Recall: Monitors and Condition Variables

- **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
  - Some languages like Java provide monitors in the language
- **Condition Variable**: a queue of threads waiting for something inside a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section
- Operations:
  - **Wait(&lock)**: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - **Signal()**: Wake up one waiter, if any
  - **Broadcast()**: Wake up all waiters
- **Rule**: Must hold lock when doing condition variable ops!

Recall: Bounded Buffer, 3rd cut (coke machine)

```
Semaphore fullSlots = 0; // Initially, no coke
Semaphore emptySlots = bufSize; // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    semaP(&emptySlots); // Wait until space
    semaP(&mutex);    // Wait until machine free
    Enqueue(item);  // Machine free
    semaV(&mutex);  // Tell consumers there is more coke
    semaV(&fullSlots); // Tell producer need more coke
}

Consumer() {
    semaP(&fullSlots); // Check if there's a coke
    semaP(&mutex);    // Wait until machine free
    item = Dequeue(); // Machine free
    semaV(&mutex);
    semaV(&emptySlots); // Tell producer need more coke
    return item;
}
```

Recall: Bounded Buffer – 4th cut (Monitors, pthread-like)

```
lock buf_lock = <initially unlocked>
condition producer_CV = <initially empty>
condition consumer_CV = <initially empty>

Producer(item) {
    acquire(&buf_lock);
    while (buffer full) { cond_wait(&producer_CV, &buf_lock); }
    enqueue(item);
    cond_signal(&consumer_CV);
    release(&buf_lock);
}

Consumer() {
    acquire(buf_lock);
    while (buffer empty) { cond_wait(&consumer_CV, &buf_lock); }
    item = dequeue();
    cond_signal(&producer_CV);
    release(buf_lock);
    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 Structure of *Mesa* Monitor Program

- Monitors represent the synchronization logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of mesa monitor-based program:
  ```
  lock
  while (need to wait) {
    condvar.wait();
  }
  unlock
  do something so no need to wait
  lock
  condvar.signal();
  unlock
  ```

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 database
    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

```java
Reader() {
  // First check self into system
  acquire(&lock);
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;
    // No. Writers exist
    cond_wait(&okToRead,&lock);// Sleep on cond var
    WR--;
    // No longer waiting
  }
  AR++;
  // Now we are active!
  release(&lock);
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  acquire(&lock);
  // No longer active
  if (AR == 0 && WW > 0) // No other active readers
    cond_signal(&okToWrite);// Wake up one writer
  release(&lock);
}
```
Code for a Writer

```c
Writer() {
    // First check self into system
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite,&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    release(&lock);
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    acquire(&lock);
    AW--; // No longer active
    if (WW > 0) { // Give priority to writers
        cond_signal(&okToWrite); // Wake up one writer
    } else if (WR > 0) { // Otherwise, wake reader
        cond_broadcast(&okToRead); // Wake all readers
    }
    release(&lock);
}
```

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 (no waiting threads)
- AR = 0, WR = 0, AW = 0, WW = 0

```c
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```
Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader()
{
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

- R1 accessing dbase (no other threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader()
{
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
- AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader()
{
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```
Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
- AR = 1, WR = 0, AW = 0, WW = 0

Reader()
{
    acquire(&lock);

    while ((AW + WW) > 0) // Is it safe to read?
    {
        WR++ // No. Writers exist
        cond_wait(&okToRead,&lock); // Sleep on cond var
        WR--; // No longer waiting
    }

    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
    release(&lock);
}

Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
- AR = 2, WR = 0, AW = 0, WW = 0

Reader()
{
    acquire(&lock);

    while ((AW + WW) > 0) // Is it safe to read?
    {
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock); // Sleep on cond var
        WR--; // No longer waiting
    }

