CS162
Operating Systems and
Systems Programming
Lecture 9

Synchronization 4: Monitors and Readers/Writers (Con’t),
Process Structure, Device Drivers

February 16th, 2021
Profs. Natacha Crooks and Anthony D. Joseph
http://cs162.eecs.Berkeley.edu
Recall: Atomic Read-Modify-Write

- **test&set (&address) { /* most architectures */**
  
  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:
    li r1, M[address];
    movi r2, 1; // Can do arbitrary computation
    sc r2, M[address];
    beqz r2, loop;
}
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!
    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: 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_WAKE
    - Atomic check that condition still holds after we disable interrupts (in kernel!)
  - 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

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

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

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

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

- 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!
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:

  ```
  lock
  while (need to wait) {
    condvar.wait();
  }
  unlock
  
  do something so no need to wait
  ```

  ```
  lock
  condvar.signal();
  unlock
  ```
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 data base
    - Check out - wake up a waiting writer
  - **Writer()**
    - Wait until no active readers or writers
    - Access database
    - Check out - wake up waiting readers or writer
  - **State variables (Protected by a lock called “lock”):**
    - int AR: Number of active readers; initially = 0
    - int WR: Number of waiting readers; initially = 0
    - int AW: Number of active writers; initially = 0
    - int WW: Number of waiting writers; initially = 0
    - Condition okToRead = NIL
    - Condition okToWrite = NIL
Recall: Code for a Reader

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
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 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 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);
}
```

```c
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

```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);    // Now we are active!
    AccessDBase(ReadOnly);
    acquire(&lock);
    AR--;             // Now we are active!
    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);`

```c
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) {
        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--;  // No. Readers exist
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--; // Now we are active!
    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

```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--; // No. Readers exist
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

```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--; // No. Readers exist
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);
}
```
Simulation of Readers/Writers Solution

- R1 and R2 accessing dbase
- AR = 2, 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);
}
```  

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

```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--;
    }
    AW++;
    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 comes along (R1 and R2 are still accessing dbase)
- AR = 2, WR = 0, 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);
    AW--;  
    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

```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--;
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);
```

```c
acquire(&lock);
AW--;  // No longer waiting
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

```c
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) {
        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--;
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
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++; // 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--; // No. Readers exist
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
Simulation of Readers/Writers Solution

• R3 comes along (R1 and R2 accessing database, 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!
lock.release();

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!
    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

```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);
}
```

```c
AccessDBase(ReadOnly);
```

```c
acquire(&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

```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);
```
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);
    AR--; // No. Readers exist
    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);
    AR--;                // No. Readers exist
    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--;  // Signal for writing
    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) {
        WR++;
        cond_wait(&okToRead,&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

```c
AccessDBase(ReadOnly);
acquire(&lock);
AR--; // Corrected
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);
    AR--;                 // Enter critical section
    if (AR == 0 && WW > 0) // Wake up writers
        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

```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--;            // No. Readers exist
    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

```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);
}
```
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++;
    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

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);
    AW--; //
    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);
}
```

```c
AccessDBase(ReadWrite);
```

```c
acquire(&lock);
AW--;  
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);
}
```

```c
acquire(&lock);
AW--; //
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

```cpp
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);
  AW--;
  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

```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);
AW--;  
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

```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);
}
```

```c
acquire(&lock);
AW--;  // AW--
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

```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--; // Now we are active!
    cond_wait(&okToWrite); // Sleep on cond var
    release(&lock);
}
```
Simulation of Readers/Writers Solution

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

```c
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) {
        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--; // No writers
    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

```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);
}
```

```c
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

```c
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) {
        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--;  // Updated: Added this line
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

```cpp
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);
```

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

• Can readers starve? Consider Reader() entry code:

```c
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?

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

• Further, what if we turn the signal() into broadcast()

```c
AR--; // 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()**

```c
// 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);
AR--;
if (AR == 0 && WW > 0)
    cond_signal(&okContinue);
release(&lock);
```

