Deadlock Prevention and Avoidance

7L. Higher level synchronization
7M. Lock-Free operations
8A. Deadlock Overview
8B. Deadlock Avoidance
8C. Deadlock Prevention
8D. Monitoring and Recovery
8E. Priority Inversion

Synchronization is Difficult

- recognizing potential critical sections
  - potential combinations of events
  - interactions with other pieces of code
- choosing the mutual exclusion method
  - there are many different mechanisms
  - with different costs, benefits, weaknesses
- correctly implementing the strategy
  - correct code, in all of the required places
  - maintainers may not understand the rules

We need a “Magic Bullet”

- We identify shared resources
  - objects whose methods may require serialization
- We write code to operate on those objects
  - just write the code
  - assume all critical sections will be serialized
- Complier generates the serialization
  - automatically generated locks and releases
  - using appropriate mechanisms
  - correct code in all required places

Monitors – Protected Classes

- each monitor class has a semaphore
  - automatically acquired on method invocation
  - automatically released on method return
  - automatically released/acquired around CV waits
- good encapsulation
  - developers need not identify critical sections
  - clients need not be concerned with locking
  - protection is completely automatic
- high confidence of adequate protection

Monitors: use

```
monitor CheckBook {
        // class is locked when any method is invoked
        private int balance;
        public int balance() {
                return(balance);
        }
        public int debit(int amount) {
                balance -= amount;
                return( balance)
        }
}
```

Evaluating: Monitors

- correctness
  - complete mutual exclusion is assured
- fairness
  - semaphore queue prevents starvation
- progress
  - inter-class dependencies can cause deadlocks
- performance
  - coarse grained locking is not scalable

Java Synchronized Methods

- each object has an associated mutex
  - acquired before calling a synchronized method
  - nested calls (by same thread) do not reacquire
  - automatically released upon final return
- static synchronized methods lock class mutex
- advantages
  - finer lock granularity, reduced deadlock risk
- costs
  - developer must identify serialized methods

Java Synchronized: use

```
class CheckBook {
        private int balance;
        public int balance() {
                return(balance);
        }
        // object is locked when this method is invoked
        public synchronized int debit(int amount) {
                balance -= amount;
                return( balance)
        }
}
```

Evaluating Java Synchronized Methods

- correctness
  - correct if developer chose the right methods
- fairness
  - priority thread scheduling (potential starvation)
- progress
  - safe from single thread deadlocks
- performance
  - fine grained (per object) locking
  - selecting which methods to synchronize

Encapsulated Locking

- opaquely encapsulate implementation details
  - make class easier to use for clients
  - preserve the freedom to change it later
- locking is entirely internal to class
  - search/update races within the methods
  - critical sections involve only class resources
  - critical sections do not span multiple operations
  - no possible interactions with external resources

Client Locking

- Class cannot correctly synchronize all uses
- critical section spans multiple class operations
  - updates in a higher level transaction
- client-dependent synchronization needs
  - locking needs depend on how object is used
  - client may control access to protected objects
  - client may select best serialization method
- potential interactions with other resources
  - deadlock prevention must be at higher level

Non-Blocking 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));
}
```

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 );
}
```

Solving the checkbook problem

```
int current_balance;
Writecheck( int amount ) {
  int oldbal, newbal;
  do {
    oldbal = current_balance;
    newbal = oldbal - amount;
    if (newbal <0) return (ERROR);
  } while (!compare_and_swap( &current_balance, oldbal, newbal))
...
}
```

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);
}
```

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;
}
```

(Spin Locks vs Atomic Update Loops)

- both involve spinning on an atomic update
  - but they are not the same
- 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
  - impossible to be preempted holding lock
  - conflicting updates are actually very rare

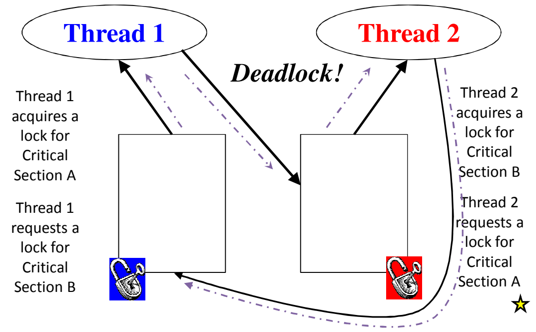
Evaluating Lock-Free Operations

- 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

What is a Deadlock?

