EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha

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, we need to construct its TCB as if it had been running and got paused

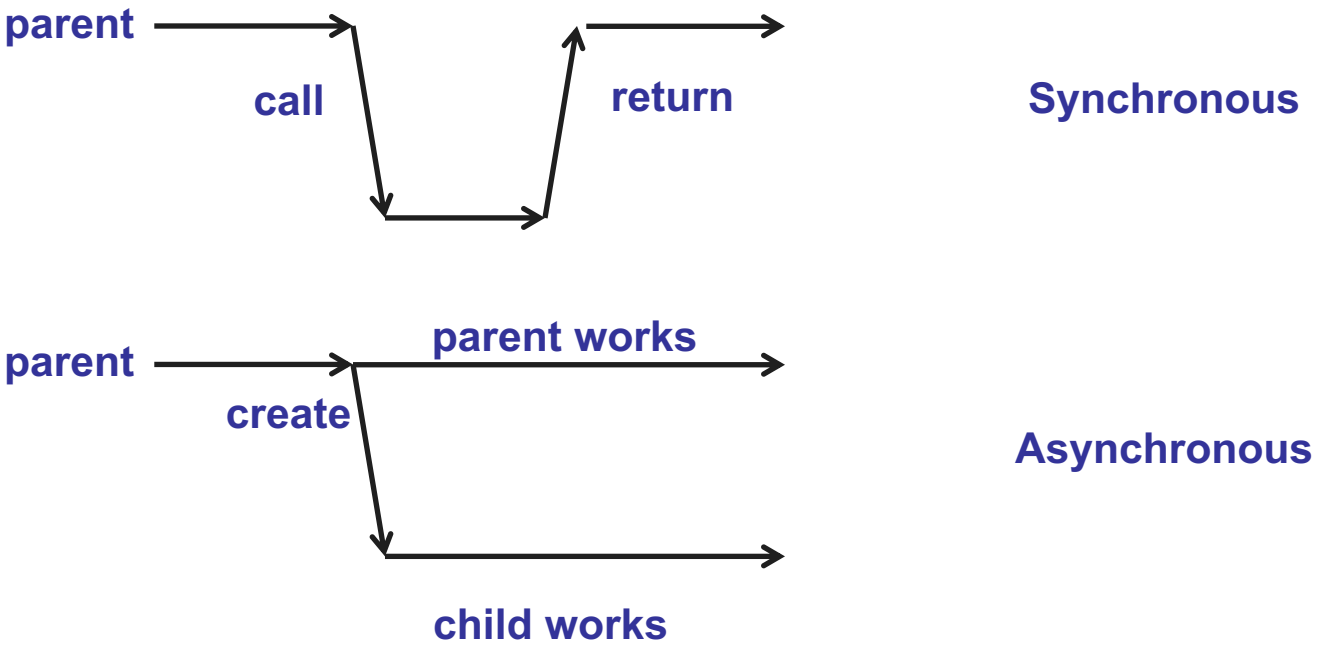
Creating a new thread

- Steps
  - Allocate and initialize new thread control block
  - Allocate and initialize new stack
    - » In Project 2, this is done via *getcontext()*/*makecontext()*
  - Add thread control block to ready queue

- Note: Unix fork() is related but different
  - fork() creates a new process (new thread + new address space)

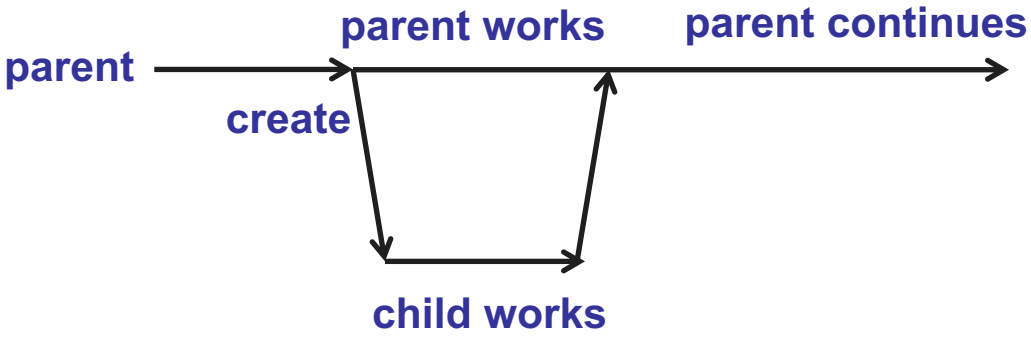
How to use new thread

- Creating a thread is like an asynchronous procedure call



Synchronizing with child

- What if parent wants to work for a while, then wait for child to finish?



