Processes and Monitors in Mesa (Lecture 6, cs262a)

Ion Stoica,
UC Berkeley
September 14, 2016

(based on presentation from John Kubiatowicz, UC Berkeley)

Discussion

What are good APIs?

- Unix file system
- SQL
- IP

Why were these APIs successful?

Today’s Lecture

Mesa Motivation

Putting theory to practice – building Pilot OS

Focus of this paper:

- Lightweight processes (threads in today's terminology), and
- how they synchronize with each other

Mesa History

2nd system Xerox Star – followed Alto

- Introduced in 1981
- 384 KB RAM, 10-40MB 8” HDD
- Ethernet connectivity
- Bundled as a network (3-4 Star computers along file server, printer) $100K+
- Only ~ 25K sold

Advent of things like server machines and networking introduced applications that are heavy users of concurrency

Mesa History (cont'd)

Chose to build a single address space system:

- Single user system, so protection not an issue
- Safety was to come from the language
- Wanted global resource sharing

Large system, many programmers, many applications:

- Module-based programming with information hiding

Clean sheet design:

- Can integrate hardware, runtime software, and language with each other

Java language considers Mesa to be a predecessor

Programming Models for IPC

Two Inter-Process Communication models:

- Shared memory (monitors) vs.
- Message passing

Needham & Lauer claimed the two models are duals of each other

- message <-> process
- process <-> monitor
- send/reply <-> call/return

Mesa developers chose shared memory model because they thought they could more naturally fit it into Mesa as a language construct

How to Synchronize Processes?

Non-preemptive scheduler: results in very delicate systems (Why?)

- Have to know whether or not a yield might be called for *every* procedure you call – this violates information hiding
- Prohibits multiprocessor systems
- Need a separate preemptive mechanism for I/O anyway
- Can’t do multiprogramming across page faults

How to Synchronize Processes? (cont'd)

Simple locking (*e.g.*, semaphores):

- Too little structuring discipline, *e.g.*, no guarantee that locks will be released on every code path
- Wanted something that could be integrated into a Mesa language construct

Chose preemptive scheduling of lightweight processes and monitors

Lightweight Processes (LWPs)

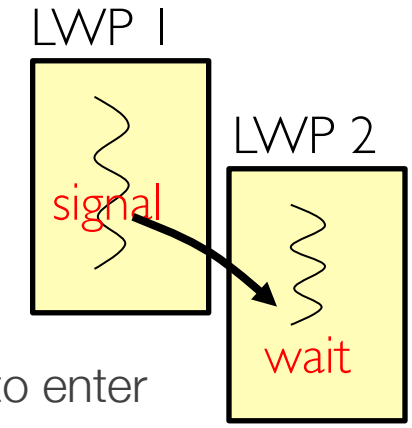
Easy forking and synchronization

Shared address space

Fast performance for creation, switching, and synchronization; low storage overhead

Today we call LWPs, “threads”

Recap: Synchronization Goals



Mutual exclusion:

- Arbitrate access to critical section (e.g., shared data)
- Only a single LWP in critical section at a given time
  - If one LWP in critical section → all other LWPs that want to enter the critical section need to wait

Scheduling constraint:

- A LWP waiting for an event to happen in another thread

Wait instruction:

- Don't want busy-waiting, so sleep()
- Waiting LWPs are woken up when the condition they are waiting on becomes FALSE

Recap: Synchronization Primitives

Locks: Implement mutual exclusion

- `Lock.Acquire()`: acquire lock before entering critical section; wait if lock not free
- `Lock.Release()`: release lock after leaving critical section; wake up threads waiting for lock

Semaphores: Like integers with restricted interface

- `P()`: Wait if zero; decrement when becomes non-zero
- `V()`: Increment and wake a sleeping task (if exists)
- Use a semaphore for each scheduling constraint and mutex
- Decided “exert too little structuring discipline on concurrent programs”

Recap: Synchronization Primitives

Monitors: A lock plus one or more condition variables

- Condition variable: a queue of LWPs waiting inside critical section for an event to happen
- Use condition variables to implement scheduling constraints
- Three Operations: `Wait()`, `Signal()`, and `Broadcast()`

Recap: Monitors

Monitors represent the logic of the program

- Wait if necessary
- Signal when change something so any waiting LWPs can proceed

