Concurrent systems

Lecture 1: Introduction to concurrency and threads

Michaelmas 2013
Dr Robert N. M. Watson
(With thanks to Dr Steven Hand)

Concurrent and distributed systems

- One course, two parts
  - 8 lectures on concurrent systems (Michaelmas term)
  - 8 further lectures of distributed systems (Lent term)
- Similar concerns:
  - Scalability given parallelism and distributed systems
  - Mask local or distributed latency
  - Correctness in the presence of concurrency (+debugging)
- Some important differences
  - Abstract: distributed systems experience communications failure
  - Concrete: quite different programmer abstractions

Concurrent systems course outline

1. Introduction to concurrency and threads
2. Mutual exclusion and process synchronisation
3. CCR, monitors, concurrency in practice
4. Safety and liveness
5. Concurrency without shared data; transactions
6. Further transactions
7. Crash recovery; lock free programming; TM
8. Concurrent systems case study

3

Recommended Reading

- "*Operating Systems, Concurrent and Distributed Software Design*", Jean Bacon and Tim Harris, Addison-Wesley 2003
- "*Modern Operating Systems*", (3rd Ed), Andrew Tannenbaum, Prentice-Hall 2007
- "*Java Concurrency in Practice*", Brian Goetz and others, Addison-Wesley 2006

4

What is Concurrency?

- Computers appear to do many things at once
  - e.g. running multiple programs on your laptop
  - e.g. writing back data buffered in memory to the hard disk while the program(s) continue to execute
- In the first case, this may actually be an **illusion**
  - e.g. processes time-sharing a single CPU
- In the second, there is **true parallelism**
  - e.g. DMA engine transfers data from memory and writes to disk at the same time as the CPU executes code
- In both cases, we have a **concurrency**
  - many things are occurring "at the same time"

5

In this course we will

- Investigate the ways in which concurrency can occur in a computer system;
  - processes, threads, interrupts, hardware
- Consider how to control concurrency;
  - mutual exclusion (locks, semaphores), condition synchronization, lock-free programming
- Learn about deadlock, livelock, priority inversion
  - And prevention, avoidance, detection, recovery
- See how abstraction can provide support for correct & fault-tolerant concurrent execution
  - transactions, serializability, concurrency control

6

Recap: Processes and Threads

- A process is a program in execution
  - Unit of protection & resource allocation
  - Has an associated virtual address space; and one or more **threads**
- A thread is an entity managed by the scheduler
  - Represents an individual execution context
  - Managed by a thread control block (TCB) which holds the saved context (registers), scheduler info, etc
- Threads run in the address spaces of their containing process
  - (or within the kernel address space)

7

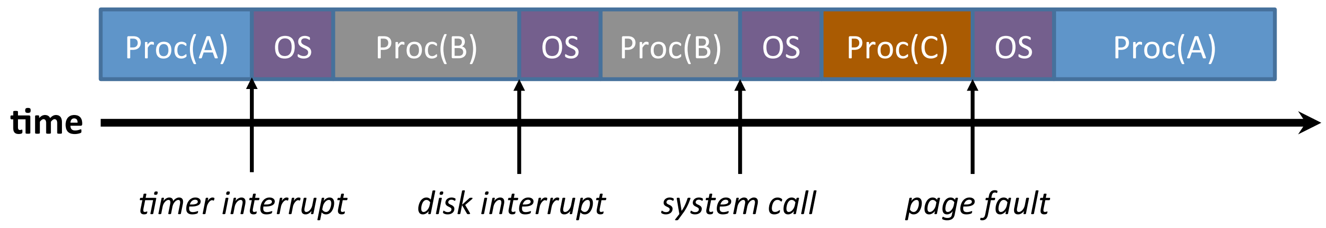
Concurrency with a single CPU

- Process / OS Concurrency
  - Process X runs for a while (until blocks or **interrupted**)
  - OS runs for a while (e.g. does some TCP processing)
  - Process X resumes where it left off…
- **Inter-Process** Concurrency
  - Process X runs for a while; then OS; then Process Y; then OS; then Process Z; etc
- **Intra-Process** Concurrency
  - Process X has multiple threads X1, X2, X3, …
  - X1 runs for a while; then X3; then X1; then X2; then …

8

Concurrency with a single CPU

- With just one CPU, can think of concurrency as **interleaving** of different executions, e.g.



