Reading Reference: Textbook: Chapter 4

W8 – CONCURRENCY AND THREADS

Aakash Tyagi
CSCE 313 Spring 2017

This Week's Conversation

- Threads
  - A bit complex topic but central to our understanding of modern computer systems (HW and SW)
  - We will build concepts incrementally and tie the picture together at the end
  - We will continue the crux of discussion on threading after the spring break

Adapted from contemporary courses in OS/Systems taught at Berkeley, UW, TAMU, UIUC, and Rice. Some slides are from Anderson and Dahlin Text.
Special acknowledgment to Profs Gu/Bettati at TAMU, Culler and Joseph at Berkeley

Why Processes & Threads?

Goals:

- **Multiprogramming:** Run multiple applications concurrently
- **Protection:** Don’t want a bad application to crash system!

Solution:

- Virtual Machine abstraction: give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)
- Process: unit of execution and allocation

Challenge:

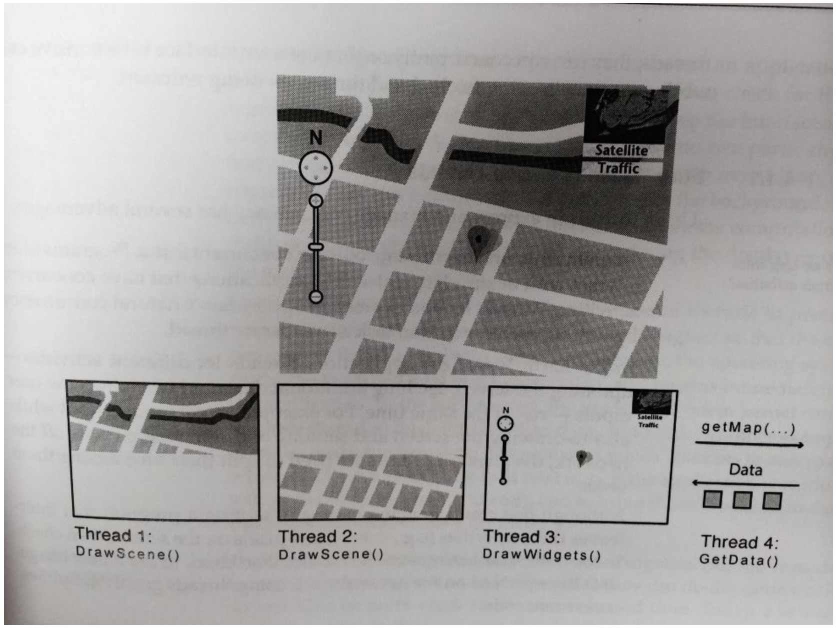
- Process creation & switching expensive
- Need concurrency within same app (e.g., web server)

Solution:

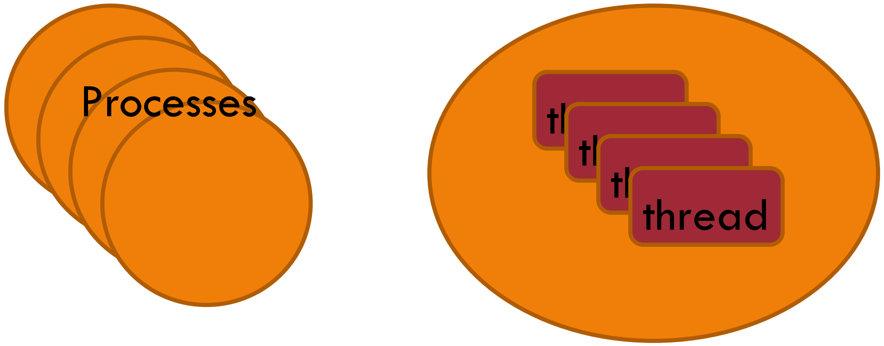
Thread: Decouple allocation and execution

- Run multiple threads within same process

Let's picture this scenario



Credit – NASA Earth Observatory



- Multiple programs (processes) doing their individual functions
- PAYING THE COST OF CONTEXT SWITCHING
- WHAT IF WE COULD REDUCE THE COST
- AT THE EXPENSE OF SOMETHING
- SO THE NET PERFORMANCE IS STILL BETTER

- Two 'threads' each draw parts of the scene, a third 'thread' manages the user interface widgets, and a fourth 'thread' fetches new data from the remote server
- In a 'traditional program' these will be sequenced
- Key differentiation is "facilitated concurrency".
- A traditional program is a single 'thread'
- Inside a program, we can represent each concurrent task as "Thread"

Motivation for SW Threads

- Operating systems need to be able to handle multiple things at once (MTAO)
  - processes, interrupts, background system maintenance
- Servers need to handle MTAO
  - Multiple connections handled simultaneously
- Parallel programs need to handle MTAO
  - To achieve better performance
- Programs with user interfaces often need to handle MTAO
  - To achieve user responsiveness while doing computation
- Network and disk bound programs need to handle MTAO
  - To hide network/disk latency

Motivation for HW Threads

- MTAO for performance
- Assists for SW threads

Motto for Threading

MTAO

Programs, Process, Threads, MultiCore, Multithreads…

- Scenario: Warehouse Accountant needs to
  - look at the invoice list
  - verify it against the purchase order list and
  - then issue a check for valid purchases.
- There are multiple sheets of purchase orders and invoices.
- More than one accountant are qualified and allowed to work on this assignment. Constraint is that an accountant can only work on his assigned list.
- An accountant must have a TAG before he can work on his list.
- There is only one tag to go around.

