SPP (Synchro et Prog Parallèle)

Overview

François Taïani

The Teaching Team

Lectures + Exercises + Labs :

- **François Taiani**
  - ➔ francois.taiani@irisa.fr
- My interests
  - ➔ system software (OS, middleware)
  - ➔ large scale distributed computing
  - ➔ decentralized and parallel systems

The module

- SPP (Synchronisation et Programmation Parallèle)

- Aims: introduction to
  - ➔ common mechanisms for synchronisation
  - ➔ common coordination problems and their solutions
  - ➔ foundation of parallel computing
  - ➔ alternative approaches to parallelism

Assessment

- 2 in-class tests (35% each, 70% in total)
  - ➔ 35% for each test
  - ➔ 1 hour each + 1h correction
  - ➔ Dates: see module's site on moodle

- 2 marked practical labs (10% + 20% = 30% in total)
  - ➔ marked during lab sessions
  - ➔ schedule will depend on how the labs progress

Module Outline

- Unit 1: Introduction and Motivation
- Unit 2: Locks and Mutual Exclusion
- Unit 3: Implementing Locks
- Unit 4: Solving typical problems with locks
- Unit 5: Beyond Locks: Semaphores and monitors
- Unit 6: Conditions, Barriers, and Rendez-Vous
- Unit 7: Petri Nets
- Unit 8: Alternative Approaches: Actors & Immutability
- Unit 9: Formalising Parallelism: Atomicity
- Unit 10: Transactions and Wait-Free Programming

Module Outline

- Unit 1: Introduction and Motivation
- Unit 2: Locks and Mutual Exclusion
- Unit 3: Implementing Locks
- Unit 4: Solving typical problems with locks
- Unit 5: Beyond Locks: Semaphores and monitors
- Unit 6: Conditions, Barriers, and Rendez-Vous
- Unit 7: Petri Nets
- Unit 8: Alternative Approaches: Actors & Immutability
- Unit 9: Formalising Parallelism: Atomicity
- Unit 10: Transactions and Wait-Free Programming

SPP (Synchro et Prog Parallèle)

Unit 1: Introduction & Motivation

François Taïani

What is Concurrent Computing?

- **Multiple** executions of one or several programs
  - ➔ occurring "**at the same time**" (more on this)
  - ➔ **interacting** with each other in some ways
- Different models of **interaction**
  - ➔ shared memory
  - ➔ shared resources (printer, hardware bus)
  - ➔ through messages / events
- In this course
  - ➔ we focus on **shared memory**
- Link to **distributed** computing: strong
  - ➔ but not the same focus & problems

Concurrency & Parallelism

- **Concurrent Computing**
  - ➔ = multiple executions
  - ➔ two main ways to do this
- **Option 1: Parallel Computing**
  - ➔ each execution occurs on different cores / CPUs
  - ➔ “True” Parallelism
- **Option 2: Multiplexing / Time sharing**
  - ➔ one CPU is shared by multiple executions
  - ➔ OS scheduler shares CPU time between executions
- Nowadays : multi-core machines, but #core << #executions
  - ➔ combination of parallelism and timesharing

Why Concurrent Computing?

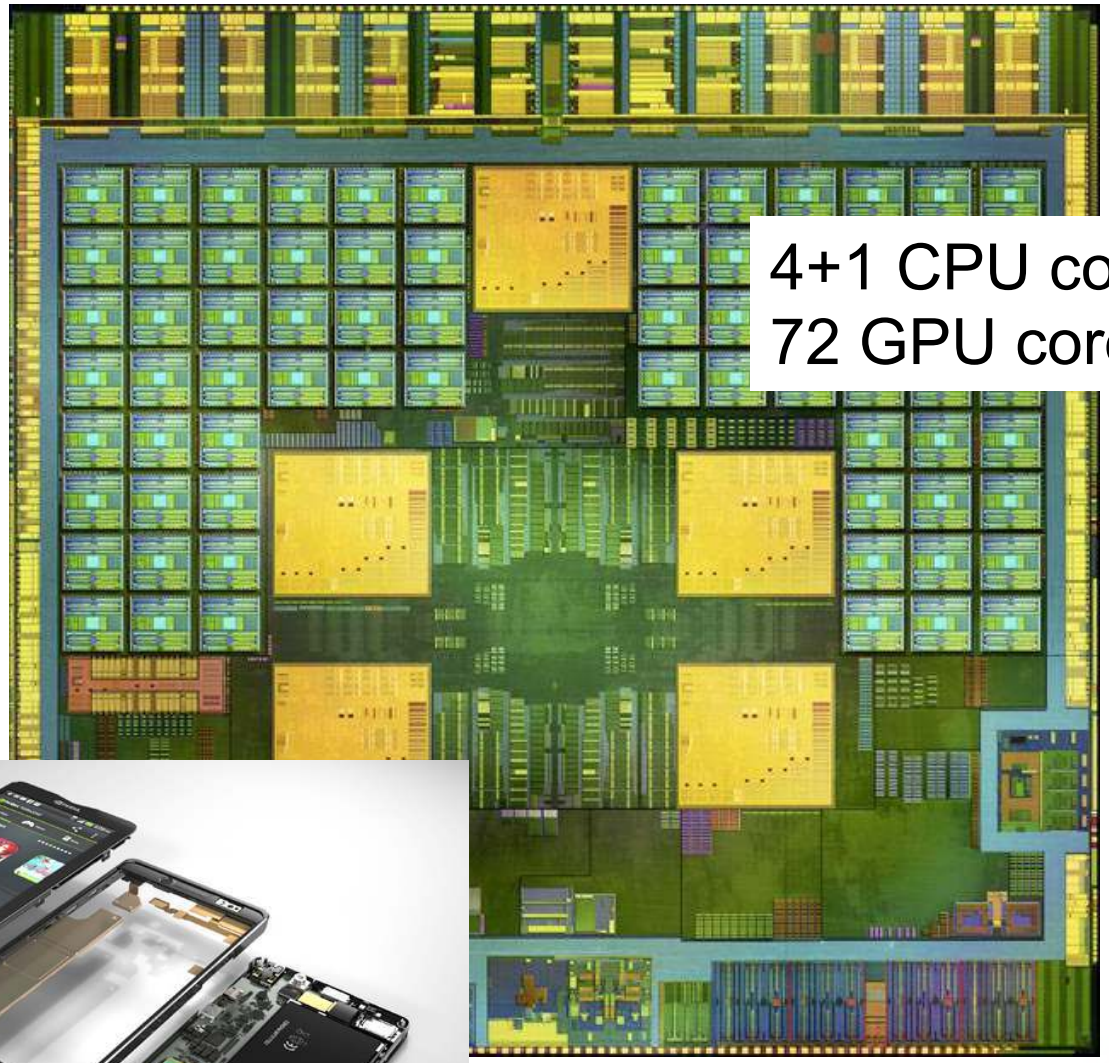
- Concurrent computing = much harder than sequential
  - ➔ So why bother?

- Take 5 minutes to think of
  - ➔ 3 examples of domain that use concurrent computing
  - ➔ 3 reasons why concurrent computing is useful

Why Concurrent Computing?

- Examples
  - ➔ 3D graphics on GPUs (Graphical Processing Units)
  - ➔ web servers (100s of concurrent requests)
  - ➔ scientific simulations (High Performance Computing)

- Reasons
  - ➔ key reason 1 go faster (e.g. multi-core CPUs and GPUs)
  - ➔ to maximise resource utilisation e.g. CPU during I/O
  - ➔ to structure inherently concurrent applications: GUI, RT



4+1 CPU cores
72 GPU cores

NVIDIA®

Tegra 4 processor
from Nvidia

http://www.theregister.co.uk/Print/2013/01/11/ten_stars_of_ces_2013/

Spotlight on GP-GPU

GPU: Highly parallel architecture (true parallelism)

- up to several 100s of cores on one card
- now used for more than 3D: finance, e-science
- highly specialized hardware
  - ➔ all cores = same instruction flow or big penalty
  - ➔ different types of memory with different latencies
- known as SIMD archi (Single Inst. Multiple Data)
  - ➔ opposite is MIMD (typical multi core CPU)
- Specialised library for scientific computation
  - ➔ e.g. CUDA from NVidia

Spotlight on I/O

- Multi-core processors quite recent
  - ➔ for a long time only very high-end machines
- So why concurrent computing? Because of I/O
- Example (pseudo code)
  - ➔ for i = 1 to 1000
    read file_input$_i$
    compute simulation based on file_input$_i$
    save simulation results to file_results$_i$
    endfor
  - ➔ Question: utilisation rate of CPU?
    (what kind of data do you need)

Spotlight on I/O (cont.)

- Example (pseudo code)
  - ➔ for i = 1 to 1000
    read file_input$_i$
    compute simulation based on file_input$_i$
    save simulation results to file_results$_i$
    endfor

- Question: utilisation rate of CPU? Assume
  - ➔ each input and result file = 200 kB
  - ➔ read / write bandwidth of hard-drive is 20 MB / s
  - ➔ simulations takes 80,000,000 assembly instructions
  - ➔ CPU frequency is 4 GHz

Spotlight on I/O (cont.)

- Some comments
  - ➔ big simplification: only disk bandwidth, no latency
  - ➔ in practice OS would ensure caching of reused files
- Main message: moving data much slower than CPU!
  - ➔ for same “complexity” of operations

```
AdminMacBook ftaiani [ftaiani] $ ls -hl bigfile
-rw-r--r--  1 ftaiani  staff    95M  5 Jan  2010 bigfile

AdminMacBook ftaiani [ftaiani] $ time cp bigfile bigfile2
cp bigfile bigfile2  0.00s user 0.24s system 4% cpu 6.008 total
```

Spotlight on I/O (cont.)

- Solution: use parallel programming (one core)
  - ➔ for i = 1 to 1000 **do in parallel**
    read file_input$_i$
    compute simulation based on file_input$_i$
    save simulation results to file_results$_i$
    endfor
  - ➔ Question: What is the new CPU utilisation rate?

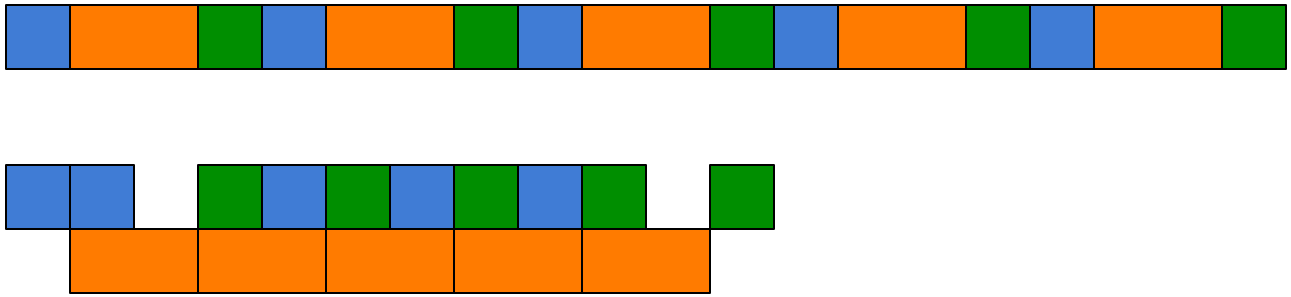
Spotlight on I/O (cont.)

- Data bottleneck found in many applications
  - ➔ web-servers: many requests, data on HD or DB
  - ➔ database servers: same issue
  - ➔ data-heavy application: reading from files
  - ➔ the reason why applications takes time to start!

- one CPU + several flows of executions = **concurrent programming**
  - ➔ illusion of “parallelism” provided by OS (see OS course)
  - ➔ same problems and same programming techniques

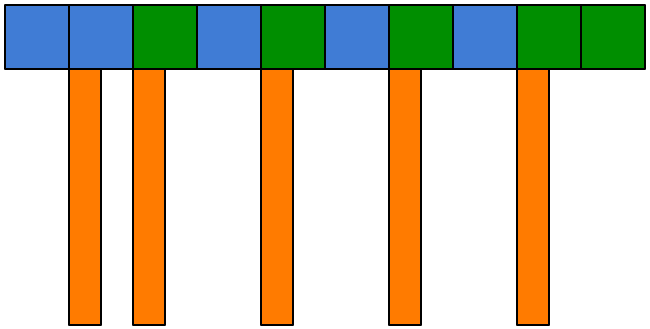
Limits of Parallelism

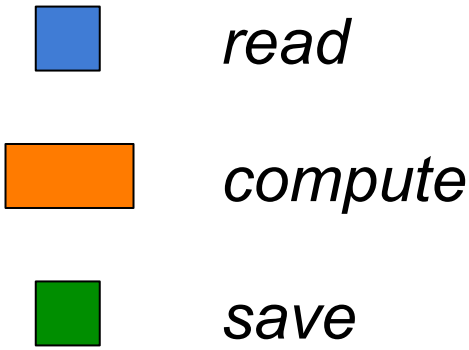
- Simulation example



…

- Can we go arbitrarily faster with more CPUs / cores ?
  - ➔ No, disk becomes bottleneck





Limits (cont) Amdahl's law

- Captures impact of **CPU bottlenecks**
- Idea: sequential execution of program (1 processor)
  - ➔ total execution time with 1 processor T_{seq}
  - ➔ proportion that cannot be parallelised: α
- Question:

  New "best possible" duration with n processors?

$$T_{par}(n) = \alpha\, T_{seq} + (1-\alpha)\, T_{seq} / n$$

$$T_{par}(n) = [\, \alpha\, (n-1) + 1\,] / n \times T_{seq}$$

- Best speedup: $n / [\, \alpha\, (n-1) + 1\,]$

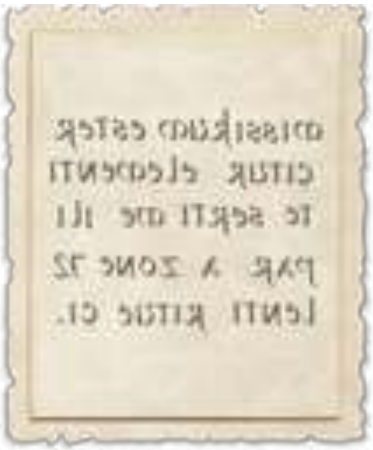
Model: Processes

- To reason about parallelism: programming model
- Based on the notion of **processes**
  - ➔ = one **sequential** execution of one program
- Process ≠ program ≠ processor
  - ➔ program = code = recipe
  - ➔ process = worker, active entity making progress
  - ➔ one program can be executed several times

```
AdminMacBook ftaiani [ftaiani] $ ps -Uftaiani | grep gv
 2518 ttys000    0:00.07 gv
 2712 ttys002    0:00.05 gv
 2745 ttys003    0:00.00 grep gv
AdminMacBook ftaiani [ftaiani] $ kill 2518
AdminMacBook ftaiani [ftaiani] $ kill 2712
```

A Metaphor

The recipe =
the program (on HD)



The cauldron =
the memory

The potion =
the data / state

The cook = the process

Where is the processor?

Shared memory

- One cauldron (memory) for several cooks (processes)
  - ➔ usually one single recipe (program)
  - ➔ but each cook might use a different bit of the recipe

Vocabulary

- In **this module** process = thread (of execution)
  - ➔ "**process**" more common in algorithms, **theory**
  - ➔ "**thread**" used in **programming**
- In **operating systems**: meaning can be different
  - ➔ process = thread(s) + address space + program
  - ➔ 1 thread = monothreaded
  - ➔ > 1 threads = multithreaded
  - ➔ but in some OS, each thread receives their own PID…

Multithreading in Java

Based on the class **java.lang.Thread**

Two ways to define what a Thread should do

- **Way N. 1:** By **extending** the class **Thread**
  - ➔ class MyThread extends Thread
  - ➔ implement the method run()
  - ➔ create an instance of your class: t = new MyThread()
  - ➔ start your thread: t.start()

Multithreading in Java

- **Way N. 2:** implement the **Runnable interface**
  - ➔ class MyRunnable implements Runnable
  - ➔ implement method run()
  - ➔ create an instance of your class: r = new MyRunnable()
  - ➔ start a thread pointing to your instance:
    new Thread(r).start()
  - ➔ (or call new Thread(this).start() in your constructor)

Analysing a simple example

- Consider the following example (pseudo code)

```
shared var count int = 0

thread t1 is
var a int
begin
  for a=1 to 10,000,000
    count++
  endfor
end
```

```
thread t2 is
var b int
begin
 for b=1 to 10,000,000
    count++
  endfor
end
```

- Two questions:
  - ➔ How to implement it in Java?
  - ➔ What is the value of count after the program executes?

First a simplification

- Notice how both thread execute the same code
  - ➔ just replace b by a. Both are private variables.

```
shared var count int = 0

function f is
var a int
begin
  for a=1 to 10,000,000
    count++
  endfor
end
```

```
thread t1 is f()
thread t2 is f()
```

In Java

- Java has no functions, but classes so
  - ➔ define our own extension of Thread
  - ➔ define run() to be the same as function f
  - ➔ global variable count implemented as a static variable

```
public class MultiThreadedCounter extends Thread {

   static long count = 0;

   public void run() {
      for(int i=1; i<=10000000; i++) {
         MultiThreadedCounter.count++ ;
      } // EndFor
   } // EndMethod run

   public static void main(String[] args) {
      Thread t1 = new MultiThreadedCounter();
      Thread t2 = new MultiThreadedCounter();
      t1.start();
      t2.start();
   }

} // EndClass MultiThreadedCounter
```

What is the result of this?

- Quizz: What is the value of count after t1 & t2 end?

- Experiment: running the program

So what is happening?

- Best way to know: let's look at the byte code
  - ➔ javap -c MultiThreadedCounter : disassemble .class

```
public void run();
  Code:
   Stack=4, Locals=2, Args_size=1
   0: iconst_1
   1: istore_1
   2: iload_1
   3: ldc #2; //int 10000000
   5: if_icmpgt 22
   8: getstatic #3; //Field count:J
   11:  lconst_1
   12:  ladd
   13:  putstatic #3; //Field count:J
   16:  iinc  1, 1
   19:  goto  2
   22:  return
```

What we have

- A case of **contention**: 2 threads, 1 resource (count)
  - ➔ resource corrupted because access not controlled
- Reason: **lack of atomicity** (more on this later)
  - ➔ Even basic instructions are not necessarily atomic
  - ➔ **++ is not atomic in java**: decomposed in steps
  - ➔ these steps might overlap with that of other threads!
- We need mechanisms to bring order to this
  - ➔ need for synchronisation
- That is what we'll study in the rest of the module

Summary

- Parallel & concurrent computing extremely important
  - ➔ to exploit multi-core architectures
  - ➔ to mask I/O slow latencies and bandwidth
  - ➔ to program inherently concurrent applications
- Difference between parallelism and concurrency
- Parallelism / concurrency can speed up programs
  - ➔ but limited by bottlenecks (e.g. disk, sequential sections)
- Key notion of *process* (= thread)
- java.lang.Thread
- Concurrent programming can easily go wrong!