    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
    release(&lock);
}

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()
acquire(&lock);
while ((AW + AR) > 0) { // Is it safe to write?
    WW++; // No. Active users exist
    cond_wait(&okToWrite,&lock); // Sleep on cond var
    WW--; // No longer waiting
    AW++; // Now we are active!
    release(&lock);
}

acquire(&lock);
if (WW > 0){
    cond_signal(&okToWrite);
} else if (WR > 0){
    cond_broadcast(&okToRead);
} release(&lock);

AccessDBase(ReadWrite);

Simulation of Readers/Writers Solution

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

Writer()
acquire(&lock);
while ((AW + AR) > 0) { // Is it safe to write?
    WW++; // No. Active users exist
    cond_wait(&okToWrite,&lock); // Sleep on cond var
    WW--; // No longer waiting
    AW++; // Now we are active!
    release(&lock);
}

AccessDBase(ReadWrite);

Simulation of Readers/Writers Solution

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

Reader()
acquire(&lock);
while ((AW + WR) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    cond_wait(&okToRead,&lock); // Sleep on cond var
    WR--; // No longer waiting
    AR++; // Now we are active!
    release(&lock);
}

AccessDBase(ReadOnly);

Simulation of Readers/Writers Solution

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

Reader()
acquire(&lock);
while ((AW + WR) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    cond_wait(&okToRead,&lock); // Sleep on cond var
    WR--; // No longer waiting
    AR++; // Now we are active!
    release(&lock);
}

AccessDBase(ReadOnly);
Simulation of Readers/Writers Solution

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

Reader()
{
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}

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 and R3 waiting)
- AR = 2, WR = 1, AW = 0, WW = 1

Reader()
{
    acquire(&lock);
    while ((AW + WW > 0) { // Is it safe to read?
        WR++; // No Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}

AccessDBase(ReadOnly);
acquire(&lock);
AR--; if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);

Simulation of Readers/Writers Solution

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

Reader()
{
    acquire(&lock);
    while ((AW + WW > 0) { // Is it safe to read?
        WR++; // No Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}

AccessDBase(ReadOnly);
acquire(&lock);
AR--; if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);
Simulation of Readers/Writers Solution

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

`Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);acquire(&lock);
    AR--;if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}

Simulation of Readers/Writers Solution

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

`Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}

Simulation of Readers/Writers Solution

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

`Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}

Simulation of Readers/Writers Solution

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

`Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}`
Simulate the Readers/Writers Problem

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

```c
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WM++;
        cond_wait(&okToWrite,&lock); // Sleep on cond var
    } // No longer waiting
    AW++; release(&lock);
    AccessDBase(ReadWrite);
    acquire(&lock);
    if (WW > 0){
        cond_signal(&okToWrite);
    } else-if (WR > 0) { // No. Active users exist
        cond_broadcast(&okToRead);
    } else release(&lock);
}
```

Simulation of Readers/Writers Solution

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

```c
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WM++;
        cond_wait(&okToWrite,&lock); // Sleep on cond var
    } // No longer waiting
    AW++; release(&lock);
    AccessDBase(ReadWrite);
    acquire(&lock);
    if (WW > 0){
        cond_signal(&okToWrite);
    } else-if (WR > 0) { // No. Active users exist
        cond_broadcast(&okToRead);
    } else release(&lock);
}
```

Simulation of Readers/Writers Solution

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

```c
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite,&lock);// Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;release(&lock);
    AccessDBase(ReadWrite);
    acquire(&lock);
    if (WW > 0) {
        cond_signal(sokToWrite);
    } else if (WR > 0) {
        cond_broadcast(sokToRead);
    } release(&lock);
}
```

Simulation of Readers/Writers Solution

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

```c
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite,&lock);// Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;release(&lock);
    AccessDBase(ReadWrite);
    acquire(&lock);
    if (WW > 0) {
        cond_signal(sokToWrite);
    } else if (WR > 0) {
        cond_broadcast(sokToRead);
    } release(&lock);
}
```

Simulation of Readers/Writers Solution

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

```c
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite,&lock);// Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;release(&lock);
    AccessDBase(ReadWrite);
    acquire(&lock);
    if (WW > 0) {
        cond_signal(sokToWrite);
    } else if (WR > 0) {
        cond_broadcast(sokToRead);
    } release(&lock);
}
```

Simulation of Readers/Writers Solution

- W1 signaling readers (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

```c
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite,&lock);// Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;release(&lock);
    AccessDBase(ReadWrite);
    acquire(&lock);
    if (WW > 0) {
        cond_signal(sokToWrite);
    } else if (WR > 0) {
        cond_broadcast(sokToRead);
    } release(&lock);
}
```
Simulation of Readers/Writers Solution

• R3 gets signal (no waiting threads)
• AR = 0, WR = 1, AW = 0, WW = 0

Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        if (AR == 0 && WW > 0) {
            cond_signal(&okToWrite);
            release(&lock);
        } else {
            WR++;
            cond_wait(&okToRead,&lock); // Sleep on cond var
            WR--;
        }
        AR++;
        // Now we are active!
        release(&lock);
    }
    AccessDBase(ReadOnly);
    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
    release(&lock);
}

Simulation of Readers/Writers Solution

• R3 gets signal (no waiting threads)
• AR = 0, WR = 0, AW = 0, WW = 0

Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        if (AR == 0 && WW > 0) {
            cond_signal(&okToWrite);
            release(&lock);
        } else {
            WR++;
            cond_wait(&okToRead,&lock); // Sleep on cond var
            WR--;
        }
        AR++;
        // Now we are active!
        release(&lock);
    }
    AccessDBase(ReadOnly);
    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
    release(&lock);
}

Simulation of Readers/Writers Solution

• R3 accessing dbase (no waiting threads)
• AR = 1, WR = 0, AW = 0, WW = 0

Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        if (AR == 0 && WW > 0) {
            cond_signal(&okToWrite);
            release(&lock);
        } else {
            WR++;
            cond_wait(&okToRead,&lock); // Sleep on cond var
            WR--;
        }
        AR++;
        // Now we are active!
        release(&lock);
    }
    AccessDBase(ReadOnly);
    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
    release(&lock);
}

Simulation of Readers/Writers Solution

• R3 finishes (no waiting threads)
• AR = 1, WR = 0, AW = 0, WW = 0

Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        if (AR == 0 && WW > 0) {
            cond_signal(&okToWrite);
            release(&lock);
        } else {
            WR++;
            cond_wait(&okToRead,&lock); // Sleep on cond var
            WR--;
        }
        AR++;
        // Now we are active!
        release(&lock);
    }
    AccessDBase(ReadOnly);
    acquire(&lock);
    AR--; if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
    release(&lock);
Simulation of Readers/Writers Solution

- R3 finishes (no waiting threads)
- AR = 0, WR = 0, AW = 0, WW = 0

```c
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++;
        cond_wait(&okToRead,&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
}
```

```c
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        cond_wait(&okToWrite,&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    release(&lock);
    AccessDBase(ReadWrite);
}
```

Administrivia

- Still grading Midterm 1 (Sorry)
  - Finishing soon!
  - Solutions also will be up soon.
- Homework #2 due Thursday, 2/22
  - Option to do this in Rust!
  - Rust crash course on Monday 2/26
- Professor Kubi's office hours changed:
  - Monday (1:00-2:00PM), Thursday (3:00-4:00PM)
  - 673 Soda Hall
- FYI: Next Midterm on 3/14
  - PI Day!

Questions

- Can readers starve? Consider Reader() entry code:
  - if (AR == 0 && WW > 0)
    - cond_signal(&okToWrite);
- What if we erase the condition check in Reader exit?
  - if (AR == 0 && WW > 0)
    - cond_signal(&okToWrite);
- Further, what if we turn the signal() into broadcast()
  - if (AR == 0 && WW > 0)
    - cond_broadcast(&okContinue);
- Finally, what if we use only one condition variable (call it "okContinue") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Use of Single CV: okContinue

```c
Reader() {
    // check into system
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++;
        cond_wait(&okContinue,&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    AccessDBase(ReadOnly);
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_signal(&okContinue);
    release(&lock);
}
```

```c
Writer() {
    // check into system
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        cond_wait(&okContinue,&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    release(&lock);
    AccessDBase(ReadWrite);
    // check out of system
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_signal(&okContinue);
    release(&lock);
}
```

What if we turn okToWrite and okToRead into okContinue (i.e. use only one condition variable instead of two)?
Use of Single CV: \texttt{okContinue}

\begin{verbatim}
Reader() {  
  // check into system  
  acquire(&lock);  
  while ((AW + WW) > 0) {  
    WR++;  
    cond_wait(&\texttt{okContinue},&lock);  
    WR--;  
  }  
  AR++;  
  release(&lock);  
}  

// read-only access  
AccessDbase(ReadOnly);  

// check out of system  
acquire(&lock);  
if (AR == 0 \&\& WW > 0)  
  cond_signal(&\texttt{okContinue});  
release(&lock);  
}  

Writer() {  
  // check into system  
  acquire(&lock);  
  while ((AW + AR) > 0) {  
    WW++;  
    cond_wait(&\texttt{okContinue},&lock);  
    WW--;  
  }  
  AW++;  
  release(&lock);  
}  

// read/write access  
AccessDbase(ReadWrite);  

// check out of system  
acquire(&lock);  
if (WW > 0)  
  cond_broadcast(&\texttt{okContinue});  
release(&lock);  
else if (WR > 0)  
  cond_broadcast(&\texttt{okContinue});  
}  
\end{verbatim}

Consider this scenario:
- \texttt{R1} arrives
- \texttt{W1, R2} arrive while \texttt{R1} still reading \texttt{\rightarrow} \texttt{W1 and R2 wait for R1 to finish}
- Assume \texttt{R1}'s signal is delivered to \texttt{R2} (not \texttt{W1})

Need to change to \texttt{broadcast()!}
Must broadcast() to sort things out!

Can we construct Monitors from Semaphores?
- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?
  \begin{verbatim}
  Wait(Semaphore *thesema) { semap(thesema); }  
  Signal(Semaphore *thesema) { semaV(thesema); }  
  \end{verbatim}
- Does this work better?
  \begin{verbatim}
  Wait(Lock *thelock, Semaphore *thesema) {  
    release(thelock);  
    semap(thesema);  
    acquire(thelock);  
  }  
  Signal(Semaphore *thesema) {  
    if semaphore queue is not empty  
      semaV(thesema);  
  }  
  \end{verbatim}
- Not legal to look at contents of semaphore queue
- There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()}
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Construction of Monitors from Semaphores (con't)
- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?
  \begin{verbatim}
  Wait(Lock *thelock, Semaphore *thesema) {  
    release(thelock);  
    semap(thesema);  
    acquire(thelock);  
  }  
  Signal(Semaphore *thesema) {  
    if semaphore queue is not empty  
      semaV(thesema);  
  }  
  \end{verbatim}
- Not legal to look at contents of semaphore queue
- There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()}
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?
C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know all the code paths out of a critical section
    ```c
    int Rtn() {
        acquire(&lock);
        if (exception) {
            release(&lock);
            return errReturnCode;
        }
        release(&lock);
        return OK;
    }
    ```
  - Watch out for `std::setjmp`/`std::longjmp`!
    » Can cause a non-local jump out of procedure
    » In example, procedure E calls `std::longjmp`, popping stack back to procedure B
    » If Procedure C had `acquire(&lock)`, problem!

