

## Lecture #10: Synchronization wrap up

\*\*\*\*\*

### Review -- 1 min

\*\*\*\*\*

Monitor = lock + condition variables

Mesa v. Hoare semantics

Advice/Summary

### Fall 2001 midterm:

- Every program with incorrect semantic behavior violated at least one rule
- >90% of programs that violated at least one rule were “obviously” semantically incorrect (that is, I could see the bug within seconds of looking at the program; there may have been additional bugs...)
  - All that violate one rule are *wrong* – they are harder to read, understand, maintain

\*\*\*\*\*

### Outline - 1 min

\*\*\*\*\*

Readers/Writers

Monitors v. Semaphores

Concurrency Summary

\*\*\*\*\*

### Preview - 1 min

\*\*\*\*\*

Other issues – scheduling, deadlock

\*\*\*\*\*

### Lecture - 20 min

\*\*\*\*\*

# 1. Readers/Writers

## 1.1 Motivation

Shared database (for example, bank balances, or airline seats)

Two classes of users:

**Readers** – never modify database

**Writers** – read and modify data

Using a single mutex lock would be overly restrictive.

Instead, want:

many readers at same time

only one writer at same time

## 1.2 Constraints

Notice: for every constraint, there is a synchronization variable.

This time different types for different purposes.

- 1) Reader can access database when no writers (Condition okToRead)
- 2) Writers can access database when no readers or writers (condition okToWrite)
- 3) Only one thread manipulates shared variables at a time (mutex)

## 1.3 Solution

Basic structure

Database::read()

check in -- wait until no writers

access database

check out – wake up waiting writer

Database::write()

check in -- wait until no readers or writers

access database

check out – wake up waiting readers or writers

State variables:

```

AR = 0; // # active readers
AW = 0; // # active writers
WR = 0; // # waiting readers
WW = 0; // # waiting writers

```

```

Condition okToRead = NIL;
Condition okToWrite = NIL;
Lock lock = FREE;

```

Code:

```

Database::read(){
    startRead(); // first, check self into the system
    Access Data
    doneRead(); // Check self out of system
}

Database::startRead(){
    lock.Acquire();
    while((AW + WW) > 0){
        WR++;
        okToRead.Wait(&lock);
        WR--;
    }
    AR++;
    lock.Release();
}

Database::doneRead(){
    lock.Acquire();
    AR--;
    if(AR == 0 && WW > 0){ // if no other readers still
        okToWrite.Signal(); // active, wake up writer
    }
    lock.Release();
}

Database::write(){ // symmetrical
    startWrite(); // check in
    accessData

```

```

        doneWrite(); // check out
    }

Database::startWrite(){
    lock.Acquire();
    while((AW + AR) > 0){ // check if safe to write
        // if any readers or writers, wait
        WW++;
        okToWrite->Wait(&lock);
        WW--;
    }
    AW++;
    lock.Release();
}

Database::doneWrite(){
    lock.Acquire();
    AW--;
    if(WW > 0){
        okToWrite->Signal(); // give priority to writers
    }
    else if (WR > 0){
        okToRead->Broadcast();
    }
    lock.Release();
}

```

### Question

- 1) Can readers starve?
- 2) Why does checkRead need a while?
- 3) Suppose we had a large DB with many records, and we want many users to access it at once. Probably want to allow two different people to update their bank balances at the same time, right? What are issues?

## 2. Example: Sleeping Barber (Midterm 2002)

The shop has a barber, a barber chair, and a waiting room with `NCHAIRS` chairs. If there are no customers present, the barber sits in the barber chair and falls asleep. When a customer arrives, he wakes the sleeping barber. If an additional customer arrives while the barber is cutting hair, he sits in a waiting room chair if one is available. If no chairs are available, he leaves the shop. When the barber finishes cutting a customer's hair, he tells the customer to leave; then, if there are any customers in the waiting room he announces that the next customer can sit down. Customers in the waiting room get their hair cut in FIFO order.

