CS162
Operating Systems and Systems Programming
Lecture 6

Semaphores, Conditional Variables, Deadlocks

September 19, 2011

Anthony D. Joseph and Ion Stoica

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

Review: Monitors

- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
- Monitor are dual purpose:
  – Both mutual exclusion and scheduling constraints
  – Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Lock: provides mutual exclusion to shared data:
  – Always acquire before accessing shared data structure
  – Always release after finishing with shared data
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  – Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  – Contrast to semaphores: Can't wait inside critical section

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
      WR++;
      okToRead.wait(&lock);
      WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okToWrite.signal();
    lock.Release();
  }
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
      WW++;
      okToWrite.wait(&lock);
      WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
      okToWrite.signal();
    } else if (WR > 0) {
      okToRead.broadcast();
    }
    lock.Release();
  }
```

What if we turn okToWrite and okToRead into okContinue?

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okContinue.signal();
    lock.Release();
  }
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
      okContinue.signal();
    } else if (WR > 0) {
      okContinue.broadcast();
    }
    lock.Release();
  }
```

What if we turn okToWrite and okToRead into okContinue?

"Homework": Read/Writer Question

```
Reader() {
  // check into system
  lock.Acquire();
  while ((AW + WW) > 0) {
    WR++;
    okContinue.wait(&lock);
    WR--;
  }
  AR++;
  lock.release();
```

- R1 arrives
- AR = 0, WR = 0, AW = 0, WW = 0

```
  // check out of system
  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)
    okContinue.signal();
  lock.Release();
}
```

```
Writer() {
  // check into system
  lock.Acquire();
  while ((AW + AR) > 0) {
    WW++;
    okContinue.wait(&lock);
    WW--;
  }
  AW++;
  lock.release();

  // check out of system
  lock.Acquire();
  AW--;
  if (WW > 0){
    okContinue.signal();
  } else if (WR > 0) {
    okContinue.broadcast();
  }
  lock.Release();
}
```

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
      okContinue.signal();
    } else if (WR > 0) {
      okContinue.broadcast();
    }
    lock.Release();
}
```

- R1 arrives
- AR = 0, WR = 0, AW = 0, WW = 0

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- R1 arrives
- AR = 1, WR = 0, AW = 0, WW = 0

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- R1 arrives
- AR = 1, WR = 0, AW = 0, WW = 0

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okContinue.signal();
    lock.Release();
  }
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
      okContinue.signal();
    } else if (WR > 0) {
      okContinue.broadcast();
    }
    lock.Release();
}
```

- R1 arrives
- AR = 1, WR = 0, AW = 0, WW = 0

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- W1 arrives (R1 in AccessDbase())
- AR = 1, WR = 0, AW = 0, WW = 0

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0){
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- W1 arrives (R1 in AccessDbase())
- AR = 1, WR = 0, AW = 0, WW = 0

"Homework": Read/Writer Question

```
Reader() {
   // check into system
   lock.Acquire();
   while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // read-only access
   AccessDbase(ReadOnly);

   // check out of system
   lock.Acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.signal();
   lock.Release();
}
```

```
Writer() {
   // check into system
   lock.Acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // read/write access
   AccessDbase(ReadWrite);

   // check out of system
   lock.Acquire();
   AW--;
   if (WW > 0){
      okContinue.signal();
   } else if (WR > 0) {
      okContinue.broadcast();
   }
   lock.Release();
}
```

- W1 arrives (R1 in AccessDbase())
- AR = 1, WR = 0, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- W1 arrives (R1 in AccessDbase())
- AR = 1, WR = 0, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
   // check into system
   lock.Acquire();
   while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // check out of system
   lock.Acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.signal();
   lock.Release();
}
```

```
Writer() {
   // check into system
   lock.Acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   lock.Acquire();
   AW--;
   if (WW > 0){
      okContinue.signal();
   } else if (WR > 0) {
      okContinue.broadcast();
   }
   lock.Release();
}
```

- R2 arrives (R1 in AccessDbase(), W1 waits)
- AR = 1, WR = 0, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- R2 arrives (R1 in AccessDbase(), W1 waits)
- AR = 1, WR = 0, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
   // check into system
   lock.Acquire();
   while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // read-only access
   AccessDbase(ReadOnly);

   // check out of system
   lock.Acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.signal();
   lock.Release();
}
```

```
Writer() {
   // check into system
   lock.Acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // read/write access
   AccessDbase(ReadWrite);

   // check out of system
   lock.Acquire();
   AW--;
   if (WW > 0){
      okContinue.signal();
   } else if (WR > 0) {
      okContinue.broadcast();
   }
   lock.Release();
}
```