**Writer()**

```c
// 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);
AW--;
if (WW > 0)
    cond_signal(&okContinue);
else if (WR > 0) {
    cond_broadcast(&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);
    AR--;
    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);
    AW--;
    if (WW > 0){
        cond_signal(&okContinue);
    } else if (WR > 0) {
        cond_broadcast(&okContinue);
    }
} else {
    // Let WR and WW be the number of writers and readers, respectively.
    // When all readers and writers have finished, the lock is released.
    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)
Use of Single CV: \textit{okContinue}

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

    // read-only access
    AccessDatabase(ReadOnly);

    // check out of system
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_broadcast(&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
    AccessDatabase(ReadWrite);

    // check out of system
    acquire(&lock);
    AW--;
    if (WW > 0 || WR > 0){
        cond_broadcast(&okContinue);
    }
    release(&lock);
}
\end{Verbatim}
Administrivia

- Midterm 1: Thu February 18th, 5-6:30pm
  - Video Proctored, Use of computer to answer questions
  - More details as we get closer to exam
- Midterm topics:
  - Everything up to lecture 9 – lecture will be released early
  - Homework 1 and Project 1 (high-level design) are fair game
- Midterm Review: Tuesday February 16th, 5-7pm
Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?
  ```c
  Wait(Semaphore *thesema) { semaP(thesema); }
  Signal(Semaphore *thesema) { semaV(thesema); }
  ```

- Does this work better?
  ```c
  Wait(Lock *thelock, Semaphore *thesema) {
    release(thelock);
    semaP(thesema);
    acquire(thelock);
  }
  Signal(Semaphore *thesema) {
    semaV(thesema);
  }
  ```

- Doesn't work: Wait() may sleep with lock held
- Does this work better?

  » What if thread signals and no one is waiting?
  
  » What if thread later waits?
  
  » What if thread V's and no one is waiting?
  
  » What if thread later does P?

- No: Condition vars have no history, semaphores have history:
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?
  ```c
  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?
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:

```java
lock
while (need to wait) {
    condvar.wait();
}
unlock

do something so no need to wait

lock
condvar.signal();
unlock
```

Check and/or update state variables
Wait if necessary

Check and/or update state variables
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!

```text
Proc A
Proc B
Calls setjmp
Proc C
acquire(&lock)
Proc D
Proc E
Calls longjmp
```
Concurrent 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();
    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;
    }
    ...
    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:
    ```
    void Rtn() {
        lock.acquire();
        try {
            ...
            DoFoo();
            ...
        } catch (...) { // catch exception
            lock.release(); // release lock
            throw; // re-throw the exception
        }
        lock.release();
    }
    void 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

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

    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();`
Recall: User/Kernel Threading Models

Almost all current implementations

Simple One-to-One Threading Model

Many-to-One

Many-to-Many
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

Pintos: `thread.c`
(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

Pintos: list.c
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

- User code
- User stack
- User stack
- Kernel code
- Kernel thread stack
- Kernel thread stack
- TCB
- TCB
- TCB
- PTBR
- PTBR
- PTBR

Time

syscall / interrupt

Saves

Schedule

Ready to resume

Switch kernel threads

Pintos: switch.S
Kernel Structure So Far (1/3)
Kernel Structure So Far (2/3)
These two threads:
• Are used internally by the kernel
• Don’t correspond to any particular user thread or process
• 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
• Mechanism to resume k-thread goes through interrupt vector
Kernel → User

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

intrNN_stub()

push 0x20 (int #)
jmp intr_entry

push 0x21 (int #)
jmp intr_entry

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

intr_exit:
restore regs
iret

Wrapper for
generic handler

stubs.S
• Interrupt transfers control through the Interrupt Vector (IDT in x86)
• iret restores user stack and priority level (PL)
Switch to Kernel Thread for Process
Pintos Interrupt Processing

**intrNN_stub()**

- push 0x20 (int #)
- jmp intr_entry

**intr_entry:**
- save regs as frame
- set up kernel env.
- call intr_handler

**intr_exit:**
- restore regs
- iret

---

**Wrapper for generic handler**

- Intr_handler(*frame)
  - classify
  - dispatch
  - ack IRQ
  - maybe thread yield

**intr_handlers**

- timer_intr(*frame)
  - tick++
  - thread_tick()

---

**Hardware interrupt vector**

- 0x20
  - 0
  - push 0x20 (int #)
  - jmp intr_entry

- 255
  - push 0x21 (int #)
  - jmp intr_entry
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 (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`
Resume Some Thread

intrNN_stub()

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

intr_exit:
    restore regs
    iret

Interrupt Handler

intr_entry stub

Push 0x20 (int #)
Jmp intr_entry

Push 0x20 (int #)
Jmp intr_entry

intr_exit stub

Push 0x20 (int #)
Jmp intr_entry

Wrapper for generic handler

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

intr_exit:
    restore regs
    iret

intr_handlers

Pintos

interrupt.c

Intr_handler(*frame)
    - classify
    - dispatch
    - ack IRQ
    - maybe thread yield

timer_intr(*frame)
    tick++
    threadTick()

thread_yield()
    - schedule

schedule()
    - switch

Hardware interrupt vector

stubs.S

Pintos Return from Processing
**Kernel → Different User Thread**

- `iret` restores user stack and priority level (PL)
Famous Quote WRT Scheduling: Dennis Richie