Synchronizing with child

```
parent()
        create child thread
        print "parent works"
        …
        print "parent continues"
        …

child()
        …
        print "child is done"
```

When would this work?

Desired output

parent works
child is done
parent continues

OR

child is done
parent works
parent continues

Synchronizing with child

```
parent()
        create child thread
        print "parent works"
        …
        yield()
        print "parent continues"
        …

child()
        …
        print "child is done"
```

Desired output

parent works
child is done
parent continues

OR

child is done
parent works
parent continues

Does this work?

Synchronizing with join

```
parent()
        create child thread
        print "parent works"
        …
        childThread.join()
        print "parent continues"
        …

child()
        …
        print "child is done"
```

Desired output

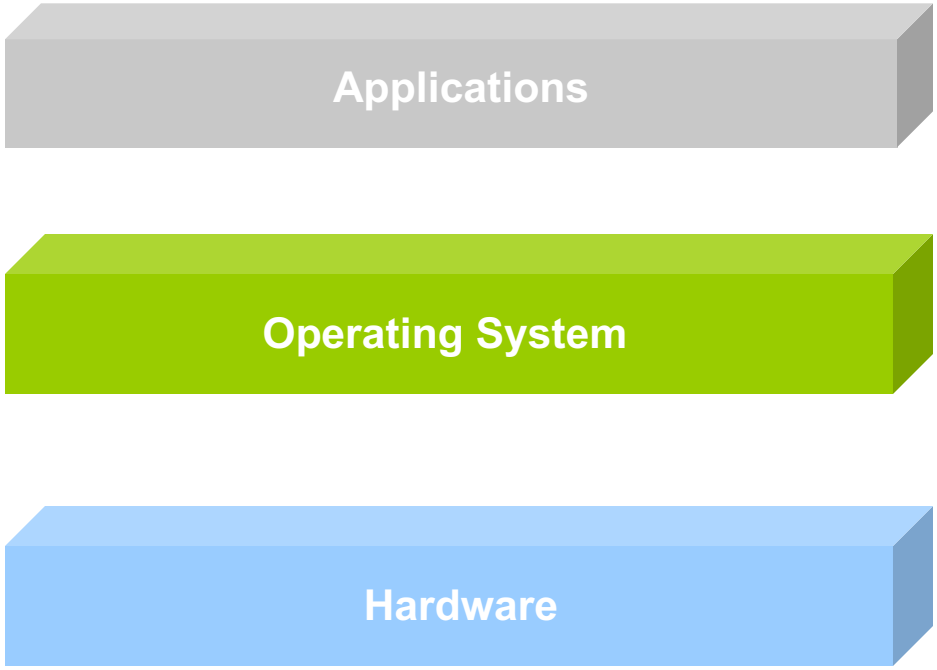
parent works
child is done
parent continues

OR

child is done
parent works
parent continues

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 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?

Lock implementation #1

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

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

Need for other atomic primitives

- On uniprocessor, disabling interrupts prevents current thread from being switched out
- But this doesn't work on a multiprocessor
  - Other processors are still running threads
  - Not acceptable to stop all other CPUs from executing
- Could use atomic load/store
  - Example: "Too much milk" solution #3

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

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

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

- In Project 2, *exchange* in std::atomic

Lock implementation #2

```
// status=0 means lock is free
lock() {
      while (test_and_set(status) == 1) {
      }
}

unlock() {
      status = 0
}
```

- test_and_set is atomic, so only one thread will see transition from 0 to 1

Busy waiting

- Problem with lock implementations #1 and #2
  - 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 #3

```
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 #3

```
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 #3

```
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 #3

```
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 #3

```
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
}
```

