CS162
Operating Systems and Systems Programming
Lecture 9

Synchronization, Readers/Writers example, Scheduling

February 14th, 2018

Profs. Anthony D. Joseph and Jonathan Ragan-Kelley
http://cs162.eecs.Berkeley.edu

Complete Monitor Example (with cond. variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();           // Get Lock
   queue.enqueue(item);      // Add item
   dataready.signal();       // Signal any waiters
   lock.Release();           // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();           // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();   // Get next item
   lock.Release();           // Release Lock
   return(item);
}
```


Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

```
while (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
```

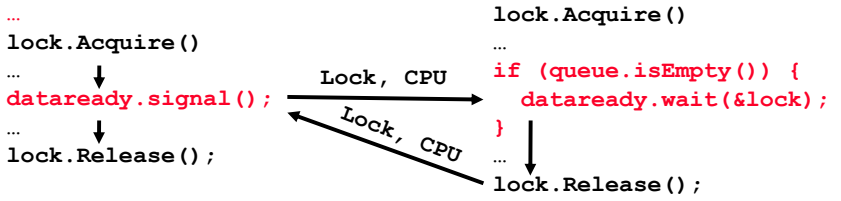
  - Why didn't we do this?

```
if (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
```

- Answer: depends on the type of scheduling
  - Hoare-style
  - Mesa-style

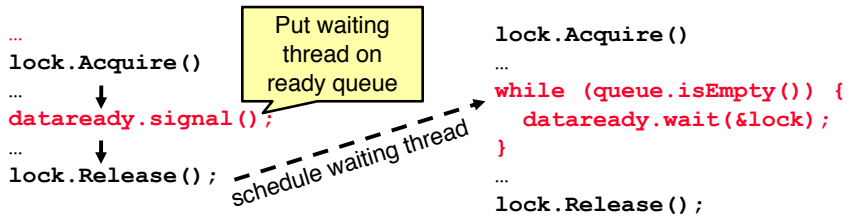
Hoare monitors

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
- Most textbooks



Mesa monitors

- Signaler keeps lock and processor
- Waiter placed on ready queue with no special priority
- Practically, need to check condition again after wait
- Most real operating systems

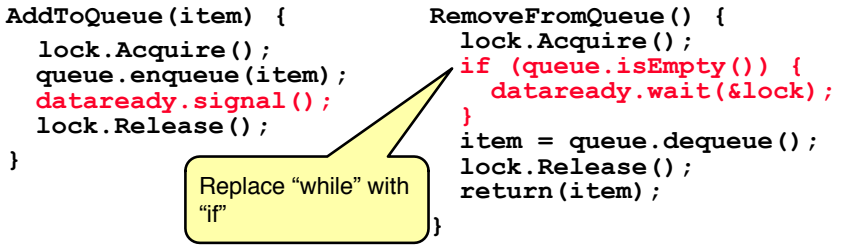


Mesa Monitor: Why "while()"?

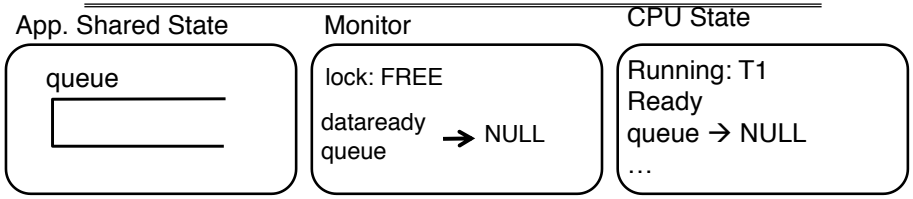
- Why do we use "while()" instead of "if() with Mesa monitors?
  - Example illustrating what happens if we use "if()", e.g.,