Dennis Richie, Unix V6, slp.c:

```
2230 * If the new process paused because it was
2231 * swapped out, set the stack level to the last call
2232 * to savu(u_ssav). This means that the return
2233 * which is executed immediately after the call to aretu
2234 * actually returns from the last routine which did
2235 * the savu,
2236 *
2237 * You are not expected to understand this.
```

“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)

Physical Address Space

Translation Map 1

- Code
- Data
- Heap
- Stack

Translation Map 2

- Code
- Data
- Heap
- Stack

(user process view of memory)

Prog 1
Virtual Address Space 1

Data 2
Stack 1
Heap 1
Code 1
Stack 2
Data 1
Heap 2
Code 2
OS code
OS data
OS heap/Stacks

Prog 2
Virtual Address Space 2

Code
Static Data
Heap
Stack

0x000...

0xFFF...
User Process View of Memory

Process Virtual Address Space

Kernel

argv

stack

heap

user data

user code

Processor registers

sp

ip

Physical Memory

Page Table

Page
Processor Mode (Privilege Level)

Process Virtual Address Space

Processor registers

- sp
- ip

CPL: 3 - user

0x00000000

0xc0000000

0x08048000

0xffffffff

kernel

argv

stack

heap

user data

user code

Page Table

0x08048000

0x00000000

Physical Memory

Page
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

Processor registers

sp

ip

CPL: 3 - user

0x00000000

argv

user data

user code

0xc0000000

stack

heap

kernel

0xffffffff

Page Table

Physical Memory
User → Kernel

Process Virtual Address Space

- kernel
- ker data
- ker code
- argv
- stack
- heap
- user data
- user code

Processor registers

- sp
- ip

CPL: 0 - sys

0x00000000
0x08048000
0x08040000

Physical Memory

Page Table

- u/s

0xffffffff
0xc0000000

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
Kernel Code, Many Kernel Stacks

Process Virtual Address Space

- kernel
- ker data
- ker code
- argv
- stack
- heap
- user data
- user code

Processor registers
- sp
- ip

CPL: 0 - sys

PTBR: [empty]

Page Table
- u/s

Physical Memory

0x00000000
0x08048000
0xffffffff
0xc0000000
argv
Recall: I/O and Storage Layers

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

Streams

File Descriptors

open(), read(), write(), close(), …

Open File Descriptions

Files/Directories/Indexes

Commands and Data Transfers

Disks, Flash, Controllers, DMA

What we've covered so far…

What we'll peek at next
**Layers of I/O...**

```c
length = read(input_fd, buffer, BUFFER_SIZE);

ssize_t read(int, void *, size_t) {
    marshal args into registers
    issue syscall
    register result of syscall to rtn value
}

void syscall_handler (struct intr_frame *f) {
    unmarshall call#, args from regs
    dispatch : handlers[call#](args)
    marshal results fo syscall ret
}

ssize_t vfs_read(struct file *file, char __user *buf,
                 size_t count, loff_t *pos) {
    User Process/File System relationship
call device driver to do the work
}
```

**Exception U→K, interrupt processing**
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
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct ioVEC *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct ioVEC *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, ffile_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]}
```
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;
}
File System: From Syscall to Driver

```c
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;
}
```

Make sure we are allowed to read this file
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;
}
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;
}
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);
            inc_syscr(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
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;
}
### File System: From Syscall to Driver

```c
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 that the file was read (see [http://www.fieldses.org/~bfields/kernel/vfs.txt](http://www.fieldses.org/~bfields/kernel/vfs.txt))

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_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;
}
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;
}
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

User Program
- request I/O
- can already satisfy request?
  - yes
    - return from system call
  - no
    - send request to device driver, block process if appropriate

Kernel I/O Subsystem
- determine which I/O completed, indicate state change to I/O subsystem
- transfer data (if appropriate) to process, return completion or error code

Device Driver
- Top Half
  - process request, issue commands to controller, configure controller to block until interrupted
- Bottom Half
  - receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
  - monitor device, interrupt when I/O completed
  - generate interrupt

Device Hardware
- Hardware
- I/O completed, input data available, or output completed

Time

2/16/21 Crooks & Joseph CS162 © UCB Spring 2021 Lec 9.128
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