ECE 570/670

David Irwin

Lecture 8

Administrative Details

- ECE670 projects
  - 1) Form project teams – respond to Google Form
  - 2) Schedule 3/10 appointment – submit on Google Calendar
    - Have some idea of what you want to do to discuss
  - 3) Proposal due 3/24 – requirements listed on website

- Project Options
  - 1) Stock web server project
  - 2) Other OS project idea
  - **Requirements:** *software implementation component + performance evaluation component*

Administrative Details

- New TA
  - Xiaoding (Rebecca) Guan
  - Office hours – see website
    - Wednesdays 12:30pm – 2:30pm
    - Marcus 220

Administrative Details

- Assignment 1 is out: two parts
  - Project 1 disk scheduler (1d)
    - How to use concurrency primitives
  - Project 1 thread library (1t)
    - How concurrency primitives actually work by actually building them

- Today's paper: Erasure

Eraser: A Dynamic Data Detector for Multi-Threaded Programs

Stevan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson

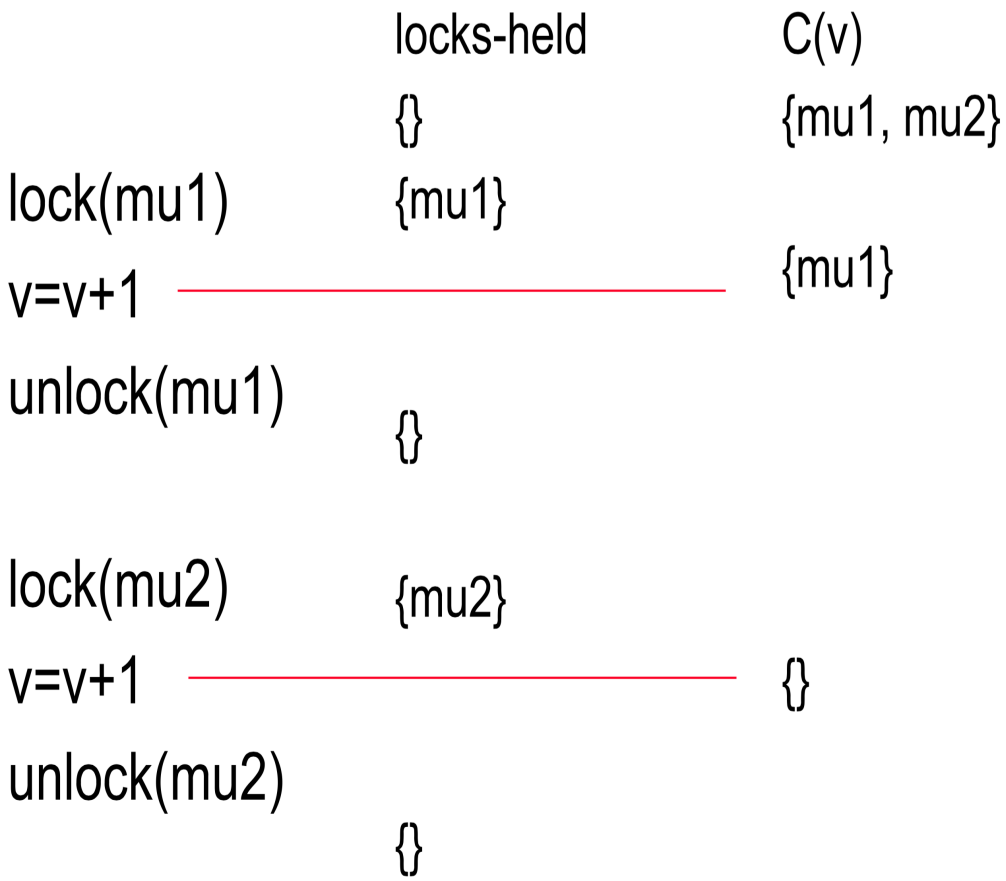
- Dynamically detect data races in lock-based programs
- Led to significant research on detecting concurrency bugs

Defines Lockset Algorithm

- Locking discipline
  - Every shared variable is protected by some locks
- Infer protection relation
  - Infer which locks protect which variable from execution history.

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\ \}$, then issue a warning.

Lockset Algorithm Example

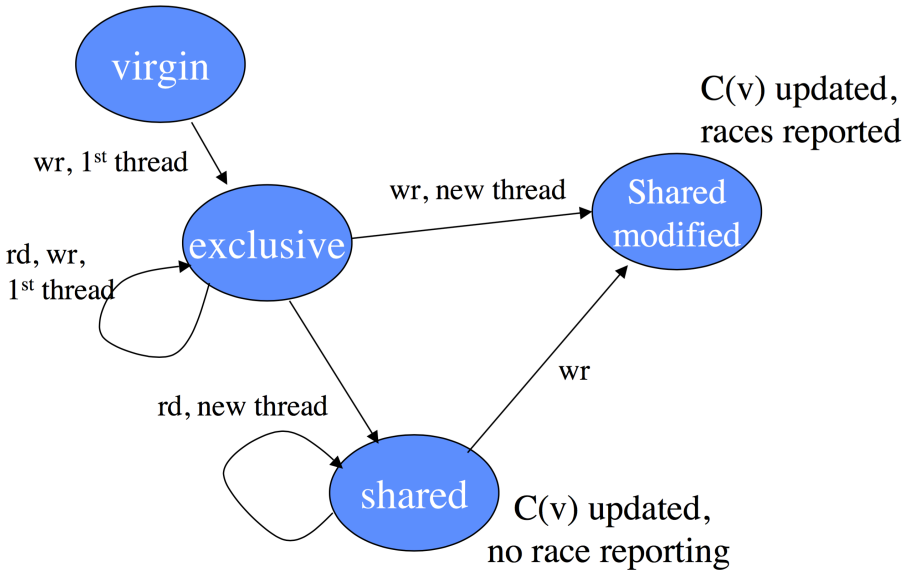


Limitations

- Initialization
  - shared variables are frequently initialized without holding a lock
- Read-Shared Data
  - Some shared variables are written during initialization only and are read-only thereafter
- Read-Write Locks
  - Read-write locks allow multiple readers to access a shared variable, but allow only a single writer to do so