```
if (queue.isEmpty()) {
  dataready.wait(&lock); // If nothing, sleep
}
```

- We'll use the synchronized (infinite) queue example



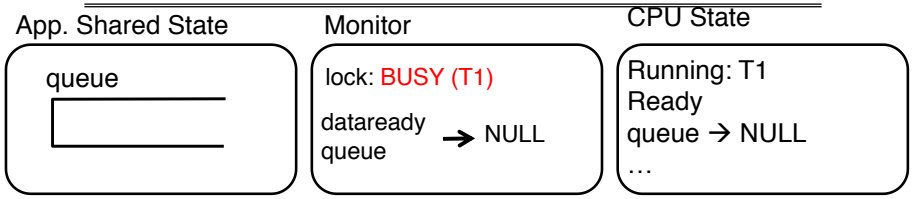
Mesa Monitor: Why "while()"?



T1 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

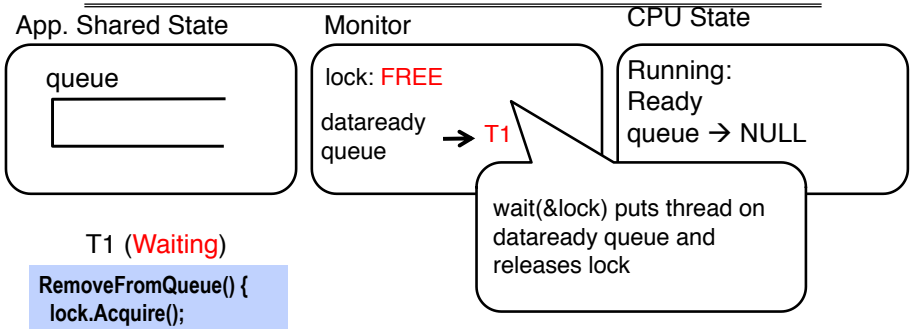

Mesa Monitor: Why "while()"?



T1 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

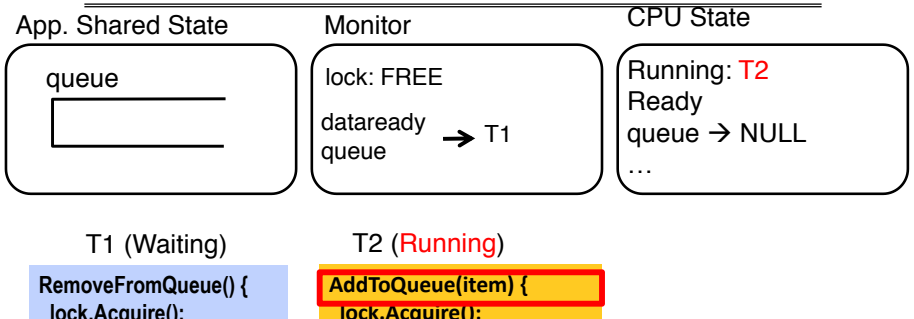
Mesa Monitor: Why "while()"?



T1 (Waiting)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```


Mesa Monitor: Why "while()"?



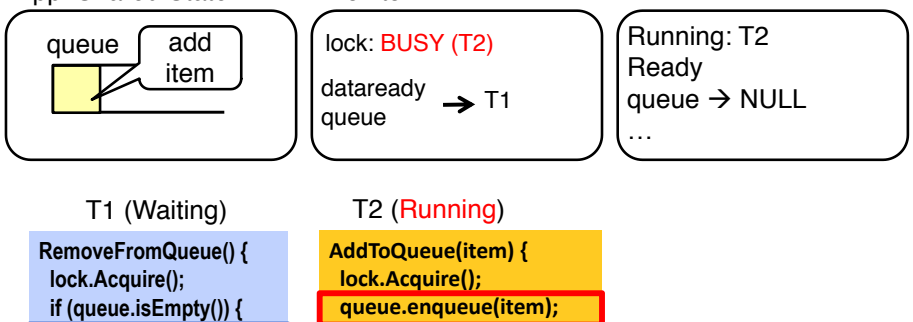
T1 (Waiting)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```


Mesa Monitor: Why "while()"?



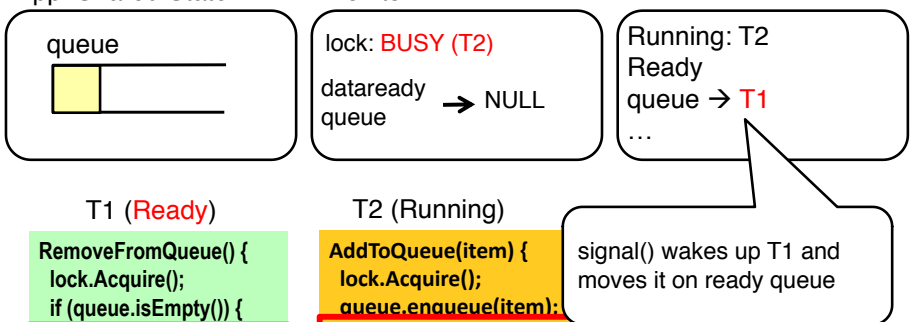
T1 (Waiting)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```


Mesa Monitor: Why "while()"?



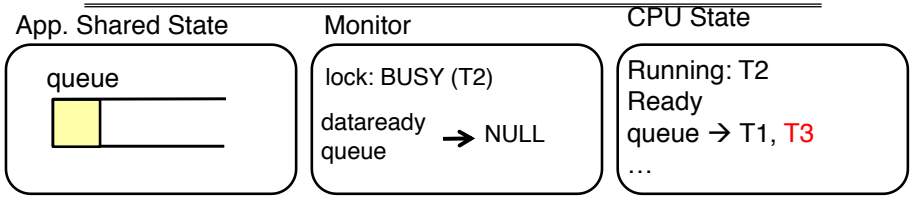
T1 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```

Mesa Monitor: Why "while()"?



T1 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```

T3 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```


Mesa Monitor: Why "while()"?

T1 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

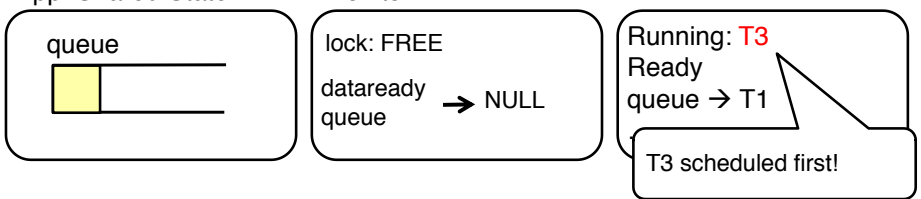
T2 (Terminate)