SPP (Synchro et Prog Parallèle)

Unit 2: Locks & Mutual Exclusion

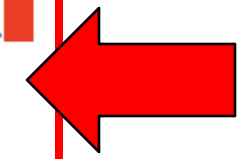
François Taïani

Unit Outline

- Locks and mutual exclusion
- Fairness
- Locks, Safety & Liveliness
  - ➔ Graphical Representation
- Locks and performance
- Deadlocks
- Advanced locks
  - ➔ Reentrant locks
  - ➔ Read/Write locks

Revisiting Unit 1's problem

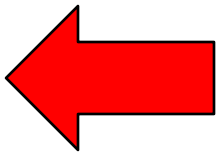
```
public class MultiThreadedCounter extends Thread {

   static long count = 0;

   public void run() {
      for(int i=1; i<=10000000; i++) {
         MultiThreadedCounter.count++ ;
      } // EndFor
   } // EndMethod run

   public static void main(String[] args) {
      Thread t1 = new MultiThreadedCounter();
      Thread t2 = new MultiThreadedCounter();
      t1.start();
      t2.start();
   }

} // EndClass MultiThreadedCounter
```



Revisiting Unit 1's problem

- Key problem: ++ is not atomic
  - ➔ implemented as several byte code instructions
  - ➔ the 2 threads overwrite each other's results

```
public class MultiThreadedCounter extends Thread {

  static long count = 0;

  public void run() {
    for(int i=1; i<=10000000; i++) {
      MultiThreadedCounter.count++ ;
    } // EndFor
  } // EndMethod run

  public static void main(String[] args) {
    Thread t1 = new MultiThreadedCounter();
    Thread t2 = new MultiThreadedCounter();
    t1.start();
    t2.start();
  }

} // EndClass MultiThreadedCounter
```



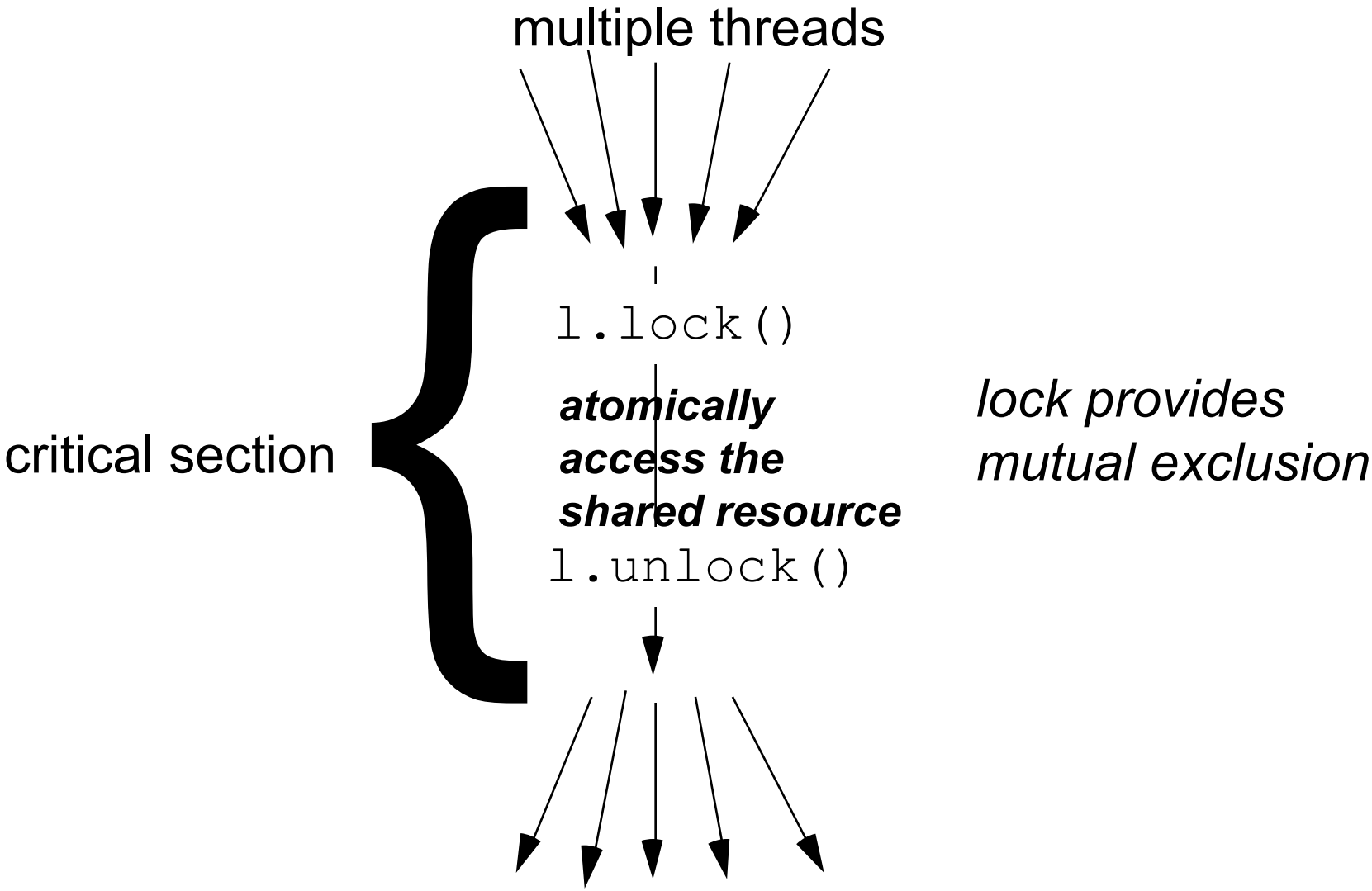
Solution

- Use locks!
- What is a lock?
  - ➔ a runtime structure (in Java an object)
  - ➔ can be in 2 states: “locked” / “unlocked”
  - ➔ state toggled with 2 methods: lock() and unlock()
    - other names are possible: e.g. take() and release()
  - ➔ the last thread to return from lock() “owns” the lock
  - ➔ only one thread at a time can own the lock
    - others attempting to lock it are kept blocked

Locks and Mutual Exclusion

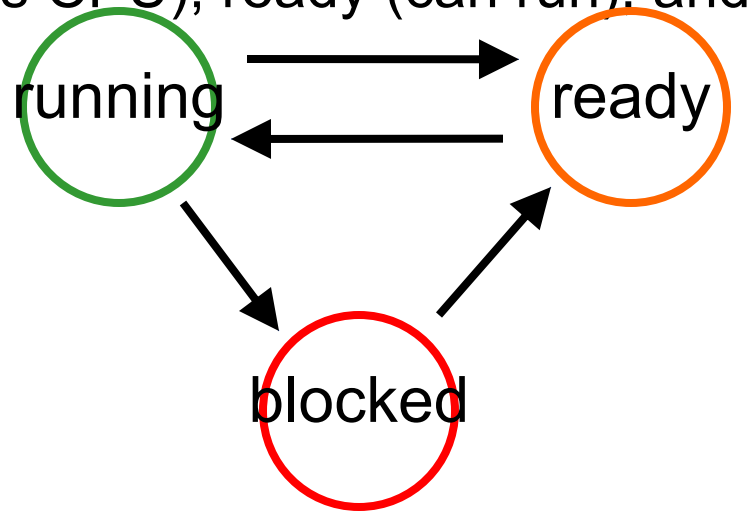
- Locks create **"safe regions"** of code
  - ➔ invariant:
    "*only one thread at a time in the regions of the lock*"
  - ➔ known as "critical sections"
- If thread t calls l.lock() when l is already locked
  - ➔ t **remains blocked** in call until lock is unlocked
  - ➔ when lock unlocked: one of the blocked threads **unblocked**
- Similar to a **lock on a bathroom**
  - ➔ l.lock() same as: try open the door
    if unlocked, enter and lock; if locked, wait until unlocked
  - ➔ except ≥ 1 regions can be associated with same lock

Locks & Mutual Exclusion



Locks and OS Scheduler

- Locks often implemented at **OS** level
  - ➔ OS **scheduler** decides **which** process / thread runs, **when**
  - ➔ based on heuristics, priorities, etc. (see OS module)
- Each process / thread can be in **3 states**
  - ➔ running (has CPU), ready (can run), and blocked



Locks and Scheduler

- Switch running / ready
  - ➔ known as a **context switch**
  - ➔ goal: share n cores between m >> n threads / processes
  - ➔ See OS course
- Blocked state
  - ➔ process / thread cannot run: e.g. **waiting on I/O or lock**
- If lock l already taken
  - ➔ l.lock() puts current thread in **blocked state**
- When thread that has lock calls l.unlock()
  - ➔ **one** of the threads blocked on lock put in “ready” state

Fairness of Locks

- If **multiple threads** are **waiting** for a lock
  - ➔ who gets the lock next is non-deterministic
  - ➔ depends on **heuristics** of underlying OS scheduler
  - ➔ or result of hardware-level scheduling in multicores

- Lock implementations possible where
  - ➔ some threads get lock **preferably** when competing (unfair)
  - ➔ opposite property: **fairness**
  - ➔ important property when looking at lock implementations

Fairness of Locks

- Example
  - ➔ server application, accepts 2 types of requests: increment and decrement

```
shared var  count int = 0
shared lock l

request increment is
begin
  l.lock
  count++
  l.unlock
end

request decrement is
begin
  l.lock
  count--
  l.unlock
end
```

  - ➔ If always mix of 2 requests waiting to take the lock l
  - ➔ Fair lock = both should get lock as often as the other

Locks, Safety & Liveliness

- Locks prevents **bad situations** from happening
  - ➔ i.e. no more than one thread in critical section
- Known as **"safety"** property
  - ➔ **nothing bad** happens
- However one key **danger**: that nothing happens at all
  - ➔ known as "**deadlock**": so many locks, program is blocked
- Avoiding this: known as "**liveliness**" property
  - ➔ **something happens**
- Continuous **tension** between **safety** and **liveliness**
  - ➔ locks help with safety, but can hurt liveliness

Locks in Java

- (Caveat: Not standard Java synchronisation
  - ➔ see ‘synchronized’ keyword and monitors in Unit 5)
- Interface

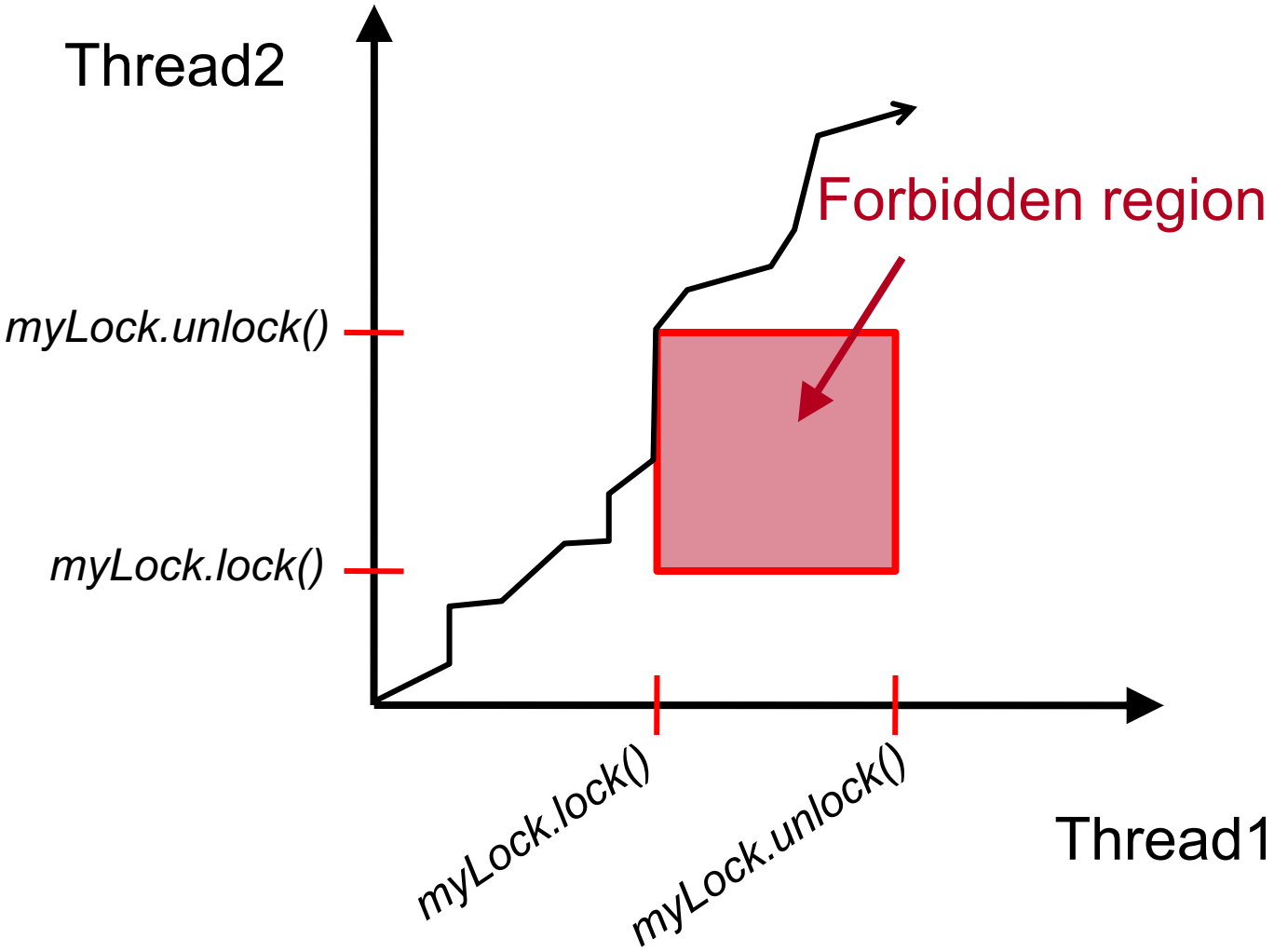
  java.util.concurrent.locks.Lock
- Implementation:

  java.util.concurrent.locks.ReentrantLock
  - ➔ we will see what “Reentrant” mean
- lock and unlock methods + others variants
  - ➔ tryLock(..), lockInterruptibly()

```
import java.util.concurrent.locks.ReentrantLock;

public class MultiThreadedCounterWithLock extends Thread {

   static long count = 0;
   static ReentrantLock myLock = new ReentrantLock();

   public void run() {
      for(int i=1; i<=10000000; i++) {
         myLock.lock();
         MultiThreadedCounterWithLock.count++ ;
         myLock.unlock();
      } // EndFor
   } // EndMethod run

   public static void main(String[] args) {
      Thread t1 = new MultiThreadedCounterWithLock();
      Thread t2 = new MultiThreadedCounterWithLock();
      t1.start();
      t2.start();
   }

} // EndClass MultiThreadedCounterWithLock
```

Revisiting our Example

Graphical Representation



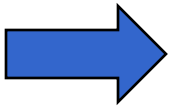
Locks and Performance

```
AdminMacBook ftaiani [SIMPLIFIED] $ time java MTCounter
java MTCounter 0.31s user 0.08s system 117% cpu 0.335 total

AdminMacBook ftaiani [SIMPLIFIED] $ time java MTCounterLock
java MTCounterLock 1.53s user 0.11s system 127% cpu 1.289 total
```

- Locks introduce a performance penalty
  - ➔ bookkeeping, scheduling, blocking, unblocking
- But do we pay this price all the time?