- Two (or more) processes or threads
  - cannot complete without all required resources
  - each holds a resource the other needs
- No progress is possible
  - each is blocked, waiting for another to complete
- Related problem: livelock
  - processes not blocked, but cannot complete
- Related problem: priority inversion
  - high priority actor blocked by low priority actor

Resource Dependency Graph



Thread 1 acquires a lock for Critical Section A

Thread 1 requests a lock for Critical Section B

Thread 2 acquires a lock for Critical Section B

Thread 2 requests a lock for Critical Section A

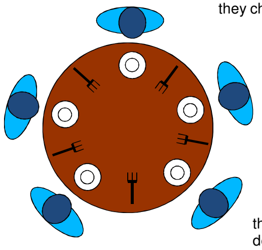
Why Study Deadlocks?

- A major peril in cooperating parallel processes
  - they are relatively common in complex applications
  - they result in catastrophic system failures
- Finding them through debugging is very difficult
  - they happen intermittently and are hard to diagnose
  - they are much easier to prevent at design time
- Once you understand them, you can avoid them
  - most deadlocks result from careless/ignorant design
  - an ounce of prevention is worth a pound of cure

The Dining Philosophers Problem

Five philosophers
five plates of pasta
five forks

they eat whenever
they choose to



one requires two forks to eat pasta, but must take them one at a time

they will not negotiate with one-another

the problem demands an <u>absolute</u> solution

(The Dining Philosophers Problem)

- the classical illustration of deadlocking
- it was created to illustrate deadlock problems
- it is a very artificial problem
  - it was carefully designed to cause deadlocks
  - changing the rules eliminate deadlocks
  - but then it couldn't be used to illustrate deadlocks

Deadlocks May Not Be Obvious

- process resource needs are ever-changing
  - depending on what data they are operating on
  - depending on where in computation they are
  - depending on what errors have happened
- modern software depends on many services
  - most of which are ignorant of one-another
  - each of which requires numerous resources
- services encapsulate much complexity
  - we do not know what resources they require
  - we do not know when/how they are serialized

Many Types of Deadlocks

- Different deadlocks require different solutions
- Commodity resource deadlocks
  - e.g. memory, queue space
- General resource deadlocks
  - e.g. files, critical sections
- Heterogeneous multi-resource deadlocks
  - e.g. P1 needs a file, P2 needs memory
- Producer-consumer deadlocks
  - e.g. P1 needs a file, P2 needs a message from P1

Approaches

- Avoidance
  - evaluate each proposed action
  - avoid taking actions that would deadlock
- Prevention
  - design system to make deadlock impossible
- Detection and Recovery
  - wait for it to happen
  - try to detect that it has happened
  - take some action to break the deadlock

Commodity vs. General Resources

- Commodity Resources
  - clients need an amount of it (e.g. memory)
  - deadlocks result from <u>over-commitment</u>
  - avoidance can be done in resource manager
- General Resources
  - clients need a specific instance of something
    - a particular file or semaphore
    - a particular message or request completion
  - deadlocks result from specific <u>dependency network</u>
  - prevention is usually done at <u>design time</u>

Commodity Resource Problems

- memory deadlock
  - we are out of memory
  - we need to swap some processes out
  - we need memory to build the I/O request
- critical resource exhaustion
  - a process has just faulted for a new page
  - there are no free pages in memory
  - there are no free pages on the swap device

Avoidance – Advance Reservations

- advance reservations for commodities
  - resource manager tracks outstanding reservations
  - only grants reservations if resources are available
- over-subscriptions are detected early
  - before processes ever get the resources
- client must be prepared to deal with failures
  - but these do not result in deadlocks
- dilemma: over-booking vs. under-utilization

Real Commodity Resource Management

- advanced reservation mechanisms are common
  - Unix setbreak system call to allocate more memory
  - disk quotas, <u>Quality of Service</u> contracts
- once granted, reservations are guaranteed
  - allocation failures only happen at reservation time ... hopefully before the new computation has begun
  - failures will not happen at request time
  - system behavior more predictable, easier to handle
- but clients must deal with reservation failures

Dealing with Rejection

- reservations eliminate difficult failures
  - recovering from a failure in mid-computation
  - may involve awkward and complex unwinding
- graceful handling of reservation failures
  - fail new request, but continue running
  - try to reserve essential resources at start-up time
- keep trying until it works ... not so good
  - may impose un-bounded delay on requestor
  - freeing resources or shedding load could help

