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

```c
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);
    semaV(&mutex);
    semaV(&fullSlots); // Tell consumers there is more coke
}

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

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: 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
Recall: Code for a Reader

```c
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);
    AR--; // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        cond_signal(&okToWrite);// Wake up one writer
    release(&lock);
}
```

Recall: 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

```c
class 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);
    AccessDatabase(ReadOnly);
}
```

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);
    AccessDatabase(ReadOnly);
    acquire(&lock);
    AR--; // No longer active
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```
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);
    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

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);
Simulation of Readers/Writers Solution

- R1 and R2 accessing database
- AR = 2, WR = 0, AW = 0, WW = 0

**Reader()**

```c
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);
```

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 database)
- AR = 2, WR = 0, AW = 0, WW = 0

**Writer()**

```c
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);
```

Simulation of Readers/Writers Solution

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

**Writer()**

```c
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);
```

**Writer()**

```c
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);
```
Simulation of Readers/Writers Solution

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

```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

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

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

- R1 and R2 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);
    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 = 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);
    // Fix here
    AR--; // Fix here
    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);
    // Fix here
    AR--; // Fix here
    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

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

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

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

```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 signals a writer (W1 and 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

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, 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);
    acquire(&lock);
    if (WW > 0)
        cond_signal(&okToWrite);
    else if (WR > 0)
        cond_broadcast(&okToRead);
    release(&lock);
}
Simulation of Readers/Writers Solution

- W1 accessing dbase (R3 still waiting)
- AR = 0, WR = 1, AW = 1, 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++;      // Is it safe to write?
    release(&lock);

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

Simulation of Readers/Writers Solution

- W1 finishes (R3 still waiting)
- AR = 0, WR = 1, AW = 1, 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++;      // Is it safe to write?
    release(&lock);

    acquire(&lock);
    if (WW > 0){
        cond_signal(&okToWrite);
    } else-if (WR > 0) {
        cond_broadcast(&okToRead);
    }
    release(&lock);
}
Simulation of Readers/Writers Solution

- W1 signaling readers (R3 still waiting)
- AR = 0, WR = 1, 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
        AW--; // No longer waiting
    }
    AW++; // Now we are active!
    release(&lock);
    AccessDBase(ReadWrite);

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

- 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?
        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

- 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?
        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);

Questions

- Can readers starve? Consider Reader() entry code:
    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!
- What if we erase the condition check in Reader exit?
    AR--; // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        cond_signal(&okToWrite);
    release(&lock);
- Further, what if we turn the signal() into broadcast()
  WR++; // No longer active
  cond_broadcast(&okToWrite); // Wake up sleepers
- 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

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

AccessDBase(ReadOnly);

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

    // read-only access
    AccessDBase(ReadOnly);
    // 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: `okContinue`

Reader() {
    // check into system
    acquire(&lock);
    while ((AW + WW) > 0) {
        WR++; cond_wait(&okContinue,&lock);
        WR--;
    }
    AR++; release(&lock);
    // read-only access
    AccessDbase(ReadOnly);
    // check out of system
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_signal(&okContinue);
    release(&lock);
}

Writer() {
    // check into system
    acquire(&lock);
    while ((AW + AR) > 0) {
        WW++; cond_wait(&okContinue,&lock);
        WW--;
    }
    AW++; release(&lock);
    // read/write access
    AccessDbase(ReadWrite);
    // check out of system
    acquire(&lock);
    if (AR == 0 && WW > 0)
        cond_broadcast(&okContinue);
    release(&lock);
}

Consider this scenario:
- R1 arrives
- W1, R2 arrive while R1 still reading → W1 and R2 wait for R1 to finish
- Assume R1’s signal is delivered to R2 (not W1)

Can we construct Monitors from Semaphores?
- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?
  
  ```
  Wait(Semaphore *thesema) { semap(thesema); }
  Signal(Semaphore *thesema) { semav(thesema); }
  ```
- Does this work better?
  ```
  Wait(Lock *thelock, Semaphore *thesema) {
    release(thelock);
    semap(thesema);
    acquire(thelock);
  }
  Signal(Semaphore *thesema) {
    semav(thesema);
  }
  ```

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?
  ```
  Wait(Lock *thelock, Semaphore *thesema) {
    release(thelock);
    semap(thesema);
    acquire(thelock);
  }
  ```
  ```
  Signal(Semaphore *thesema) {
    if semaphore queue is not empty
      semav(thesema);
  }
  ```
  - 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?
Administrivia

- Still grading Midterm 1 (Sorry)
  - Finishing soon!
  - Solutions also will be up soon.
- Homework #2 due Thursday
- Professor Kubl’s office hours changed slightly:
  - Monday 2-3 (same), Wednesday 3-4 (different)
  - 673 Soda Hall

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 `setjmp`/`longjmp`!
    » Can cause a non-local jump out of procedure
    » In example, procedure E calls `longjmp`, popping stack back to procedure B
    » If Procedure C had `lock.acquire`, problem!

Concurrency and Synchronization in C

- 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();
  }
  ```
  - 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;
      }
      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:
    ```c
    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:
    ```c++
    void Rtn() {
      lock.acquire();
      try {
        DoFoo();
        // catch exception
        lock.release(); // release lock
        throw; // re-throw the exception
      } catch (...) {
        lock.release();
      }
      void DoFoo() {
        // Mutex released when ‘lock’ goes out of scope
        if (exception) throw errMsg;
      }
    }
    ```

Much better: C++ Lock Guards

```c++
#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

• More versatile than we show here (can be used to close files, database connections, etc.)

```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

Recall: What Do the Stacks Look Like?

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

- Suppose we have 2 threads:
  - Threads S and T

  Thread S
  `A`
  `B(while)`
  `yield`
  `run_new_thread`
  `switch`

  Thread T
  `A`
  `B(while)`
  `yield`
  `run_new_thread`
  `switch`

  Thread S's switch returns to Thread T's (and vice versa)

Hardware context switch support in x86

- Syscall/Intr (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;
  - ERegs, CS:EIP ← popped off k-stack
  - SS:ESB ← popped off k-stack

```
Pintos: Kernel Crossing on Syscall or Interrupt

- User code
- User stack
- User thread stack
- Kernel code
- Kernel thread stack
- TCB
- PTBR
- cs:eip
- ss:esp
- syscall / interrupt
- Saves
- iret
- TCB
- Processing ready to resume

Pintos: Context Switch – Scheduling

- User stack
- Kernel code
- Kernel thread stack
- syscall / interrupt
- TCB
- Processing ready to resume
- Schedule switch kernel threads

MT Kernel 1T Process ala Pintos/x86

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

In User thread, w/ Kernel thread 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 (exceptions, sycalls)

- Mechanism to resume k-thread goes through interrupt vector

Kernel → User

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

Pintos Interrupt Processing

```c
intrNN_stub() {
    push 0x20 (int #)
    jmp intr_entry
}
```

```c
intr_entry:
    save regs as frame
    set up kernel env.
    call intr_handler

intr_exit:
    restore regs
    iret
```
Interrupt transfers control through the Interrupt Vector (IDT in x86).
iret restores user stack and priority level (PL).

Switch to Kernel Thread for Process

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 Richie**

Dennis Richie, Unix V6, slp.c:

“If the new process paused because it was swapped out, set the stack level to the last call to savu(u_ssav). This means that the return which is executed immediately after the call to aretu actually returns from the last routine which did the savu.”

“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:**
  
<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

  Suppose processes arrive in the order: P₁, P₂, P₃

  The Gantt Chart for the schedule is:

  - Waiting time for P₁ = 0; P₂ = 24; P₃ = 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: P₂, P₃, P₁
  - Now, the Gantt chart for the schedule is:

  - Waiting time for P₁ = 6; P₂ = 0; P₃ = 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!
Round Robin (RR) Scheduling

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

RR Scheduling (Cont.)

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

Example of RR with Time Quantum = 20

- Example:
<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>53</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>68</td>
</tr>
<tr>
<td>$P_4$</td>
<td>24</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

  P1 P2 P3 P4 P1 P3 P4

- Waiting time for $P_1=(68-20)+(112-88)=72$
- $P_2=(20-0)=20$
- $P_3=(28-0)+(88-48)+(125-108)=85$
- $P_4=(48-0)+(108-68)=88$

- Average waiting time $= (72+20+85+88)/4=66\frac{1}{4}$
- Average completion time $= (125+28+153+112)/4 = 104\frac{1}{2}$

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

How to Implement RR in the Kernel?

- FIFO Queue, as in FCFS
- But preempt job after quantum expires, and send it to the back of the queue
  - How? Timer interrupt!
  - And, of course, careful synchronization

Project 2: Scheduling
**Round-Robin Discussion**

- How do you choose time slice?
  - What if too big?
    » Response time suffers
  - What if infinite (∞)?
    » Get back FIFO
  - What if time slice too small?
    » Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    » Worked ok when UNIX was used by one or two people.
  - What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput:
    » Typical time slice today is between 10ms – 100ms
    » Typical context-switching overhead is 0.1ms – 1ms
    » Roughly 1% overhead due to context-switching

**Comparisons between FCFS and Round Robin**

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  - RR scheduler quantum of 1s
  - All jobs start at the same time
- Completion Times:

<table>
<thead>
<tr>
<th>Job #</th>
<th>FIFO</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>991</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>992</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>999</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

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

**Earlier Example with Different Time Quantum**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantum</td>
<td>P = 32</td>
<td>P = 8</td>
<td>P = 8</td>
<td>P = 153</td>
</tr>
<tr>
<td>Q = 1</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
</tr>
<tr>
<td>Q = 5</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
</tr>
<tr>
<td>Completion Time</td>
<td>P = 85</td>
<td>P = 8</td>
<td>P = 153</td>
<td>P = 32</td>
</tr>
<tr>
<td>Q = 1</td>
<td>137</td>
<td>30</td>
<td>153</td>
<td>81</td>
</tr>
<tr>
<td>Q = 5</td>
<td>135</td>
<td>28</td>
<td>153</td>
<td>82</td>
</tr>
<tr>
<td>Q = 8</td>
<td>133</td>
<td>16</td>
<td>153</td>
<td>80</td>
</tr>
<tr>
<td>Q = 10</td>
<td>135</td>
<td>18</td>
<td>153</td>
<td>92</td>
</tr>
<tr>
<td>Q = 20</td>
<td>125</td>
<td>28</td>
<td>153</td>
<td>112</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>121</td>
<td>153</td>
<td>68</td>
<td>145</td>
</tr>
</tbody>
</table>

**Conclusion**

- Monitors represent the logic of the program
  » Wait if necessary
  » Signal when change something so any waiting threads can proceed
  » Monitors supported natively in a number of languages
- Readers/Writers Monitor example
  » Shows how monitors allow sophisticated controlled entry to protected code
- 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
- Next Time: Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  » Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  » Pros: Optimal (average response time)
  » Cons: Hard to predict future, Unfair