Operating Systems

06. Synchronization

Paul Krzyzanowski
Rutgers University
Spring 2015

Concurrency

Concurrent threads/processes (informal)

– Two processes are concurrent if they run at the same time or if their execution is interleaved *in any order*

Asynchronous

– The processes require occasional synchronization

Independent

– They do not have any reliance on each other

Synchronous

– Frequent synchronization with each other – order of execution is guaranteed

Parallel

– Processes run at the same time on separate processors

Race Conditions

A race condition is a bug:

– The outcome of concurrent threads are unexpectedly dependent on a specific sequence of events.

Example

– Your current bank balance is $1,000.
– Withdraw $500 from an ATM machine while a $5,000 direct deposit is coming in

Execute concurrently

Withdrawal	Deposit
• Read account balance • Subtract 500 • Write account balance	• Read account balance • Add 5000 • Write account balance

Possible outcomes:

Total balance = $5500 $500 $6000

Synchronization

Synchronization deals with developing techniques to avoid race conditions

Something as simple as

x = x + 1;

Compiles to this and may cause a race condition:

```
movl   _x (%rip), %eax
addl   $1, %eax
movl   %eax, _x (%rip)
```

Potential points of preemption for a race condition

Mutual Exclusion

Critical section:

Region in a program where race conditions can arise

Mutual exclusion:

Allow only one thread to access a critical section at a time

Deadlock:

A thread is perpetually blocked (circular dependency on resources)

Starvation:

A thread is perpetually denied resources

Livelock:

Threads run but with no progress in execution

Avoid race conditions with locks

- Grab and release locks around critical sections
- Wait if you cannot get a lock

Execute concurrently

	Withdrawal	Deposit	
Enter Critical Section	• Acquire(transfer_lock)	• Acquire(transfer_lock)	Enter Critical Section
Critical Section	• Read account balance • Subtract 500 • Write account balance	• Read account balance • Add 5000 • Write account balance	Critical Section
Exit Critical Section	• Release(transfer_lock)	• Release(transfer_lock)	Exit Critical Section

The Critical Section Problem

Design a protocol to allow threads to enter a critical section

Conditions for a solution

- Mutual exclusion: No threads may be inside the same critical sections simultaneously
- Progress: If no thread is executing in its critical section but one or more threads want to enter, the selection of a thread cannot be delayed indefinitely.
  - If one thread wants to enter, it should be permitted to enter.
  - If multiple threads want to enter, exactly one should be selected.
- Bounded waiting: No thread should wait forever to enter a critical section
- No thread running outside its critical section may block others
- A good solution will make no assumptions on:
  - No assumptions on # processors
  - No assumption on # threads/processes
  - Relative speed of each thread

Critical sections & the kernel

- Multiprocessors
  - Multiple processes on different processors may access the kernel simultaneously
  - Interrupts may occur on multiple processors simultaneously
- Preemptive kernels
  - Preemptive kernel: process can be preempted while running in kernel mode (the scheduler may preempt a process even if it is running in the kernel)
  - Nonpreemptive kernel: processes running in kernel mode cannot be preempted (but interrupts can still occur!)
- Single processor, nonpreemptive kernel
  - Free from race conditions!

Solution #1: Disable Interrupts

Disable all system interrupts before entering a critical section and re-enable them when leaving

Bad!

- Gives the thread too much control over the system
- Stops time updates and scheduling
- What if the logic in the critical section goes wrong?
- What if the critical section has a dependency on some other interrupt, thread, or system call?
- What about multiple processors? Disabling interrupts affects just one processor

Advantage

- Simple, guaranteed to work
- Was often used in the uniprocessor kernels

Solution #2: Software Test & Set Locks

Keep a shared lock variable:

```
while (locked) ;
locked = 1;
/* do critical section */
locked = 0;
```

Disadvantage:

- Buggy! There's a race condition in setting the lock

Advantage:

- Simple to understand. It's been used for things such as locking mailbox files

Solution #3: Lockstep Synchronization

Take turns

Thread 0

```
while (turn != 0);
critical_section();
turn = 1;
```

Thread 1

```
while (turn != 1);
critical_section();
turn = 0;
```

Disadvantage:

- Forces strict alternation; if thread 2 is really slow, thread 1 is slowed down with it. Turns *asynchronous threads* into *synchronous threads*

Software solutions for mutual exclusion

- Peterson's solution (page 207 of text) , Dekker's, & others
- Disadvantages:
  - Difficult to implement correctly
    Have to rely on `volatile` data types to ensure that compilers don't make the wrong optimizations
  - Difficult to implement for an arbitrary number of threads