```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```

T3 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```


Mesa Monitor: Why "while()"?



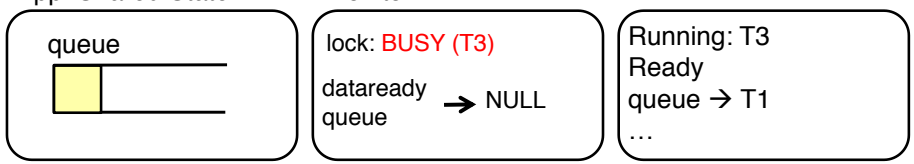
T1 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```


Mesa Monitor: Why "while()"?



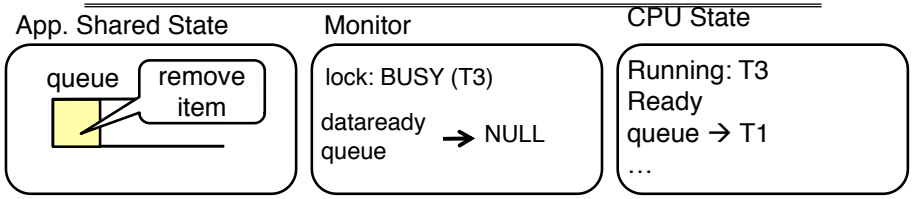
T1 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

Mesa Monitor: Why "while()"?



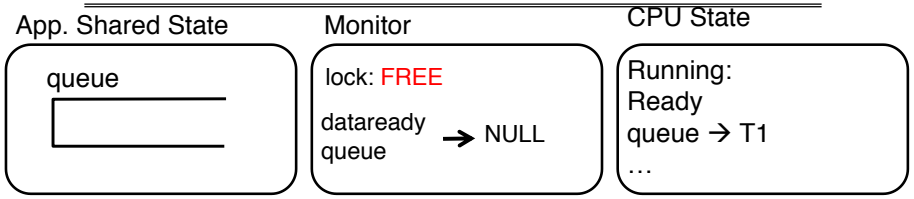
T1 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```


Mesa Monitor: Why "while()"?



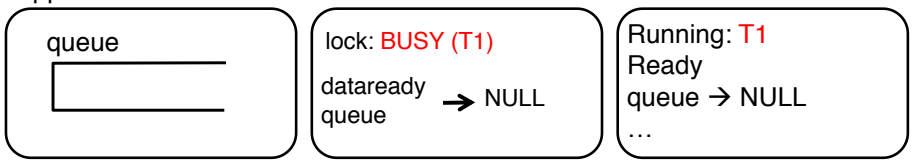
T1 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Finished)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

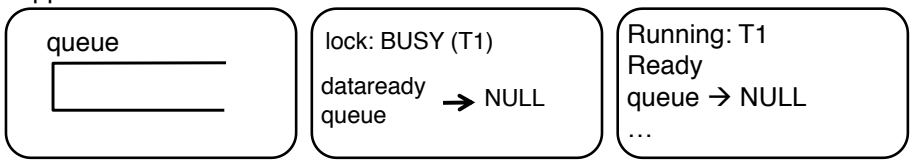

Mesa Monitor: Why "while()"?



