Mutual Exclusion and Asynchronous Completion

Critical Section Problem
- Most common for multithreaded apps
- Updating object state
    - Updates to a single object
    - May be related updates to multiple objects

- Generally involves multi step operations
    - Object state inconsistent until operation finishes
    - Pre-emption compromises object or operation

- Correct operation requires mutual exclusion (one thread has access to the object)

Atomicity
- Before or After
    - A enter critical section before B starts
    - B enters critical section after A completes
    - No overlap in entering the critical section

- All or None
    - An update or operation that starts must complete or will be undone
    - An uncompleted update has no effect (gets removed)
    - No incomplete computations
    - Ex. If an operation requires updating 2 data structures, then wither both get updated or non get updated

- Correctness relies on both of these

Options for protecting critical sections
- Turn off interrupts (not practical)
- Avoid sharing data (no critical sections or only read only sections)
- Protect critical sections using hardware mutual exclusion
    - Trust CPU instructions are atomic
- Software locking

Avoiding Shared Data
- Don't share data we don't need to share (not always practical)
- May lead to inefficient resource use
- Sharing read only data

Atomic Instructions
- CPU instructions are uninterruptible
    - Read and modify write operations
    - Can be applied to contiguous bytes
    - Increment, decrement and binary operations

Locking

- Protect critical sections with a data structure
- Locks
    - Party with the lock can access the critical section
    - Parties not holding the lock cant access it
- A party needing to use the critical section tries to acquire the lock
- When finished with critical section, the lock is released

Software Locks

- ISA doesn't include instructions for building locks
- Individual instructions are serialized, while multiple instructions aren't
- Building locks in software leads to issues with mutual exclusion

Building Locks

- The operation of locking and unlocking a lock is also a critical section
    - Must be protected to avoid 2 threads getting the same lock

- Hardware assistance
    - Individual CPU instructions are atomic
    - We can implement a lock with one instruction (either thread which runs the instruction get the lock)

Lock Building Single Instructions

- Assembly level instruction
- Single instruction, therefore is atomic

Test and Set

```
bool TS( char *p) {
   bool rc;
   rc = *p;                    /* note the current value          */
   *p = TRUE;                  /* set the value to be TRUE        */
   return rc;                  /* return the value before we set it */
}

if !TS(flag) {
        /* We have control of the critical section! */
}
```

- Describes the testing set instructions which returns what the lock was before it was set
- Only when TS() gets called will the lock be set
- If False, we got the lock. If True, then we didn't get the lock (since the lock was True before the function call)

Compare and Swap

```
bool compare_and_swap( int *p, int old, int new ) {
  if (*p == old) {     /* see if value has been changed
      *p = new;        /* if not, set it to new value
      return( TRUE);   /* tell caller he succeeded
  } else               /* someone else changed *p
    return( FALSE);    /* tell caller he failed
}

if (compare_and_swap(flag,UNUSED,IN_USE) {
      /* I got the critical section! */
} else {
      /* I didn't get it.  */
}
```

- If they lock is free, set the lock to used and return True
- Else if the lock isn't free return False

Implementing Locks

```
bool getlock( lock *lockp) {
  if (TS(lockp) == 0 )
     return( TRUE);
  else
     return( FALSE);
}
void freelock( lock *lockp )
  *lockp = 0;
}
```

Lock Enforcement

- Locking resources works when:
    - Not possible to use a locked resource without the lock
    - Anyone who uses the resources follows locking rules

- If rules aren't followed, a thread could use the resource even when it doesn't hold the lock

Spin Locking

- Spin Locking: Trying to get the lock again, it could still not be available

  Advantages
    - Properly enforces access to critical sections
    - Assumes properly implemented locks

Disadvantages

- Wasteful: spinning uses processor cycles
- Likely to delay freeing of desired resource
    - Cycles burned could have been used by locking party to finish its work
- Bug could result in infinite spin wait

Asynchronous Completion Problem

- Parallel activities move at different speeds
- Problem: One activity may need to wait for another process to complete (so it can't run in its allocated time slice)
- Async completion problem deals with how to perform waits without reducing performance
    - Waiting process goes to sleep then is woken by the OS when the blocking operation is finished
- Examples of async completion
    - Waiting for I/O operation to complete
    - Waiting for a response to a network request

Spin Locking for Synchronization

1. When awaited operation is in parallel
    - Hardware device accepts a command
    - Another core releases a briefly held spin lock

2. When awaited operation is guaranteed to happen soon
    - Spinning is less expensive than sleeping the waking up

3. When spinning doesn't delay an awaited operation
    - Burns memory bandwidth and slows I/O
    - Burning CPu delays running another process

4. When the contention is rare
    - Multiple waiters increases the cost to run

Yield and Spin

- Check if your event occurred
- Maybe check a few more times, then yield and allow another process to run
- Cycle of checking and yielding

Problems

- Extra context switches
- Wastes cycles spinning each time
- Might not get scheduled to check until long after event occurs
- Works poorly with multiple waiters (could be unfair)

