Operating Systems Principles

Mutual Exclusion, Asynchronous Completion

Mark Kampe
(markk@cs.ucla.edu)

Mutual Exclusion, Asynchronous Completion

Obstacles to Atomic Execution

- Blocking
  - thread requests a resource in the critical section
- Scheduling Preemption
  - thread experiences time-slice-end
- Shared Memory Multi-Processor
  - shared resources between cores or CPUs
- I/O Devices
  - program and device accessing same memory
  - program and ISR accessing same resources

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

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);
```

Mutual Exclusion and Asynchronous Completion 7

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.

Mutual Exclusion and Asynchronous Completion 8

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

Mutual Exclusion and Asynchronous Completion 9

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;
```

Mutual Exclusion and Asynchronous Completion 10

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

Mutual Exclusion and Asynchronous Completion 11

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

Mutual Exclusion and Asynchronous Completion 12

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
  - not available with (slower) I/O bus operations
- ordinary user-mode instructions
  - may be supported by libraries or even compiler
- 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;
}
```


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 can be extremely expensive (CPU, bus)

Approach: Lock-Free Operations

- MT safe data structures and operations
  - an alternative to mutual-exclusion
- abilities
  - single reader/writer w/ordinary instructions
  - multi-reader/writer w/atomic instructions
  - all-or-none and before-or-after semantics
- limitations
  - unusable for complex critical sections
  - unusable as a waiting mechanism

Atomic Instructions – Compare & Swap

```
/*
 * Concept: Atomic Compare and Swap
 *      this is implemented in hardware, not code
 */
int CompareAndSwap( int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return( actual );
}
```


Lock-Free Multi-Writer

```
// push an element on to a singly linked LIFO list
void SLL_push(SLL *head, SLL *element) {
    do {
        SLL *prev = head->next;
        element->next = prev;
    } while ( CompareAndSwap(&head->next, prev, element) != prev);
}
```


Lock-Free Single Reader/Writer

```
int SPSC_put(SPSC *fifo, unsigned char c) {
    if (SPSC_bytesIn(fifo) == fifo->full)
        return(-1);
    *(fifo->write) = c;
    if (fifo->write == fifo->wrap)
        fifo->write = fifo->start;
    else
        fifo->write++;
    return( c );
}
```

```
int SPSC_get(SPSC *fifo) {
    if (SPSC_bytesIn(fifo) == 0)
        return(-1);
    int ret = *(fifo->read);
    if (fifo->read == fifo->wrap)
        fifo->read = fifo->start;
    else
        fifo->read++;
    return(ret);
}
```

```
int SPSC_bytesIn(SPSC *fifo) {
    return(fifo->write >= fifo->read ?
        fifo->write – fifo->read :
        fifo->full – (fifo->read – fifo->write));
}
```


Evaluating Lock-Free Operations

- Effectiveness/Correctness
  - effective against all conflicting updates
  - cannot be used for complex critical sections
- Progress
  - no possibility of deadlock or convoy
- Fairness
  - small possibility of brief spins
- Performance
  - expensive instructions, but cheaper than syscalls

Spin Locks vs Atomic Update Loops

- both involve spinning on an atomic update
- a spin-lock
  - spins until the lock is released
  - which could take a very long time
- an atomic update loop
  - spins until there is no conflict during the update
  - conflicting updates are actually very rare
- comparable for <u>very</u> brief critical sections
  - e.g. a one-digit number of instructions

Spin Locks vs Atomic Updates

```
void SLL_push(SLL *head, SLL *element) {
    do {
        SLL *prev = head->next;
        element->next = prev;
    } while ( CompareAndSwap(&head->next, prev,  element) != prev);
}
```

```
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;
}
```

Locking comes in many flavors

- lock and wait
  - block until resource becomes available
- non-blocking
  - return an error if resource is unavailable
- timed wait
  - block a specified maximum time, then fail
- spin and wait (futex)
  - spin briefly, and then join a waiting list
- strict FIFO

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

Who to Wake-Up - Waiting Lists

- random yielding and polling is foolish
  - all waiters should block
  - each should wake up when his event happens
- this suggests all events need a waiting list
  - when posting an event, look up who to awaken
    - wake up everyone on the list?
    - one-at-a-time in FIFO order?
    - one-at-a-time in priority order (possible starvation)?
  - choice depends on event and application

Evaluating Waiting Lists

- Effectiveness/Correctness
  - should be very good
- Progress
  - there is a trade-off involving *cutting* in line
- Fairness
  - should be very good
- Performance
  - should be very efficient
  - depends on frequency of 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?

Sleep/Wakeup Races

```
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 then blockee sleeps
- solutions (may require OS assistance)
  - interrupts should be disabled in this crit section
  - hyper-awake state prevents the next sleep

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;
}
```

assignments

- reading for the next lecture
  - Arpaci ch 29 … Locked Data Structures
  - Arpaci ch 30 … Condition Variables
  - Arpaci ch 31 … Semaphores
  - flock(2) … Posix file locking
  - lockf(3) … ranged file locks