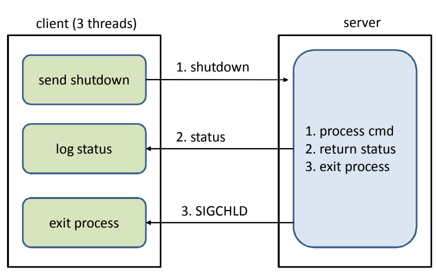
Mutual Exclusion, Async Completions

Why We Wait

- We await completion of non-trivial operations
  - data to be read from disk
  - a child process to be created
- We wait for important events
  - a request/notification from another process
  - an out-of-band error that must be handled
- We wait to ensure correct ordering
  - B cannot be performed until A has completed
  - if A precedes B, B must see the results of A

Correct Ordering



"Surely the final status message will be received and processed before the SIGCHLD causes the client to shut down! "

Problem 2: asynchronous completion

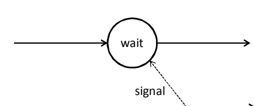
- most procedure calls are synchronous
  - we call them, they do their job, they return
  - when the call returns, the result is ready
- many operations cannot happen immediately
  - waiting for a held lock to be released
  - waiting for an I/O operation to complete
  - waiting for a response to a network request
  - delaying execution for a fixed period of time
- we call such completions <u>asynchronous</u>

Approaches to Waiting

- spinning … "busy waiting"
  - works well if event is independent and prompt
  - wasted CPU, memory, bus bandwidth
  - may actually delay the desired event
- yield and spin … "are we there yet?"
  - allows other processes access to CPU
  - wasted process dispatches
  - works very poorly for multiple waiters
- either may still require mutual exclusion

Condition Variables

- create a synchronization object
  - associate that object with a resource or request
  - requester blocks awaiting event on that object
  - upon completion, the event is "posted"
  - posting event to object unblocks the waiter



Awaiting Asynchronous Events

```
pthread_mutex_t lock = PTHEAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHEAD_COND_INITIALIZER;
…
pthread_mutex_lock(&lock);
while (ready == 0)
    pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock)
…
                if (pthread_mutex_lock(&lock)) {
                        ready = 1;
                        pthread_cond_signal(&cond);
                        pthread_mutex_unlock(&lock);
                }
```


The Mutual Exclusion Challenge

- We cannot prevent parallelism
  - it is fundamental to our technology
- We cannot eliminate all shared resources
  - increasingly important to ever more applications
- What we can do is ...
  - identify the at risk resources, and risk scenarios
  - design those classes to enable protection
  - identify all of the critical sections
  - ensure each is correctly protected (case by case)

Evaluating Mutual Exclusion

- Effectiveness/Correctness
  - ensures before-or-after atomicity
- Fairness
  - no starvation (un-bounded waits)
- Progress
  - no client should wait for an available resource
  - susceptibility to convoy formation, deadlock
- Performance
  - delay, instructions, CPU load, bus load
  - in contended and un-contended scenarios

Approaches

- Avoid shared mutable resources
  - the best choice … if it is an option
- Interrupt Disables
  - a good tool with limited applicability
- Spin Locks
  - very limited applicability
- Atomic Instructions
  - very powerful, but difficult w/limited applicability
- Mutexes
  - higher level, broad applicability

What Happens During an Interrupt?

- Interrupt controller requests CPU for service
- CPU stops the executing program
- Interrupt vector table is consulted
  - PC/PS of Interrupt Service Routine (ISR)
- ISR handles the interrupt (just like a trap)
  - save regs, find/call 2nd level handler, restore regs
- Upon return, CPU state is restored
  - code resumes w/no clue it was interrupted

Approach: Interrupt Disables

- temporarily block some or all interrupts
  - can be done with a privileged instruction
  - side-effect of loading new Processor Status
- abilities
  - prevent Time-Slice End (timer interrupts)
  - prevent re-entry of device driver code
- dangers
  - may delay important operations
  - a bug may leave them permanently disabled

Preventing Preemption

```
DLL_insert(DLL *head, DLL*element) {
    int save = disableInterrupts();
    DLL *last = head->prev;
    element->prev = last;
    element->next = head;
    last->next = element;
    head->prev = element;

}
```

```
DLL_insert(DLL *head, DLL*element) {
    DLL *last = head->prev;
    element->prev = last;
    element->next = head;
    last->next = element;
    head->prev = element;
}
```

```
    restoreInterrupts(save);
```


