

Operating Systems

Lecture 12

Readers/Writers and Deadlock

Prof. Mengwei Xu

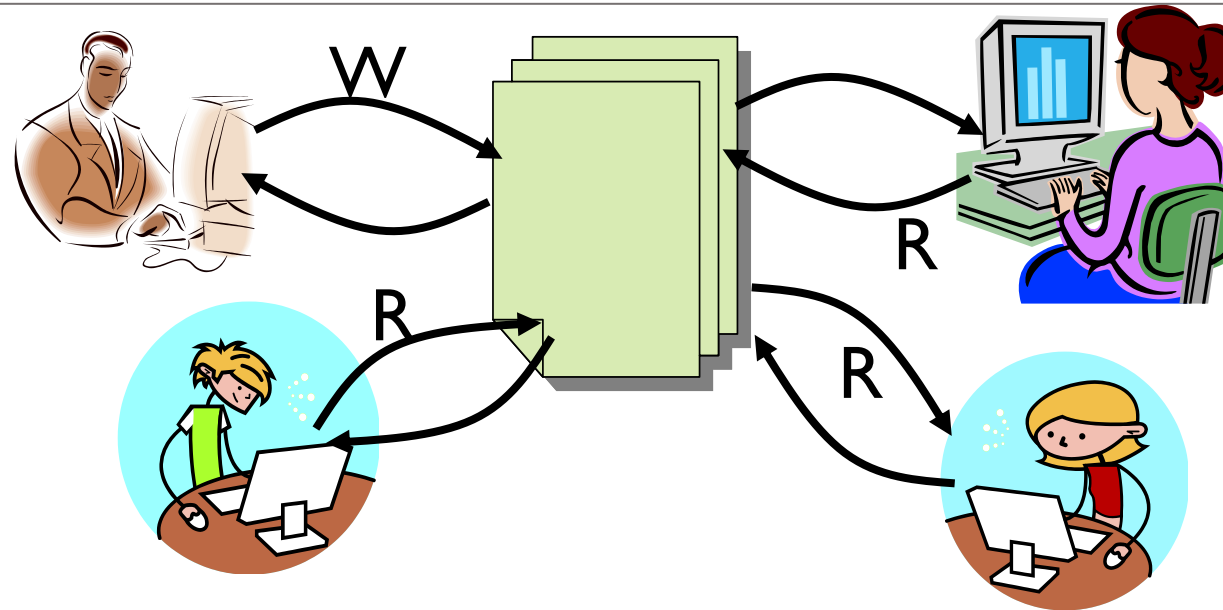
Goals for Today

- Readers/Writers Lock
- Deadlock

Goals for Today

- Readers/Writers Lock
- Deadlock

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

Basic 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 data base
 - 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    AR--;                  // No longer active  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();  
}
```

Why release lock here?

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 AccessDatabase(Re
    // Now, check out
    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();
}

```

Why broadcast()
here instead of
signal()?

Why Give priority to
writers?

Simulation of Readers/Writers Solution

- Use an example to simulate the solution
- Consider the following sequence of operators:
 - R1, R2, W1, R3
- Initially: $AR = 0$, $WR = 0$, $AW = 0$, $WW = 0$

Simulation of Readers/Writers Solution

- R1 comes along
- $AR = 0, WR = 0, AW = 0, WW = 0$

```
Reader() {  
    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();  
  
    AccessDbase(ReadOnly);  
  
    lock.Acquire();  
    AR--;  
    if (AR == 0 && WW > 0)  
        okToWrite.signal();  
    lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R1 comes along
- $AR = 0, WR = 0, AW = 0, WW = 0$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly);

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R1 comes along
- $AR = 1$, $WR = 0$, $AW = 0$, $WW = 0$

```
Reader() {  
    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();  
  
    AccessDbase(ReadOnly);  
  
    lock.Acquire();  
    AR--;  
    if (AR == 0 && WW > 0)  
        okToWrite.signal();  
    lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R1 comes along
- $AR = 1, WR = 0, AW = 0, WW = 0$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly) ;

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R1 comes along
- $AR = 1, WR = 0, AW = 0, WW = 0$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly),

```
    lock.Acquire();  
    AR--;  
    if (AR == 0 && WW > 0)  
        okToWrite.signal();  
    lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R2 comes along
- $AR = 1, WR = 0, AW = 0, WW = 0$

```
Reader() {  
    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();  
  
