Recall: Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Mostly. Idea: only busy-wait to atomically check lock value
  - int guard = 0; // Global Variable!

```c
int mylock = FREE; // Interface: acquire(&mylock);
// release(&mylock);

acquire(int *thelock) {
    // Short busy-wait time
    while (test&set(guard));
    if (*thelock == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
        // guard == 0 on wakup!
    } else {
        *thelock = BUSY;
        guard = 0;
    }
}

release(int *thelock) {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        *thelock = FREE;
    }
    guard = 0;
}
```

- Note: sleep has to be sure to reset the guard variable
  - Why can’t we do it just before or just after the sleep?

Recall: Atomic Read-Modify-Write

```c
• test&set (address) {
    result = M[address]; // return result from "address" and
    M[address] = 1; // set value at "address" to 1
    return result;
}
• swap (address, register) {
    // x86
    temp = M[address];
    // swap register's value to
    M[address] = register;
    // value at "address"
    register = temp;
}
• compare&swap (address, reg1, reg2) {
    // x86 (returns old value), 68000 */
    if (reg1 == M[address]) {
        // If memory still == reg1,
        M[address] = reg2;
        // then put reg2 => memory
        return success;
    } else {
        // Otherwise do not change memory
        return failure;
    }
}
• load-linked&store-conditional(address) {
    // R4000, alpha */
    loop:
    ll r1, M[address];
    movi r2, 1;
    // Can do arbitrary computation
    sc r2, M[address];
    beqz r2, loop;
}
```

Recall: Linux futex: Fast Userspace Mutex

```c
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
          const struct timespec *timeout);
```

- uaddr points to a 32-bit value in user space
- futex_op
  - FUTEX_WAIT — if val == *uaddr sleep till FUTEX_WAIT
  - FUTEX_WAKE — wake up at most val waiting threads
  - FUTEX_FD, FUTEX_WAKE_OP, FUTEX_CMP_REQUEUE: More interesting operations!

- timeout
  - ptr to a timespec structure that specifies a timeout for the op
- Interface to the kernel sleep() functionality!
  - Let thread put themselves to sleep — conditionally!
- futex is not exposed in libc; it is used within the implementation of pthreads
  - Can be used to implement locks, semaphores, monitors, etc…
**Recall: Lock Using Atomic Instructions and Futex**

- Three (3) states:
  - **UNLOCKED**: No one has lock
  - **LOCKED**: One thread has lock
  - **CONTESTED**: Possibly more than one (with someone sleeping)
- Clean interface!
- Lock grabbed cleanly by either
  - `compare_and_swap()`
  - `First_swap()`
- No overhead if uncontested!
- Could build semaphores in a similar way!

```c
typedef enum { UNLOCKED, LOCKED, CONTESTED } Lock;
Lock mylock = UNLOCKED; // Interface: acquire(&mylock);
// release(&mylock);

acquire(Lock *thelock) {
    // If unlocked, grab lock!
    if (compare_swap(thelock, UNLOCKED, LOCKED))
        return;

    // Keep trying to grab lock, sleep in futex
    while (swap(mylock, CONTESTED) != UNLOCKED)
        // Sleep unless someone releases hear!
        futex(thelock, FUTEX_WAIT, CONTESTED);
}

release(Lock *thelock) {
    // If someone sleeping,
    if (swap(thelock, UNLOCKED) == CONTESTED)
        futex(thelock, FUTEX_WAKE, 1);
}
```

**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: 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:
  ```c
  lock while (need to wait) {
      condvar.wait();
  }
  unlock
  do something so no need to wait
  lock
  condvar.signal();
  ```
  - Check and/or update state variables
  - Wait if necessary
  ```c
  check_and_update();
  ```
  - Check and/or update state variables
Recall: 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

Recall: Code for a Reader

```c
Reader() {
    // First check self into system
    acquire(&lock);
    while ((AW + WW) > 0) {
        // Is it safe to read?
        WR++;
        if (WR > 0) {
            cond_wait(&okToRead,&lock); // Sleep on cond var
            WR--;
            if (WR == 0) cond_signal(&okToWrite); // Wake up one writer
        }
    }
    AR++;
    // Now we are active!
    release(&lock);
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    acquire(&lock);
    if (AR == 0 && WW > 0) 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++;
        if (WW > 0) {
            cond_wait(&okToWrite,&lock); // Sleep on cond var
            WW--;
            if (WW == 0) cond_signal(&okToWrite); // Wake up one writer
        }
    }
    AW++;
    // Now we are active!
    release(&lock);
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    acquire(&lock);
    if (WW > 0) {
        // Give priority to writers
        cond_signal(&okToWrite);
    } else if (WR > 0) {
        // Otherwise, wake reader
        cond_broadcast(&okToRead);
    }
    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

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 comes along (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);
    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

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);
```c
acquire(&lock);
AR--; // No longer waiting
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()
```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);
```c
acquire(&lock);
AR--; // No longer waiting
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()
```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);
```c
acquire(&lock);
AR--; // No longer waiting
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);
```
Simulation of Readers/Writers Solution

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

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

Simulation of Readers/Writers Solution

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

Writer() { 
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write? 
        WW++;
        cond_wait(&okToWrite,&lock); // Sleep on cond var
        WW--;
        if (WW > 0)
            cond_signal(&okToWrite);
    }
    AW++;
    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 comes along (R1 and R2 are still accessing dbase)
- AR = 2, WR = 0, AW = 0, WW = 1

Writer()
    acquire(&lock);
    while ((AW + AR) > 0) {
        // Is it safe to write?
        AW++;
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        // No. Active users exist
        cond_broadcast(&okToRead);
        // No longer waiting
        AW--;
    }
    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 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++;
        cond_wait(&okToRead, &lock); // Sleep on cond var
        // No. Writers exist
        cond_broadcast(&okToWrite);
        // No longer waiting
        WR--;
    }
    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 comes along (R1, R2 accessing dbase, W1 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

- 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);
AR--;
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

- 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

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 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);
    AR--;
    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 = 1

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

- W1 gets signal (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(&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

```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(&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

```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(&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

```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(&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 = 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++;
        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 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
            WW--; // No longer waiting
        }
        AW++;
        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 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?
            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 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?
            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 waiting!
release(&lock);
AccessDBase(ReadOnly);
acquire(&lock);
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);

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 waiting!

• What if we erase the condition check in Reader exit?
AR--;
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);

• Further, what if we turn the signal() into broadcast()
AR--;
if (AR == 0 && WW > 0) // No other active readers
    cond_broadcast(&okToWrite);// Wake up one writer

• 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: \textit{okContinue}

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

\textbf{Writer()}: \\
\begin{verbatim}
// check into system acquire(&lock); \\
while ((\texttt{AR} + \texttt{WW} > 0) { \\
  \texttt{WW}++; \\
  cond_wait(\texttt{okContinue},&lock); \\
} \\
\texttt{WR}++; \\
release(&lock); \\
\end{verbatim}

Use of Single CV: \textit{okContinue}

\textbf{Reader()}: \\
\begin{verbatim}
// check into system acquire(&lock); \\
// read-only access AccessDbase(ReadOnly); \\
// check out of system acquire(&lock); \\
\texttt{AR}++; \\
if (\texttt{AR} == 0 \&\& \texttt{WW} > 0) \\
  \texttt{cond_signal(\texttt{okContinue});} \\
else if (\texttt{WW} > 0) \\
  cond_broadcast(\texttt{okContinue}); \\
release(&lock); \\
\end{verbatim}

\textbf{Writer()}: \\
\begin{verbatim}
// check into system acquire(&lock); \\
// read/write access AccessDbase(ReadWrite); \\
// check out of system acquire(&lock); \\
\texttt{WW}++; \\
if (\texttt{WW} > 0) \\
  \texttt{cond_signal(\texttt{okContinue});} \\
else if (\texttt{WR} > 0) \\
  cond_broadcast(\texttt{okContinue}); \\
release(&lock); \\
\end{verbatim}

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

Use of Single CV: \textit{okContinue}

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

\textbf{Writer()}: \\
\begin{verbatim}
// check into system acquire(&lock); \\
while ((\texttt{AR} + \texttt{WW} > 0) { \\
  \texttt{WW}++; \\
  cond_wait(\texttt{okContinue},&lock); \\
} \\
\texttt{WR}++; \\
release(&lock); \\
\end{verbatim}

**Administrivia**

- **Midterm 1:** Thursday (October 1st): 5-7PM
  - We understand that this partially conflicts with CS170, but those of you in CS170 can start the exam after 7PM (according to CS170 staff)
  - Video Proctored, No curve, Use of computer to answer questions
  - More details as we get closer to exam
- **Midterm topics:**
  - Everything from lecture up to today's lecture
  - Scheduling is not part of exam...
  - Homework and Project work is fair game
- **Midterm Review:** Tomorrow, 7-9pm
  - Details TBA
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) {
    if (semaphore queue is not empty)
      semaV(thesema);
  }
  ```

- Doesn't work: Wait() may sleep with lock held
- Does this work better?
  ```
  Wait(Lock *thelock, Semaphore *thesema) {
    release(thelock);
    semaP(thesema);
    acquire(thelock);
  }
  Signal(Semaphore *thesema) {
  }
  ```

- No: Condition vars have no history, semaphores have history:
  - What if thread signals and no one is waiting?
    NO-OP
  - What if thread later waits?
  - What if thread V's and no one is waiting?
  - What if thread later does P?

Construction of Monitors from Semaphores (cont'd)

- 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 *thesema) {
    release(thelock);
    semaP(thesema);
    acquire(thelock);
  }
  Signal(Semaphore *thesema) {
    if (semaphore 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?

Mesa Monitor Conclusion

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

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know all the code paths out of a critical section
  ```
  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, poping stack back to procedure B
    » If Procedure C had lock.acquire, problem!
Concurrence and Synchronization in C