- R2 arrives (R1 in AccessDbase(), W1 waits)
- AR = 1, WR = 1, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- R2 arrives (R1 in AccessDbase(), W1 waits)
- AR = 1, WR = 1, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- R1 completes (W1 & R2 wait)
- AR = 1, WR = 1, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
   // check into system
   lock.Acquire();
   while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // read-only access
   AccessDbase(ReadOnly);

   // check out of system
   lock.Acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.signal();
   lock.Release();
}
```

```
Writer() {
   // check into system
   lock.Acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // read/write access
   AccessDbase(ReadWrite);

   // check out of system
   lock.Acquire();
   AW--;
   if (WW > 0){
      okContinue.signal();
   } else if (WR > 0) {
      okContinue.broadcast();
   }
   lock.Release();
}
```

- R1 completes (W1 & R2 wait)
- AR = 1, WR = 1, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
   // check into system
   lock.Acquire();
   while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // read-only access
   AccessDbase(ReadOnly);

   // check out of system
   lock.Acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.signal();
   lock.Release();
}
```

```
Writer() {
   // check into system
   lock.Acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // read/write access
   AccessDbase(ReadWrite);

   // check out of system
   lock.Acquire();
   AW--;
   if (WW > 0){
      okContinue.signal();
   } else if (WR > 0) {
      okContinue.broadcast();
   }
   lock.Release();
}
```

- R1 completes (W1 & R2 wait)
- AR = 0, WR = 1, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
   // check into system
   lock.Acquire();
   while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // read-only access
   AccessDbase(ReadOnly);

   // check out of system
   lock.Acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.signal();
   lock.Release();
}
```

```
Writer() {
   // check into system
   lock.Acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // read/write access
   AccessDbase(ReadWrite);

   // check out of system
   lock.Acquire();
   AW--;
   if (WW > 0){
      okContinue.signal();
   } else if (WR > 0) {
      okContinue.broadcast();
   }
   lock.Release();
}
```

- R1 completes (W1 & R2 wait)
- AR = 0, WR = 1, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
   // check into system
   lock.Acquire();
   while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // read-only access
   AccessDbase(ReadOnly);

   // check out of system
   lock.Acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.signal();
   lock.Release();
}
```

signal

```
Writer() {
   // check into system
   lock.Acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // read/write access
   AccessDbase(ReadWrite);

   // check out of system
   lock.Acquire();
   AW--;
   if (WW > 0){
      okContinue.signal();
   } else if (WR > 0) {
      okContinue.broadcast();
   }
   lock.Release();
}
```

- R1 signals; assume signal is delivered to R2 (W1 & R2 wait)
- AR = 0, WR = 1, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
```

signal

- R2 continues (W1 waits)
- AR = 0, WR = 1, AW = 0, WW = 1

```
    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
```

```
    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- R2 continues (W1 waits)
- AR = 0, WR = 0, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
      okContinue.signal();
    } else if (WR > 0) {
      okContinue.broadcast();
    }
    lock.Release();
}
```

- R2 continues (W1 waits)
- AR = 0, WR = 0, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}
```

- R2 continues (W1 waits)
- AR = 0, WR = 1, AW = 0, WW = 1

"Homework": Read/Writer Question

```
Reader() {
   // check into system
   lock.Acquire();
   while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release()

   // read-only access
   AccessDbase(ReadOnly);

   // check out of system
   lock.Acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.signal();
   lock.Release();
}
```

```
Writer() {
   // check into system
   lock.Acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   rite access
   se(ReadWrite);

   // check out of system
   lock.Acquire();
   AW--;
   if (WW > 0){
      okContinue.signal();
   } else if (WR > 0) {
      okContinue.broadcast();
   }
   lock.Release();
}
```

DEADLOCK !!!!