    AccessDbase(ReadOnly);  
  
    lock.Acquire();  
    AR--;  
    if (AR == 0 && WW > 0)  
        okToWrite.signal();  
    lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R2 comes along
- $AR = 1, WR = 0, AW = 0, WW = 0$

```
Reader() {  
    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();  
  
    AccessDbase(ReadOnly);  
  
    lock.Acquire();  
    AR--;  
    if (AR == 0 && WW > 0)  
        okToWrite.signal();  
    lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R2 comes along
- $AR = 2$, $WR = 0$, $AW = 0$, $WW = 0$

```
Reader() {
    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();

    AccessDbase(ReadOnly);

    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```


Simulation of Readers/Writers Solution

- R2 comes along
- $AR = 2$, $WR = 0$, $AW = 0$, $WW = 0$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly) ;

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R2 comes along
- $AR = 2, WR = 0, AW = 0, WW = 0$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly),

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
1  
}
```

Assume readers take a while to access database
Situation: Locks released, only AR is non-zero

Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- $AR = 2$, $WR = 0$, $AW = 0$, $WW = 0$

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

// Is it safe to write?
// No. Active users exist
// Sleep on cond var
// No longer waiting

AccessDbase(ReadWrite) ;

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

Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- $AR = 2$, $WR = 0$, $AW = 0$, $WW = 0$

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

// Is it safe to write?
// No. Active users exist
// Sleep on cond var
// No longer waiting

AccessDbase(ReadWrite);

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

Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- $AR = 2$, $WR = 0$, $AW = 0$, $WW = 1$

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

// Is it safe to write?
// No. Active users exist
// Sleep on cond var
// No longer waiting

AccessDbase(ReadWrite);

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

Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- $AR = 2$, $WR = 0$, $AW = 0$, $WW = 1$

```
Writer() {
    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++;
    lock.release();
}
```

AccessDbase(ReadWrite) ;

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

W1 cannot start because of readers, so goes to sleep

Simulation of Readers/Writers Solution

- R3 comes along (R1, R2 accessing dbase, W1 waiting)
- $AR = 2$, $WR = 0$, $AW = 0$, $WW = 1$

```
Reader() {  
    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();  
  
    AccessDbase(ReadOnly);  
  
    lock.Acquire();  
    AR--;  
    if (AR == 0 && WW > 0)  
        okToWrite.signal();  
    lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R3 comes along (R1, R2 accessing dbase, W1 waiting)
- $AR = 2$, $WR = 0$, $AW = 0$, $WW = 1$

```
Reader() {
    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();

    AccessDbase(ReadOnly);

    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```


Simulation of Readers/Writers Solution

- R3 comes along (R1, R2 accessing dbase, W1 waiting)
- $AR = 2$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly);

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R3 comes along (R1, R2 accessing dbase, W1 waiting)
- $AR = 2$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {  
    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();  
  
    AccessDbase(ReadOnly);  
  
    lock.Acquire();  
    AR--;
```

Status:

- R1 and R2 still reading
- W1 and R3 waiting on okToWrite and okToRead, respectively

Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1, R3 waiting)
- $AR = 2$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly);

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1, R3 waiting)
- $AR = 1$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {
    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();
}
```

AccessDbase(ReadOnly);

```
lock.Acquire();
AR--;
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();
}
```

Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1, R3 waiting)
- $AR = 1$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {
    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();

    AccessDbase(ReadOnly);

    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1, R3 waiting)
- $AR = 1$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {
    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();

    AccessDbase(ReadOnly);

    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- $AR = 1$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly) ;

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- AR = 0, WR = 1, AW = 0, WW = 1

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly);

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```


Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- $AR = 0$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {
    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();

    AccessDbase(ReadOnly);

    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- $AR = 0$, $WR = 1$, $AW = 0$, $WW = 1$

```
Reader() {  
    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();  
}
```

AccessDbase (ReadOnly) ;

```
    lock.Acquire();  
    AR--;  
    if (AR == 0 && WW > 0)  
        okToWrite.signal();  
    lock.Release();  
}
```

All reader finished, signal writer – note, R3 still waiting

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- $AR = 0$, $WR = 1$, $AW = 0$, $WW = 1$

```
Writer() {  
    lock.Acquire();  
    while ((AW + AR) > 0) {  
        WW++;  
        okToWrite.wait(&lock);  
        WW--;
```

```
// Is it safe to write?  
// No. Active users exist  
// Sleep on cond var  
// No longer waiting
```

Got signal
from R1

```
    ++;  
    lock.release();
```

```
AccessDbase(ReadWrite);
```

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

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- $AR = 0$, $WR = 1$, $AW = 0$, $WW = 0$