Refining the Algorithm

- Initialization
  - Don't start until see a second thread
- Read-shared data
  - Report only after it becomes write shared



Refining the Algorithm

- Reader-writer locking

  Change algorithm to reflect lock type

Let *locks_held*(t) be the set of locks held in any mode by thread t.
Let *write_locks_held*(t) be the set of locks held in write mode by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each read of v by thread t,
  set $C(v) := C(v) \cap$ *locks_held*(t);
  if $C(v) := \{\ \}$, then issue a warning.
On each write of v by thread t,
  set $C(v) := C(v) \cap$ *write_locks_held*(t);
  if $C(v) = \{\ \}$, then issue a warning.

Implementation

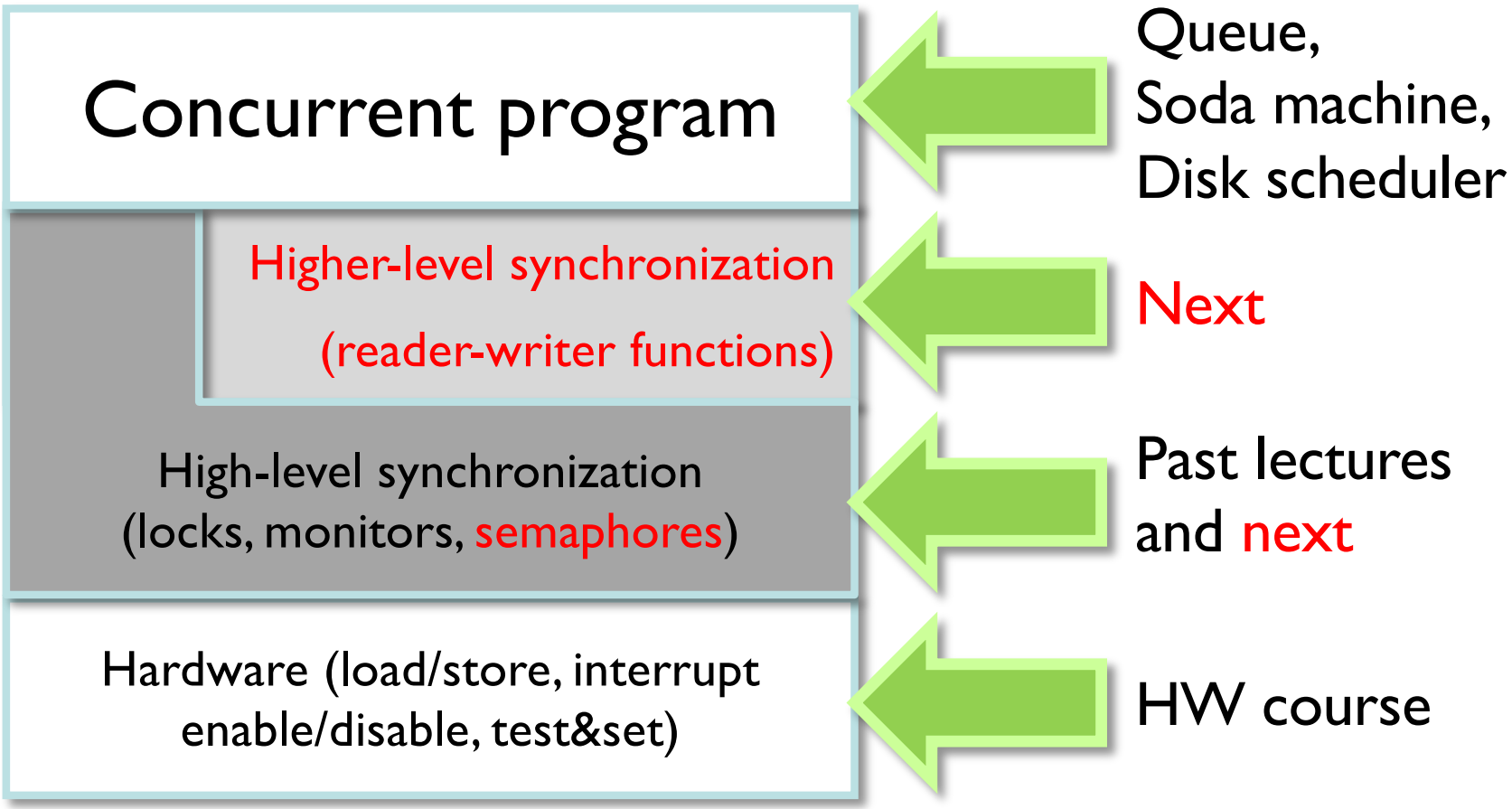
- Binary rewriting used
  - Add instrumentation to call Eraser runtime
  - Calls to storage allocator initializes C(v)
  - Each Acquire and Release call updates locks-held(t)
  - Each load and store updates C(v)

- Lock ids stored in a lookup table

Performance

- Reported on experience detecting a number of data races
- Slowdown by factor of 10 to 30x
  - Overhead of making a procedure call at every load and store instruction
  - Overhead may change thread order
  - Also affects the behavior of time-sensitive applications

