Operating Systems
Lecture 12

Readers/Writers and Deadlock

Prof. Mengwei Xu

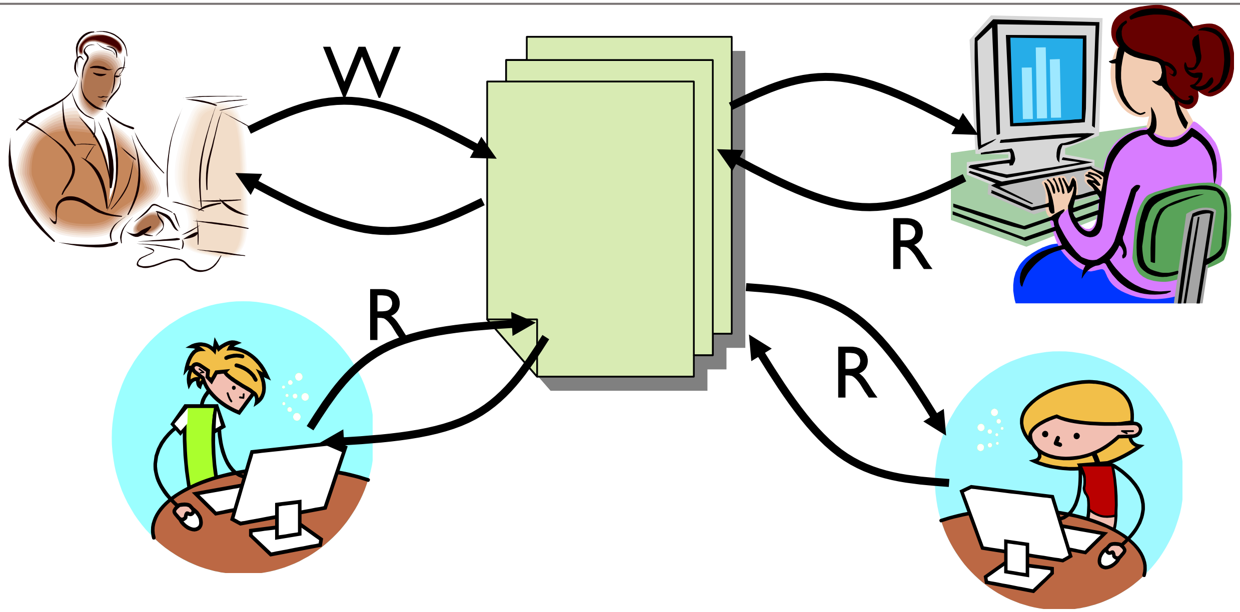
Goals for Today

- 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  AR--;                    // No longer active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                ss
  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 Give priority to writers?

Why broadcast() here instead of signal()?

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();
    }
}
```

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) {    // 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 comes along (R1 and R2 are still accessing dbase)
- AR = 2, WR = 0, 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 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();
  }
```

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) {   // Is it safe to write?
      WW++;                   // No. Active users exist
      okToWrite.wait(&lock);  // Sleep on cond var
      WW--;                   // No longer waiting
    }
      +;
     k.release();

    AccessDbase(ReadWrite);

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

Got signal from R1

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) {  // 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) {  // 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) {  // 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();
}
```

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!
      .release();

    AccessDbase(ReadOnly);

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

Got signal from W1

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
    lock.Acquire
    AR--;
    if (AR == 0 && W > 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* rwlock;

rwlock->startRead();
// Read shared data
rwlock->doneRead();

rwlock->startWrite();
// Write shared data
rwlock->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 and 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 and 1 condition variable