Basic structure of monitor-based program:

```
lock.Acquire()
while (need to wait) {
   condvar.wait(&lock);
}
lock.Release()

do something so no need to wait

lock.Acquire()
condvar.signal();
lock.Release()
```

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

Check and/or update state variables

Mesa Monitors

Monitor lock (for synchronization)

Condition variable (for scheduling) – when to wait

Tied to module structure of the language – makes it clear what's being monitored

Language automatically acquires and releases the lock

Tied to a particular invariant, which helps users think about the program

- Invariant holds on entry and must be maintained before exit or wait

Design Choices and Implementation Issues

Module: packages a collection of related procedures and protect their private data from external access

Three types of procedures in a monitor module:

- Entry (acquires and releases lock)
  - Typically allocates, initializes, or free data (e.g., constructor, destructor)
- Internal (no locking done): can't be called from outside the module
  - (Similar to private methods in a class)
- External (no locking done): externally callable. Why is this useful?
  - (Similar to public methods in a class)
  - Allows grouping of related things into a module
  - Allows doing some of the work outside the monitor lock

Design Choices and Implementation Issues

Choices for notify semantics:

- (Hoare monitors) Immediately cede CPU and lock to waking process
  - Causes many context switches but why would this approach be desirable? (Waiting process knows the condition it was waiting on is guaranteed to hold)
  - Also, doesn't work in the presence of priorities
- (Mesa monitors) Notifier keeps lock, wakes process with no guarantees => waking process must recheck its condition

Mesa Monitor: Why “while()”?

Why do we use “while()” instead of “if() with Mesa monitors?
We’ll use the synchronized (infinite) queue example

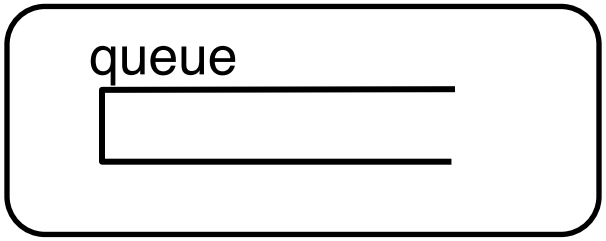
```
AddToQueue(item) {
   lock.Acquire();
   queue.enqueue(item);
   dataready.signal();
   lock.Release
}
```

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

Replace “while” with “if”

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: FREE

dataready → NULL

queue

CPU State

Running: T1

Ready

queue → NULL

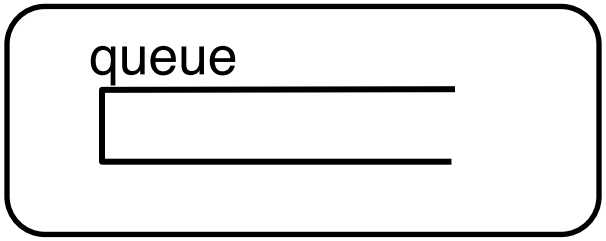
…

T1 (Running)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T1)

dataready → NULL

queue

CPU State

Running: T1

Ready

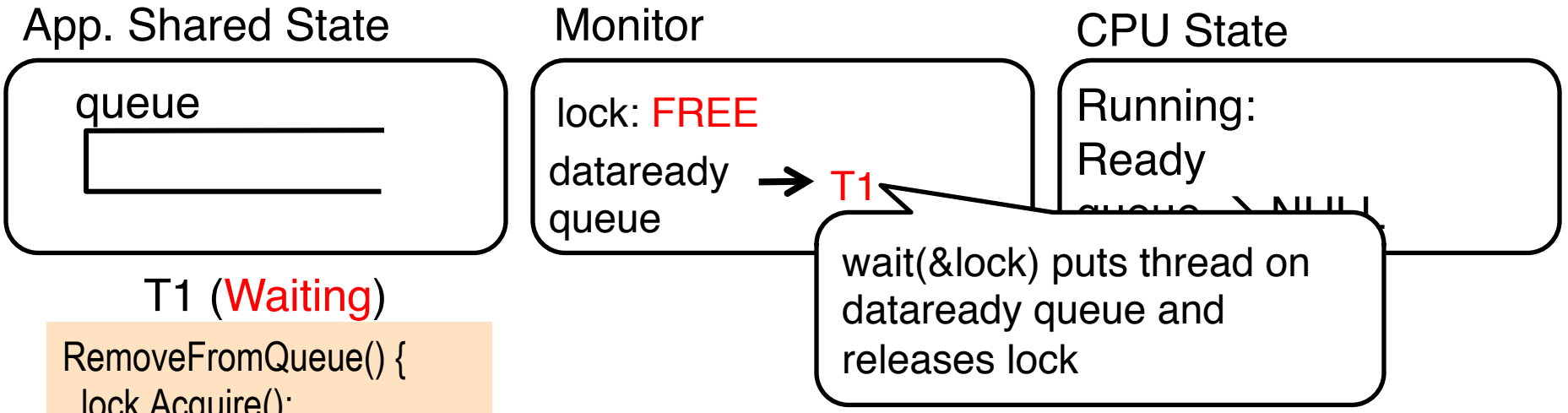
queue → NULL

…

T1 (Running)

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

Mesa Monitor: Why “while()”?

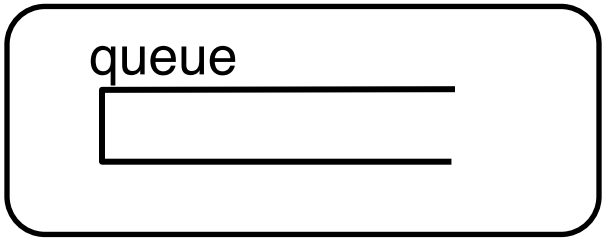


T1 (Waiting)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: FREE

dataready → T1

queue

CPU State

Running: T2

Ready

queue → NULL

…

T1 (Waiting)

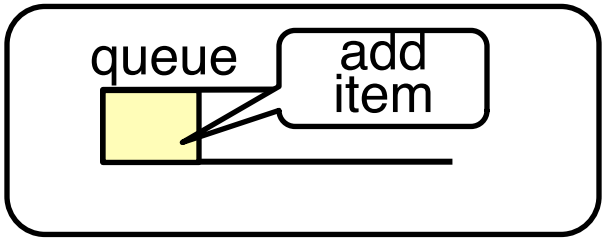
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T2)

dataready → T1

queue

CPU State

Running: T2

Ready

queue → NULL

…

T1 (Waiting)

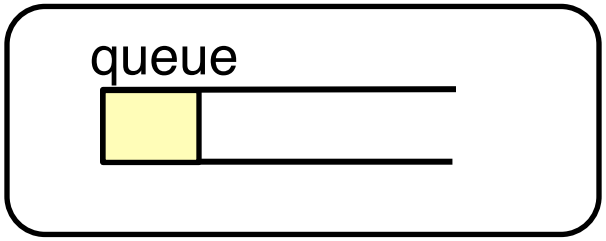
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T2)

dataready → NULL

queue

CPU State

Running: T2

Ready

queue → T

…

signal() wakes up T1 and moves it on ready queue

T1 (Ready)

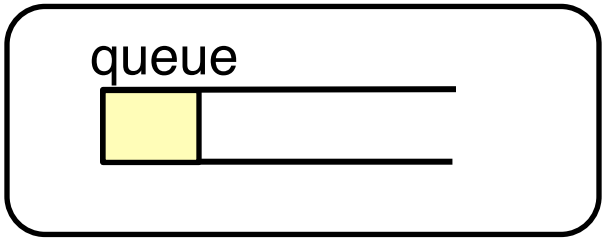
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T2)

dataready → NULL

queue

CPU State

Running: T2

Ready

queue → T1, T3

…

T1 (Ready)

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

T2 (Running)

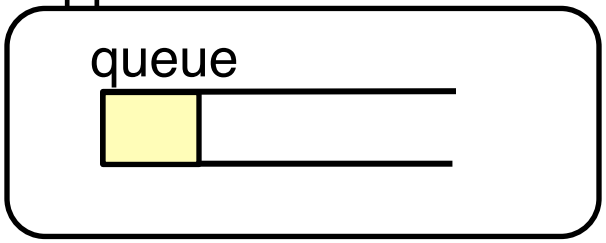
```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```

T3 (Ready)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: FREE

dataready queue → NULL

CPU State

Running:
Ready
queue → T1, T3

…

T1 (Ready)

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

T2 (Terminate)

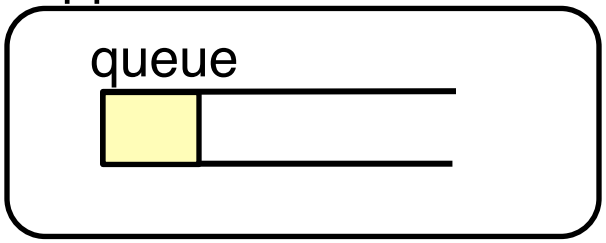
```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```

T3 (Ready)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: FREE

dataready queue → NULL

CPU State

Running: T3

Ready

T3 scheduled first!