Fairness and Mutual Exclusion

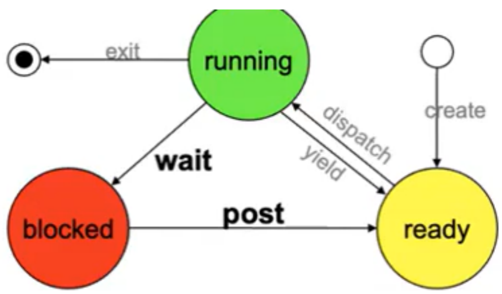
- Spin locking and yield and spin, but are unfair
- Both spin options require mutual exclusion

Completion Events

- If we can't get the lock, block
- Ask the OS to wake when the lock is available
- Similar for anything else you need to wait for
    - I/O completion
    - Waiting for another process to finish
- Implemented with condition variables

Condition Variables

- Create a synchronization object associated with a resource or request
- Requester blocks and is queued awaiting event on that object
- Posting event to object unblocks the waiter



Condition Variables and the OS

- Generally the OS provides condition variables (library code with threads also has this)
- Block a process or thread when a condition variable is used
- It observers when the desired event occurs
- Unblocks the blocked process or thread
    - Places it back in the ready queue

Handling Multiple Waits

- Threads can wait on several different conditions
- Pointless to wake up everyone on every event
    - Each should only wake when their event happens
- OS should allow easy selection of the right condition
- Several threads could also be waiting for the same condition

Waiting LIst

- Each completion event needs an associated waiting list
- When an event occurs, consult list to determine who's waiting for that event
    - Order in which waiting list is notified depends on the event and application

Alerting Threads

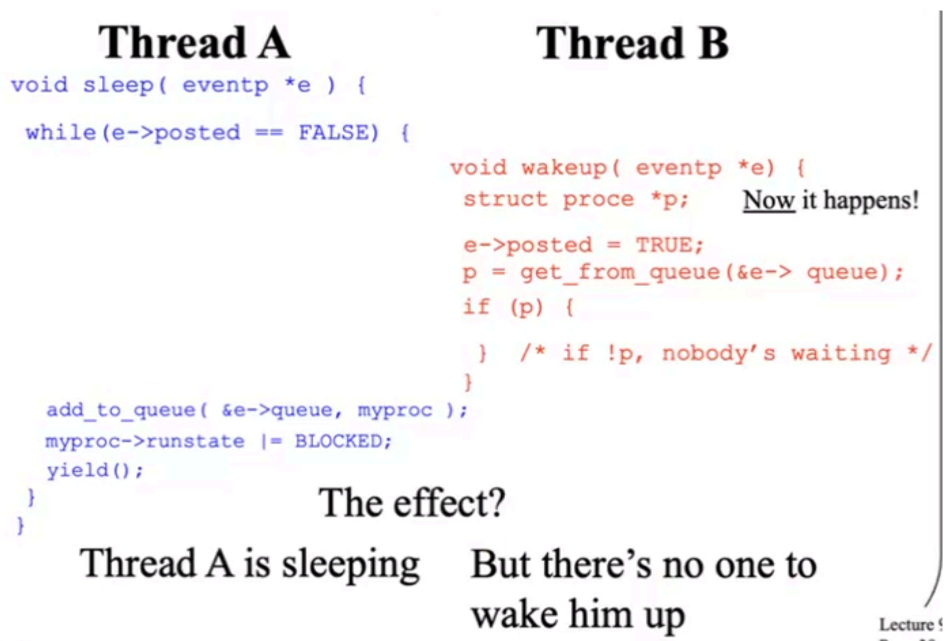
- Pthread_cond_wait: at least one blocked thread
- Pthread_cond_broadcast: all blocked threads
- Broadcast approach could be wasteful (unbound waiting times)
- Waiting queue solves these problems (a post wakes the first client in the queue)

Locking and Waiting Lists

- Spinning is a bad thing, locks need waiting lists
- Waiting list is a shared DS
    - Will have critical sections which are protected by other licks
- Potential circular dependency

Sleep/wakeup race condition

- Thread B has locked a resource and thread A needs to get that lock
- Thread A will call sleep() to wait for the lock to be free
- Thread B finishes using the resource
    - Thread B calls wakeup() to release the lock
- Currently no other threads waiting, just A
- However, if we never reached the code to add A to the waiting list due to a context switch for B
- Code to add A to the waiting list is reached, and A gets blocked with no one to wake up A
- The lock is free, but A is asleep



Sleep() problem

- Critical section in sleep()
  - Starting before testing the posted flag
  - Ending after adding us to the waiting list and block

- During this section, we need to prevent:
  - Wakeup of the event
  - Other threads waiting on the event

- Mutual exclusion problem
  - Need a lock to control the sleep()

Summary

- Mutual Exclusion
  - Allows only one of several things to occur at once

- Asynchronous completion
  - Properly synchronize cooperating events

- Locks are one way to assure mutual exclusion

- Spinning and completion events are ways to handle asynchronous competitions