- Harder with more locks
  ```c
  void Rtn() {
      lock1.acquire();
      if (error) {
          lock1.release();return;
      }
      lock2.acquire();
      if (error) {
          lock2.release();lock1.release();return;
      }
      lock2.release();lock1.release();
  }
  ```

- Is `goto` a solution???
  ```c
  void Rtn() {
      lock1.acquire();
      if (error) {
          goto release_lock1_and_return;
      }
      lock2.acquire();
      if (error) {
          goto release_both_and_return;
      }
      lock2.release();lock1.release();
      release_both_and_return:
      lock2.release();
      release_lock1_and_return:
      lock1.release();
  }
  ```

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
    - Consider:
      ```cpp
      void Rtn() {
          lock.acquire();
          DoFoo();
          lock.release();
      }
      void DoFoo() {
          if (exception) throw errException;
      }
      ```
    - Notice that an exception in `DoFoo()` will exit without releasing the lock!

- C++ Language Support for Synchronization (con't)
  - Must catch all exceptions in critical sections
    - Catch exceptions, release lock, and re-throw exception:
      ```cpp
      void Rtn() {
          lock.acquire();
          try {
              DoFoo();
          } catch (...) { // catch exception
              lock.release(); // release lock
              throw; // re-throw the exception
          }
          lock.release();
          DoFoo() {
              if (exception) throw errException;
          }
      }
      ```
Much better: C++ Lock Guards

```cpp
#include <mutex>
int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
    std::lock_guard<std::mutex> lock(global_mutex);
    global_i++;
    // Mutex released when 'lock' goes out of scope
}
```

Python with Keyword

```python
lock = threading.Lock()
...
with lock: # Automatically calls acquire()
    some_var += 1
    ...
# release() called however we leave block
```

Java synchronized Keyword

- Every Java object has an associated lock:
  - Lock is acquired on entry and released on exit from a synchronized method
  - Lock is properly released if exception occurs inside a synchronized method
  - Mutex execution of synchronized methods (beware deadlock)

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

Java Support for Monitors

- Along with a lock, every object has a single condition variable associated with it
- To wait inside a synchronized method:
  - void wait();
  - void wait(long timeout);

- To signal while in a synchronized method:
  - void notify();
  - void notifyAll();
Goal for Today

- Discussion of Scheduling:
  - Which thread should run on the CPU next?
- Scheduling goals, policies
- Look at a number of different schedulers

if (readyThreads(TCBs)) {
  nextTCB = selectThread(TCBs);
  run( nextTCB );
} else {
  run_idle_thread();
}

Recall: Stacks for Yield with Multiple Threads

- Consider the following code blocks:
  proc A() {
    B();
  }
  proc B() {
    while(TRUE) {
      yield();
    }
  }

- Suppose we have 2 threads:
  - Threads S and T
  - Assume that both have been running for a while

Hardware context switch support in x86

- Syscall/Irr (U → K)
  - PL 3 → 0
  - TSS ← EFLAGS, CS:EIP
  - SS:ESP ← k-thread stack (TSS PL 0);
  - push (old) SS:ESP onto (new) k-stack
  - push (old) EFLAGS, CS:EIP, <err>
  - CS:EIP ← <k target handler>

- Then
  - Handler saves other regs, etc
  - Does all its works, possibly choosing other threads, changing PTBR (CR3)
  - kernel thread has set up user GPRs
- iret (K → U)
  - PL 0 → 3
  - EFLAGS, CS:EIP ← popped off k-stack
  - SS:ESP ← popped off k-stack

Pintos: Kernel Crossing on Syscall or Interrupt

- User code
- User stack
- Time

Pintos: tss.c, intr-stubs.S

pg 2,942 of 4,922 of x86 reference manual
**Pintos: Context Switch – Scheduling**

- Each user process/thread associated with a kernel thread, described by a 4KB page object containing TCB and kernel stack for the kernel thread.

**MT Kernel single Thread Process ala Pintos/x86**

- Kernel stack and TCB stored in one 4K page.

