Concurrent systems
Lecture 3: CCR, monitors, and concurrency in practice

Dr Robert N. M. Watson

Reminder from last 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
• Invariants and locks
From last time: 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...

Semaphores are a low-level implementation primitive – they say **what to do**, rather than describe **programming goals**

This time

• Multi-Reader Single-Writer (MRSW) locks
  – Starvation and fairness
• Alternatives to semaphores/locks:
  – Conditional critical regions (CCRs)
  – Monitors
  – Condition variables
    – Signal-and-wait vs. signal-and-continue semantics
• Concurrency primitives in practice
• Concurrency primitives wrap-up
Multiple-Readers Single-Writer (MRSW)

- Another common synchronisation 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
  - MRSW locks have read lock and write lock operations
  - Mutual exclusion vs. data stability
- Simple implementation uses a two semaphores
- First semaphore is a mutual exclusion lock (mutex)
  - Any writer must wait to acquire this
- Second semaphore protects a reader count
  - Reader count incremented whenever a reader enters
  - Reader count decremented when a reader exits
  - First reader acquires mutex; last reader releases mutex

Simplest MRSW solution

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

```
int nr = 0;                 // number of readers
rSem = new Semaphore(1);   // protects access to nr
wSem = new Semaphore(1);   // protects writes to data
turn = new Semaphore(1);   // write is awaiting a turn

Once a writer tries to enter, it 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);
```
Conditional Critical Regions

- Implementing synchronisation with locks is difficult
  - Only the developer knows what data is protected by which locks
- One early (1970s) effort to address this problem was CCRs
  - Variables can be explicitly declared as ‘shared’
  - Code can be tagged as using those variables, e.g.
    ```
    shared int A, B, C;
    region A, B {
        await( /* arbitrary condition */);
        // critical code using A and B
    }
    ```
- Compiler automatically declares and manages underlying primitives for mutual exclusion or synchronization
  - e.g. wait/signal, read/await/advance, ...
- Easier for programmer (c/f previous implementations)

CCR example: Producer-Consumer

```
shared int buffer[N];
shared int in = 0; shared int out = 0;

// producer thread
while(true) {
    item = produce();
    region in, out, buffer {
        await((in-out) < N);
        buffer[in % N] = item;
        in = in + 1;
    }
}

// consumer thread
while(true) {
    region in, out, buffer {
        await((in-out) > 0);
        item = buffer[out % N];
        out = out + 1;
    }
    consume(item);
}
```

- Explicit (scoped) declaration of critical sections
  - automatically acquire mutual exclusion lock on region entry
- Powerful `await()`: any evaluable predicate
CCR pros and cons

• On the surface seems like a definite step up
  – Programmer focuses on variables to be protected, compiler generates appropriate semaphores (etc)
  – Compiler can also check that shared variables are never accessed outside a CCR
  – (still rely on programmer annotating correctly)
• But await(expr) is problematic...
  – What to do if the (arbitrary) expr is not true?
  – very difficult to work out when it becomes true?
  – Solution was to leave region & try to re-enter: this is busy waiting, which is very inefficient...

Monitors

• Monitors are similar to CCRs (implicit mutual exclusion), but modify them in two ways
  – Waiting is limited to explicit condition variables
  – All related routines are combined together, along with initialization code, in a single construct
• Idea is that only one thread can ever be executing ‘within’ the monitor
  – If a thread calls a monitor method, it will block (enqueue) if another thread is holding the monitor
  – Hence all methods within the monitor can proceed on the basis that mutual exclusion has been ensured
• Java’s synchronized primitive implements monitors
Example Monitor syntax

```
monitor <foo> {
  // declarations of shared variables
  // set of procedures (or methods)
  procedure p1(...) { ... }
  procedure p2(...) { ... }
  ...
  procedure pn(...) { ... }
  {
    /* monitor initialization code */
  }
}
```

- All related data and methods kept together
- Shared variables only accessible from within monitor methods
- Invoking any procedure causes an (implicit) mutual exclusion lock to be taken
- Shared variables can be initialized here

Condition Variables

- Mutual exclusion not always sufficient
  - Condition synchronization -- e.g., wait for a condition to occur
- Monitors allow condition variables
  - Explicitly declared and managed by programmer
  - NB: No integrated counter – not a stateful semaphore!
  - Support three operations:

```
wait(cv) {
  suspend thread and add it to the queue for cv,
  release monitor lock;
}
signal(cv) {
  if any threads queued on cv, wake one thread;
}
broadcast(cv) {
  wake all threads queued on cv;
}
```
Monitor Producer-Consumer solution?

```c
monitor ProducerConsumer {
    int in, out, buf[N];
    condition notfull = TRUE, notempty = FALSE;

    procedure produce(item) {
        if ((in-out) == N) wait(notfull);
        buf[in % N] = item;
        if ((in-out) == 0) signal(notempty);
        in = in + 1;
    }

    procedure int consume() {
        if ((in-out) == 0) wait(notempty);
        item = buf[out % N];
        if ((in-out) == N) signal(notfull);
        out = out + 1;
        return(item);
    }

    /* init */ { in = out = 0; }
}
```

If buffer is full, wait for consumer
If buffer was empty, signal the consumer
If buffer was full, signal the producer
If buffer was empty, wait for producer

Does this work?

