Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 4: Lock and cv

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

Office hours survey

The staff is really concerned that some of you may be in very different timezones.

They'd like to do their best to meet your needs by scheduling OH at times that work for you.

Please take their When2Meet survey.

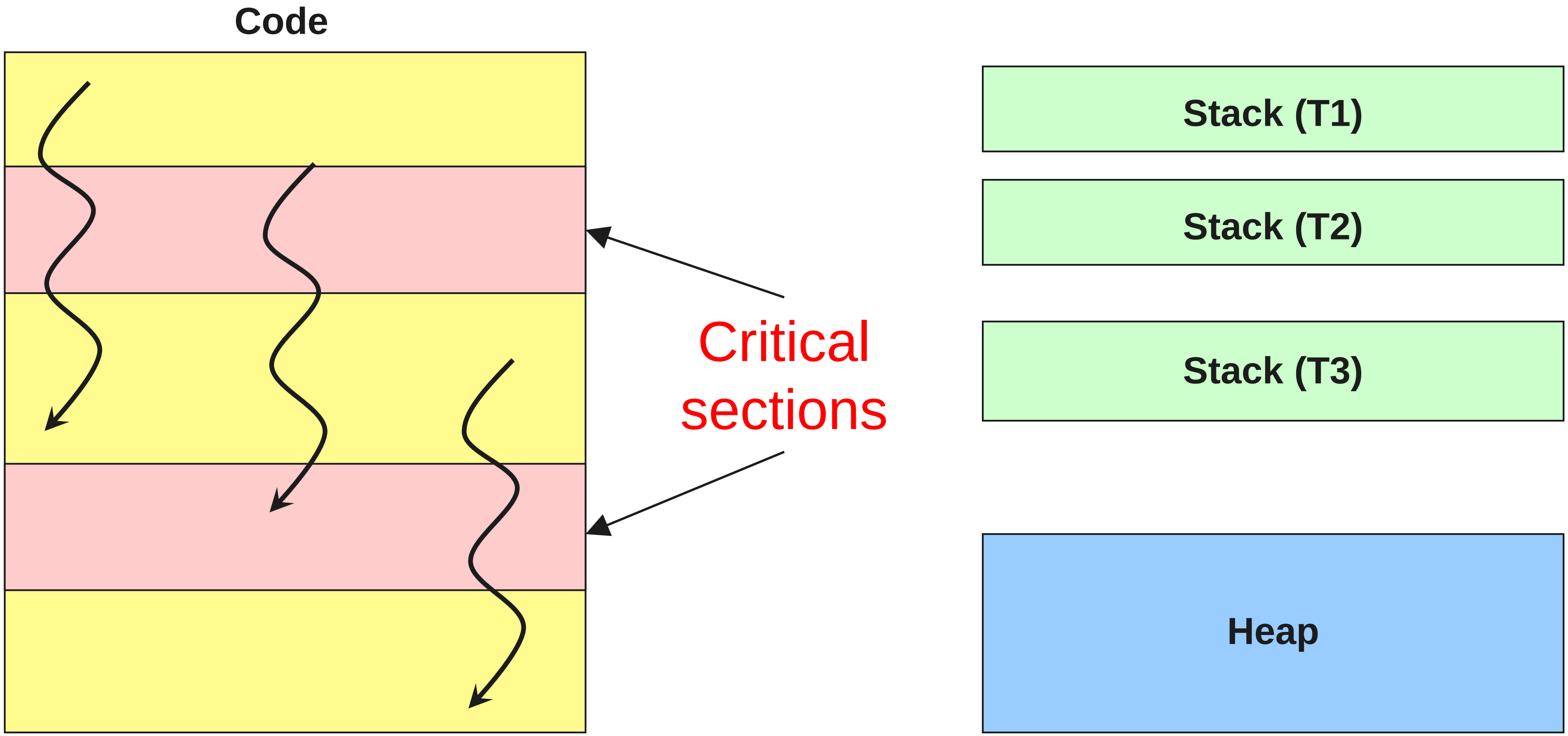
https://www.when2meet.com/?9130395-c3GGK

What to work on now

1. Please attempt the homework questions before lab on Friday.
2. Group declaration due on Friday.
3. If you don't choose, we will put you in a randomly-assigned group of 3 (which could include adding you to an existing group of 2.)
4. Read handout for Project 1 and try the autograder.
5. After today's lecture, we'll have covered all material to do the project.