```
Writer() {
    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++;
    lock.release();
}
```

AccessDbase(ReadWrite);

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

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- $AR = 0$, $WR = 1$, $AW = 1$, $WW = 0$

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

// Is it safe to write?
// No. Active users exist
// Sleep on cond var
// No longer waiting

AccessDbase(ReadWrite);

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

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- $AR = 0$, $WR = 1$, $AW = 1$, $WW = 0$

```
Writer() {
    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++;
    lock.release();
}
```

AccessDbase(ReadWrite);

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

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- $AR = 0$, $WR = 1$, $AW = 0$, $WW = 0$

```
Writer() {
    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++;
    lock.release();
}
```

AccessDbase(ReadWrite);

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

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- $AR = 0$, $WR = 1$, $AW = 0$, $WW = 0$

```
Writer() {
    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++;
    lock.release();
}
```

AccessDbase(ReadWrite);

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


Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- $AR = 0$, $WR = 1$, $AW = 0$, $WW = 0$

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

// Is it safe to write?
// No. Active users exist
// Sleep on cond var
// No longer waiting

AccessDbase(ReadWrite);

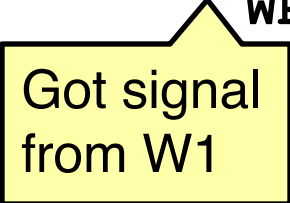
```
lock.Acquire();  
AW--;  
if (WW > 0) {  
    okToWrite.signal();  
} else if (WR > 0) {  
    okToRead.broadcast();  
}  
lock.Release();  
}
```

No waiting writer, signal reader R3

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- $AR = 0$, $WR = 1$, $AW = 0$, $WW = 0$

```
Reader() {  
    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  
    };                      // Now we are active!  
    lock.release();  
}
```



AccessDbase(ReadOnly);

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- $AR = 0$, $WR = 0$, $AW = 0$, $WW = 0$

```
Reader() {
    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();

    AccessDbase(ReadOnly);

    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- $AR = 0$, $WR = 0$, $AW = 0$, $WW = 0$

```
Reader() {
    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();
}
```

AccessDbase(ReadOnly)

```
lock.Acquire();
AR--;
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();
}
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- $AR = 0$, $WR = 0$, $AW = 0$, $WW = 0$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly);

```
lock.Acquire();  
AR--;  
if (AR == 0 && WW > 0)  
    okToWrite.signal();  
lock.Release();  
}
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- $AR = 0$, $WR = 0$, $AW = 0$, $WW = 0$

```
Reader() {  
    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();  
}
```

AccessDbase(ReadOnly);

```
    lock.Acquire();  
    AR--;  
    if (AR == 0 && WW > 0)  
        okToWrite.signal();  
    lock.Release();  
}
```

DONE!

Read/Writer Questions

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

```
// read-only
AccessDbase(
```

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

What if we
remove this
line?

```
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();
}
```

Read/Writer Questions

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

```
// read-only
AccessDbase(
```

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

What if we turn
signal to
broadcast?

```
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();
}
```


Read/Writer Questions

```
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?

Read/Writer Questions

```
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
- W1, R2 arrive while R1 still reading → W1 and R2 wait for R1 to finish
- Assume R1's signal is delivered to R2 (not W1)

Read/Writer Questions

```
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!

Implementing RWLock

- Let's wrap the code into a RWLock class

```
RWLock*  rlock;
```

```
rlock->startRead();  
// Read shared data  
rlock->doneRead();
```

```
rlock->startWrite();  
// Write shared data  
rlock->startRead();
```

Implementing RWLock

```
class RWLock {  
    Lock lock;  
    CV canRead;  
    CV canWrite;  
    int AR, AW, WR, WW;  
}
```

```
void RWLock::startRead() {  
    lock.acquire();  
    WR ++;  
    while ((AW + WW > 0)) {  
        canRead.Wait(&lock);  
    }  
    WR --;  
    AR ++;  
    lock.release();  
}
```

```
void RWLock::doneRead() {  
    lock.acquire();  
    AR --;  
    if ((AR == 0) && (WW > 0)) {  
        canWrite.signal();  
    }  
    lock.release();  
}
```

Implementing RWLock

```
class RWLock {  
    Lock lock;  
    CV canRead;  
    CV canWrite;  
    int AR, AW, WR, WW;  
}
```

```
void RWLock::startWrite() {  
    lock.acquire();  
    WW ++;  
    while ((AW + AR > 0)) {  
        canWrite.Wait(&lock);  
    }  
    WW --;  
    AW ++;  
    lock.release();  
}
```