Pre-reserving critical resources

- system services must never deadlock for memory
- potential deadlock: swap manager
  - invoked to swap out processes to free up memory
  - may need to allocate memory to build I/O request
  - If no memory available, unable to swap out processes
- solution
  - pre-allocate and hoard a few request buffers
  - keep reusing the same ones over and over again
  - little bit of hoarded memory is a small price to pay

Over-Booking vs. Under Utilization

- Problem: reservations overestimate requirements
  - clients seldom need all resources all the time
  - all clients won't need max allocation at the same time
- question: can one safely over-book resources?
  - for example, seats on an airplane :-)
- what is a safe resource allocation?
  - one where everyone will be able to complete
  - some people may have to wait for others to complete
  - we must be sure there are no deadlocks

Deadlock Prevention

- Deadlock has <u>four necessary conditions</u>:
  1. **mutual exclusion**
     P1 cannot use a resource until P2 releases it
  2. **hold and wait**
     process already has R1 blocks to wait for R2
  3. **no preemption**
     R1 cannot be taken away from P1
  4. **circular dependency**
     P1 has R1, and needs R2
     P2 has R2, and needs R1

Attack #1 – Mutual Exclusion

deadlock requires mutual exclusion
  - P1 having the resource precludes P2 from getting it
- you can't deadlock over a shareable resource
  - perhaps maintained with atomic instructions
  - even reader/writer locking can help
    - readers can share, writers may be attacked in other ways
- you can't deadlock if you have private resources
  - can we give each process its own private resource?

Attack #2: hold and block

deadlock requires you to block holding resources
1. allocate all resources in a single operation
   - you hold nothing while blocked
   - when you return, you have <u>all or nothing</u>
2. disallow blocking while holding resources
   - you must release all held locks prior to blocking
   - reacquire them again after you return
3. non-blocking requests
   - a request that can't be satisfied immediately will fail

Attack #3: non-preemption

- deadlock prevents forwards progress
  - can we *back-out* of the deadlock?
  - reclaim resource(s) from current holders
- use *leases* rather than locks
  - process only has resource for a limited time
  - after which ownership is automatically lost
- forceful resource confiscation
- termination ... with extreme prejudice

When is Preemption Feasible?

- Is access mediated by the operating system?
  - e.g. all object access is via system calls
  - we can revoke access, and return errors
- Can we force a graceful release of resource?
  - make a *claw-back* call to the current owner
- Does confiscation leave resource corrupted?
  - we can un-map a segment or kill a process
  - can we return resource to a default initial state?
  - is it protected by all-or-none updates?

Attack #4: circular dependencies

- total resource ordering
  - all requesters allocate resources in same order
  - first allocate R1 and then R2 afterwards
  - someone else may have R2 but he doesn't need R1
- assumes we know how to order the resources
  - order by ID (e.g. I-node #, IP-address, mem address)
  - order by resource type (e.g. groups before members)
  - order by relationship (e.g. parents before children)
- may require a lock dance
  - release R2, allocate R1, reacquire R2

“Lock Dances” to preserve ordering

list head must be locked for searching, adding & deleting

individual buffers must be locked to perform I/O & other operations

To avoid deadlock, we must always lock the list head before we lock an individual buffer.

To find a desired buffer:

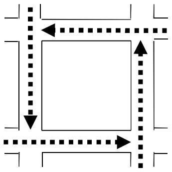
read lock list head
search for desired buffer
lock desired buffer
unlock list head
return (locked) buffer

To delete a (locked) buffer from list

unlock buffer
write lock list head
search for desired buffer
lock desired buffer
remove from list
unlock list head

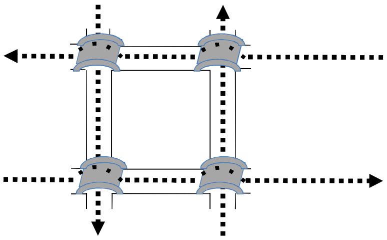
Deadlock – Practical Examples

- the problem – urban gridlock
  - resource: being in the intersection
  - deadlock: nobody can get through