Recap: Synchronization

Avoid race conditions via mutual exclusion



Code

Critical sections

Stack (T1)

Stack (T2)

Stack (T3)

Heap

Too much milk

Problem definition:

1. Obama family wants to always have one jug of milk.
2. No room for two jugs of milk.
3. Whoever sees the fridge empty goes to buy milk.

Solution 0, no synchronization.

Barack

```
if ( noMilk )
    buy milk;
```

Michelle

```
if ( noMilk )
    buy milk;
```

Problems?

Race condition!

Solution 3

Decide who buys milk when both leave notes at same time.

Barack hangs around to make sure job is done.

Barack's “while (noteMichelle)” prevents him from entering the critical section at the same time as Michelle.

Barack

```
leave noteBarack;
while ( noteMichelle )
   ;
if ( noMilk )
    buy milk;
remove noteBarack;
```

Michelle

```
leave noteMichelle;
if ( no noteBarack )
   if ( noMilk )
      buy milk;
remove noteMichelle;
```

Analysis of solution 3

Good

1. It works!
2. Relies on simple atomic operations.

Bad

1. Complicated and not obviously correct.
2. Asymmetric.
3. Not obvious how to scale to three people.
4. Barack consumes CPU time while waiting, called *busy-waiting.*

Barack

```
leave noteBarack;
while ( noteMichelle )
   ;
if ( noMilk )
    buy milk;
remove noteBarack
```

Michelle

```
leave noteMichelle;
if ( no noteBarack )
   if ( noMilk )
      buy milk;
remove noteMichelle;
```

Locks (mutexes)

A lock prevents another thread from entering a critical section

> *"Lock fridge while checking milk status and shopping"*

Two operations:

1. lock( ), wait until the lock is free, then acquire it.
2. unlock( ), release the lock.

Checking and acquiring must be atomic.

Why was the note solutions 1 and 2 not a good lock?

```
lock( )
    {
    while ( true )
        if ( lock is free )
            {
            acquire lock;
            break;
            }
    }

unlock( )
    {
    release lock;
    }
```

Atomic

Solution using locks

Lock usage:

1. Initialized to free.
2. Acquire lock before entering critical section.
3. Release lock when done with critical section.

All synchronization involves waiting.

Threads can be running or blocked.

Barack

```
milk.lock( );
if ( noMilk )
    buy milk;
milk.unlock( );
```

Michelle

```
milk.lock( );
if ( noMilk )
    buy milk;
milk.unlock( );
```

Efficiency

But this prevents Michelle from doing things while Barack is buying milk.

Can we minimize the time the lock is held?

Barack

```
milk.lock( );
if ( noMilk )
    buy milk;
milk.unlock( );
```

Michelle

```
milk.lock( );
if ( noMilk )
    buy milk;
milk.unlock( );
```

Efficiency

Use a lock to protect posting or viewing of any notes.

```
note.lock( );
if ( noNote )
   {
   leave note;
   note.unlock( );
   if ( noMilk )
      buy milk;
   note.lock( );
   remove note;
   }
note.unlock( );
```

Shared queue

Consider a simple queue.

```
class Queue
   {
   private:
      class Node
         {
         public:
            int data;
            Node *next;
            Node( int data );
            ~Node( );
         };

      Node *first, *last;

   public:
      void Enqueue( int data );
      int Dequeue( );
      bool Empty( );
      Queue( );
      ~Queue( );
   };
```

Shared queue

Let’s focus in on the Enqueue and Dequeue routines.

What could go wrong if it’s multi-threaded?

```
void Enqueue( int data )
   {
   Node *n = new Node( data );
   if ( last )
      last = last->next = n;
   else
      first = last = n;
   }

int Dequeue( )
   {
   assert( first );
   Node *p = first;
   int d = first->data;
   first = first->next;
   delete p;
   if ( !first )
      last = nullptr;
   return d;
   }
```

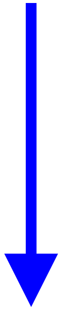
Shared queue

Suppose there was only one node on the list and we did this.

We'd link the new node to a just-deleted node.

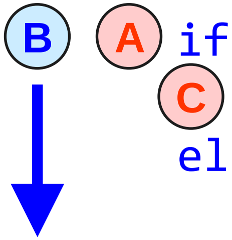
```
void Enqueue( int data )
   {
   Node *n = new Node( data );
Ⓐ  if ( last )
  Ⓒ   last = last->next = n;
   else
      first = last = n;
   }

int Dequeue( )
   {
   assert( first );
Ⓑ  Node *p = first;
   int d = first->data;
   first = first->next;
   delete p;
   if ( !first )
      last = nullptr;
   return d;
   }
```



Shared queue

In this failure mode, we'd throw away the node enqueued by the other thread.



```
void Enqueue( int data )
   {
   Node *n = new Node( data );
   if ( last )
      last = last->next = n;
   else
      first = last = n;
   }

int Dequeue( )
   {
   assert( first );
   Node *p = first;
   int d = first->data;
   first = first->next;
   delete p;
   if ( !first )
      last = nullptr;
   return d;
   }
```

Shared queue

Consider this pattern.

In this failure mode, we'd double-delete and one of the new nodes would be lost.

```
      void Enqueue( int data )
         {
         Node *n = new Node( data );
         if ( last )
            last = last->next = n;
         else
            first = last = n;
         }

      int Dequeue( )
         {
         assert( first );
B  A     Node *p = first;
         int d = first->data;
         first = first->next;
D  C     delete p;
         if ( !first )
            last = nullptr;
         return d;
         }
```

Shared queue

We need for Enqueue and Dequeue to be thread-safe.

We ensure that by adding a mutex (mutual exclusion) lock.

Any routine that wants to inspect or change the state should cooperate and first take the lock.

```
class Queue
   {
   private:
      class Node
         {
         public:
            int data;
            Node *next;
            Node( int data );
            ~Node( );
         };
      Node *first, *last;
      Mutex lock;

   public:
      void Enqueue( int data );
      int Dequeue( );
      bool Empty( );
      Queue( );
      ~Queue( );
   };
```

Shared queue

The design pattern is that we take the lock at the very beginning of these routines that contain critical sections and then release the lock at the end.

```
void Enqueue( int data )
   {
   lock.Lock( );
   Node *n = new Node( data );
   if ( last )
      last = last->next = n;
   else
      first = last = n;
   lock.Unlock( );
   }

int Dequeue( )
   {
   lock.Lock( );
   assert( first );
   Node *p = first;
   int d = first->data;
   first = first->next;
   delete p;
   if ( !first )
      last = nullptr;
   lock.Unlock( );
   return d;
   }
```

Shared queue

Avoid the temptation to release and retake the same lock multiple times in the same routine as that often introduces new race conditions.

Always lock at the beginning, release at the end.

```
void Enqueue( int data )
   {
   lock.Lock( );
   Node *n = new Node( data );
   if ( last )
      last = last->next = n;
   else
      first = last = n;
   lock.Unlock( );
   }

int Dequeue( )
   {
   lock.Lock( );
   assert( first );
   Node *p = first;
   int d = first->data;
   first = first->next;
   delete p;
   if ( !first )
      last = nullptr;
   lock.Unlock( );
   return d;
   }
```

Shared queue

Take the lock anytime you need for look at the object or change it.