- Depends on implementation of `wait()` & `signal()`
- Imagine two threads, T1 and T2
  - T1 enters the monitor and calls `wait(C)` – this suspends T1, places it on the queue for C, and unlocks the monitor
  - Next T2 enters the monitor, and invokes `signal(C)`
  - Now T1 is unblocked (i.e. capable of running again)...
  - … but can only have one thread active inside a monitor!
- If we let T2 continue (signal-and-continue), T1 must queue for re-entry to the monitor
  - And no guarantee it will be next to enter
- Otherwise T2 must be suspended (signal-and-wait), allowing T1 to continue...
Signal-and-Wait ("Hoare Monitors")

- Consider a queue $E$ to enter monitor
  - If monitor is occupied, threads are added to $E$
  - May not be FIFO, but should be fair
- If thread $T_1$ waits on $C$, added to queue $C$
- If $T_2$ enters monitor & signals, waking $T_1$
  - $T_2$ is added to a new queue $S$ “in front of” $E$
  - $T_1$ continues and eventually exits (or re-waits)
- Some thread on $S$ chosen to resume
  - Only admit a thread from $E$ when $S$ is empty

Signal-and-Wait pros and cons

- We call \texttt{signal()} exactly when condition is true, then directly transfer control to waking thread
  - Hence condition will still be true!
- But more difficult to implement...
- And can be complex to reason about (a call to \texttt{signal} may or may not result in a context switch)
  - Hence we must ensure that any invariants are maintained at time we invoke \texttt{signal()}
- With these semantics, our example is broken:
  - We \texttt{signal()} before incrementing in/out
Monitor Producer-Consumer solution?

```plaintext
monitor ProducerConsumer {
    int in, out, buf[N];
    condition notfull = TRUE, notempty = FALSE;
    procedure produce(item) {
        if ((in-out) == N) wait(notfull);
        buf[(in % N)] = item;
        if ((in-out) == 0) signal(notempty);
        in = in + 1;
    }
    procedure int consume() {
        if ((in-out) == 0) wait(notempty);
        item = buf[(out % N)];
        if ((in-out) == N) signal(notfull);
        out = out + 1;
        return(item);
    }
    /* init */ { in = out = 0; }
}
```

Signal-and-Continue

- Alternative semantics introduced by Mesa programming language (Xerox PARC)
- An invocation of `signal()` moves a thread from the condition queue C to the entry queue E
  - Invoking threads continues until exits (or waits)
- Simpler to build... but now not guaranteed that condition is true when resume!
  - Other threads may have executed after the signal, but before you continue
Signal-and-Continue example (1)

- Consider multiple producer-consumer threads
  1. P1 enters. Buffer is full so blocks on queue for C
  2. C1 enters.
  3. P2 tries to enter; occupied, so queues on E
  4. C1 continues, consumes, and signals C (“notfull”)
  5. P1 unblocks; monitor occupied, so queues on E
  6. C1 exits, allowing P2 to enter
  7. P2 fills buffer, and exits monitor
  8. P1 resumes and tries to add item – BUG!

- Hence must re-test condition:
  - i.e. while( (in - out) == N) wait(not full);
Monitor Producer-Consumer solution?

```c
monitor ProducerConsumer {
    int in, out, buf[N];
    condition notfull = TRUE, notempty = FALSE;

    procedure produce(item) {
        while ((in-out) == N) wait(notfull);
        if ((in-out) == 0) signal(notempty);
        buf[in % N] = item;
        in = in + 1;
    }

    procedure int consume() {
        while ((in-out) == 0) wait(notempty);
        item = buf[out % N];
        if ((in-out) == N) signal(notfull);
        out = out + 1;
        return(item);
    }

    /* init */ { in = out = 0; }
}
```

- if() replaced with while() for conditions

Monitors: summary

- Structured concurrency control
  - groups together shared data and methods
  - (today we’d call this object-oriented)
- Considerably simpler than semaphores, but still perilous in places
- May be overly conservative sometimes:
  - e.g. for MRSW cannot have >1 reader in monitor
  - Typically must work around with entry and exit methods (BeginRead(), EndRead(), BeginWrite(), etc)
- Exercise: sketch a MRSW monitor implementation
Concurrency in practice

