Definition

• A thread is deadlocked when it’s waiting for an event that can never occur
  – I’m waiting for you to clear the intersection, so I can proceed
    • but you can’t move until he moves, and he can’t move until she moves, and she can’t move until I move
  – Thread A is in critical section 1, waiting for access to critical section 2; thread B is in critical section 2, waiting for access to critical section 1
  – I’m trying to book a vacation package to Tahiti – air transportation, ground transportation, hotel, side-trips. It’s all-or-nothing – one high-level transaction – with the four databases locked in that order. You’re trying to do the same thing in the opposite order.
Four conditions must exist for deadlock to be possible

1. Mutual Exclusion

2. Hold and Wait

3. No Preemption

4. Circular Wait

We’ll see that deadlocks can be addressed by attacking any of these four conditions.
Resource Graphs

- Resource graphs are a way to visualize the (deadlock-related) state of the threads, and to reason about deadlock.

- 1 or more identical units of a resource are available
- A thread may hold resources (arrows to threads)
- A thread may request resources (arrows from threads)
A deadlock exists if there is an *irreducible cycle* in the resource graph (such as the one above).

- R1 is held by
- is waiting for R1
- R2 is held by
- is waiting for R2
Graph reduction

• A graph can be reduced by a thread if all of that thread’s requests can be granted
  – in this case, the thread eventually will terminate – all resources are freed – all arcs (allocations) to/from it in the graph are deleted

• Miscellaneous theorems (Holt, Havender):
  – There are no deadlocked threads iff the graph is completely reducible
  – The order of reductions is irrelevant
Resource allocation graph with no cycle

What would cause a deadlock?
Resource allocation graph with a deadlock
Resource allocation graph with a cycle but no deadlock
Handling Deadlock

• Eliminate one of the four required conditions
  – Mutual Exclusion
    • Clearly we’re not going to eliminate this one!
  – Hold and Wait
  – No Preemption
  – Circular Wait

• Broadly classified as:
  – Prevention, or
  – Avoidance, or
  – Detection (and recovery)
Prevention

Applications must conform to behaviors guaranteed not to deadlock

• Eliminating hold and wait
  • each thread obtains all resources at the beginning
  • blocks until all are available
    • drawback?

• Eliminating circular wait
  • resources are ordered (numbered)
  • each thread obtains resources in sequence order (which could require acquiring some before they are actually needed)
    • why does this work?
    • pros and cons?
Avoidance

Less severe restrictions on program behavior

• Eliminating circular wait
  – each thread states its maximum claim for every resource type
  – system runs the Banker’s Algorithm at each allocation request
    • Banker ⇒ incredibly conservative
    • if I were to allocate you that resource, and then everyone were to request their maximum claim for every resource, could I find a way to allocate remaining resources so that everyone finished?
  – More on this in a moment…
Detect and recover

• Every once in a while, check to see if there’s a deadlock
  – how?

• If so, eliminate it
  – how?
Avoidance: Banker’s Algorithm example

• Background
  – The set of controlled resources is known to the system
  – The number of units of each resource is known to the system
  – Each application must declare its maximum possible requirement of each resource type

• Then, the system can do the following:
  – When a request is made
    • pretend you granted it
    • pretend all other legal requests were made
    • can the graph be reduced?
      – if so, allocate the requested resource
      – if not, block the thread until some thread releases resources, and then try pretending again
1. I request a pot
Suppose we allocate, and then everyone requests their max? It’s OK; there is a way for me to complete, and then you can complete.
2. You request a pot
Suppose we allocate, and then everyone requests their max? It’s OK; there is a way for me to complete, and then you can complete.
3a. You request a pan
Suppose we allocate, and then everyone *incrementally* requests their max? **NO!** Both of us might be unable to complete!
3b. I request a pan

Me

You

Max: 1 pot 2 pans

Max: 2 pots 1 pan

Pots

Pans
Suppose we allocate, and then everyone incrementally requests their max? **NO!** Both of us might be unable to complete!
Current practice

• Microsoft SQL Server
  – “The SQL Server Database Engine automatically detects deadlock cycles within SQL Server. The Database Engine chooses one of the sessions as a deadlock victim and the current transaction is terminated with an error to break the deadlock.”