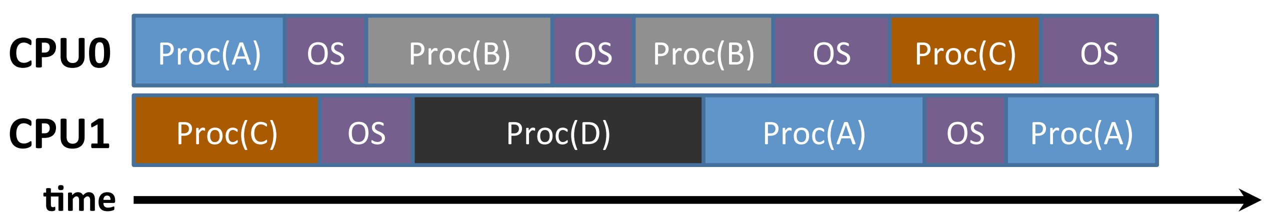
- Exactly where execution is interrupted and resumed is not usually known in advance…
  - this makes concurrency challenging!
- Generally should assume worst case behavior

Non-deterministic or so complex as to be **unpredictable**

9

Concurrency with multiple processors

- Many modern systems have multiple CPUs
  – And even if don't, have other processing elements
- Hence things can occur in parallel, e.g.



- Notice that the OS runs on both CPUs: tricky!
- More generally can have different threads of the same process executing on different CPUs too

10

Threading models

- Threads can be implemented by userspace or kernel
- User-level threads
  - OS schedules a single process (e.g. JVM)
  - User-code (or a user-mode library) implements threading calls, a scheduler, and context switching code
- Advantages include:
  - lightweight creation/termination and context switch; application-specific scheduling; OS independence
- Disadvantages:
  - awkward to implement preemption, or to handle blocking system calls or page faults; and cannot use multiple CPUs
- Examples: Java greenthreads, stackless Python, Haskell

11

Threading models

- Kernel-level threads
  - OS aware of both processes and threads within processes
  - By default, a process has one main thread…
  - … but can create more via system call interface
  - Kernel schedules threads (and performs context switching)
- Advantages:
  - Easy to handle preemption or blocking system calls
  - Relatively straightforward to utilize multiple CPUs
- Disadvantages:
  - Higher overhead (trap to kernel); less flexible; less portable
- Examples: Windows, Linux, Mac OS X, FreeBSD, Solaris, …
  - Most transitioned from user threads over last decade and a half

12

Hybrid threading models

- Ideally would like the best of both worlds
  - i.e. advantages of user- and kernel-level threads
- Various hybrid solutions proposed (first-class threads, scheduler activations, FreeBSD KSE, Solaris LWP)
  - OS and user-space co-operate in scheduling
  - User-space registers an activation handler
  - OS either resumes a context, or "upcalls" the handler
  - The former provides transparent kernel-thread scheduling; the latter, notifications of blocking events
  - On an upcall, handler can switch to another thread
- Mostly experimental or even deprecated (why?) in OSes, widely used in VMMs
  - Reappearing in work distribution frameworks e.g., Grand Central Dispatch (GCD)

13

Advantages of concurrency

- Allows us to overlap computation and I/O on a single machine
- Can simplify code structuring and/or improve responsiveness
  - e.g. one thread redraws the GUI, another handles user input, and another computes game logic
  - e.g. one thread per HTTP request
  - e.g. background GC thread in JVM/CLR
- Enables the seamless (?!) use of multiple CPUs – greater performance through parallel processing

14

Concurrent systems

- In general, have some number of processes…
  - … each with some number of threads …
  - … running on some number of computers…
  - … each with some number of CPUs.
- For this half of the course we'll focus on a single computer running a multi-threaded process
  - most problems & solutions generalize to multiple processes, CPUs, and machines, but more complex
  - (we'll look at distributed systems in Lent term)
- Challenge: threads will access shared resources concurrently via their common address space

15

Example: Housemates Buying Beer

- Thread 1 (person 1)
  1. Look in fridge
  2. If no beer, go buy beer
  3. Put beer in fridge
- Thread 2 (person 2)
  1. Look in fridge
  2. If no beer, go buy beer
  3. Put beer in fridge
- In most cases, this works just fine…
- But if both people look (step 1) before either refills the fridge (step 3)… we'll end up with too much beer!
- Obviously more worrying if "look in fridge" is "check reactor", and "buy beer" is "toggle safety system" ;-)

16

Solution #1: Leave a Note

- Thread 1 (person 1)
  1. Look in fridge
  2. If no beer & no note
     1. Leave note on fridge
     2. Go buy beer
     3. Put beer in fridge
     4. Remove note
- Thread 2 (person 2)
  1. Look in fridge
  2. If no beer & no note
     1. Leave note on fridge
     2. Go buy beer
     3. Put beer in fridge
     4. Remove note
- Probably works for human beings…
  - But computers are stooopid!
- Can you see the problem?

