Slides on threads
borrowed by Chase

Landon Cox
Thread states

- **Blocked**
  - Thread is scheduled
  - Thread calls Lock or wait (or makes I/O request)
  - Another thread calls unlock or signal (or I/O completes)

- **Ready**
  - Thread is scheduled
  - Thread is Pre-empted (or yields)

- **Running**
  - Thread is scheduled
  - Thread calls Lock or wait (or makes I/O request)
Project 1 (test cases)

- **Remember to work on your thread library test suite too**
- **For each of your test cases**
  - No inputs (e.g., no input files, no command-line args)
  - All output to stdout (i.e., use cout)
  - Do not call start_preemption()
  - Diff output of test case w/ your library vs. w/ thread.o
- **Writing test cases**
  - Read through spec
    - Write test cases to stress each required part
    - E.g., lock() blocks if lock is already held
    - Pre-emption is off, so use yield() to create the right interleavings
  - Read through your code
    - Write test cases to exercise all lines of code
    - E.g. each clause of an if/else statement
- **Micro-tests are better for debugging than macro-tests**
Project 1 (garbage collection)

• Garbage collecting threads

```cpp
// simple network server
while (1) {
    int s = socket.accept ();
    thread_create (handle_request, s);
}
```

• Do not want to run out of memory

• What needs to be (C++) “deleted”?
  • Any state associated with the thread (e.g., stack, TCB)
Project 1 (garbage collection)

- Two key questions:
  - When can a stack be deleted and by whom?

- When can a stack be deleted?
  - Only after the thread has finished its “work”
  - Work = function passed to `thread_create`, `thread_libinit`

- Who definitely cannot delete a thread’s stack?
  - The thread itself!
  - Try deleting the stack you are running on …

- So which thread should delete the stack?
Project 1 (garbage collection)

- Hint: don’t use uc_link
  - Only use set/swapcontext to jump to threads
- Anyone want to guess why?
- What can you say about state of interrupts?
  - Interrupts are enabled inside “work” function
  - After it exits, interrupts must be disabled
  - Tricky to guarantee this using uc_link
  - Can get an interrupt while switching to uc_link
Project 1 (garbage collection)

• What makes `swapcontext` simpler?
  • `uc_link` loads threads outside of your lib
  • Calls to `swapcontext` are explicit
  • “Keep everything in front of you”
Using interrupt disable-enable

- Disable-enable on a uni-processor
  - Assume atomic (can use atomic load/store)
- How do threads get switched out (2 ways)?
  - Internal events (yield, I/O request)
  - External events (interrupts, e.g., timers)
- Easy to prevent internal events
- Use disable/enable to prevent external events
Lock implementation #1

- Disable interrupts, busy-waiting

```c
lock () {
    disable interrupts
    while (value != FREE) {
        enable interrupts
        disable interrupts
    }
    value = BUSY
    enable interrupts
}
unlock () {
    disable interrupts
    value = FREE
    enable interrupts
}
```

Why is it ok for lock code to disable interrupts?
It’s in the trusted kernel (we have to trust something).
Lock implementation #1

- Disable interrupts, busy-waiting

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

```c
unlock () {
    disable interrupts
    value = FREE
    enable interrupts
}
```

Do we need to disable interrupts in unlock?
Only if “value = FREE” is multiple instructions (safer)
Lock implementation #1

- Disable interrupts, busy-waiting

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

```c
unlock () {
    disable interrupts
    value = FREE
    enable interrupts
}
```

Why enable-disable in lock loop body?
Otherwise, no one else will run (including unlockers)
Using read-modify-write instructions

- Disabling interrupts
  - Ok for uni-processor, breaks on multi-processor
  - Why?
- Could use atomic load-store to make a lock
  - Inefficient, lots of busy-waiting
- Hardware people to the rescue!
Using read-modify-write instructions

• Most modern processor architectures
  • Provide an atomic read-modify-write instruction
• Atomically
  • Read value from memory into register
  • Write new value to memory
• Implementation details
  • Lock memory location at the memory controller
Lock implementation #3

- Interrupt disable, no busy-waiting

```c
lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        add thread to queue of threads waiting for lock
        switch to next ready thread // don’t add to ready queue
    }
    enable interrupts
}

unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}
```
Lock implementation #3

```c
lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        add thread to queue of threads waiting for lock
        switch to next ready thread // don’t add to ready queue
    }
    enable interrupts
}

unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}
```

This is called a “hand-off” lock.

Who gets the lock after someone calls unlock?
Lock implementation #3

This is called a "hand-off" lock.

Who might get the lock if it weren’t handed-off directly? (i.e., if value weren’t set BUSY in unlock)
Lock implementation #3

```c
lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        add thread to queue of threads waiting for lock
        switch to next ready thread // don’t add to ready queue
    }
    enable interrupts
}

unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}
```

This is called a “hand-off” lock.

What kind of ordering of lock acquisition guarantees does the hand-off lock provide?
Lock implementation #3

```c
lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        add thread to queue of threads waiting for lock
        switch to next ready thread // don’t add to ready queue
    }
    enable interrupts
}
```

```c
unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}
```

This is called a “hand-off” lock.

What does this mean? Are we saving the PC?
Lock implementation #3

This is called a "hand-off" lock.

No, just adding a pointer to the TCB/context.

```c
lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        lockqueue.push(&current_thread->ucontext);
        swapcontext(&current_thread->ucontext,
                    &new_thread->ucontext));
    }
    enable interrupts
}

unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}
```
Lock implementation #3

```c
lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        add thread to queue of threads waiting for lock
        switch to next ready thread // don’t add to ready queue
    }
    enable interrupts
}
```

```c
unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}
```

This is called a “hand-off” lock.

Why a separate queue for the lock?
Lock implementation #3

This is called a “hand-off” lock.

Project 1 note: you must guarantee FIFO ordering of lock operations.
Lock implementation #3

```c
lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        enable interrupts
        add thread to queue of threads waiting for lock
        switch to next ready thread // don’t add to ready queue
    }
    enable interrupts
}

unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}
```

When and how could this fail?
Lock implementation #3

```c
lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        add thread to queue of threads
        enable interrupts
        switch to next ready thread // don’t add to ready queue
    }
    enable interrupts
}

unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}
```

When and how could this fail?
Lock implementation #3

lock () {
    disable interrupts
    if (value == FREE) {
        value = BUSY // lock acquire
    } else {
        add thread to queue of threads waiting for lock
        switch to next ready thread // don’t add to ready queue
    }
    enable interrupts
}

unlock () {
    disable interrupts
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    enable interrupts
}

Putting lock thread on lock wait queue, switch must be atomic. Must call switch with interrupts off.
How is switch returned to?

• Think of switch as three phases
  1. Save current location (SP, PC)
  2. Point processor to another stack (SP’)
  3. Jump to another instruction (PC’)

• Only way to get back to a switch
  • Have another thread call switch
What is lock() assuming about the state of interrupts after switch returns?
Interrupts and returning to switch

• *Lock() can assume that switch*
  • Is always called with interrupts disabled

• **On return from switch**
  • Previous thread must have disabled interrupts

• **Next thread to run**
  • Becomes responsible for re-enabling interrupts

• **Invariants**: threads promise to
  • Disable interrupts before switch is called
  • Re-enable interrupts after returning from switch
Thread A

```c
yield () {
  disable interrupts
  ...
  switch (B->A)
  enable interrupts
} // exit thread library

lock () {
  disable interrupts
  ...
  switch (A->B)
  back from switch (B->A)
} // exit lock
```

Thread B

```c
yield () {
  disable interrupts
  ...
  switch (B->A)
} // exit yield

unlock () // moves A to ready queue

yield () {
  disable interrupts
  ...
  switch (B->A)
  back from switch (B->A)
  ...
  enable interrupts
} // exit yield
```

B holds lock
Lock implementation #4

• Test&set, minimal busy-waiting

```c
lock () {
    while (test&set (guard)) {} // like interrupt disable
    if (value == FREE) {
        value = BUSY
    } else {
        put on queue of threads waiting for lock
        switch to another thread // don't add to ready queue
    }
    guard = 0  // like interrupt enable
}

unlock () {
    while (test&set (guard)) {} // like interrupt disable
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    guard = 0  // like interrupt enable
}
```
Lock implementation #4

Why is this better than only spinning with test&set?

lock () {
    while (test&set (guard)) {} // like interrupt disable
    if (value == FREE) {
        value = BUSY
    } else {
        put on queue of threads waiting for lock
        switch to another thread // don’t add to ready queue
    }
    guard = 0  // like interrupt enable
}

unlock () {
    while (test&set (guard)) {} // like interrupt disable
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    guard = 0  // like interrupt enable
}
Lock implementation #4

What is the switch invariant?
Threads promise to call switch with guard set to 1.

```c
lock () {
    while (test&set (guard)) {} // like interrupt disable
    if (value == FREE) {
        value = BUSY
    } else {
        put on queue of threads waiting for lock
        switch to another thread // don’t add to ready queue
    }
    guard = 0  // like interrupt enable
}

unlock () {
    while (test&set (guard)) {} // like interrupt disable
    value = FREE
    if anyone on queue of threads waiting for lock {
        take waiting thread off queue, put on ready queue
        value = BUSY
    }
    guard = 0  // like interrupt enable
}
```
```c
void C () {
    A (0);
}

void B () {
    C ();
}

void A (int tmp){
    if (tmp) B ();
}

int main () {
    A (1);
    return 0;
}
```
Creating a new thread

- Also called “forking” a thread
- Idea: create initial state, put on ready queue

1. Allocate, initialize a new TCB
2. Allocate a new stack
3. Make it look like thread was going to call a function
   - PC points to first instruction in function
   - SP points to new stack
   - Stack contains arguments passed to function
   - Project 1: use makecontext
4. Add thread to ready queue
Example: thread-safe queue

What can go wrong?

1) 2 elements in queue, ptr->next is non-null, switch to other thread

2) previous head->next set to null

3) ptr->next now null, set ptr to null

4) dereference null pntr
Thread-safe queue

- Can enqueue unlock anywhere?
  - No
- Must leave shared data
  - In a consistent/sane state
- Data invariant
  - “consistent/sane state”
  - “always” true

enqueue () {
  lock (qLock)
  // ptr is private
  // head is shared
  new_element = new node();
  if (head == NULL) {
    head = new_element;
  } else {
    node *ptr;
    // find queue tail
    for (ptr=head;
         ptr->next!=NULL;
         ptr=ptr->next){}
    ptr->next=new_element;
  }
  unlock(qLock); // safe?
  new_element->next=0;
}

Another thread can call enqueue
I’m always holding a lock while accessing shared state.

ptr may not point to tail after lock/unlock.

Lesson:
• Thinking about individual accesses is not enough
• Must reason about dependencies between accesses