```
// Thread A

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

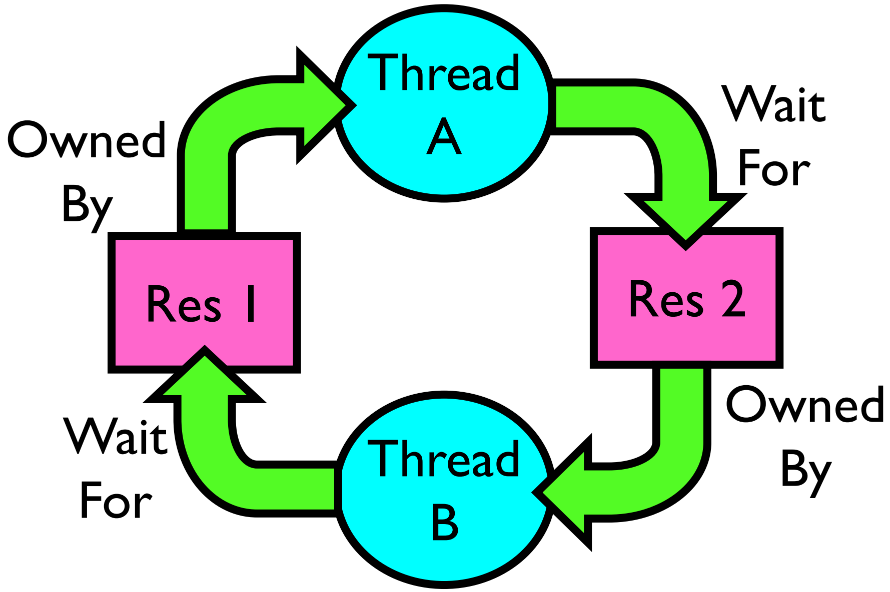
```
// Thread B

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

Any deadlock?

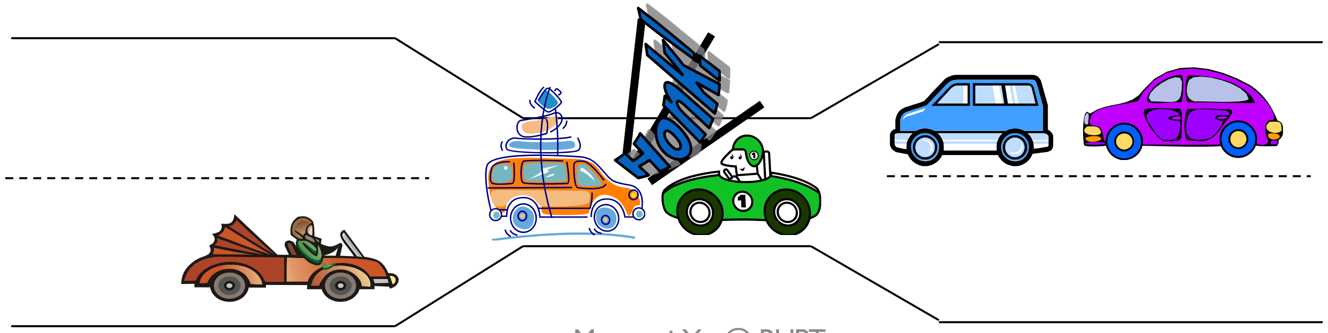
Starvation vs Deadlock

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



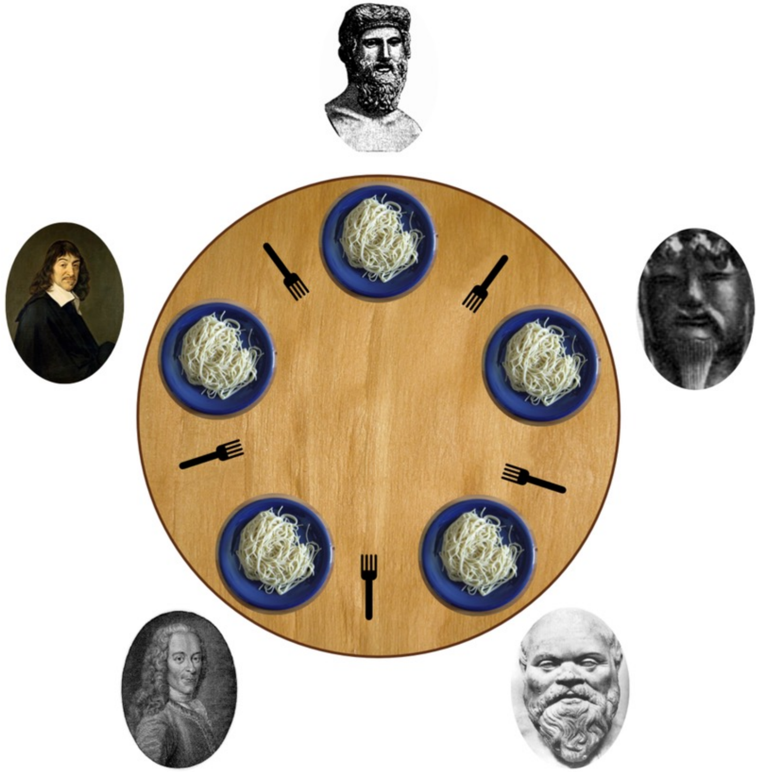
Bridge Crossing Example

- 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        Thread B
x.P();          y.P();
y.P();          x.P();
y.V();          x.V();
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 $\{T_1, \ldots, T_n\}$ 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 $\{T_1, \ldots, T_n\}$ 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.

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

Preventing Deadlocks

1. No circular wait
2. No hold-and-wait
3. No mutual exclusion
4. Smart scheduling
   - banking algorithm

Preventing Deadlocks

1. No circular wait
2. No hold-and-wait
3. No mutual exclusion
4. Smart scheduling
   - banking algorithm

Removing Circular Wait

- Just make sure all locks acquired in the same order!
  - Total ordering
  - Partial ordering
  - An excellent example: memory mapping code in Linux

