Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 5: Producers-Consumers and Readers-Writers

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Two types of synchronization

- Mutual exclusion with locks
  - Ensures that only one thread is in critical section
  - *“Not at the same time”*
  - lock/unlock
- Ordering constraints with condition variables
  - One thread waits for another to do something
  - *“Before after”*
  - E.g., dequeuer must wait for enqueuer to add something to queue

Condition variables

Need a way to go to sleep, consuming no resource while waiting for a condition.

But we can't lose any races, so part of it has to be atomic.

We do this with a *condition variable.*

```
Wait( lock )
    {
    release the lock;
    put the thread on the waiting list;
    sleep;
    wake when condition satisfied;
    retake the lock;
    }
```

Condition variables

Each condition variable has a list of waiting threads.

They're "waiting on the condition" meaning they're waiting for whatever condition you decide to associate with that condition variable, e.g., queue is empty, queue is full, or whatever.

```
Wait( lock )
   {
   release the lock;
   put the thread on the waiting list;
   sleep;
   wake when condition satisfied;
   retake the lock;
   }
```

Condition variables

You always use a condition variable in combination with a lock, releasing and then retaking the lock inside the condition variable's wait operation.

```
Wait( lock )
    {
    release the lock;
    put the thread on the waiting list;
    sleep;
    wake when condition satisfied;
    retake the lock;
    }
```

Condition variables

Since you're giving up the lock, you must guarantee that all the representation invariants of your datastructures have been restored.

```
Wait( lock )
    {
    release the lock;
    put the thread on the waiting list;
    sleep;
    wake when condition satisfied;
    retake the lock;
    }
```

Condition variables interface

wait(mutex)

- Atomically release lock, add thread to waiting list, sleep.
- Thread must hold the lock when calling wait( ).
- Must re-establish invariants before calling wait( ).

signal( )

- Wake up one thread waiting on this condition variable.

broadcast( )

- Wake up all threads waiting on this condition variable.

If no thread is waiting, signal and broadcast do nothing.

Avoiding busy waiting

So, let's rewrite these sections with a condition variable.

```
Enqueue( )
    lock
    add new item to tail of queue
    if (Dequeuer is waiting) {
        take waiting dequeuer off waiting list
        wake up dequeuer
    }
    unlock

Dequeue( )
    lock
    if (queue is empty) {
        add myself to waiting list
        sleep
    }
    remove item from queue
    unlock
```

We could give up the lock before sleeping, then retake when we wake up.

What is wrong with this code?

Another thread might beat us to it. So must always recheck the condition.

```
Enqueue( )
    lock
    add new item to tail of queue
    cv.signal( lock )
    unlock

Dequeue( )
    lock
    if ( queue is empty )
        cv.wait( lock )
    remove item from queue
    unlock
    return the removed item
```

To solve the race condition you must always, always check that the condition you hoped for is satisfied when you wake up by using a loop, not an if.

Another thread might beat us to it.

```
Enqueue( )
    lock
    add new item to tail of queue
    cv.signal( lock )
    unlock

Dequeue( )
    lock
    while ( queue is empty )
        cv.wait( lock )
    remove item from queue
    unlock
    return the removed item
```

Condition variables eliminate busy waiting and they free up the resource by releasing the lock while you're waiting but promise you'll get the lock back when wait returns.

```
Enqueue( )
    lock
    add new item to tail of queue
    cv.signal( lock )
    unlock

Dequeue( )
    lock
    while ( queue is empty )
        cv.wait( lock )
    remove item from queue
    unlock
    return the removed item
```

Spurious wakeups

There's clearly a race between when a cv is signaled and when you wake and another thread simply beating you to it. That's often called a "*stolen wakeup*".

But many definitions of cv's also allow wait to return for no reason whatsoever, even if never signaled, to allow implementation flexibility in dealing with error conditions and races inside the OS. That's called a "*spurious wakeup*".

The argument is you were going to have to check the condition anyway.

Monitors

Combine two types of synchronization

- Locks for mutual exclusion
- Condition variables for ordering constraints

A monitor = a lock + the condition variables associated with that lock

Mesa vs. Hoare monitors

Mesa monitors

- When waiter is woken, it must contend for the lock

We (and most OSes) use Mesa monitors

Waiter is solely responsible for ensuring condition is met

Hoare monitors

- Special priority to woken-up waiter
- Signaling thread immediately gives up lock
- Signaling thread reacquires lock after waiter unlocks

Typical monitor code

You use a lock and a condition variable together.

When you do something that creates a condition another thread might be waiting for, you signal it.

Other threads can then wake up. But they must always check that the condition is satisfied when they wake.

```
Enqueue( )
    lock
    add new item to tail of queue
    cv.signal( lock )
    unlock

Dequeue( )
    lock
    while ( queue is empty )
        cv.wait( lock )
    remove item from queue
    unlock
    return the removed item
```

Producer-consumer (bounded buffer)

Producers fill a shared buffer; consumers empty it
Need to synchronize actions of producers and consumers

Used in many situations
- Unix pipes (grep “keyword” foo.txt | wc -l)
- Project 1!
- Cafeterias

Why use a shared buffer?
- Lets producers and consumers operate somewhat independently

Serving hamburgers with monitors

Step 1: Identify shared state

- State of the hamburger serving area
- Number of burgers waiting for customers
- Probably have minimums and maximums

Step 2: Assign locks

- One lock to protect all shared data

Burgers with monitors

Step 3: Identify before-after conditions

- Before hamburger sold, at least 1 burger available now (or soon).
- Before cooking more burgers, you have to be below the minimum.

Step 4: Assign condition variables

- Consumer waits on *waitingConsumers* if no burgers.
- Cooks wait on *waitingProducers* if still enough burgers.

Coke machine with monitors

Consumer

```
burgerLock.lock( );

while ( burgers == 0 )
   waitingConsumers.wait( );

take hamburger;
burgers--;

waitingProducers.signal( );

burgerLock.unlock( );
```

Producer

```
burgerLock.lock( );

while ( burgers >= Minimum )
    waitingProducers.wait( );

make hamburger;
burgers++;

waitingConsumers.signal( );

burgerLock.unlock( );
```

Wait-signal pairing

Consumer

```
burgerLock.lock( );

while ( burgers == 0 )
   waitingConsumers.wait( );

take hamburger;
burgers--;

waitingProducers.signal( );

burgerLock.unlock( );
```

Producer

```
burgerLock.lock( );

while ( burgers >= Minimum )
    waitingProducers.wait( );

make hamburger;
burgers++;

waitingConsumers.signal( );

burgerLock.unlock( );
```

Looping while holding lock

Consumer

```
burgerLock.lock( );

while ( burgers == 0 )
   waitingConsumers.wait( );

take hamburger;
burgers--;

waitingProducers.signal( );

burgerLock.unlock( );
```

Producer

```
burgerLock.lock( );

while ( true )
   {
   sleep( 1 hour )
   while ( burgers >= Minimum )
       waitingProducers.wait( );

   make hamburger;
   burgers++

   waitingConsumers.signal( );
   }

burgerLock.unlock( );
```

Reducing number of signals

Consumer

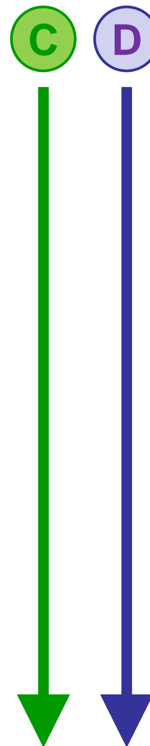
```
burgerLock.lock( );

while ( burgers == 0 )
    waitingConsumers.wait( );

take hamburger;
burgers--;

waitingProducers.signal( );

burgerLock.unlock( );
```



Producer

```
burgerLock.lock( );

while ( burgers >= Minimum )
    waitingProducers.wait( );

make hamburger;
burgers++;

if ( burgers == 1 )
    waitingConsumers.signal( );

burgerLock.unlock( );
```

A B

Why is this wrong?

Better to signal the condition *is* true than that it just *became* true.

Only one consumer will wake up.

Reducing condition variables

Consumer

```
burgerLock.lock( );

while ( burgers == 0 )
    anyoneWaiting.wait( );

take hamburger;
burgers--;

anyoneWaiting.signal( );

burgerLock.unlock( );
```

Producer

```
burgerLock.lock( )

while ( burgers < 1 )
    anyoneWaiting.wait( );

make hamburger;
burgers++;

anyoneWaiting.signal( );

burgerLock.unlock( );
```

F B A (at consumer `anyoneWaiting.wait( );`), E (at consumer `take hamburger;`), C (at producer `burgerLock.lock( )`), D (at producer `anyoneWaiting.wait( );`)

Why is this wrong?

D never wakes up because the signal was consumed at F.

Reducing condition variables

Consumer

```
burgerLock.lock( );

while ( burgers == 0 )
   anyoneWaiting.wait( );

take hamburger;
burgers--;

anyoneWaiting.broadcast( );

burgerLock.unlock( );
```

Producer

```
burgerLock.lock( )

while ( burgers < 1 )
    anyoneWaiting.wait( );

make hamburger;
burgers++;

anyoneWaiting.broadcast( );

burgerLock.unlock( );
```

To make this work, you need to use broadcast to wake everyone up.

Remarks on Project 1

I hope you've started with Project 1.

Due next Wednesday.

Beware of needing a signal inside a wait loop or the need to use broadcast rather than signal.

(Probably indicates a design error.)

```
while ( !condition )
    {
    cv.signal( );
    cv.wait( );
    }
```

Reader-writer locks

Recall: Threads need to lock to read shared data.

- Implication: No concurrent reads!

How to safely allow more concurrency?

Problem definition:

- Shared data will be read and written by multiple threads.
- Allow multiple readers, if no threads are writing data.
- A thread can write only when no other thread is reading or writing.