Scenario Build

- **Scenario1**: No accountant: **Analogous to a "Program"**

Scenario Build

- **Scenario2**: Accountant present in Room1 with a TAG, along with all lists and a checkbook: **Akin to a Process in execution**

Scenario Build

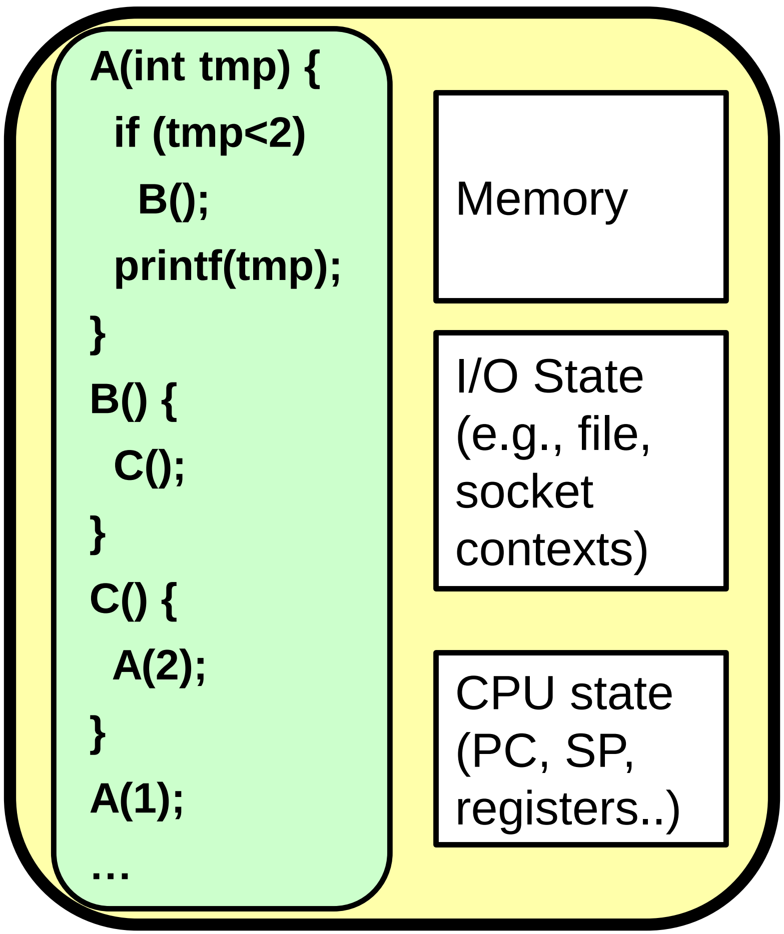
- **Scenario3**: Accountant2 shows up in Room2. At some point Accountant1 goes on lunch break
  - His stopping point and local records are saved (aka PCB).
  - Tag and Global lists (invoice and PO) are walked over to Room2 so Accountant 2 can take over.
- When Accountant 2 takes a break, the tag and lists are walked over to Room1 again…..**Akin to 2 processes and context switch from Process1 to Process2.**

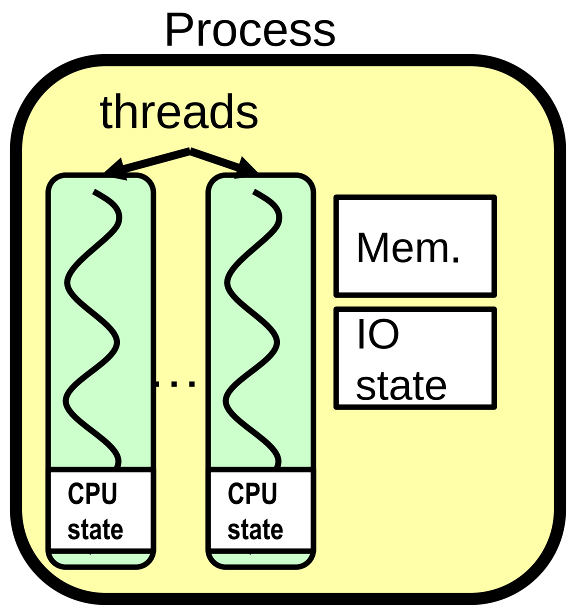
Scenario Build

- **Scenario4**: Now picture Accountants 1 and 2 are sitting in the **same room** with global lists visible to both.
  - Whoever gets the TAG recalls his previous stopping point and resumes.
  - There’s no walking across the rooms, carrying lists etc.
  - They just have to be careful not to clobber over each other’s lists. This is akin to **Threads**.

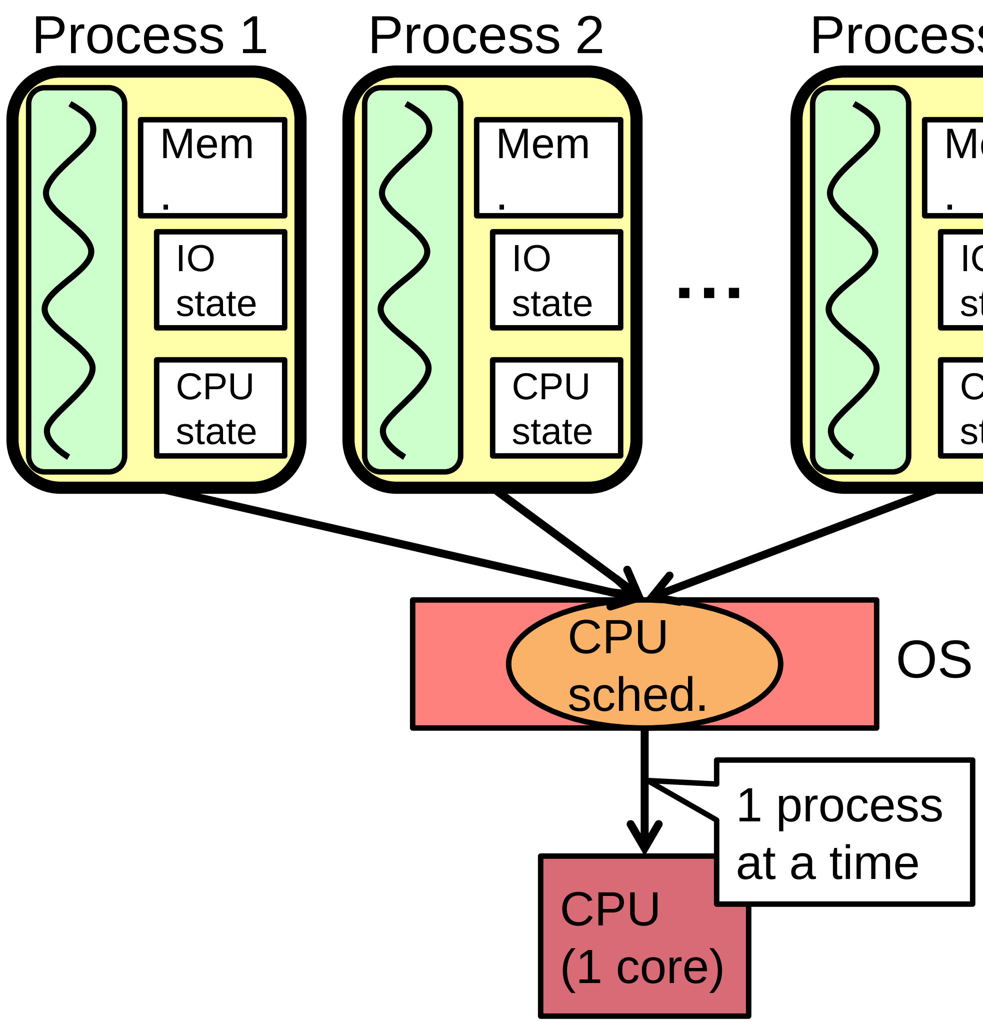
Putting it together: Process

(Unix) Process



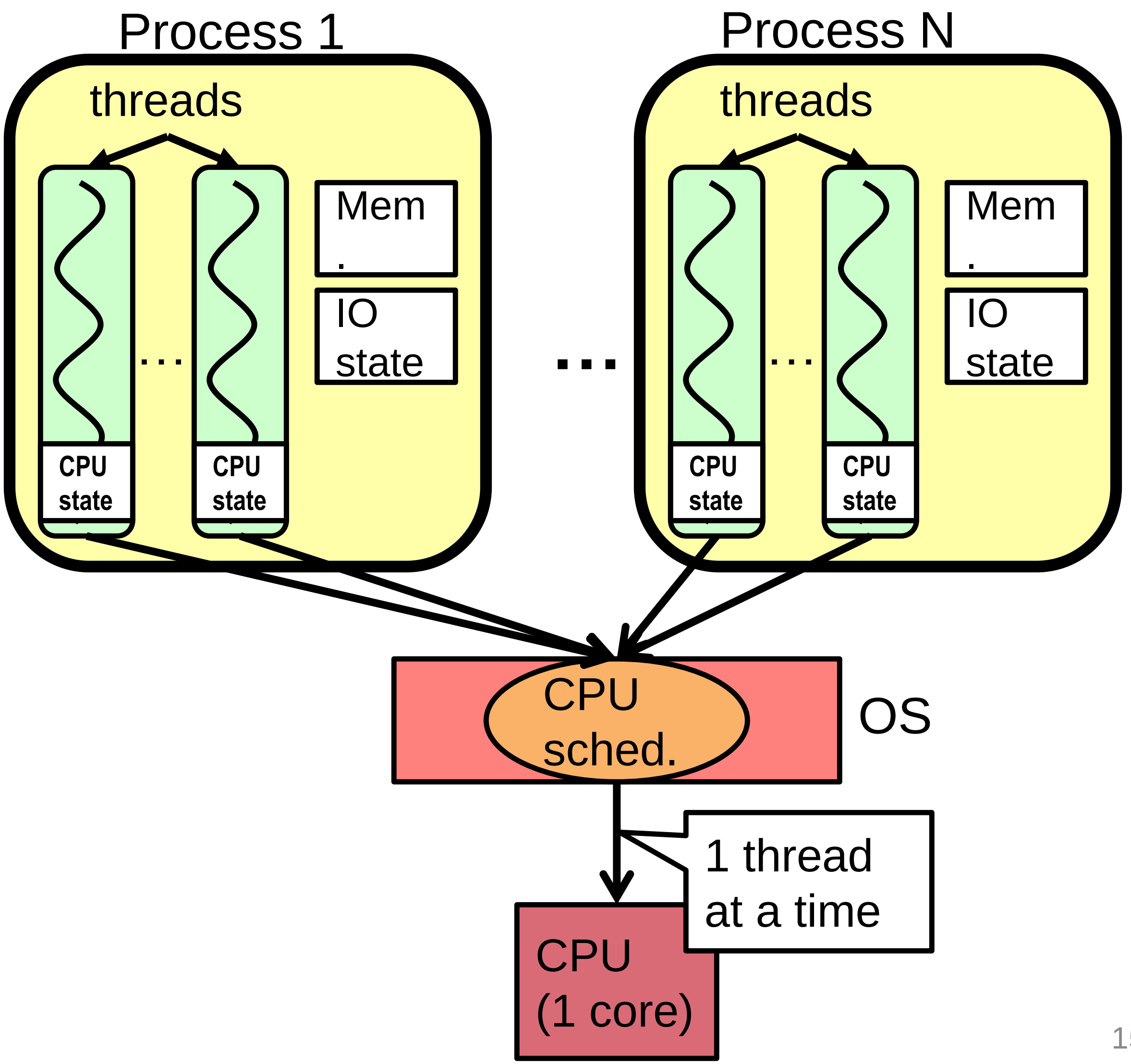


Putting it together: Processes