Layers of synchronization



Prioritizing waiting writers

```
writerStart () {
   lock (RWLock)
   while (actWriters > 0 ||
          numReaders > 0) {
     waitWriters++
     wait (RWlock, rowfCV)
     waitWriters--
   }
   actWriters++;
   unlock (RWLock)
}
```

```
readerStart () {
   lock (RWLock)
   while ((actWriters +
           waitWriters) > 0){
     wait (RWlock, rowfCV)
   }
   numReaders++
   unlock (RWLock)
}
```

```
writerFinish () {
   lock (RWLock)
   actWriters--
   broadcast (rowfCV)
   unlock (RWLock)
}
```

```
readerFinish () {
   lock (RWLock)
   numReaders--
   broadcast (rowfCV)
   unlock (RWLock)
}
```

When to use broadcast in RW locks

- When more than one thread might need to run
  - If writer leaves, **all** waiting readers should be woken
  - But if reader leaves, who should be woken?
    - No one if other readers are present
    - A waiting writer, as long as there aren't more readers
      - Readers don't block because of other readers
- Signal (instead of broadcast) could wakeup "wrong" thread
  - "Wrong" thread checks condition, re-sleeps
    - i.e., writer wakes up when there are still readers present
    - i.e., reader wakes up when there are writers present
  - (producer-consumer problem)
- Still have the spurious wakeup problem w/ broadcast though
  - We wake up writers unnecessarily
  - (let's try to fix that...)

Removing spurious wakeups

```
writerStart () {
  lock (RWLock)
  while (actWriters > 0 ||
         numReaders > 0) {
    waitWriters++
    wait (RWlock, rowfCV)
    waitWriters--
  }
  actWriters++;
  unlock (RWLock)
}
```

```
writerFinish () {
  lock (RWLock)
  actWriters--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

```
readerStart () {
  lock (RWLock)
  while ((actWriters +
          waitWriters) > 0){
    wait (RWlock, rowfCV)
  }
  numReaders++
  unlock (RWLock)
}
```

```
readerFinish () {
  lock (RWLock)
  numReaders--
  if (numReaders == 0)
    broadcast (rowfCV)
  unlock (RWLock)
}
```

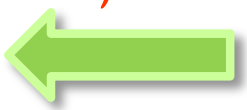
Is it ok to use signal?

```
writerStart () {
  lock (RWLock)
  while (actWriters > 0 ||
         numReaders > 0) {
    waitWriters++
    wait (RWlock, rowfCV)
    waitWriters--
  }
  actWriters++;
  unlock (RWLock)
}
```

```
readerStart () {
  lock (RWLock)
  while ((actWriters +
          waitWriters) > 0){
    wait (RWlock, rowfCV)
  }
  numReaders++
  unlock (RWLock)
}
```

```
writerFinish () {
  lock (RWLock)
  actWriters--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

```
readerFinish () {
  lock (RWLock)
  numReaders--
  if (numReaders == 0)
    signal(rowfCV)
  unlock (RWLock)
}
```



No: might wake up a waiting reader, who needs to continue waiting

Reader-writer interface vs locks

- R-W interface looks a lot like locking
  - *Start ~ lock
  - *Finish ~ unlock
- Standard terminology
  - Four functions called "reader-writer locks"
  - Between readStart/readFinish has "read lock"
  - Between writeStart/writeFinish has "write lock"
- Pros/cons of R-W vs standard locks?
  - Trade-off concurrency for complexity
  - Must know how data is being accessed in critical section

Moving on…

Semaphores

- First defined by Dijkstra in late 60s
- Two operations: up and down

```
// aka "V" ("verhogen")
up () {
  // begin atomic
  value++
  // end atomic
}
```

```
// aka "P" ("proberen")
down () {
  do {
    // begin atomic
    if (value > 0) {
      value--
      break
    }
    // end atomic
  } while (1)
}
```

What is going on here?

Can value ever be < 0?

More semaphores

- Key state of a semaphore is its **value**
  - Initial value determines semaphore's behavior
  - Value cannot be accessed outside semaphore
    - (i.e. there is no semaphore.getValue() call)
- Semaphores can provide both types of synchronization
  - Mutual exclusion (like locks)
  - Ordering constraints (like monitors)

Semaphore mutual exclusion

- Ensure that 1 (or < N) thread(s) is in critical section

```
s.down ();
// critical section
s.up ();
```

- How do we make a semaphore act like a lock?
  - Set initial value to 1 (or N)
  - Like lock/unlock, but more general
  - (could allow 2 threads in critical section if initial value = 2)
  - Lock is equivalent to a binary semaphore (value = 1)
    - lock() = down(); unlock() = up()

Semaphore ordering constraints

- Thread A waits for B before proceeding

```
// Thread A
s.down ();
// continue
```

```
// Thread B
// do task
s.up ();
```

- How to make a semaphore behave like a monitor?
  - Set initial value of semaphore to 0
- A is guaranteed to wait for B to finish
  - Doesn't matter if A or B run first
- Like a CV in which condition is "`sem.value==0`"
  - Can think of this as a "prefab" condition variable

Prod.-cons. with semaphores

- Same before-after constraints
  - If buffer empty, consumer waits for producer
  - If buffer full, producer waits for consumer
- Semaphore assignments
  - mutex (binary semaphore)
  - fullBuffers (counts number of full slots)
  - emptyBuffers (counts number of empty slots)



Prod.-cons. with semaphores

- Initial semaphore values?
  - Mutual exclusion
    - sem mutex (?)
  - Machine is initially empty
    - sem fullBuffers (?)
    - sem emptyBuffers (?)

Prod.-cons. with semaphores

- Initial semaphore values
  - Mutual exclusion
    - sem mutex (1)
  - Machine is initially empty
    - sem fullBuffers (0)
    - sem emptyBuffers (MaxSodas)

Prod.-cons. with semaphores

```
Semaphore mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)
```

```
consumer () {
  down (fullBuffers)

  down (mutex)

  take soda out

  up (mutex)

  up (emptyBuffers)
}
```

```
producer () {
  down (emptyBuffers)

  down (mutex)

  put soda in

  up (mutex)

  up (fullBuffers)
}
```

Use one semaphore for fullBuffers and emptyBuffers?

