Concurrent systems

Lecture 2: Mutual exclusion and process synchronisation

Dr Robert N. M. Watson

1

Reminder from last 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

2

From last time: beer-buying example

- 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!

We spotted race conditions in obvious concurrent implementations
Ad hoc solutions (e.g., leaving a note) failed
Even naïve application of atomic operations failed
What we want is a general solution for mutual exclusion

This time

- General mutual exclusion: how to [not] do it
- Hardware support for mutual exclusion
- Semaphores for **mutual exclusion**, **process synchronisation**, and **resource allocation**
- Two-party and general producer-consumer relationships
- Multi-reader single-writer locks

4

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

5

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

6

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;
}
```

7

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"

8

Compare and Swap (CAS)

- Found on CISC systems such as x86
- Test and Set (TAS) another variation
- Caller provides previous value as argument
- If memory contents match, assignment occurs
- Return value can be tested to trigger loop

```
        mov           %edx, 1             # New value
spin:
        mov           foo_lock, %eax      # Load old value
        test          %eax, %eax          # If non-zero (owned),
        jnz           spin                #   loop
        lock cmpxchg  %edx, foo_lock      # If foo_lock == %eax,
        test          %eax, %eax          #   swap in value from
        jnz           spin                #   %edx; else loop
```

Load linked-store conditional (LL/SC)

- Found on RISC systems (MIPS, Alpha, ARM, …)
- Load value from memory with LL
- Manipulate value in register
- SC fails if memory modified since load linked
- Return value can be tested to trigger loop

```
spin:
        lld    $t0, 0($a0)         # Load old value
        bnez   $t0, spin           # If non-zero (owned), loop
        dli    $t0, 1             # New value (branch-delay slot)
        scd    $t0, 0($a0)         # Conditional store to $a0
        beqz   $t0, spin           # If failed ($t0 zero), loop
        nop                        # Branch-delay slot
```

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
  - Probably doesn't even work on modern hardware!
- In practice, we almost always build mutual exclusion on top of atomic instructions like CAS, TAS, LL/SC, ...

11

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 <u>originally</u> called **P()** and **V()** ... blurk!)

12

Semaphore implementation

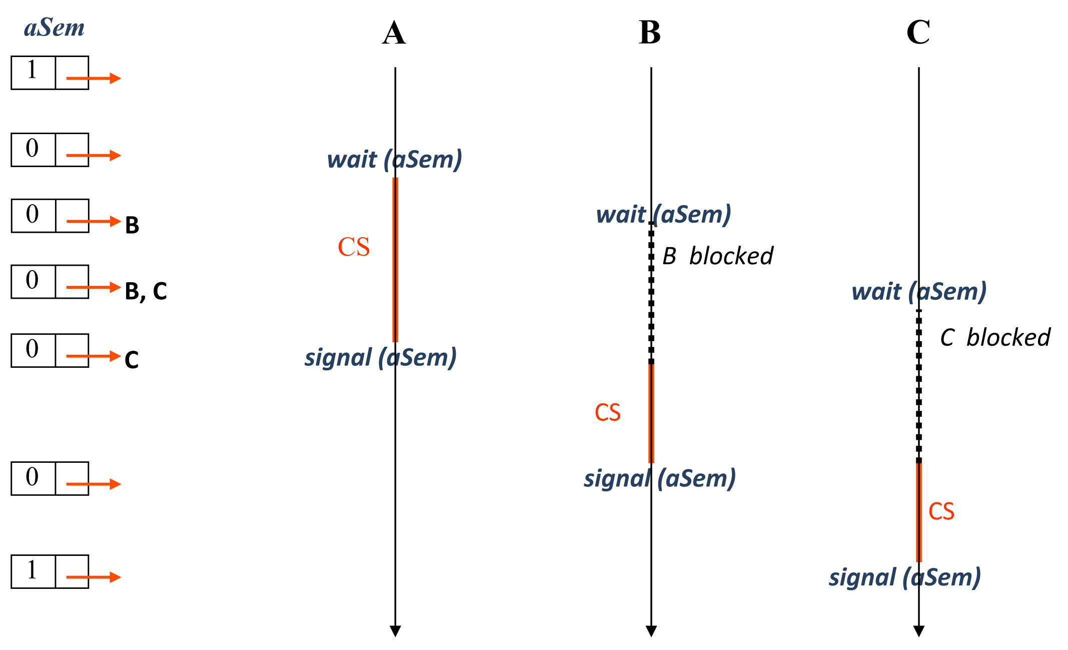
- Implemented as an integer and a queue

