

Concurrent systems

Lecture 3: CCR, monitors, and
concurrency in practice

Dr Robert N. M. Watson

1

Reminder from last 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
- **Multi-Reader Single-Writer** (MRSW) locks

2

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

- Alternatives to simple 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

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)

5

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

6

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...

7

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 invokes a monitor method, it will block (queue) if there is another thread active inside
 - Hence all methods within the monitor can proceed on the basis that mutual exclusion has been ensured
- Java's **synchronized** primitive implements monitors

8

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

Invoking any procedure causes an [implicit] mutual exclusion lock to be taken

Shared variables can be initialized here

9

Condition Variables

- Mutual exclusion not always sufficient
 - e.g. may need to wait for a condition to occur
- Monitors allow **condition variables**
 - Explicitly declared & managed by programmer
 - 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;
}
broadcast(cv) {
  wake all threads queued on cv;
}

```

10

Monitor Producer-Consumer solution?

```

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 ($in==out+N$),
must wait for consumer

If buffer was full ($in==out$),
signal the consumer

If buffer is empty ($in==out$),
must wait for producer

If buffer was full before,
signal the producer

11

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...

12

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 **T1** waits on **C**, added to queue **C**
- If **T2** enters monitor & signals, waking **T1**
 - **T2** is added to a new queue **S** “in front of” **E**
 - **T1** continues and eventually exits (or re-waits)
- Some thread on **S** chosen to resume
 - Only admit a thread from **E** when **S** is empty

13

Signal-and-Wait pros and cons

- We call `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 difficult to reason about (a call to `signal` *may or may not* result in a context switch)
 - Hence we must ensure that any invariants are maintained at time we invoke `signal()`
- With these semantics, our example is broken:
 - we `signal()` before incrementing in/out

14

Same code as slide 11

Monitor Producer-Consumer solution?

```

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 ($in == out + N$),
must wait for consumer

If buffer was full ($in == out$),
signal the consumer

If buffer is empty ($in == out$),
must wait for producer

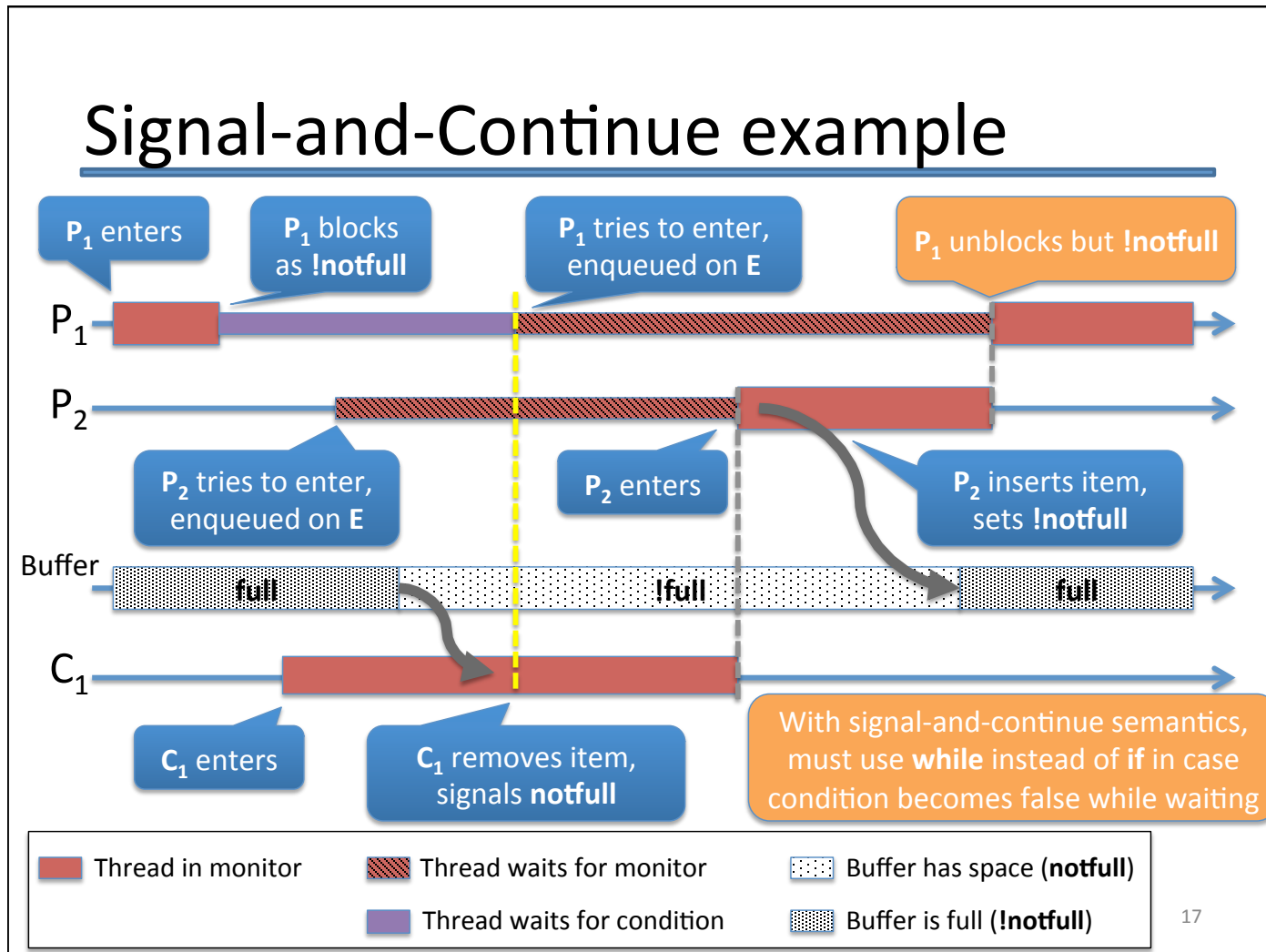
If buffer was full before,
signal the producer

15

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

16



Signal-and-Continue example

- 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(notfull);`

if() replaced with while() for conditions

Monitor Producer-Consumer solution?