A variant

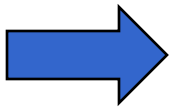
- Removing contention: count private to each thread



```
public class MTCountNoContention extends Thread {

  long count = 0;

  public void run() {
    for(int i=1; i<=10000000; i++) {
      this.count++ ;
    } // EndFor
  } // EndMethod run

  public static void main(String[] args) {
    Thread t1 = new MTCountNoContention();
    Thread t2 = new MTCountNoContention();
    t1.start();
    t2.start();
  }

} // EndClass MTCountNoContention
```

And the lock version

- count and lock private to each thread



```
public class MTCountLockNoContention extends Thread {

  long count = 0;
  ReentrantLock myLock = new ReentrantLock();

  public void run() {
    for(int i=1; i<=10000000; i++) {
      this.myLock.lock();
      this.count++ ;
      this.myLock.unlock();
    } // EndFor
  } // EndMethod run

  public static void main(String[] args) {
    Thread t1 = new MTCountLockNoContention();
    Thread t2 = new MTCountLockNoContention();
    t1.start();
    t2.start();
  }

} // EndClass MTCountLockNoContention
```

But …

- We no longer need locks here:
  - ➔ each thread has its own count variable
- So why would we use locks?
  - ➔ to test the impact of contention of performance
  - ➔ more generally as a form of “defensive programming”
- Defensive Programming
  - ➔ my code (classes) will be used (and abused) by others
  - ➔ hope for the best, plan for the worse

The results

```
AdminMacBook ftaiani [SIMPLIFIED] $ time java MTCntNoCont
java MTCntNoCont 0.33s user 0.08s system 121% cpu 0.340 total

AdminMacBook ftaiani [SIMPLIFIED] $ time java MTCntLockNoCont
java MTCntLockNoCont 1.44s user 0.09s system 166% cpu 0.921 total
```

- Still some substantial cost (+170%)
  - ➔ but not as bad as with contention (+284%)

- Note: these Java locks not particularly optimised
  - ➔ The linux kernel offers fast mutex (futex)
    futex with no contention: almost no impact

Analysing a Code with Locks

- Consider the following program (pseudo-code)

```
shared lock printer
shared lock network

thread t1 is
begin
  printer.lock()
  network.lock()
  network.unlock()
  printer.unlock()
end
```

```
thread t2 is
begin
  network.lock()
  printer.lock()
  printer.unlock()
  network.unlock()
end
```

Java Version

- The two locks as static global variables

```
public class BuggyWithLocks {

  public static ReentrantLock printer = new ReentrantLock();
  public static ReentrantLock network = new ReentrantLock();
```

Java Version

- The 2 threads extending class Thread

```
class Thread1 extends Thread {
  public void run() {
    for(int i=1; i<=10000000; i++) {
      BuggyWithLocks.printer.lock();
      BuggyWithLocks.network.lock();
      // work with printer and network
      BuggyWithLocks.network.unlock();
      BuggyWithLocks.printer.unlock();
    }
  }
}
```

```
class Thread2 extends Thread {
  public void run() {
    for(int i=1; i<=10000000; i++) {
      BuggyWithLocks.network.lock();
      BuggyWithLocks.printer.lock();
      // work with printer and network
      BuggyWithLocks.printer.unlock();
      BuggyWithLocks.network.unlock();
    }
  }
}
```

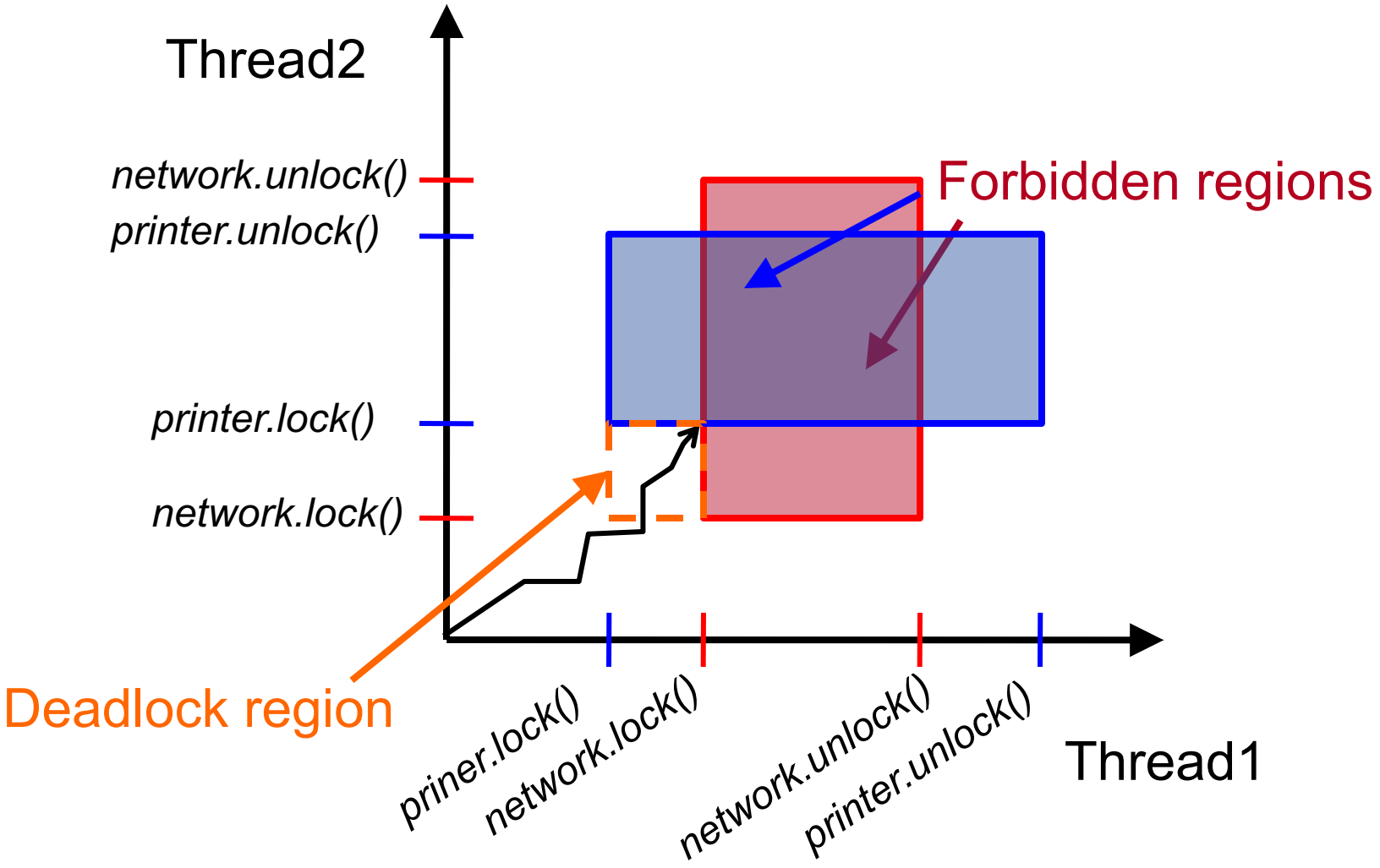
Java Version

- All started in method main

```
public static void main(String[] args) {
   Thread t1 = new Thread1();
   Thread t2 = new Thread2();
   t1.start();
   t2.start();
}
```

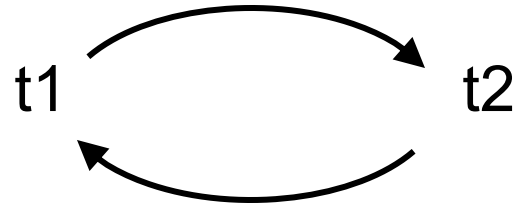
- Demonstration of execution
  - ➔ Quiz: Where is the problem?
  - ➔ Give a graphical representation of the problem.

Graphical Representation



Deadlocks

- t1 is waiting for t2 to release the network lock
- but t2 is waiting for t1 to release the printer lock



- Here only 2 threads
  - ➔ but could involve any number
  - ➔ t1 ➔ t2 ➔ t3 ➔ … ➔ tn ➔ t1
- Possible to detect (cycle detection)
  - ➔ but performance hit + what to do then?

Deadlocks

- Typical case of **loss of liveliness**
  - ➔ system is safe (does not enter any forbidden state)
  - ➔ but no progress is made (no liveliness)
- Seems easy to avoid
  - ➔ always take locks in same order
- But harder to achieve that it appears
  - ➔ reason: locks hidden in libraries / function calls

Hidden Locks

```
shared lock printer
shared lock network

thread t1 is
begin
  printer.lock()
  foo()
  printer.unlock()
end
```

```
thread t2 is
begin
  network.lock()
  bar()
  network.unlock()
end
```

- Apparently no problem, except
  - ➔ foo() uses the network (e.g. for logging)
  - ➔ and bar connects to the printer (e.g. to get font info)
  - ➔ result: deadlock!!

Advanced locks

- Consider the following code

```
shared lock printer

thread t1 is
begin
  printer.lock()
  bar()
  printer.unlock()
end
```

```
function bar is
begin
  printer.lock()
  // do something
  printer.unlock()
end
```

➔ What will happen?

Reentrant Locks

- Aka **recursive** locks
- The previous program
  - ➔ blocks with plain (non-recursive) locks
  - ➔ thread t1 is **deadlocked with itself!**
- Solution: **recursive / reentrant** locks: use a counter
  - ➔ incremented each time thread gets lock
  - ➔ decremented each time thread releases lock
  - ➔ lock freed when counter gets to zero
- Are all locks reentrant / recursive?
  - ➔ no: POSIX locks (“mutex”, in C) are not by default
  - ➔ special option to be used

Read Write Locks

- Normal locks can be bit blunt at times
- Consider following object
  - ➔ 2 counters: a and b
  - ➔ 2 ops: get sum a+b, or transfer an amount from a to b

```
int a = 1000
int b = 0

method getSum is
begin
  return a+b
end
```

```
method transfer(int x) is
begin
  a = a - x
  b = b + x
end
```

One Solution: Normal Locks

```
int a = 1000
int b = 0
lock l

method getSum() is
begin
  l.lock()
  int result = a + b
  l.unlock()
  return result
end
```

```
method transfer(int x) is
begin
  l.lock()
  a = a - x
  b = b + x
  l.unlock()
end
```

- What is the possible downside of this code?

Normal Locks

- Previous solution
  - ➔ prevents transfer() to interfere with getSum() ✔
  - ➔ but also forbids concurrent executions of getSum() ✘
- If getSum() invoked by many threads: performance hit
- Solution: use **read/write locks**
  - ➔ aka as "shared locks"
- 2 lock operations
  - ➔ **lock_read**: locked in reading, multiple threads allowed
  - ➔ **lock_write**: locked in writing, exclusive access

With Read/Write Locks

```
int a = 1000
int b = 0
rw_lock l

method getSum is
begin
  l.lock_read()
  int result = a + b
  l.unlock()
  return result
end
```

```
method transfer(int x) is
begin
  l.lock_write()
  a = a - x
  b = b + x
  l.unlock()
end
```

- Concurrent execution of getSum() now allowed

Summary

Introduction to principle and use of locks

- What are locks?
- What property to they provide?
- What is their impact on performance?
- Danger in using locks: Deadlocks
- Advanced locks: Recursive locks, Read/Write locks

Next session: How can locks be implemented?

SPP (Synchro et Prog Parallèle)

Unit 3: Implementing Locks

François Taïani

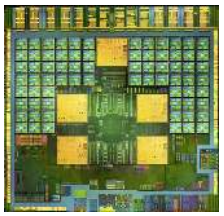
Why looking at this?

- Unless you work on very low level code
  - ➔ very unlikely to ever have to re-implement locks

- But …
  - ➔ a **fundamental** algorithmic problem: **Mutual Exclusion**
  - ➔ a “simple” **example** of concurrent programming
  - ➔ a good illustration of **potential pitfalls**
  - ➔ and of type of **techniques** used for higher abstractions
  - ➔ some input on **which implementation** to choose & when

A Bit of Vocabulary

- We have seen that true **parallelism ≠ concurrency**
  - ➔ true parallelism implies concurrency, not the reverse
- **True parallelism** is found on
  - ➔ multi-core machines
  - ➔ multi-processor machines with shared memory
  - ➔ hyper-threaded processors



- These machines are usually called **SMP**
  - ➔ Symmetric Multi-Processor, or Shared-memory Multi-Processor
  - ➔ Opposite: Uni-Processor machines or UP
  - ➔ 'SMP' acronym found in the Linux kernel for instance

Remainder of Session

- Approach 1: Spin Locks & Interrupt-based Locks
  - ➔ First attempt
  - ➔ On mono processor machines
  - ➔ Dangers, rules of use
  - ➔ On multi processor machines
  - ➔ Problems, rules of use
- Approach 2: Blocking Locks
  - ➔ Principle
  - ➔ Example code

Approach 1: Spin Locks

- Basic Idea for lock operations: For each lock
  - ➔ use a bit (or an integer) as a flag: 0 free, 1 taken
  - ➔ continuously check in a loop whether flag==0 (spinning)
  - ➔ if free (flag==0 is true), set the flag to 1 (taken)

```
object spin_lock is
  flag = 0

  method lock() is
    while (flag!=0) do {} // spin while lock taken
    flag = 1              // take lock
  end

  method unlock() is
    flag = 0              // release lock
  end
end
```

Spin Locks: Assumptions

- Relies on low level assembly instructions so that
  - ➔ testing (flag==0) and setting (flag=1) the flag are atomic
- However, even with this, the code below is **flawed**
  - ➔ Why?

```
object spin_lock is
  flag = 0

  method lock() is
    while (flag!=0) do {} // spin while lock taken
    flag = 1              // take lock
  end

  method unlock() is
    flag = 0              // release lock
  end
end
```

Spin Locks: Version 2

- On a mono-processor machine
  - ➔ concurrency occurs through context switches
  - ➔ in kernel context switches = interrupts (IO, scheduler,...)
- Simple solution: stop interrupts (cli() in Linux kernel)

```
method lock() is
  label try_again:
    disable_interrupts()
    if (flag!=0) do {
      enable_interrupts() // re-enable concurrency
      goto try_again      // spin while lock taken
    }
  flag = 1                // take lock
  enable_interrupts()
end
method unlock() is
  flag = 0                // release lock
end
```

~~Spin~~ Locks: Version 3

- Works but in practice **Linux kernel** use simpler version
  - ➔ **Interrupt-based locks**
  - ➔ Warning: **only** on Uni-Processor (UP) machines!

```
object very_basic_lock is // not spin lock anymore
  method lock is
    disable_interrupts()
  end

  method unlock is
    enable_interrupts()
  end
end
```

- This solutions comes with quite a few dangers?
  - ➔ Can you see which ones?

Dangers of Version 3

- Disabling interrupts pretty **brutal**
  - ➔ in effect one single big lock for all critical sections
  - ➔ risk of losing interrupts: losing I/O data, throttling reactivity
  - ➔ what if we forget unlock: kernel freezes! (Kernel Hang)
  - ➔ (note also that previous code is not reentrant)

- So in practice, on UP machines
  - ➔ interrupt-based locks only used in **kernel code**
  - ➔ for small snippets of code
  - ➔ that is guaranteed to **finish**, and **finish fast**

Spin Locks on SMP Machines

- Disabling interrupts: Does not work on SMP machines
  - ➔ other cores still working, might access locked data
- Solution: use **special atomic assembly instruction**
  - ➔ different forms: test and set, compare and swap, …
- **Test and set**

```
function test_and_set(x) is equivalent to
  y  = *x // temporary variable
  *x = 1 // memory @ address x
  return y
end
```

  - ➔ but executed as a **single atomic instruction**
  - ➔ the 3 lines above cannot overlap on an SMP machine

Spin Lock Version 4

- Keep writing 1s, until 0 is returned as previous value

```
object spin_lock is
  flag = 0

  method lock() is
    while (test_and_set(flag)!=0) do {
    } // spin
  end

  method unlock() is
    flag = 0  // release lock
  end
end
```

➔ That's it! But are we done?

➔ What problems can you see?

Pb.1: Spin Lock + Interrupts

- 1st pb: In a kernel, we usually want to disable interrupts
  - ➔ needed to prevents interrupt handlers on local core
  - ➔ needed when IHs use the same lock
  - ➔ note: cases when not needed
    so different variants of spin locks available

- How would you do this?

Spin Lock Version 5a

```
object spin_lock is
  flag = 0
  // following assumes interrupts don't touch flag
  method lock() is
    while (test_and_set(flag)!=0) do {
    } // spin
    disable_interrupts()
  end
  method unlock() is
    flag = 0  // release lock
    enable_interrupts()
  end
end
```

How to fix this?

- But this code is problematic
  - ➔ context switch possible just after test_and_set(..)
  - ➔ if control passes to handler that tries to take same lock danger of deadlock

Spin Lock Version 5b

- Better solution
  - ➔ once lock taken, no context switch possible

```
object spin_lock is
  flag = 0
  method lock() is
    label try_again:
      disable_interrupts()
      if (test_and_set(flag)!=0) do {
        enable_interrupts()
        goto try_again
      } // spin
  end
  method unlock() is
    flag = 0  // release lock
    enable_interrupts()
  end
end
```

Pb.2: Dangers of Spin Locks

- 2nd pb: spinning means
  - ➔ The relevant core is used 100% while the thread waits
  - ➔ If unlock() does not occur: the whole core is lost

- So the rules of use = pretty similar to v.3 (basic_locks)
  - ➔ spin locks **only used in kernel code**
  - ➔ for **small snippets** of code
  - ➔ that is guaranteed to **finish, and finish fast**

In Linux Code in Practice

- Linux Kernel provides a spin_lock API
  - ➔ with variants that disable or not local interrupts
  - ➔ which can be used on SMP and uniprocessor machine

- Conditional compilation (in C) is used so that
  - ➔ on SMP machines code v5 is used
  - ➔ on UP machines code v3 is used

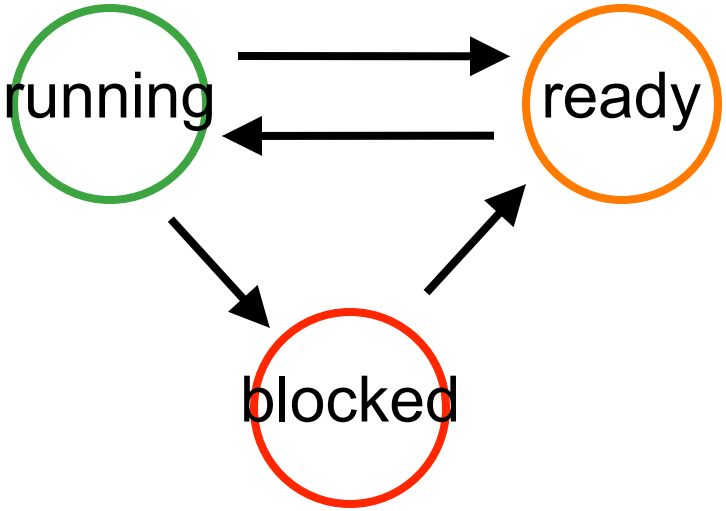
We need more than spin!

- Spin locks limited to low level and very short ops
  - ➔ very inefficient if critical section long
  - ➔ dangerous when interrupts disabled

- 2nd type of locks: **blocking locks**
  - ➔ while waiting → thread is put in a blocked state
  - ➔ works together with OS scheduler (see OS module)
  - ➔ that's how locks provided to users are implemented
  - ➔ and this uses in turn spin_locks() internally!