Let's turn to hardware for help

Help from the processor

Atomic (indivisible) CPU instructions that help us get locks

- Test-and-set
- Compare-and-swap
- Fetch-and-Increment

These instructions execute in their entirety: they cannot be interrupted or preempted partway through their execution

Test & Set

Set the lock but get told if it already was set (in which case you don't have it)

ATOMIC

```
int test_and_set(int *x) {
    last_value = *x;
    *x = 1;
    return last_value;
}
```

How you use it to lock a critical section (i.e., enforce mutual exclusion):

```
while (test_and_set(&lock) == 1) ;  /* spin */
/* do critical section */
lock = 0;    /* release the lock */
```


Compare & swap (CAS)

Compare the value of a memory location with an old value. If they match then replace with a new value

ATOMIC

```
int compare_and_swap(int *x, int old, int new) {
    int save = *x;
    if (save == old)
        *x = new;
    return save;  /* always return location contents */
}
```

How you use it to grab a critical section:
Avoid the race condition – set *locked* to 1 only if *locked* was still set to 0.

```
while (compare_and_swap(&locked, 0, 1) != 0) ;
          /* spin until locked == 0 */
/* if we got here, locked got set to 1 and we have it */
/* do critical section */
locked = 0;  /* release the lock */
```


Fetch & Increment

Increment a memory location; return previous value

ATOMIC

```
int fetch_and_increment(int *x) {
    last_value = *x;
    *x = *x + 1;
    return last_value;
}
```

Fetch & Increment

Check that it's your turn for the critical section

Ticket lock

```
ticket = 0; turn = 0;
...
myturn = fetch_and_increment(&ticket);
while (turn != myturn) ;
/* do critical section */
fetch_and_increment(&turn);
```

turn

ticket

The problem with spin locks

- All these solutions require busy waiting
  - Tight loop that spins waiting for a turn: busy waiting or spin lock
- Nothing useful gets done!
  - Wastes CPU cycles

Priority Inversion

- Spin locks may lead to priority inversion
- The process with the lock may not be allowed to run!
  - Suppose a lower priority process obtained a lock
  - Higher priority process is always ready to run but loops on trying to get the lock
  - Scheduler always schedules the higher-priority process
  - Priority inversion
    - If the low priority process would get to run & release its lock, it would then accelerate the time for the high priority process to get a chance to get the lock and do useful work
    - Try explaining that to a scheduler!

Priority Inheritance

- Technique to avoid priority inversion
- Increase the priority of any process in a critical section to the maximum of any process waiting on any resource for which the process has a lock
- When the lock is released, the priority goes to its normal level

Spin locks aren't great

Can we block until we can get the critical section?

How about this?

```
public class Lock
{
    private int val = UNLOCKED;
    private ThreadQueue waitQueue = new ThreadQueue();

    public void acquire() {
        Thread me = Thread.currentThread();
        while (TestAndSet(val) == LOCKED) {
            waitQueue.waitForAccess(me);  // Put self in queue
            Thread.sleep();        // Put self to sleep
        }
        // Got the lock
    }
    public void release() {
        Thread next = waitQueue.nextThread();
        val = UNLOCKED;
        if (next != null)
            next.ready(); // Wake up a waiting thread
    }
}
```


Sorry…

- Accessing the wait queue is a critical section
  - Need to add mutual exclusion
- Need extra lock check in *acquire*
  - Thread may find the lock busy
  - Another thread may release the lock but before the first thread enqueues itself
- This can get ugly!

Semaphores

- Count # of wake-ups saved for future use
- Two atomic operations:

```
down(sem s) {
    if (s > 0)
        s = s - 1;
    else
        sleep on event s
}

up(sem s) {
    if (someone is waiting on s)
        wake up one of the threads
    else
        s = s + 1;
}
```

```
//initialize
mutex = 1;

down(&mutex)

// critical section

up(&mutex)
```

Binary semaphore

Semaphores

Count the number of threads that may enter a critical section at any given time.