• Seen a number of abstractions for concurrency control
  – Mutual exclusion and condition synchronization

• Next let’s look at some concrete examples:
  – POSIX pthreads (C/C++ API)
  – FreeBSD kernels
  – Java

Example: pthreads

• Standard (POSIX) threading API for C, C++, etc
  • mutexes, condition variables, and barriers
  • Mutexes are essentially binary semaphores:

```c
int pthread_mutex_init(pthread_mutex_t *mutex, ...);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

• A thread calling lock() blocks if the mutex is held
  – trylock() is a non-blocking variant: returns immediately; returns 0 if lock acquired, or non-zero if not.
Example: pthreads

• Condition variables are Mesa-style:

```c
int pthread_cond_init(pthread_cond_t *cond, ...);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

• No proper monitors: must manually code e.g.

```c
pthread_mutex_lock(&M);
while (!condition)
  pthread_cond_wait(&C,&M);
// do stuff
if (condition)
  pthread_cond_broadcast(&C);
pthread_mutex_unlock(&M);
```

Notice: while() and not if() due to signal-and-continue semantics

Example: pthreads

• **Barriers**: explicit synchronization mechanism
  • Wait until all threads reach some point
  • E.g., in discrete event simulation, all parallel threads must complete one epoch before any begin on the next

```c
int pthread_barrier_init(pthread_barrier_t *b, ..., N);
int pthread_barrier_wait(pthread_barrier_t *b);
```

```c
void worker() {
  while(!done) {
    // do work for this round
    pthread_barrier_wait(&B);
  }
}
```
Example: FreeBSD kernel

- Kernel provides spin locks, mutexes, conditional variables, reader-writer + read-mostly locks
  - Semantics (roughly) modeled on POSIX threads
- A variety of **deferred work primitives**
  - “Fully preemptive” and highly threaded (e.g., interrupt processing in threads)
- Interesting debugging tools such as DTrace, **lock contention measurement**, **lock-order checking**
- Concurrency case study for our last lecture

Example: Java synchronization (1)

- Inspired by monitors – objects have **intrinsic locks**
- **Synchronized methods:**

```java
public synchronized void myMethod() throws ...
{
    // This code runs with the intrinsic lock held.
}
```

- **Synchronized statements:**

```java
public void myMethod() throws ...
{
    synchronized(this) {
        // This code runs with the intrinsic lock held.
    }
}
```

- Method return / statement exit release lock
- Locks are **reentrant**: a single thread can reenter **synchronized** statements/methods without waiting
- **synchronized()** can accept other objects than **this**
Example: Java synchronization (2)

- Objects have condition variables for guarded blocks
- `wait()` puts the thread to sleep:

```java
public synchronized void waitDone() {
    while (!done) {
        wait();
    }
}
```

- `notify()` and `notifyAll()` wake threads up:

```java
public synchronized void notifyDone() {
    done = true;
    notifyAll();
}
```

- As with Mesa, signal-and-continue semantics
- As with locks, can name object `(thatObject.wait())`

Example: Java synchronization (3)

- Java also specifies memory consistency and atomicity properties that make some lock-free concurrent access safe – if used very carefully
  – We will consider lock-free structures later in the term
- `java.util.concurrent` (especially as of Java 8) includes many higher-level primitives – for example, thread pools, concurrent collections, semaphores, cyclic barriers, ...
- Because Java is a type-safe, managed language, it is a much safer place to experiment with concurrent programming than (for example) C
Concurrency Primitives: Summary

- Concurrent systems require means to ensure:
  - **Safety** (mutual exclusion in critical sections), and
  - **Progress** (condition synchronization)
- Spinlocks (busy wait); semaphores; MRSWs, CCRs, and monitors
  - Hardware primitives for synchronisation
  - Signal-and-Wait vs. Signal-and-Continue
- Many of these are used in practice
  - Subtle minor differences can be dangerous
  - Much care required to avoid bugs
  - E.g., “lost wakeups” – signal w/o waiter
- More detail on implementation in our case study

Summary + next time

- **Multi-Reader Single-Writer** (MRSW) locks
- Alternatives to semaphores/locks:
  - Conditional critical regions (CCRs)
  - Monitors
  - Condition variables
  - Signal-and-wait vs. signal-and-continue semantics
- Concurrency primitives in practice
- Concurrency primitives wrap-up

Next time:
- Problems with concurrency: deadlock, livelock, priorities
- Resource allocation graphs; deadlock {prevention, detection, recovery}
- Priority and scheduling; priority inversion; priority inheritance