Release the lock only when the representation invariant is maintained.

```
void Enqueue( int data )
   {
   lock.Lock( );
   Node *n = new Node( data );
   if ( last )
      last = last->next = n;
   else
      first = last = n;
   lock.Unlock( );
   }

int Dequeue( )
   {
   lock.Lock( );
   assert( first );
   Node *p = first;
   int d = first->data;
   first = first->next;
   delete p;
   if ( !first )
      last = nullptr;
   lock.Unlock( );
   return d;
   }
```

Fine-grained locking

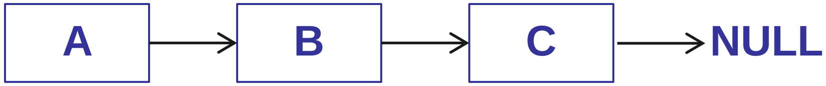
What if you only want to read the data, not make any changes?

Fine-grained locking

Instead of one lock for entire queue, use one lock per node

Why would you want to do this?

Lock each node as the queue is traversed, then release as soon as it’s safe, so other threads can also access the queue



1. lock A
2. get pointer to B
3. unlock A
4. lock B
5. read B
6. unlock B

Another thread could lock A and dequeue all nodes

What problems could occur?
How to fix?

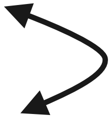
How to fix?

lock A

get pointer to B

lock B

unlock A



read B

unlock B

Hand-over-hand locking

- Lock next node before releasing last node
- Used in Project 4

Ordering constraints

What if you wanted Dequeue() to wait without holding the lock if the queue is empty?

Would this work?

```
int Dequeue( )
   {
   while ( !first )
      ;
   lock.Lock( );
   Node *p = first;
   int d = first->data;
   first = first->next;
   delete p;
   if ( !first )
      last = nullptr;
   lock.Unlock( );
   return d;
   }
```

Ordering constraints

Suppose we only look at first when we hold the lock.

Is the solution better?

Works (sort of) but involves busy-waiting that denies other the processor.

```
int Dequeue( )
   {
   lock.Lock( );
   while ( !first )
      {
      lock.Unlock( );
      lock.Lock( );
      }
   lock.Lock( );
   Node *p = first;
   int d = first->data;
   first = first->next;
   delete p;
   if ( !first )
      last = nullptr;
   lock.Unlock( );
   return d;
   }
```

Avoiding busy waiting

Have waiting dequeuer put itself onto a waiting list and then go to sleep.

The enqueuer wakes up sleeping dequeuer.

What could go wrong?

```
if (queue is empty )
   {
   add myself to waiting list;
   go to sleep;
   }
```

Avoiding busy waiting

What is wrong here?

Can’t go to sleep holding the lock.

```
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, but consider this failure mode.

```
Enqueue()
B   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) {
        unlock
    A   add myself to the waiting list
        sleep
    C   lock
    }
    remove item from queue
    unlock
```

Two types of synchronization

- Mutual exclusion ← Locks
  - Ensures that only one thread is in critical section
  - *“Not at the same time”*
  - lock/unlock
- Ordering constraints ← 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

Programming with monitors in P1

- Design
  - List the shared data needed for the problem
  - Assign locks to each group of shared data
    - Tradeoff between complexity and concurrency
  - List the waiting conditions for the problem
  - Assign condition variable to each condition
- Implementation
  - Add lock/unlock around all accesses to shared data
    - Remember invariant
  - Add while (!cond) { wait } where condition must hold
  - Add signal/broadcast after making condition true

Typical monitor code

You use a lock and a condition variable together.

When you do something that creates a condition a thread might be interested in, you signal it.

Other threads can then wait for that condition. 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
```

Project 1

Now, you should know everything you need to know to do project 1

Due soon. May 27.