Concurrent Systems
8L for Part IB

Handout 1

Dr Robert Watson
Recommended Reading

• “Operating Systems, Concurrent and Distributed Software Design“, Jean Bacon and Tim Harris, Addison-Wesley 2003
  – or “Concurrent Systems”, (2nd Ed), Jean Bacon, Addison-Wesley 1997

• “Modern Operating Systems”, (3rd Ed), Andrew Tannenbaum, Prentice-Hall 2007

• “Java Concurrency in Practice”, Brian Goetz and others, Addison-Wesley 2006
What is Concurrency?

• Computers can 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”
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 how to handle deadlock; and
  – prevention, avoidance, detection, recovery
• See how abstraction can provide support for correct & fault-tolerant concurrent execution
  – transactions, serializability, concurrency control
Recap: Processes and Threads

• A process is a program in execution
  – Unit of protection & resource allocation
  – Has an associated virtual address space (VAS); 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 VAS of their containing process
  – (or within the kernel address space)
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 ...
Concurrency with a single CPU

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

![Concurrency Diagram](image)

- **time**
  - *timer interrupt*
  - *disk interrupt*
  - *system call*
  - *page fault*

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

```
CPU0
Proc(A) OS Proc(B) OS Proc(B) OS Proc(C) OS

CPU1
Proc(C) OS Proc(D) Proc(A) OS Proc(A)
```

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

• Threads can be user-level or kernel-level
• 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
Threading Models

• Kernel-level threads
  – OS aware of both processes and threads
  – 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 NT, modern Linux, Mac OS X, FreeBSD
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, Solaris LWP, FreeBSD KSE)
  – 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)
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
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 share the address space
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” ;-)
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?
Non-Solution #1: Leave a Note

- Easier to see with pseudo-code...

```c
// 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;
  }
}
```
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...
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
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
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;
  }
}

We decide to enter the critical section here...

But only mark the fact here ...

context switch

context switch
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
Solution #2: Atomic Note

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

```c
// 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 :-)

23
Non-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;
    }
}

• Our critical section doesn’t cover enough!
General Mutual Exclusion

• More generally, we would like the ability to define a region of code as a critical section e.g.

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

```c
// 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
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
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():

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

UNLOCK(L) {
    L = 0;
}
```
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)
What if No Hardware Support?

• Solution #3 requires an atomic ‘read-and-set’ operation... but what if we don’t have one?
• Option 1:
  – Fake atomic operation by disabling interrupts (or context switches) between read and set
  – But doesn’t work across multiple CPUs
• Option 2:
  – Build a mutual exclusion scheme which only relies on atomic reads and writes!
  – Hot topic in the 1970s/80s; mostly irrelevant now
• In practice, we almost always build mutual exclusion on top of atomic instructions like CAS, TAS, LL/SC, ...
<< in case you’re interested >>

- Examples for N-process mutual exclusion are:
  - Lamport L, *A new solution to Dijkstra’s concurrent programming problem*, CACM, 17(8), 1974 (this is his N-process bakery algorithm)

- These algorithms impose large overhead, and may not even be correct in modern CPUs
<< Solution – or Non-Solution? - #4 >>

// thread 1
flag1 = 1;
while(flag2 == 1) ; // do nothing
beer = checkFridge();
if(!beer)
    buyBeer();
flag1 = 0;

// thread 2
flag2 = 1;
if(!flag1) {
    beer = checkFridge();
    if(!beer)
        buyBeer();
}
flag2 = 0;

• Question: does this work?
• (And even if it does, would you want to have to write – or read – this kind of code??)
Semaphores

• Even with atomic operations, busy waiting for a lock is inefficient...
  – Better to sleep until resource available
• Dijkstra (THE, 1968) proposed semaphores
  – New type of variable
  – Initialized once to an integer value (default 0)
• Supports two operations: `wait()` and `signal()`
  – Sometimes called `down()` and `up()`
  – (and originally called `P()` and `V()` ... blurrk!)
Semaphore Implementation

• Implemented as an integer and a queue

```java
wait(sem) {
    if(sem > 0) {
        sem = sem - 1;
    } else suspend caller & add to queue for sem
}

signal(sem) {
    if no threads are waiting {
        sem = sem + 1;
    } else wake up some thread on queue
}
```

• Method bodies are implemented atomically
• “suspend” and “wake” invoke threading APIs
Mutual Exclusion with a Semaphore

- Initialize semaphore to 1; \texttt{wait()} is lock(), \texttt{signal()} is unlock()
Two Process Synchronization

- Initialize semaphore to 0; A proceeds only after B signals

![Diagram showing the synchronization process between processes A and B using a semaphore](image-url)
N-resource Allocation

• Suppose there are N instances of a resource
  – e.g. N printers attached to a DTP system
• Can manage allocation with a semaphore sem, initialized to N
  – Anyone wanting printer does \texttt{wait}(sem)
  – After N people get a printer, next will sleep
  – To release resource, \texttt{signal}(sem)
    • Will wake someone if anyone is waiting
• Will typically also require mutual exclusion
  – e.g. to decide which printers are free
Semaphore Programming Examples

• Semaphores are quite powerful
  – Can solve mutual exclusion...
  – Can also provide condition synchronization
    • Thread waits until some condition is true

• Let’s look at some examples:
  1. One producer thread, one consumer thread, with a N-slot shared memory buffer
  2. Any number of producer and consumer threads, again using an N-slot shared memory buffer
  3. Multiple reader, single writer synchronization
Producer-Consumer Problem

• Shared buffer B[] with N slots, initially empty
• Producer thread wants to:
  – Produce an item
  – If there’s room, insert into next slot;
  – Otherwise, wait until there is room
• Consumer thread wants to:
  – If there’s anything in buffer, remove an item (and consume it)
  – Otherwise, wait until there is something
• General concurrent programming paradigm
  – e.g. pipelines in Unix; staged servers; work stealing
Producer-Consumer Solution