```

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

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

```

While buffer is full ($in == out + N$),
must wait for consumer

If buffer was full ($in == out$),
signal the consumer

While buffer is empty ($in == out$),
must wait for producer

If buffer was full before,
signal the producer

19

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

20

Concurrency in practice

- Seen a number of abstractions for concurrency control
 - Mutual exclusion and condition synchronization
- Next let's look at some concrete examples:
 - FreeBSD kernels
 - POSIX pthreads (C/C++ API)
 - Java
 - C#

21

Example: FreeBSD kernel

- Kernel provides spin locks, mutexes, conditional variables, reader-writer + read-mostly locks
- 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



22

Example: pthreads

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

```
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.

23

Example: pthreads

- Condition variables are Mesa-style:

```
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.

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

24

Example: pthreads

- Barriers: explicit synchronization mechanism
 - Wait until all threads reach some point

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

```
pthread_barrier_init(&B, ..., NTHREADS);
for(i=0; i<NTHREADS; i++)
    pthread_create(..., worker, ...);

worker() {
    while(!done) {
        // do work for this round
        pthread_barrier_wait(&B);
    }
}
```

25

Example: Java [original]

- Synchronization inspired by monitors
 - Objects already encapsulate data & methods!
- Mesa-style, but no explicit condition variables

```
public class MyClass {
    //
    public synchronized void myMethod() throws ...{
        while(!condition)
            wait();
        // do stuff
        if(condition)
            notifyAll();
    }
}
```

- Java 5 provides many additional options...

26

Example: C#

- Very similar to Java, but with explicit arguments

```
public class MyClass {
    //
    public void myMethod() {
        lock(this) {
            while(!condition)
                Monitor.Wait(this);
            // do stuff
            if(condition)
                Monitor.PulseAll(this);
        }
    }
}
```

- Also provides spinlocks, reader-writer locks, semaphores, barriers, event synchronization, ...

27

Concurrency Primitives: Summary

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

28

Summary + next time

- Alternatives to simple 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 inversion; priority inheritance