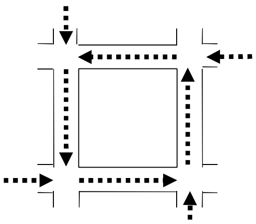
Prevention: Mutual Exclusion

- Build overpass bridges for east/west traffic



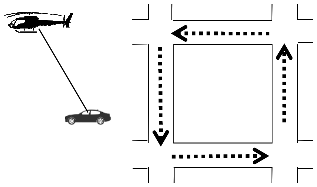
Prevention: Hold and Block

- illegal to enter the intersection if you can’t exit
  - thus, preventing “holding” of the intersection



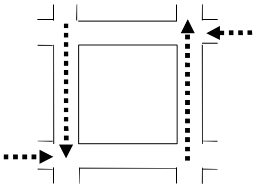
Prevention: Preemption

- Helicopters forcibly remove blocking vehicles



Prevention: Circular Dependencies

- decree a total ordering for right of way
  - e.g., North beats West beats South beats East



Deadlocks: divide and conquer!

- There is no one universal solution to all deadlocks
  - fortunately, we don't need a universal solution
  - we only need <u>a solution for each resource</u>
- Solve each individual problem any way you can
  - make resources sharable wherever possible
  - use reservations for commodity resources
  - ordered locking or no hold-and-block where possible
  - as a last resort, leases and lock breaking
- OS must prevent deadlocks in all system services
  - applications are responsible for their own behavior

Closely related forms of "hangs"

- live-lock
  - process is running, but won't free R1 until it gets msg
  - process that will send the message is blocked for R1
- Sleeping Beauty, waiting for “Prince Charming”
  - a process is blocked, awaiting some completion
  - but, for some reason, it will never happen
- neither of these is a true deadlock
  - wouldn't be found by deadlock detection algorithm
  - both leave the system just as hung as a deadlock

Deadlock vs. "hang" detection

- deadlock detection seldom makes sense
  - it is extremely complex to implement
  - only detects true deadlocks for known resources
- service/application "health monitoring" does
  - monitor application progress/submit test transactions
  - if response takes too long, declare service "hung"
- health monitoring is easy to implement
- it can detect a wide range of problems
  - deadlocks, live-locks, infinite loops & waits, crashes

Hang/Failure Detection Methodology

- look for obvious failures
  - – process exits or core dumps
- passive observation to detect hangs
  - – is process consuming CPU time, or is it blocked
  - – is process doing network and/or disk I/O
- external health monitoring
  - – “pings”, null requests, standard test requests
- internal instrumentation
  - – white box audits, exercisers, and monitoring

Automated Recovery

- kill and restart “all of the affected software”
- how will this affect service/clients
  - – design services to automatically fail-over
  - – components can warm-start, fall back to last check-point, or cold start
- which, and how many processes to kill?
  - – define service failure/recovery zones
  - – processes to be started/killed as a group
  - – progressive levels of increasingly scope/severity

When formal detection makes sense

- Problem: Priority Inversion (a demi-deadlock)
  - – preempted low priority process P1 has mutex M1
  - – high priority process P2 blocks for mutex M1
  - – process P2 is effectively reduced to priority of P1
- Consequences:
  - – depends on what high priority process does
    - might go unnoticed
    - might be a minor performance issue
    - might result in disaster

Priority Inversion on Mars

- occurred on the Mars Pathfinder rover
- caused serious problems with system resets
- very difficult to find

The Pathfinder Priority Inversion

- Special purpose h/w, VxWorks real-time OS
- preemptive priority scheduling
  - – to ensure execution of most critical tasks
- shared an “information bus”
  - – shared memory region
  - – used to communicate between components
  - – shared data protected by a mutex lock

A Tale of Three Tasks

- P1: critical, high priority bus management task
  - – ran frequently for brief periods, holding bus lock
  - – watchdog timer made sure that P1 was still running
- P3: low priority meteorological task
  - – ran occasionally, for brief periods, holding bus lock
  - – Also for brief periods, during which it locked the bus
- P2: medium priority communications task
  - – ran rarely, for longtime, did not need or hold bus loc
- A very rare race condition:
  - – P3 had the lock, and was preempted by P2
  - – P1 can preempt P2, but blocks until P3 completes
  - – P1 is now waiting for (much lower priority) P3
  - – watchdog timer concludes P1 has failed, resets system

