Concurrent systems

Lecture 2: More mutual exclusion, semaphores, and producer-consumer relationships

Dr Robert N. M. Watson

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
- Mutual exclusion locks (mutexes)

From last lecture

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
Mutexes provide a general mechanism for mutual exclusion

This time

- Implementing **mutual exclusion**
- Hardware support for **atomicity**, **condition synchronisation**
- Semaphores for mutual exclusion, condition synchronisation, and **resource allocation**
- Two-party and generalised **producer-consumer** relationships

4

From last lecture

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;
}
```

5

Hardware foundations for atomicity

- How can we implement atomic read-and-set?
- Simple pair of load and store instructions fail the atomicity test (obviously divisible!)
- Need a new **ISA primitive** for protection against parallel access to memory from another CPU
- Two common flavours:
  - Atomic **Compare and Swap** (CAS)
  - **Load Linked, Store Conditional** (LL/SC)
  - Atomic conditionals: if a race is lost, software will retry
- NB: May need to **disable interrupts** (i.e., preemption)
  - Typically a special supervisor-only instruction

6

Atomic Compare and Swap (CAS)

- Instruction operands memory address, prior + new values
  - If prior value **matches** in-memory value, **new value stored**
  - If prior value **does not match** in-memory value, **instruction fails**
  - Software checks return value, can loop on failure
- Found on CISC systems such as x86 (`cmpxchg`)
  - Atomic **Test and Set** (TAS) another variation
  - NB: Also added to recent ARMv8 ISA revision – why?

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

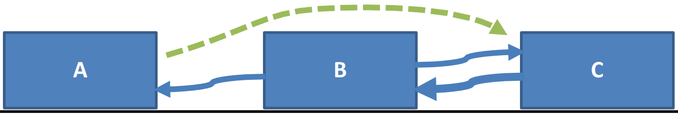
Load Linked-Store Conditional (LL/SC)

- Found on RISC systems (MIPS, Alpha, ARM, …)
  - Load value from memory location with **LL**
  - Manipulate value in register (e.g., add, assign, …)
  - **SC** fails if memory location modified since **LL**
  - **SC** writes back register indicating success (or not)
  - Software checks return value, can loop on failure
- Foundation for a more general technique seeing early deployment: **Software Transactional Memory** (STM)

```
spin:
        lld     $t0, 0($a0)    # Load prior 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
```

Mutual exclusion and invariants

- One important goal of locking is to avoid exposing **inconsistent intermediate states** to other threads
- This suggests a more general invariants strategy:
  - Invariants **hold** when mutex is acquired
  - Invariants **may be violated** while mutex is held
  - Invariants **must be restored** before mutex is released
- E.g., deletion from a doubly linked list
  - Invariant: an entry is in the list, or not in the list
  - Individually non-atomic updates of forward and backward pointers around a deleted object are fine as long as the lock isn't released in between the two pointer writes



9

Semaphores

- Even with atomic operations, busy waiting for a lock is inefficient…
  - Recall from last lecture: **lock contention**
  - 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()** ... blurk!)
- Also provides **condition synchronisation**
  - Wake up another waiting thread on a condition or event
  - E.g., "There is an item available for processing in a queue"

10

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

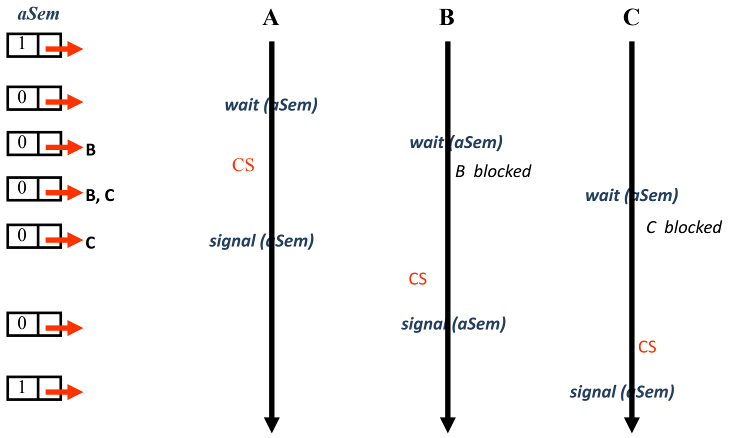
11

Hardware support for wakeups: IPIs

- CAS/LLSC/… support atomicity via shared memory
- But what about “**wake up thread**”?
  - On a single CPU, wakeup triggers context switch
  - How to wake up a thread on another CPU that is already busy doing something else?
- **Inter-Processor Interrupts** (IPIs)
  - Wakeup sends an interrupt to the target CPU
  - IPI handler runs thread scheduler, preempts running thread, triggers context switch
- Together, shared memory and IPIs provide **atomicity** and **condition synchronisation** between CPUs

12

Mutual exclusion with a semaphore



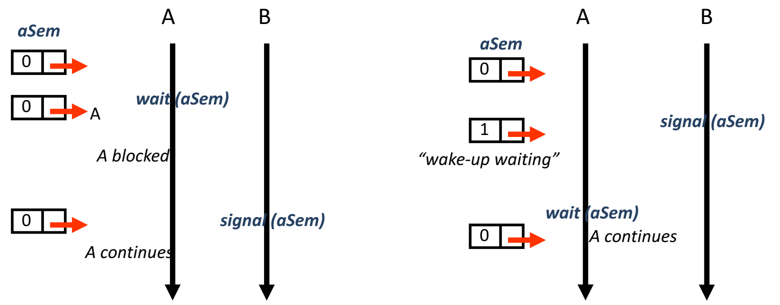
- Initialize semaphore to 1; **wait()** is lock(), **signal()** is unlock()

13

Condition synchronisation

wait before signal

signal before wait



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

14

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

15

Semaphore design patterns

- Semaphores are quite powerful
  - Can solve mutual exclusion…
  - Can also provide **condition synchronization**
    - Thread waits until some **condition** set by another thread
- 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

16

Producer-consumer problem

- General "pipe" concurrent programming paradigm
  - E.g. pipelines in Unix; staged servers; work stealing; download thread vs. rendering thread in web browser
- 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 (+consume it)
  - Otherwise, wait until there is something
- Maintain order, use parallelism, avoid context switches

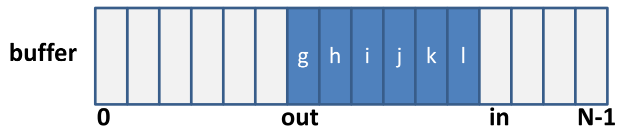
17

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);
}
```



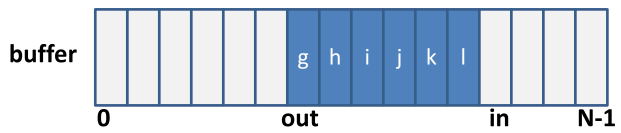
18

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();
  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…

21

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

22

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…

Summary + next time

- Implementing **mutual exclusion**: hardware support for **atomicity** and **inter-processor interrupts**
- Semaphores for mutual exclusion, **condition synchronisation**, and **resource allocation**
- Two-party and generalised **producer-consumer** relationships
- **Starvation** and **fairness**

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