Synchronization

CS 416: Operating Systems Design, Spring 2011
Department of Computer Science
Rutgers University
Synchronization

Basic problem:

Threads are concurrently accessing shared variables
The access should be controlled for predictable result.

Solution: Need a mechanism to control the access

Low-Level Mechanism
- Locks

Higher Level Mechanism
- Semaphores
- Mutexes
- Condition Variables
- Monitors
Higher-Level Synchronization primitives

Locks are useful for mutual exclusion. But programs have different requirements.

Examples:

• Say we had a shared variable where any number of threads could read but only one thread could write.

• How would you do this with locks?

```
Reader() {
    lock.acquire();
    mycopy = shared_var;
    lock.release();
    return mycopy;
}
```

```
Writer() {
    lock.acquire();
    shared_var = NEW_VALUE;
    lock.release();
}
```

• What’s wrong with this code?
Semaphores

Semaphore

• Higher level construct
• Shared Counter

Operations on Semaphores:
P() or wait() or down()

• Atomically wait for semaphore value to become > 0, then decrement it

V() or signal() or up()

• Atomically increments semaphore by 1
Semaphore Example

Semaphores can be used to implement Mutual Exclusion

Semaphore my_semaphore = 1 // Initialize to nonzero
int withdraw(account, amount) {
    P(my_semaphore)
    balance = get_balance()
    balance = balance - amount;
    put_balance(balance, account);
    V(my_semaphore)
}

A semaphore where the counter value is only 0 or 1 is called a binary semaphore. A Binary Semaphore similar to lock

Rutgers University
Simple Semaphore Implementation

```c
struct semaphore {
    int val;
    thread_list waiting;  // List of threads waiting for semaphore
}

P(semaphore Sem):  // Wait until > 0 then decrement
    while (Sem.val <= 0) {
        add this thread to Sem.waiting;
        block(this thread);  // What does this do??
    }
    Sem.val = Sem.val -1;
    return;

V(semaphore Sem):  // Increment value and wake up next thread
    Sem.val = Sem.val + 1;
    if (Sem.waiting is nonempty) {
        remove a thread T from Sem.waiting;
        wakeup(T);
    }
```

What is wrong with the above code?
Simple Semaphore Implementation

```c
struct semaphore {
    int val;
    thread_list waiting; // List of threads waiting for semaphore
}

P(semaphore Sem): // Wait until > 0 then decrement
    while (Sem.val <= 0) {
        add this thread to Sem.waiting;
        block(this thread); // What does this do??
    }
    Sem.val = Sem.val - 1;
    return;

V(semaphore Sem): // Increment value and wake up next thread
    Sem.val = Sem.val + 1;
    if (Sem.waiting is nonempty) {
        remove a thread T from Sem.waiting;
        wakeup(T);
    }
```

P() and V() must be atomic actions!
Semaphore Implementation

How do we ensure that the semaphore operations are atomic?

This is similar to Lock:

One Approach: Make them *System Calls* and ensure only one P() or V() operation can be executed by any process at a time

• This effectively puts a lock around the P() and V() operations

• Since system calls are executed in privileged mode, interrupts could be disabled to preserve atomicity

Second Approach: Use hardware support:

• Say the CPU had atomic P() and V() operations
Why are semaphores any better than Lock?

- A binary semaphore is basically a lock

- The real value of Semaphores becomes apparent when the counter can be initialized to a value other than 0 or 1

- Say we initialize a semaphore’s value to 50
  - What does this mean about P() and V() operations?
The Producer/Consumer Problem

Producer pushes items into a buffer
Consumer pulls items from the buffer
Producer needs to wait when buffer is full
Consumer needs to wait when the buffer is empty
One Implementation

int count=0 -> Shared Variable

Producer(){
    int item;
    while(TRUE) {
        item = produce();
        if(count==N) sleep();
        insert_item(item);
        count = count + 1;
        if(count == 1)
            wakeup(consumer);
    }
}

Consumer(){
    int item;
    while(TRUE) {
        if(count==0)
            sleep();
        item = remove_item();
        count = count-1;
        if(count == N-1)
            wakeup(producer);
        consume(item);
    }
}

What is the problem with this code ?

