Mutual Exclusion

- 7B Critical Sections and Atomicity
- 7D Mutual Exclusion
- 7E Implementing Mutual Exclusion

Synchronization - evolution of problem

- batch processing - serially reusable resources
  - process A has tape drive, process B must wait
  - process A updates file, then process B reads it
- cooperating processes
  - exchanging messages with one-another
  - continuous updates against shared files
- shared data and multi-threaded computation
  - interrupt handlers, symmetric multi-processors
  - parallel algorithms, preemptive scheduling
- WAN-scale distributed computing

The benefits of parallelism

- improved throughput
  - blocking of one activity does not stop others
- improved modularity
  - separating complex activities into simpler pieces
- improved robustness
  - the failure of one thread does not stop others
- a better fit to emerging paradigms
  - client server computing, web based services
  - our universe is cooperating parallel processes

What's the big deal?

- sequential program execution is easy
  - first instruction one, then instruction two, ...
  - execution order is obvious and deterministic
- independent parallel programs are easy
  - if the parallel streams do not interact in any way
- cooperating parallel programs are hard
  - if the two execution streams are not synchronized
    - results depend on the order of instruction execution
    - parallelism makes execution order non-deterministic
    - interactions become intractable (transitive closure of all possible execution orders for all threads)

Race Conditions

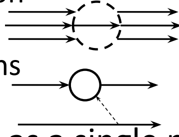
- shared resources and parallel operations
  - where outcome depends on execution order
  - these happen all the time, most don’t matter
- some race conditions affect correctness
  - conflicting updates (mutual exclusion)
  - check/act races (sleep/wakeup problem)
  - multi-object updates (all-or-none transactions)
  - distributed decisions based on inconsistent views
- each of these classes can be managed
  - if we recognize the race condition and danger

Non-Deterministic Execution

- processes block for I/O or resources
- time-slice end preemption
- interrupt service routines
- unsynchronized execution on another core
- queuing delays
- time required to perform I/O operations
- message transmission/delivery time

What is "Synchronization"

- true parallelism is imponderable
  - pseudo-parallelism may be good enough
    - identify and serialize key points of interaction
- there are two interdependent problems
  - critical section serialization
  - asynchronous completions



- they are often discussed as a single problem
  - many mechanisms simultaneously solve both
  - solution of either requires solution to the other

A Synchronization Problem
(multi-thread, shared memory, circular buffer)

```
write(buf, toSend):
    while toSend > 0
        wait(nextWrite < endOfBuffer)
        free = endOfBuffer – nextWrite
        count = min(free, toSend)
        copy(buf, nextWrite, count)
        nextWrite += count
        toSend -= count;
```

```
read(buf, desired):
    while desired > 0
        wait (nextWrite > lastRead)
        avail = nextWrite – lastRead
        count = min(avail, desired)
        copy(lastRead, buf, count)
        lastRead += count
        if lastRead == endOfBuffer
            lastRead = startOfBuffer
            nextWrite =
          startOfBuffer
        desired -= count
```

Critical Section   Await Event   Signal Event

Problem 1: Critical Sections

- a resource shared by multiple threads
  - multiple concurrent threads, processes or CPUs
  - interrupted code and interrupt handler
- use of the resource changes its state
  - contents, properties, relation to other resources
  - updates are non-atomic (or non-global)
- correctness depends on execution order
  - when scheduler runs/preempts which threads
  - true (e.g. multi-processor) parallelism
  - relative timing of independent events
  - leading to "*indeterminate*" results

Reentrant & MT-safe code

- consider a simple recursive routine:
  `int factorial(x) { tmp = factorial( x-1 ); return x*tmp}`
- consider a possibly multi-threaded routine:
  `void debit(amt) {tmp = bal-amt; if (tmp >=0) bal = tmp)}`
- neither would work if tmp was shared/static
  - must be dynamic, each invocation has own copy
  - this is not a problem with read-only information
- some variables must be shared
  - and proper sharing often involves critical sections