- R2 and W1 both wait!
- AR = 0, WR = 1, AW = 0, WW = 1

Homework: Read/Writer Question

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okContinue.broadcast();
    lock.Release();
  }
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
      okContinue.signal();
    } else if (WR > 0) {
      okContinue.broadcast();
    }
    lock.Release();
}
```

Need to change to broadcast!
Does this work?

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```

Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() {
   if semaphore queue is not empty
      semaphore.V();
}
```

  - Not legal to look at contents of semaphore queue
  - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
   condvar.wait();
}
unlock

do something so no need to wait

lock

condvar.signal();

unlock
```

Check and/or update state variables
Wait if necessary
(release lock when waiting)

Check and/or update state variables

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  – Just make sure you know *all* the code paths out of a critical section

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

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - 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 (con't)

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

```
void Rtn() {
   lock.acquire();
   try {
      …
      DoFoo();
      …
   } catch (…) {       // catch exception
      lock.release(); // release lock
      throw;           // re-throw the exception
   }
   lock.release();
}
void DoFoo() {
   …
   if (exception) throw errException;
   …
}
```

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

  – Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization (con't)

- Java also has *synchronized* statements:

```
synchronized (object) {
      …
}
```

  – Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the code block

  – Works properly even with exceptions:

```
synchronized (object) {
   …
   DoFoo();
   …
}
void DoFoo() {
   throw errException;
}
```

Java Language Support for Synchronization (2/2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method of block:
    - » `void wait();`
    - » `void wait(long timeout); // Wait for timeout`
    - » `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!

Summary: Semaphores and Cond. Variables

- Semaphores: Like integers with restricted interface
  - Two operations:
    - » `P()`: Wait if zero; decrement when becomes non-zero
    - » `V()`: Increment and wake a sleeping task (if exists)
    - » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint

- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`

- Language support for synchronization:
  - Java provides synchronized keyword and one condition-variable per object (with `wait()` and `notify()`)

Announcements

- Project 1 initial design phase AND Project 1 individual part due: Tuesday, September 27 @ 11.59pm
- Midterm: Thursday, October 13, 5-6:30pm – 155 Dwinell Hall
- Discussion sections update: could **not** get a second 6-7pm section, so we will maintain second 10-11am section
  - We've moved some groups consistent with their constraints
    - » 7 groups move to a better choice
    - » 2 groups from 1st to 2nd
    - » 1 group from 1st to 3rd
- CSUA Hackathon: September 23-24

5min Break

Resource Contention and Deadlock

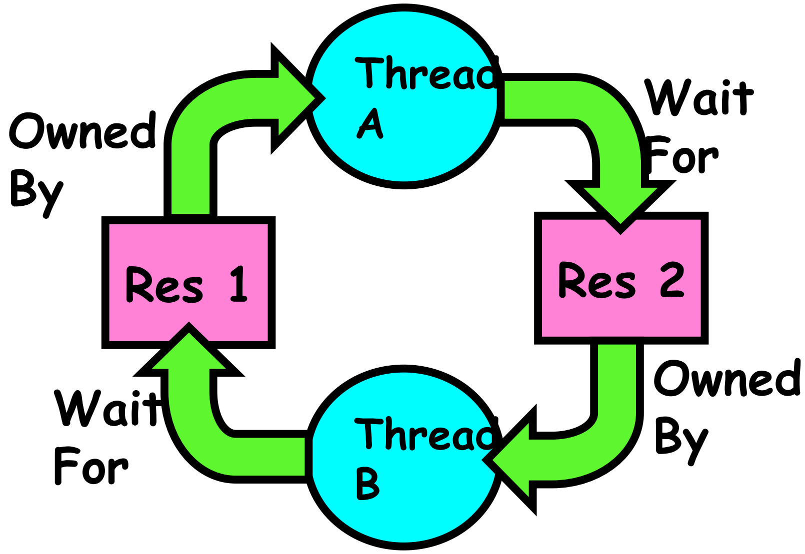
Resources

- 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:

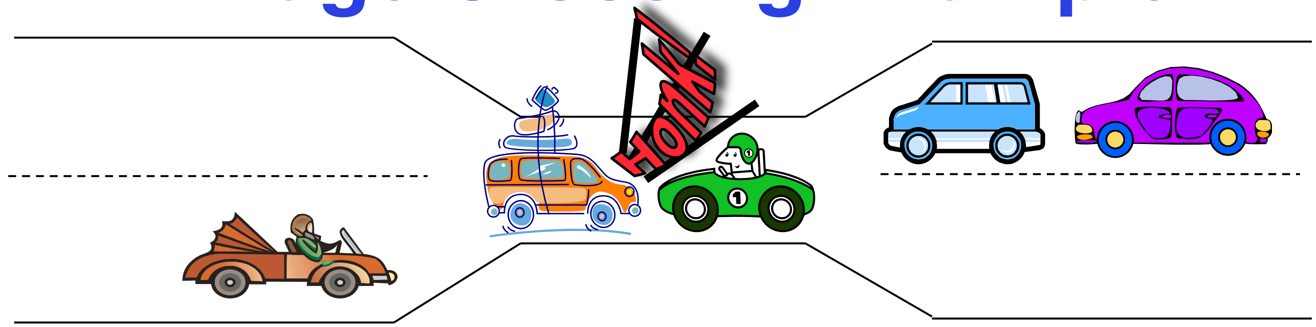
```
Thread A        Thread B
x.P();          y.P();
y.P();          x.P();
…               …
y.V();          x.V();
x.V();          y.V();
```

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

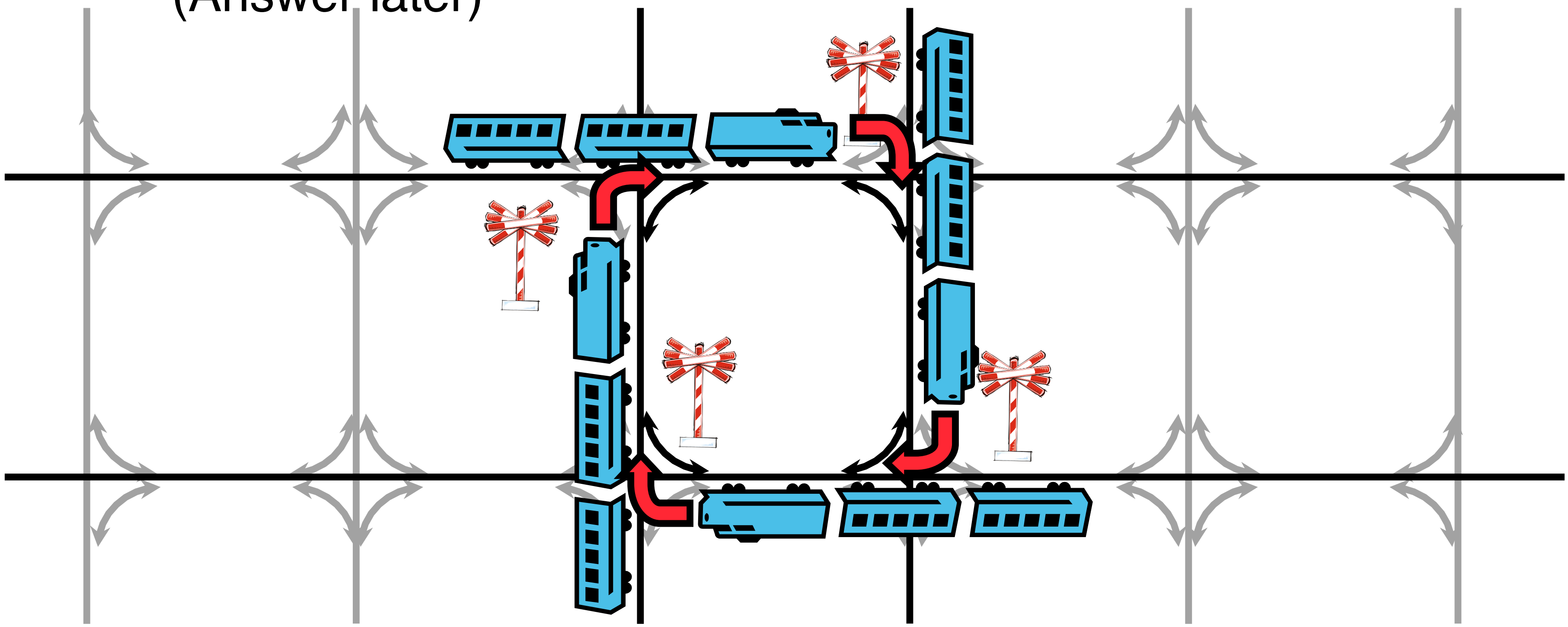
Bridge Crossing Example



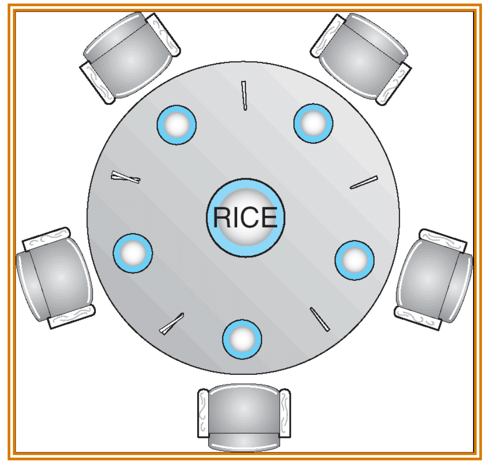
- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires 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
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Ho do you prevent deadlock?
  - (Answer later)



Dining Philosopher Problem

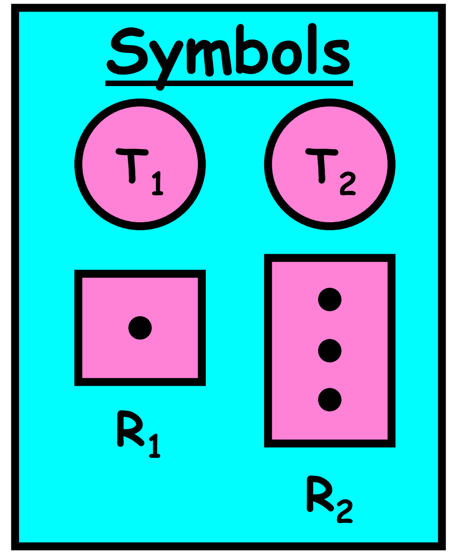


- Five chopsticks/Five philosopher (really cheap restaurant)
  - – Free-for all: Philosopher will grab any one they can
  - – Need two chopsticks to eat
- What if all grab at same time?
  - – Deadlock!
- How to fix deadlock?
  - – Make one of them give up a chopstick (Hah!)
  - – Eventually everyone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

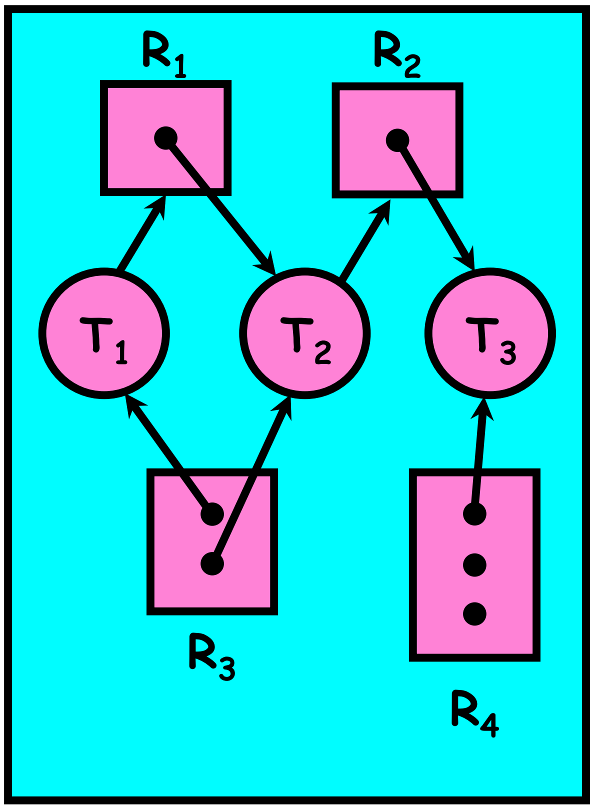


- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

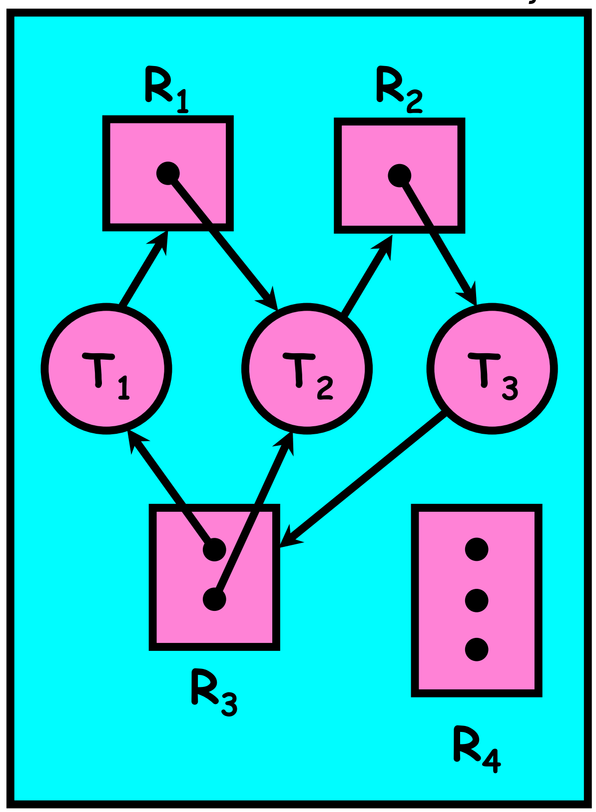
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
  - Each thread utilizes a resource as follows:
    - » `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the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

Resource Allocation Graph Examples

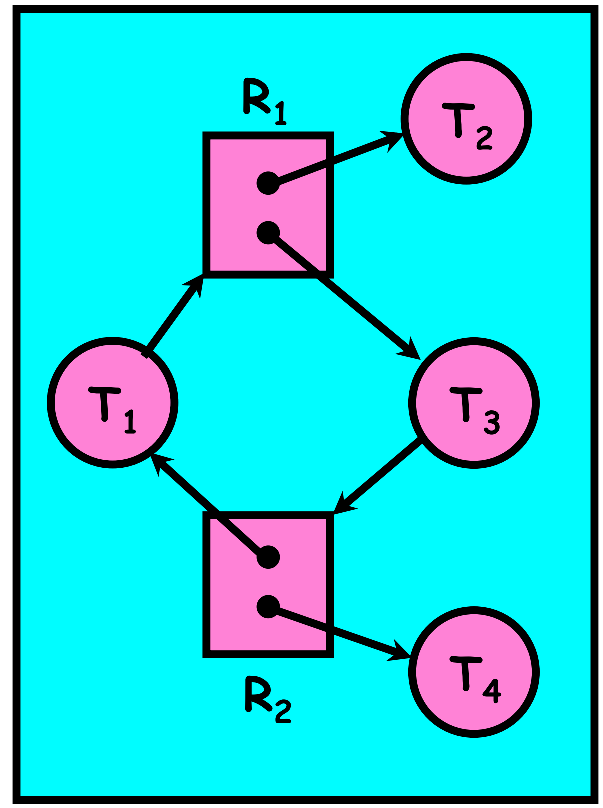
- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- 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

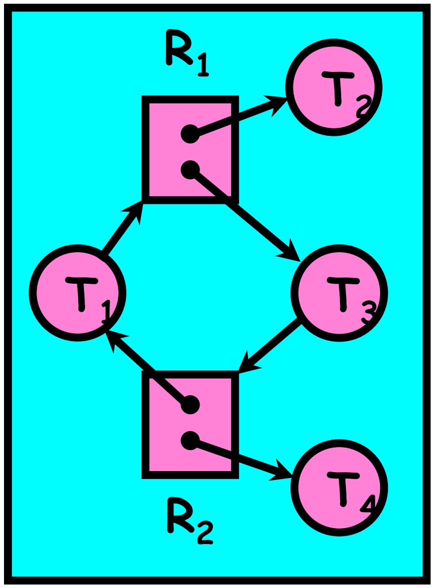
Deadlock Detection Algorithm

- 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)
```



