CS162
Operating Systems and Systems Programming
Lecture 7

Language Support for Concurrent Programming, Deadlocks

February 12, 2014

Guest Lecturer: Prof. David Wagner

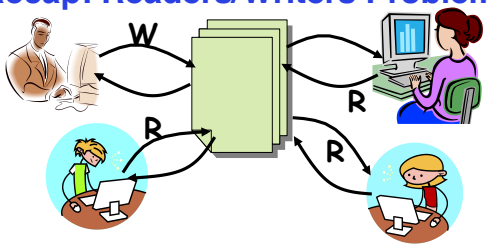
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Recap: Readers/Writers
- Language Support for Synchronization
- Discussion of Resource Contention and Deadlocks
  - Conditions for its occurrence
  - Solutions for breaking and avoiding deadlock

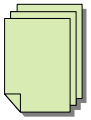
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.

Recap: Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Recap: Readers/Writers Solution



- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - **`Reader()`**
    ```
    Wait until no writers
    Access database
    Check out – wake up a waiting writer
    ```
  - **`Writer()`**
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called "lock"):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Condition okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                 // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--;                 // No longer waiting
  }
  AR++;                   // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                   // No longer active
  if (AR == 0 && WW > 0)  // No other active readers
    okToWrite.signal();   // Wake up one writer
  lock.Release();
}
```


Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    okToWrite.wait(&lock); // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    okToWrite.signal();   // Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    okToRead.broadcast(); // Wake all readers
  }
  lock.Release();
}
```


C-Language Support for Synchronization

- C language: All locking/unlocking is explicit: you need to check every possible exit path from a critical section.

```
int Rtn() {
   lock.acquire();
   …
   if (error) {
      lock.release();
      return errReturnCode;
   }
   …
   lock.release();
   return OK;
}
```


C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are more challenging: exceptions create many new exit paths from the critical section.
  - Consider:

```
void Rtn() {
   lock.acquire();
   …
   DoFoo();
   …
   lock.release();
}
void DoFoo() {
   …
   if (exception) throw errException;
   …
}
```

  - Notice that an exception in DoFoo() will exit without releasing the lock

C++ Language Support for Synchronization (cont'd)

- Must catch **all** exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
  lock.acquire();
  try {
    …
    DoFoo();
    …
  } catch (...) {      // really three dots!
                       // catch all exceptions
    lock.release();    // release lock
    throw;             // re-throw unknown exception
  }
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```


C++ Language Support for Synchronization (cont'd)

- Alternative (Recommended by Stroustrup): Use the lock class destructor to release the lock.
- Set it on entry to critical section contained in a { } block, gets automatically destroyed (& released) on block exit.
- Exceptions will unwind the stack, call destructor, free the lock

```
class lock {
  mutex &m_;
public:
  lock(mutex &m) : m_(m) {
    m.acquire();
  }
  ~lock() {
    m_.release();
  }
};
```

```
mutex m;
...
{
  lock mylock(m);
  ...
  ...
  ... // no explicit unlock
}
```

Critical Section

Java Language Support for Synchronization

Java supports both low-level and high-level synchronization:

- Low-level:
  - Lock class: a lock, with methods:
    - » lock.lock()
    - » lock.unlock()
  - Condition: a condition variable associated with a lock, methods:
    - » condvar.await()
    - » condvar.signal()
- High-level: every object has an implicit lock and condition var
  - synchronized keyword, applies to methods or blocks
  - Implicit condition variable methods:
    - » wait()
    - » notify() and notifyAll()

Java Language Low-level Synchronization

```
public class SynchronizedQueue {
   private Lock lock = new ReentrantLock();
   private Condition cv = lock.newCondition();
   private LinkedList<Integer> q = new LinkedList<Integer>();

   public void enqueue(int item) {
      try {
         lock.lock();
         q.add(item);
         cv.signal();
      } finally {
         lock.unlock();
      }
   }
```

Java Language Low-level Synchronization

```
...
   public synchronized int dequeue() {
      int retval = 0;
      try {
         lock.lock();
         while (q.size() == 0) {
            cv.await();
         }
         retval = q.removeFirst();
      } finally {
         lock.unlock();
      }
      return retval;
   }
}
```


Java High-Level Synchronization

KISS Principle:

KEEP IT SIMPLE, STUDENT!

Explicit locks can help efficiency, but are difficult to analyze.

They also make code more brittle and hard to maintain – constraints and invariants must hold in original code, but also in all modified versions.

Q: What is the typical lifetime of a piece of code?

A: At least a decade longer than any of the original developers anticipated!

Concurrency Bugs (Lu et al. 2008)

Most concurrency bugs (98%) are either

1. Atomicity violations (not protecting shared resources)
2. Order violations
3. Deadlocks

Type 1. problems are caused by under-protecting shared resources, type 3. often caused by over-protection.
Fixes to type 3. bugs often create type 1. bugs. ☹

Good news:

4. Most non-deadlock bugs involve *only one variable*.
5. Most (97%) of deadlocks involve *two threads* which *access at most two resources*.

Not-so-good news: concurrency bugs seem to be a small fraction of all reported bugs, but consume a large fraction of debugging time (days per bug instead of hours).

Java Language High-level Synchronization

- Every object in Java has an implicit lock associated with it.
- The synchronized keyword wraps this lock around a method or a block:

```
public class TheBank {
   public synchronized Withdraw(..) {
      ... // the implicit lock (on "this") is held in here
   }
}
OR
   synchronized (that) { // Specify which object to lock
      ... // the implicit lock on "that" is held in here
   }
```

The JVM takes care of releasing the lock on normal and abnormal exits from the method or block.

Java Language High-level Synchronization

- In addition to an implicit lock, every object has a single implicit condition variable associated with it
  - How to wait inside a synchronization method of block:
    - » `void wait();`
    - » `void wait(long timeout); // Wait for timeout (msecs)`
    - » `void wait(long timeout, int nanoseconds); //variant`
  - How to signal in a synchronized method or block:
    - » `void notify();  // wakes up oldest waiter`
    - » `void notifyAll(); // like broadcast, wakes everyone`
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

```
t1 = time.now();
while (!ATMRequest()) {
   wait (CHECKPERIOD);
   t2 = time.new();
   if (t2 - t1 > LONG_TIME) checkMachine();
}
```

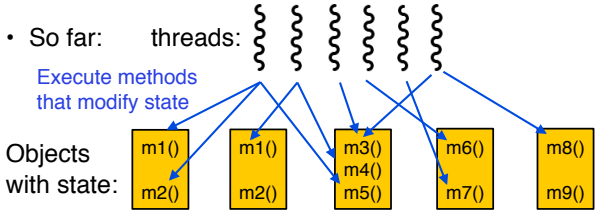
  - Not all Java VMs equivalent!
    - » Different scheduling policies, not necessarily preemptive!

Java Language High-level Synchronization

```
public class SynchronizedQueue {
   public LinkedList<Integer> q = new LinkedList<Integer>();

   public synchronized void enqueue (int item) {
      q.add(item);
      notify();
   }

   public synchronized int dequeue () {
      try {
         while (q.size() == 0) {
            wait();
         }
         return q.removeFirst();
      } catch (InterruptedException e) {
         return 0;
      }
   }
}
```

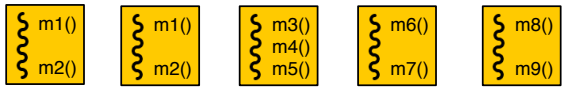

Scala Language: Actors

- Scala is a state-of-the-art language which runs Scala or Java code on a Java Virtual Machine.
- Scala supports Actors, a higher-level abstraction for concurrent programming.
- So far: threads:



Scala Language: Actors

- Actors combine state, methods, and a single thread.



- Actor state is not shared, actors interact by sending and receiving messages.
- Each actor has a single, synchronized message queue, which is part of its implementation.
- Actor code typically comprises a while loop which waits for inbound messages, and dispatches to a message handler.

Scala Actor Bank Account Example

```
val b = actor {            // b is an actor representing a bank account
  var balance = 0.0
  loop {
    react {                // dispatch on the message type
      case ("deposit", amount:Double) => balance += amount
      case ("withdraw", amount:Double) => balance -= amount
      case ("interest", rate:Double) => balance += balance*rate
      case "balance" => println("balance="+balance)
    }
  }
}
var grow = true

val g = actor {            // g is an actor that periodically adds interest
  while (grow) {
    b ! ("interest", 0.05)  // send an interest update message to b
    Thread.sleep(3000)
  }
}
```

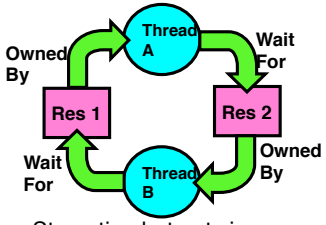

Resource Contention and Deadlock

Resources

- Resources – passive entities needed by threads to do their work
  - CPU time, disk space, memory
- Two types of resources:
  - Preemptable – can take it away
    - » CPU, Embedded security chip
  - Non-preemptable – must leave it with the thread
    - » Disk space, printer, chunk of virtual address space
    - » Critical section
- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing
- One of the major tasks of an operating system is to manage resources

Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - » Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - » Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



  - Deadlock ⇒ Starvation but not vice versa
    - » Starvation can end (but doesn't have to)
    - » Deadlock can't end without external intervention

Conditions for Deadlock

- Deadlock not always deterministic – Example 2 mutexes:

```
x=1, y=1   Thread A          Thread B
            x.P();           y.P();
            y.P();           x.P();
            …                …
            y.V();           x.V();
            x.V();           y.V();
```

Deadlock

```
A: x.P();
B: y.P();
A: y.P();
B: x.P();
...
```

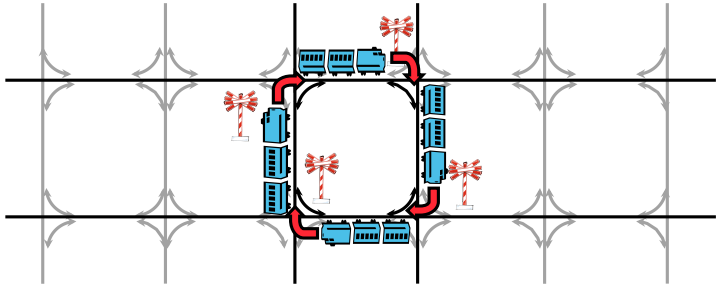
  - Deadlock won't always happen with this code
    - » Have to have exactly the right timing ("wrong" timing?)
- Deadlocks occur with multiple resources
  - Means you can't decompose the problem
  - Can't solve deadlock for each resource independently
- Example: System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one

Bridge Crossing Example

- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ⇒ no one goes west

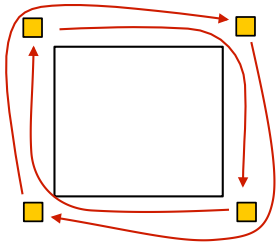
Train Example

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Cannot turn on a track segment if occupied by another train
  - Similar problem to multiprocessor networks
- Ho do you prevent deadlock?
  - (Answer later)

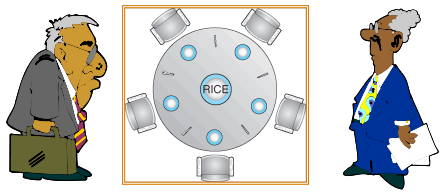


Routing Example

- Circular dependency (Deadlock!)
  - Packets trying to reach a destination two hops away
  - Try to reserve the path to destination – grab first link, then…
  - Important problem to multiprocessor networks
- Ho do you prevent deadlock?
  - (Answer later)



Dining Philosopher Problem



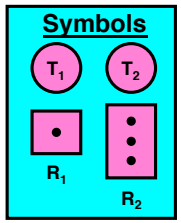
- Five chopsticks/Five philosopher (really cheap restaurant)
  - Free for all: Philosopher will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - (Answer later)

Four requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1

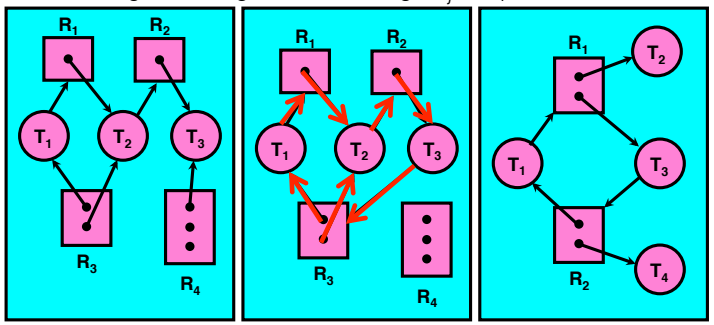
Resource-Allocation Graph

- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
  - Each thread utilizes a resource as follows:
    - » `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Resource Allocation Graph Examples

- Recall:
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph | **Allocation Graph With Deadlock** | **Allocation Graph With Cycle, but No Deadlock**

Administrivia

- Reminder: Nachos Project I design document due tomorrow (2/13) at 11:59PM
  - No slip days allowed

5min Break

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm (Java JMX findDeadlockedThreads(), try also **jvisualvm**)
  - Some technique for forcibly preempting resources and/or terminating tasks
- Deadlock prevention: ensure that system will *never* enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

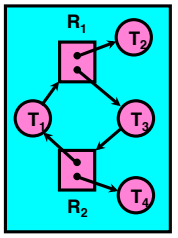
Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

`[FreeResources]:`	Current free resources each type
`[Request`$_X$`]:`	Current requests from thread X
`[Alloc`$_X$`]:`	Current resources held by thread X

  - See if tasks can eventually terminate on their own

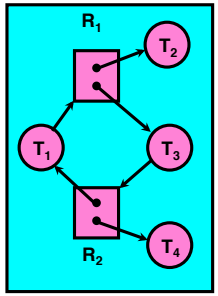
```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in UNFINISHED ⇒ deadlocked

Deadlock Detection Algorithm Example

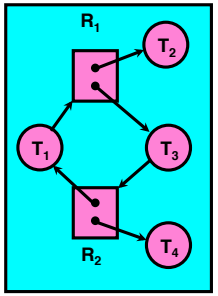
```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



Deadlock Detection Algorithm Example

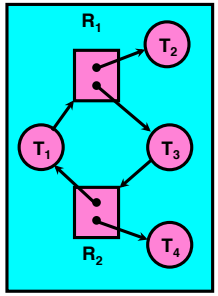
```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T1] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T1]
      done = false
    }
  }
} until(done)
```

False



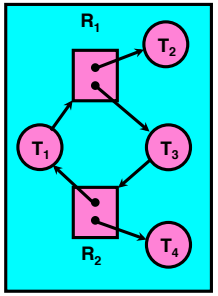
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



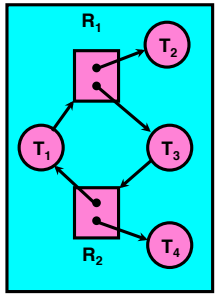
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T2] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T2]
      done = false
    }
  }
} until(done)
```



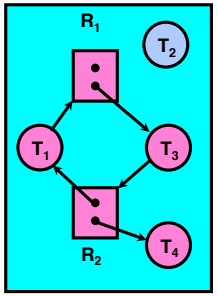
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T2] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T2]
      done = false
    }
  }
} until(done)
```



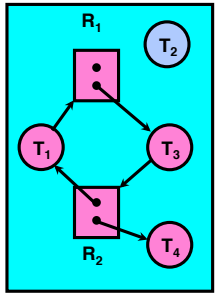
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T2] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T2]
      done = false
    }
  }
} until(done)
```



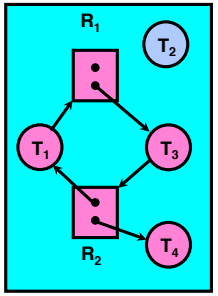
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T2] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T2]
      done = false
    }
  }
} until(done)
```



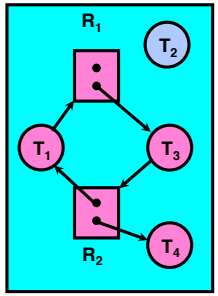
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



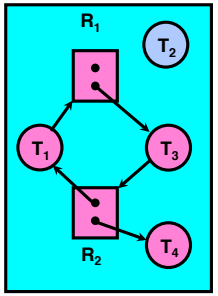
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T3] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T3]
      done = false
    }
  }
} until(done)
```



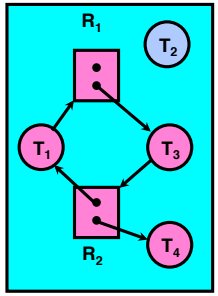
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



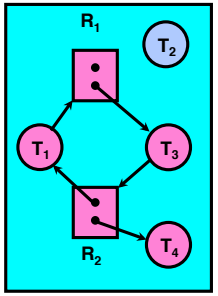
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T4] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T4]
      done = false
    }
  }
} until(done)
```



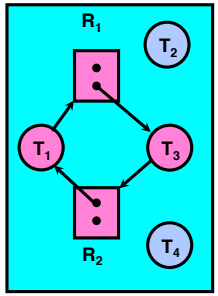
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T4] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T4]
      done = false
    }
  }
} until(done)
```



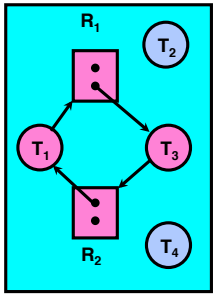
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T4] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T4]
      done = false
    }
  }
} until(done)
```