Solution: Priority Inheritance

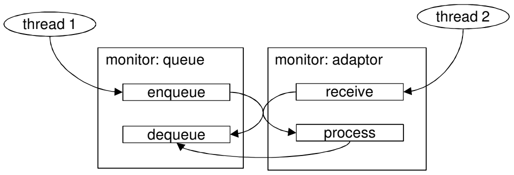
- Identify resource that is blocking P1
- Identify current owner of that resource (P3)
- Temporarily raise P3 priority to that of P1
  - until P3 releases the mutex
- P3 now preempts P2, runs to completion
- P3 releases lock, and loses inherited priority
- P1 preempts P2 and runs
- P2 resumes execution

Assignments

- Reading
  - Metrics and Measurement
  - Load and Stress Testing
- Lab
  - get started on 2B

Supplementary Slides

nested monitors – example



(nested monitors – simpler isn't safer)

- consider two monitors:
  - QUEUE with methods: enqueue, dequeue
  - ADAPTOR with methods: process, receive
    - where ADAPTORs are implemented with QUEUEs
- possible static deadlocks:
  - QUEUE.enqueue adds entry, calls ADAPTOR.process
  - ADAPTOR.process calls QUEUE.dequeue
- possible dynamic deadlocks:
  - thread 1 calls QUEUE.enque, calls ADAPTOR.process
  - thread 2 calls ADAPTOR.receive, calls QUEUE.enqueue

Monitors: simplicity vs. performance

- monitor locking is very conservative
  - lock the entire class (not merely a specific object)
  - lock for entire duration of any method invocations
- this can create performance problems
  - they eliminate conflicts by eliminating parallelism
  - if a thread blocks in a monitor a convoy can form
- There Ain't No Such Thing As A Free Lunch
  - fine-grained locking is difficult and error prone
  - coarse-grained locking creates bottle-necks

Monitors: implementation

```
monitor generic {
   semaphore mutex = 1;
   … other private data …
// public external entrypoints … all protected by mutex
   public:          method_1(parms) {
                              p(&mutex);
                              _method_1(parms);
                              v(&mutex); }
// real implementations
           _method_1(parms) { … }
}
```

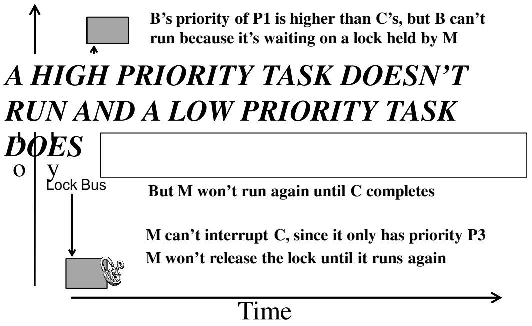
Solutions that do work

- avoid shared data whenever possible
- eliminate critical sections w/atomic instructions
  - atomic (uninteruptable) read/modify/write operations
  - can be applied to 1-8 contiguous bytes
  - simple: increment/decrement, and/or/xor
  - complex: test-and-set, exchange, compare-and-swap
- use atomic instructions to implement locks
  - use the lock operations to protect critical sections

Limitations of atomic instructions

- only update a small number of contiguous bytes
  - cannot be used to atomically change multiple locations (e.g. insertions in a doubly-linked list)
- they operate on a single memory bus
  - cannot be used to update records on disk
  - cannot be used across a network
  - lock-out and synchronized write are very expensive
- they are not higher level locking operations
  - they cannot "wait" until a resource becomes available

The Priority Inversion at Work

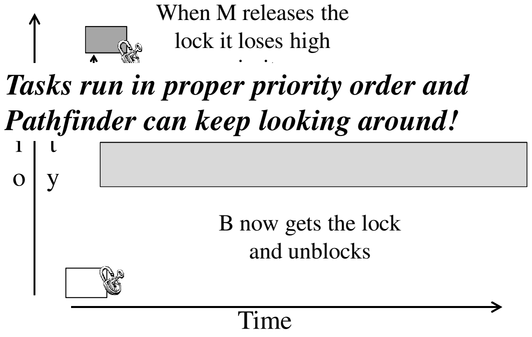


A HIGH PRIORITY TASK DOESN'T RUN AND A LOW PRIORITY TASK DOES

Handling Priority Inversion Problems

- In a priority inversion, lower priority task runs because of a lock held elsewhere
  - Preventing the higher priority task from running
- In the Mars Rover case, the meteorological task held a lock
  - A higher priority bus management task couldn't get the lock
  - A medium priority, but long, communications task preempted the meteorological task
  - So the medium priority communications task ran instead of the high priority bus management task

The Fix in Action



Tasks run in proper priority order and Pathfinder can keep looking around!