T1 (Ready)

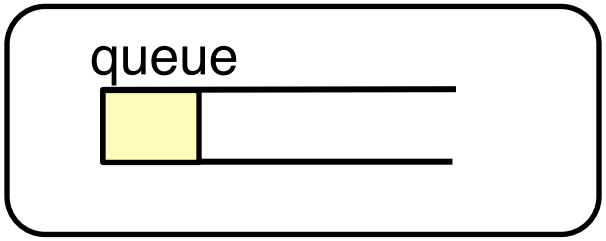
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Running)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T3)

dataready queue → NULL

CPU State

Running: T3
Ready
queue → T1

…

T1 (Ready)

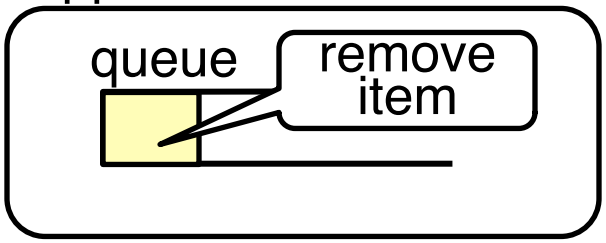
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Running)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T3)

dataready → NULL

queue

CPU State

Running: T3

Ready

queue → T1

…

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()”?

App. Shared State

queue

Monitor

lock: FREE

dataready queue → NULL

CPU State

Running:
Ready
queue → T1

…

T1 (Ready)

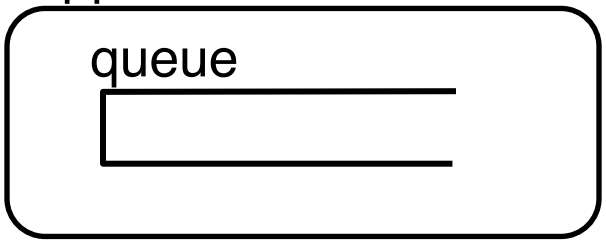
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Finished)

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

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T1)

dataready → NULL

queue

CPU State

Running: T1

Ready

queue → NULL

…

T1 (Running)

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

Mesa Monitor: Why “while()”?

App. Shared State

queue

Monitor

lock: BUSY (T1)

dataready queue → NULL

CPU State

Running: T1

Ready queue → NULL

…

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

queue

Monitor

lock: BUSY (T1)

dataready queue → NULL

CPU State

Running: T1

Ready

queue → NULL

…

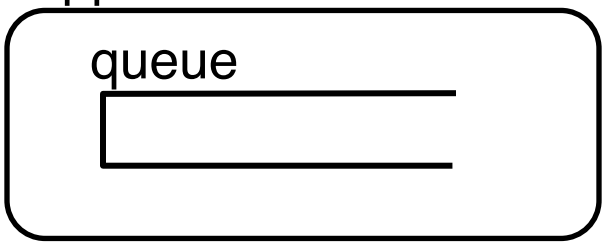
T1 (Running)

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

Replace “if” with “while”

Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T1)