- Each *down* decreases the number of future accesses
- When no more are allowed, processes have to wait
- Each *up* lets a waiting process get in

Producer-Consumer example

- Producer
  - Generates items that go into a buffer
  - Maximum buffer capacity = *N*
  - If the producer fills the buffer, it must wait (sleep)
- Consumer
  - Consumes things from the buffer
  - If there's nothing in the buffer, it must wait (sleep)
- This is known as the *Bounded-Buffer Problem*

Producer-Consumer example

```
sem mutex=1, empty=N, full=0;
producer() {
    for (;;) {
        produce_item(&item);   // produce something
        down(&empty);          // decrement empty count
        down(&mutex);          // start critical section
        enter_item(item);      // put item in buffer
        up(&mutex);            // end critical section
        up(&full);             // +1 full slot
    }
}
consumer() {
    for (;;) {
        down(&full);           // one less item
        down(&mutex);          // start critical section
        remove_item(item);     // get the item from the buffer
        up(&mutex);            // end critical section
        up(&empty);            // one more empty slot
        consume_item(item);    // consume it
    }
}
```


Readers-Writers example

- Shared data store (e.g., database)
- Multiple processes can read concurrently
- Allow only one process to write at a time
  - And no readers can read while the writer is writing

Readers-Writers example

```
sem mutex=1;       // critical sections used only by the reader
sem canwrite=1;    // critical section for N readers vs. 1 writer
int readcount = 0; // number of concurrent readers

writer() {
   for (;;) {
      down(&canwrite);        // block if we cannot write
      // write data
      up(&canwrite);          // end critical section
   }
}
```


Readers-Writers example

```
sem mutex=1;       // critical sections used only by the reader
sem canwrite=1;    // critical section for N readers vs. 1 writer
int readcount = 0; // number of concurrent readers

reader() {
   for (;;) {
      down(&mutex);
      readcount++;
      if (readcount == 1)  // first reader
         down(canwrite);  // sleep or disallow the writer from writing
      up(&mutex);
      // do the read
      down(&mutex);
      readcount--;
      if (readcount == 0)
         up(canwrite);    // no more readers! Allow the writer access
      up(&mutex);
      // other stuff
   }
}
```

critical section

critical section

Event Counters

Avoid race conditions without using mutual exclusion

An event counter is an integer

Three operations:

- ***read***(*E*): return the current value of event counter *E*
- ***advance***(*E*): increment *E* (atomically)
- ***await***(*E, v*): wait until $E \geq v$

Producer-Consumer example

```
#define N 4              // four slots in the buffer
event_counter in=0;      // number of items inserted into buffer
event_counter out=0;     // number of items removed from buffer

producer() {
    int item, sequence=0;
    for (;;) {
        produce_item(&item);      // produce something
        sequence++;               // item # of item produced
  ----> await(out, sequence-N);   // wait until there's room  (0≥-3), (0≥-2), …
        enter_item(item);         // put item in buffer
  ----> advance(&in);             // let consumer know there's one more item
    }
}
consumer() {
    int item, sequence=0;
    for (;;) {
        sequence++;               // item # we want to consume
  ----> await(in, sequence);      // wait until that item is present(0≥1)
        remove_item(item);        // get the item from the buffer
  ----> advance(&out);            // let producer know item's gone
        consume_item(item);       // consume it
    }
}
```


Producer-Consumer example

```
#define N 4              // four slots in the buffer
event_counter in=0;      // number of items inserted into buffer
event_counter out=0;     // number of items removed from buffer

producer() {
    int item, sequence=0;
    for (;;) {
        produce_item(&item);      // produce something
        sequence++;               // item # of item produced
  ----> await(out, sequence-N);   // wait until there's room (0≥-3), (0≥-2), …
        enter_item(item);         // put item in buffer
  ----> advance(&in);             // let consumer know there's one more item */
    }
}
```

Suppose the producer runs for a while and the consumer does not:

Iteration 1: out=0, sequence=1, *await*(0, 1-4): continue since $0 \geq -3 \Rightarrow$ in=1

Iteration 2: out=0, sequence=2, *await*(0, 2-4): continue since $0 \geq -2 \Rightarrow$ in=2

Iteration 3: out=0, sequence=3, *await*(0, 3-4): continue since $0 \geq -1 \Rightarrow$ in=3

Iteration 4: out=0, sequence=4, *await*(0, 4-4): continue since $0 \geq 0 \Rightarrow$ in=4

Iteration 5: out=0, sequence=5, *await*(0, 5-4): wait since $0 < 1$

Producer-Consumer example

```
#define N 4              // four slots in the buffer
event_counter in=0;      // number of items inserted into buffer
event_counter out=0;     // number of items removed from buffer

consumer() {
    int item, sequence=0;
    for (;;) {
        sequence++;               // item # we want to consume
  ----> await(in, sequence);      // wait until that item is present (0≥1)
        remove_item(item);        // get the item from the buffer
  ----> advance(&out);            // let producer know item's gone
        consume_item(item);       // consume it
    }
}
```

Suppose the consumer runs first:

Iteration 1: sequence = 1, *await*(0, 1) ⇒ sleep since $0 < 1$

When the producer runs its first iteration, it will increment *in*
The consumer's *await* will wake up since it's now *await*(1,1) and $1 \geq 1$

Condition Variables / Monitors

- Higher-level synchronization primitive
- Implemented by the programming language / APIs
- Two operations:
  - *wait* (*condition_variable*)
    - Block until *condition_variable* is "signaled"
  - *signal*(*condition_variable*)
    - Wake up **one** process that is waiting on the condition variable
    - Also called *notify*

Synchronization
Part II: Inter-Process Message Passing

Communicating processes

- Must:
  - Synchronize
  - Exchange data
- Message passing offers:
  - Data communication
  - Synchronization (via waiting for messages)
  - Works with processes on different machines

Message passing

- Two primitives:
  - *send*(destination, message)
  - *receive*(source, message)
- Operations may or may not be blocking

Producer-consumer example

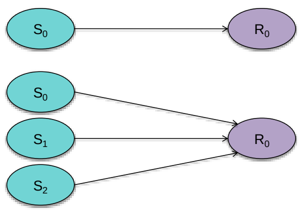
```
#define N 4       // number of slots in the buffer */

consumer() {
    int item, i;
    message m;

    for (i=0; i < N; ++i)
        send(producer, &m);  // send N empty messages
    for (;;) {
        receive(producer, &m);    // get a message with the item
        extract_item(&m, &item);  // take item out of message
        send(producer, &m);   // send an empty reply
        consume_item(item);   // consume it
    }
}
producer() {
    int item;
    message m;

    for (;;) {
        produce_item(&item);      // produce something
        receive(consumer, &m);        // wait for an empty message
        build_message(&m, item);  // construct the message
        send(consumer, &m);       // send it off
    }
}
```


Messaging: Rendezvous

- Sending process blocked until receive occurs
- Receive blocks until a send occurs
- Advantages:
  - No need for message buffering if on same system
  - Easy & efficient to implement
  - Allows for tight synchronization
- Disadvantage:
  - Forces sender & receiver to run in lockstep

Messaging: Direct Addressing

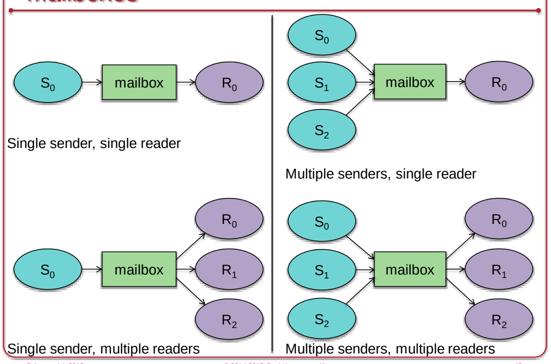
- Sending process identifies receiving process
- Receiving process can identify sending process
  - Or can receive it as a parameter



Messaging: Indirect Addressing

- Messages sent to an intermediary data structure of FIFO queues
- Each queue is a *mailbox*
- Simplifies multiple readers

Mailboxes



Other common IPC mechanisms

- Shared files
  - File locking allows concurrent access control
  - Mandatory or advisory
- Signal
  - A simple poke
- Pipe
  - Two-way data stream using file descriptors (but not names)
  - Need a common parent or threads in the same process
- Named pipe (FIFO file)
  - Like a pipe but opened like a file
- Shared memory

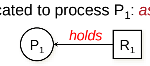
Conditions for deadlock

Four conditions must hold

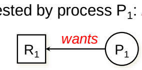
1. Mutual exclusion
   - Only one thread can access a critical section (resource) at a time
2. Hold and wait
   - A thread holds a resource but waits for another resource
3. Non-preemption of resources
   - Resources can only be released voluntarily
4. Circular wait
   - There is a cyclic dependency of threads waiting on resources

Deadlock

- Resource allocation
  - Resource R_1 is allocated to process P_1: *assignment edge*

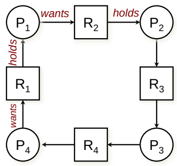


  - Resource R_1 is requested by process P_1: *request edge*



- Deadlock is present when the graph has cycles

Deadlock example



Circular dependency among four processes and four resources leads to deadlock

Dealing with deadlock

- Deadlock prevention
  - Ensure that at least one of the necessary conditions cannot hold
- Deadlock avoidance
  - Provide advance information to the OS on which resources a process will request.
  - OS can then decide if the process should wait
  - *But knowing which resources will be used (and when) is hard!* (impossible, really)
- Deadlock detection
  - Detect when a deadlock occurs and then deal with it
- Ignore the problem
  - Let the user deal with it (most common approach)

The End