17

Non-Solution #1: Leave a Note

```
// thread 1
beer = checkFridge();
if(!beer) {
  if(!note) {
     note = 1;
     buyBeer();
     note = 0;
  }
}
```

```
// thread 2
beer = checkFridge();
if(!beer) {
  if(!note) {
     note = 1;
     buyBeer();
     note = 0;
  }
}
```

- Easier to see with pseudo-code…

18

Non-Solution #1: Leave a Note

```
// thread 1
beer = checkFridge();
if(!beer) {
  if(!note) {
```

context switch

```
// thread 2
beer = checkFridge();
if(!beer) {
  if(!note) {
     note = 1;
     buyBeer();
     note = 0;
```

context switch

```
     note = 1;
     buyBeer();
     note = 0;
  }
}
```

```
  }
}
```

- Easier to see with pseudo-code…

Non-Solution #1: Leave a Note

- Of course this won't happen all the time
  - Need threads to interleave in the just the right way (or just the wrong way ;-)
- Unfortunately code that is 'mostly correct' is much worse than code that is 'mostly wrong'!
  - Difficult to catch in testing, as occurs rarely
  - May even go away when running under debugger
    - e.g. only context switches threads when they block
    - (such bugs are sometimes called "Heisenbugs")

Critical Sections & Mutual Exclusion

- The high-level problem here is that we have two threads trying to solve the same problem
  - Both execute buyBeer() concurrently
  - Ideally want only one thread doing that at a time
- We call this code a **critical section**
  - a piece of code which should never be concurrently executed by more than one thread
- Ensuring this involves **mutual exclusion**
  - If one thread is executing within a critical section, all other threads are prohibited from entering it

21

Achieving Mutual Exclusion

- One way is to let only one thread ever execute a particular critical section – e.g. a nominated beer buyer – but this restricts concurrency
- Alternatively our (broken) solution #1 was ***trying*** to provide mutual exclusion via the note
  - Leaving a note means "I'm in the critical section";
  - Removing the note means "I'm done"
  - But, as we saw, it didn't work ;-)
- This was since we could experience a context switch between reading 'note', and setting it

22

Non-Solution #1: Leave a Note

```
// thread 1
beer = checkFridge();
if(!beer) {
  if(!note) {
```

We decide to enter the critical section here…

context switch

```
// thread 2
beer = checkFridge();
if(!beer) {
  if(!note) {
     note = 1;
     buyBeer();
     note = 0;
```

context switch

But only mark the fact here …

```
      note = 1;
      buyBeer();
      note = 0;
  }
}
```

These problems are referred to as **race conditions** in which multiple **threads race** with one another during conflicting access to shared resources

23

Atomicity

- What we want is for the checking of note and the (conditional) setting of note to happen without any other thread being involved
  - We don't care if another thread reads it after we're done; or sets it before we start our check
  - But once we start our check, we want to continue without any interruption
- If a sequence of operations (e.g. read-and-set) occur as if one operation, we call them **atomic**
  - Since indivisible from the point of view of the program
- An atomic "read-and-set" operation is sufficient for us to implement a correct beer program

24

Solution #2: Atomic Note

```
// thread 1
beer = checkFridge();
if(!beer) {
  if(read-and-set(note)) {
     buyBeer();
     note = 0;
  }
}
```

```
// thread 2
beer = checkFridge();
if(!beer) {
  if(read-and-set(note)) {
     buyBeer();
     note = 0;
  }
}
```

- read-and-set(&address) **atomically** checks the value in memory and iff it is zero, sets it to one
  - returns 1 iff the value was changed from 0 -> 1
- This prevents the behavior we saw before, and is sufficient to implement a correct program…
  - although this is not that program :-)

25

Non-Solution #2: Atomic Note

```
// thread 1
beer = checkFridge();
if(!beer) {
```

context switch

```
// thread 2
beer = checkFridge();
if(!beer) {
  if(read-and-set(note)) {
     buyBeer();
     note = 0;
```

context switch

```
  if(read-and-set(note)) {
     buyBeer();
     note = 0;
  }
}
```

```
  }
}
```

- Our critical section doesn't cover enough!

26

Foreshadowing of next lecture

General mutual exclusion

- We would like the ability to define a region of code as a critical section e.g.

```
// thread 1
ENTER_CS();
beer = checkFridge();
if(!beer)
     buyBeer();
LEAVE_CS();
```

```
// thread 2
ENTER_CS();
beer = checkFridge();
if(!beer)
     buyBeer();
LEAVE_CS();
```

- This should work …
  - … providing that our implementation of ENTER_CS() / LEAVE_CS() is correct

27

Summary + next time

- Definition of a concurrent system
- Origins of concurrency within a computer
- Processes and threads
- Challenge: concurrent access to shared resources
- Mutual exclusion, race conditions, and atomicity

- Next time:
  - More on mutual exclusion
  - Hardware support for mutual exclusion
  - Semaphores for mutual exclusion, process synchronisation, and resource allocation
  - Producer-consumer relationships.

28