dataready → NULL

queue

CPU State

Running: T1

Ready

queue → NULL

…

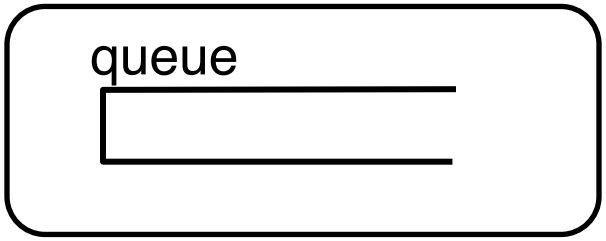
T1 (Ready)

```
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

lock: FREE

dataready → T1

queue

CPU State

Running: T1

Ready

queue → NULL

…

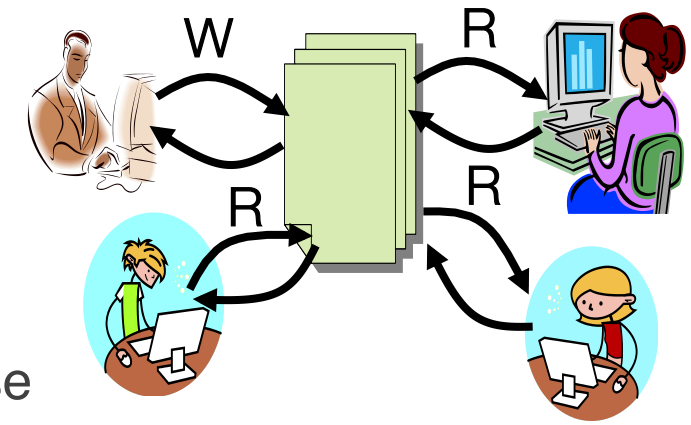
T1 (Waiting)

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

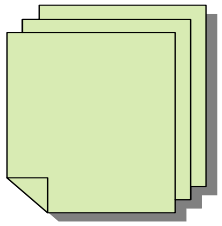
Readers/Writers Problem

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.wa                    ep on cond var
        WR--;  // No
    }
    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 act
    if (WW > 0){
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

Why give priority to writers

Why broadcast() 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
      loc
   }
```

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();
   }
```

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 = 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);
```

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();
```

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
      }
      release();
      ssDbase(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();
   }
```

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();
   }
```

No waiting writer, signal reader R3

Simulation of Readers/Writers Solution

R3 finishes

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!
      elease();

      AccessDbase(ReadOnly);

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

Got signal from W1

Simulation of Readers/Writers Solution

R3 finishes

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

R3 finishes

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

R3 finishes

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

R3 finishes

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

R3 finishes

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!

Readers/Writers Questions

```
Reader() {
      lock.Acquire();
      while ((AW + WW) > 0) {
          WR++;
          okToRead.wait(&lock);
          WR--;
      }
      AR++;
      lock.release(
      AccessDbase(R
      lock.Acquire();
      AR--;
      if (AR == 0 && WW > 0)
           okToWrite.signal();
      lock.Release();
   }
```

What if we remove this line?

```
Writer() {
      lock.Acquire();
      while ((AW + AR) > 0) {
          WW++;
          okToWrite.wait(&lock);
          WW--;
      }
      AW++;
      lock.release();
      AccessDbase(ReadWrite);
      lock.Acquire();
      AW--;
      if (WW > 0){
             okToWrite.signal();
      } else if (WR > 0) {
          okToRead.broadcast();
      }
      lock.Release();
}
```

Readers/Writers Questions

```
Reader() {
      lock.Acquire();
      while ((AW + WW) > 0) {
          WR++;
          okToRead.wait(&lock);
          WR--;
      }
      AR++;
      lock.release(
      AccessDbase(R
      lock.Acquire();
      AR--;
      if (AR == 0 && WW > 0)
           okToWrite.broadcast();
      lock.Release();
   }
```

What if turn signal to broadcast?

```
Writer() {
      lock.Acquire();
      while ((AW + AR) > 0) {
          WW++;
          okToWrite.wait(&lock);
          WW--;
      }
      AW++;
      lock.release();
      AccessDbase(ReadWrite);
      lock.Acquire();
      AW--;
      if (WW > 0){
             okToWrite.signal();
      } else if (WR > 0) {
          okToRead.broadcast();
      }
      lock.Release();
}
```

Readers/Writers Questions

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

      AccessDbase(ReadOnly);

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

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

Readers/Writers Questions

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

      AccessDbase(ReadOnly);

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

```
Writer() {
      lock.Acquire();
      while ((AW + AR) > 0) {
          WW++;
          okContinue.wait(&lock);
          WW--;
      }
      AW++;
      lock.release();
      AccessDbase(ReadWrite);
      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
- R1 signals R2

Readers/Writers Questions

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

      AccessDbase(ReadOnly);

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

```
Writer() {
      lock.Acquire();
      while ((AW + AR) > 0) {
          WW++;
          okContinue.wait(&lock);
          WW--;
      }
      AW++;
      lock.release();
      AccessDbase(ReadWrite);
      lock.Acquire();
      AW--;
      if (WW > 0){
           okContinue.signal();
      } else if (WR > 0) {
          okContinue.broadcast();
      }
```

Need to change to broadcast!