Remember semaphores are like prefab CVs.

We don't have "broadcast()" any more.

Prod.-cons. with semaphores

```
Semaphore mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)
```

```
consumer () {
  down (mutex)
  down (fullBuffers)
  take soda out
  up (emptyBuffers)
  up (mutex)
}
```

```
producer () {
  down (mutex)
  down (emptyBuffers)
  put soda in
  up (fullBuffers)
  up (mutex)
}
```

Does the order of the down calls matter?

Yes. Can cause "deadlock."

Prod.-cons. with semaphores

```
Semaphore mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)
```

```
consumer () {
  down (fullBuffers)

  down (mutex)

  take soda out

  up (emptyBuffers)

  up (mutex)
}
```

```
producer () {
  down (emptyBuffers)

  down (mutex)

  put soda in

  up (fullBuffers)

  up (mutex)
}
```

Does the order of the up calls matter?

Not for correctness (possible efficiency issues).

Prod.-cons. with semaphores

```
Semaphore mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)
```

```
consumer () {
  down (fullBuffers)

  down (mutex)

  take soda out

  up (mutex)

  up (emptyBuffers)
}
```

```
producer () {
  down (emptyBuffers)

  down (mutex)

  put soda in

  up (mutex)

  up (fullBuffers)
}
```

What about multiple consumers and/or producers?

Doesn't matter; solution stands.

Prod.-cons. with semaphores

```
Semaphore mtx(1),fullBuffers(1),emptyBuffers(MaxSodas-1)
```

```
consumer () {
  down (fullBuffers)

  down (mutex)

  take soda out

  up (mutex)

  up (emptyBuffers)
}
```

```
producer () {
  down (emptyBuffers)

  down (mutex)

  put soda in

  up (mutex)

  up (fullBuffers)
}
```

What if 1 full buffer and multiple consumers call down?

Only one will see semaphore at 1, rest see at 0.

Monitors vs. semaphores

- Monitors
  - Separate mutual exclusion and ordering
- Semaphores
  - Provide both with same mechanism
- Semaphores are more "elegant"
  - Single mechanism
  - Can be harder to program
  - "Remember" past up() calls

Monitors vs. semaphores

```
// Monitors
lock (mutex)

while (condition) {
  wait (CV, mutex)
}

unlock (mutex)
```

```
// Semaphores
down (semaphore)
```

- Where are the conditions in both?
- Why do monitors need a lock, but not semaphores?

Monitors vs. semaphores

```
// Monitors
lock (mutex)

while (condition) {
  wait (CV, mutex)
}

unlock (mutex)
```

```
// Semaphores
down (semaphore)
```

- Where are the conditions in both?
  - "condition" and CV in monitor, sem.value==0 in semaphore
- Why do monitors need a lock, but not semaphores?
  - Semaphores increment/decrement atomically

Monitors vs. semaphores

```
// Monitors
lock (mutex)

while (condition) {
  wait (CV, mutex)
}

unlock (mutex)
```

```
// Semaphores
down (semaphore)
```

- When are semaphores appropriate?
  - When shared integer maps naturally to problem at hand
  - (i.e. when the condition involves a count of **one** thing)

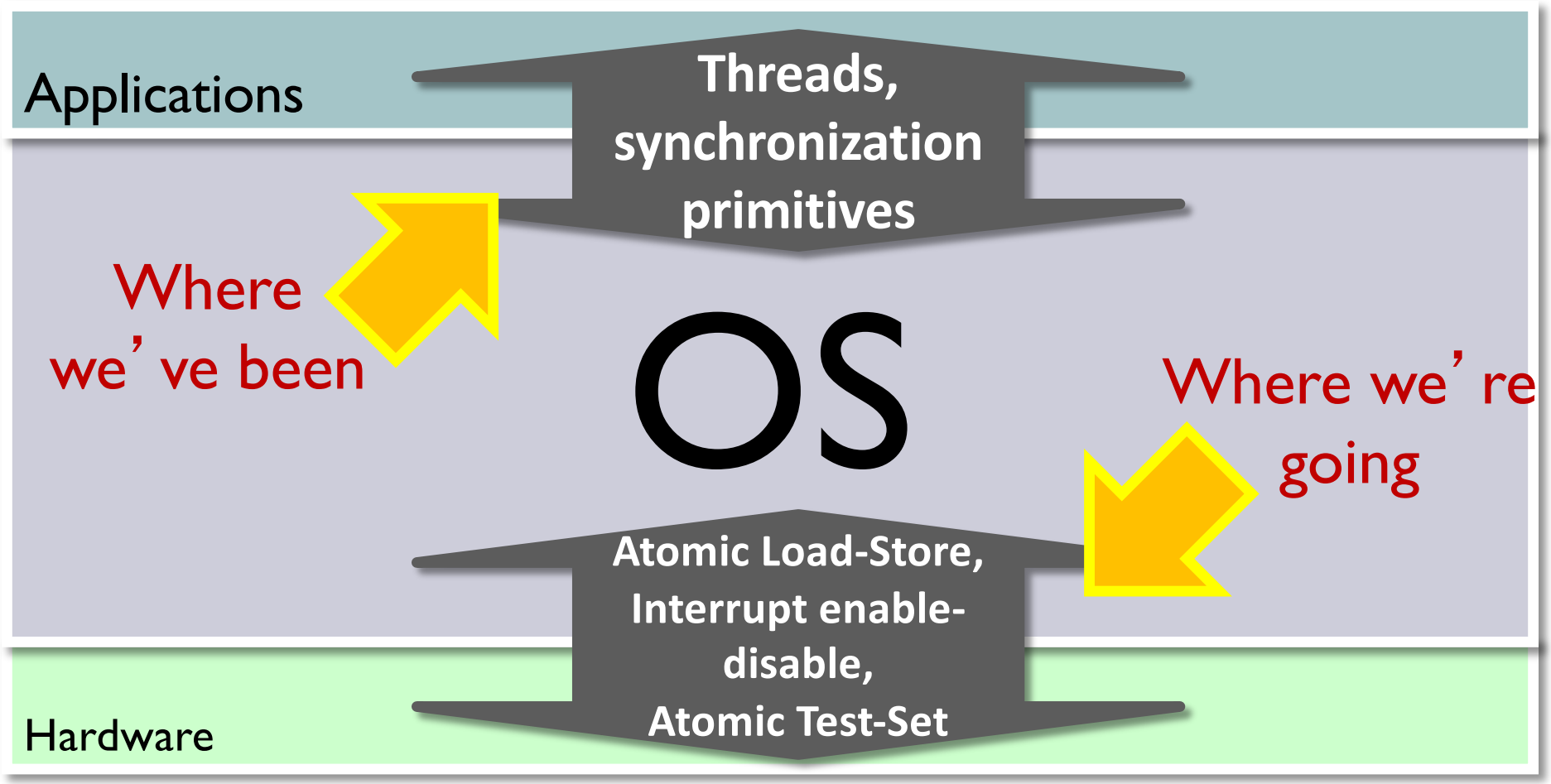
Monitors vs. semaphores

```
// Monitors
lock (mutex)

while (condition) {
  wait (CV, mutex)
}

unlock (mutex)
```

```
// Semaphores
down (semaphore)
```

- Which is more flexible?
  - Monitors; they allow you to define arbitrary conditions
  - Easy to implement semaphores with monitors
    - Harder to implement monitors with semaphores

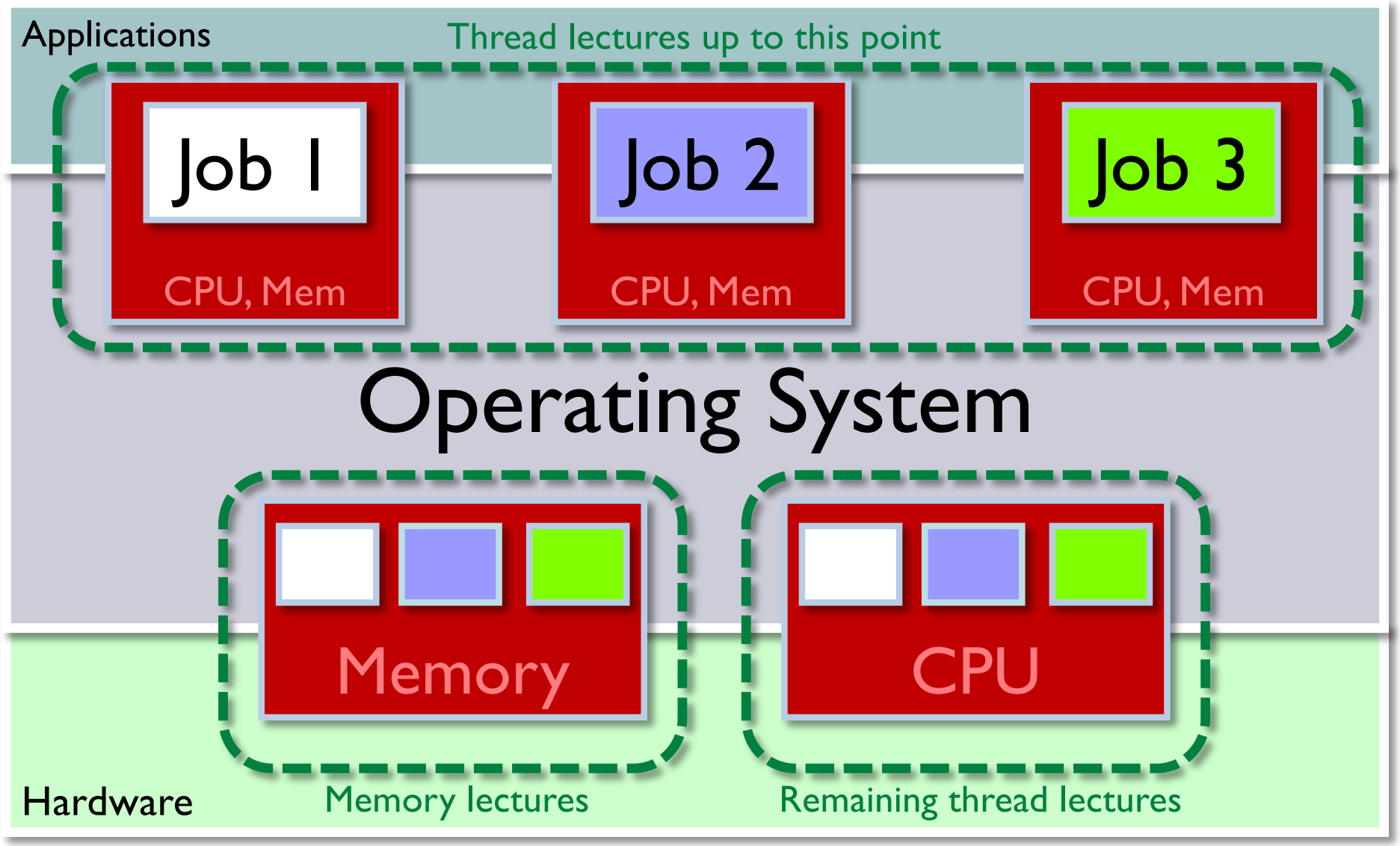
Recap and looking ahead



Recall, thread interactions

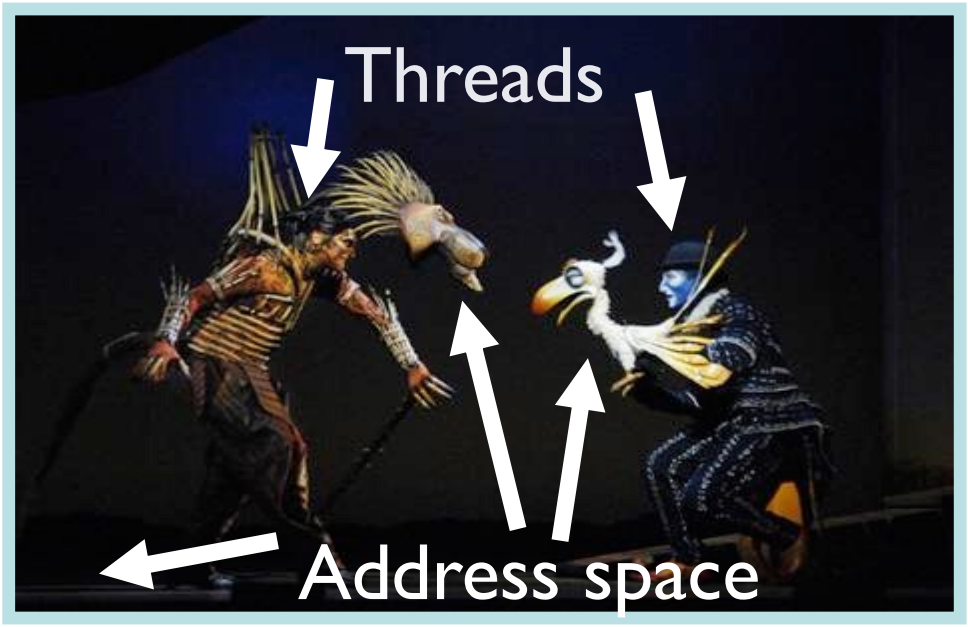
1. Threads can access shared data
   - Use locks, monitors, and semaphores
   - What we've done so far
     - Assumed that threads might be executing simultaneously (on multiple processors)
2. Threads also share hardware (if we don't have a multi-core processor)
   - CPU (uni-processor)
   - Memory

Hardware, OS interfaces



The play analogy

- Process is like a play performance
