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
  - ➔ contrary are Uni-Processor machines or UP
  - ➔ 'SMP' term found in the Linux kernel for instance

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) or setting (flag=1) the flag is 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
  - ➔ 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 come 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
  x = 1
  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 avoid context switching on local core
  - ➔ 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
```

- But this code is problematic
  - ➔ context switch possible just after test_and_set(..)
  - ➔ if control passes to thread t2 that tries to take same lock then t2 spins until next context switch! (CPU waste)

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

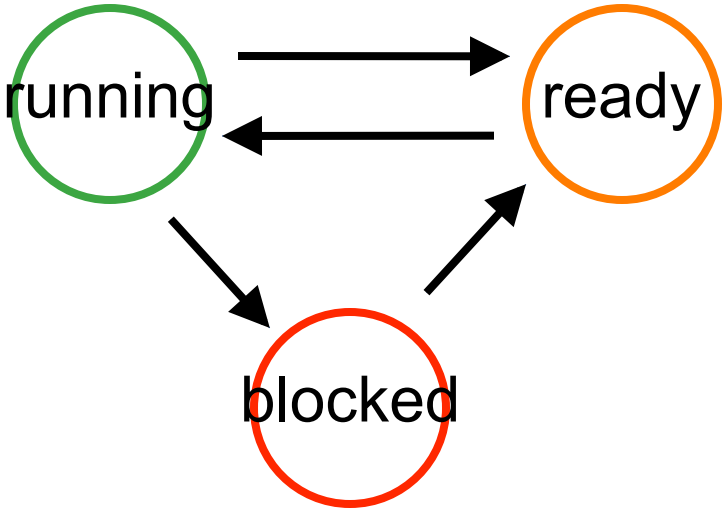
We need more than spin!

- 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!

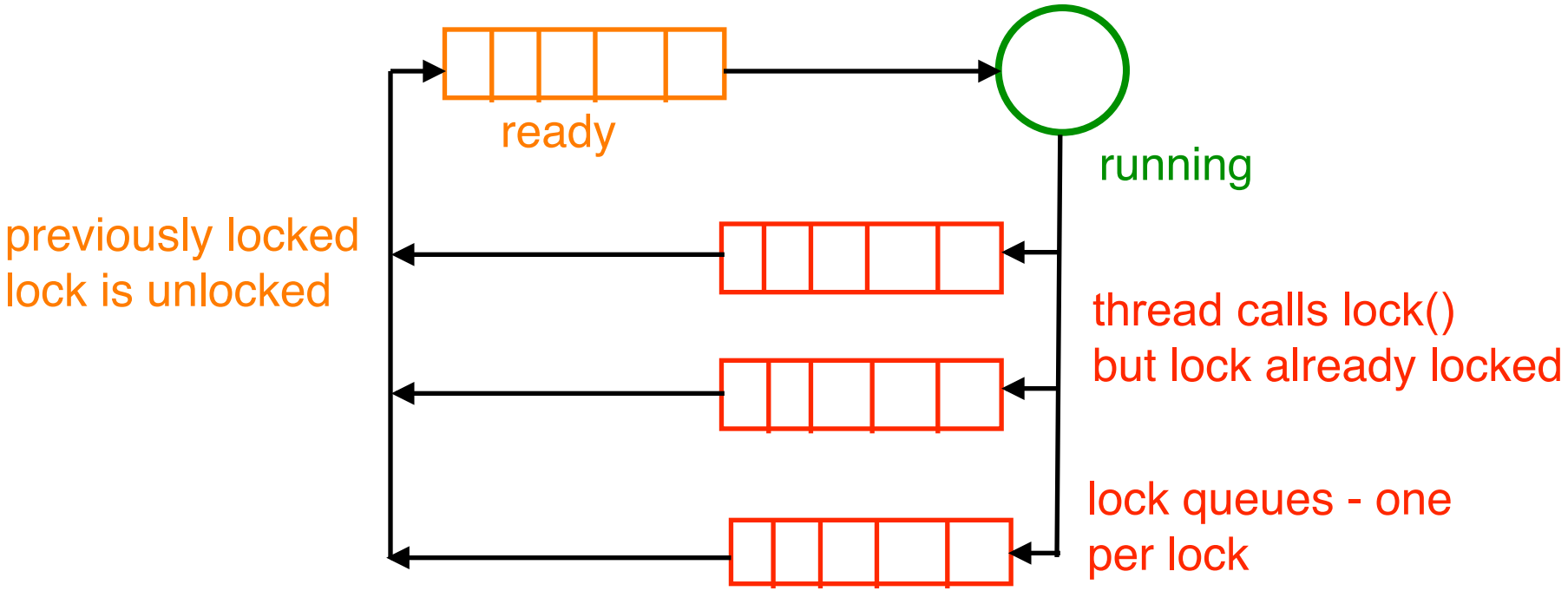
Reminder: OS Scheduler

- 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?