```java
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items = new Semaphore(0);

// producer thread
while(true) {
    item = produce();
    if there is space {
        buffer[in] = item;
        in = (in + 1) % N;
    }
}

// consumer thread
while(true) {
    if there is an item {
        item = buffer[out];
        out = (out + 1) % N;
    }
    consume(item);
}
```
Producer-Consumer Solution

```java
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items = new Semaphore(0);

// producer thread
while(true) {
    item = produce();
    wait(spaces);
    buffer[in] = item;
    in = (in + 1) % N;
    signal(items);
}

// consumer thread
while(true) {
    wait(items);
    item = buffer[out];
    out = (out + 1) % N;
    signal(spaces);
    consume(item);
}
```
Producer-Consumer Solution

• Use of semaphores for N-resource allocation
  – In this case, “resource” is a slot in the buffer
  – “spaces” allocates empty slots (for producer)
  – “items” allocates full slots (for consumer)

• No explicit mutual exclusion
  – threads will never try to access the same slot at the same time; if “in == out” then either
    • buffer is empty (and consumer will sleep on ‘items’), or
    • buffer is full (and producer will sleep on ‘spaces’)

Generalized Producer-Consumer

• Previously had exactly one producer thread, and exactly one consumer thread
• More generally might have many threads adding items, and many removing them
• If so, we **do** need explicit mutual exclusion
  – e.g. to prevent two consumers from trying to remove (and consume) the same item
• Can implement with one more semaphore...
Generalized P-C Solution

```java
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items = new Semaphore(0);
guard = new Semaphore(1); // for mutual exclusion

// producer threads
while(true) {
    item = produce();
    wait(spaces);
    wait(guard);
    buffer[in] = item;
    in = (in + 1) % N;
    signal(guard);
    signal(items);
}

// consumer threads
while(true) {
    wait(items);
    wait(guard);
    item = buffer[out];
    out = (out + 1) % N;
    signal(guard);
    signal(spaces);
    consume(item);
}
```

- Exercise: allow 1 producer and 1 consumer concurrent access
Multiple-Readers Single-Writer

• Another common paradigm is MRSW
  – Shared resource accessed by a set of threads
    • e.g. cached set of DNS results
  – Safe for many threads to read simultaneously, but a writer (updating) must have exclusive access

• Simplest solution uses a single semaphore as a mutual exclusion lock for write access
  – Any writer must wait to acquire this
  – First reader also acquires this; last reader releases it
  – Manage reader counts using another semaphore
Simplest MRSW Solution

```java
int nr = 0; // number of readers
rSem = new Semaphore(1); // protects access to nr
wSem = new Semaphore(1); // protects access to data

// a writer thread
wait(wSem);
.. perform update to data
signal(wSem);

// a reader thread
wait(rSem);
nr = nr + 1;
if (nr == 1) // first in
   wait(wSem);
signal(rSem);
.. read data
wait(rSem);
nr = nr - 1;
if (nr == 0) // last out
   signal(wSem);
signal(rSem);
```

Code for writer is simple...

.. but reader case more complex: must track number of readers, and acquire or release overall lock as appropriate
Simplest MRSW Solution

• Solution on previous slide is “correct”
  – Only one writer will be able to access data structure, but – providing there is no writer – any number of readers can access it

• However writers can **starve**
  – If readers continue to arrive, a writer might wait forever (since readers will not release wSem)
  – Would be fairer if a writer only had to wait for all current readers to exit…
  – Can implement this with an additional semaphore
A Fairer MRSW Solution

```java
int nr = 0; // number of readers
rSem = new Semaphore(1); // protects access to nr
wSem = new Semaphore(1); // protects access to data
turn = new Semaphore(1); // for more fairness!
```

Once a writer tries to enter he will acquire turn…

… which prevents any further readers from entering

```java
// a reader thread
wait(turn);
signal(turn);
wait(rSem);
nr = nr + 1;
if (nr == 1) // first in
  wait(wSem);
signal(rSem);
.. read data
wait(rSem);
nr = nr - 1;
if (nr == 0) // last out
  signal(wSem);
signal(rSem);
```
Semaphores: Summary

• Powerful abstraction for implementing concurrency control:
  – mutual exclusion & condition synchronization

• Better than read-and-set()... **but** correct use requires considerable care
  – e.g. forget to wait(), can corrupt data
  – e.g. forget to signal(), can lead to infinite delay
  – generally get more complex as add more semaphores

• Used internally in some OSes and libraries, but generally deprecated for other mechanisms...