The barber shop can be modeled as 2 shared objects, a `BarberChair` with the methods `napInChair()`, `wakeBarber()`, `sitInChair()`, `cutHair()`, and `tellCustomerDone()`. The `BarberChair` must have a state variable with the following states: `EMPTY`, `BARBER_IN_CHAIR`, `LONG_HAIR_CUSTOMER_IN_CHAIR`, `SHORT_HAIR_CUSTOMER_IN_CHAIR`. Note that neither a customer or barber should sit down until the previous customer is out of the chair (`state == EMPTY`). Note that `cutHair()` must not return until the customer is sitting in the chair (`LONG_HAIR_CUSTOMER_IN_CHAIR`). And note that a customer should not get out of the chair (e.g., return from sit in chair) until his hair is cut (`SHORT_HAIR_CUSTOMER_IN_CHAIR`). The barber should only get in the chair (`BARBER_IN_CHAIR`) if no customers are waiting. **You may need additional state variables.**

The `WaitingRoom` has the methods `enter()` which immediately returns `WR_FULL` if the waiting room is full or (immediately or eventually) returns `MY_TURN` when it is the caller's turn to get his hair cut, and it has the method `callNextCustomer()` which returns `WR_BUSY` or `WR_EMPTY` depending on if there is a customer in the waiting room or not. Customers are served in FIFO order.

Thus, each customer thread executes the code:

```
Customer(WaitingRoom *wr, BarberChair *bc)
{
    status = wr->enter();
    if(status == WR_FULL){
        return;
    }
    bc->wakeBarber();
    bc->sitInChair(); // Wait for chair to be EMPTY
                    // Make state LONG_HAIR_CUSTOMER_IN_CHAIR
                    // Wait until SHORT_HAIR_CUSTOMER_IN_CHAIR
                    // then make chair EMPTY and return
    return;
}
```

The barber thread executes the code:

```
Barber(WaitingRoom *wr, BarberChair *bc)
{
    while(1){ // A barber's work is never done
        status = wr->callNextCustomer();
        if(status == WR_EMPTY){
            bc->napInChair(); // Set state to BARBER_IN_CHAIR; return with state EMPTY
        }
        bc->cutHair(); // Block until LONG_HAIR_CUSTOMER_IN_CHAIR;
                     // Return with SHORT_HAIR_CUSTOMER_IN_CHAIR
        bc->tellCustomerDone(); // Return when EMPTY
    }
}
```

Write the code for the `WaitingRoom` class and the `BarberChair` class. Use locks and condition variables for synchronization and follow the coding standards specified in the handout.

**Hint and requirement reminder:** remember to start by asking for each method “when can a thread wait?” and writing down a synchronization variable for **each** such situation.

List the member variables of class **WaitingRoom** including their type, their name, and their initial value

| Type         | Name               | Initial Value (if applicable) |
|--------------|--------------------|-------------------------------|
| <i>mutex</i> | <i>lock</i>        |                               |
| <i>cond</i>  | <i>cond</i>        |                               |
| <i>int</i>   | <i>nfull</i>       | 0                             |
| <i>int</i>   | <i>ticketAvail</i> | 0                             |
| <i>int</i>   | <i>ticketTurn</i>  | -1                            |

```
int WaitingRoom::custEnter()
    lock.acquire();
    int ret;
    if(nfull == NCHAIRS){
        ret = WR_FULL;
    }
    else{
        ret = MY_TURN;
        myTicket = ticketAvail++;
        nfull++;
        while(myTicket > ticketTurn){
            cond.wait(&lock);
        }
        nfull--;
    }
    lock.release();
    return ret;
```

```
int WaitingRoom::callNextCustomer()
    lock.acquire();
    ticketTurn++;
    if(nfull == 0){
        ret = EMPTY;
    }
    else{
        ret = BUSY;
        cond.broadcast();
    }
    lock.release();
    return ret;
```