```
/*
 * Lock ordering:
 *
 *  ->i_mmap_rwsem                  (truncate_pagecache)
 *    ->private_lock                (__free_pte->block_dirty_folio)
 *      ->swap_lock                 (exclusive_swap_page, others)
 *        ->i_pages lock
 *
 *  ->i_rwsem
 *    ->invalidate_lock             (acquired by fs in truncate path)
 *      ->i_mmap_rwsem              (truncate->unmap_mapping_range)
 *
 *  ->mmap_lock
 *    ->i_mmap_rwsem
 *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
 *        ->i_pages lock            (arch-dependent flush_dcache_mmap_lock)
 *
 *  ->mmap_lock
 *    ->invalidate_lock             (filemap_fault)
 *      ->lock_page                 (filemap_fault, access_process_vm)
 *
 *  ->i_rwsem                       (generic_perform_write)
 *    ->mmap_lock                   (fault_in_readable->do_page_fault)
 *
 *  bdi->wb.list_lock
 *    sb_lock                       (fs/fs-writeback.c)
 *    ->i_pages lock                (__sync_single_inode)
 *
 *  ->i_mmap_rwsem
 *    ->anon_vma.lock               (vma_merge)
 *
 *  ->anon_vma.lock
 *    ->page_table_lock or pte_lock     (anon_vma_prepare and various)
 *
 *  ->page_table_lock or pte_lock
 *    ->swap_lock                   (try_to_unmap_one)
 *    ->private_lock                (try_to_unmap_one)
 *    ->i_pages lock                (try_to_unmap_one)
 *    ->lruvec->lru_lock            (follow_page_mask->mark_page_accessed)
 *    ->lruvec->lru_lock            (check_pte_range->folio_isolate_lru)
 *    ->private_lock                (folio_remove_rmap_pte->set_page_dirty)
 *    ->i_pages lock                (folio_remove_rmap_pte->set_page_dirty)
 *    bdi.wb->list_lock             (folio_remove_rmap_pte->set_page_dirty)
 *    ->inode->i_lock               (folio_remove_rmap_pte->set_page_dirty)
 *    ->memcg->move_lock            (folio_remove_rmap_pte->folio_memcg_lock)
 *    bdi.wb->list_lock             (zap_pte_range->set_page_dirty)
 *    ->inode->i_lock               (zap_pte_range->set_page_dirty)
 *    ->private_lock                (zap_pte_range->block_dirty_folio)
 */
```

https://github.com/torvalds/linux/blob/master/mm/filemap.c

Removing Circular Wait

- Just make sure all locks acquired in the same order!
  - Total ordering
  - Partial ordering

```
func(mutex_t *m1, mutex_t *m2)
```

How to guarantee the ordering in func? Think about this case:

```
In Thread A: func(L1, L2)
In Thread B: func(L2, L1)
```

Removing Circular Wait

- Just make sure all locks acquired in the same order!
  - Total ordering
  - Partial ordering
- Enforce lock ordering by lock address

func(mutex_t *m1, mutex_t *m2)

How to guarantee the ordering in func? Think about this case:

In Thread A: func(L1, L2)
In Thread B: func(L2, L1)

```
if (m1 > m2) { // grab in high-to-low address order
  pthread_mutex_lock(m1);
  pthread_mutex_lock(m2);
} else {
  pthread_mutex_lock(m2);
  pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (not the same lock)
```

Preventing Deadlocks

1. No circular wait
    - Cons: needs careful design and programming from developers.
2. No hold-and-wait
3. No mutual exclusion
4. Smart scheduling
    - banking algorithm

Preventing Deadlocks

1. No circular wait
2. No hold-and-wait
3. No mutual exclusion
4. Smart scheduling
   - banking algorithm

Preventing Hold and Wait

- Just use another lock to lock the locks

```
pthread_mutex_lock(prevention); // begin acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

Preventing Deadlocks

1. No circular wait
2. No hold-and-wait
   - Cons: must know which locks will be used beforehand; concurrency decreased.
3. No mutual exclusion
4. Smart scheduling
   - banking algorithm

Preventing Deadlocks

1. No circular wait
2. No hold-and-wait
3. No mutual exclusion
4. Smart scheduling
   - banking algorithm

Preventing Mutual Exclusion

- Design lock-free (or wait-free) data structures and algorithms using powerful hardware instructions

```
int CompareAndSwap(int *address, int expected,
int new) {
    if (*address == expected) {
      *address = new;
      return 1; // success
    }
    return 0; // failure
}
```

Preventing Mutual Exclusion

- Using **CompareAndSwap** to implement "increment a value by n".

```
void AtomicIncrement(int *value, int n) {
  do {
    int old = *value;
  } while (CompareAndSwap(value, old, old + n)==0);
}
```

Preventing Mutual Exclusion

- Using **CompareAndSwap** to implement "insert an element to a list head".

```
// without deadlock prevention
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  n->next = head;
  head = n;
}
```