• Oracle
  – As Microsoft SQL Server, plus “Multitable deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order... For example, all application developers might follow the rule that when both a master and detail table are updated, the master table is locked first and then the detail table.”
Windows internals (Linux no different)

- “Unless they did a huge change in Vista (and from what I've heard they haven't modified this area), the NT kernel architecture is a deadlock minefield. With the multi-threaded re-entrant kernel there is plenty of deadlock potential.”
- “Lock ordering is great in theory, and NT was originally designed with mutex levels, but they had to be abandoned. Inside the NT kernel there is a lot of interaction between memory management, the cache manager, and the file systems, and plenty of situations where memory management (maybe under the guise of its modified page writer) acquires its lock and then calls the cache manager. This happens while the file system calls the cache manager to fill the cache which in turn goes through the memory manager to fault in its page. And the list goes on.”
Non-blocking algorithms

• An algorithm is non-blocking if a slow thread cannot prevent another faster thread from making progress
  – Using locks is not non-blocking because a thread may acquire the lock and then run really really slowly
    • (Why?)

• Non-blocking algorithms are often built on an atomic hardware instruction, Compare And Swap (CAS)

```cpp
bool CAS(ptr, old, new) {
    if ( *ptr == old ) { *ptr = new; return true; }  
    return false;
}
```
Non-blocking atomic integer

- int atomic_int_add(atomic_int *p, int val) {
  int oldval;
  do {
    oldval = *p;
  } while ( ! CAS(p, oldval, oldval+val) );

- What happens if multiple threads execute this concurrently?
  - Does every thread make progress?
  - Does at least one thread make progress in bounded number of steps?
Why non-blocking

• What if a thread is pre-empted while holding a lock?

• If there are no locks, can there be deadlock?

• Suppose a low priority thread holds a lock needed by a high priority thread
  – (Alternative solution: priority inheritance)
Why not non-blocking?
(Non-blocking FIFO implementation)

Pointers are stored with a generation number in one 8-byte quantity
(32-bit pointer + 32-bit generation number)

From Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
by Michael & Scott.
Non-blocking FIFO: enqueue()

```
enqueue(Q: pointer to queue, value: data type)
E1: node = new_node()
E2: node->value = value
E3: node->next.ptr = NULL
E4: loop
E5: tail = Q->Tail
E6: next = tail.ptr->next
E7: if tail == Q->Tail
E8: if next.ptr == NULL
E9: if CAS(&tail.ptr->next, next, <node, next.count+1>)
E10: break
E11: endif
E12: else
E13: CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
E14: endif
E15: endif
E16: endloop
E17: CAS(&Q->Tail, tail, <node, tail.count+1>)
```

# Allocate a new node from the free list
# Copy enqueued value into node
# Set next pointer of node to NULL
# Keep trying until Enqueue is done
# Read Tail.ptr and Tail.count together
# Read next ptr and count fields together
# Are tail and next consistent?
# Was Tail pointing to the last node?
# Try to link node at the end of the linked list
# Enqueue is done. Exit loop
# Tail was not pointing to the last node
# Try to swing Tail to the next node
# Enqueue is done. Try to swing Tail to the inserted node
Non-blocking FIFO: dequeue

```
dequeue(Q: pointer to queue, pvalue: pointer to data type): boolean
D1:    loop
D2:        head = Q->Head
D3:        tail = Q->Tail
D4:        next = head->next
D5:        if head == Q->Head
D6:            if head.ptr == tail.ptr
D7:                if next.ptr == NULL
D8:                    return FALSE
D9:            endif
D10:        CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
D11:    else
D12:        *pvalue = next.ptr->value
D13:        if CAS(&Q->Head, head, <next.ptr, head.count+1>)
D14:            break
D15:        endif
D16:    endif
D17:    endif
D18:    endloop
D19:    free(head.ptr)
D20:    return TRUE
```

# Keep trying until Dequeue is done
# Read Head
# Read Tail
# Read Head.ptr->next
# Are head, tail, and next consistent?
# Is queue empty or Tail falling behind?
# Is queue empty?
# Queue is empty, couldn’t dequeue

# Tail is falling behind. Try to advance it
# No need to deal with Tail

# Read value before CAS, otherwise another dequeue might free the next node

# Try to swing Head to the next node
# Dequeue is done. Exit loop

# It is safe now to free the old dummy node
# Queue was not empty, dequeue succeeded

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski
Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

12 processor Silicon Graphics Challenge
Summary

• Deadlock is bad!

• We can deal with it either statically (prevention) or dynamically (avoidance and/or detection)

• In practice, you’ll encounter lock ordering, periodic deadlock detection/correction, and minefields