```
void RWLock::doneWrite() {  
    lock.acquire();  
    AW --;  
    assert(AW == 0);  
    if (WW > 0) {  
        canWrite.signal();  
    }  
    else {  
        canRead.broadcast();  
    }  
    lock.release();  
}
```

Goals for Today

- Readers/Writers Lock
- **Deadlock**

Deadlock

- Deadlock (死锁): a cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some action.
- A simple case: mutually recursive locking

// Thread A

```
lock1.acquire() ;  
lock2.acquire() ;  
lock2.release() ;  
lock1.release() ;
```

// Thread B

```
lock2.acquire() ;  
lock1.acquire() ;  
lock1.release() ;  
lock2.release() ;
```


Deadlock

- Deadlock (死锁): a cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some action.
- Another example with 2 locks with 1 condition variable

// Thread A

```
lock1.acquire();  
lock2.acquire();  
while (need to wait) {  
    cv.wait(&lock2);  
}  
lock2.release();  
lock1.release();
```

// Thread B

```
lock1.acquire();  
lock2.acquire();  
cv.signal();  
lock2.release();  
lock1.release();
```

Deadlock

- Deadlock (死锁): a cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some action.
- Another example with 2 locks with 1 condition variable

// Thread A

```
lock1.acquire();  
lock2.acquire();  
while (need to wait) {  
    cv.wait(&lock2);  
}  
lock2.release();  
lock1.release();
```

// Thread B

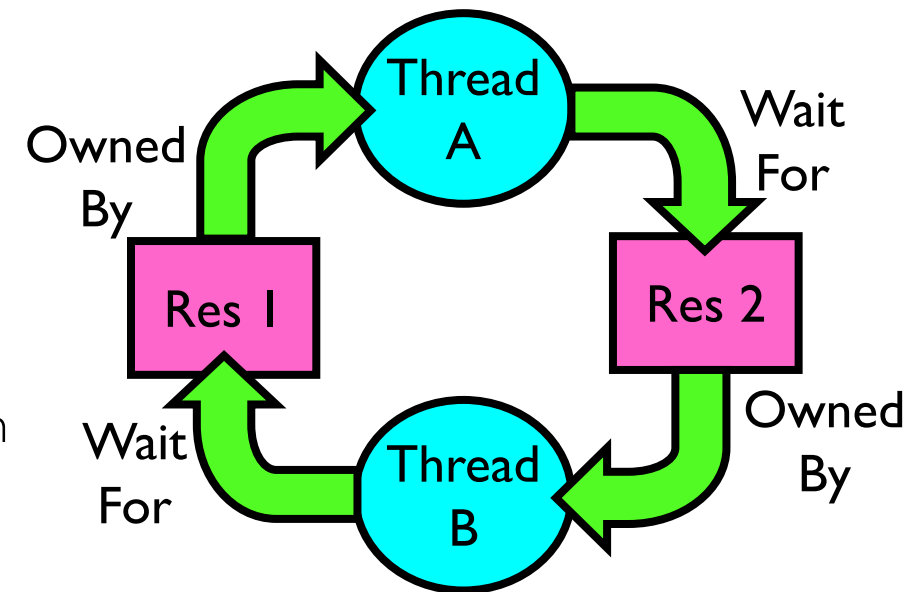
```
lock1.acquire();  
lock2.acquire();  
cv.signal();  
lock2.release();  
lock1.release();
```

Any deadlock?

Starvation vs Deadlock

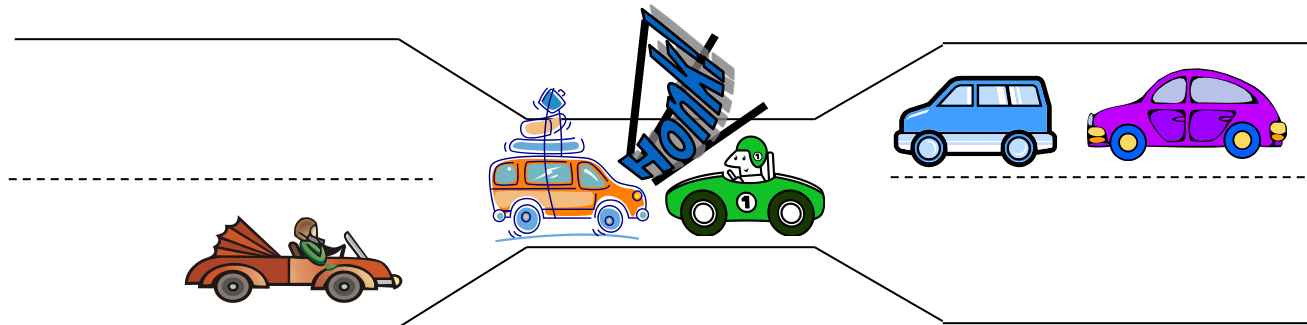
- Starvation vs. Deadlock

- 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 \Rightarrow Starvation but not vice versa
 - ❑ Starvation can end (but doesn't have to)
 - ❑ Deadlock can't end without external intervention



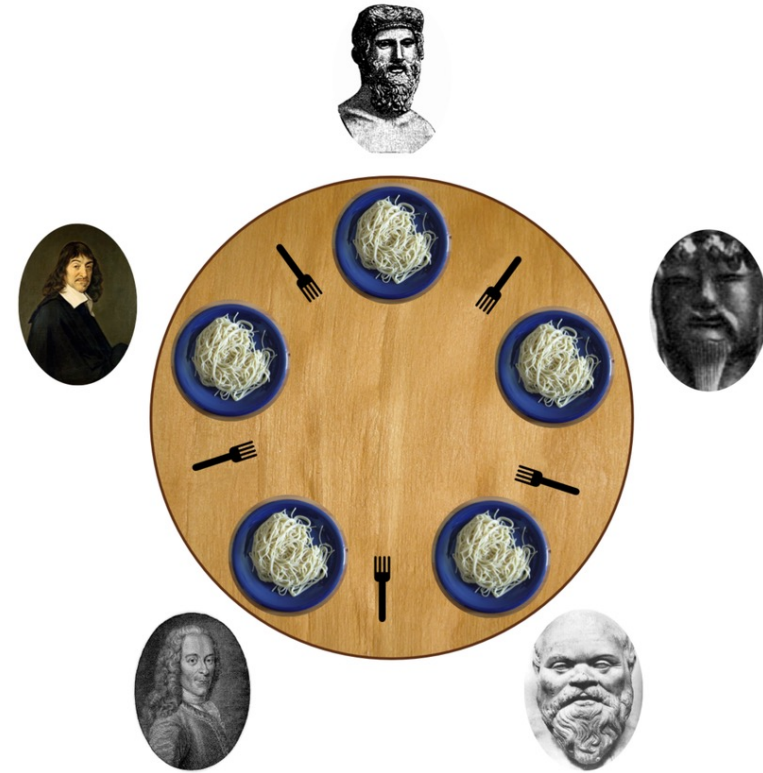
Bridge Crossing Example

- 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 \Rightarrow no one goes west



Dining Philosophers Problem

- Dining Philosophers Problem (哲学家进餐问题)
 - For example: 5 philosophers, 5 plate, and 5 chopsticks
 - When a philosopher thinking, he holds nothing
 - When a philosopher wants to eat, he first picks up the left chopstick, and then the right chopstick. After eating, he puts down both chopsticks.
 - Stuck when everyone holds the left chopstick
 - A general case of mutually recursive locking



Conditions for Deadlock

- Deadlock not always deterministic – Example 2 mutexes:

Thread A

x.P();

y.P();

y.V();

x.V();

Thread B

y.P();

x.P();

x.V();

y.V();

- Deadlock won't always happen with this code
 - ☐ Have to have exactly the right timing (“wrong” timing?)
 - ☐ So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant...
- 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

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 $\{T_1, \dots, T_n\}$ of waiting threads
 - T_i is waiting for a resource that is held by T_{i+1}

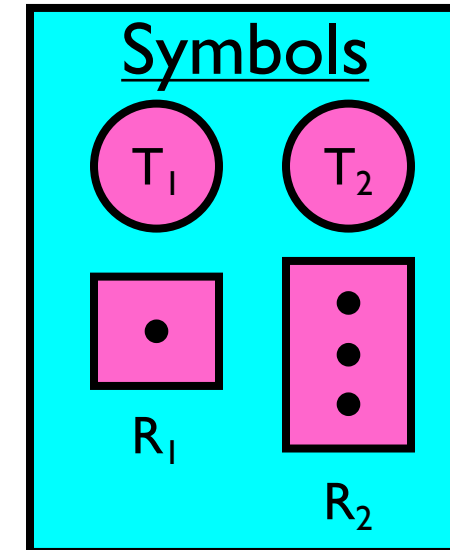
Four requirements for Deadlock

- Mutual exclusion
 - Only one thread at a time can use a resource.
 - Each chopstick can be held by a single philosopher at a time
- Hold and wait
 - Thread holding at least one resource is waiting to acquire additional resources held by other threads
 - When a philosopher needs to wait for a chopstick, he continues to hold onto any chopsticks he has already picked up
- No preemption
 - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
 - Once a philosopher picks up a chopstick, he does not release it until he is done eating.
- Circular wait
 - There exists a set $\{T_1, \dots, T_n\}$ of waiting threads
 - T_i is waiting for a resource that is held by T_{i+1}
 - Everyone is holding the left chopstick but waiting for the right one.

Resource-Allocation Graph

- System Model

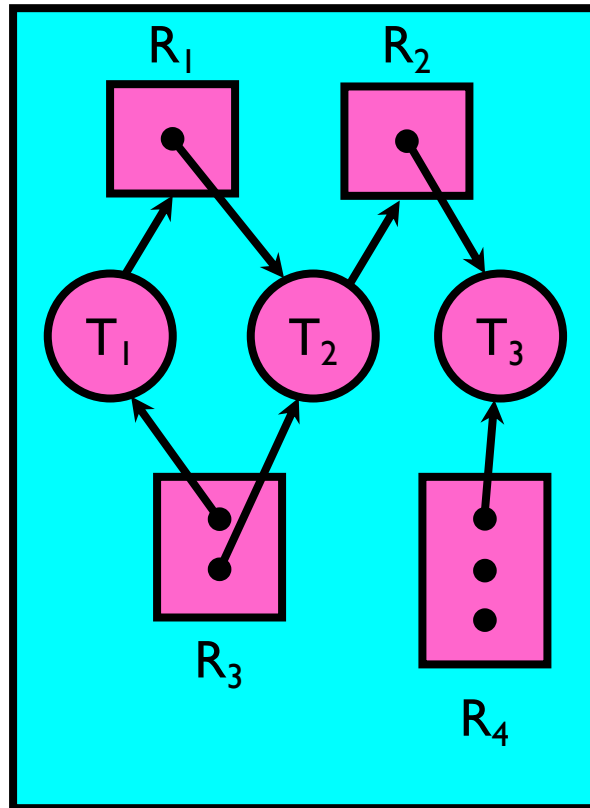
- A set of Threads T_1, T_2, \dots, T_n
- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each thread utilizes a resource as follows:
□ Request () / Use () / Release ()



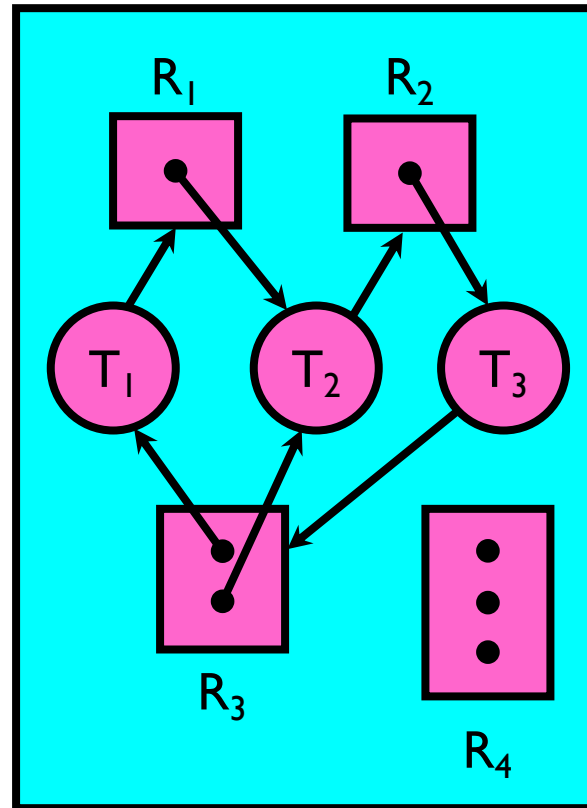
- Resource-Allocation Graph:

- V is partitioned into two types:
□ $T = \{T_1, T_2, \dots, T_n\}$, the set threads in the system.
□ $R = \{R_1, R_2, \dots, R_m\}$, the set of resource types in system
- request edge – directed edge $T_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow T_i$

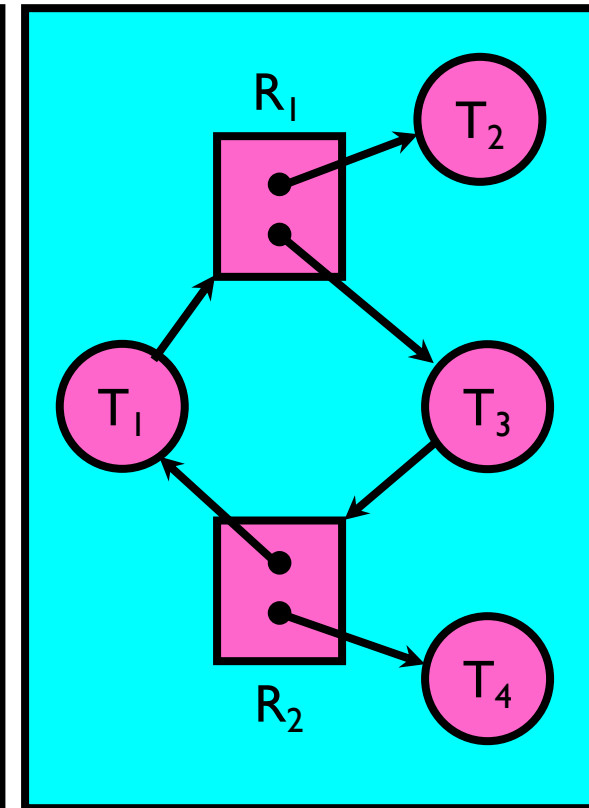
Resource Allocation Graph Examples



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 \Rightarrow 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

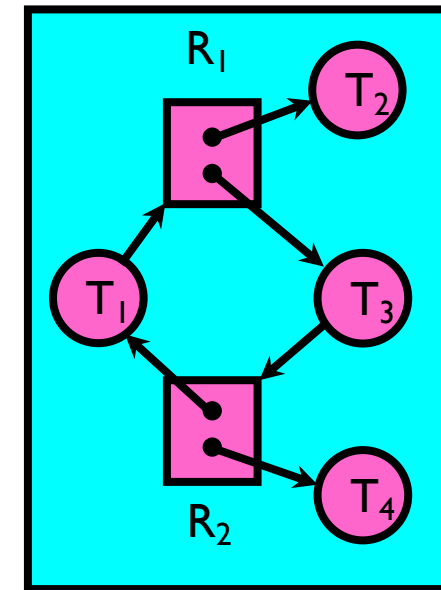
- See if tasks can eventually terminate on their own

