PLEASE
SCAN

Synchronization

Today: Implementation issues

Readers/Writers Lock

- A common variant for mutual exclusion
  - One writer at a time, if no readers
  - Many readers, if no writer
- How might we implement this?
  - ReaderAcquire(), ReaderRelease()
  - WriterAcquire(), WriterRelease()
  - Need a lock to keep track of shared state
  - Need condition variables for waiting if readers/ writers are in progress
  - Some state variables

Readers/Writers Lock

```
Lock lock = FREE
CV okToRead = nil
CV okToWrite = nil
AW = 0  //active writers
AR = 0   // active readers
WW = 0 // waiting writers
WR = 0  // waiting readers
```

```
Lock lock = FREE

CV okToRead = nil
CV okToWrite = nil

AW = 0
AR =  0
WW = 0
WR = 0
```

```
lock.Acquire();
while (AW > 0 || WW > 0) {
   WR++;
   okToRead.wait(&lock);
   WR--;
}
AR++;
lock.Release();

Read data

lock.Acquire();
AR--;
if (AR == 0 && WW > 0)
  okToWrite.Signal();
lock.Release();
```

```
lock.Acquire();
while (AW > 0 || AR > 0) {
   WW++;
   okTo~~Read~~Write.wait(&lock);
   WW--;
}
AW++;
lock.Release();

Write data

lock.Acquire();
AW--;
if (WW > 0)
  okToWrite.Signal();
else if (WR > 0)
 okToRead.Broadcast();
lock.Release();
```

Readers/Writers Lock

- Can readers starve?
  - Yes: writers take priority
- Can writers starve?
  - Yes: a waiting writer may not be able to proceed, if another writer slips in between signal and wakeup

Readers/Writers Lock, w/o Writer Starvation Take 1

```
Writer() {
 lock.Acquire();
 // check if another thread is already waiting
 while ((AW + AR + WW) > 0) {
     WW++;
     okToWrite.Wait(&lock);
     WW--;
  }
 AW++;
 lock.Release();
```

Readers/Writers Lock w/o Writer Starvation Take 2

```
// check in
lock.Acquire();
myPos = numWriters++;
while ((AW + AR > 0 ||
        myPos > nextToGo) {
   WW++;
   okToWrite.Wait(&lock);
   WW--;
}
AW++;
lock.Release();
```

```
// check out
lock.Acquire();
AW--;
nextToGo++;
if (WW > 0) {
  okToWrite.Signal(&lock);
} else if (WR > 0)
   okToRead.Bcast(&lock);
lock.Release();
```

Readers/Writers Lock w/o Writer Starvation Take 3

```
// check in
lock.Acquire();
myPos = numWriters++;
myCV = new CV;
writers.Append(myCV);
while ((AW + AR > 0 | |
          myPos > nextToGo) {
   WW++;
   myCV.Wait(&lock);
   WW--;
}
AW++;
delete myCV;
lock.Release();
```

```
// check out
lock.Acquire();
AW--;
nextToGo++;
 if (WW > 0) {
     cv = writers.RemoveFront();
     cv.Signal(&lock);
 } else if (WR > 0)
     okToRead.Broadcast(&lock);
lock.Release();
```

Mesa vs. Hoare semantics

- Mesa
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock goes back to signaller
- All systems you will use are Mesa

FIFO Bounded Buffer
(Hoare semantics)

```
get() {
   lock.acquire();
   if (front == tail) {
      empty.wait(&lock);
   }
   item = buf[front % MAX];
   front++;
   full.signal(&lock);
   lock.release();
   return item;
}
```

```
put(item) {
   lock.acquire();
   if ((tail – front) == MAX) {
      full.wait(&lock);
   }
   buf[last % MAX] = item;
   last++;
   empty.signal(&lock);
  // CAREFUL: someone else ran
   lock.release();
}
```

Initially: front = tail = 0; MAX is buffer capacity

empty/full are condition variables

FIFO Bounded Buffer (Mesa semantics)

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!

- Easily extends to case where queue is LIFO, priority, priority donation, …
  - With Hoare semantics, not as easy

FIFO Bounded Buffer
(Mesa semantics, put() is similar)

```
get() {
    lock.acquire();
    myPosition = numGets++;
    cv = new CV;
    nextGet.append(cv);
    while (front < myPosition
            || front == tail) {
      cv.wait(&lock);
     }
    delete cv;
    item = buf[front % MAX];
    front++;
    if (next = nextPut.remove()) {
        next->signal(&lock);
    }
    lock.release();
    return item;
}
```

Initially: front = tail = numGets = 0; MAX is buffer capacity

nextGet, nextPut are queues of Condition Variables

Implementing Synchronization

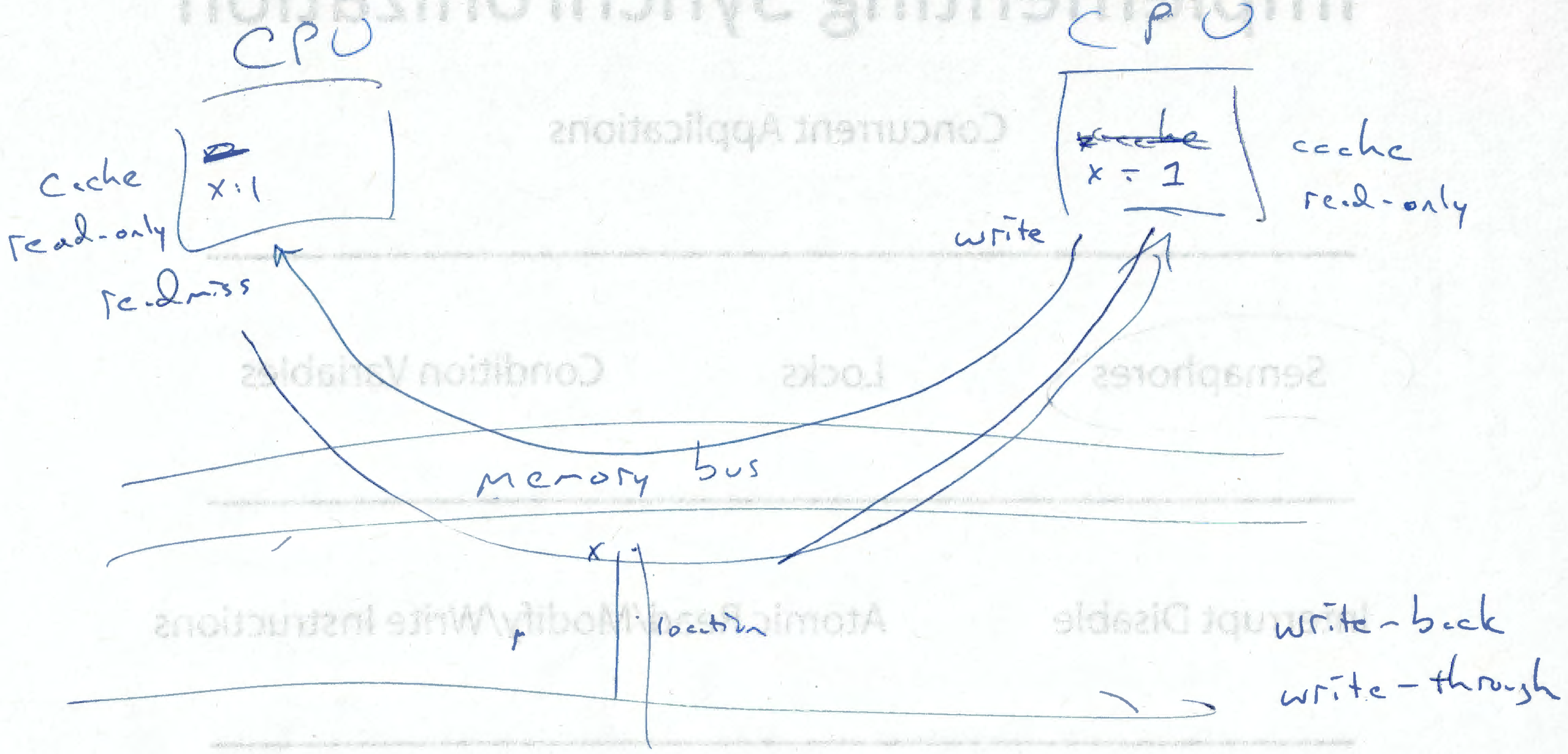
Concurrent Applications

Semaphores   Locks   Condition Variables

Interrupt Disable   Atomic Read/Modify/Write Instructions

Multiple Processors   Hardware Interrupts

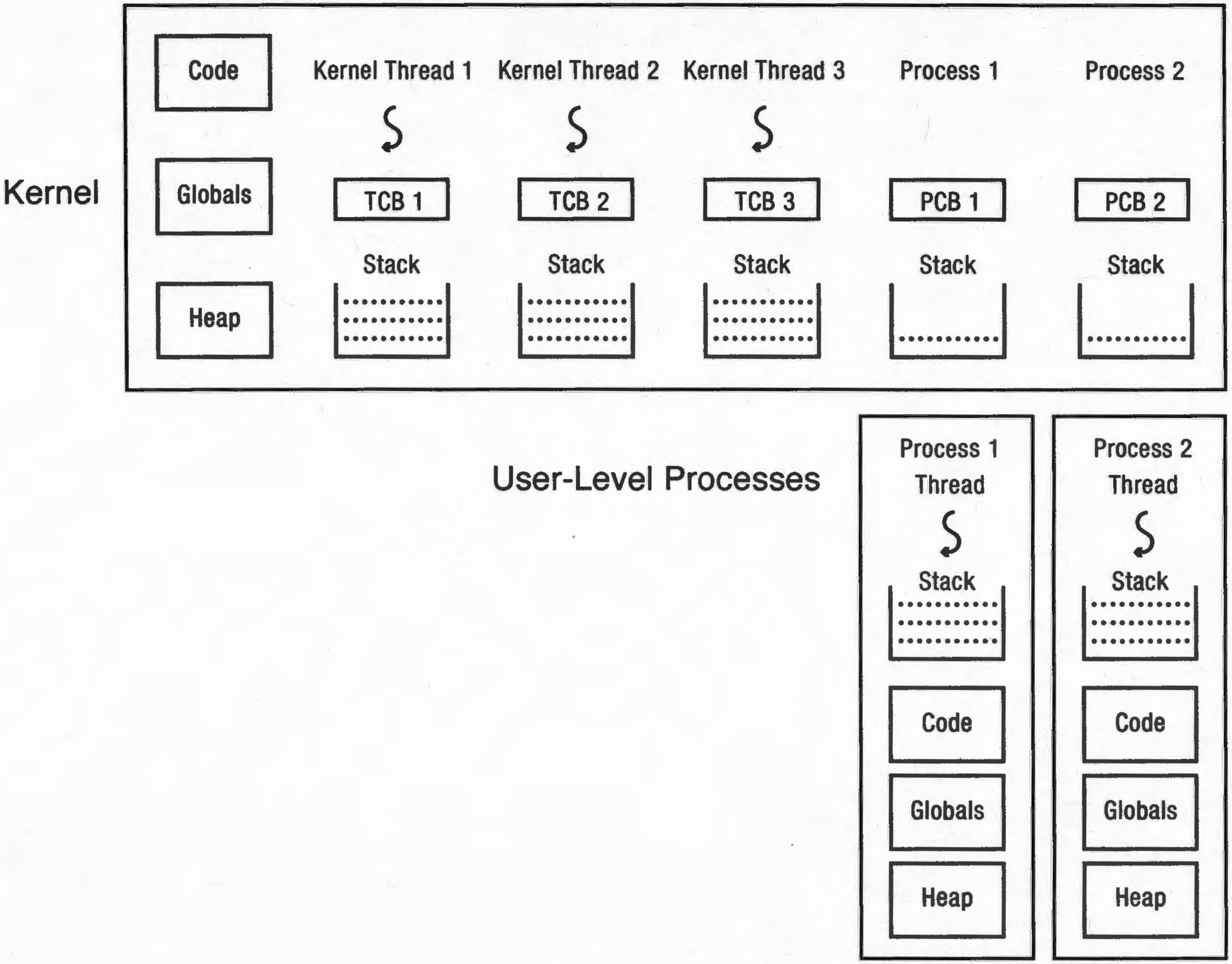
CACHE COHERENCE



Implementing Threads: Roadmap

- Kernel threads
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads (Linux, MacOS, Windows)
  - Kernel thread operations available via syscall
- User-level threads (Windows)
  - Thread operations without system calls

Multithreaded OS Kernel



Thread Context Switch

- Voluntary
  - Thread_yield
  - Thread_join (if child is not done yet)

Lock Acquire
– lock is busy

Condition Wait

- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

Voluntary thread context switch

- Called by old thread
- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return to new thread

- Exactly the same with kernel threads or user threads

x86 swtch

```
push %rbp
push %rbx
push %r11
push %r12
push %r13
push %r14
push %r15

mov %rsp, (%rdi)
mov %rsi, %rsp
```

```
pop %r15
pop %r14
pop %r13
pop %r12
pop %r11
pop %rbx
pop %rbp

ret
```

```
// save/restore callee save registers, not caller save
```

A Subtlety

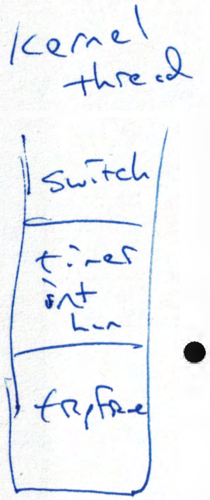
forkret

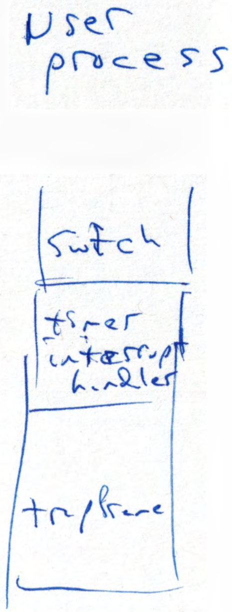
- Thread_create puts new thread on ready list
- Some thread calls switch, picks that thread to run next
  - Saves old thread state to stack
  - Restores new thread state from stack
- What does the new thread stack contain so this will work?
  - Set up thread's stack as if it had saved its state in switch
  - "returns" to PC saved at base of stack to run thread

Two Threads Call Yield

Thread 1's instructions	**Thread 2's instructions**	**Processor's instructions**
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch

Involuntary Thread/Process Switch (Simple, Slow Version)

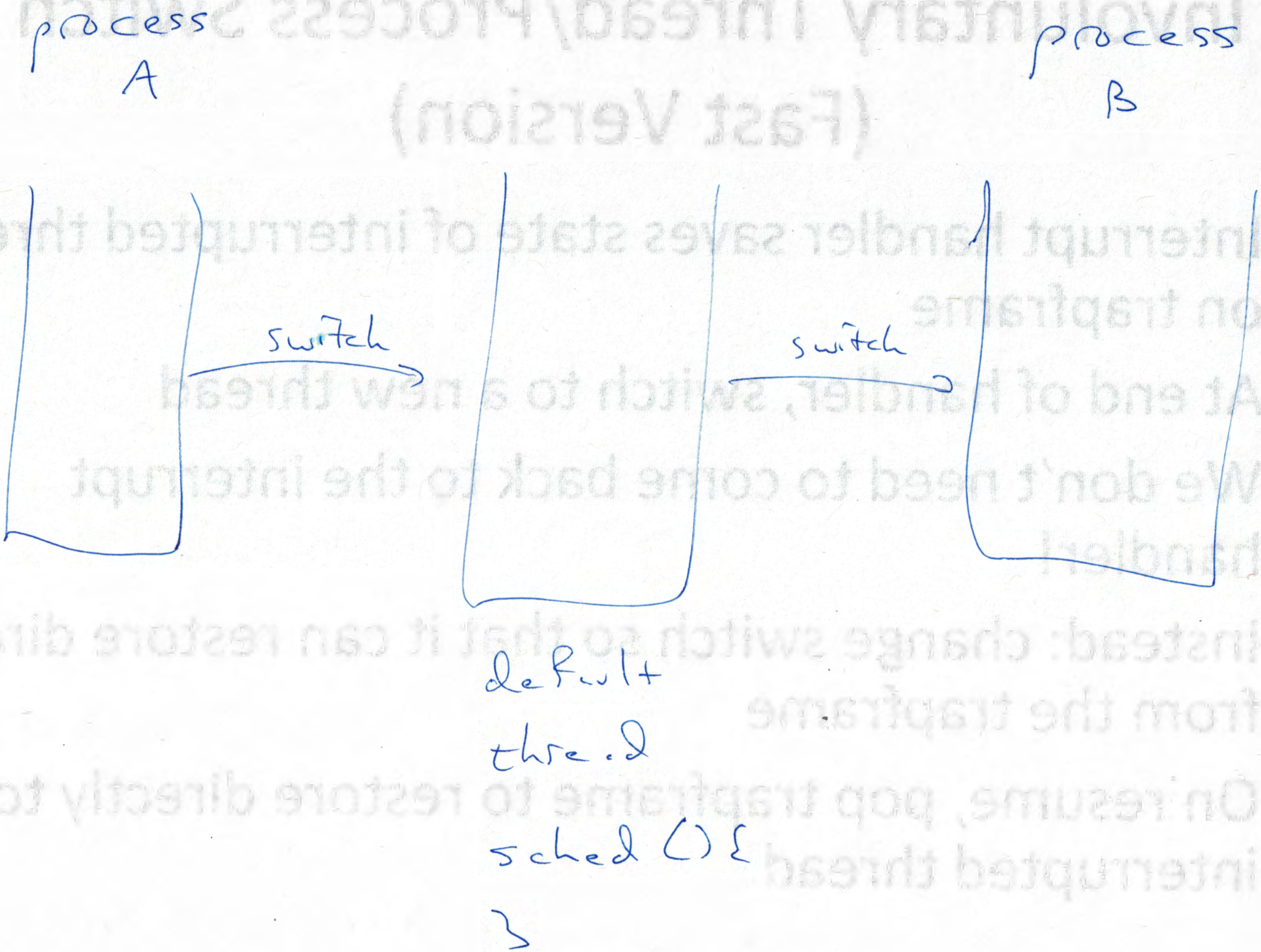




- Timer or I/O interrupt
  - Tells OS some other thread/process should run
- End of interrupt handler calls switch, before resuming the trapframe
- When thread is switched back in, resumes the handler
- Handler restores the trapframe to resume the user process

Involuntary Thread/Process Switch (Fast Version)

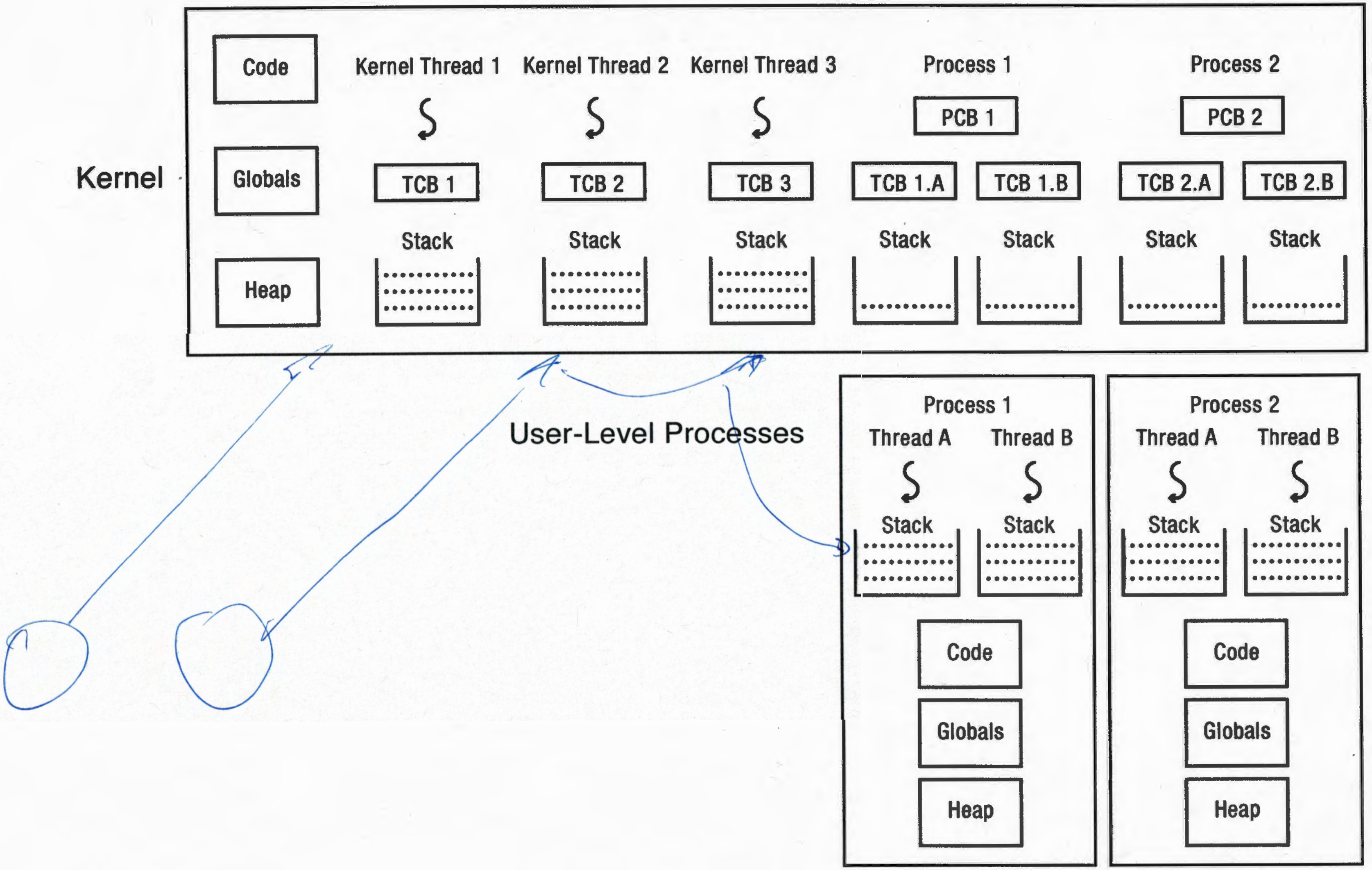
- Interrupt handler saves state of interrupted thread on trapframe
- At end of handler, switch to a new thread
- We don’t need to come back to the interrupt handler!
- Instead: change switch so that it can restore directly from the trapframe
- On resume, pop trapframe to restore directly to the interrupted thread


process A
process B
switch
switch
default thre.d
sched(){
}

Multithreaded User Processes (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread fork, join, exit (and lock, unlock,...)
  - Kernel does context switch
  - Simple, but a lot of transitions between user and kernel mode

Multithreaded User Processes (Take 1)



Kernel

Code

Globals

Heap

Kernel Thread 1

TCB 1

Stack

Kernel Thread 2

TCB 2

Stack

Kernel Thread 3

TCB 3

Stack

Process 1

PCB 1

TCB 1.A

Stack

TCB 1.B

Stack

Process 2

PCB 2

TCB 2.A

Stack

TCB 2.B

Stack

User-Level Processes

Process 1

Thread A

Stack

Thread B

Stack

Code

Globals

Heap

Process 2

Thread A

Stack

Thread B

Stack

Code

Globals

Heap

Multithreaded User Processes (Take 2)

- Green threads (early Java)
  - User-level library, within a single-threaded process
  - Library does thread context switch
  - Preemption via upcall/UNIX signal on timer interrupt
  - Use multiple processes for parallelism
    - Shared memory region mapped into each process

Multithreaded User Processes (Take 3)

- Scheduler activations (Windows 8)
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - Thread library decides what thread to run next
- Upcall whenever kernel needs a user-level scheduling decision
  - Process assigned a new processor
  - Processor removed from process
  - System call blocks in kernel