- Program is like the play's script
- One CPU is like a one-man-show
  - (Actor switches between roles)



Threads that aren't running

- What is a non-running thread?
  - thread="sequence of executing instructions"
  - non-running thread="paused execution"
- How do we restart a "paused" thread?
- Must save thread's private state before pausing
  - To re-run, re-load private state
  - Want thread to start where it left off

Private vs global thread state

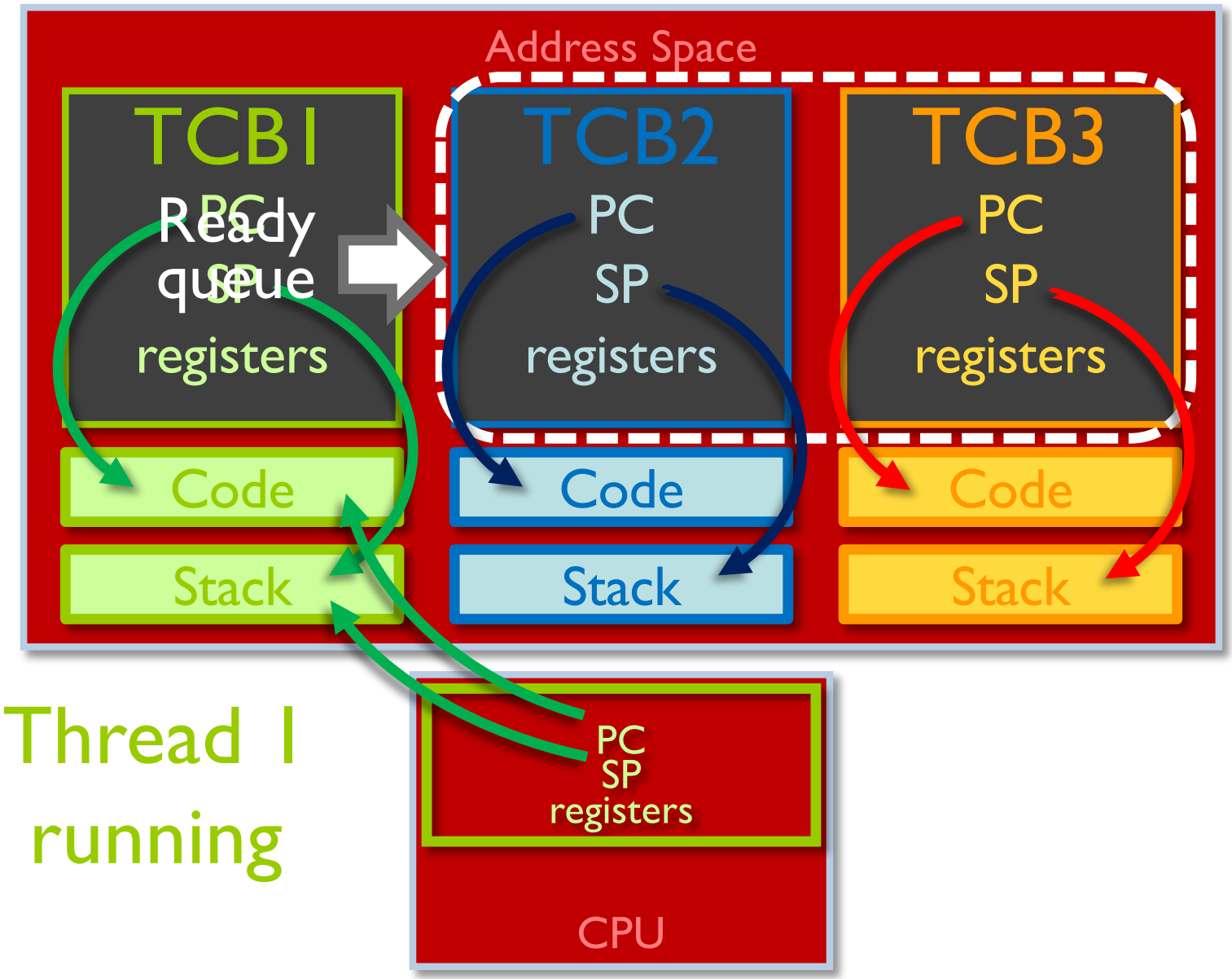
- What state is private to each thread?
  - Code (like lines of a play)
  - PC (where actor is in his/her script)
  - Stack, SP (actor's mindset)
- What state is shared?
  - Address space
  - Global variables, heap
  - (props on set)

Thread's "context"

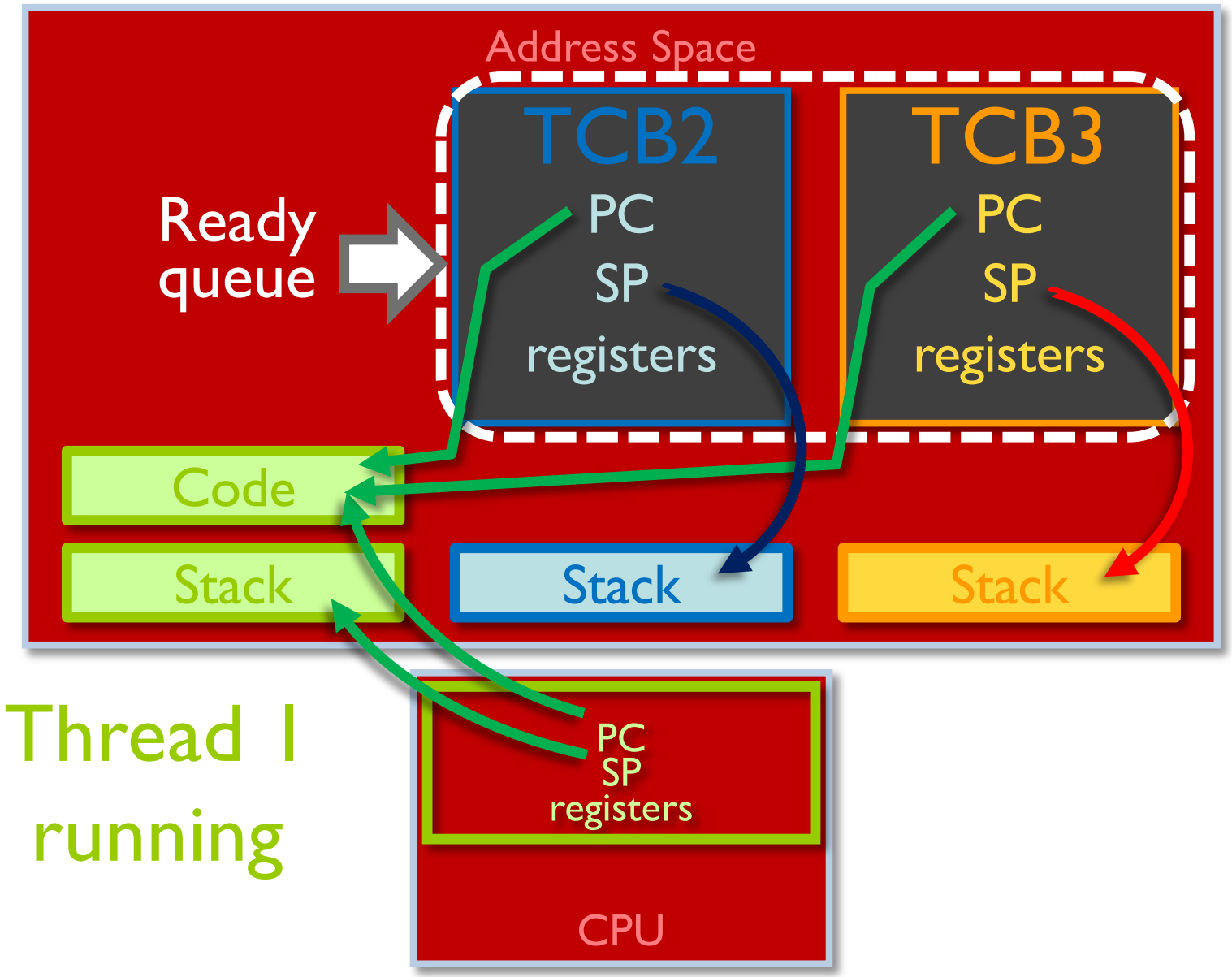
Thread control block (TCB)

- What needs to access threads' private data?
  - The CPU
  - This info is stored in the PC, SP, other registers
- The OS needs pointers to non-running threads' data
  - **Thread control block (TCB)**
    - Container for non-running thread's private data
    - Values of PC, code, SP, stack, registers
  - How can we save space?
    - Share code among threads
    - Don't copy stack to TCB
      - Use multiple stacks in same address space
      - Keep copy of SP in TCB