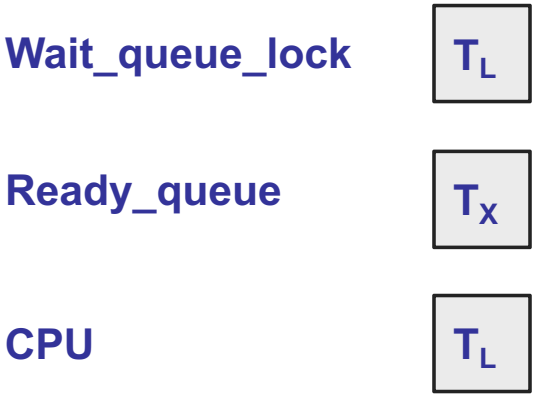
Lock implementation #3

Wait_queue_lock T_L

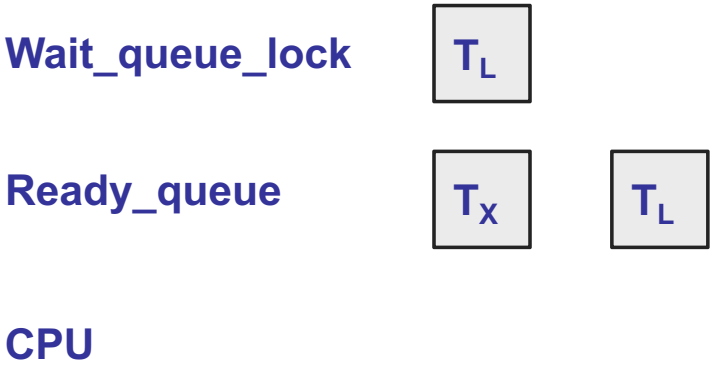
Ready_queue

CPU T_L

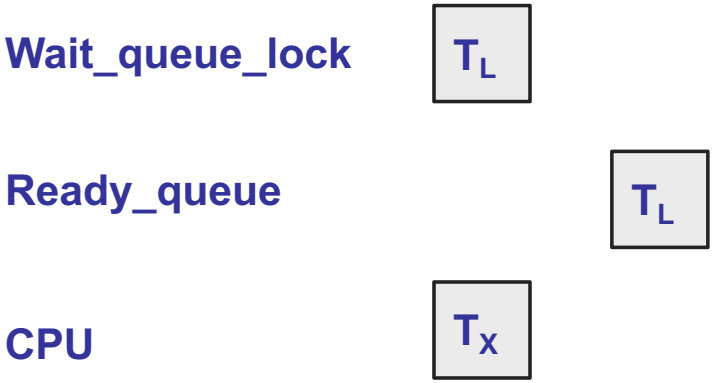
Lock implementation #3



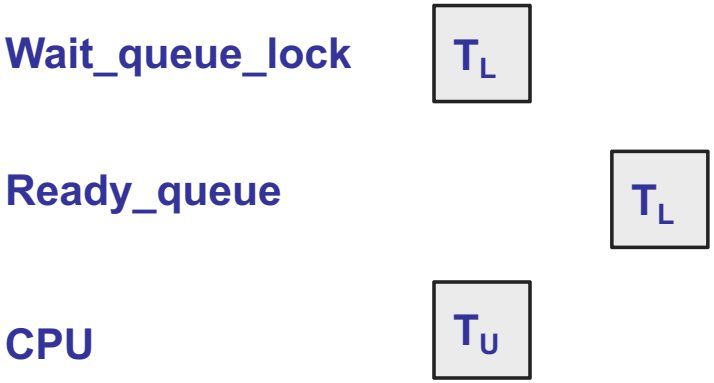
Lock implementation #3



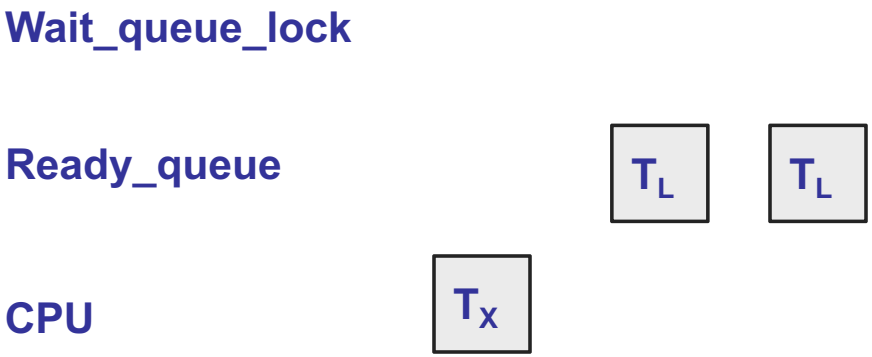
Lock implementation #3



Lock implementation #3



Lock implementation #3



Lock implementation #3

```
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
}
```

Interrupt enable/disable pattern

- Adding thread to lock wait queue + switching must be **atomic**
- Thread must leave interrupts disabled when calling switch

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

- How does lock() wake up from switch?

- Switch invariant
  - All threads promise to have interrupts disabled when calling switch
  - All threads assume interrupts are disabled when returning from switch