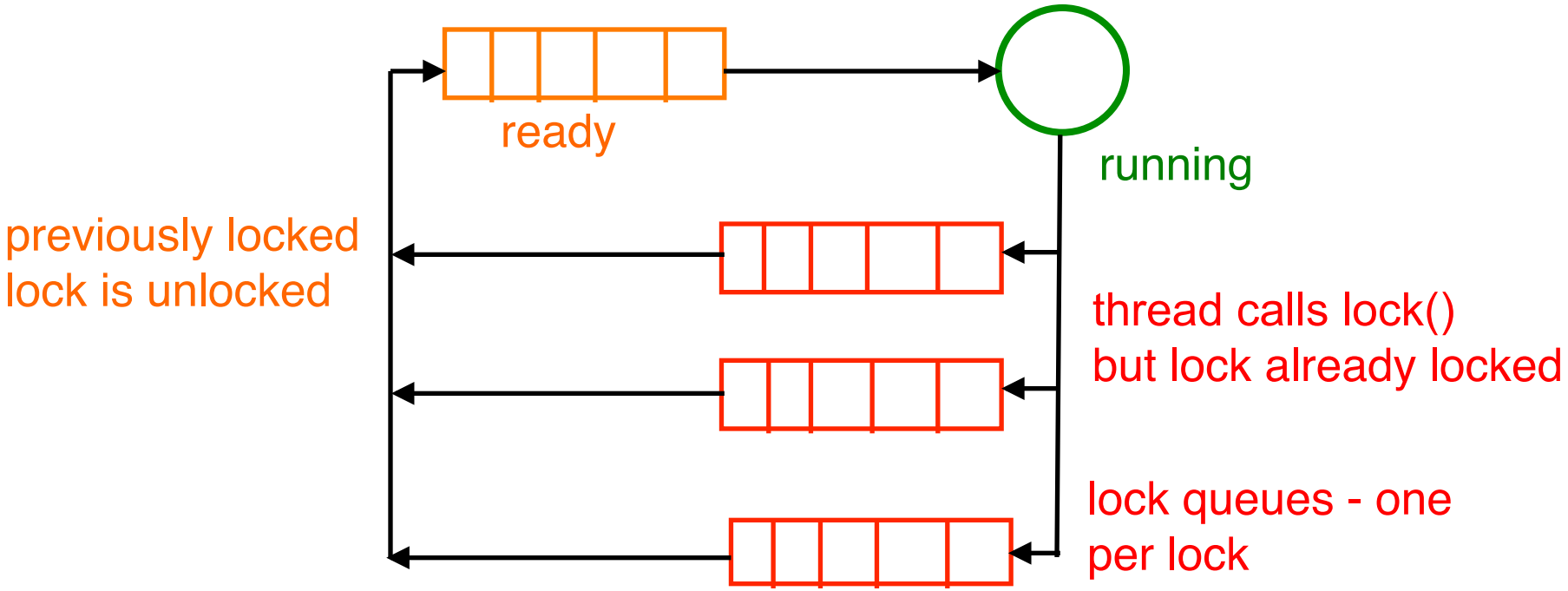
Reminder: OS Scheduler

- Each process / thread can be in 3 states
  - ➔ running (has CPU), ready (can run), and blocked
- OS scheduler in charge of
  - ➔ switching threads between states
  - ➔ which thread should run next on which core



Implementing Blocking Locks

- Each lock associated with a queue



➔ Why do we need spin locks here at all?

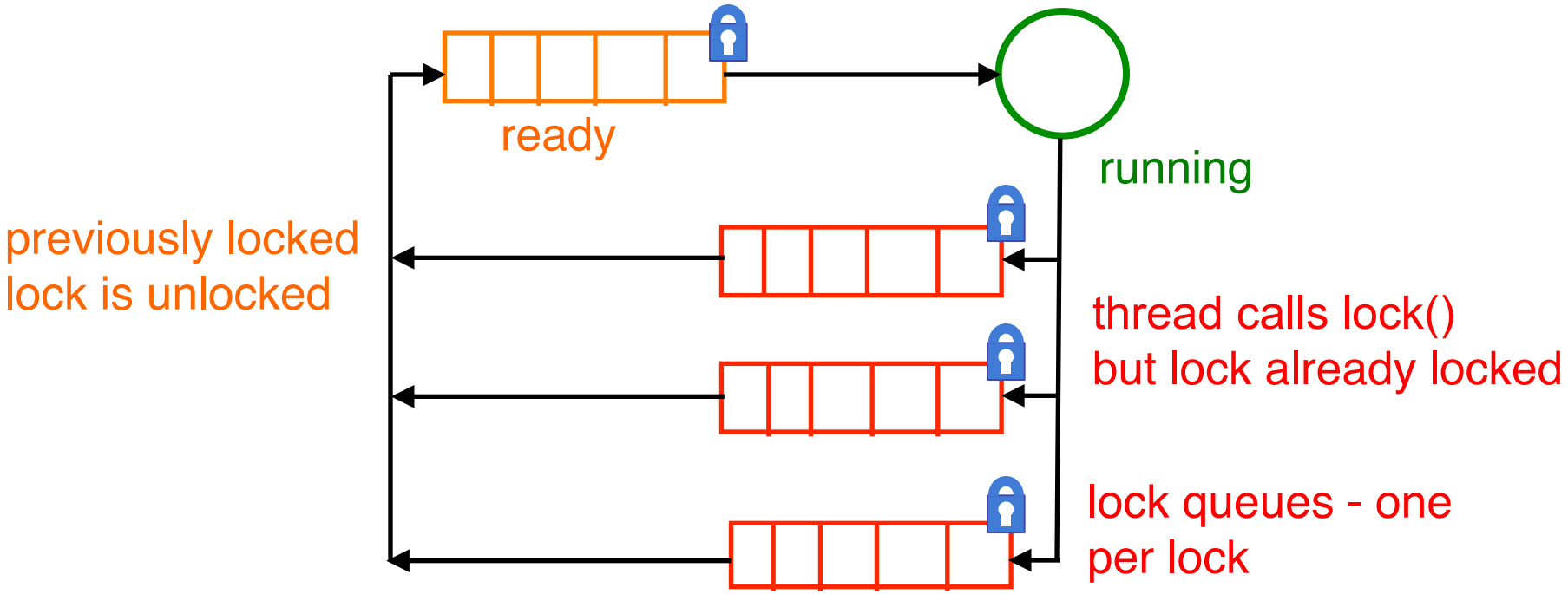
Implementing Blocking Locks

- Queues are accessed concurrently
  - ➔ they need to be protected
  - ➔ (a case of producer / consumer, see unit 4)
  - ➔ but we cannot use blocking locks!

- Solution: use one spin lock per queue
  - ➔ queue operations fast: only manipulating pointers

Blocking Locks

- Each queue associated with a spin lock



Blocking Locks: Example

```
object blocking_lock is
  queue  = [ ]                  // empty at start
  s_lock = new spin_lock()      // to protect flag & queue
  flag   = 0                    // 0 means lock is free
  method lock() is
    s_lock.lock()
    while(flag!=0) { // QUIZZ: Why while instead of if?
      add thread to queue
      this_thread = BLOCKED
      s_lock.unlock()
      activate_OS_scheduler()
      s_lock.lock()
    }
    flag=1
    s_lock.unlock()
  end
  method unlock() is
    // EXERCISE: What does unlock look like?
  end
end
```

here current thread suspended, stops being scheduled by OS

only resumes here after current thread has been de-queued

Blocking Locks: Example

- Unlock simpler
  - ➔ just de-queue one blocked thread
  - ➔ put flag back to zero
  - ➔ put de-queued thread in waiting state
  - ➔ OS scheduler does the rest

```
method unlock() is
  s_lock.lock()
  flag=0
  lucky_thread = get one thread from queue
  lucky_thread = READY // now considered by scheduler
  s_lock.unlock()
end
```

Summary

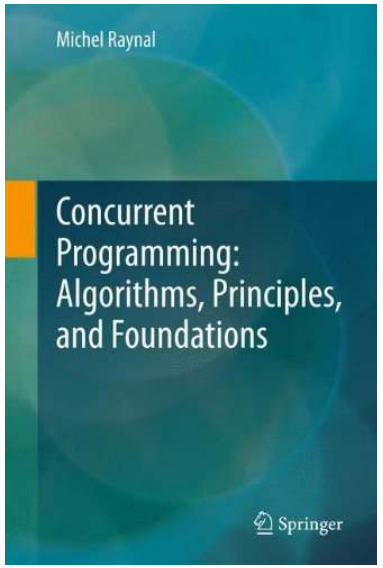
- We have seen three types of locks
  - ➔ Spin locks, basic locks, and blocking locks
- We have seen when each type of locks is used
  - ➔ spin and basic locks for low level short synchronisation
  - ➔ spin locks on SMP machines, basic locks on UP ones
  - ➔ blocking locks are the usual locks a programmer uses
- Spin locks are implemented using special assembly
  - ➔ single instruction that is atomic e.g. test and set
- Blocking locks use spin locks internally
  - ➔ but not for the whole critical section

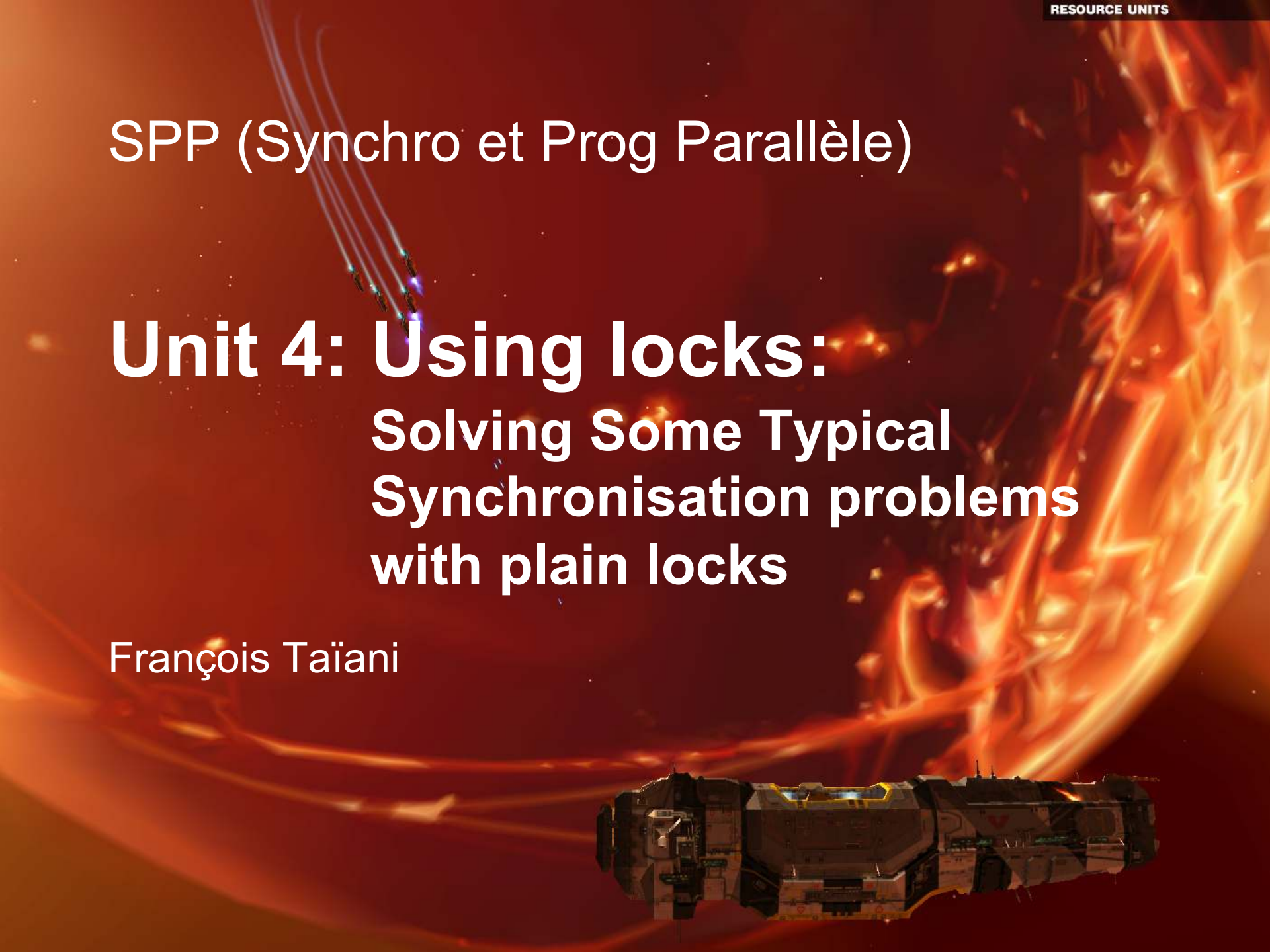
To Look Further

- **Kernel Locking Techniques** by Robert Love
  - ➔ http://www.linuxjournal.com/article/5833
  - ➔ http://james.bond.edu.au/courses/inft73626@033/Assigs/Papers/kernel_locking_techniques.html
- **Linux Spinlock Implementation** by Ted Baker
  - ➔ http://www.cs.fsu.edu/~xyuan/cop5611/spinlock.html
- **Spin locks doc from kernel code** by Linus Torvald
  - ➔ https://www.kernel.org/doc/Documentation/locking/spinlocks.txt
- Note:
  - ➔ Linux constantly evolving so exact details might change
  - ➔ but good example of implementation issues in OS

To Look Further (cont.)

- Section 5.5 of **Linux Device Drivers, 3rd Edition** By J. Corbet, G. Kroah-Hartman, A. Rubini
  - ➔ **"Spinlocks"**
  - ➔ http://www.makelinux.net/ldd3/chp-5-sect-5
- **Concurrent Programming: Algorithms, Principles, and Foundations** by Michel Raynal
  - ➔ Springer; 2012 edition
    ISBN: 978-3642320262
  - ➔ Chapter 1 "The Mutual Exclusion Problem"
  - ➔ Chapter 2 "Solving Mutual Exclusion"





SPP (Synchro et Prog Parallèle)

Unit 4: Using locks:
Solving Some Typical Synchronisation problems with plain locks

François Taïani

Why look at typical problems?

- To recognise them when you see them
  - ➔ To be able to indentify and assess synchronisation needs
  - ➔ To be able to apply existing solutions
  - ➔ No need to reinvent the wheel when solutions exist
- To practice your understanding of locks
  - ➔ Locks can be tricky to use
- To organise your understanding of the field
  - ➔ How do problems and solutions relate to each other
  - ➔ A kind of mental map of what is out there

Two big families of problems

- When concurrent processes share resources
  - ➔ they can be in **competition** (contention)
  - ➔ or in **collaboration**
- **Competition**
  - ➔ amount of resources is limited
  - ➔ no all processes can use resources at the same time
  - ➔ synchronisation needed to decided **who use what when**
- **Collaboration**
  - ➔ processes work together to achieve a goal
  - ➔ their work is interdependent
  - ➔ synchronisation needed to control **interdependencies**

Examples

- Competition
  - ➔ 2 applications using the same printer
  - ➔ 2 users trying to book the same room at the same time
  - ➔ 2 players trying to pick a unique object in a game
- Collaboration
  - ➔ 1 application sending data to print to printer
  - ➔ 1 simulation providing results to another simulation
  - ➔ 1 player giving an object to another player in a game

Competition vs Collaboration

- Both often mixed in the same parallel program
  - ➔ see example with on-line game

- Solving 1 type of pb ➔ often need to solve other type
  - ➔ e.g. constructing a house: mainly collaboration
  - ➔ workers need to coordinate their work: workplan
  - ➔ but access to work plan is a competition problem

Typical problems

- What we have seen so far: **Mutual Exclusion**
  - ➔ a problem of competition
  - ➔ only one process / thread in the **critical section** at a time
  - ➔ we have seen one tool to solve it: **locks** (aka mutex)

- Today: Two typical problems of **collaboration**
  - ➔ **readers / writers** (simpler variant of getSum, transfer)
  - ➔ **producer / consumer**

Readers / Writers

- The context:
  - ➔ one variable
  - ➔ a number of threads write to it
  - ➔ a number of threads read from it (the latest value written)
  - ➔ e.g. a status variable (# pages printed, # files closed, etc.)
- Unsafe solution

```
shared int a

method read() is
  return a
end
```

```
method write(int x) is
  a = x
end
```

  - ➔ Quiz: How would you make it safe?

Readers / Writers

- Solutions
  - ➔ (1) use one plain lock to protect a: works but suboptimal
  - ➔ (2) use a read / write lock (aka shared lock, cf unit 2)

```
shared a // any resource, : image, string, etc
rw_lock l

method read() is
  l.lock_read()
  result = a // copy
  l.unlock()
  return result
end
```

```
method write(x) is
  l.lock_write()
  a = x
  l.unlock()
end
```

- Problem: What do you do if you don't have RW locks?

Readers / Writers

- Readers / writers problem reformulated:
  - ➔ one shared variable, with read and write operations
  - ➔ readers should block writers, but not other readers
  - ➔ writers should block both other writers and readers
  - ➔ **we only have plain locks** to solve the problem
- Start of a solution
  - ➔ **intuition**: readers should act "**collectively**"
  - ➔ keep count of # readers already accessing variable
  - ➔ 0 readers: variable accessible in writing and reading
  - ➔ ≥1 readers: variable only accessible in reading

RW Pb: First Attempt

```
shared a
shared int nb_readers = 0
lock l

method read() is
  if nb_readers==0 {
    l.lock()
  }
  nb_readers++
  result = a
  nb_readers--
  if nb_readers==0 {
    l.unlock()
  }
  return result
end
```

```
method write(x) is
  l.lock()
  a = x
  l.unlock()
end
```

- This code is unfortunately unsafe. Why?

2nd attempt

```
shared int a
shared int nb_readers = 0
lock l, count_l

method read() is
  if nb_readers==0 {
    l.lock()
  }
  count_l.lock()
  nb_readers++
  count_l.unlock()
  result = a
  count_l.lock()
  nb_readers--
  count_l.unlock()
  if nb_readers==0 {
    l.unlock()
  }
  return result
end
```

```
method write(x) is
  l.lock()
  a = x
  l.unlock()
end
```

- This code is still faulty. Why?