Need for reader-writer locks

Use of normal mutex locks limits concurrency.

Reader

```
lock( );
print catalog;
unlock( );
```

Writer:

```
lock( );
change catalog;
unlock( );
```

Reader-writer locks

Implement set of functions that a program can use to follow *"multiple-reader, single-writer"* constraint.

Pros and cons compared to normal mutex locks?

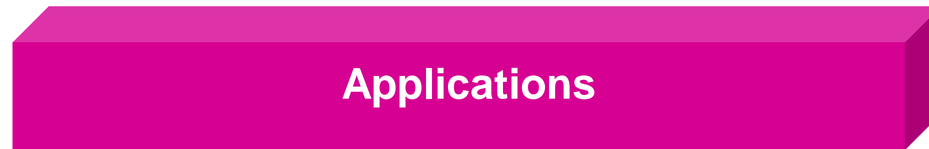
Reader

```
ReadLock( );
print catalog;
ReadUnlock( );
```

Writer:

```
WriteLock( );
change catalog;
WriteUnlock( );
```

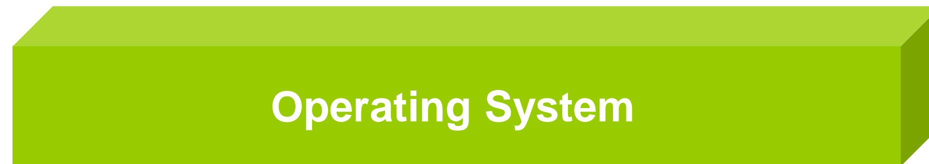
Another level of abstraction



Even higher-level synchronization primitives

Concurrent programs

(ReadLock, ReadUnlock, WriteLock, WriteUnlock)



Higher-level synchronization primitives

(lock, monitor, semaphore)

Atomic operations

(load/store, interrupt enable/ disable, test&set)

R/W lock with monitors

Step 1: What state is shared?

readers

writers

Step 2: Assign locks to shared state

rwLock

Step 3: What are the before-after conditions?

readers must wait if thread is writing

writers must wait if thread is reading or writing

Step 4: Assign condition variables

waitingReaders, waitingWriters

R/W lock with monitors

```
void ReadLock ( )
   {
   rwLock.lock( );
   while ( writers > 0 )
      waitingReaders.wait( );
   readers++;
   rwLock.unlock( );
   }
```

```
void ReadUnlock( )
   {
   rwLock.lock( );
   readers--;
   if ( readers == 0 )
      waitingWriters.signal( );
   rwLock.unlock( );
   }
```

```
void WriteLock( )
   {
   rwLock.lock( )
   while ( readers > 0 ||
         writers > 0 )
      waitingWriters.wait( );
   writers++;
   rwLock.unlock( );
   }
```

```
void WriteUnlock( )
   {
   rwLock.lock( );
   writers--;
   waitingReaders.broadcast( );
   waitingWriters.signal( );
   rwLock.unlock( );
   }
```

R/W lock with monitors

```
void ReadLock ( )
   {
   rwLock.lock( )
   while ( writers > 0 )
      waitingReaders.wait( );
   readers++;
   rwLock.unlock( );
   }
```

```
void ReadUnlock( )
   {
   rwLock.lock( )
   if ( readers == 1 )
      waitingWriters.signal( );
   readers--;
   rwLock.unlock( )
   }
```

```
void WriteLock( )
   {
   rwLock.lock( );
   while ( readers > 0 ||
         writers > 0 )
      waitingWriters.wait( );
   writers++;
   rwLock.unlock( );
   }
```

```
void WriteUnlock( )
   {
   rwLock.lock( );
   writers--;
   waitingReaders.broadcast( );
   waitingWriters.signal( );
   rwLock.unlock( );
   }
```

R/W lock with monitors

What will happen if a writer finishes and there are several waiting readers and writers?

- Will WriteLock return, or will 1 ReadLock return, or will all ReadLock return?

How long will a writer wait?

Could we give priority to a waiting writer?

Avoiding writer starvation

```
void ReadLock ( )
   {
   rwLock.lock( )
   while ( writers > 0 ||
         waitingWriters > 0 )
      waitingReaders.wait( );
   readers++;
   rwLock.unlock( );
   }
```

```
void WriteLock( )
   {
   rwLock.lock( );
   while ( readers > 0 ||
        writers > 0 )
     waitingWriters.wait( );
   writers++;
   rwLock.unlock( );
   }
```

```
void ReadUnlock( )
   {
   rwLock.lock( )
   if ( readers == 1 )
      waitingWriters.signal( );
   readers--;
   rwLock.unlock( )
   }
```

```
void WriteUnlock( )
   {
   rwLock.lock( );
   writers--;
   waitingReaders.broadcast( );
   waitingWriters.signal( );
   rwLock.unlock( );
   }
```

Programming with monitors

Key challenges in monitor programming:

- Adding more locks (deadlock!)
- Enforcing ordering/preventing starvation