Other Kinds of Notifications

Timeouts: e.g., retry, and eventually go back to wait

Aborts: e.g., tell a process to terminate, clean up the state, etc

Deadlocks:

- Wait only releases the lock of the current monitor, not any nested calling monitors
- General problem with modular systems and synchronization:
  - Synchronization requires *global* knowledge about locks, which violates information hiding paradigm of modular programming

Four Requirements for Deadlock

Mutual exclusion

- Only one thread at a time can use a resource.

Hold and wait

- Thread holding at least one resource is waiting to acquire additional resources held by other threads

No preemption

- Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

Circular wait

- There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
  - T_1 is waiting for a resource that is held by T_2
  - T_2 is waiting for a resource that is held by T_3
  - …
  - T_n is waiting for a resource that is held by T_1

Deadlock

Why is monitor deadlock less onerous than the yield problem for non-preemptive schedulers?

- Want to generally insert as many yields as possible to provide increased concurrency; only use locks when you want to synchronize
- Yield bugs are difficult to find (symptoms may appear far after the bogus yield)

Deadlock

Basic deadlock rule: no recursion, direct or mutual

- Alternatives? Impose ordering on acquisition
- “It is unreasonable to blame the tool when poorly chosen constraints lead to deadlock”

Lock granularity for concurrent access to objects

- Introduced monitored records so that the same monitor code could handle multiple instances of something in parallel

Interrupts

Devices can’t afford to wait to acquire a monitor lock

Introduced naked notifies: device notifies without holding the monitor lock

Had to worry about a timing race:

- The notify could occur between a monitor’s condition check and its call on Wait
- Added a wakeup-waiting flag (basically a binary semaphore) to condition variables

Priority Inversion

High-priority processes may block on lower-priority processes

A solution:

- Temporarily increase the priority of the holder of the monitor to that of the highest priority blocked process
- Somewhat tricky – what happens when that high-priority process finishes with the monitor?
  - You have to know the priority of the next highest one – keep them sorted or scan the list on exit

Exceptions

Must restore monitor invariant as you unwind the stack

- But, requires explicit UNWIND handlers (enable processes to cleanup before it's destroyed), otherwise lock is not released

Failure to handle exceptions results in debugger invocation

- "not much comfort, however, when a system is in operational use"

What does Java do?

- Release lock, no UNWIND primitive

Hints vs. Guarantees

Notify is only a hint

- Don’t have to wake up the right process
- Don’t have to change the notifier if we slightly change the wait condition (the two are decoupled)
- Easier to implement, because it’s always OK to wake up too many processes. If we get lost, we could even wake up everybody (broadcast)
  - Can we use broadcast everywhere there is a notify? Yes
  - Can we use notify everywhere there is a broadcast? No, might not have satisfied OK to proceed for A, have satisfied it for B

Enables timeouts and aborts

Hints vs. Guarantees

General Principle: use hints for performance that have little or better yet no effect on the correctness

- Many commercial systems use hints for fault tolerance: if the hint is wrong, things timeout and use a backup strategy
  - Performance hit for incorrect hint, but no errors

Performance

Assumes simple machine architecture

- Single execution, non-pipelined – what about multi-processors?

Context switch is very fast: 2 procedure calls (60 ticks)

Ended up not mattering much for performance:

- Ran only on uniprocessor systems
- Concurrency mostly used for clean structuring purposes

Performance

Procedure calls are slow: 30 instructions (RISC proc. calls are 10x faster); Why?

- Due to heap allocated procedure frames. Why did they do this?
  - Didn't want to worry about colliding process stacks
- Mental model was "any procedure call might be a fork": transfer was basic control transfer primitive

Process creation: ~ 1100 instructions

- Good enough most of the time
- Fast-fork package implemented later that keeps around a pool or "available" processes

3 Key Features about the Paper

Describes the experiences designers had with designing, building and using a large system that aggressively relies on lightweight processes and monitor facilities for all its software concurrency needs

Describes various subtle issues of implementing a threads-with-monitors design in real life for a large system

Discusses the performance and overheads of various primitives and presents three representative applications, but doesn’t give a big picture of how important various decisions and features turned out to be

Some Flaws

Gloss over how hard it is to program with locks and exceptions sometimes – not clear if there are better ways

Performance discussion doesn't give the big picture

- Tries to be machine-independent (ticks), but assumes particular model

A takeaway lesson: The lightweight threads-with-monitors programming paradigm can be used to successfully build large systems, but there are subtle points that have to be correct in the design and implementation in order to do so