Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 8: Create, Join and Lock

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

Project 2

By the end of today, you should know everything needed to do project 2.

Work on the project incrementally.

1. Start with CPU and thread basics: cpu and thread constructors and yield.
2. Enable and disable interrupts for atomicity and implement mutex and cv.
3. Add support for multi-processors (optional.)

Use lots of asserts and create test cases.

Project 2

Covered everything you need to know to implement all of the project

Work on the project incrementally

- CPU and thread basics (cpu::init, thread constructor, and yield)
- Enable/disable interrupts for atomicity
- Implement mutex and cv
- Add support for multi-processors

Reminders

Do homework questions on semaphores before the lab on Friday.

Honor code:

- It's okay to discuss or lookup questions related to problems, project specs, or C++ syntax.
- Not okay to discuss solutions!

Recap

How to handle non-running threads?

Save private state in TCB to resume execution later.

How to switch between threads?

Transfer control from current thread to OS.

Save state of current thread and load state of next thread.

Creating a new thread

What state should a new thread be put into?

Recall: When a thread is paused, its state is put in ready queue.

Implication:

When creating a thread, construct its TCB as if it had been paused at the start of its main procedure.

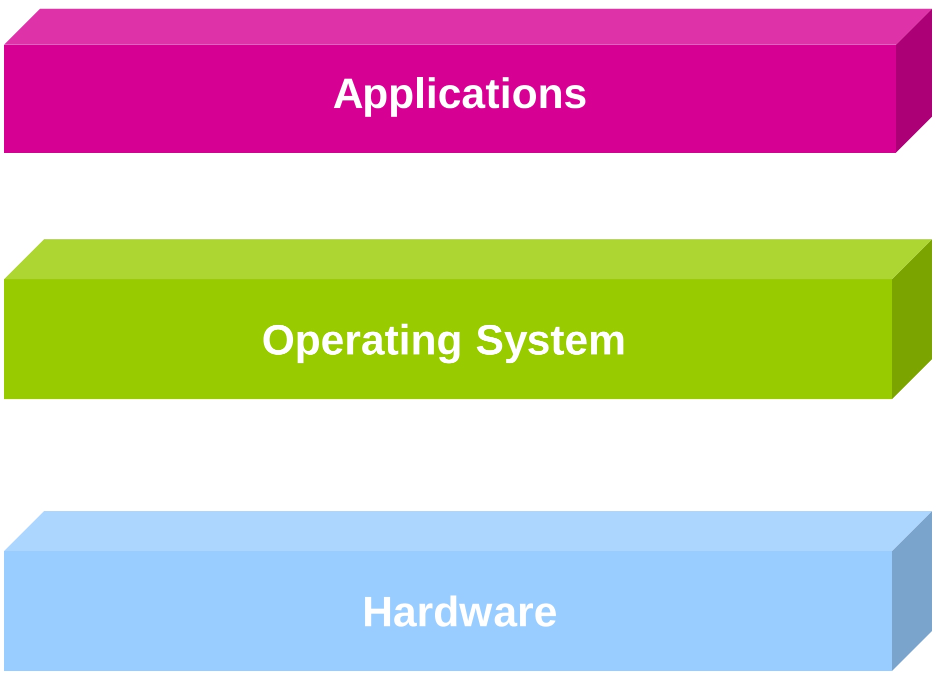
Creating a new thread

Steps

1. Allocate and initialize new thread control block.
2. Allocate and initialize new stack. In Project 2, this is done via *makecontext( ).*
3. Add thread control block to ready queue.

High-level synchronization

Raise the level of abstraction to make life easier for programmers



Concurrent programs

High-level synchronization primitives
(lock, monitor, semaphore)

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

Implementing high-level synchronization primitives

Data structures used must be thread-safe.

Cannot use high-level synchronization primitives.

Need to use atomic operations provided by hardware.

Atomicity on a uniprocessor

Potential approach if single CPU:

> Prevent context switches during an operation by preventing events that cause context switches.

Example: Disable interrupts to ensure atomicity.

```
disable interrupts;
if ( no milk )
   buy milk;
enable interrupts;
```

Problems?

```
disable interrupts
while ( 1 )
   ;
```

Should not allow interrupts to be disabled in user code.

Lock implementation #1

```
lock( )
   {
   disable interrupts;
   while ( status != FREE )
      {
      enable interrupts;
      disable interrupts;
       }
   status = BUSY;
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   status = FREE;
   enable interrupts;
   }
```

What is wrong with this?
It’s busy waiting. It would be better to give up the processor.

Busy waiting

Problem with lock implementation #1:

- Waiting thread uses lots of CPU time just checking for lock to become free.
- Better for thread to sleep and let other threads run.

Solution: Integrate lock implementation with thread dispatch:

- Have lock manipulate thread queues.
- Waiting thread gives up CPU, so other threads can run.
- Someone wakes up thread when lock is free.

Lock implementation #2

```
lock( )
   {
   disable interrupts;
   if ( status == FREE )
      status = BUSY;
   else
      {
      add thread to queue of
         threads waiting for lock;
      switch to next ready thread;
      }
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   status = FREE;
   if ( any thread is waiting
         for this lock )
      {
      move waiting thread to
         ready queue;
      status = BUSY;
      }
   enable interrupts;
   }
```

Lock implementation #2

```
lock( )
   {
   disable interrupts;
   if ( status == FREE )
      status = BUSY;
   else
      {
      add thread to queue of
         threads waiting for lock;
      switch to next ready thread;
      }
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   status = FREE;
   if ( any thread is waiting
         for this lock )
      {
      move waiting thread to
         ready queue;
      status = BUSY;
      }
   enable interrupts;
   }
```

We don’t want to busy wait.

Lock implementation #2

```
lock( )
   {
   disable interrupts;
   if ( status == FREE )
      status = BUSY;
   else
      {
      add thread to queue of
         threads waiting for lock;
      switch to next ready thread;
      }
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   status = FREE;
   if ( any thread is waiting
         for this lock )
      {
      move waiting thread to
         ready queue;
      status = BUSY;
      }
   enable interrupts;
   }
```

When we run again, the lock will have been taken for us.
This is called a hand-off pattern that guarantees that only the waiter gets the lock.

Okay to sleep with interrupts disabled?

```
lock( )
   {
   disable interrupts;
   if ( status == FREE )
      status = BUSY;
   else
      {
      add thread to queue of
         threads waiting for lock;
      switch to next ready thread;
      }
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   status = FREE;
   if ( any thread is waiting
         for this lock )
      {
      move waiting thread to
         ready queue;
      status = BUSY;
      }
   enable interrupts;
   }
```

How does it work that we are going to sleep with interrupts disabled?
Could we re-enable the interrupts before doing that?

Would this work?

```
lock( )
   {
   disable interrupts;
   if ( status == FREE )
      status = BUSY;
   else
      {
      enable interrupts;
      add thread to queue of
         threads waiting for lock;
      switch to next ready thread;
      }
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   status = FREE;
   if ( any thread is waiting
         for this lock )
      {
      move waiting thread to
         ready queue;
      status = BUSY;
      }
   enable interrupts;
   }
```

The other thread holding the lock could release it before we put ourselves on the waiting list. We'd possibly sleep forever.

Would this work?

```
lock( )
   {
   disable interrupts;
   if ( status == FREE )
      status = BUSY;
   else
      {
      add thread to queue of
         threads waiting for lock;
      enable interrupts;
      switch to next ready thread;
      }
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   status = FREE;
   if ( any thread is waiting
         for this lock )
      {
      move waiting thread to
         ready queue;
      status = BUSY;
      }
   enable interrupts;
   }
```

What happens if we get a timer interrupt?

Problem scenario

Thread A is attempting to acquire mutex.

Thread B holds the mutex and is about to release it.

1. lock( ) puts A's TCB on the mutex's wait queue.
2. lock( ) enables interrupts.
3. Interrupt causes context switch to B, putting A's TCB on the ready queue.
4. unlock( ) moves A's TCB from the mutex wait queue to the ready queue.

Suddenly, there are two copies of A's TCB on the ready queue.

```
lock( )
   {
   disable interrupts;
   if ( status == FREE )
      status = BUSY;
   else
      {
      add thread to queue of
         threads waiting for lock;
      enable interrupts;
      switch to next ready thread;
      }
   enable interrupts;
   }
```

Correct pattern

```
lock( )
   {
   disable interrupts;
   if ( status == FREE )
      status = BUSY;
   else
      {
      add thread to queue of
         threads waiting for lock;
      switch to next ready thread;
      }
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   status = FREE;
   if ( any thread is waiting
         for this lock )
      {
      move waiting thread to
         ready queue;
      status = BUSY;
      }
   enable interrupts;
   }
```

When we switch context, interrupts must be disabled. So, they're still disabled when we're switched back and we must immediately re-enable them. (We can only swap back to somewhere that swapped context.)

Interrupt enable/disable pattern

Adding thread to lock wait queue + switching must be atomic.

Swapcontext invariants for uniprocessors:

1. Thread must leave interrupts disabled when calling swapcontext.
2. All threads assume interrupts are disabled when returning from swapcontext.
3. Must re-enable interrupts before returning to user code.

Enabling/disabling interrupts

Adding thread to lock wait queue + switching must be **atomic**

Thread must leave interrupts disabled when calling switch

Inside lock(), what causes thread to return from switch?

What can lock() assume about the state of interrupts when switch returns?

Switch invariant

- All threads promise to disable interrupts before switching context
- All threads assume interrupts are disabled when returning from switch
- Re-enable interrupts before returning to user-level code

Thread A	Thread B
	lock( )
	yield( )
	{
	disable interrupts
back from swapcontext	swapcontext
enable interrupts	
}	
<user code runs>	
lock( )	
{	
disable interrupts	
swapcontext	back from swapcontext
	enable interrupts
	}
	<user code runs>
	unlock( )
	{
	move thread A to ready queue;
	}
	yield( )
	{
	disable interrupts;
back from swapcontext	swapcontext
enable interrupts	
}	

Locks on multiprocessors

On uniprocessor, disabling interrupts prevents current thread from being switched out.

But this doesn’t work on a multiprocessor:

1. Other processors are still running threads.
2. Not acceptable to stop all other CPUs from executing.

Solution is an atomic *TestAndSet* in a *spin lock*.

Atomic Read-Modify-Write: Test-And-Set

Semantics of test-and-set are to atomically write 1 to a memory location and return the old value.

In Project 2, use *exchange* in `std::atomic`

```
TestAndSet( X )
   {
   old = X;
   X = 1;
   return old;
   }
```

Atomic

TestAndSet usage

If you are able to change the status from 0 to 1, it means you successfully took the lock.

TestAndSet is atomic, so only one thread will see transition from 0 to 1.

```
// lock is initially free.

int status = 0;

SpinLock( )
   {
   while ( TestAndSet( status ) )
      ;
   }

ReleaseSpinLock( )
   {
   status = 0;
   }
```

Lock implementation #3

```
int guard = 0;

lock( )
   {
   disable interrupts;
   while ( TestAndSet( guard ) )
      ;

   if (status == FREE)
      status = BUSY;
   else
      {
      add thread to queue of threads
         waiting for lock;
      switch to next ready thread;
      }
   guard = 0;
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   while ( TestAndSet( guard ) )
      ;

   status = FREE;
   if ( any thread is waiting for
         this lock )
      {
      move waiting thread to ready
         queue;
      status = BUSY;
      }

   guard = 0;
   enable interrupts;
   }
```

Would this work?

```
int guard = 0;

lock( )
   {
   // disable interrupts;
   while ( TestAndSet( guard ) )
      ;

   if (status == FREE)
      status = BUSY;
   else
      {
      add thread to queue of threads
         waiting for lock;
      switch to next ready thread;
      }
   guard = 0;
   // enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   while ( TestAndSet( guard ) )
      ;

   status = FREE;
   if ( any thread is waiting for
         this lock )
      {
      move waiting thread to ready
         queue;
      status = BUSY;
      }

   guard = 0;
   enable interrupts;
   }
```

No, we could get a timer interrupt holding the guard that might want to move us to the ready queue but it wouldn't be able to acquire the guard to do that.

Would this work?

```
int guard = 0;

lock( )
   {
   while ( TestAndSet( guard ) )
      ;

   disable interrupts;
   if (status == FREE)
      status = BUSY;
   else
      {
      add thread to queue of threads
         waiting for lock;
      switch to next ready thread;
      }
   guard = 0;
   enable interrupts;
   }
```

```
unlock( )
   {
   disable interrupts;
   while ( TestAndSet( guard ) )
      ;

   status = FREE;
   if ( any thread is waiting for
         this lock )
      {
      move waiting thread to ready
         queue;
      status = BUSY;
      }

   guard = 0;
   enable interrupts;
   }
```

No, we could be switched out, holding the guard, locking out the other processors in this spin lock.

Summary of lock solution

High-level idea:

Atomically add thread to a waiting list and go to sleep.

How did we achieve this?

1. Disable interrupts and TestAndSet(guard) to protect the critical section.
2. Switch to another thread and hand off the task of enabling interrupts and resetting the guard.

What if no other thread to run?

Atomically suspend CPU with interrupts enabled.