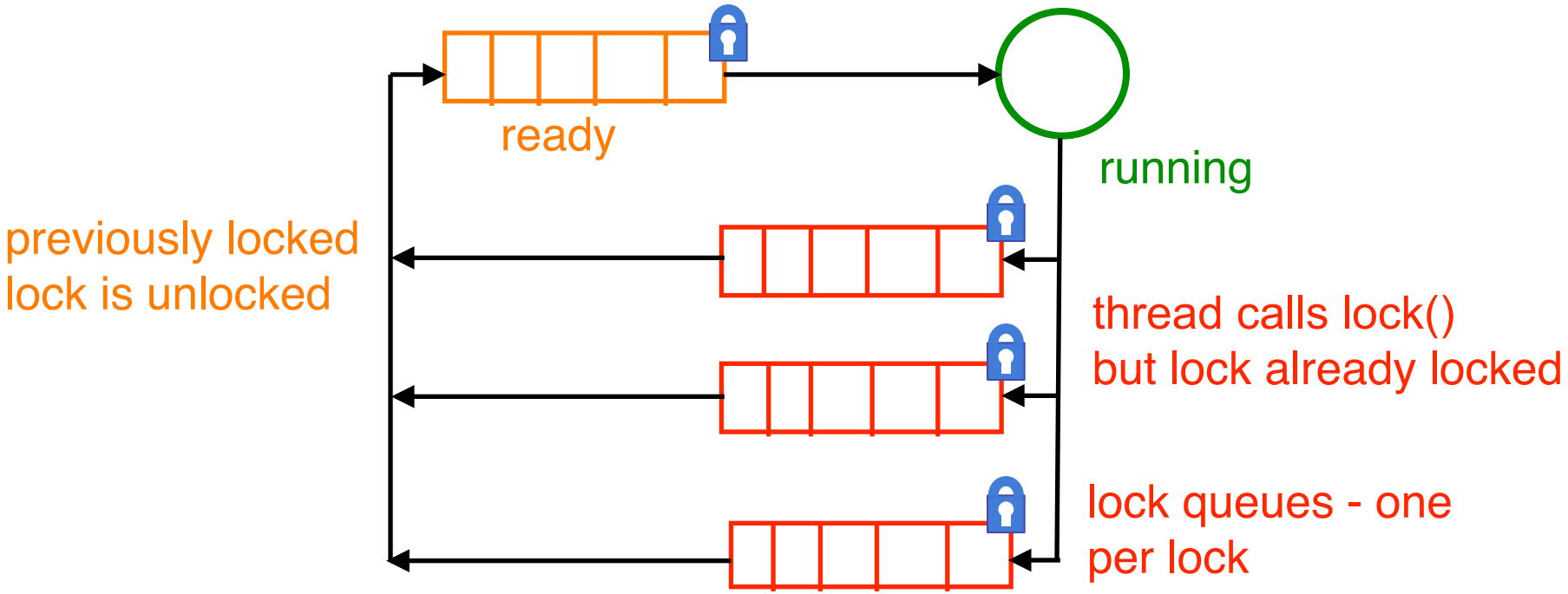
Implementing Blocking Locks

- 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
  - ➔ http://www.kernel.org/doc/Documentation/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"