Comment

- We need to protect the 2 `if nb_readers==0`
  - ➔ the “==0” is not guaranteed to be atomic
  - ➔ even if it is, test + increment/decrement need to be
  - ➔ otherwise can lead to unsafe execution

2nd attempt (again)

```
shared int a
shared int nb_readers = 0
lock l, count_l

method read() is
  if nb_readers==0 {
    l.lock()
  }
  count_l.lock()
  nb_readers++
  count_l.unlock()
  result = a
  count_l.lock()
  nb_readers--
  count_l.unlock()
  if nb_readers==0 {
    l.unlock()
  }
  return result
end
```

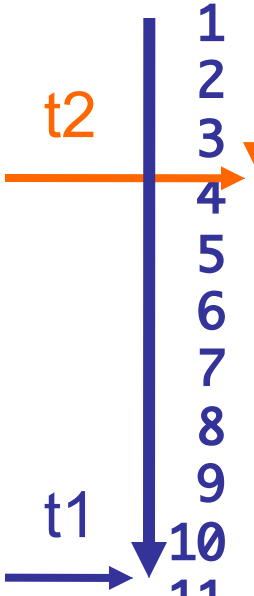
```
method write(x) is
  l.lock()
  a = x
  l.unlock()
end
```

- Exercise
  - ➔ find an example of unsafe execution even if ==0 atomic

Example of pb

```
shared int a
shared int nb_readers = 0
lock l, count_l

method read() is
1     if nb_readers==0 {
2       l.lock()
3     }
4     count_l.lock()
5     nb_readers++
6     count_l.unlock()
7     result = a
8     count_l.lock()
9     nb_readers--
10    count_l.unlock()
11    if nb_readers==0 {
12      l.unlock()
13    }
14    return result
end
```

- 2 threads t1, t2



t1:1,2,3,4,5,6,7
(l=locked,nb_readers==1)

t2:1,3
(l=locked,nb_readers==1)

t1:8,9,10,11,12,13,14)
(l=free,nb_readers==0)

t2:4,5,6
(l=free,nb_readers==1)

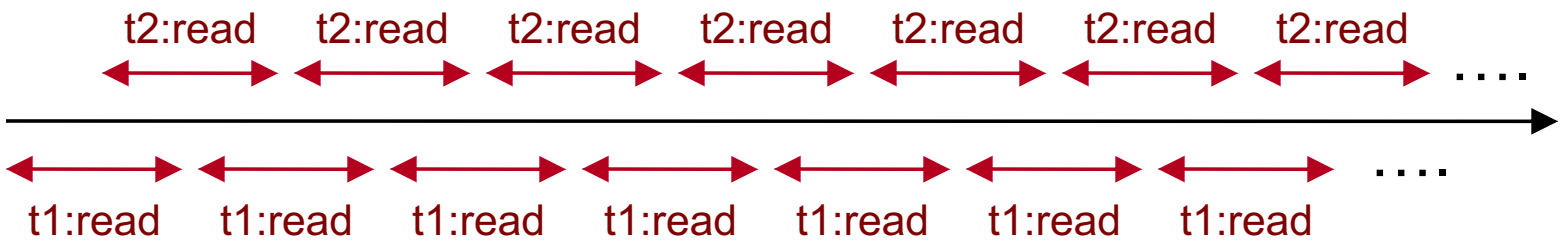
t2 about to access variable a but l is unlocked!!

Corrected Attempt: Solution 1

```
shared int a
shared int nb_readers = 0
lock l, count_l

method read() is
   count_l.lock()
   if nb_readers==0 {          // atomic
     l.lock()
   }
   nb_readers++                // atomic
   count_l.unlock()
   result = a
   count_l.lock()
   nb_readers--                // atomic
   if nb_readers==0 {          // atomic
     l.unlock()
   }
   count_l.unlock()            // atomic
   return result
end

method write(x) is
   l.lock()
   a = x
   l.unlock()
end
```

Improving Solution 1

- Consider two threads doing reads:



➔ What happens if a 3rd thread tries to do a write op?

Improving Solution 1: Fairness

- Solution 1 can lead to the starvation of writers
  - ➔ if reads are continuous and overlap
    then nb_readers always > 0
    and lock l never released
  - ➔ it's an unfair read / write lock (see Unit 2)
- How to solve this?
  - ➔ hypothesis: assume underlying locks are fair
  - ➔ intuition: get writer to prevent readers to continue
    i.e. get writer to get into the "queue" of readers
- How would you do this with 4 extra lines?

Fair and Safe: Solution 2

```
shared int a
shared int nb_readers = 0
lock l, count_l, fair_access

method read() is
   fair_access.lock()
   count_l.lock()
   if nb_readers==0 {
     l.lock() }
   nb_readers++
   count_l.unlock()
   fair_access.unlock()
   result = a
   count_l.lock()
   nb_readers--
   if nb_readers==0 {
     l.unlock() }
   count_l.unlock()
   return result
end

method write(x) is
   fair_access.lock()
   l.lock()
   a = x
   l.unlock()
   fair_access.unlock()
end
```

FIFO FIFO FIFO FIFO

- variable access in order in which `fair_access` taken

Comments

- We have a solution to the readers/writers problem
- This also gives an implementation of read/write locks
  - ➔ note the two unlock operations (as in Java btw)
  - ➔ merging 2 unlock operations requires a bit of leg work (usually not worth the extra complexity)

Producers / Consumers

- 2nd typical collaboration synchronisation problem
  - ➔ one or more processes produce data
  - ➔ another or more processes consume this data
  - ➔ they communicate through a shared queue
- Different from previous example
  - ➔ no data should get lost
  - ➔ if the queue is empty, consumers should block
  - ➔ if the queue is full, producers should block
- We have already seen such a situation
  - ➔ scheduling queues in the blocking locks implementation

Producers / Consumers

- Assumptions: We have a bounded queue object with
  - ➔ add: add to the end of the queue
  - ➔ get: removed from the beginning of the queue (FIFO)
  - ➔ empty?: true if queue is empty
  - ➔ full?: true if queue is full
  - ➔ size: nb of elements currently in queue
- The queue is not thread-safe
  - ➔ concurrent operations can yield arbitrary results

Prod/Cons: 1st Attempt

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  result = q.get()
  l.unlock()
  return result
end
```

- What is the problem with this?

Prod/Cons: 1st Attempt

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  result = q.get()
  l.unlock()
  return result
end
```

- Empty and full cases not taken into account
  - ➔ empty: consumers should wait until item available
  - ➔ full: producers should wait until free slot available

Prod/Cons: 2nd Attempt

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  while (q.full?) {
  }
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  while (q.empty?) {
  }
  result = q.get()
  l.unlock()
  return result
end
```

- We now test if q is full (resp. empty)
  - ➔ But this is incorrect. Why? How would you correct it?

3rd Attempt: Solution 1

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  while (q.full?) {
    l.unlock()
    l.lock()
  }
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  while (q.empty?) {
    l.unlock()
    l.lock()
  }
  result = q.get()
  l.unlock()
  return result
end
```

- This now works correctly. (Safe and lively)
  - ➔ However not optimal? Find 2 reasons why.

Solution 1b with yield

- Problems with solution 1
  - 1. busy waiting: use yield (1b) to mitigate (but not eliminate!)
  - 2. does not allow concurrent produce / consume

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  while (q.full?) {
    l.unlock()
    yield()
    l.lock()
  }
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  while (q.empty?) {
    l.unlock()
    yield()
    l.lock()
  }
  result = q.get()
  l.unlock()
  return result
end
```

Summary

- Two general types of synchronisation problems
  - ➔ competition (mutual exclusion)
  - ➔ collaboration (readers and writers, consumer/producers)
- Lock-based solutions of
  - ➔ readers and writers (explain how RW locks are done)
  - ➔ consumer/producers (aka bounded buffer)
- Note on consumer/producers
  - ➔ uses busy waiting
  - ➔ we will see a better solution with semaphores in unit 5

SPP (Synchro et Prog Parallèle)

Unit 5: Beyond Locks: Semaphores and Monitors

François Taïani

Outline

- Semaphores
  - Definition
  - Semaphore Invariant
  - Semaphores and locks
  - Cons/Prod with Semaphores
  - Java API
- Monitors
  - Definition, Wait + Signal
  - Precise Semantic of Signal
  - In Java

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

Semaphores

- 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

- Each semaphore guarantees that
  - ➔ 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 empty cells
  - ➔ one to count full 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

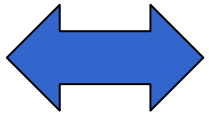
Monitors

- 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 (provided as a method)
  - ➔ releases monitor
  - ➔ blocks current thread (“asleep”, not schedulable)
- **signal**() operation (also known as **notify**() e.g. Java)
  - ➔ 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

```
1 monitor semaphore
2  int count = init_value
3  method up() is
4     count++
5     signal()
6  end
7  method down() is
8     if count==0 { wait() }
9     count--
A  end
B 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)

SPP (Synchro et Prog Parallèle)

Unit 6: Conditions, Barriers, and RendezVous

François Taïani

So far …

- We have seen several synchronisation mechanisms
  - ➔ locks (plain, re-entrant, read/write)
  - ➔ semaphores
  - ➔ monitors
- And several implementations of these mechanisms
  - ➔ spinning locks, blocking locks
  - ➔ locks using semaphores
  - ➔ semaphores using locks, using monitors
- Today: 3 more mechanisms
  - ➔ condition variables, barriers, rendez-vous

Why more?

- Same philosophy as monitors
  - ➔ monitors help structure concurrent programs
  - ➔ provide implicit locking, and wait/signal mechanism
  - ➔ (note: locking & waiting quite different, why?)

- Condition variables
  - ➔ extension of the wait/signal mechanism we have seen
  - ➔ waiting on a particular condition to be fulfilled
- Barrier
  - ➔ getting threads to wait on each other
- RendezVous
  - ➔ getting 2 threads to wait on each other and exchange data

Condition Variables

- Revisiting the producer / consumer pb with monitors

```
monitor ConcurrentBoundedBuffer is
  Buffer b

  method produce(x) is
    while(b.full?) {
      wait()
    }
    b.add(x)
    notifyAll()
  end

  method consume() is
    while(b.empty?) {
      wait()
    }
    result = b.get()
    notifyAll()
    return result
  end

end
```

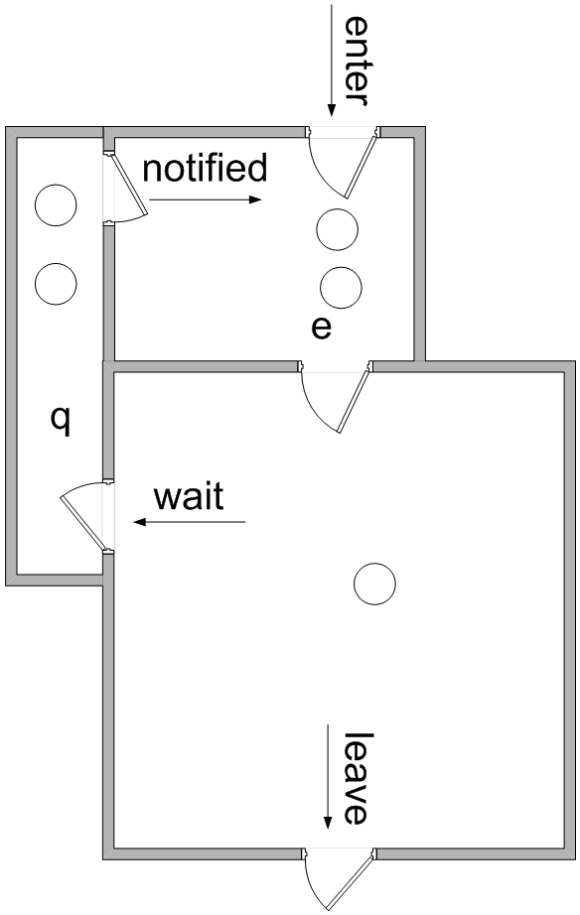
➔ revision: Why do we need notifyAll?
(Explain in a few sentences)

Avoiding notifyAll

- In previous example, we need notifyAll because
  - ➔ monitor queue might contain both producers & consumers
  - ➔ in this case a plain notify might wake up the wrong thread
  - ➔ (e.g. a consumer while the queue is empty)
  - ➔ this thread will go back to sleep
  - ➔ and the program will block, although it should progress
- Fundamental reason
  - ➔ one single queue for two situations: b.full, b.empty
  - ➔ notify (aka signal) does not know which thread to pick
  - ➔ notifyAll quite waste-full: many threads will just go back to sleep

Solution: From 1 waiting queue

- Monitors in Java: One single queue, unnamed

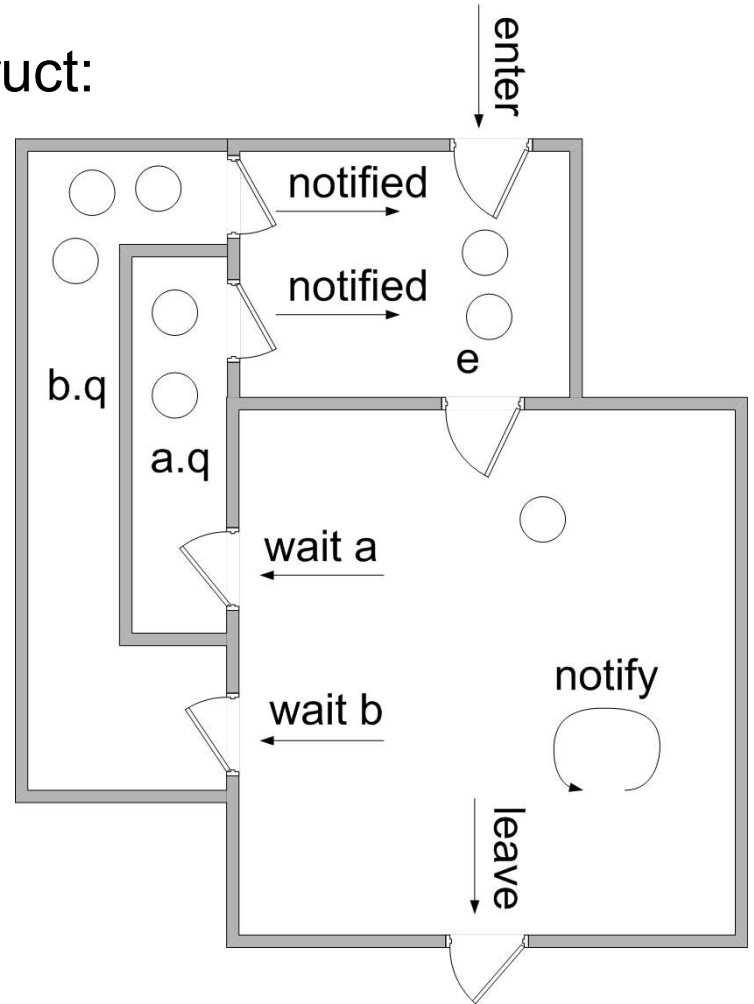


source: wikipedia, © Theodore.norvell

Solution: … to several queues

- Monitor with condition variables
  - ➔ several queues
  - ➔ access through special construct: condition variables

- Here: 2 condition variables
  - ➔ a and b
  - ➔ can do “wait a” or “wait b”
  - ➔ “notify a” or “notify b”



source: wikipedia, © Theodore.norvell

Condition Variables

A structure that encapsulates a wait/notify mechanism

- created within a monitor
  - ➔ e.g. Condition myCondition
- once created, 2 operations: wait, and notify (or signal)
  - ➔ (and often notifyAll/ signalAll as well)
- associated with an implicit waiting queue
- same mechanism as unamed queues
  - ➔ wait: relinquishes monitor, and puts thread in waiting queue
  - ➔ notify: wakes up one of the threads in the waiting queue
- but allows several queues in the same monitor
  - ➔ wait and notify do not impact other condition variables

Revisiting Producer Consumer

- With condition variables (aka *Conditions*)
  - ➔ different types of thread in different waiting queues

- Exercise:
  - ➔ Write the code of of multithreaded bounded buffer that uses distinct condition variables for each buffer conditions.

Revisiting Producer Consumer

- With condition variables (aka *Conditions*)
  - ➔ different types of thread in different waiting queues

```
monitor ConcurrentBoundedBuffer is
  Buffer b
  Condition bIsNotFull
  Condition bIsNotEmpty

  method produce(x) is
    while(b.full?) {
      bIsNotFull.wait()
    }
    b.add(x)
    bIsNotEmpty.notify()
  end

  method consume() is
    while(b.empty?) {
      bIsNotEmpty.wait()
    }
    result = b.get()
    bIsNotFull.notify()
    return result
  end

end
```

Conditions Variables in Java

- Slightly different from “traditional” condition variables
  - ➔ not associated with a monitor
  - ➔ but directly associated with a lock
  - ➔ operations called **await** & **signal** (instead of wait & notify)
- Result is the same
  - ➔ thread must hold the lock to call wait or signal
  - ➔ await() releases the lock instead of the monitor

Example in Java

```
public class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
```

Example in Java

```
public class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr = 0, takeptr = 0, count = 0;
```

Example in Java

```
public class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr = 0, takeptr = 0, count = 0;

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {









    } finally {
      lock.unlock();
    }
  }
...
```

Example in Java

```
public class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr = 0, takeptr = 0, count = 0;

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();




    } finally {
      lock.unlock();
    }
  }
...
```

Example in Java

```
public class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr = 0, takeptr = 0, count = 0;

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;


    } finally {
      lock.unlock();
    }
  }
...
```

Example in Java

```
public class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr = 0, takeptr = 0, count = 0;

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
...
```

Barriers

- Condition variables in monitors
  - ➔ waiting for a particular state to occur
- But sometimes more natural to wait for threads
  - ➔ e.g. N threads to complete some work
  - ➔ only after all have finished can the next phase start
- Often the case in scientific computation
  - ➔ several phases
  - ➔ data exchange between threads between phases
  - ➔ next phase can only start if previous one has finished

Barriers

- A special construct initialised for N threads
  - ➔ N is fixed, set when barrier is created
- One operation:
  - ➔ barrier.wait() (or await)

Barriers in Java

- **class java.util.concurrent.CyclicBarrier**
  - ➔ "cyclic" because reusable several times
  - ➔ constructor contains the number of threads to wait for **CyclicBarrier**(int parties)
  - ➔ main method: **await**()
  - ➔ + variants (with timeout)
  - ➔ + bookkeeping (nb of threads waiting etc.)
- Related mechanisms (more general)
  - ➔ java.util.concurrent.CountDownLatch
  - ➔ java.util.concurrent.Phaser

RendezVous

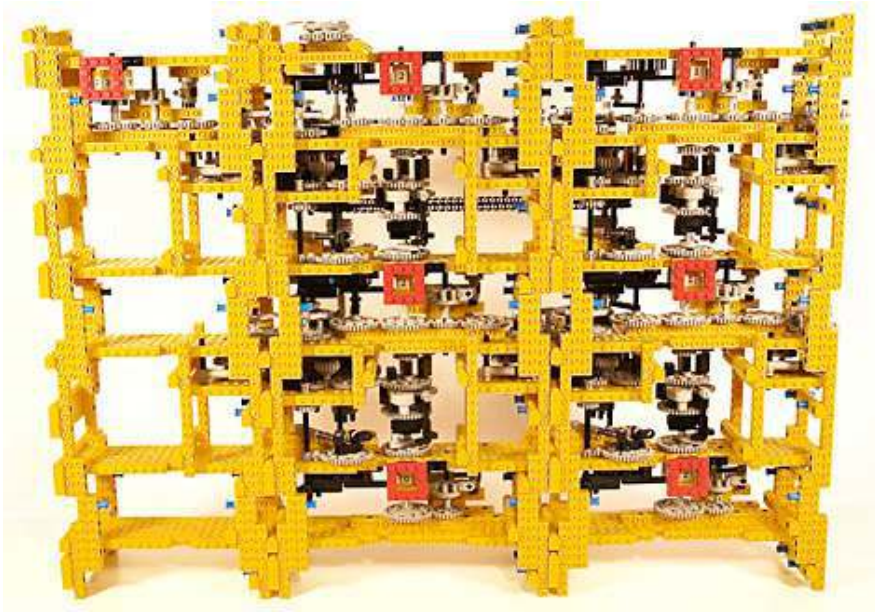
- Related to barriers, except
  - ➔ only two threads
  - ➔ data is exchanged when both reach the rdv point
- Original mechanism (in Ada)
  - ➔ one thread produces the data (“accept” keyword)
  - ➔ other thread consumes it (plain invocation)
  - ➔ if invocation before accept -> caller waits
  - ➔ if “accept” before invocation -> callee waits
- In Java
  - ➔ symmetric role of threads:
    class java.util.concurrent.Exchanger<V>

Note on Ada

- Winner of competition launched by US DoD in 1978
  - ➔ originally for embedded mission-critical systems
  - ➔ but very wide spectrum, many innovations
- Designed by a French team (CII Honeywell Bull)
  - ➔ led by Jean Ichbiah (1940-2007)

Note on Ada

- Augusta Ada King, Countess of Lovelace (1815-1852)
  - ➔ worked with Charles Babbage
  - ➔ considered first programmer ever



Summary

- Three new synchronisation mechanisms
  - ➔ condition variables (with monitors)
  - ➔ barriers
  - ➔ rendezVous
- Goal: propose toolset of synchronisation patterns
  - ➔ essentially for cooperation
  - ➔ optimised implementations proposed over the years
  - ➔ so you can focus on the rest of your program

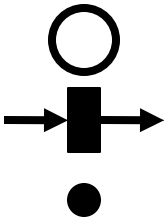
SPP (Synchro et Prog Parallèle)

Unit 7: Petri Nets

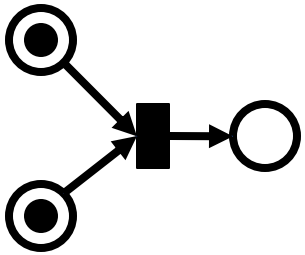
François Taïani

Petri Nets

- Invented by Carl Adam Petri (1939)
- Main elements
  - places
  - transitions
  - tokens



- Transitions consume and produce tokens
  - consume one or more tokens from one or more places
  - produces one or more tokens into one or more places

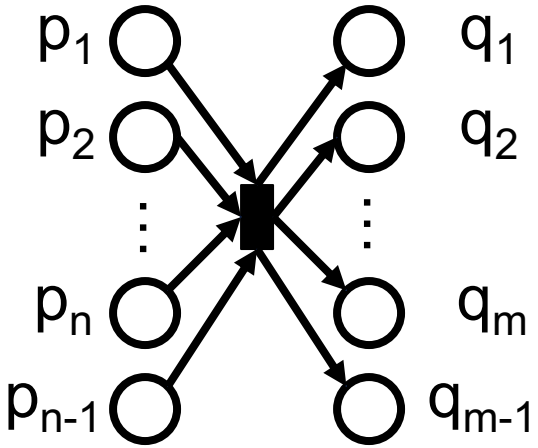


Tokens

- Tokens & places
  - ➔ One place can contain 0, 1, 2, … tokens
- Token & Transitions
  - ➔ Transitions need tokens to fire
  - ➔ Tokens need transitions to arrive in a place
- Meaning: Tokens
  - ➔ represent the dynamic part of a Petri net
  - ➔ model either resources (produced and consumed)
  - ➔ or the execution pointer of a program

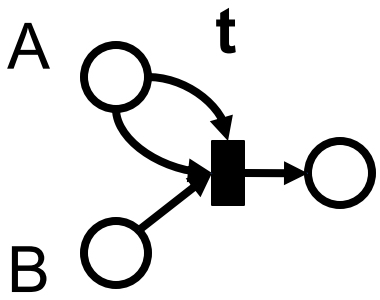
Transitions

- Transitions
  - ➔ link a set of *input places* to a set of *output places*
  - ➔ a transition needs to be *enabled* before it can fire
  - ➔ enabling = all input places contain one token
  - ➔ firing = input places lose 1 token, output places gain 1

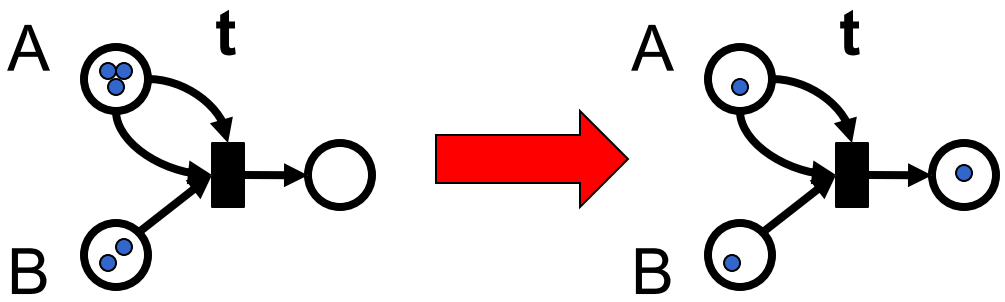


Multiple Edges

- Multiple edges possible between a place & a transition



➔ Transition **t** needs two tokens in **A** and one in **B** to fire

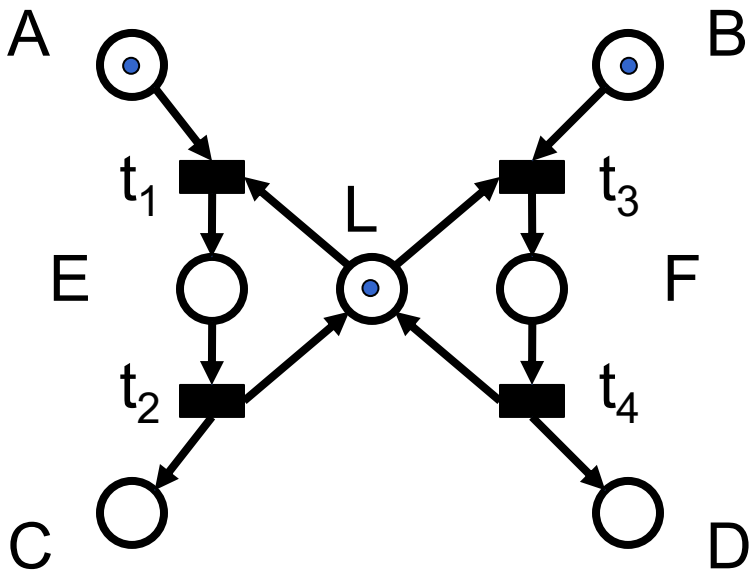


Executing a Petri Net

- While some transitions are enabled
  - ➔ select one of the enabled transitions
  - ➔ fire this transitions (e.g. update tokens)
- Vocabulary
  - ➔ distribution of tokens in places: “marking”
  - ➔ start marking = initial marking (M_0)

Small Exercise

- What are the possible executions of this Petri net?

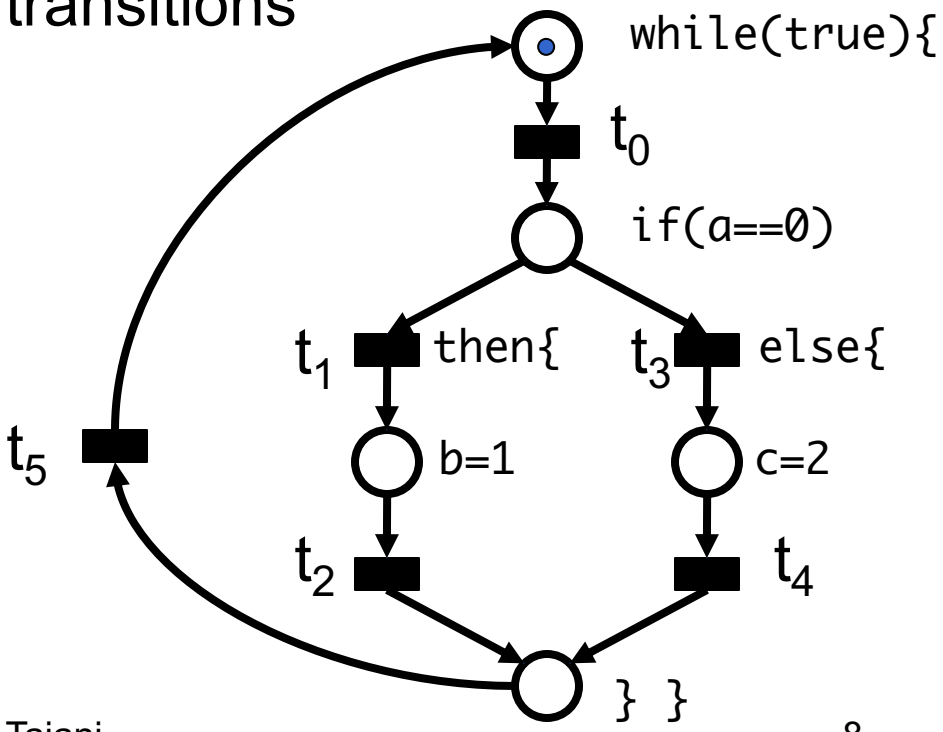


Link with Parallelism

Petri nets can be used to model flow of parallel code

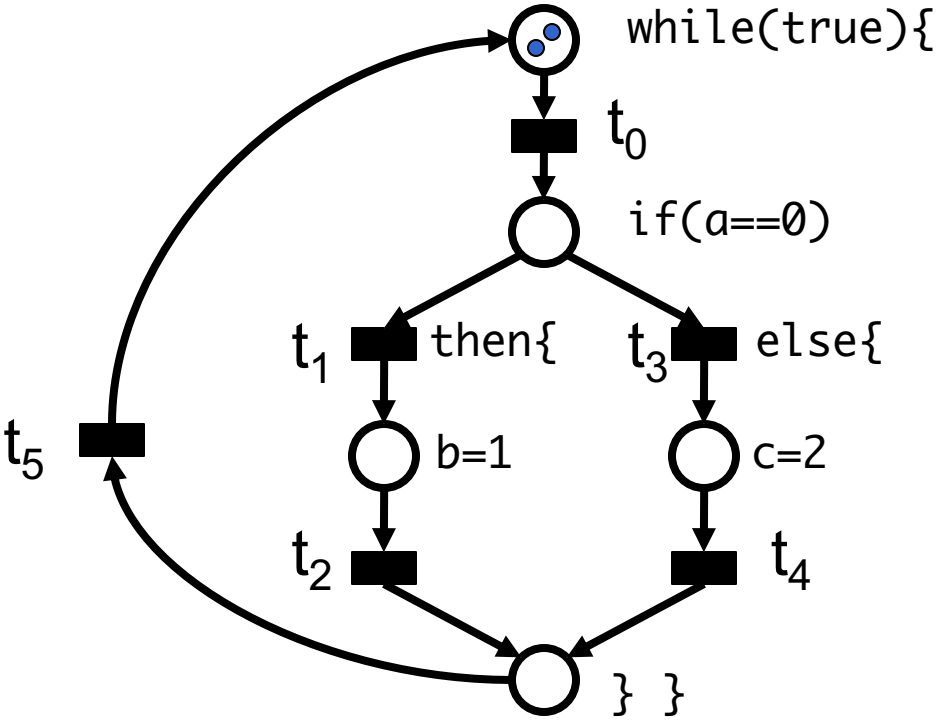
- Sequential code -> sequence of transitions
- Loops -> transition looping back
- Conditions -> alternative transitions

```
while (true) {
  if (a==0) then {
    b = 1
  } else {
    c = 2
  }
}
```



Modelling Multithreading

- Several tokens!
  - ➔ e.g. here for 2 threads



Note on previous example

- Not all aspects of program modelled
  - ➔ in program if condition is deterministic
  - ➔ in Petri net: random: one or the other might happen
- Consequence
  - ➔ PN runs = superset of real runs
  - ➔ Some Petri net runs might not be possible in real code
  - ➔ Important to keep in mind when doing analysis

Approximation of Modelling

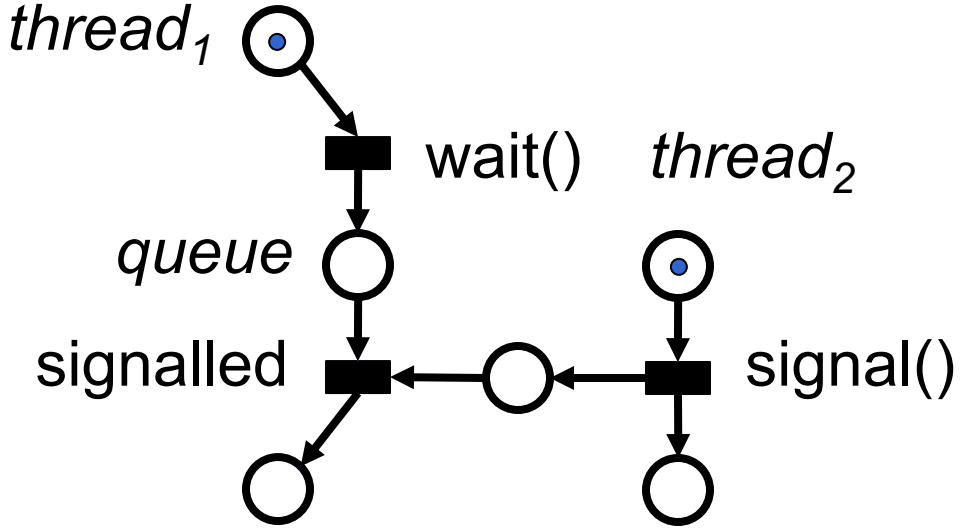
- Most often {code runs} ⊊ {PN runs}
- Implication: when {code runs} ⊊ {PN runs} then
  - ➔ all PN runs “safe” -> all application runs “safe”
  - ➔ some PN runs “unsafe” -> application might be unsafe
  - ➔ some PN runs block -> application might block
  - ➔ all PN runs lively -> ?
- (Note: Common issue in formal analysis)
- In cases when code runs = PN runs (mod congruence)
  - ➔ safety, un-safety, deadlocks, and liveliness translate

Modelling Resources

- Simply as a shared place with n tokens
  - ➔ n = number of resources
- a lock = one place with one token
  - ➔ lock: consuming transition going out from place
  - ➔ unlock: producing transition coming into place
- a semaphore = one place with n tokens
  - ➔ down: consuming transition going out from place
  - ➔ up: producing transition coming into place

Modelling Synchronisation

- Condition variables can be approximated
  - ➔ waiting queue = a place, waiting threads: tokens in place
  - ➔ “wait()” moving to queue
  - ➔ “signal()” enabling out transition



- **Quiz:** What are the problems with this model?

Problem with previous model

- If waiting queue is empty when signal() triggered
  - ➔ the PN model remembers the signal operation
  - ➔ a real condition variable would not
  - ➔ OK is can prove this never happens
  - ➔ otherwise PN extension needed (zero testing)
- Impossible to realise “notifyAll”
  - ➔ possible (but complex) with zero testing
- Fundamental reason
  - ➔ PN are not Turing Complete
  - ➔ (They are with zero-testing)

Exercise

- Ping-Pong example from TD
  - ➔ What assumption are we making on locks?

```
shared lock l1, l2
init() { l2.lock() } // l2 initialised as locked

thread t1 is
  while(true) {
    l1.lock()
    println("ping”)
    l2.unlock()
  }
end
```

```
thread t2 is
  while(true) {
    l2.lock()
    println("pong")
    l1.unlock()
  }
end
```

- Exercise: Represent above program as a Petri Net
  - ➔ Is there a 1-1 mapping between your PN and the program?

Analysis Petri Nets

- Possible to enumerate reachable markings
  - ➔ markings linked to each other through transitions
  - ➔ result = *reachability graph* (might be infinite)
- Depends on where PN starts: M_0
  - ➔ $R(M_0)$ = all markings reachable from M_0
- Allow us to answer reachability questions
  - ➔ are bad states reachable -> (possibly) unsafe code
  - ➔ no bad state reachable -> safe code
  - ➔ some desirable state becomes unreachable
    -> (possibly) unlively code

Example with Ping and Pong

Bad state

- places for println("ping”) and println("ping”) both taken
  - ➔ violation of mutual exclusion, unsafe

Desirable states

- states always reachable where println("ping”) taken
  - ➔ ping thread never blocked, lively
- states always reachable where println(“pong”) taken
  - ➔ pong thread never blocked, lively

Special properties (for this example)

- ping and pong states alternate

Exercise

- Construct reachability graph of ping / pong example
  - ➔ Is the program safe?
  - ➔ Is the program lively?