**Running User Code with Kernel stack waiting**

- x86 CPU holds interrupt SP in register.
- During user thread execution, associated kernel thread is "standing by".

**In Kernel Thread: No User Component**

- Kernel threads execute with small stack in thread structure.
- Pure kernel threads have no corresponding user-mode thread.
**User → Kernel (interrupts, syscalls)**

- Mechanism vectors through "interrupt vector"

**Kernel → User**

- Interrupt return (iret) restores user stack, IP, and PL

**User → Kernel via “interrupt vector” (interrupts & traps)**

- Interrupts (timer) or trap (syscall, page fault) transfers through interrupt vector (IDT)
  - Each slot for different exception type

**Pintos Interrupt Processing for Timer (0x20)**

- Wrapper for generic handler
Switch to Kernel Stack for Thread

- Information required to restart thread stored on kernel stack
  - Switching becomes simple to different kernel stack and restoring

Pintos Interrupt Processing for Timer (0x20)

- intrNN_stub(): Wrapper for generic handler
  - intr_entry: save regs as frame
    - set up kernel env.
    - call intr_handler
  - intr_exit: restore regs
    - iret

- Timer may trigger thread switch
  - thread_tick
    - Updates thread counters
    - If quanta exhausted, sets yield flag
  - thread_yield
    - On path to rtn from interrupt
    - Sets current thread back to READY
    - Pushes it back on ready_list
    - Calls schedule to select next thread to run upon iret
  - Schedule
    - Selects next thread to run
    - Calls switch_threads to change regs to point to stack for thread to resume
    - Sets its status to RUNNING
    - If user thread, activates the process
    - Returns back to intr_handler
Thread Switch (switch.S)

- switch_threads: save regs on current small stack, change SP, return from destination threads call to switch_threads

Kernel → Different User Thread

- iret restores user stack and priority level (PL)

Famous Quote WRT Scheduling: Dennis Ritchie

Dennis Ritchie, Unix V6, slp.c:

```
2336 */
2337 /* If the new process paused because it was swapped out, set the
2338 stack level to the last call to savu(_ssav). This means that the
2339 return which is executed immediately after the call to aretu actually
2340 returns from the last routine which did the savu.
2341 */
2342 */
2343 /* You are not expected to understand this. */
2344 */

```

"""You are not expected to understand this."

Source: Dennis Ritchie, Unix V6 slp.c (context-switching code) as per The Unix Heritage Society(tuhs.org); gif by Eddie Koehler.

Included by Ali R. Butt in CS3204 from Virginia Tech
Recall: Scheduling

- Question: How is the OS to decide which of several tasks to take off a queue?
- **Scheduling**: deciding which threads are given access to resources from moment to moment
  - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access

---

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
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - 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

---

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 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
- **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)
- **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:**
  - Processes: $P_1, P_2, P_3$
  - Table:
    | Process | Burst Time |
    |---------|------------|
    | $P_1$   | 24         |
    | $P_2$   | 3          |
    | $P_3$   | 3          |
  - Suppose processes arrive in the order: $P_1, P_2, P_3$
  - Gantt Chart:
    - 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 stuck behind long process

### Convoy effect

- With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running.

### FCFS Scheduling (Cont.)

- **Example continued:**
  - Suppose that processes arrive in order: $P_2, P_3, P_1$
  - Gantt Chart:
    - 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 items!
    - Upside: get to read about Space Aliens!
Conclusion

- **Monitors**: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Readers/Writers Monitor example
  - Shows how monitors allow sophisticated controlled entry to protected code
  - Mesa scheduling allows a more relaxed checking of wait conditions
- Monitors supported natively in a number of languages

- **Scheduling Goals**:
  - Minimize Response Time (e.g. for human interaction)
  - Maximize Throughput (e.g. for large computations)
  - Fairness (e.g. Proper Sharing of Resources)
  - Predictability (e.g. Hard/Soft Realtime)
- **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