Thread control block



Thread control block

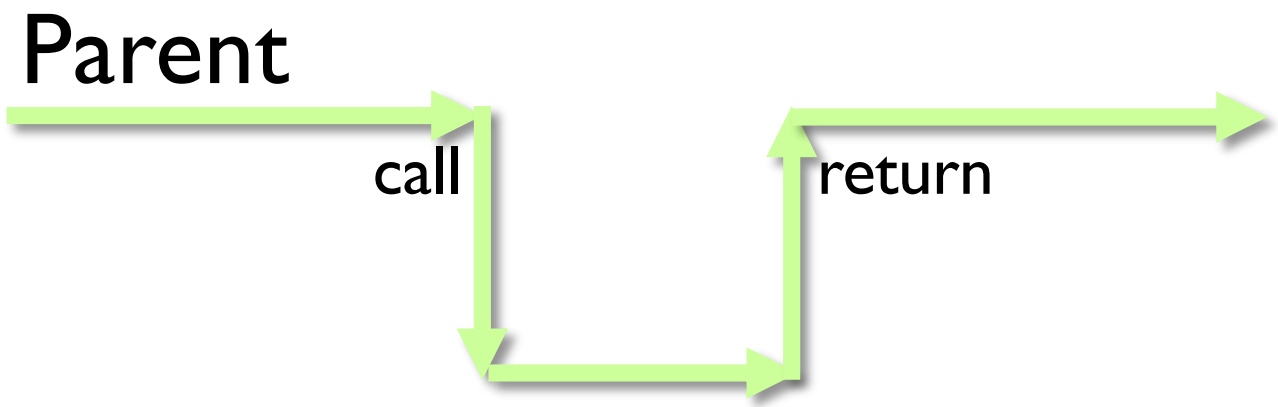


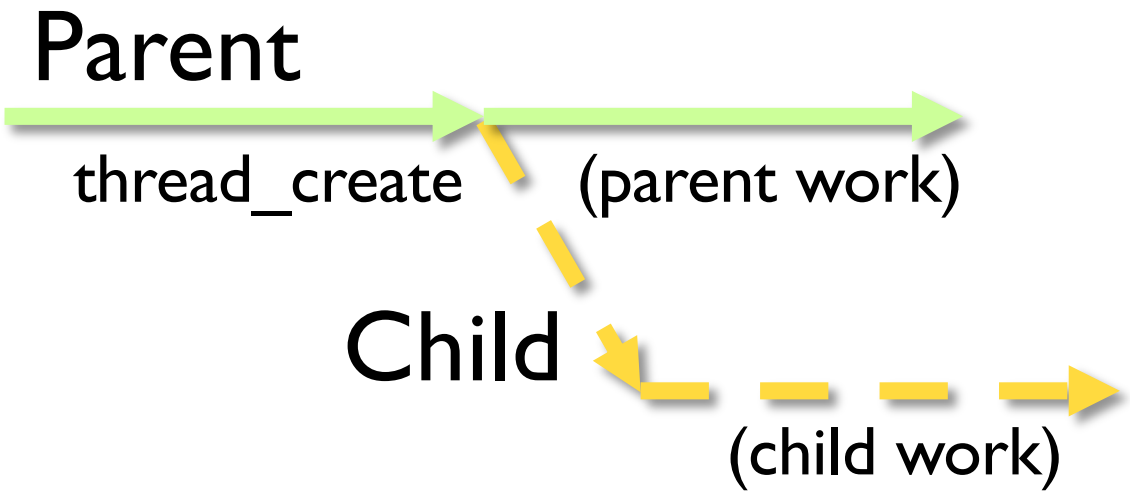
Creating a new thread

- 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
4. **Add thread to ready queue**

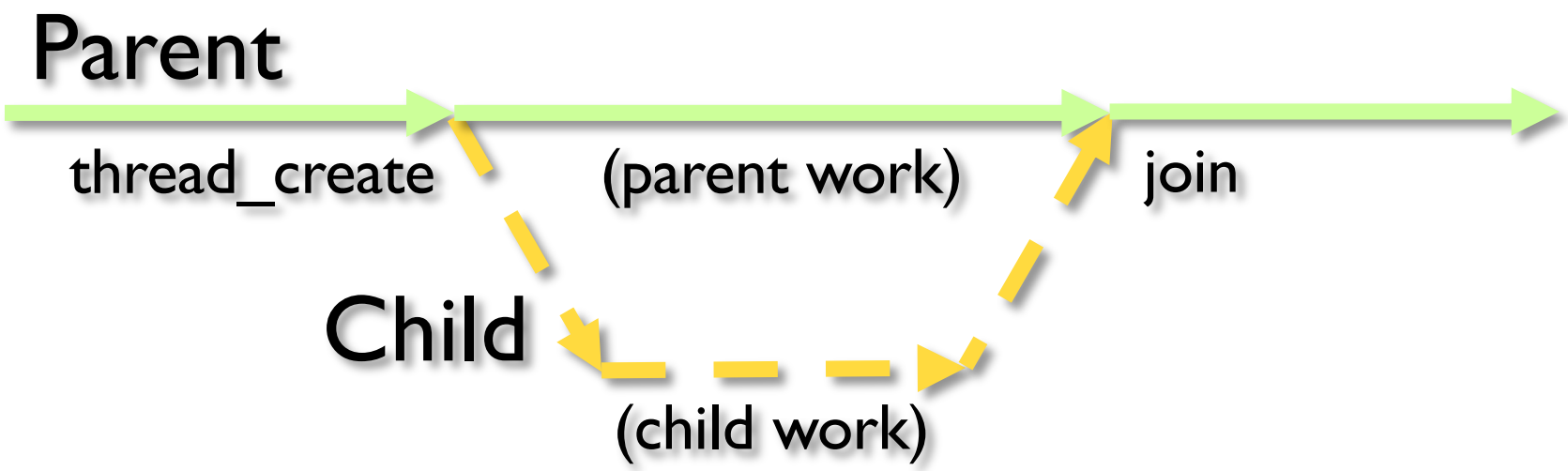
Creating a new thread





Thread join

- How can the parent wait for child to finish?



Thread join

- Will this work?
- Sometimes, assuming
  - Uni-processor
  - No pre-emptions
- Never, ever assume these things!
- Yield is like slowing the CPU
  - Program must work +- any yields

```
child () {
  print "child works"
}
```

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

```
parent works
child works
parent continues
```

```
child works
parent works
parent continues
```