```

[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ( $[Request_{node}] \leq [Avail]$ ) {
            remove node from UNFINISHED
             $[Avail] = [Avail] + [Alloc_{node}]$ 
            done = false
        }
    }
} until(done)

```

- Nodes left in **UNFINISHED** \Rightarrow deadlocked

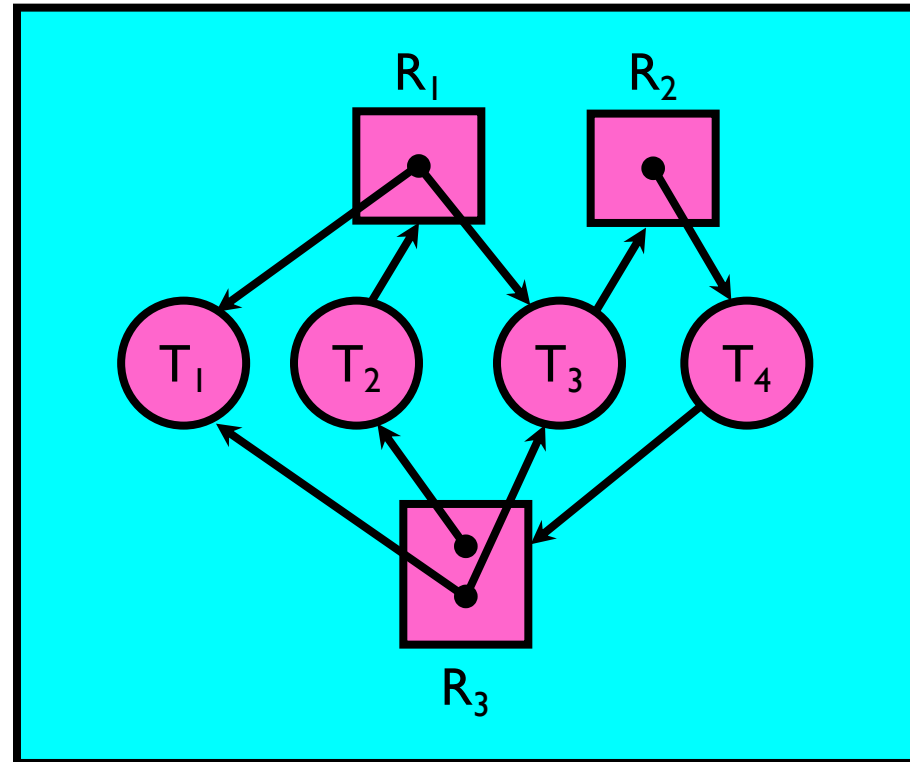


What to do when detect deadlock?

- Terminate thread, force it to give up resources
 - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
 - Shoot a Zax
 - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
 - Take away resources from thread temporarily
 - Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
 - Hit the rewind button, pretend last few minutes never happened
 - For bridge example, make one car roll backwards (may require others behind him)
 - Common technique in databases (transactions)
 - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

Resource Requests over Time

- Applications usually don't know exactly when/what they're going to request
- Resources are taken/released over time



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)
- No Sharing of resources (totally independent threads)
 - Often true (most things don't depend on each other)
 - Not very realistic in general (can't guarantee)
- 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.
 - Or straight to voicemail on cell phones
 - Technique used in Ethernet/some multiprocessor nets
 - ☐ Everyone speaks at once. On collision, back off and retry
 - Inefficient, since have to keep retrying
 - ☐ Consider: driving to San Francisco; when hit traffic jam, suddenly you're transported back home and told to 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:
 - ❑ 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; only one car on the Bay Bridge at a time
- 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...
 - ❑ Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Bankers Algorithm

- What if you don't know the order/amount of requests ahead of time?
- Must assume some worst-case “max” resource needed by each process
- Toward right idea:
 - State maximum resource needs in advance
 - Allow particular thread to proceed if:
 $(\text{available resources} - \# \text{requested}) \geq \max$
remaining that might be needed by any thread
 - Invariant: At all times, every request would succeed
□ Really conservative!

Banker's Algorithm

- Invariant: At all times, there exists some order of requests that would succeed.
- How to implement this?
 - Allocate resources dynamically
 - ❑ Evaluate each request and grant if some ordering of threads is still deadlock free afterward
 - Use deadlock detection algorithm presented earlier:
 - ❑ BUT: Assume each process needs "max" resources to finish

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ( $[Max_{node}] - [Alloc_{node}] \leq [Avail]$ ) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Allocnode]
            done = false
        }
    }
} until(done)
```