Context Switching ? – Lost wakeup problem !
A Fix using Semaphore

Semaphore mutex=1;
Semaphore empty=N;
Semaphore full=0;

Producer {
    int item;
    while(TRUE) {
        item=produce();
        P(empty);
        p(mutex);
        insert_item(item);
        v(mutex);
        v(full);
    }
}

Consumer {
    while(TRUE){
        p(full);
        p(mutex);
        item = remove_item();
        v(mutex);
        v(empty);
        consume(item);
    }
}

Does the order matter?
Readers/Writers Problem

• Want any number of threads to read simultaneously
• But only one thread should be able to write to an object at a time

```
semaphore wrt = 1
int readcount = 0

reader() {
    readcount ++;
    if(readcount == 1)
        p(wrt);
    do_read();
    readcount --;
    if(readcount == 0 )
        V(wrt);
}

writer() {
    P(wrt)
    do_write()
    V(wrt)
}
```

Where is the race condition?
Readers/Writers Fixed

```c
semaphore mutex = 1
semaphore wrt = 1
int readcount = 0

writer() {
    P(wrt)
    do_write()
    V(wrt)
}

reader() {
    p(mutex)
    readcount ++;
    if(readcount == 1)
        p(wrt);
    }
    v(mutex)

do_read();
    p(mutex)
    readcount --;
    if(readcount ==0 )
        V(wrt);
    v(mutex)
}
```

This is also called: First Readers Writers Problem

- Can lead to **writer starvation**
Second Readers/Writers Problem

-No writer should starve (The readers could starve).
-If a writer is waiting, no new reader can enter the shared memory

```c
semaphore mutex = 1
semaphore wrt = 1, read = 1
semaphore mutex2 = 1, mutex3 = 1
int readcount = 0
int writecount = 0

reader() {
  p(mutex3)
  p(read)
  p(mutex)
  readcount++;
  if(readcount == 1)
    p(wrt);
  v(mutex)
  v(read)
  V(mutex3)
}

writer() {
  P(mutex2)
  writecount++
  if(writecount == 1)
    p(read)
  V(mutex2)
  P(wrt)
  do_write()
  V(wrt)
  P(mutex2)
  writecount--;
  if(writecount == 0)
    v(read)
}

Only when all writers finish, the writer releases the read lock
```
Issues with Semaphore

Unlike locks, P() and V() do not have to be paired

Therefore, it is a lot easier to get into trouble with semaphore

• User needs to ensure its correctness

Wouldn’t it be nice if we had a clean, well-defined language support for synchronization.

• Java does!
Every Java object can be used as a mutex

```java
Object foo;
    synchronized (foo) {
        /* Do some stuff with 'foo' locked... */
        foo.counter++;
    }
```

Compiler ensures that the lock is released before exiting the synchronized block – Even if there is an exception.

```java
try {
    synchronized (foo) {
        if (foo.doSomething() == false) {
            throw new Exception("Bad!!");
        }
    }
}
catch (Exception e) {
    /* Lock was released before getting here! */
    System.err.println("Something bad happened!");
}
```
A *conditional variable* represents some condition that a thread can:

- *Wait on*, until the condition occurs
- *Notify*, other waiting threads that the condition has occurred

**Three operations on Condition Variables**

- `wait()`: Block on the condition variable
- `notify()`: Wake up one thread waiting on a CV
- `notifyAll()`: Wake up all threads waiting on a CV
Revisiting Producer/Consumer

int count=0 -> Shared Variable

Producer(){
    int item;
    while(TRUE) {
        item = produce();
        lock->acquire()
        if(count==N) sleep();
        insert_item(item);
        count = count + 1;
        if(count == 1)
            wakeup(consumer);
        lock->release()
    }
}

Consumer(){
    int item;
    while(TRUE) {
        lock->acquire()
        if(count==0)
            sleep();
        item = remove_item();
        count = count-1;
        if(count == N-1)
            wakeup(producer)
        lock->release()
        consume(item);
    }
}
Producer/Consumer Fix-1

Producer(){
    int item;
    while(TRUE) {
        item = produce();
        lock->acquire()
        if(count==N){
            lock->release()
            sleep();
        }
        insert_item(item);
        count = count + 1;
        if(count == 1)
            wakeup(consumer);
        lock->release()
    }
}

Consumer(){
    int item;
    while(TRUE) {
        lock->acquire()
        if(count==0){
            lock->release()
            sleep();
        }
        item = remove_item();
        count = count-1;
        if(count == N-1)
            wakeup(producer)
        lock->release()
        consume(item);
    }
}

int count=0 -> Shared Variable

What's wrong with this?