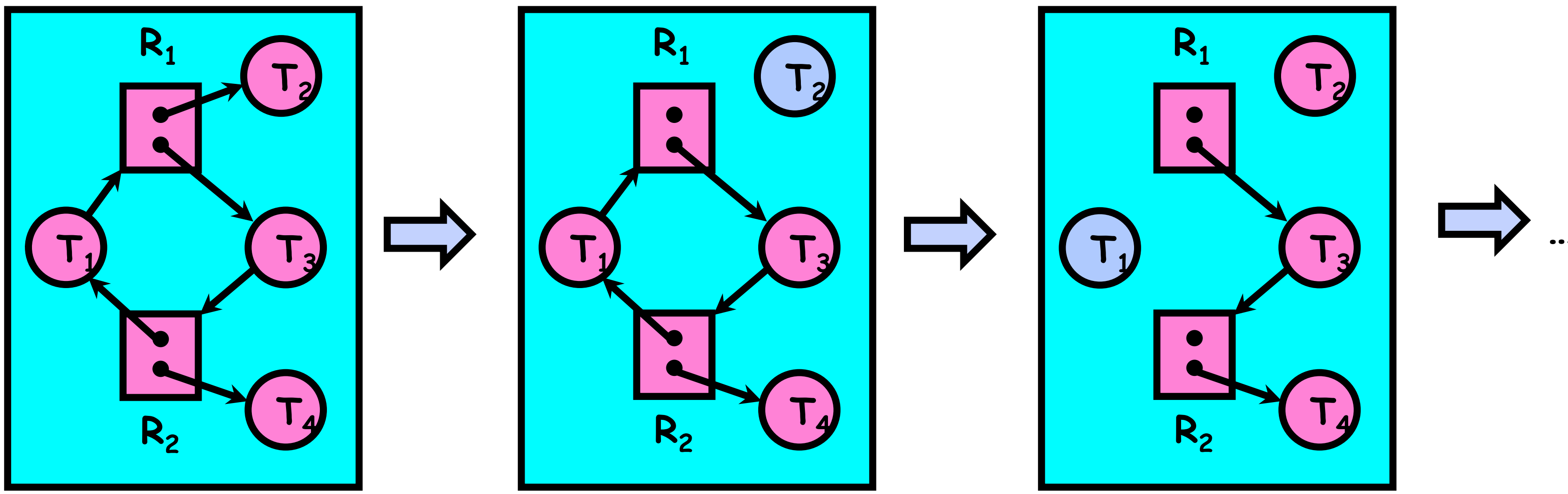
  - Nodes left in UNFINISHED ⇒ deadlocked

