIX century
  - ➔ optical telegraph
  - ➔ first deployed in France: Paris – Toulon in 20 minutes
  - ➔ a grand total of 556 stations used until the 1850
- Computer programming
  - ➔ proposed by Edsger Dijkstra (1930-2002) in 1965

Semaphores

A semaphore: a shared entity with

- an internal counter (count) counts “permits” or “tokens”
  - ➔ start value set at when semaphore is created
  - ➔ value otherwise not directly accessible (usually)
- two operations: up() and down()
  - ➔ also called V() and P(): comes from Dutch!
  - ➔ other names: signal / wait, release / acquire, …
- Semantics
  - ➔ sem.up() ➔ sem.count++ (atomically, never blocking)
  - ➔ sem.down() ➔ if sem.count==0 block until > 0
    sem.count--

Semaphore Invariant

- Two invariants. Each semaphore guarantees that
  - ➔ sem.counter ≥ 0
  - ➔ sem.counter =
    start_value + #up_returned - #down_returned

Semaphores and locks

Link with locks: Semaphores generalise locks

- A simple lock can be implemented with a semaphore

```
class lock is
  semaphore sem = new semaphore(1)
  lock() is
    sem.down()
  end
  unlock() is
    sem.up()
  end
end
```

➔ called “Binary semaphore” / “Mutex semaphore”

➔ Quiz: Is this a reentrant lock? Does it behave exactly like a lock in all situations?

Semaphores and locks (cont.)

- A semaphore can be implemented with a lock
- Exercise
  - ➔ write the pseudo-code implementing a semaphore using **one** lock and a counter

Semaphores and locks (cont.)

- A semaphore can be implemented with a lock
- Exercise
  - ➔ write the pseudo-code implementing a semaphore using **one** lock and a counter

```
int count
lock l

up() is
  l.lock()
  count++
  l.unlock()
end
```

```
down() is
  l.lock()
  while (count==0) {
    l.unlock()
    yield()
    l.lock()
  }
  count--
  l.unlock
end
```

Semaphores and locks (cont.)

Notes on previous solution

- not optimal for CPU usage: busy waiting
  - ➔ because of “yield()” CPU usage ≤ 100%, but not 0%
- implementation possible that avoids busy waiting
  - ➔ see lab exercises

Using Semaphores

- Semaphores can be used as a replacement for locks
  - ➔ initialised as 1, alternate between 0s and 1s
  - ➔ called “binary semaphores” or “mutex semaphores”
- Semaphores very good at modelling pools of resources
  - ➔ e.g. 10 printers available, sem = new semaphore(10)
  - ➔ all the bookkeeping handled by the semaphore
- Can be use to solve the consumers / producers problem

Cons/Prod with Semaphores

- Three semaphores
  - ➔ one to protect the queue (binary semaphore)
  - ➔ one to count free cells
  - ➔ one to count empty cells
- Exercise:
  - ➔ Try to write the code of produce(..) and consume(..) using these 3 semaphores

Cons/Prod with Semaphores

- Three semaphores
  - ➔ one to protect the queue (binary semaphore)
  - ➔ one to count free cells
  - ➔ one to count empty cells

```
queue q
sem q_sem = new sem(1)
sem empty = new sem(N)
sem full  = new sem(0)

method produce(x) is
  empty.down()
  q_sem.down()
  q.add(x)
  q_sem.up()
  full.up()
end
```

```
method int consume() is
  full.down()
  q_sem.down()
  result = q.get()
  q_sem.up()
  empty.up()
  return result
end
```

Java API

- **java.util.concurrent.Semaphore**
  - ➔ **Semaphore**(int permits)
  - ➔ **acquire**() // like down
  - ➔ **release**() // like up()
- Many other methods
  - ➔ to get current value of counter (permits)
  - ➔ to try to acquire a permit
  - ➔ to acquire / release several permits at once
  - ➔ to get queue of blocked threads

Monitors

- primary synchronisation primitive in Java
- monitor = an object with additional properties
  - ➔ all methods protected in mutual exclusion
  - ➔ offer a signalling mechanism (more in a moment)
- motivation
  - ➔ good practice: release lock in same method as taken
  - ➔ better to group all methods using same lock in one object

Monitor Example

- Monitor:
  - ➔ equivalent to implicit lock + lock ops on each methods

```
monitor account is
  int value
  credit(x) is
    value += x
  end
  withdraw(x) is
    value -= x
  end
end
```

```
class account is
  lock l_account
  int value
  credit(x) is
    l_account.lock()
    value += x
    l_account.unlock()
  end
  withdraw(x) is
    l_account.lock()
    value -= x
    l_account.unlock()
  end
end
```

Wait + Signal

In addition to previous mechanism, signalling feature

- wait() operation
  - ➔ releases monitor
  - ➔ blocks current thread (“asleep”, not schedulable)
- signal() operation (also known as notify)
  - ➔ wakes up one of the threads waiting as a result of wait()
  - ➔ this waiting thread retakes the monitor and resumes execution just after the wait
- Note
  - ➔ advanced version of the ping/pong exercise

Precise Semantic of Signal

- Tricky bit: it varies
- First semantic (“Hoare’s semantic”)
  - ➔ the thread calling signal (signaller) loses the monitor
  - ➔ the signalled thread gains access immediately
  - ➔ signaller regains access just after signalled has left
- Second semantic (“Mesa’s semantic”, used in Java)
  - ➔ signaller does not lose the monitor
  - ➔ signalled put on queue of thds waiting to get the monitor

Why bother about difference?

- Can mean a lot
- Example: Implementing semaphore with a monitor

```
monitor semaphore
  int count = init_value
  method up() is
    count++
    signal()
  end
  method down() is
    if count==0 { wait() }
    count--
  end
end
```

➔ Does this work with Hoare-style monitors?

➔ Does this work with Mesa-style monitors?

In Java

- each object is associated with a monitor
  - ➔ also known as the object's "intrinsic lock"
- methods put into the monitor with "synchronized"
  - ➔ effect: locks the intrinsic lock of enclosing object
  - ➔ the class object is used for static methods
- possible to put a block of code in a monitor
  - ➔ you must specify the monitor explicitly e.g. synchronized(this) { .. }
- Signalling done using
  - ➔ wait(), notify(), notifyAll()
  - ➔ semantic: mesa-style

Summary

- Semaphores
  - ➔ extend locks
  - ➔ associated with counter, block when counter == 0
  - ➔ for pools of resources, can be used as locks (binary)
- Monitors
  - ➔ language construct
  - ➔ structure the use of locks
  - ➔ come with signalling facilities: “wait” / “signal” (or “notify”)
  - ➔ two semantics for “signal”: Hoare’s and Mesa’s
  - ➔ prime mechanism in Java (Mesa style)