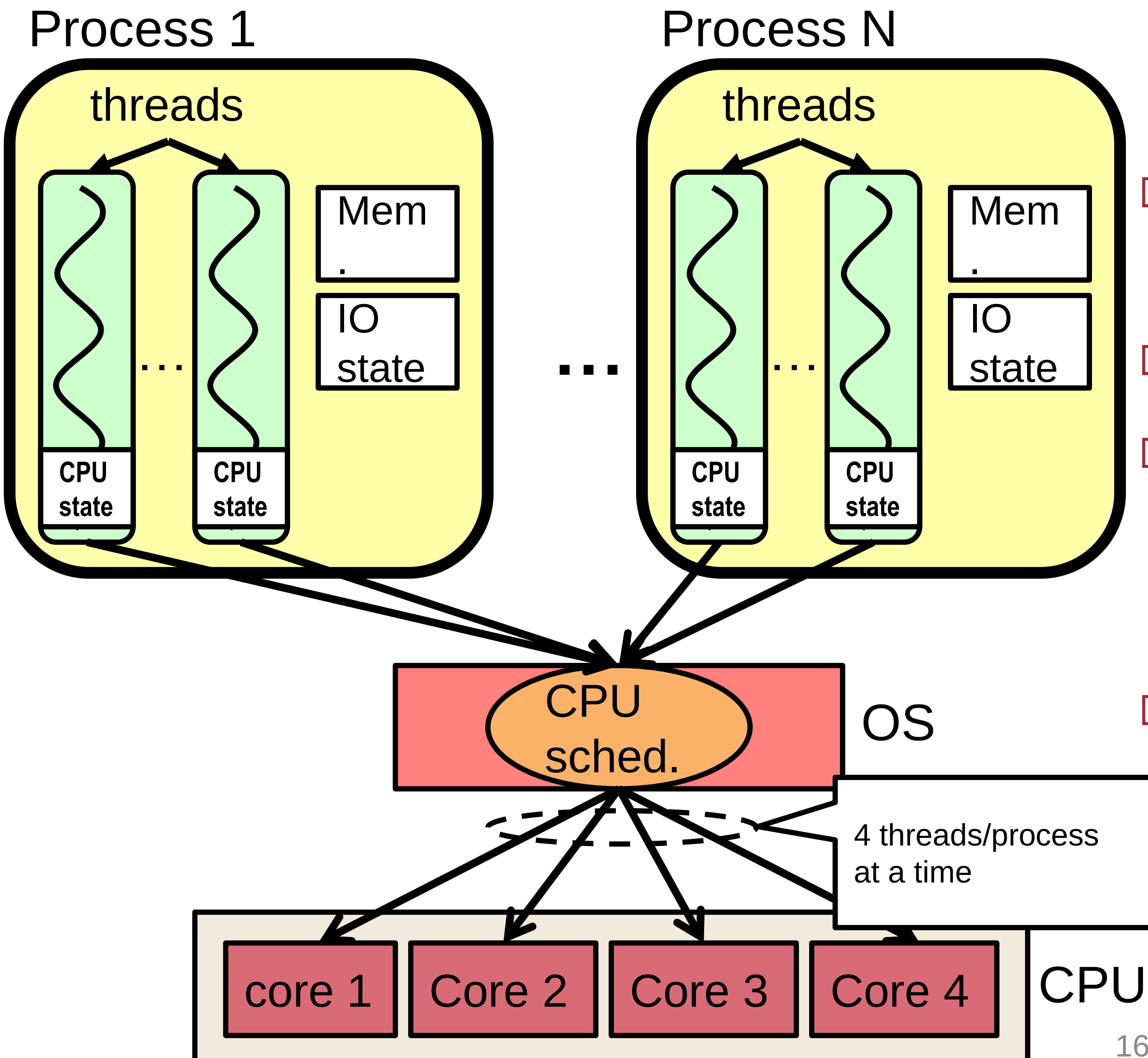
- **Process** Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: **yes**
  - Memory/IO: **yes**
- Sharing overhead: high (involves at least a context switch)

Putting it together: Threads



- **Thread** Switch overhead: **low** (only CPU state)
- Thread creation: **low**
- Protection
  - CPU: **yes**
  - Memory/IO: No
- Sharing overhead: **low**

Putting it together: Multi-Cores

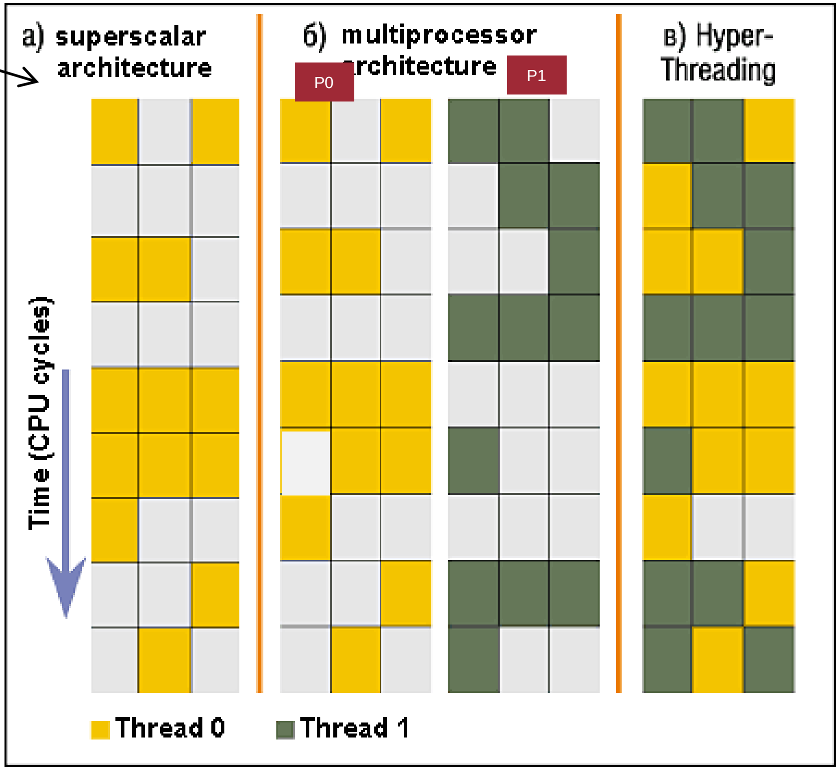


- Switch overhead: **low** (only CPU state)
- Thread creation: **low**
- Protection
  - CPU: **yes**
  - Memory/IO: No
- Sharing overhead: **low** (thread switch overhead low)

Hardware Parallelism (only for reference)