Preventing Mutual Exclusion

- Using **CompareAndSwap** to implement "insert an element to a list head".

```
// with deadlock prevention
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  do {
    n->next = head;
  } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

Preventing Deadlocks

1. No circular wait
2. No hold-and-wait
3. No mutual exclusion
   - Cons: too complicated; hardware support needed (possibly performance degradation).
4. Smart scheduling
   - banking algorithm

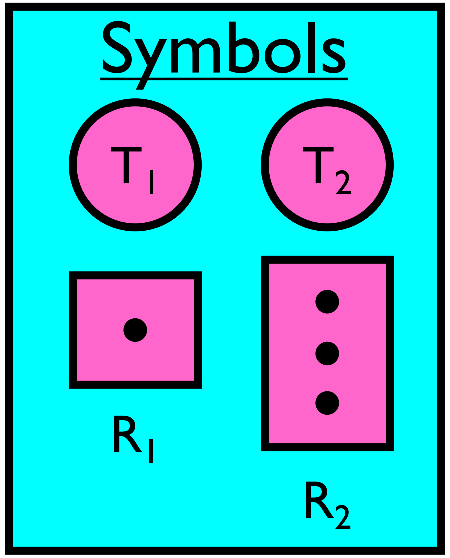
Preventing Deadlocks

1. No circular wait
2. No hold-and-wait
3. No mutual exclusion
4. Smart scheduling
   - banking algorithm

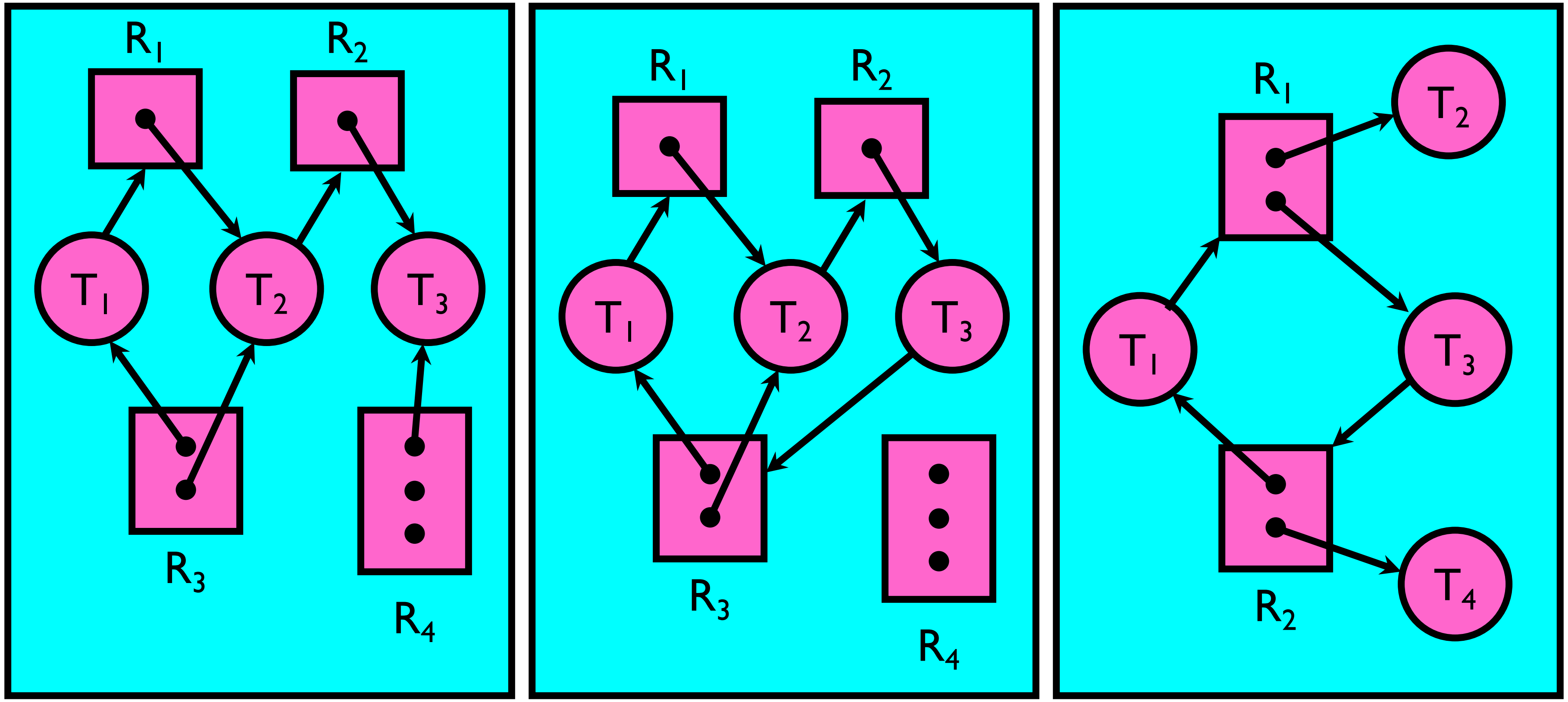
Resource-Allocation Graph

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

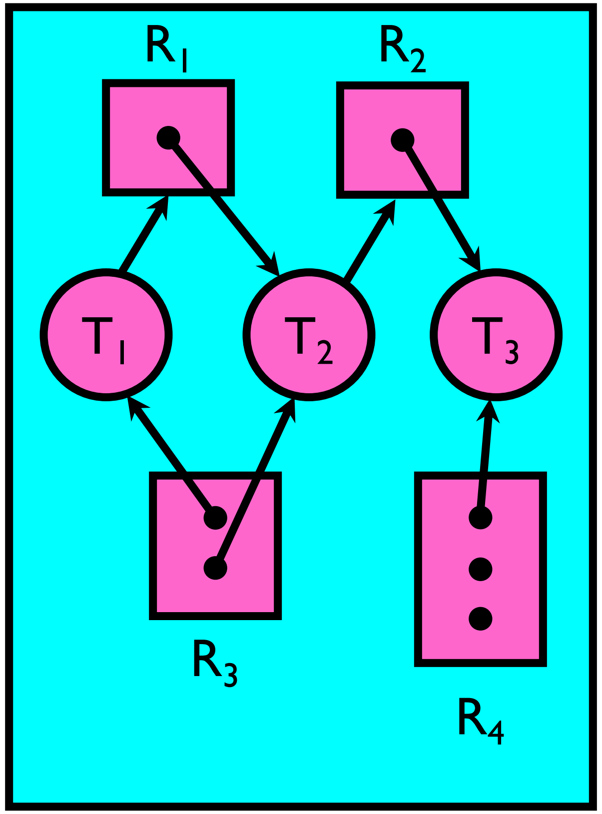
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances
  - Each thread utilizes a resource as follows:
    - ❑`Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - ❑$T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.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