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

- Is goto a solution???
  ```
  void Rtn() {
    lock1.acquire();
    if (error) {
      goto release_lock1_and_return;
    } else {
      lock2.acquire();
      if (error) {
        goto release_both_and_return;
      } else {
        release_both_and_return:
          lock2.release();
        release_lock1_and_return:
          lock1.release();
      }
      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:
    ```
    void Rtn() {
      lock.acquire();
      try {
        DoFoo();
      } catch (...) { // catch exception
        lock.release(); // re-release the lock
        throw; // re-throw the exception
      }
      lock.release();
    }
    ```

  Notice that an exception in DoFoo() will exit without releasing the lock!

Much better: C++ Lock Guards

```
#include <mutex>

int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
  std::lock_guard<std::mutex> lock(global_mutex);
  global_i++;
}
```
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 (be aware of 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();

Recall: User/Kernel Threading Models

- Simple One-to-One Threading Model
- Many-to-One
- Many-to-Many

Almost all current implementations
Recall: Thread State in the Kernel

- For every thread in a process, the kernel maintains:
  - The thread’s TCB
  - A kernel stack used for syscalls/interrupts/traps
    - This kernel state is sometimes called the “kernel thread”
    - The “kernel thread” is suspended (but ready to go) when thread is running in user-space
- Additionally, some threads just do work in the kernel
  - Still has TCB
  - Still has kernel stack
  - But not part of any process, and never executes in user mode

In Pintos, Processes are Single-Threaded

- Pintos processes have only one thread
- TCB: Single page (4 KiB)
  - Stack growing from the top (high addresses)
  - struct thread at the bottom (low addresses)
- struct thread defines the TCB structure and PCB structure in Pintos

(Aside): Linux “Task”

- Linux “Kernel Thread”: 2 pages (8 KiB)
  - Stack and thread information on opposite sides
  - Containing stack and thread information + process descriptor
- One task_struct per thread

Multithreaded Processes (not in Pintos)

- Traditional implementation strategy:
  - One PCB (process struct) per process
  - Each PCB contains (or stores pointers to) each thread’s TCB
- Linux’s strategy:
  - One task_struct per thread
  - Threads belonging to the same process happen to share some resources
    - Like address space, file descriptor table, etc.
- To what extent does this actually matter?
Aside: Polymorphic Linked Lists in C

- Many places in the kernel need to maintain a “list of X”
  - This is tricky in C, which has no polymorphism
  - Essentially adding an interface to a package
- In Linux and Pintos this is done by embedding a list_elem in the struct
  - Macros allow shift of view between object and list
  - You saw this in Homework 1

Kernel Structure So Far

- These two threads:
  - Are used internally by the kernel
  - Don’t correspond to any particular user thread or process
Recall: Multithreaded Stack Example

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

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

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

Recall: Kernel Crossing on Syscall or Interrupt

Recall: Context Switch – Scheduling

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, syscalls)

- Mechanism to resume k-thread goes through interrupt vector

Kernel → User

- Interrupt return (iret) restores user stack, IP, and PL
The document includes sections on interrupt handling and kernel processing. Here are the key points:

- **Interrupt Processing**: The process starts with a hardware interrupt, which is handled by a generic handler. This handler saves the current context and calls the appropriate interrupt handler function.

- **Kernel Environment**: The kernel environment is set up, and the process is switched to the kernel thread for the interrupted process.

- **Interrupt Vector**: Interrupts are transferred through the Interrupt Vector (IDT in x86), and `iret` restores the user stack and priority level (PL).

- **Pintos Interrupt Processing**: The document includes diagrams showing the flow of control through interrupts and the process of switching from user mode to kernel mode.

- **IntrNN_stub()**: This is a wrapper function for generic handlers, which sets up the kernel environment and calls the appropriate interrupt handler.

- **Intr_entry**: This function saves the current context and invokes the `intr_handler` function.

- **Intr_exit**: This function restores the saved registers and returns using `iret`.
Timer may trigger thread switch

- **thread_yield**
  - Updates thread counters
  - If quanta exhausted, sets yield flag
- **thread_tick**
  - Calls schedule to select next thread to run upon iret
  - Returns back to intr_handler
  - Pushes it back on ready_list
  - Calls schedule to select next thread to run upon iret
- **Schedule (Next Lecture!**
  - 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

Pintos Return from Processing

- iret restores user stack and priority level (PL)
- Kernel → Different User Thread
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
• Next time: we dive into scheduling!

Recall: Address Space

• Program operates in an address space that is distinct from the physical memory space of the machine

Understanding "Address Space"

• Page table is the primary mechanism
• Privilege Level determines which regions can be accessed
  – Which entries can be used
• System (PL=0) can access all, User (PL=3) only part
• Each process has its own address space
• The "System" part of all of them is the same

All system threads share the same system address space and same memory
Page Table Mapping (Rough Idea)

Translation Map 1

Translation Map 2

Physical Address Space

User Process View of Memory

Process Virtual Address Space

Physical Memory

Aside: x86 (32-bit) Page Table Entry

- Controls many aspects of access
- Later – discuss page table organization
  - For 32 (64?) bit VAS, how large? vs size of memory?
  - Used sparsely

Pintos: page_dir.c
User → Kernel

### Process Virtual Address Space

- Kernel
- User data
- User code
- Stack
- Heap

### Physical Memory

- Page Table

### Page Table Resides in Memory*

* In the simplest case. Actually more complex. More later.

### Kernel Portion of Address Space

- Kernel memory is mapped into address space of every process
- Contains the kernel code
  - Loaded when the machine booted
- Explicitly mapped to physical memory
  - OS creates the page table
- Used to contain all kernel data structures
  - Lists of processes/threads
  - Page tables
  - Open file descriptions, sockets, ttys, ...
- Kernel stack for each thread
1 Kernel Code, Many Kernel Stacks

Recall: I/O and Storage Layers

Layers of I/O...

Many different types of I/O
Recall: Internal OS File Description

- Internal Data Structure describing everything about the file
  - Where it resides
  - Its status
  - How to access it

- Pointer: `struct file *file`
  - Everything accessed with file descriptor has one of these

- `struct file_operations *f_op`:
  - Describes how this particular device implements its operations
    - For disks: points to file operations
    - For pipes: points to pipe operations
    - For sockets: points to socket operations

File_operations: Why everything can look like a file

- Associated with particular hardware device or environment (i.e. file system)
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations

```c
def ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    } else
        inc_syscr(current);
    return ret;
}
```

Linux: `fs/read_write.c`
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!file->f_flags & FMODE_READ) return -EBADF;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret > 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        fsnotify_access(file->f_path.dentry);
        add_rchar(current, ret);
        inc_syscr(current);
    }
    return ret;
}

• Check if file has read methods

Check whether we write from a valid range in the file.

If driver provide a read function (f_op->read) use it; otherwise use do_sync_read()
File System: From Syscall to Driver

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!((file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || !((file->f_op->read && !file->f_op->aio_read)))
        return -EINVAL;
    if (unlikely(access_ok(VERIFY_WRITE, buf, count)) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret > 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}

Notify the parent of this file if the file was read (see http://www.fieldses.org/~bfields/kernel/vfs.txt)

Linux: fs/read_write.c

File System: From Syscall to Driver

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!((file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || !((file->f_op->read && !file->f_op->aio_read)))
        return -EINVAL;
    if (unlikely(access_ok(VERIFY_WRITE, buf, count)) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret > 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}

Update the number of bytes read by "current" task (for scheduling purposes)

Linux: fs/read_write.c

File System: From Syscall to Driver

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!((file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || !((file->f_op->read && !file->f_op->aio_read)))
        return -EINVAL;
    if (unlikely(access_ok(VERIFY_WRITE, buf, count)) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret > 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}

Update the number of read syscalls by "current" task (for scheduling purposes)

Linux: fs/read_write.c

Device Drivers

- **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call

- Device Drivers typically divided into two pieces:
  - **Top half**: accessed in call path from system calls
    - Implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - This is the kernel's interface to the device driver
    - Top half will start I/O to device, may put thread to sleep until finished
  - **Bottom half**: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete
### Life Cycle of An I/O Request

1. **User Program**
   - User program requests I/O.
   - System call to kernel.
   - Kernel processes request.

2. **Kernel I/O Subsystem**
   - Kernel sends request to device driver.
   - Device driver executes.

3. **Device Driver**
   - Device driver processes request.
   - Device driver completes I/O.

4. **Device Hardware**
   - Device hardware processes I/O request.

### 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
  - Monitors supported natively in a number of languages

- **Readers/Writers Monitor** example
  - Shows how monitors allow sophisticated controlled entry to protected code

- **Kernel Thread**: Stack + State for independent execution in kernel
  - Every user-level thread paired one-to-one with kernel thread
  - Kernel thread associated with user thread is “suspended” (ready to go) when user-level thread is running

- **Device Driver**: Device-specific code in kernel that interacts directly with device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers