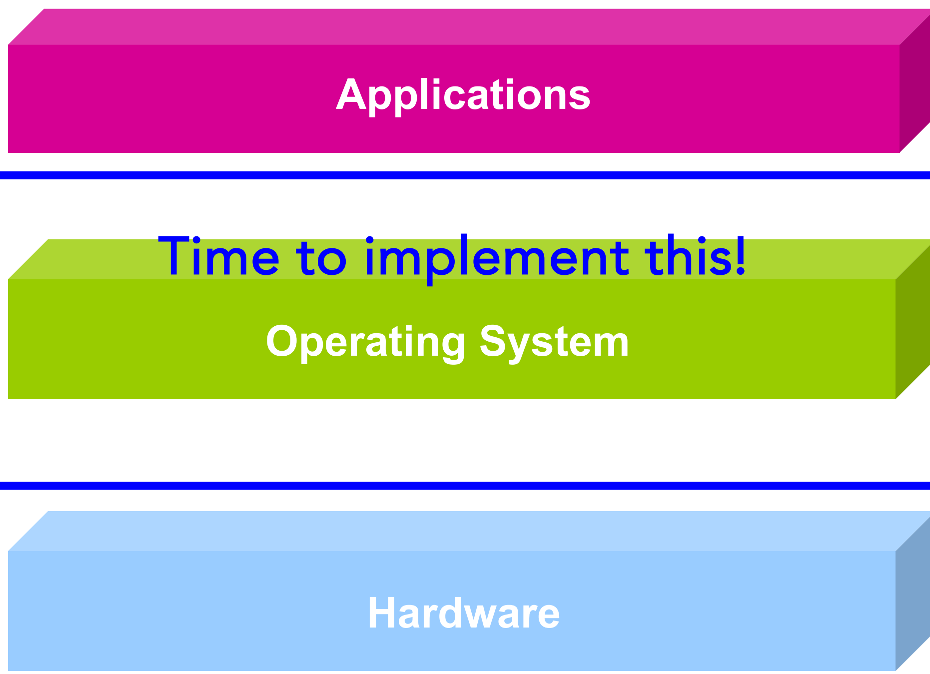
Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 8: Lock implementation

Prof. Ryan Huang

High-level synchronization



Concurrent programs

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

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

Lock implementation #0

```
lock() {
   while (status != FREE) {
   }
   status = BUSY
}
```

```
unlock() {
   status = FREE
}
```

Problems?

Implementing high-level sync operations

Need to use shared data (e.g., status for lock)

- The code that implements these operations must be thread safe

Use synchronization (mutual exclusion, ordering) to implement synchronization ?!

- Can't use the normal high-level synchronization operations
- Instead, use atomic operations provided by hardware, e.g., atomic load, atomic store, etc.

Writing OS code

Disadvantage: OS code can’t use high-level synchronization operations, since it is implementing these

Advantage: OS trusts its own code (OS doesn’t trust user code)

How to provide atomicity for OS code?

Remember: can't use mutex or semaphores

What breaks atomicity for a section of code?

- ~~Code may call yield, etc.~~ Don't do this ☺
- ~~An interrupt may occur~~ Disable interrupts around critical sections
- ~~Another processor may execute instructions~~ Handled later

How to fix each of these?

Atomicity on uniprocessor

Prevent events that allow other threads to run

- Don't call yield() in a critical section
- Disable interrupts around critical section

Example

```
disable interrupts
if (no milk) {
  buy milk
}
enable interrupts
```

Problems?

- Unsafe to run user code with interrupts disabled
  - Why?
- How to provide multiple locks?

Who may disable interrupts?

User code may *not* run with interrupts disabled

- OS doesn't trust user code (to re-enable interrupts)

But OS code may!

Disable interrupts to provide atomicity for critical sections of OS code

Disabling and enabling interrupts serves as a low-level lock for the OS

Lock implementation #1 (uniprocessor, busy waiting)

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

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

Busy waiting

Problem with lock implementation #1

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

Solution: integrate lock implementation with thread switching

- `lock()` gives up CPU, so other threads can run
- `unlock()` wakes up waiting thread when lock is free

Avoid busy waiting by switching threads

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

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

- What does it mean for lock() to "add thread to queue of threads waiting for lock"?
- Why have a separate waiting queue for the lock? Why not put waiting thread onto ready queue?

When to re-enable interrupts?

```
lock() {
    disable interrupts
    while (status != FREE) {
        enable interrupts
        add thread to queue of
            threads waiting for lock

        switch to next ready thread
        disable interrupts
    }
    status = BUSY
    enable interrupts
}
```

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

lock waiting queue

ready queue

CPU

thread U

interrupted (force to yield)

When to re-enable interrupts?

Problem: adding thread to waiting queue and going to sleep must be atomic

When have we seen this problem before?

How did we solve it before?

Interrupt enable/disable pattern

Adding thread to lock wait queue + going to sleep must be atomic

Thread must leave interrupts disabled when calling switch

Leave interrupts disabled when calling switch

```
lock() {
    disable interrupts
    while (status != FREE) {

        add thread to queue of
            threads waiting for lock
        enable interrupts
        switch to next ready thread
        disable interrupts
    }
    status = BUSY
    enable interrupts
}
```

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

- Switch without re-enabling interrupts ?!

```
lock          disable interrupts
do stuff      do stuff
unlock        enable interrupts
```

Lock implementation #2 (uniprocessor, no busy waiting)

```
lock() {
    disable interrupts
    while (status != FREE) {

        add thread to queue of
            threads waiting for lock

        switch to next ready thread

    }
    status = BUSY
    enable interrupts
}
```

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

- What can lock() assume about the state of interrupts after switch returns?
- How does lock() wake up from switch?

Switch invariant (for uniprocessors)

All threads promise to have interrupts disabled when calling switch

All threads can assume that interrupts are "still" disabled when switch returns

Thread A

```
        enable interrupts
}
<user code runs>
lock() {
        disable interrupts
        ...
        swapcontext
```

```
        back from swapcontext
        enable interrupts
}
```

Thread B

```
yield() {
        disable interrupts
        swapcontext
```

```
        back from swapcontext
        enable interrupts
}

<user code runs>

unlock() (move thread A to
ready queue)

yield() {
        disable interrupts
        swapcontext
```

Lock implementation #3 (uniprocessor, no busy waiting, handoff)

```
lock() {
   disable interrupts
 if ~~while~~ (status != FREE) {

      add thread to queue of
         threads waiting for lock

      switch to next ready thread
   } else {
    status = BUSY
    }
   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 to provide atomicity for OS code?

Remember: can't use mutex or semaphores

What breaks atomicity for a section of code?

- ~~Code may call yield, etc.~~ Don't do this ☺

- ~~An interrupt may occur~~ Disable interrupts around critical sections. Use switch to "handoff" responsibility for interrupts to next thread.

- Another processor may execute instructions

Atomicity on multiprocessor

Disabling interrupts prevents current thread from being switched out on that processor, but other threads may be running on other processors

Could use atomic load and atomic store

- Example: “Too much milk” solution #3

Modern processors make it easier with instructions that atomically {read + write} memory

test_and_set

Atomically write 1 to a memory location and return old value

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

atomic

In Project 2, std::atomic provides similar operations

Lock implementation #4 (multiprocessor, busy waiting)

```
lock() {
    while (test_and_set(status) == 1){}
}
```

atomic

```
unlock() {
    status = 0
}
```

atomic

status=0: lock is free

status=1: lock is busy

Does this work? Why?

This is called a spin lock

- Uses busy waiting

Busy waiting

Some busy waiting is unavoidable on multiprocessors

With lock implementation #4, how long could lock() busy wait?

```
lock()
if (no milk) {
  buy milk
}
unlock()
```

How to minimize busy waiting?

Lock implementation #5
(multiprocessor, minimal busy waiting)

```
lock() {
    disable interrupts
    while (test_and_set(guard)) {}
    if (status != FREE) {

        add thread to queue of
            threads waiting for lock

        switch to next ready thread
    } else {
        status = BUSY
    }
    guard = 0
    enable interrupts
}
```

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


    guard = 0
    enable interrupts
}
```

How long could lock() busy wait?

Switch invariant

Multiprocessor

~~Uniprocessor~~ switch invariant

- When calling switch: interrupts must be disabled and guard must be 1
- When switch returns: thread may assume interrupts are "still" disabled and guard is "still" 1
- Remember: before running user code, interrupts must be enabled and guard must be set to 0

Summary of lock solution

Use low-level mechanism to provide mutual exclusion for OS code

- Disable interrupts
- Spin lock (for multiprocessors)

OS provides higher-level mechanism (locks) to provide mutual exclusion for user-level code

- User code runs with interrupts enabled
- User code runs with no spin locks held

Summary of lock solution

Hard problem: how to atomically add thread to a waiting list and sleep

How did we achieve this?

- If there is another thread to run: don't sleep! Instead, switch to other thread (obeying switch invariant)

But what if there is no other thread to run?

- Need hardware support to atomically enable interrupts and suspend CPU
- `HLT` (x86), `WFE` (ARM), `interrupt_enable_suspend` (Project 2)

Project 2

Write and test the project incrementally

- 1 CPU, no interrupts
- cpu and thread class
- mutex, cv, join
- Add support for interrupts
- Add support for multiple CPUs

Project 2

Use lots of assertions

- Assert any property that you expect to be true
- Helps catch errors closer to where they occur

Example: switch to null thread context. Would you rather:

- Stop when you switch to the bad thread?
- Stop when you put the bad thread onto the ready queue?
- Stop when you put the bad thread onto a waiting queue?

Example: At any time, a thread context must only be in one "place"

- Where can a thread be?