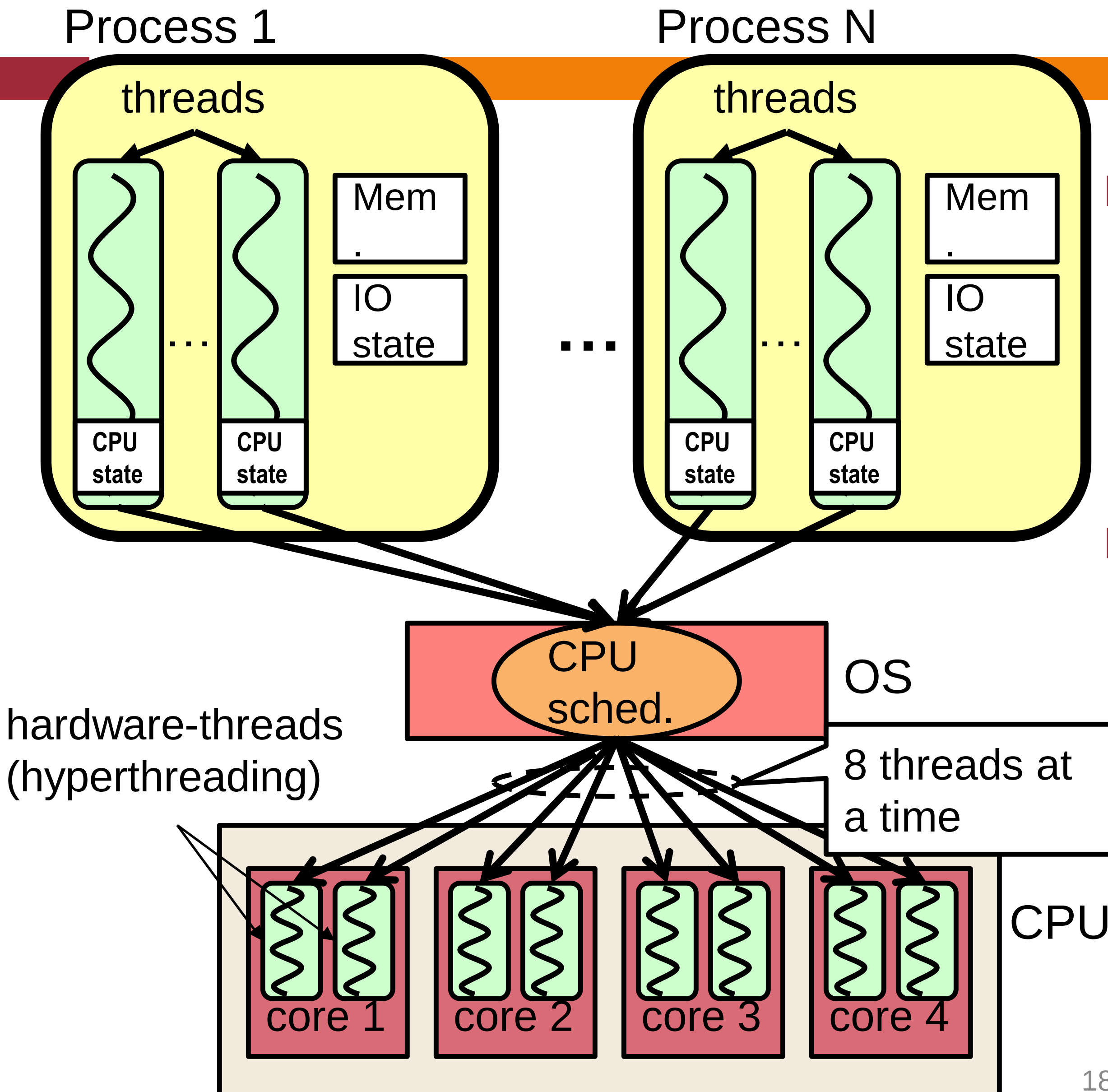
- Hardware technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyper-threading duplicates register state to make a second “thread,” allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!
- Original technique called “Simultaneous Multithreading”
  - See http://www.cs.washington.edu/research/smt/index.html
  - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5