Critical Section - updating a file

Process #1

```
remove( "database");
fd = create( "database" );
write(fd, newdata, length);
close(fd);
```

Process #2

```
fd = open( "database", READ);
count = read(fd, buffer, length);
...
```


What could go wrong with an add?

thread #1	thread #2
counter = counter + 1;	counter = counter + 1;
mov counter, %eax	
add $0x1, %eax	
	mov counter, %eax
	add $0x1, %eax
	mov %eax, counter
mov %eax, counter	

Achieving Mutual Exclusion

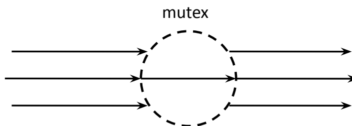
```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
…
if (pthread_mutex_lock(&lock) == 0) {
    counter = counter + 1;
    pthread_mutex_unlock(&lock);
}
```


Recognizing Critical Sections

- generally involves updates to object state
  - may be updates to a single object
  - may be related updates to multiple objects
- generally involves multi-step operations
  - object state inconsistent until operation finishes
  - preemption compromises object or operation
- correct operation requires mutual exclusion
  - only one thread at a time has access to object(s)
  - client 1 completes before client 2 starts

Atomicity … two different types

- <u>Before or After</u> (mutual exclusion)
  - A enters critical section before B starts
  - A enters critical section after B completes
- <u>All or None</u> (atomic transactions)
  - an update that starts will complete w/o interruption
  - an uncompleted update has no effect



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 serialized (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 deadlock, convoy formation
- Performance
  - delay, throughput, instructions, memory 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
  - expensive approach with very limited applicability
- Atomic Instructions
  - very powerful, 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
  - may decide to preempt the interrupted process
- 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;
    restoreInterrupts(save);
}
```

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

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 Deadlocks

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

unlock(event_list);
yield();
…
```

```
xx_interrupt:
    …
    lock(event_list);
    post(event_list);
    unlock(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 ISR completes
- ideally they should never need to wait
  - needed resources are already allocated
  - operations implemented w/lock-free code
- brief spins may be acceptable
  - if prompt completion is guaranteed
    - 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
  - not usable by user-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 very 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
- Fairness
  - possible unbounded waits
- Performance
  - waiting is extremely expensive (CPU, bus, mem)

Which One is Best?

- all/none of them
  - they solve different problems
- atomic instructions
  - atomic updates to small scalar data items
- interrupt disables
  - prevent device driver reentrancy
  - prevent scheduling preemption
- spinning
  - await imminent events from parallel sources

Queues rather than Spinning

```
void lock(lock_t *m) {
	while(TestAndSet(&m->guard, 1) == 1);
	if (m->flag == 0) {
		m->flag = 1;		// take the lock
		m->guard = 0;
	} else {
		queue_add(mp_q, gettid());
		m->guard = 0;
		park();		// block this process
	}
}
```

```
void unlock(lock_t *m) {
	while(TestAndSet(&m->guard, 1) == 1);
	if (queue_empty(m->q))
		m->flag = 0;		// release lock
	else
		// unblock next proc on queue
		unpark(queue_remove(m->queue));
		// leave lock held for that process
	m->guard = 0;
}
```

(Arpacci-Dusseau Figure 28.9)

Reading and Assignments

Reading:

- Arpaci C29 … protecting data
- flock(2) system call, lockf(3) library routine

Projects:

- we will help w/project 2A problems in the lab

Supplementary Slides

Synchronization – a Parable

- consider the “Garden of Eden”
  - we were warned of the “knowledge of good and evil”
    - which would forever end our innocent lives in paradise
  - but we were then tempted to eat that fruit
    - it would open our eyes, and give us God-like knowledge
  - we ate, lost our innocence, were banished from paradise
- consider the “Garden of computation”
  - we were warned about cooperating parallel processes
    - which would forever cost us our innocence
    - but we also saw the power that knowledge could unleash
  - again we took the bait, and lost our innocence
    - programming, once simple, has been made impossibly difficult