Matrix interpretation

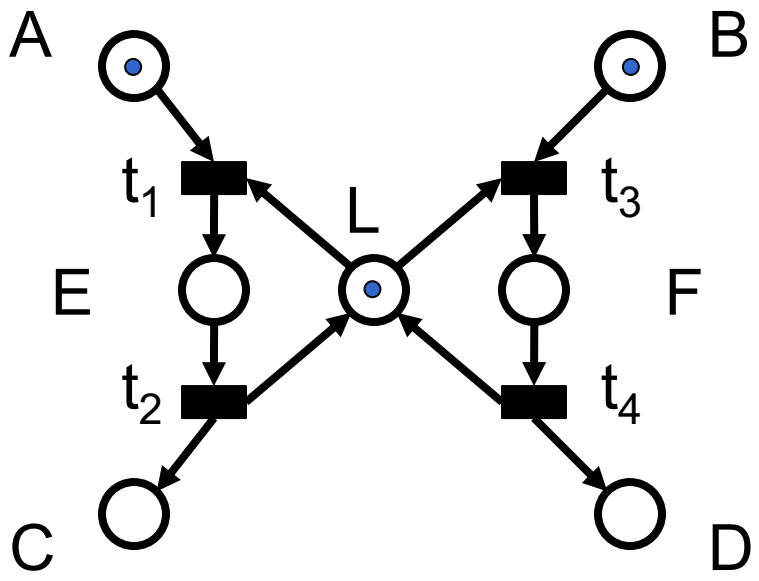
- A marking in a Petri net
  - ➔ a vector: one entry per place
- A transition: also a vector
  - ➔ -x in places where consuming
  - ➔ +y in places where producing
- A complete Petri net: Transition matrix M
  - ➔ m places and n transitions
  - ➔ m rows × n columns
  - ➔ each column: one transition
  - ➔ each row: change to place for each transition
    (note: some definition exchange rows and columns)

Computing with Matrix Rep

- If $v_0 = v(M_0)$ is initial vector
- $t_{i1}, t_{i2}, t_{i3}, \ldots t_{ik}$ sequence of transitions fired
  - ➔ T = [nb_t_1,
    nb_t_2,
    …] vector of transitions fired
  - ➔ nb_t_i = nb of times t_i appears in sequence
- Marking reached is
  - ➔ $v_k = v_0 + M \times T$
  - ➔ provided no place is ever negative during sequence (i.e. the sequence is actually possible)

Exercise

- Write down the transition matrix for the following PN
- Compute marking if t1, t2, t3, t4 are all fired



Extensions to PN

- We have seen one: zero-testing
- Others
  - ➔ inhibitor arcs (prevent transitions)
  - ➔ coloured PNs (tokens of different types)
  - ➔ hierarchical PNs, object PNs
  - ➔ timed PNs, stochastic PNs
- Software (many!)
  - ➔ http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html
- Link to similar formalisms:
  - ➔ Statecharts, UML state machines

Summary

- Petri Nets: Powerful formalism to model concurrency
  - ➔ sequential flow
  - ➔ tests (if), loops (while, for)
  - ➔ locks, semaphore, synchronisation (with approximation)
- Can be used for automatic analysis
  - ➔ reachability graph
  - ➔ link to linear algebra and in particular matrix algebra
- Can help detect:
  - ➔ unsafe states
  - ➔ deadlocks
  - ➔ lack of liveliness

SPP (Synchro et Prog Parallèle)

Unit 8: Immutability & Actors

François Taïani

Questioning Locks

- Why do we need locks on data?
  - ➔ because concurrent accesses can lead to wrong outcome
- But not all concurrent accesses problematic
  - ➔ concurrent writes -> potential corruption
  - ➔ concurrent reads + 1 write -> potential inconsistent obs
  - ➔ concurrent reads (no writes) -> OK
- Hence the idea
  - ➔ avoid the writes!
  - ➔ the extreme way: functional programming
  - ➔ less extreme: immutability

Functional Programming

- Lambda calculus in a readable form
- Purely functional languages
  - ➔ mainly Haskell (www.haskell.org)
  - ➔ functions = functions in mathematical term
  - ➔ one input -> always same output
  - ➔ no side effect (no “hidden state” that is modified)
- Implications
  - ➔ variables and their content are immutable
  - ➔ once value has been assigned, cannot change
- Wide ranging influence: see HFS, GSF
  - ➔ Hadoop file system, Google File System

Other Functional Languages

- With side effects / mutable fields in special cases
  - ➔ ML
  - ➔ OCaml (originally derived from ML)

- Hybrid
  - ➔ mix ideas of functional, imperative, and OO languages
  - ➔ the largest class
  - ➔ examples: Scala, Ruby, Erlang, F#, Groovy

Example

- Haskell Factorial
  - `factorial n = if n==0 then 1 else n * factorial (n-1)`
  - once n bound in a scope (invocation), no longer changes

Interest for Concurrency

- Functional languages encourage stateless code
  - ➔ no state, no risk of corruption
  - ➔ also good in distributed implementation
- Immutable data
  - ➔ once produced, cannot change
  - ➔ so no risk of observing it in non-consistent state
  - ➔ no risk of having it corrupted
- Can also be applied to Java
  - ➔ **final** keyword: means field cannot change

Immutability in Java

- (Taken from Oracle Tutorial on Immutability)

```
final public class ImmutableRGB {
    final private int red;
    final private int green;
    final private int blue;

    public ImmutableRGB(int r, int g, int b) {
      this.red = r; this.green = g; this.blue = b;
    }
    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }
    public ImmutableRGB invert() {
      return new ImmutableRGB(255-red,255-green,255-blue);
    }
}
```

Notes on Example

- `final public class ImmutableRGB {`
  - ➔ means cannot be subclassed
  - ➔ so a ref typed w/ `ImmutableRGB` can't point to subclass
  - ➔ subclass dangerous: may change methods, or add fields
- `public ImmutableRGB(int r, int g, int b)`
  - ➔ constructor: only place where state is set
  - ➔ remaining methods: only getters, no setters
- `final private int red;`
  - ➔ int basic type: direct value, not reference
  - ➔ with objects: must make sure they too are immutable

Does it solve all problems?

- No
  - ➔ only `ImmutableRGB` is thread safe
  - ➔ but can be used in ways that are not

```
ImmutableRGB a,b;
a = new ImmutableRGB(0,0,0);
b = new ImmutableRGB(255,0,0);

class MyThread extends Thread {
  void run() {
    ImmutableRGB c;
    c = a;
    a = b;
    b = c;
  } }
```

More on Immutable Collections

- A collection of immutable objects → might be mutable
  - → E.g. **new ArrayList[ImmutableRGB]()**

- But possible to get immutable collections
  - → Using **java.util.Collections.unmodifiableXXX()**
  - → See Java documentation

More on the sematic of final

- **final** fields are *guaranteed* to be initialised
  - ➔ before any other thread can see them
  - ➔ *if* no ref to the containing object leaves the const. midway

- This is not true of non-final fields
  - ➔ and the reason why you might see final fields in MT code

Example

- (Taken from Oracle's documentation Chapter 17. Threads and Locks)

```
class FinalFieldExample {
    final int x;
    int y;
    static FinalFieldExample f;

    public FinalFieldExample() {
        x = 3;
        y = 4;
    }
    static void writer() {
        f = new FinalFieldExample();
    }
    static void reader() {
        if (f != null) {
            int i = f.x;  // guaranteed to see 3
            int j = f.y;  // could see 0
} } }
```

Actors

- Inspired from CSP notion of Processes
  - ➔ Communicating sequential processes (1978)
  - ➔ proposed by Hoare (the same as Hoare's semantic)
  - ➔ CSP: no shared data / message passing
- Actors (Scala, Erlang)
  - ➔ system = a finite set of actors
  - ➔ actors interact through messages only (no shared mem)
  - ➔ messages are asynchronous (queued)
  - ➔ messages treated by message handlers
  - ➔ one handler at a time at most active in each actor

Actors in Erlang

- Called “Processes” (like in CSP)
- Directly built into the language
- Used for concurrency and distribution

Example

```
pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.
```

- Entry method for pong processes
  - ➔ prepare itself to receive either “finished” or “ping”
  - ➔ if ping, sends a pong message to Ping_PID, repeat

Source: http://erlang.org/doc/getting_started/conc_prog.html

Example (cont.)

```
ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).
```

- methods for ping process
  - ➔ functional: no state, no loop, replaced by recursion
  - ➔ sends a ping message, waits for pong reply

Example (finish)

```
start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

- "main" program: entry point
  - ➔ starts 2 processes ("tut15" = name of package)
  - ➔ pong and ping = entry points of processes
  - ➔ [..] = parameters passed to entry points
- Similar to threads! but
  - ➔ no shared data
  - ➔ only interaction: asynchronous messages