Triple issue SS arch



Colored blocks show instructions executed

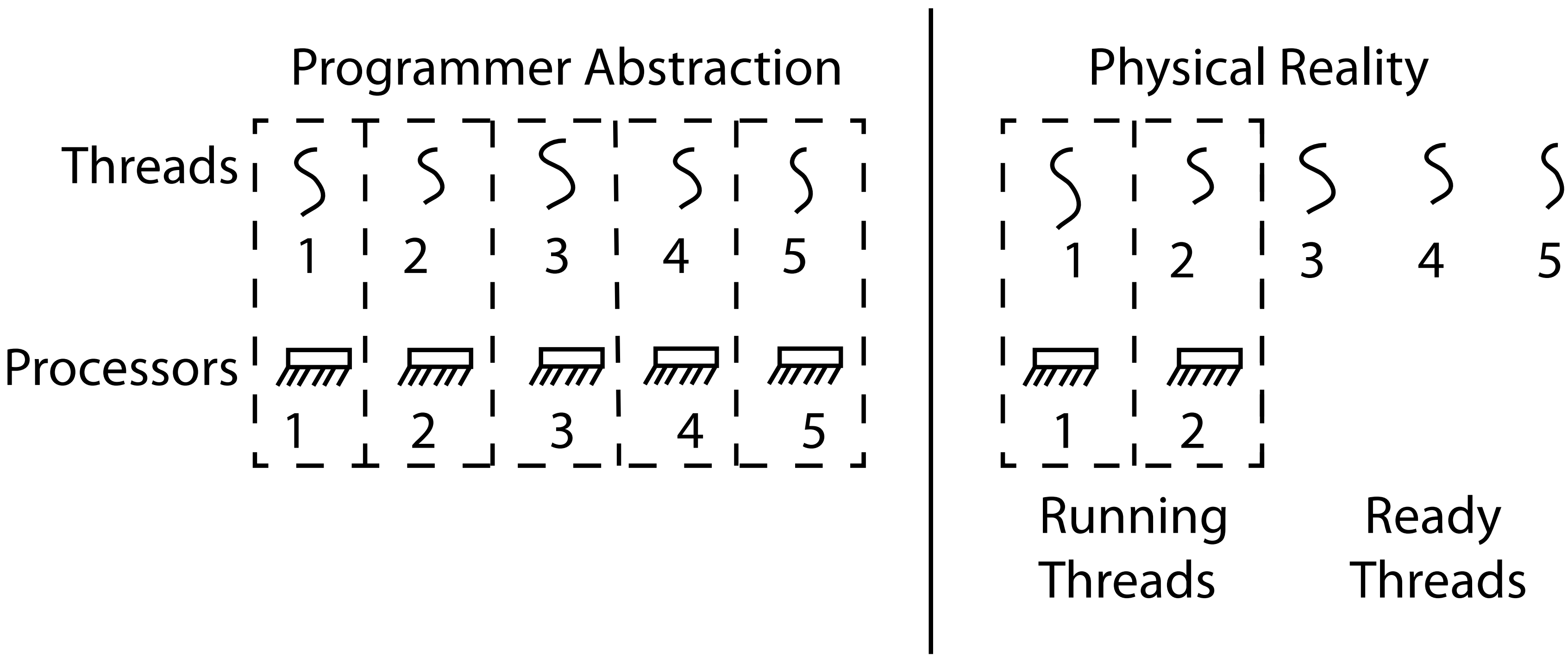
Putting it together: Hyper-Threading



- Switch overhead between hardware-threads: **very-low** (done in hardware)
- Contention for ALUs/FPUs may hurt performance

Thread Abstraction

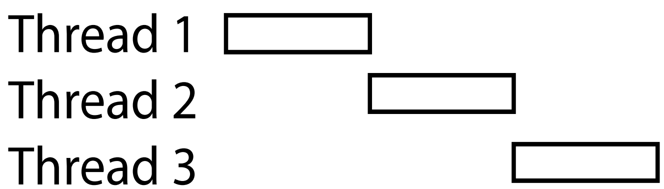
- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



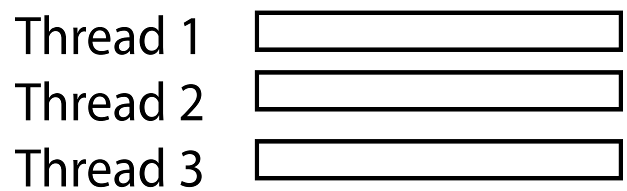
Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1	x = x + 1
y = y + x;	y = y + x;	....	y = y + x
z = x +5y;	z = x + 5y;	thread is suspended	....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	....	thread is resumed
		y = y + x	....
		z = x + 5y	z = x + 5y