Preventing Driver Reentrancy

```
zz_io_startup( struct iorq *bp ) {
    …
    save = intr_enable( ZZ_DISABLE );

    /* program the DMA request */
    zzSetReg(ZZ_R_ADDR, bp->buffer_start );
    zzSetReg(ZZ_R_LEN, bp->buffer_length);
    zzSetReg(ZZ_R_BLOCK, bp->blocknum);
    zzSetReg(ZZ_R_CMD, bp->write?
        ZZ_C_WRITE : ZZ_C_READ );
    zzSetReg(ZZ_R_CTRL, ZZ_INTR+ZZ_GO);

    /* reenable interrupts    */
    intr_enable( save );
```

```
zz_intr_handler() {
    …
    /* update data read count */
    resid = zzGetReg(ZZ_R_LEN);

    /* turn off device ability to interrupt */
    zzSetReg(ZZ_R_CTRL, ZZ_NOINTR);
    …
```

Serious consequences could result if the interrupt handler was called while we were half-way through programming the DMA operation.

Preventing Driver Reentrancy

- interrupts are usually self-disabling
  - CPU may not deliver #2 until #1 is *acknowledged*
  - interrupt vector PS usually disables causing intr
- they are restored after servicing is complete
  - ISR may explicitly *acknowledge* the interrupt
  - return from ISR will restore previous (enabled) PS
- drivers usually disable during critical sections
  - updating registers used by interrupt handlers
  - updating resources used by interrupt handlers

Interrupts and Resource Allocation

```
…
lock(event_list);
add_to_queue(event_list, my_proc);
unlock(event_list);
yield();
…
```

```
xx_interrupt:
    …
    lock(event_list);
    post(event_list);
    return;
```


Interrupts and Resource Allocation

- interrupt handlers are not allowed to block
  - only a scheduled process/thread can block
  - interrupts are disabled until call completes
- ideally they should never need to wait
  - needed resources are already allocated
  - operations implemented w/lock-free code
- brief spins may be acceptable
  - wait for hardware to acknowledge a command
  - wait for a co-processor to release a lock

Evaluating Interrupt Disables

- Effectiveness/Correctness
  - ineffective against MP/device parallelism
  - only usable by kernel mode code
- Progress
  - deadlock risk (if ISR can block for resources)
- Fairness
  - pretty good (assuming disables are brief)
- Performance
  - one instruction, much cheaper than system call
  - long disables may impact system performance

Approach: Spin Locks

- loop until lock is obtained
  - usually done with atomic test-and-set operation
- abilities
  - prevent parallel execution
  - wait for a lock to be released
- dangers
  - likely to delay freeing of desired resource
  - bug may lead to infinite spin-waits

Atomic Instructions

- atomic read/modify/write operations
  - implemented by the memory bus
  - effective w/multi-processor or device conflicts
- ordinary user-mode instructions
  - may be supported by libraries or even compiler
  - limited to a few (e.g. 1-8) contiguous bytes
- very expensive (e.g. 20-100x) instructions
  - wait for all cores to write affected cache-line
  - force all cores to drop affected cache-line

Atomic Instructions – Test & Set

```
/*
 * Concept: Atomic Test-and-Set
 *       this is implemented in hardware, not code
 */
int TestAndSet( int *ptr, int new) {
    int old = *ptr;
    *ptr = new;
    return( old );
}
```


Spin Locks

```
DLL_insert(DLL *head, DLL*element) {
    while(TestAndSet(lock,1) == 1);
        DLL *last = head->prev;
        element->prev = last;
        element->next = head;
        last->next = element;
        head->prev = element;
    lock = 0;
}
```


What If You Don't Get the Lock?

- give up?
  - but you can't enter your critical section
- try again?
  - OK if we expect it to be released <u>very</u> soon
- what if another process has to free the lock?
  - spinning keeps that process from running
- what lock release will take a long time?
  - we are burning a lot of CPU w/useless spins

Evaluating Spin Locks

- Effectiveness/Correctness
  - effective against preemption and MP parallelism
  - ineffective against conflicting I/O access
- Progress
  - deadlock danger in ISRs, convoy formation
- Fairness
  - possible unbounded waits
- Performance
  - waiting is extremely expensive (CPU, bus, mem)

Which One Should We Use?

- all of them
  - they solve different problems
- atomic instructions
  - prevent conflicting parallel updates
- interrupt disables
  - prevent device driver reentrancy
  - prevent scheduling preemption
- spinning
  - await imminent events from parallel sources

Asynchronous Completions

- Synchronous operations
  - you call a subroutine
  - it does what you need, and returns promptly
- Asynchronous operations/completions
  - will happen at some future time
    - when an I/O operation completes
    - when a lock is released
  - how do we block to await some future event?
- spin-locks combine lock and await
  - good at locking, not so good at waiting

Spinning Sometimes Makes Sense

1. awaited operation proceeds in parallel
   - a hardware device accepts a command
   - another CPU releases a briefly held spin-lock
2. awaited operation guaranteed to be soon
   - spinning is less expensive than sleep/wakeup
3. spinning does not delay awaited operation
   - burning CPU delays running another process
   - burning memory bandwidth slows I/O
4. contention is expected to be rare
   - multiple waiters greatly increase the cost

The Classic "spin-wait"

```
/* set a specified register in the ZZ controller to a specified value        */
zzSetReg( struct zzcontrol *dp, short reg, long value ) {
            while( (dp->zz_status & ZZ_CMD_READY) == 0);
                        /* it may take a few ns to process the last set      */
            dp->zz_value = value;
            dp->zz_reg = reg;
            dp->zz_cmd = ZZ_SET_REG;
}

/* program the ZZ for a specified DMA read or write operation              */
zzStartIO( struct zzcontrol *dp, struct ioreq *bp ) {
            zzSetReg(dp, ZZ_R_ADDR, bp->buffer_start);
            zzSetReg(dp, ZZ_R_LEN, bp->buffer_length);
            zzSetReg(dp, ZZ_R_CMD, bp->write ? ZZ_C_WRITE : ZZ_C_READ );
            zzSetReg(dp, ZZ_R_CTRL, ZZ_INTR + ZZ_GO);
}
```


Correct Completion

- Correctness
  - no lost wake-ups
- Progress
  - if event has happened, process should not block
- Fairness
  - no un-bounded waiting times
- Performance
  - cost of waiting
  - promptness of resuming
  - minimal spurious wake-ups

Spinning and Yielding

- yielding is a good thing
  - avoids burning cycles busy-waiting
  - gives other tasks an opportunity to run
- spinning and yielding is not so good
  - which process runs next is random
  - when yielder next runs is random
- Progress: potentially un-bounded wait times
- Performance: each try is wasted cycles

Sleep/Wakeup Operations

- sleep (e.g. pthread_cond_wait)
  - block caller until condition has been posted
- wakeup (e.g. pthread_cond_signal)
  - post condition and awaken blocked waiter(s)
- potential problems:
  - race conditions between sleep and wakeup
    - wakeup called before (or during) sleep
  - *spurious* wakeups
    - woken up, but event is not (currently) available

Race: wakeup called before sleep

- model #1 (e.g. pthread condition variables)
  - purely a signaling mechanism
  - client responsible for checking condition

```
pthread_mutex_lock(&mutex);
while( !condition )
  pthread_cond_wait(&condvar, &mutex);
pthread_mutex_unlock(&mutex);
```

- model #2 (e.g. semaphores)
  - return guarantees condition has been satisfied
  - if condition already satisfied, caller will not block

Spurious Wakeups

- waking up does not mean condition satisfied
  - perhaps multiple processes were woken up
  - perhaps you were woken up for another reason
  - perhaps another process got to resource first
- check/sleep should be done in a loop
  - after each wakeup, check condition again
- spurious wakeups are a minor cost/irritation
- lost wakeups are a serious problem

Evaluating pthread_cond_signal/wait

- Effectiveness/Correctness
  - good (if used properly)
- Progress
  - good (if used properly)
- Fairness
  - who gets resource is random
- Performance
  - good for single consumers
  - potential spurious wakeups w/more consumers

Waiting Lists

- Who wakes up when a CV is signaled
  - pthread_cond_wait … at least one blocked thread
  - pthread_cond_broadcast … all blocked threads
- this may be wasteful
  - if the event can only be consumed once
  - many processes wake and try, most will fail
  - potentially unbounded waiting times
- a waiting queue would solve these problems
  - post wakes up one (first, highest priority) client