List the member variables of class **BarberChair** including their type, their name, and their initial value

| Type         | Name                | Initial Value (if applicable) |
|--------------|---------------------|-------------------------------|
| <i>mutex</i> | <i>lock</i>         |                               |
| <i>cond</i>  | <i>custUp</i>       |                               |
| <i>cond</i>  | <i>barberGetUp</i>  |                               |
| <i>cond</i>  | <i>sitDown</i>      |                               |
| <i>cond</i>  | <i>seatFree</i>     |                               |
| <i>cond</i>  | <i>cutDone</i>      |                               |
| <i>int</i>   | <i>state</i>        | EMPTY                         |
| <i>int</i>   | <i>custWalkedIn</i> | 0                             |

```
void BarberChair::napInChair()
    lock.acquire();
    if(custWalkedIn == 0){ // Cust could arrive before I sit down
        state = BARBER_IN_CHAIR;
    }
    while(custWalkedIn == 0){
        barberGetUp.wait(&lock);
    }
    custWalkedIn = 0;
    if(state == BARBER_IN_CHAIR){ // Cust could have beaten us
        state = EMPTY
        seatFree.signal(&lock);
    }
    lock.release();
```

```
void BarberChair::wakeBarber()
    lock.acquire();
    custWalkedIn = 1;
    barberGetUp.signal(&lock);
    lock.release()
```

```
void BarberChair::sitInChair()
    lock.acquire()
    while(state != EMPTY){
        seatFree.wait(&lock);
    }
    state = LONG_HAIR_CUSTOMER_IN_CHAIR;
    sitDown.signal(&lock);
    while(state != SHORT_HAIR_CUSTOMER_IN_CHAIR){
        cutDone.wait(&lock);
    }
    state = EMPTY;
    custUp.signal(&lock);
    lock.release();
}
```

```
void BarberChair::cutHair()
    lock.acquire();
    while(state != LONG_HAIR_CUSTOMER_IN_CHAIR){
        sitDown.wait(&lock);
    }
    state = SHORT_HAIR_CUSTOMER_IN_CHAIR;
    cutDone.signal(&lock);
```

```
lock.release();
```

```
void BarberChair::tellCustomerDone()
```

```
lock.acquire();
```

```
while(state != EMPTY){ // NOTE: No other cust can arrive until I call call_next_cust()
```

```
    custUp.wait(&lock);
```

```
}
```

```
lock.release();
```

### 3. Semaphores v. Condition variables

Illustrate the difference by considering: can we build monitors out of semaphores? After all, semaphores provide atomic operations and queuing.

Does this work:

```
Wait(){ semaphore->P() }
Signal{ semaphore->V() }
```

No: Condition variables only work inside a lock. If try to use semaphores inside a lock, have to watch for deadlock.

Does this work:

```
Wait(Lock *lock){
    lock->Release();
    semaphore->P();
    lock->Acquire();
}

Signal(){
    semaphore->V();
}
```

Condition variables have no history, but semaphores do have history.

What if thread signals and no one is waiting?

→ No Op

What if thread later waits?

→ Thread waits.

What if thread V's and no one is waiting?

Increment

What if thread later does P

## Decrement and continue

In other words, P+V are commutative – result is the same no matter what order they occur. Condition variables are not commutative. That's why they must be in a critical section – need to access state variables to do their job.

Does this fix the problem?

```
Signal(){
    if semaphore queue is not empty
        semaphore->V();
}
```

For one, not legal to look at contents of semaphore queue. Also, race condition – signaller can slip in after lock is released and before wait. Then waiter never wakes up

Need to release lock and go to sleep atomically.

Is it possible to implement condition variables using semaphores?  
Yes, but exercise left to the reader!

### 3.1

\*\*\*\*\*

Admin - 3 min

\*\*\*\*\*

Project 1 out...