T1 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```


Mesa Monitor: Why "while()"?



T1 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

ERROR: Nothing in the queue!

Mesa Monitor: Why "while()"?

App. Shared State | Monitor | CPU State

queue

lock: BUSY (T1)
datareadyqueue → NULL

Running: T1
Ready
queue → NULL
…

T1 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  while (queue.isEmpty())
  {
    dataready.wait(&lock);
  }
  item = que
  lock.Relea
  return(item
}
```

Replace "if" with "while"

Mesa Monitor: Why "while()"?

App. Shared State | Monitor | CPU State

queue

lock: BUSY (T1)
datareadyqueue → NULL

Running: T1
Ready
queue → NULL
…

T1 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  while (queue.isEmpty())
  {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

Check again if empty!

Mesa Monitor: Why "while()"?

App. Shared State | Monitor | CPU State

queue

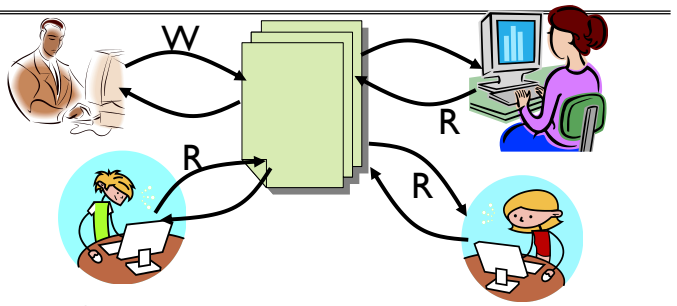
lock: FREE
datareadyqueue → T1

Running: T1
Ready
queue → NULL
…

T1 (Waiting)

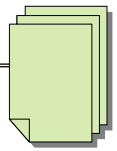
```
RemoveFromQueue() {
  lock.Acquire();
  while (queue.isEmpty())
  {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```


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 read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                    // No longer active
  if (WW > 0){             // Give priority to writers
    okToWrite.signal();    // Wake up one writer
  } else if (WR > 0) {     // Otherwise, wake reader
    okToRead.broadcast();  // Wake all readers
  }
  lock.Release();
}
```


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();
    l
  }
```