Each process might
need "max" resources
in order to finish

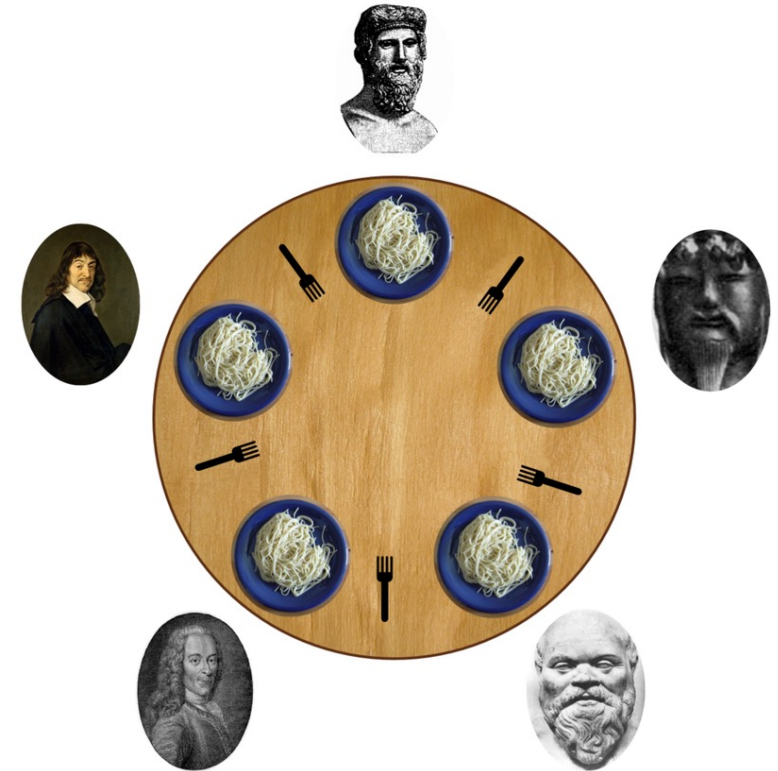
Banker's Algorithm: Key Properties

- 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
$$([Max_{node}] - [Alloc_{node}] \leq [Avail]) \text{ for } ([Request_{node}] \leq [Avail])$$

Grant request if result is deadlock free (conservative!)
 - ❑ Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \dots, T_n\}$ 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 philosopher? 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
 - ☐ ...



Deadlock Prevention – The Reality

- Deadlock Prevention is HARD
 - How many resources will each thread need?
 - How many total resources are there?
- Also Slow/Impractical
 - Matrix of resources/requirements could be big and dynamic
 - Re-evaluate on every request (even for small/non-contended)
 - Banker's algorithm assumes everyone asks for max
- REALITY
 - Most OSs don't bother
 - Programmers job to write deadlock-free programs (e.g. by ordering all resource requests).

Homework

- Modify our RWLock implementation to readers priority than writers
- Modify our RWLock implementation to use only one condition variable