Notes:

- don't assume FIFO behavior of Locks and CV's
- Implementation must provide minimal fairness to threads calling scheduler – freedom from starvation – eventually all waiting threads are **guaranteed** to make progress

- GDB with threads on Linux sees “unknown signal” – anyone know workaround? (“handle” call?)
- Use `gettimeofday` to get current time (but document how this limits precision of scheduling)
- A comment “multiple flows not supported” in my sender test program is not correct; multiple flows are supported. Sorry for the confusion
- Slight ambiguity – given permission to send v. when do you really send. OK to assume you send as soon as you get permission to send (I thought about more complex interface to deal with this, but not worth the trouble...)

■

\*\*\*\*\*

Lecture - 23 min

\*\*\*\*\*

## 4. Concurrency conclusion

### 4.1 Summary

Basic idea in all CS: abstract complexity behind clean interfaces

We’ve done that!!

#### **Physical Hardware**

single CPU, interrupts, test&set

#### **Programming Abstraction**

sequential execution

infinite # CPUs

semaphores and monitors

**Every** major OS built since 1985 has provided threads – Linux, Mach, OS/2, NT (Microsoft), Solaris, OSF (Dec alphas)

Why? B/c makes it a lot easier to write concurrent programs, from Web servers to databases to embedded systems

So does this mean you should all go out and use threads?

## 4.2 Cautionary tales

Illustrate why abstraction doesn't always work the way you want it to

### 4.2.1 OS/2

Microsoft OS/2 (around 1988): initially, a spectacular failure. Since then IBM has completely re-written from scratch

Use threads for everything: window systems, communication between programs, etc. Threads are good idea, right?

Thus, system created a lot of threads, but few actually running at any one time – most waiting around for user to type in a window, or for a network packet to arrive, etc.

Might have 90 threads, but just a few at any time on the ready queue, but each thread needs its own execution stack, say 9K, whether runnable or waiting

Result: system needs an extra **1 MB** of memory, mostly consumed by waiting threads. 1 MB of memory cost \$200 in 1988

Put yourself in customer's shoes; Did OS/2 run Excel or Word better? OK, it gave you the ability to keep working while you use the printer, but is that worth \$200?

**Moral: threads are cheap, but they're not free**

Who are OS features for?

Operating system developer?

End user?

Lots of OS research has been focused on making it easier for OS **developers**, because it is so complicated to build operating systems.

But the trick to selling it is to make it better for the **end user**.

## 4.2.2 Threads and Multiprocessors

Might think you have everything you need to know to go write a parallel program: just split program into threads, so that things can run in parallel

Example: matrix multiply

```
for(I = 0; I < N; I++){
    for(j = 0; j < N; j++){
        for(k = 0; k < N; k++){
            C[I][j] += A[I][k] * B[j][k];
        }
    }
}
```

How would you parallelize this? Create a thread for every iteration of the inner loop? Each one can run concurrently, using a lock to protect access to each element in  $C[I][j]$ .

Would work, but wouldn't be efficient. In Nachos, a few hundred instructions to create a thread. Here, maybe a few ten instructions to do each iteration.

**Repeat: threads are cheap, but they aren't free**

Instead: group iterations so that each thread does a fair amount of work.

## 4.2.3 The case against threads

Several prominent operating systems researchers have argued that one should almost never use threads because (a) it is just too hard to write multi-threaded programs that are correct and (b) most things that threads are commonly used for can be accomplished in other, safer ways.

I think they may go too far, but there is more than a grain of truth in their arguments.

The class web page has pointers to two documents that may interest you:

John Ousterhout "Why Threads Are A Bad Idea (for most purposes)."

Robert van Renesse "Goal-Oriented Programming, or Composition using Events, or Threads Considered Harmful"

These are important arguments to understand -- even if you disagree with them, they may point out pitfalls that you can avoid.

#### 4.2.4 Event-driven v. thread-driven programming

They can express the same thing  
 I can build a multi-threaded server  
 I can build an event-driven server

Event queue ~ ready queue  
 Event ~ thread control block // how same, how different  
 Waiting for disk/object ~ waiting for lock/signal  
 Event loop ~ scheduler

QUESTION: what are advantages and disadvantages of different approaches? When should you use one and not the other?

\*\*\*\*\*

Summary - 1 min

\*\*\*\*\*