Deadlock Detection Algorithm Example

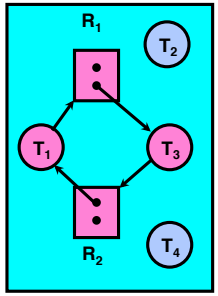
```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T4] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T4]
      done = false
    }
  }
} until(done)
```



Deadlock Detection Algorithm Example

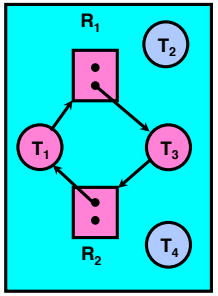
```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T4] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T4]
      done = false
    }
  }
} until(done)
```

False



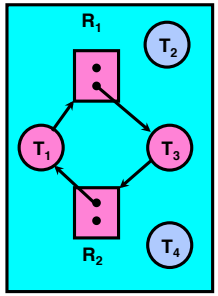
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



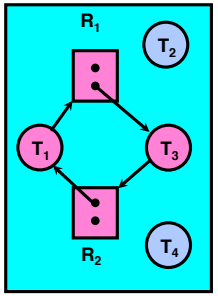
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T1] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T1]
      done = false
    }
  }
} until(done)
```



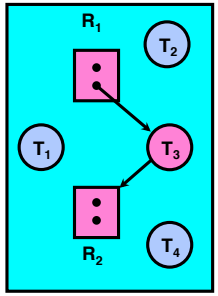
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T1] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T1]
      done = false
    }
  }
} until(done)
```



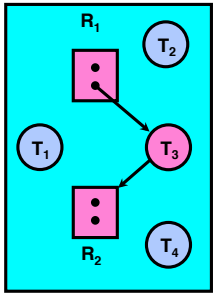
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,2]
UNFINISHED = {T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T1] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T1]
      done = false
    }
  }
} until(done)
```



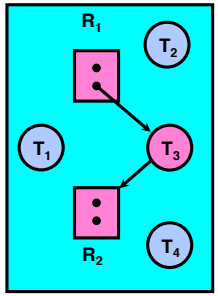
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,2]
UNFINISHED = {T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T1] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T1]
      done = false
    }
  }
} until(done)
```



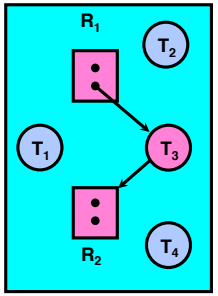
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,2]
UNFINISHED = {T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



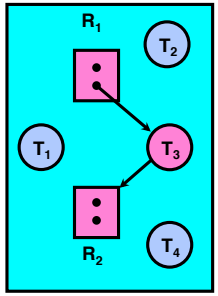
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,2]
UNFINISHED = {T3}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T3] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T3]
      done = false
    }
  }
} until(done)
```



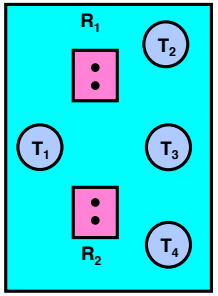
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [1,2]
UNFINISHED = {}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T3] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T3]
      done = false
    }
  }
} until(done)
```



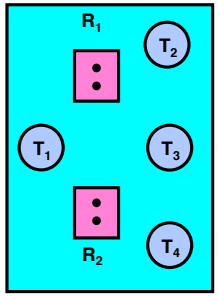
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [2,2]
UNFINISHED = {}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T3] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T3]
      done = false
    }
  }
} until(done)
```



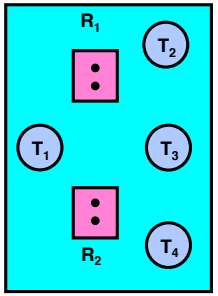
Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [2,2]
UNFINISHED = {}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T3] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T3]
      done = false
    }
  }
} until(done)
```



Deadlock Detection Algorithm Example

```
[Request_T1] = [1,0]; Alloc_T1 = [0,1]
[Request_T2] = [0,0]; Alloc_T2 = [1,0]
[Request_T3] = [0,1]; Alloc_T3 = [1,0]
[Request_T4] = [0,0]; Alloc_T4 = [0,1]
[Avail] = [2,2]
UNFINISHED = {}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_T3] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T3]
      done = false
    }
  }
} until(done)
```



DONE!

Techniques for Preventing Deadlock

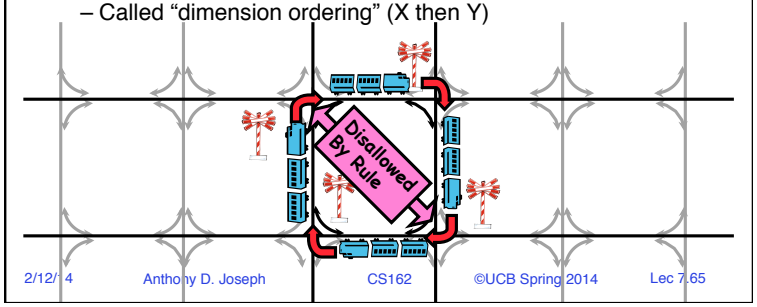
- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don't allow waiting
  - How the phone company avoids deadlock
    - » Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal
  - Technique used in Ethernet/some multiprocessor nets
    - » Everyone speaks at once. On collision, back off and retry

Techniques for Preventing Deadlock (cont'd)

- Make all threads request everything they'll need at the beginning
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » Don't leave home until we know no one is using any intersection between here and where you want to go!
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P,…)
    - » Make tasks request disk, then memory, then…

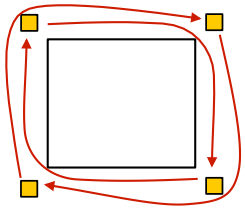
Wormhole-Routed Network

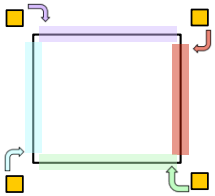
- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Cannot turn on a track segment if occupied by another train
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west (horizontally) first, then north-south (vertically)
  - Called "dimension ordering" (X then Y)



Routing Example

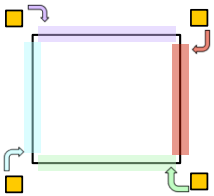
- Circular dependency (Deadlock!)
  - Packets trying to reach a destination two hops away
  - Try to reserve the path to destination – grab first link, then…
  - Important problem to multiprocessor networks
- Use dimension ordering: prioritization of requests, X first, then Y

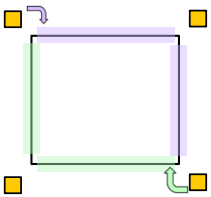




Routing Example

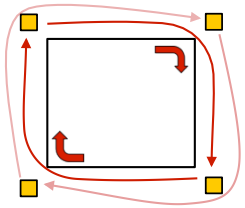
- Circular dependency (Deadlock!)
  - Packets trying to reach a destination two hops away
  - Try to reserve the path to destination – grab first link, then…
  - Important problem to multiprocessor networks
- Use dimension ordering: prioritization of requests, X first, then Y

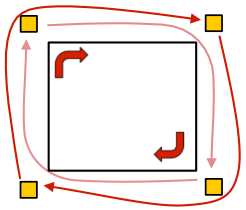




Routing Example

- Circular dependency (Deadlock!)
- Use dimension ordering: prioritization of requests, X first, then Y.
- In effect this prioritizes "East-South" and "West-North" turns when moving clockwise (and West-South and East-North turns going CCW).





Banker's Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) ≥ max remaining that might be needed by any thread

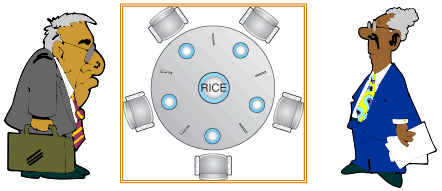
- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Banker's Algorithm

- Technique: pretend each request is granted, then run deadlock detection algorithm, substitute
  ([$Request_{node}$] ≤ [Avail]) → ([Max_{node}]-[$Alloc_{node}$] ≤ [Avail])

```
[FreeResources]:        Current free resources each type
   [Alloc_X]:           Current resources held by thread X
     [Max_X]:           Max resources requested by thread X

     [Avail] = [FreeResources]
   Add all nodes to UNFINISHED
   do {
      done = true
      Foreach node in UNFINISHED {
     if ([Max_node]-[Alloc_node]<= [Avail]) {
           remove node from UNFINISHED
           [Avail] = [Avail] + [Alloc_node]
           done = false
        }
      }
   } until(done)
```


Banker's Algorithm Example



- Banker's algorithm with dining philosophers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards
  - What if k-handed philosophers? Don't allow if:
    - » It's the last one, no one would have k
    - » It's 2nd to last, and no one would have k-1
    - » It's 3rd to last, and no one would have k-2
    - » …

Summary: Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Four conditions for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » ∃ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern
- Deadlock preemption
- Deadlock prevention (Banker's algorithm)