Progress vs. Fairness

- consider …
  - P1: lock(), park()
  - P2: unlock(), unpark()
  - P3: lock()
- progress says:
  - it is available, P3 gets it
  - spurious wakeup of P1
- fairness says:
  - FIFO, P3 gets in line
  - and a convoy forms

```
void lock(lock_t *m) {
    while(true) {
        while (TestAndSet(&m->guard, 1) == 1);
        if (!m->locked) {
            m->locked = 1;
            m->guard = 0;
            return;
        }
        queue_add(m->q, me);
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    m->locked = 0;
    if (!queue_empty(m->q))
        unpark(queue_remove(m->q);
    m->guard = 0;
}
```

Evaluating Sleep w/Waiting Lists

- Effectiveness/Correctness
  - good (if used properly)
- Progress
  - good … if we allow cutting in line
- Fairness
  - good … unless we allow cutting in line
- Performance
  - good (with few spurious wakeups)

Locking and Waiting Lists

- Spinning for a lock is usually a bad thing
  - locks should probably have waiting lists
- a waiting list is a (shared) data structure
  - implementation will likely have critical sections
  - which may need to be protected by a lock
- This seems to be a circular dependency
  - locks have waiting lists
  - which must be protected by locks
  - what if we must wait for the waiting list lock?

Race Condition within Sleep

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    if (!m->locked) {
        m->locked = 1;
        m->guard = 0;
    } else {
        queue_add(m->q, me);
        m->guard = 0;
        park();
    }
}
```

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    if (queue_empty(m->q))
        m->locked = 0;
    else
        unpark(queue_remove(m->q));
    m->guard = 0;
}
```


(sleep/wakeup races)

- possibility of long spins or deadlock
  - interrupt comes in while guard is held
  - ISR tries to wake-up the waiting list
- possibility of missed wakeup
  - wakeup is sent before blockee can sleep
  - blockee sleeps, having missed the wakeup
- solutions (may require OS assistance)
  - disable preemption in <u>this</u> critical section

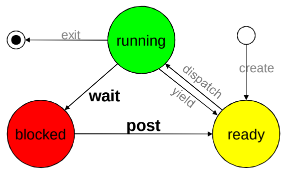
Assignments

- Projects
  - finish Edison bring-up
- Reading
  - AD 29 (protecting data)
  - AD30.2-3 (producer/consumer problems)
  - AD 31 (semaphores)
  - flock(2), lockf(2)

Supplementary Slides

Blocking and Unblocking

- blocking
  - remove specified process from the "ready" queue
  - yield the CPU (let scheduler run someone else)
- unblocking
  - return specified process to the "ready" queue
  - inform scheduler of wakeup (possible preemption)



Synchronization Objects

- combine exclusion and (optional) waiting
- operations implemented safely
  - with atomic instructions
  - with interrupt disables
- exclusion policies (one-only, read-write)
- waiting policies (FCFS, priority, all-at-once)
- additional operations (queue length, revoke)

Spin Waiting For Asynchronous Completions

- Wastes CPU, memory, bus bandwidth
  - Each path through the loop costs instructions
- May actually delay the desired event
  - One of your cores is busy spinning
  - Maybe it could be doing the work required to complete the event instead
  - But it’s spinning . . .

Spin-Waits Revisited

- spin-waits await asynchronous completions
  - but they do so by busy-waiting

    while (event_not_ready) ;
- sleep/wake-up is almost always better
  - fewer wasted cycles and faster response
  - these are software completion mechanisms and there are hardware-related situations where they don't work (or don't make sense)
- there are cases where it makes sense to spin
  - very briefly for events originating outside our CPU

Spin-waits: when to use them

- when the event does not come from our CPU
  - so spinning will not delay the completion
- and waiting time guaranteed to be very brief
  - fewer cycles than would be required to go to sleep
- examples:
  - waiting a few μ-seconds for hardware to come ready
    - **IF** it is guaranteed to be come back promptly
  - waiting for another CPU to release a lock
    - **IF** critical section is very short (e.g. 1 digit # of instructions)
    - **IF** interrupts are disabled so preemption is impossible
- almost never appropriate in user-mode code