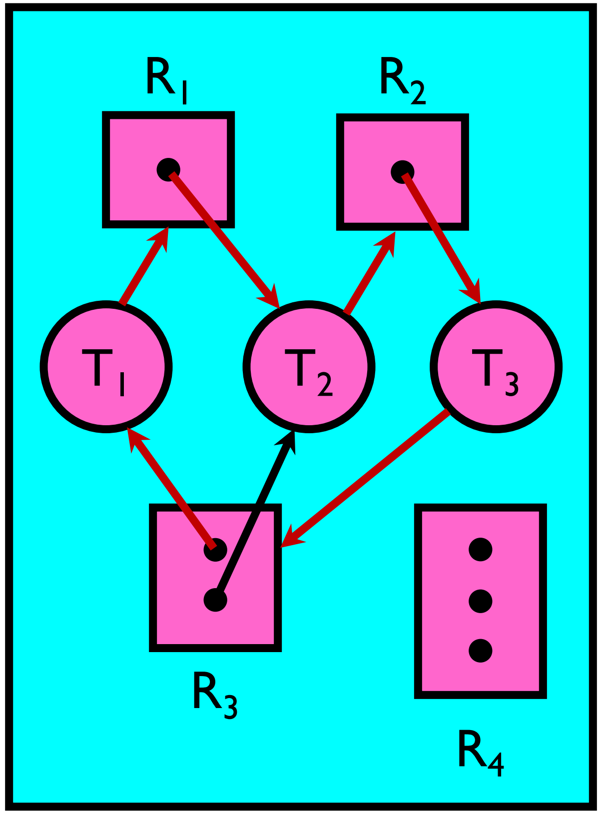
Resource Allocation Graph Examples



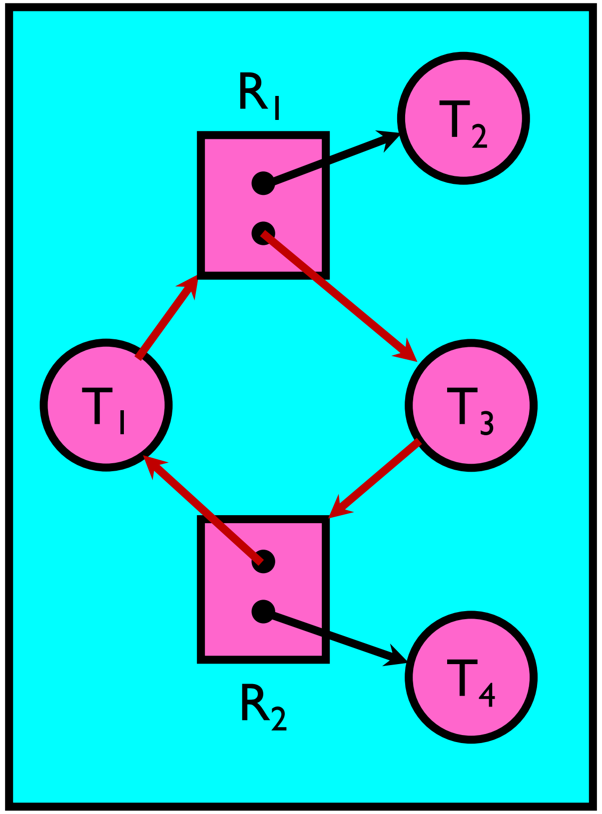
Resource Allocation Graph Examples



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

Deadlock Detection Algorithm

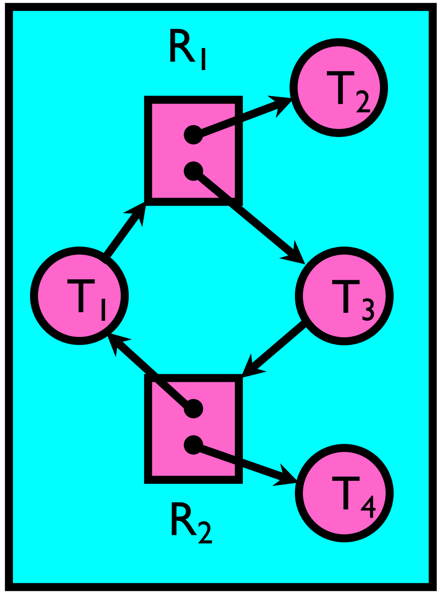
- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

`[FreeResources]:`	Current free resources each type
$[\texttt{Request}_X]$:	Current requests from thread X
$[\texttt{Alloc}_X]$:	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

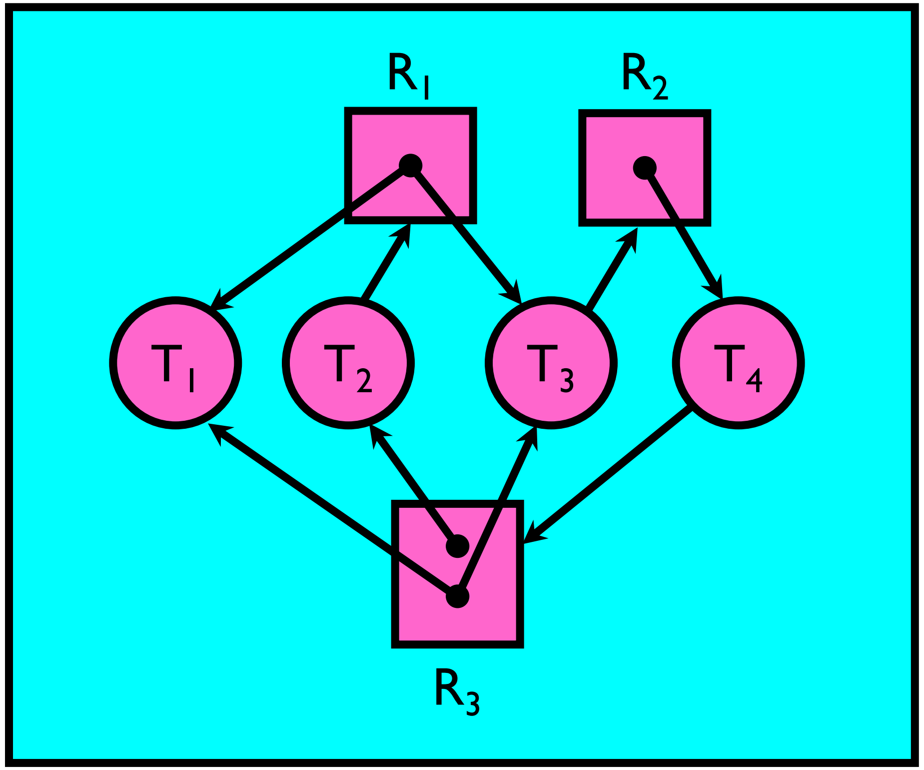


What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - 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



Bankers Algorithm (银行家算法)

- 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:

    (available resources - #requested) ≥

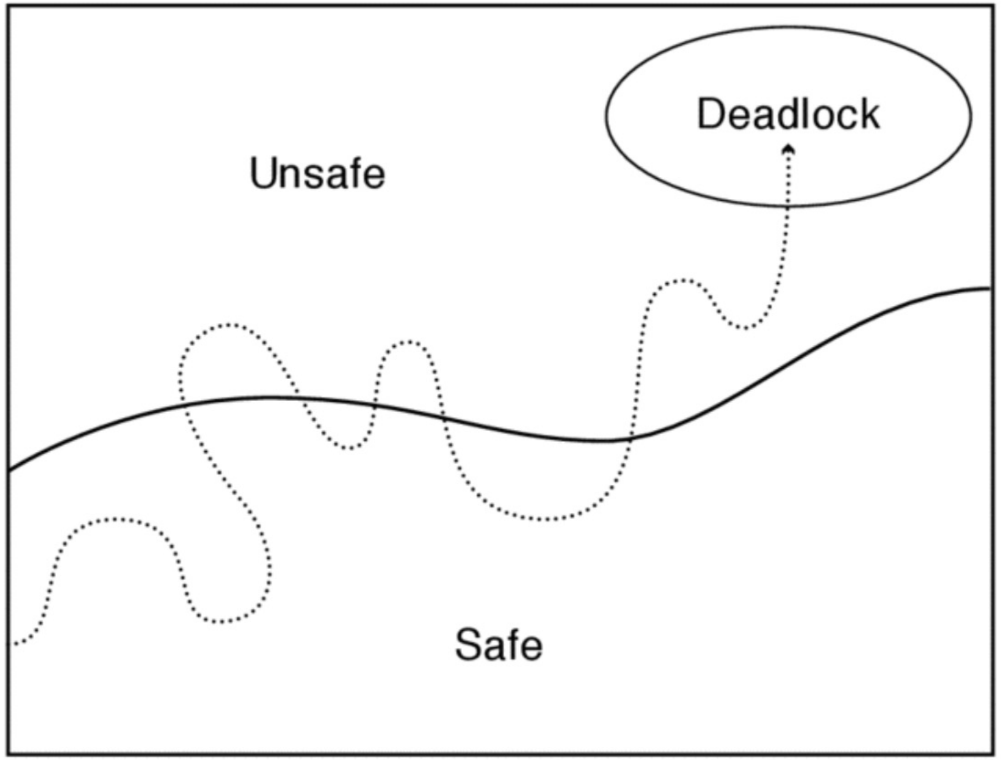
    max remaining that might be needed by any thread
  - Invariant: At all times, every request would succeed
    - ❑ Really conservative! Let's do something better.

Bankers Algorithm (银行家算法)

- Invariant: At all times, there exists some order of requests that would succeed.
- Key ideas
  - A thread states its maximum resource requirements, but acquires and releases resources incrementally as the thread executes.
  - The runtime system delays granting some requests to ensure that the system never deadlocks.

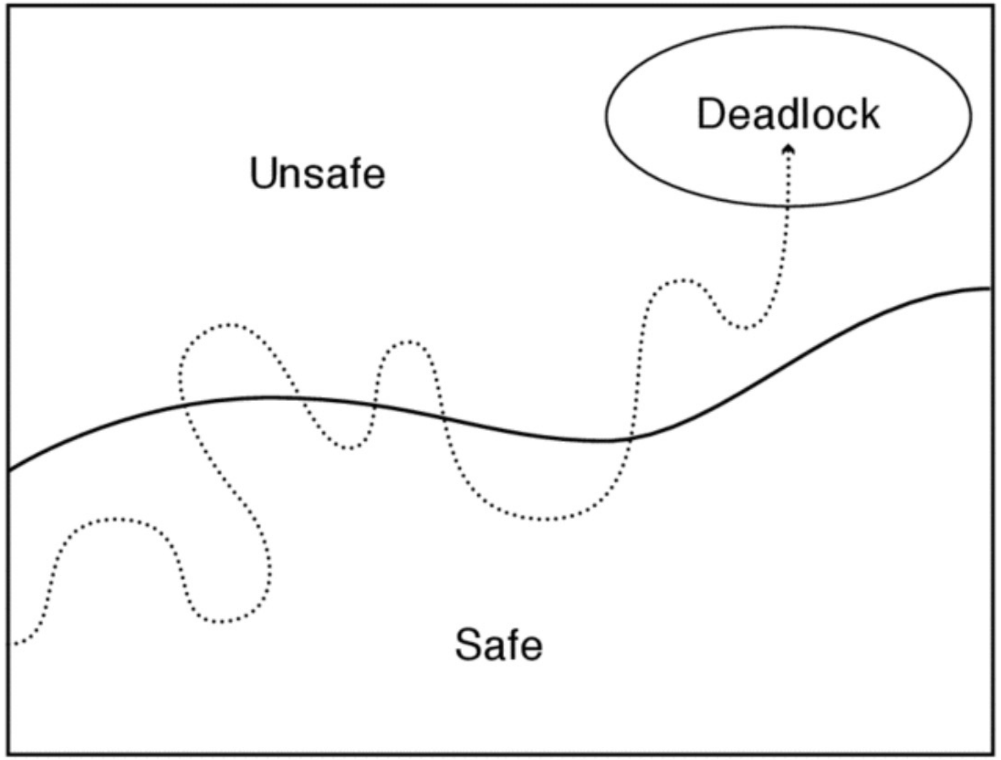
Safe State and Unsafe State

- **Safe state:** for any possible sequence of resource requests, there is at least one safe sequence of processing the requests that eventually succeeds in granting all pending and future requests.
- **Unsafe state:** there is at least one sequence of future resource requests that leads to deadlock no matter what processing order is tried.
- **Deadlocked state:** the system has at least one deadlock.



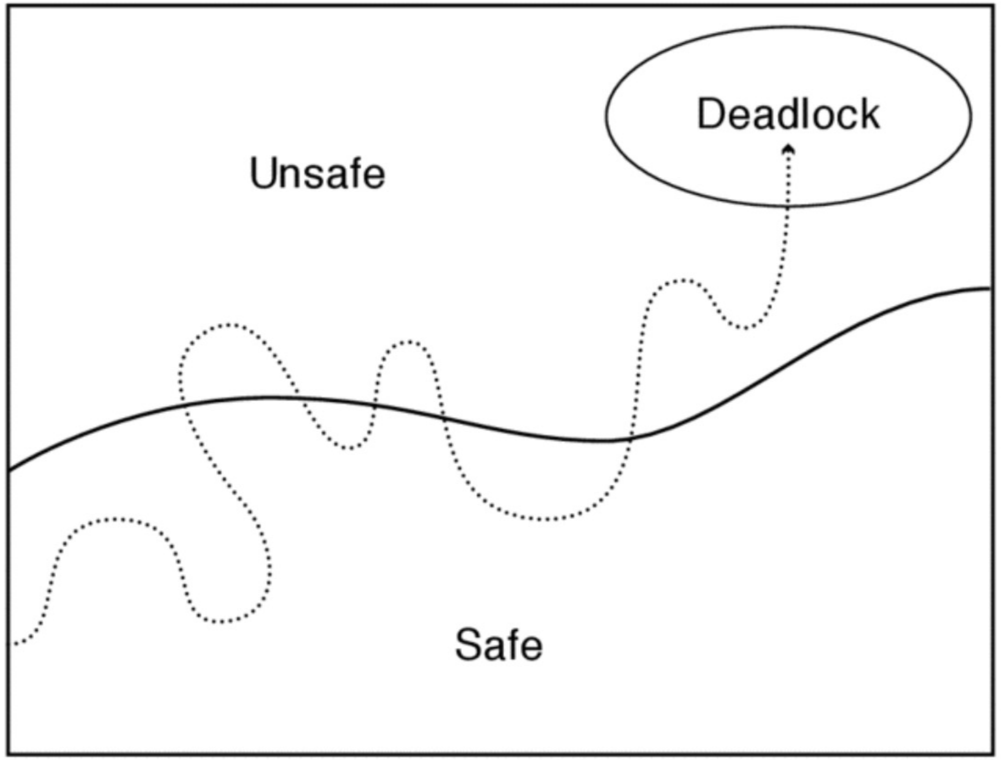
Safe State and Unsafe State

- **Safe state:** for any possible sequence of resource requests, there is at least one safe sequence of processing the requests that eventually succeeds in granting all pending and future requests.
  - *A system in a safe state controls its own destiny*: for any workload, it can avoid deadlock by delaying the processing of some requests.



Safe State and Unsafe State

- **Unsafe state:** there is at least one sequence of future resource requests that leads to deadlock no matter what processing order is tried.
    - An unsafe state does not always lead to deadlock
    - However, as long as the system remains in an unsafe state, a bad workload or unlucky scheduling of requests can force it to deadlock.



Bankers Algorithm (银行家算法)

- Invariant: At all times, there exists some order of requests that would succeed.
- The banker's algorithm delays any request that takes it from a safe to an unsafe state.

Bankers Algorithm (银行家算法)

- Delay a request that takes us into unsafe state.
- 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 ([Request_node] <= [Avail]) {
               remove node from UNFINISHED
               [Avail] = [Avail] + [Alloc_node]
               done = false
            }
         }
      } until(done)
```

Bankers Algorithm (银行家算法)

- Delay a request that takes us into unsafe state.
- 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] <= [Avail]) {
               remove node from UNFINISHED
               [Avail] = [Avail] + [Alloc_node]
               done = false
            }
         }
      } until(done)
```

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

Bankers Algorithm (银行家算法)

- Delay a request that takes us into unsafe state.
- 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
- 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..

- vs. "Require all before starting", the Banker's algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Bankers Algorithm (银行家算法)

- **EXAMPLE: Page allocation with the Banker's Algorithm.**
  - Suppose we have a system with 8 pages of memory and three processes: A, B, and C, which need 4, 5, and 5 pages to complete, respectively.
- They take turns requesting one page each, and the system grants requests in order

Bankers Algorithm (银行家算法)

- EXAMPLE: Page allocation with the Banker's Algorithm.
  - Suppose we have a system with 8 pages of memory and three processes: A, B, and C, which need 4, 5, and 5 pages to complete, respectively.
- They take turns requesting one page each, and the system grants requests in order

Oops! Deadlock!

Process	Allocation											
A	0	**1**	1	1	**2**	2	2	**3**	3	3	**wait**	**wait**
B	0	0	**1**	1	1	**2**	2	2	**3**	3	3	**wait**
C	0	0	0	**1**	1	1	**2**	2	2	**wait**	**wait**	**wait**
Total	0	1	2	3	4	5	6	7	8	8	8	8

Bankers Algorithm (银行家算法)

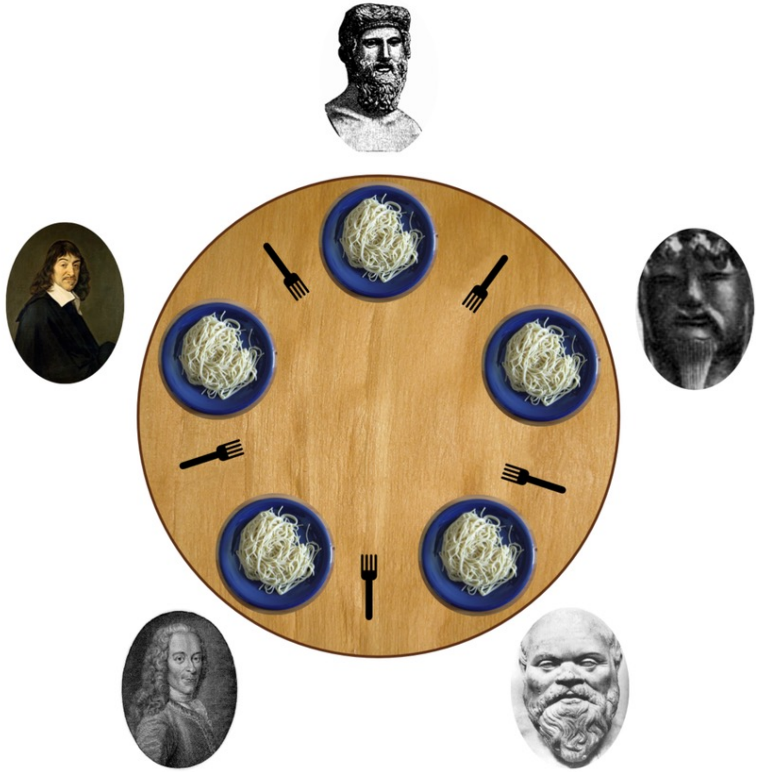
- EXAMPLE: Page allocation with the Banker's Algorithm.
  - Suppose we have a system with 8 pages of memory and three processes: A, B, and C, which need 4, 5, and 5 pages to complete, respectively.
- What if we use banker's algorithm?

Process	**Allocation**																		
A	0	**1**	1	1	**2**	2	2	**3**	3	3	**4**	**0**	0	0	0	0	0	0	0
B	0	0	**1**	1	1	**2**	2	2	**wait**	**wait**	**wait**	**wait**	**3**	**4**	4	**5**	**0**	0	0
C	0	0	0	**1**	1	1	**2**	2	2	**wait**	**wait**	**wait**	**3**	3	**wait**	**wait**	**4**	**5**	**0**
Total	0	1	2	3	4	5	6	7	7	7	8	4	6	7	7	8	4	5	0

Tasks successfully finished

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
    - ❑…



Preventing Deadlocks

1. No circular wait
2. No hold-and-wait
3. No mutual exclusion
4. Smart scheduling
   - banking algorithm
   - Cons: must know the entire set of tasks and their resource demands beforehand; concurrency decreased.
   - Only used in limited scenarios such as embedded system.

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)

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 OSes don't bother
  - Programmers job to write deadlock-free programs (e.g. by ordering all resource requests).

Homework

- Modify our RWLock implementation to use only one condition variable

- Implement Banker's Algorithm
  - Input-1: task number N, resource type number M;
  - Input-2: resource amount: for each type: R_i where i=1-M
  - Input-3: MAX resource for each task <$T_{i,j}$> where i=1-N and j=1-M;
  - Input-4: Sequence of resource request <$R_{i,j}$> where i=1-N and j=1-M
    - You can define your own way to generate this sequence

  - Test your algorithm with a large number of random sequences of resource request. Make sure deadlock never happens!