```
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
}
```

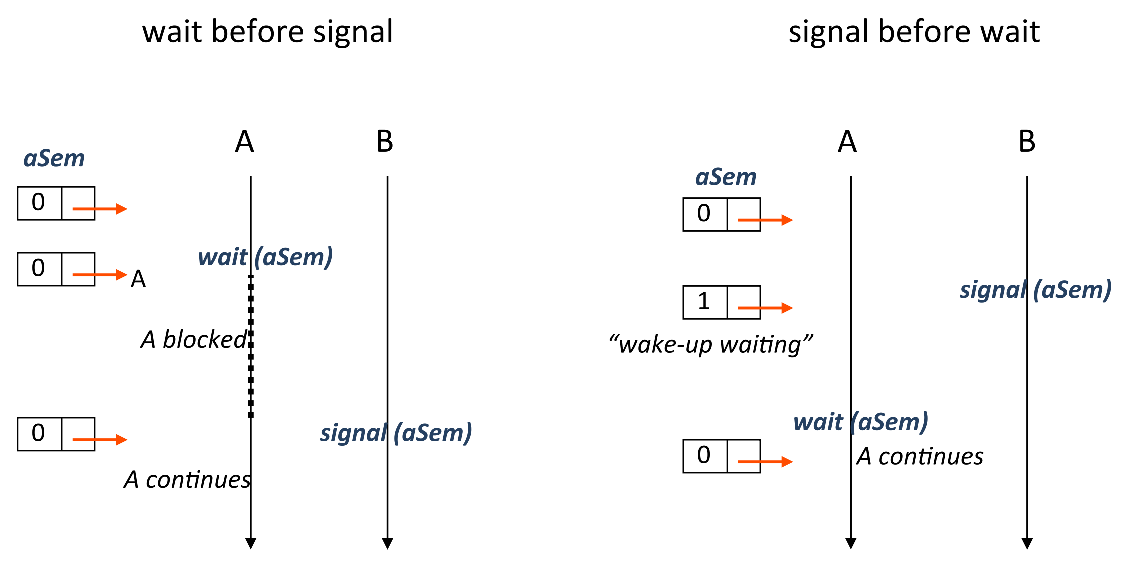
- Method bodies are implemented atomically
- “suspend” and “wake” invoke threading APIs

Mutual exclusion with a semaphore



- Initialize semaphore to 1; **wait()** is lock(), **signal()** is unlock()

Two process synchronization



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

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 **wait**(sem)
  - After N people get a printer, next will sleep
  - To release resource, **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 set by another thread becomes 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

17

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

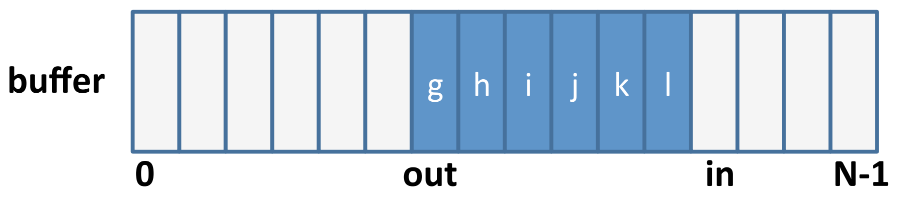
18

Producer-consumer solution

```
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);
}
```

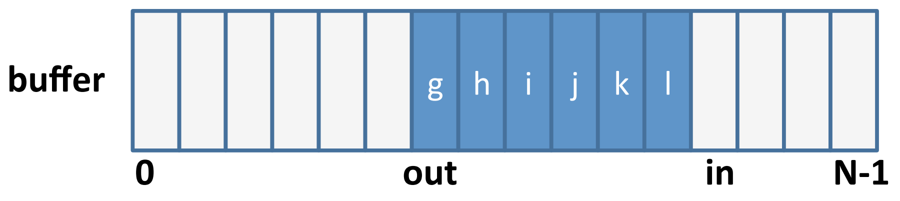


Producer-consumer solution

```
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’)

21

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…

22

Generalized P-C solution

```
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

23

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
  - **Data stability** vs. **mutual exclusion**
- 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

24

Simplest MRSW solution

```
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);
```

Code for writer is simple…

```
// 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);
```

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

25

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

26

A fairer MRSW solution

```
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

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

```
// 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);
```

27

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…

28

Summary + next time

- General mutual exclusion: how to [not] do it
- Hardware support for mutual exclusion
- Semaphores for **mutual exclusion**, **process synchronisation**, and **resource allocation**
- Two-party and generalised producer-consumer relationships
- Multi-reader single-writer locks

- Next time:
  - Conditional critical regions (CCRs); Monitors
  - Condition variables; signal-and-wait vs. signal-and-continue
  - Concurrency in practice; concurrency primitives wrap-up

29