

EECS 498 System Design of a Search Engine

Winter 2026

Lecture 9: Locks and condition variables

Nicole Hamilton

[https://web.eecs.umich.edu/~nham/
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

agenda

1. details
2. locks
3. condition variables
4. RAII
5. multi-reader/single-writer locks

agenda

1. details
2. locks
3. condition variables
4. RAI
5. multi-reader/single-writer locks

details

1. Would like to start having short meetings with each team starting with tomorrow's lab.
2. Questions about homework, reading, where you should be at this point, anything else?

agenda

1. details
2. locks
3. condition variables
4. RAII
5. multi-reader/single-writer locks

locks

1. All the threads share the same memory space. If one makes a change, it's instantly seen by the others.
2. Must avoid scribbling on data being read by another thread.
3. If a data structure is being updated, other threads should not be reading it until the update is finished.
4. They need a protocol for locking the data, establishing ownership.

locks

The problem is called *mutual exclusion*, meaning that if any is allowed, the rest are excluded.

On Linux, the mechanism we'll use is the pthread mutex.

atomic operations

Before we can reason at all about cooperating threads, we must know that some operation is *atomic*.

1. It's indivisible. It happens in its entirety or not at all.
2. No events from other threads can occur in between when it starts and when it finishes.

atomic operations

On most computers:

1. Memory load and store are atomic.
2. Many other instructions, e.g., double precision floating point, are not atomic.

Need an atomic operation to build bigger atomic operations.

example

Assume *i* is a global shared variable.

Thread A

```
i = 0;  
while ( i < 10 )  
    i++;  
print "A finished";
```

Thread B

```
i = 0;  
while ( i > -10 )  
    i--;  
print "B finished";
```

Which thread will exit its while loop first?

Is the thread that exits the while first guaranteed to print first?

Is it guaranteed that anything will print?

debugging multi-threaded programs

Challenging due to non-deterministic interleaving.

Heisenbug: a bug that occurs non-deterministically.

All possible interleavings must be correct.

synchronization

Objective:

Constrain interleavings between threads such that all possible interleavings produce a correct result.

Trivial solution:

Run each until it finishes before starting the next but that defeats the purpose of threads.

synchronization

Challenge:

Constrain thread executions as little as possible.

Insight:

Some events are independent → order is irrelevant.

Other events are dependent → order matters.

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( );
};
```

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;
}
```

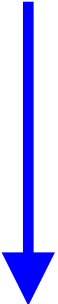
```
int Dequeue( )
{
    assert( first );
    Node *p = first;
    int d = first->data;
    first = first->next;
    delete p;
    if ( !first )
        last = nullptr;
    return d;
}
```

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 );
    A if ( last )
        C last = last->next = n;
    else
        first = last = n;
}
```

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



Consider this pattern.

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



```
void Enqueue( int data )
{
    Node *n = new Node( data );
    A if ( last )
    C 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;
}
```

Consider this pattern.

In this failure mode, we'd double-delete.

```
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;
}
```

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( );
};
```

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;
}
```

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;
}
```

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( );
}
```

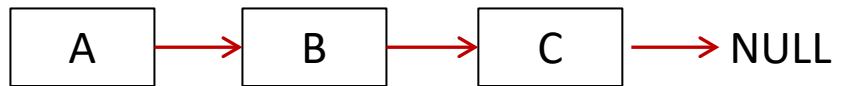
```
int Dequeue( )
{
    lock.Lock( );
    assert( first );
    Node *p = first;
    int d = first->data;
    first = first->next;
    delete p;
    if ( !first )
        last = nullptr;
    lock.Unlock( );
    return d;
}
```

fine-grained locking

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

Instead of one lock for entire queue, we can use one lock per node.

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

What could go wrong?

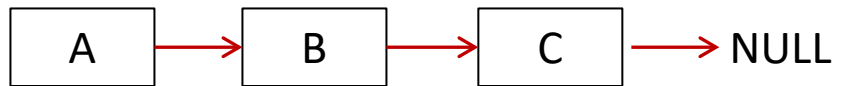
Another thread could lock A and dequeue all nodes

Fine-grained locking

Solution is to use hand-over-hand locking.

Lock the next node before releasing the lock on the last node.

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



locks on Linux

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,  
    const pthread_mutexattr_t *restrict attr);
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

pthread_mutex_init() initializes the mutex to an unlocked state. If *attr* is NULL, the default attributes are used

pthread_mutex_destroy() destroys the mutex object referenced by *mutex*.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated.

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

pthread_mutex_lock(), pthread_mutex_trylock(), and pthread_mutex_unlock() lock and unlock a mutex.

In the main thread:

```
pthread_mutex_t lock;  
  
pthread_mutex_init( &lock, nullptr );  
  
// Create lots of threads.  
  
pthread_create( ... );  
  
  
  
  
// Wait for them to finish.  
  
pthread_join( ... );  
  
pthread_mutex_destroy( &lock );
```

Within each child thread:

```
pthread_mutex_lock( &lock );  
  
// Read or write the shared  
data.  
  
pthread_mutex_unlock( &lock );
```

agenda

1. details
2. locks
3. condition variables
4. RAII
5. multi-reader/single-writer locks

Ordering constraints

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

Would this work?

No, first may be null
by the time we take
the lock.

```
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 the processor to other threads.

```
int Dequeue( )
{
    lock.Lock( );
    while ( !first )
    {
        lock.Unlock( );
        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 enqueueer 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  
      (C) sleep  
      lock  
  }  
  remove item from queue  
  unlock
```

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 enqueueer 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 data structures 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.broadcast( 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 the definition of a condition variable also allows 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.

pthread condition variables

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_destroy(pthread_cond_t *cond);
```

A condition variable allows threads to suspend execution until some predicate on shared data is satisfied.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

agenda

1. details
2. virtual memory
3. locks
4. RAII
5. multi-reader/single-writer locks

RAII

RAII = Resource Acquisition
Is Initialization.

Basic idea: Define a class
with a constructor that
acquires any lock you need.
The destructor frees the
lock.

Takes advantage of the C++
guarantee that the
destructor will always run,
even if a C++ exception is
thrown that causes the
block to exit.

```
// Usage.  
// Create the mutex.  
pthread_mutex_t mutex;  
pthread_mutex_init( &mutex, nullptr );  
  
// Lock when needed.  
if ( needLock )  
{  
    CriticalSection lock( &mutex );  
    // do something  
:  
    // Lock released automatically  
}
```

```
#include <pthread.h>
#include <cassert>
```

```
class CriticalSection
{
private:
    bool locked;
    pthread_mutex_t *mutex;

public:

    // Take the lock.
    void Take( )
    {
        assert( !locked );
        pthread_mutex_lock( mutex );
        locked = true;
    }

    // Release the lock.
    void Release( )
    {
        assert( locked );
        pthread_mutex_unlock( mutex );
        locked = false;
    }
};
```

```
// Constructor takes the lock.
CriticalSection( pthread_mutex_t *mutex ) :
    locked( false ), mutex( mutex )
{
    Take( );
}

// Destructor releases the lock.
~CriticalSection( pthread_mutex_t *mutex )
{
    if ( locked )
        Release( );
}
```

agenda

1. details
2. virtual memory
3. locks
4. RAII
5. multi-reader/single-writer locks

basic problem

1. You have object in memory you would like to share between threads.
2. It doesn't change very often but it gets read a lot.
3. Lots of threads can share access that object if they're all just reading it.
4. But if any thread wants to write to it, it must lock out all the other threads while it does that.

multi-reader / single-writer problem

It shows up on whiteboard interviews.

1. We'll assume some simple mutex and signaling facilities mapped to the OS.
2. I'll give you a chance to invent your own solution.
3. I'll show you some known solutions.

```
// Assume some basic Mutex and Signal mechanisms provided  
// by the OS that we might wrapper as follows.
```

```
class Mutex  
{  
private:  
    // ... a handle from the OS  
public:  
    // Simple mutual exclusion.  
    Take( );  
    Release( );  
    Mutex( );  
    ~Mutex( );  
};
```

```
class Signal
{
private:
    // ... a handle from the OS
public:
    // Simple signaling mechanism that can be set (true)
    // reset (false).

    // Wait for the signal to be set.
    // If it's not set, you sleep until it is.
    // Waiting for signal to be set does not reset it.
    // Other waiting threads will also wake up so long
    // as the signal remains set.

    void Wait( );

    // Set or reset it.
    void Set( );
    void Reset( );

    Signal( bool initialState = false );
    ~Signal( );
};
```

```
// Here is the multiple reader / single writer interface  
// to be built.
```

```
class SharedReader  
{  
public:  
    void ReadLock( ) = 0;  
    void ReleaseReadLock( ) = 0;  
    void WriteLock( ) = 0;  
    void ReleaseWriteLock( ) = 0;  
};
```

```
class Mutex
```

```
{  
    public:  
        Take( );  
        Release( );  
        Mutex( );  
        ~Mutex( );  
};
```

```
class SharedReader
```

```
{  
    public:  
        void ReadLock( ) = 0;  
        void ReleaseReadLock( ) = 0;  
        void WriteLock( ) = 0;  
        void ReleaseWriteLock( ) = 0;  
};
```

```
class Signal
```

```
{  
    public:  
        void Wait( );  
        void Set( );  
        void Reset( );  
        Signal( bool initialState = false );  
        ~Signal( );  
};
```

```
// This is the lock I created for my C shell in 1987,  
// rewritten in modern C++ with OS abstractions.  
  
// Assumes contention is low enough that writers are  
// not starved (which was true in my C shell.)  
  
// Requires readers to take and immediately release a  
// single lock to acquire or free the resource.  
// Writers simply take the lock to acquire and release  
// to free.
```

```

class HamiltonLock : SharedReader
{
private:
    // Mutex to control access to the number of readers.
    Mutex m;
    // Signal when the number of readers goes to zero.
    Signal zeroReaders;
    // Current number of readers
    int readers;

public:
    void ReadLock( )
    {
        m.Take( );
        readers++;
        zeroReaders.Reset( );
        m.Release( );
    }

    void ReleaseReadLock( )
    {
        m.Take( );
        if (--readers == 0)
            zeroReaders.Set( );
        m.Release( );
    }
}

```

```
void WriteLock( )
{
    while ( true )
    {
        m.Take( );
        if ( readers == 0 )
            break;
        m.Release( );
        zeroReaders.Wait( );
    }
}
```

```
ReleaseWriteLock( )
{
    m.Release( );
}
```

```
ReleaseWriteLock( )
{
    m.Release( );
}

HamiltonLock( ) : readers( 0 ), zeroReaders( true )
{
}

~HamiltonLock( )
{
}

};
```

```
// This is a variant on my lock created by Jordan Zimmerman  
// in 1999 that mitigates writer starvation by preventing  
// new readers from entering if there is a writer waiting.
```

```
// Readers take and release two locks to acquire.  
// Writers take two locks and release one to acquire.
```

```
class ZimmermanLock : SharedReadInterface
{
private:
    // Mutex to control access to the number of readers.
    Mutex m;

    // Mutex added to prevent new readers from entering
    // when writers are waiting.
    Mutex readlock;

    // Signal when the number of readers goes to zero.
    Signal zeroReaders;

    // Current number of readers
    int readers;
```

```
public:
    void ReadLock( )
    {
        readlock.Take( );
        m.Take( );
        readers++;
        zeroReaders.Reset( );
        m.Release( );
        readlock.Release( );
    }

    void ReleaseReadLock( )
    {
        m.Take( );
        if ( --readers == 0 )
            zeroReaders.Set( );
        m.Release( );
    }
```

```
void WriteLock( )
{
    readlock.Take( );
    while ( true )
    {
        m.Take( );
        if ( readers == 0 )
            break;
        m.Release( );
        zeroReaders.Wait( );
    }
    readlock.Release( );
}

void ReleaseWriteLock( )
{
    m.Release( );
}
```

```
ZimmermanLock( ) : readers( 0 ), zeroReaders( true )
{
}
~ZimmermanLock( )
{
}
};
```

```
// The 1997 GNU shared reader lock using cthreads
// and conditions rewritten as a modern C++ class.

// This lock refuses new readers if writers are waiting
// but requires both readers and writers to take two locks
// and immediately release one to acquire.

class GNULock : SharedReader
{
private:
    Mutex master;
    Condition wakeup;
    int readers,
        writersWaiting,
        readersWaiting;
```

```
public:
    void ReadLock( )
    {
        master.Take( );
        if ( reader == -1 || writersWaiting )
        {
            readersWaiting++;
            do
                ConditionWait( wakeup, master );
            while ( ; readers == -1 || writersWaiting )
        }
        reader++;
        master.Release( );
    }
```

```

void WriteLock( )
{
    master.Take( );
    if ( readers )
    {
        writersWaiting++;
        do
        {
            // Condition wait, atomically giving up the
            // mutex and waiting for wakeup signal, then
            // atomically taking the mutex.
            ConditionWait( wakeup, master );
        }
        while ( readers == -1 || writersWaiting );
        readersWaiting--;
    }
    readers++;
    master.Release( );
}

```

```
void ReleaseReadLock( )
{
    master.Take( );
    readers--;
    if ( readersWaiting || writersWaiting )
        // Broadcast the wakeup signal.
        ConditionBroadcast( wakeup );
    master.Release( );
}
```

```
void ReleaseWriteLock( )
{
    master.Take( );
    readers = 0;
    if ( readersWaiting || writersWaiting )
        // Broadcast the wakeup signal.
        ConditionBroadcast( wakeup );
    master.Release( );
}
```

```
GNULock( ) : readers( 0 ), readersWaiting( 0 ),  
            writersWaiting( 0  
            {  
            }  
  
~GNULock( );  
};
```