```
$ erl
Erlang/OTP 18 [erts-7.0] [source] [64-bit]
[smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Eshell V7.0  (abort with ^G)
1> c(tut15).
{ok,tut15}
2> tut15: start().
Pong received ping
<0.41.0>
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished
3>
```

Summary

- Alternative approaches to parallelism
  - ➔ immutability
  - ➔ actors
- Immutability
  - ➔ use immutable data structures
  - ➔ main philosophy of functional languages
  - ➔ wide ranging influence: HFS, GFS (Hadoop, Google)
- Actors
  - ➔ no shared data
  - ➔ only interaction: message passing
- Strong link to functional languages!

SPP (Synchro et Prog Parallèle)

Unit 9: Atomicity

François Taïani

Motivation

- We have use the word “atomic” quite a few times
  - ➔ “this operation is (is not) atomic”
  - ➔ “this sequence of operation needs to be atomic”

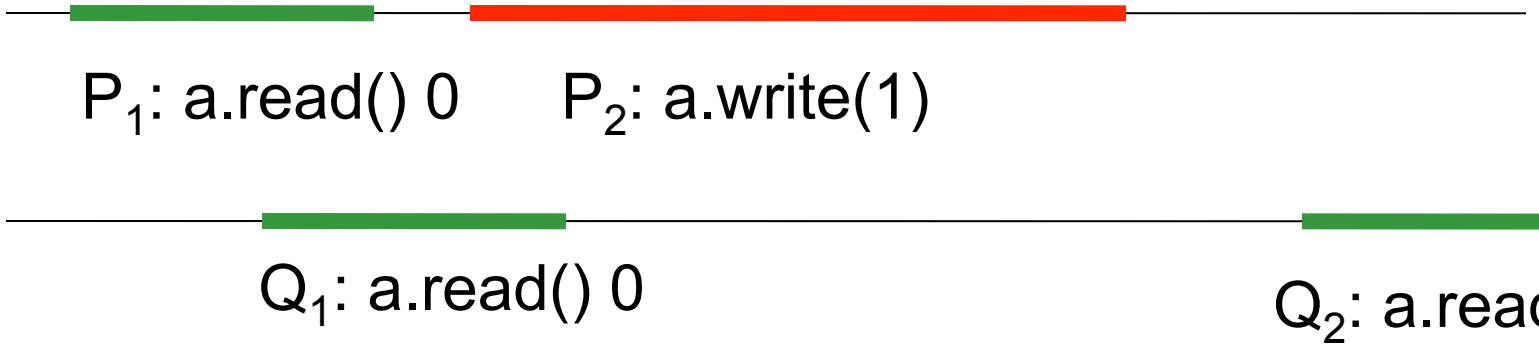
- What does this mean exactly?
  - ➔ Etymology: Ancient Greek ἄ - τομος (cannot be cut)
  - ➔ Can this be captured formally?

Case 1: Read / Write Objects

- System model
  - ➔ read / write objects: 2 operations read() and write(..)
  - ➔ operations invoked by processes, run concurrently
  - ➔ an execution captured by an history
- History
  - ➔ a **sequence** of events
  - ➔ two types of events: invocations, and return events
- invocations labelled with
  - ➔ invoker, invoked object, parameter passed
- return events pair-wise associated with an invocation
  - ➔ at most one return event per invocation

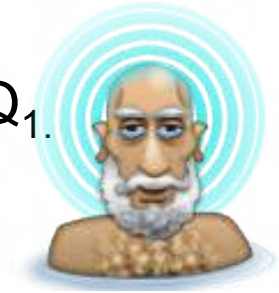
Example

- One shared object, two processes



  - ➔ history made of 8 events, 4 operation executions
  - ➔ History: P_1^{start}, Q_1^{start}, P_1^{end}, P_2^{start},Q_1^{end},P_2^{end},Q_2^{start},Q_2^{end}

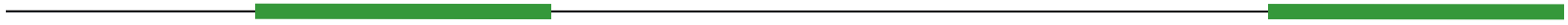
- Order exists between operations
  - ➔ P_2 happens after P_1. Q_2 happens after P_1, P_2, Q_1.
  - ➔ what about Q_1?
  - ➔ How would you formally define this order?



Precedence Order

- A history H induce a partial order $<_H$ on its operations
  - ➔ $op_1 <_H op_2$ iff op_1^{end} appears before op_2^{start} in H
  - ➔ Quiz: Draw the graph of $<_H$ for the previous example

P_1: a.read() 0 P_2: a.write(1)



Q_1: a.read() 0 Q_2: a.read() 1

Sequential History

- A history H is sequential iff
  - ➔ every op_x^{start} is immediately followed a mathing op_x^{end}
  - ➔ "every invocation is followed by its result"
- Quiz
  - ➔ Is the following history sequential? Why?

P_1: a.read() 0 P_2: a.write(1)

Q_1: a.read() 0

Q_2: a.read() 1

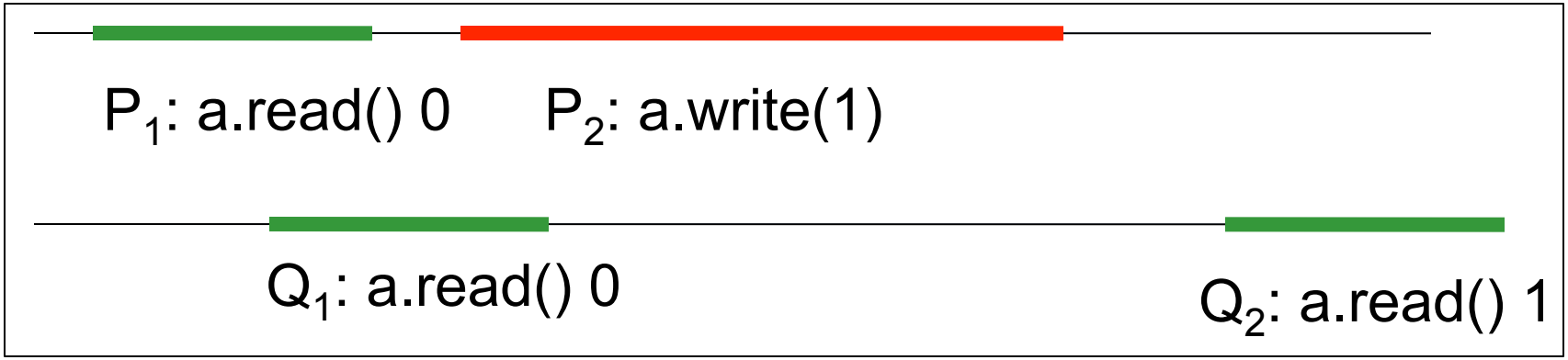
  - ➔ If H is sequential, how is $<_H$?

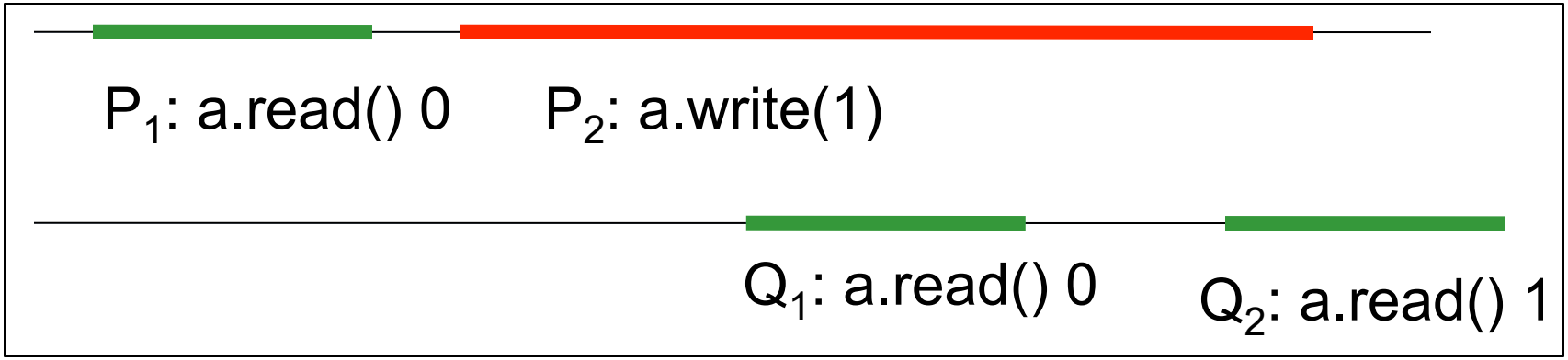
Sub-Histories and Equivalence

- A Sub-History captures the local view of an execution
- Process sub-history H | P
  - ➔ only keep events local to P
- Object sub-history H | a
  - ➔ only keep events happening on a
- Quiz: Write the sub-histories H | P and H | Q for
  - ➔ H = P_1^{start}, Q_1^{start}, P_1^{end}, P_2^{start},Q_1^{end},P_2^{end},Q_2^{start},Q_2^{end}
- 2 histories H_1 and H_2 are equivalent iff
  - ➔ for all processes P: H_1 | P = H_2 | P

Equivalence: Quiz

- Are the 2 following histories equivalent?



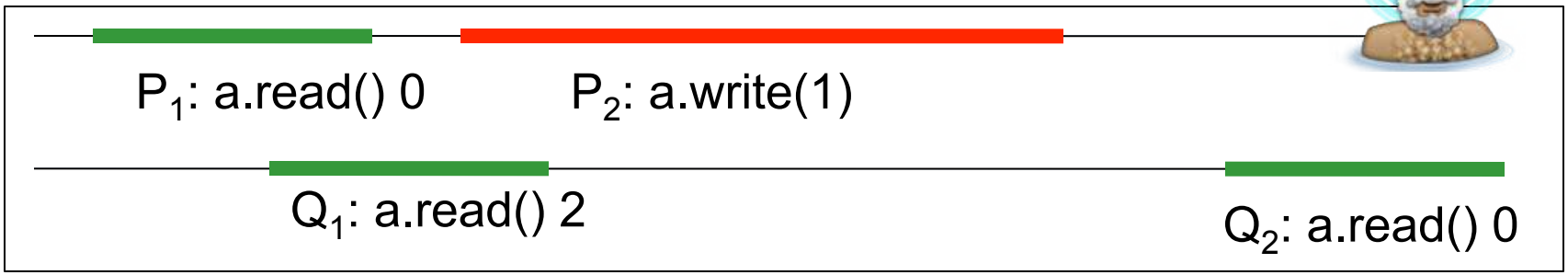


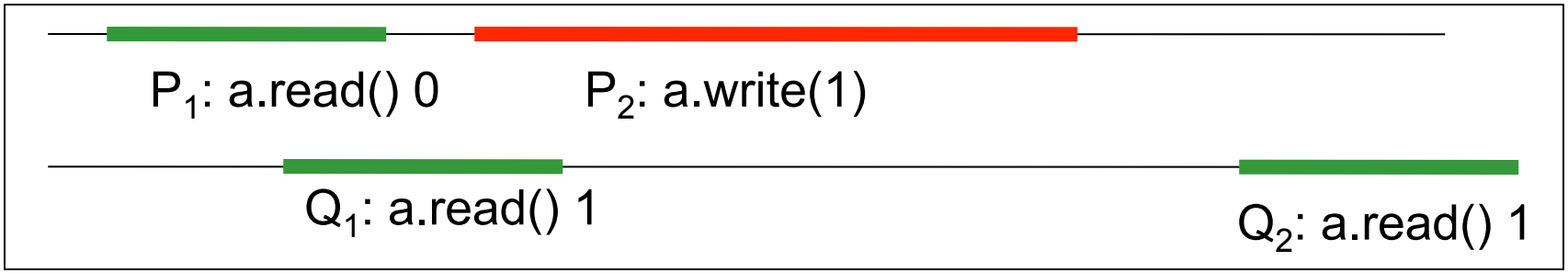
Acceptable Histories

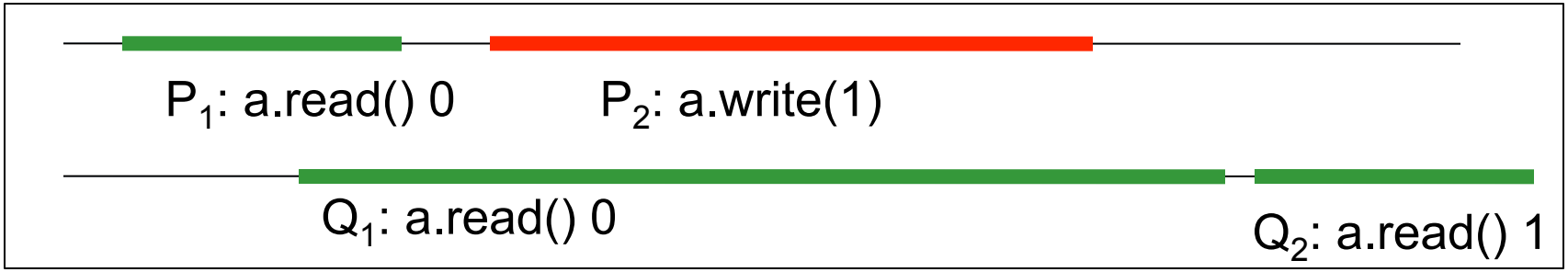
- What we have seen so far
  - ➔ histories
  - ➔ the order they imply on the ops they contain
  - ➔ special case: sequential histories
  - ➔ comparing histories: equivalence (use sub-histories)
- Our aim: capturing atomicity
  - ➔ i.e. defining how an “atomic” object should behave
  - ➔ i.e. defining which behaviour is acceptable, which is not
  - ➔ i.e. defining which histories are acceptable

Acceptable Histories: Intuition

- Are the following histories acceptable?







How to capture acceptability?

- Which histories are acceptable? Which are not?

- First step: focus on **acceptable sequential histories**
  - ➔ no concurrency
  - ➔ aim: capture sequential specification of the object
  - ➔ for R/W O: “reads return value of most recent earlier write”
  - ➔ define set of all “acceptable/legal” sequential histories

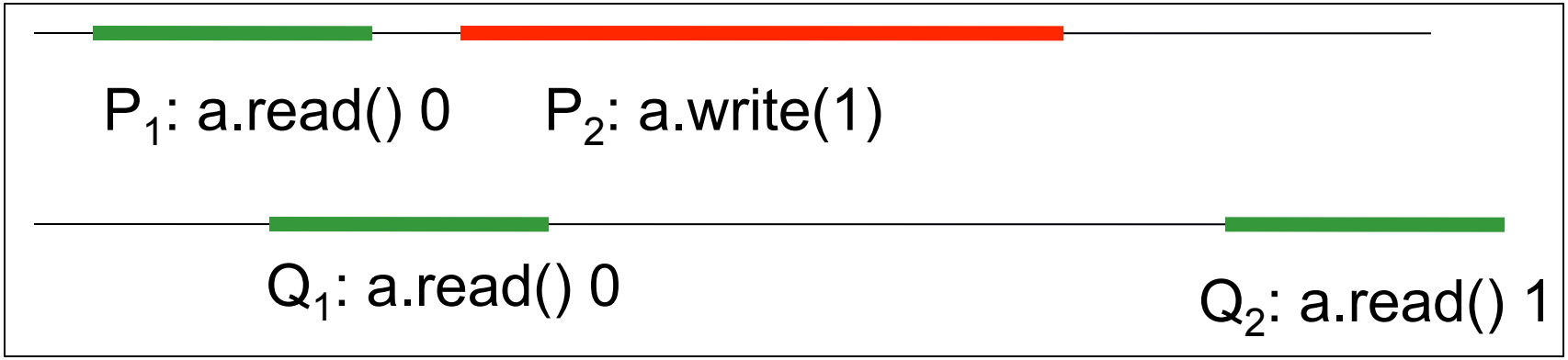
- Second step: define **acceptable concurrent histories**
  - ➔ using acceptable sequential histories as reference
  - ➔ different ways to do this: different semantics of consistency!

Atomicity

- History H is atomic iff
  - ➔ H can be extended into H' by adding (optional) return events
  - ➔ ∃ acceptable sequential history S, so that S equivalent H'
  - ➔ $<_H \subseteq <_S$
- Comments
  - ➔ H' needed to handle pending operations
  - ➔ S eq. H' : process cannot distinguish between S and H'
  - ➔ $<_H \subseteq <_S$: S cannot reorder events from H
- **Atomic Object: only accepts atomic histories**
- Also called "Linearizability" (Herlihy & Wing, 1990)

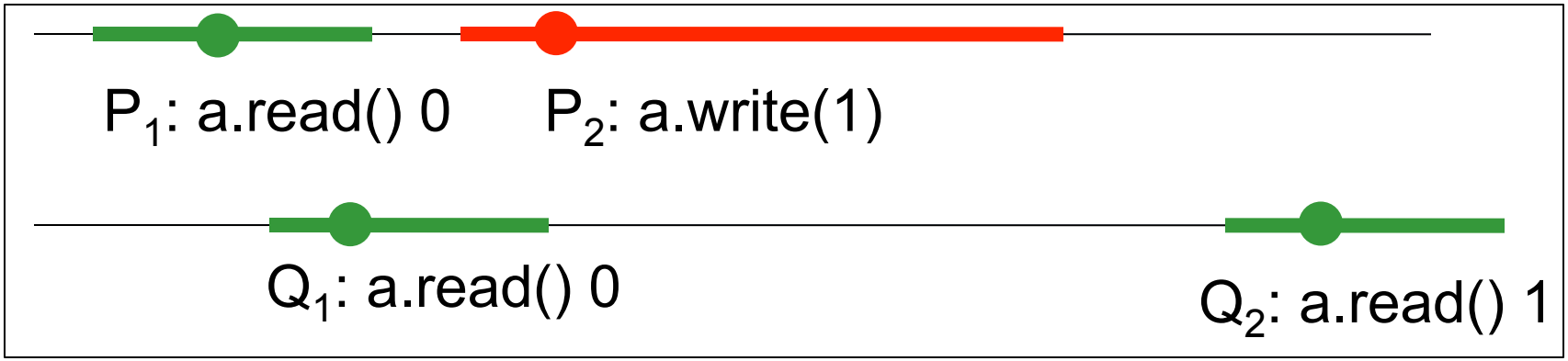
Example

- Find a history S that shows that the following is atomic
  - ➔ reminder 1: S should be sequential and acceptable
  - ➔ reminder 2: S should be equivalent to the history below



Example

- Interpretation
  - ➔ possible to find a point in each interval
  - ➔ so that resulting sequential history is acceptable



  - ➔ “Points of Linearizibility”

Case 2: Generalization

- Previous definitions generalizable beyond R/W object
  - ➔ just need to redefine acceptable sequential histories
- Example: queue: sequential specification
  - ➔ #(dequeue) ≤ #(enqueue)
  - ➔ op $dequeue_k$ return value passed by $enqueue_k$
- All other definitions follow

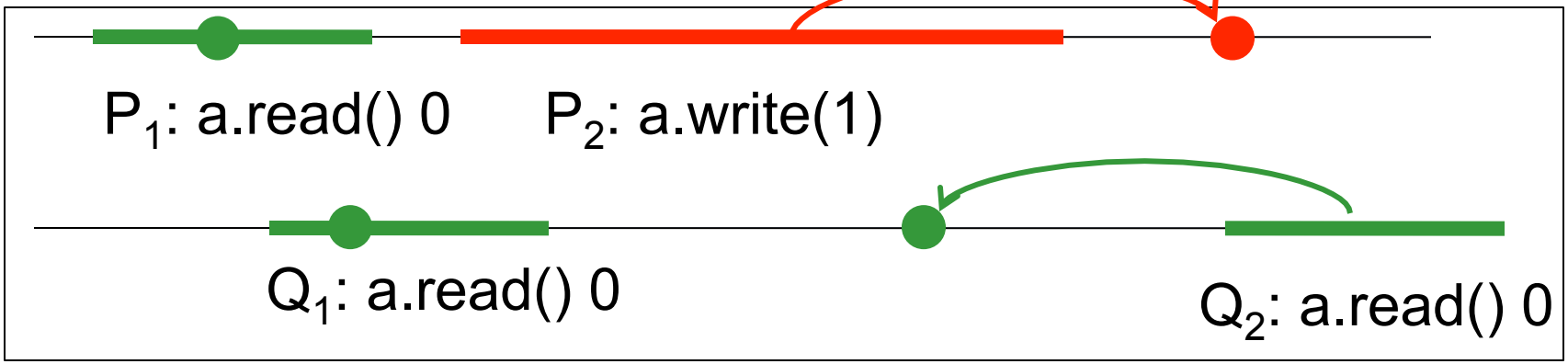
Key Properties of Atomicity

- Locality: For a system made of multiple objects x_i
  - ➔ H is atomic iff H | x_i is atomic for all x_i
  - ➔ i.e. composing atomic objects results in an atomic system

- Non-blocking
  - ➔ pending operations can always complete (*)
  - ➔ * = provided they're defined (cf. dequeue an empty queue)

Alternative Consistency Models

- Sequential Consistency (Lamport, 1979)
  - ➔ H sequentially consistent iff
    ∃ acceptable sequential history S, S equivalent to H
- Difference 1: S can reorder operations!
  - ➔ Following history is sequentially consistent, but not atomic



- Difference 2: Not a local property!

Alternative Consistency Model

- (Strict) Serializability (Transactions, Databases)
  - ➔ very close to atomicity as defined here
  - ➔ except order $<_H$ defined on transactions
  - ➔ transactions: multiple operations

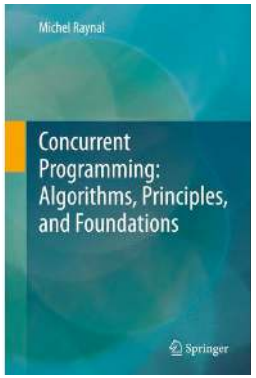
- Important Consequences / Differences
  - ➔ strict serializability not a local property
  - ➔ it is a blocking property:
    sometimes transactions must abort

Summary

- This session
  - ➔ formal approach to specifying legal parallel behaviours
- Key notions:
  - ➔ processes, shared objects
  - ➔ history, sub-histories
  - ➔ sequential histories, legal histories, equivalent histories
- Atomicity builds on all these notions
  - ➔ define legal // behaviour based on sequential specification
  - ➔ find an equivalent seq. history that meet specific criteria
- Important to reason about parallel programs
  - ➔ specifications, proofs (manual, automatic), composition

References

- Maurice P. Herlihy and Jeannette M. Wing. 1990. *Linearizability: a correctness condition for concurrent objects*. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463-492. DOI=10.1145/78969.78972 http://doi.acm.org/10.1145/78969.78972
- Chapter 4 “Atomicity: Formal Definition and Properties” in Michel Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*, Springer, Jan 2013 , ISBN-13: 978-3642320262



SPP (Synchro et Prog Parallèle)

Unit 10: Transactions and Wait-Free Programming

François Taïani

Transactions

- Main Synchronization Mechanisms used in DB
- Principle: 2 new operations
  - ➔ dbStore.startTransaction()
  - ➔ dbStore.endTransaction() (aka commit)
- Often completed with 3rd operation
  - ➔ dbStore.abortTransaction()
- Use (dbStore dropped for brevity)
  - ➔ startTransaction

    some sequence of actions (reads, writes, computation)

    endTransaction

Semantic

- Transactions
  - ➔ either execute in totality or not at all (“atomicness”)
  - ➔ concurrent transactions don’t interfere (“isolation”)
- Notes on DB
  - ➔ Other usual properties: Consistent (app) / Durable (FT)
- Notes on atomicity
  - ➔ “Atomicness” ≠ from “atomic” concurrent objects
  - ➔ “Atomicness” -> transactions must be able to roll back
- Notes on Isolation
  - ➔ Different levels / semantics of “isolation”
  - ➔ “Isolation” closely related to linearizability

Atomicness of Transactions

- Vocabulary: “Atomic” is overloaded
  - ➔ usual name in DB = atomicity, but confusing here
  - ➔ “atomicness” avoids confusion w/ atomic objects
- Key principles to provide atomicness
  - ➔ must be able to cancel transactions halfway through
  - ➔ if cancelled, any effect of the transaction should disappear
  - ➔ “as if” never happened -> might cause cascading roll backs
- Two main strategies
  - ➔ work on shadow copies of variables, only write if success
  - ➔ log all write operations, undo if abort
  - ➔ (not that easy, but we’ll ignore details here)

Isolation of Transactions

- Key problem: schedule operations of transactions
- Naïve approach: serialise all transactions
  - ➔ transactions execute one after the other
  - ➔ easy, but poor concurrency, poor resource utilisation
- General approach:
  - ➔ interleave actions of different transactions
  - ➔ use scheduling techniques to insure “correct” outcome
  - ➔ (might include aborting transaction that don’t fit)

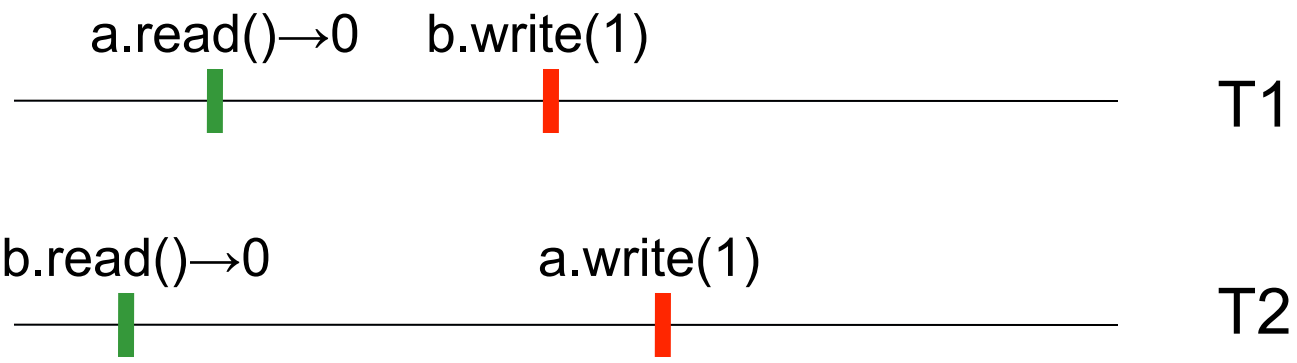
Isolation of Transactions

What is a correct outcome? Three main semantics

- Serializability: A schedule S is serializable iff
  - ➔ ∃ sequential schedule, same result of chosen schedule
  - ➔ (Note: similar to sequential consistency for conc. objects)
- Strict Serializability: S is strictly serializable iff
  - ➔ ∃ sequential schedule S', same result as S
  - ➔ S' respects precedence order of S ($<_S \subseteq <_{S'}$)
  - ➔ (Note: similar to atomicity for concurrent objects)
- Snapshot isolation: S respects snapshot isolation iff
  - ➔ each transaction = work on personal snapshot of DB
  - ➔ successful provided updated value not in conflict

Example

- Consider this example
  - ➔ Transaction 1: x := a.read() ; b.write(x) ;
  - ➔ Transaction 2: y := b.read() ; a.write(y) ;
  - ➔ x and y are local variables
  - ➔ the system is initialised as a = 0 ; b = 0
- What semantic(s) does the following schedule follow?



Comment on Example

- A case of write-skew anomaly
  - ➔ write-sets of T1 & T2 disjoint: no conflicting updates
  - ➔ so no risk of one overwriting the results of the other
  - ➔ but read-set of T2 (b) updated by T1 before end of T2
  - ➔ impossible under (strict) serializability semantic
- In General
  - ➔ snap. isolation $<_{weaker}$ seria^{lizability} $<_{weaker}$ strict serial^{izability}
  - ➔ $<_{weaker}$ means “allows more schedules than”
  - ➔ consequence: provide less guarantees

Implementing Isolation

- 2PL: 2 phase locking (provides serializability)
  - ➔ growth phase: take locks on resources
  - ➔ skrinking phase: release locks on resources
  - ➔ detect deadlocks and abort
  - ➔ or avoid deadlocks by order on resources

- Other approaches (not discussed)
  - ➔ Timestamp Ordering (provides serializability)
  - ➔ Multiversion Concurrency Control (for snapshot isolation)

Wait-Free Programming

- Principle: avoid locks!
  - ➔ dangerous: deadlocks, starvation
  - ➔ inefficient: diminish concurrency
- Ideal: Wait-Free concurrent object
  - ➔ all operation can always progress (only limited by CPU)
- Quiz: Does wait-freedom makes sense for a queue?

Wait-Free Programming

- Principle: avoid locks!
  - ➔ dangerous: deadlocks, starvation
  - ➔ inefficient: diminish concurrency
- Ideal: Wait-Free concurrent object
  - ➔ all operation can always progress (only limited by CPU)
- Quiz: Does wait-freedom make sense for a queue?
  - ➔ Yes, but need to allow “undefined” return value (⊥)
- Difficulty:
  - ➔ very complex to design and prove
  - ➔ but some practical algorithms do exists (for queues in particular)

Summary

- Broad brush presentation of transactions
  - ➔ Very rich fields, of crucial importance
  - ➔ Both for DB, but also transactional engine (J2EE…)
  - ➔ A large range of semantics for isolation
  - ➔ A large range of scheduling algorithms
- Wait free concurrent objects
  - ➔ still an area of research for practicable implementations
  - ➔ important to be aware of its existence: performance ↗
  - ➔ variant properties: non-blocking, lock-free
- We have only touched the surface on both subjects