Deadlock Detection Algorithm Example

[Available] = [0,0]
$[Request_{T2}]$ = [0,0]
$[Request_{T2}]$ <=
[Available]

[Available] = [1,0]
$[Request_{T1}]$ = [1,0]
$[Request_{T1}]$ <=
[Available]

[Available] = [1,1]
$[Request_{T3}]$ = [0,1]
$[Request_{T3}]$ <=
[Available]



Techniques for Preventing Deadlock

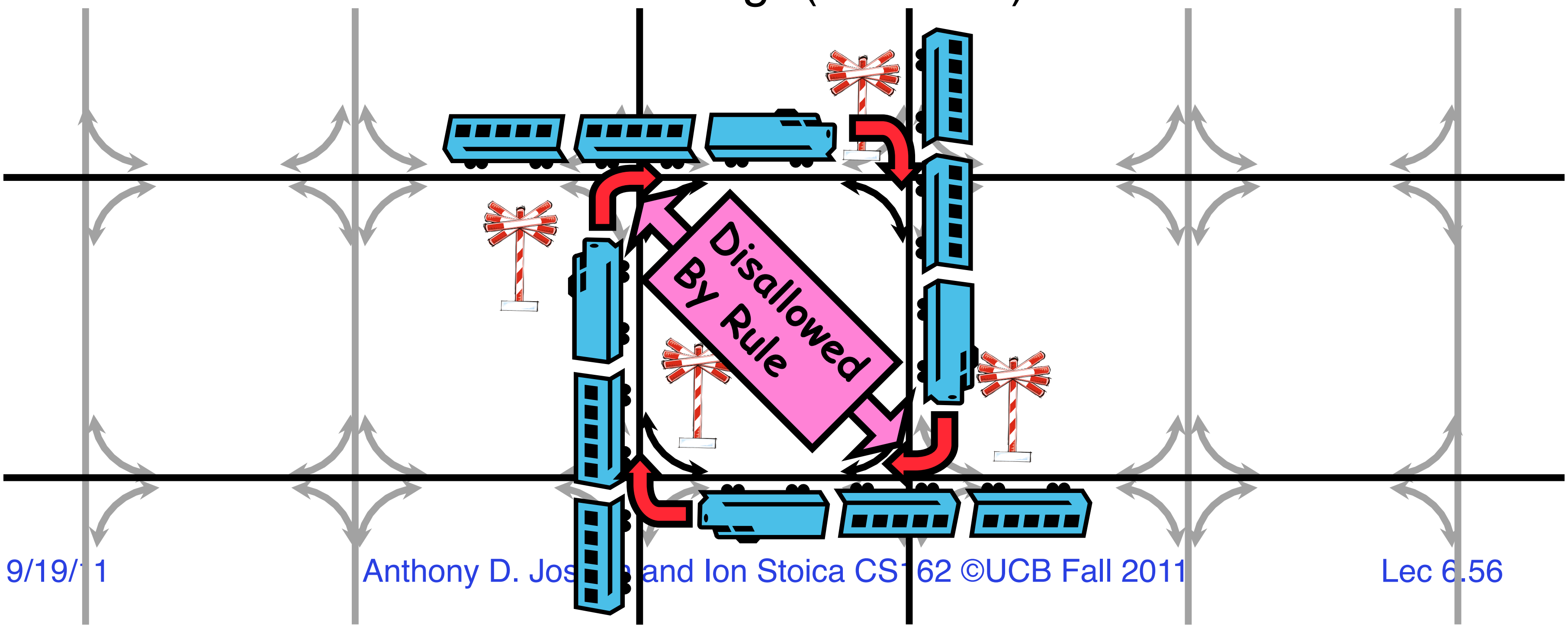
- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic 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 (con't)

- Make all threads request everything they'll need at the beginning
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 chopsticks, request both at same time
    - » 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…

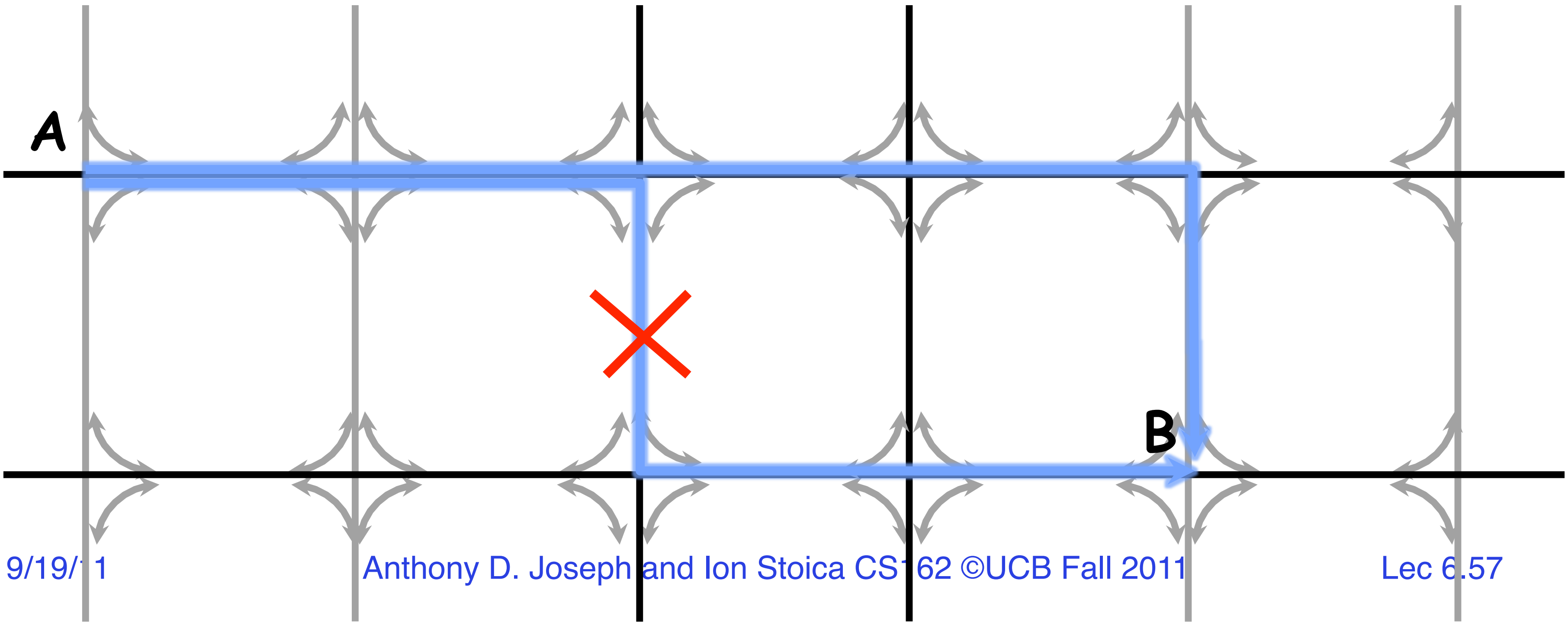
Review: Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  – Each train wants to turn right
  – Blocked by other trains
  – Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  – Force ordering of channels (tracks)
    » Protocol: Always go east-west first, then north-south
  – Called "dimension ordering" (X then Y)



Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



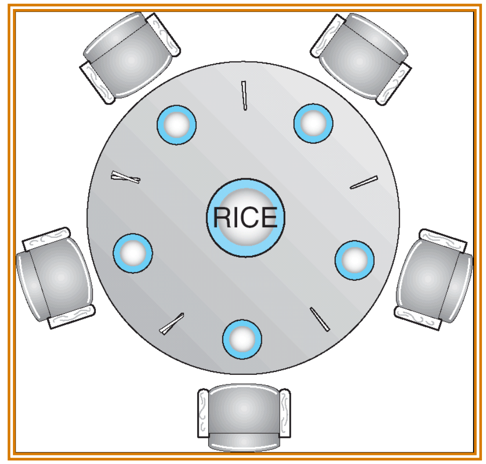
Banker's Algorithm for Preventing Deadlock

- 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
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
      ($[Max_{node}]-[Alloc_{node}] \le [Avail]$) for ($[Request_{node}] \le [Avail]$)
      Grant request if result is deadlock free (conservative!)
    - » 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 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 2^{nd} to last, and no one would have k-1
    - » It's 3^{rd} 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
    - $\exists$ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern
- Deadlock preemption
- Deadlock prevention (Banker's algorithm)