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
    }
    AW++;
    lock.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
    }
    AR++;                    // Now we are active!
    lock.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();
  }
```

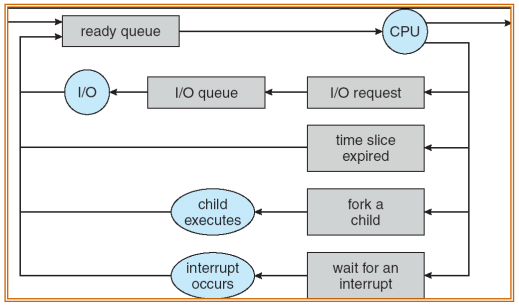
Need to change to broadcast!

Administrivia

- Project 1 Design Document due today 11:59PM
- Midterm on Wednesday 2/28 6:30-8:30PM
  - Room assignments TBD
- Closed book, no calculators, one double-side letter-sized page of handwritten notes
  - Covers Lectures 1-10, readings, homework 1, and project 1

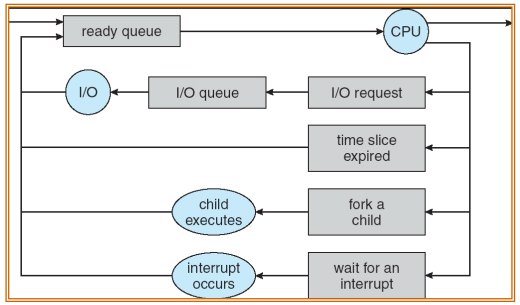
BREAK

Recall: CPU Scheduling



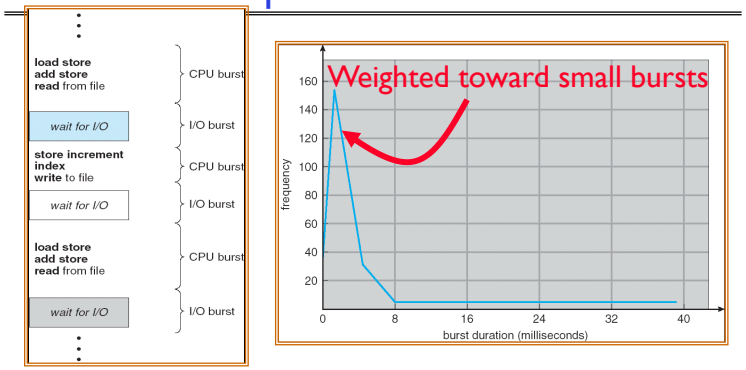
- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.

Recall: CPU Scheduling (Cont.)



- Question: How does OS decide which thread to dequeue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access to resources from moment to moment

Assumption – CPU Bursts



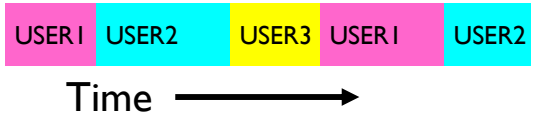
- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent

Scheduling Assumptions (Cont.)

- Clearly, unrealistic but they simplify the problem
  - For instance: is "fair" about fairness among users or programs?
    - » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system



Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time tasks: Must meet deadlines imposed by World

Scheduling Policy Goals/Criteria (Cont.)

- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)

Scheduling Policy Goals/Criteria (Cont.)

- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

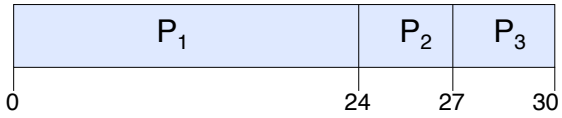
First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks

- Example:

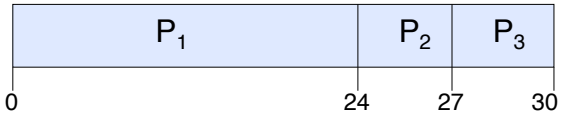
Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The Gantt Chart for the schedule is:



FCFS Scheduling (Cont.)

- Example continued:



  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1 Now, we have:

  - Waiting time for P_1 = 6; P_2 = 0, P_3 = 3
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average Completion time: (3 + 6 + 30)/3 = 13
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - » Safeway: Getting milk, always stuck behind cart full of small items

Round Robin (RR) Scheduling

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
- Round Robin Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - *n* processes in ready queue and time quantum is *q* ⇒
    - » Each process gets 1/*n* of the CPU time
    - » In chunks of at most *q* time units
    - » No process waits more than (*n*-1)*q* time units

RR Scheduling (Cont.)

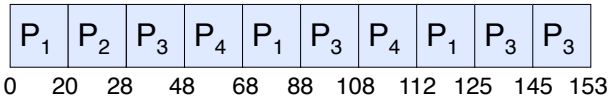
- Performance
  - *q* large ⇒ FCFS
  - *q* small ⇒ Interleaved (really small ⇒ hyperthreading?)
  - *q* must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

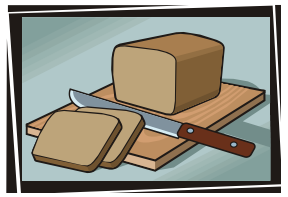
  - The Gantt chart is:



  - Waiting time for P_1=(68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  - Average waiting time = (72+20+85+88)/4=66¼
  - Average completion time = (125+28+153+112)/4 = 104½
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    - » Response time suffers
  - What if infinite (∞)?
    - » Get back FIFO
  - What if time slice too small?
    - » Throughput suffers!



- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - » Worked ok when UNIX was used by one or two people.
    - » What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput:
    - » Typical time slice today is between 10ms – 100ms
    - » Typical context-switching overhead is 0.1ms – 1ms
    - » Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:

10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs start at the same time

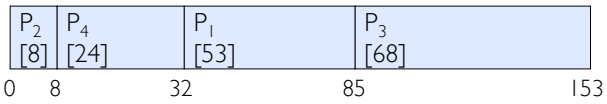
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time is much worse under RR!
    - » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

Best FCFS:



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Synchronization Summary

- Monitors: A lock plus zero or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, `Broadcast()`
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs