Concurrent systems

Lecture 1: Introduction to concurrency, threads, and mutual exclusion

Michaelmas 2016
Dr Robert N. M. Watson
(With thanks to Dr Steven Hand)

1

Concurrent and distributed systems

- One course, two parts
  - 8 lectures on concurrent systems (Michaelmas term)
  - 8 further lectures of distributed systems (Lent term)
- Similar interests and concerns:
  - **Scalability** given parallelism and distributed systems
  - Mask local or distributed **communications latency**
  - Importance in observing (or enforcing) **execution orders**
  - **Correctness** in the presence of concurrency (+debugging)
- Important differences
  - Underlying primitives: **shared memory** vs. **message passing**
  - Distributed systems experience **communications failure**
  - Distributed systems (may) experience **unbounded latency**
  - (Further) difficulty of **distributed time**

2

Concurrent systems course outline

1. Introduction to concurrency, threads, and mutual exclusion
2. More mutual exclusion, semaphores, producer-consumer, and MRSW
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. Direct Memory Access (DMA) transfers data between memory and I/O devices (e.g., NIC, SATA) at the same time as the CPU executes code
  - e.g., two CPUs execute code at the same time
- 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, serialisability, concurrency control
- Explore a detailed concurrent software case study
- Next term, extend these ideas to distributed systems

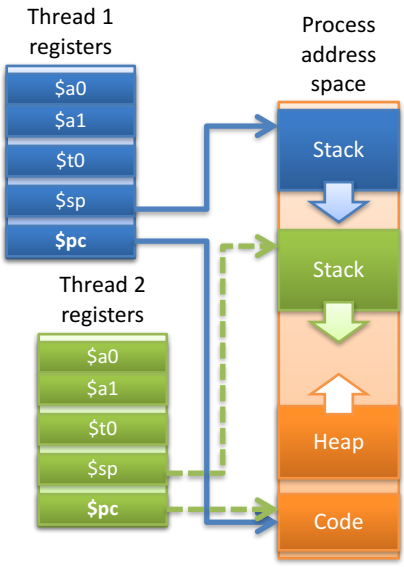
6

Recall: Processes and threads

- **Processes** are instances of programs in execution
  - OS unit of protection & resource allocation
  - Has a virtual **address space**; and one or more threads
- **Threads** are entities managed by the **scheduler**
  - Represents an individual execution context
  - A thread control block (TCB) holds the saved context (registers, including stack pointer), scheduler info, etc
- Threads run in the **address spaces** of their process
  - (and sometimes in the kernel address space)
- **Context switches** occur when the OS saves the state of one thread and restores the state of another
  - If a switch is between threads in different processes, then process state is also switched – e.g., the address space

7

Multiple threads within a process



- A single-threaded process has **code**, a **heap**, a **stack**, **registers**
- **Additional threads** have their own registers and stacks
  - Per-thread **program counters** (**$pc**) allow execution flows to differ
  - Per-thread **stack pointers** (**$sp**) allow local variables to differ
- Heap and code (+**globals**) are shared between all threads
- Access to another thread's stack is sometimes possible – but deeply discouraged!

8

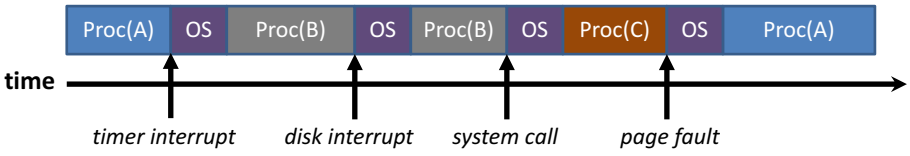
Concurrency with a single CPU (1)

- **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 …

9

Concurrency with a single CPU (2)

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



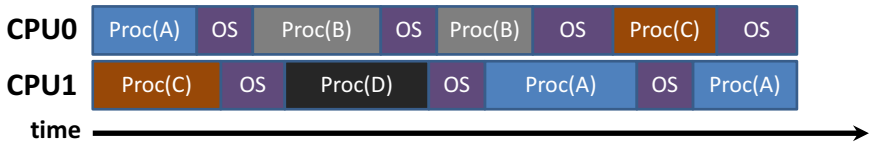
- Exactly where execution is interrupted and resumed is not usually known in advance…
  - this makes concurrency challenging!
- Generally should assume worst case behavior

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

10

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

11

What does this code do?

```
#define NUMTHREADS 4
char *threadstr = "Thread";

void threadfn(int threadnum) {
        sleep(rand(2));
        printf("%s %d\n", threadstr, threadnum);
}

void main(void) {
        threadid_t threads[NUMTHREADS];
        int i;

        for (i = 0; i < NUMTHREADS; i++)
                threads[i] = thread_create(threadfn, i);

        for (i = 0; i < NUMTHREADS; i++)
                thread_join(threads[i]);
}
```

Global variables are shared by all threads

Each thread has its own local variables

`main()` is called once at startup

Additional threads are started explicitly

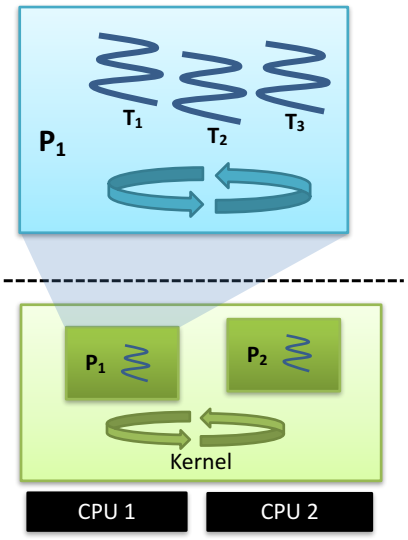
What **orders** could the `printfs` run in?

Possible orderings of this program

- What order could the **`printf()`**s occur in?
- Two sources of non-determinism in example:
  - **Program non-determinism**: Program randomly sleeps 0 or 1 seconds before printing
  - **Thread scheduling non-determinism**: Arbitrary order for unprioritised, concurrent wakeups, preemptions
- There are 4! (factorial) valid permutations
  - Assuming printf() is **indivisible**
  - Is printf() indivisible? Maybe.
- Even more potential **timings** of **`printf()`**s

13

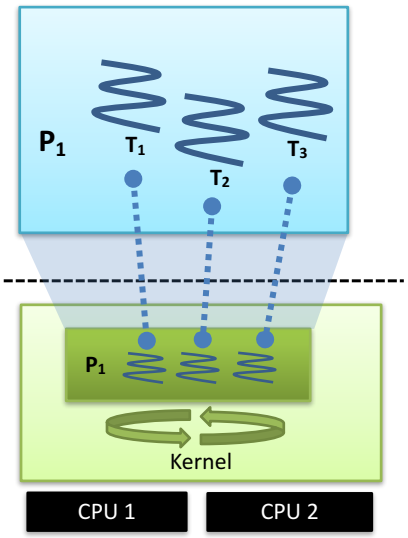
1:N - user-level threading



- **Kernel** only knows about (and schedules) processes
- **Userspace** implements threads, context switching, scheduling
  - E.g., in the JVM or a threading library
- Advantages
  - Lightweight creation/termination and context switch; application-specific scheduling; OS independence
- Disadvantages
  - Awkward to handle blocking system calls or page faults, preemption; cannot use multiple CPUs
- Very early 1990s!

14

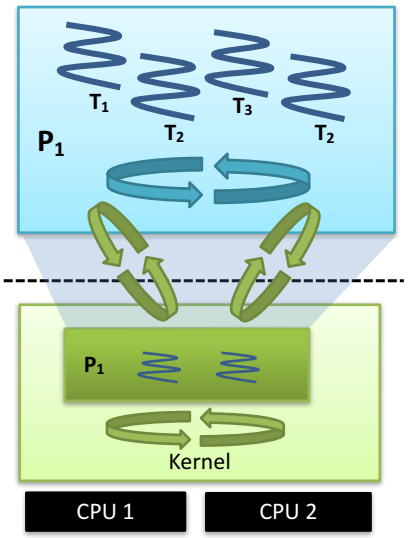
1:1 - kernel-level threading



- **Kernel** provides threads directly
  - By default, a process has one thread…
  - … but can create more via system calls
- Kernel implements threads, thread context switching, scheduling, etc.
- **Userspace** thread library **1:1** maps **user threads** into **kernel threads**
- Advantages:
  - Handles preemption, blocking system calls
  - Straightforward to utilize multiple CPUs
- Disadvantages:
  - Higher overhead (trap to kernel); less flexible; less portable
- Threading model of choice across all major OSes today
  - Windows, Linux, Mac OS X, FreeBSD, Solaris, …

15

M:N - hybrid threading



- Best of both worlds?
  - **M:N threads**, **scheduler activations**, …
- **Kernel** exposes a smaller number (**M**) of **activations** – typically 1:1 with CPUs
- **Userspace** schedules a larger number (**N**) of **threads** onto available activations
  - Kernel **upcalls** when a **thread blocks**, returning the activation to userspace
  - Kernel **upcalls** when a **blocked thread wakes up**, userspace schedules it on an activation
  - Kernel controls **maximum parallelism** by limiting number of activations
- Feature removed from most OSes – why?
- Used in **Virtual Machine Monitors** (VMMs)
  - Each **Virtual CPU (VCPU)** is an activation
- M:N reappears in concurrency frameworks
  - E.g., Apple's Grand Central Dispatch (GCD)

16

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

17

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

18

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" ;-)

19

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?

20

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…

21

Non-Solution #1: Leave a Note

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

context switch

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

```
// thread 2
```

context switch

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

- Easier to see with pseudo-code…

22

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**)

23

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

24

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 because we could experience a context switch between reading 'note', and setting it

25

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;
```

context switch

context switch

We decide to enter the critical section here…

But only mark the fact here …

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

26

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

27

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 :-)

28

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

- Our critical section doesn't cover enough!

29

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

30

Implementing mutual exclusion

- One option is to prevent context switches
  - e.g. disable interrupts (for kernel threads), or set an in-memory flag (for user threads)
- ENTER_CS() = "disable context switches"; LEAVE_CS() = "re-enable context switches"
- Can work but:
  - Rather brute force (stops all other threads, not just those who want to enter the critical section)
  - Potentially unsafe (if disable interrupts and then sleep waiting for a timer interrupt ;-)
  - And doesn't work across multiple CPUs

31

Implementing mutual exclusion

- Associate a **mutual exclusion lock** with each critical section, e.g. a variable **L**
  - (must ensure use correct lock variable!)

  ENTER_CS() = "LOCK(L)"
  LEAVE_CS() = "UNLOCK(L)"
- Can implement LOCK() using read-and-set():

```
LOCK(L) {
  while(!read-and-set(L))
    ; // do nothing
}
```

```
UNLOCK(L) {
  L = 0;
}
```

32

Solution #3: mutual exclusion locks

```
// thread 1
LOCK(fridgeLock);
beer = checkFridge();
if(!beer)
     buyBeer();
UNLOCK(fridgeLock);
```

```
// thread 2
LOCK(fridgeLock);
beer = checkFridge();
if(!beer)
     buyBeer();
UNLOCK(fridgeLock);
```

- This is – finally! – a correct program
- Still not perfect
  - Lock might be held for quite a long time (e.g. imagine another person wanting to get the milk!)
  - Waiting threads waste CPU time (or worse)
  - **Contention** occurs when consumers have to wait for locks
- Mutual exclusion locks often known as **mutexes**
  - But we will prefer this term for **sleepable locks** – see Lecture 2
  - So think of the above as a **spin lock**

33

Summary + next time

- Definition of a concurrent system
- Origins of concurrency within a computer
- Processes and threads
- Challenge: concurrent access to shared resources
- Critical sections, mutual exclusion, race conditions, and atomicity
- Mutual exclusion locks (mutexes)

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

34