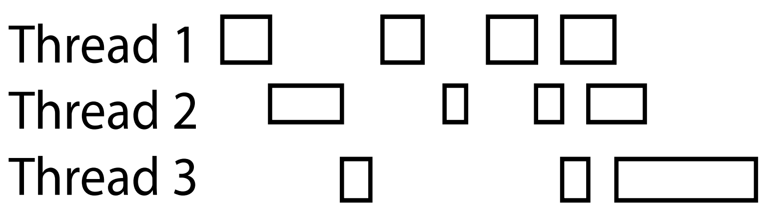
Possible Executions



a) One execution

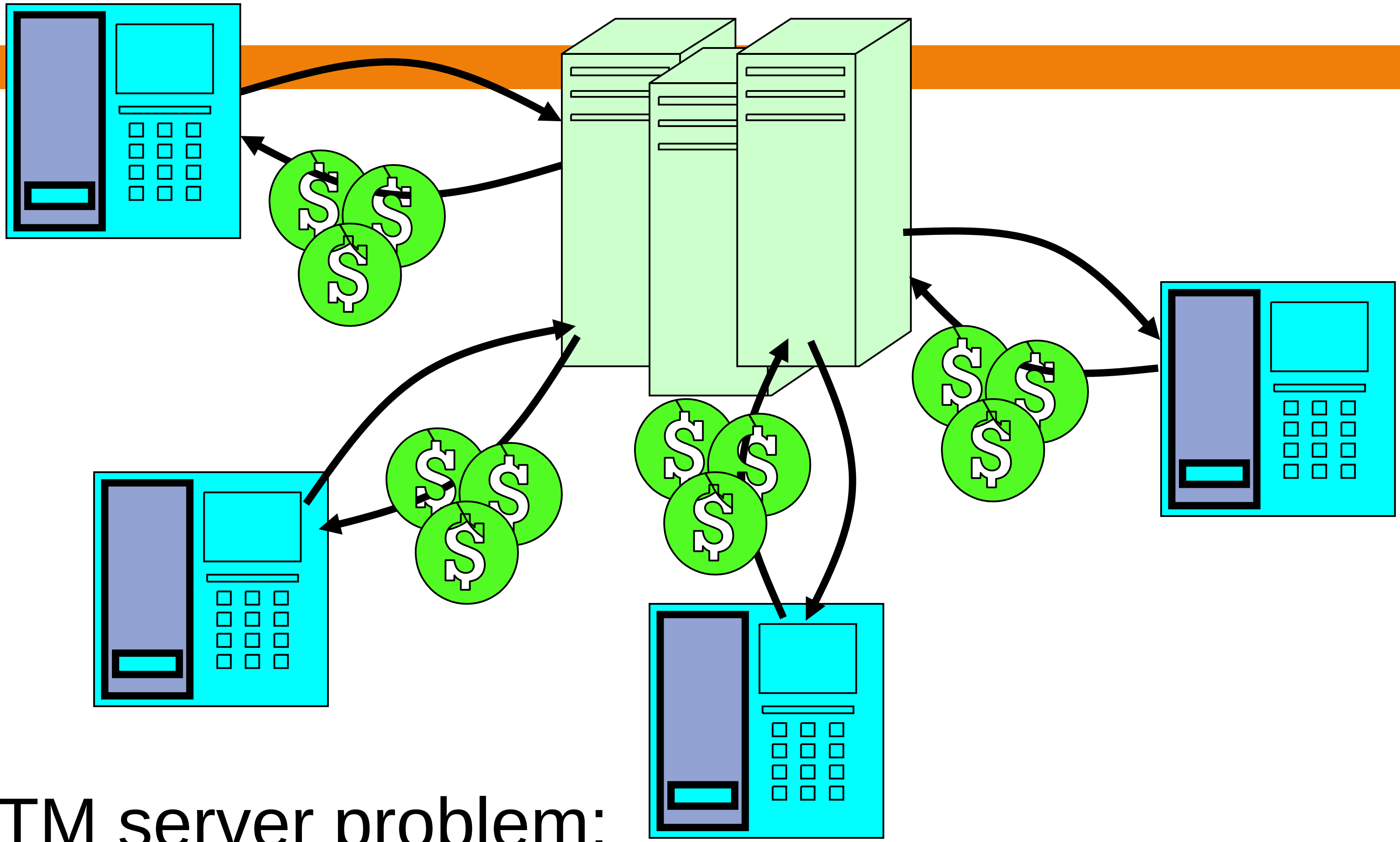


b) Another execution



c) Another execution

ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don’t hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
    while (TRUE) {
        ReceiveRequest(&op, &acctId, &amount);
        ProcessRequest(op, acctId, amount);
    }
}
ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if …
}
Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Multiple threads (multi-proc, or overlap comp and I/O)

Can Threads Help?

- One thread per request!
- Requests proceed to completion, blocking as required:

```
Deposit(acctId, amount) {
    acct = GetAccount(actId); /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}
```

Can Threads Help?

- Unfortunately, shared state can get corrupted:

```
        Thread 1                          Thread 2
load r1, acct->balance
                                  load r1, acct->balance
                                  add r1, amount2
                                  store r1,acct->balance
add r1, amount1
store r1, acct->balance
```

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x?

Thread A	Thread B
x = 1;	
x = y+1;	
	y = 2;
	y = y*2

x=13

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x?

Thread A	Thread B
x = 1; x = y+1;	

y = 2;
y = y*2;

x=5

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x?

Thread A	Thread B
	y = 2;
x = 1;	
x = y+1;	
	y= y*2;

x=3

Thread Operations API

- thread_create(thread, func, args)
  - Create a new thread to run func(args)
  - Analogous to UNIX process fork and exec
- thread_yield()
  - Relinquish processor voluntarily
- thread_join(thread)
  - In parent, wait for forked thread to exit, then return
  - Analogous to UNIX process wait
- thread_exit
  - Quit thread and clean up, wake up joiner if any

Example: threadhello


```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
    for (i = 0; i < NTHREADS; i++)  thread_create(&threads[i], &go, i);
    for (i = 0; i < NTHREADS; i++) {
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n", i, exitValue);
    }
    printf("Main thread done.\n");
}
void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
    // REACHED?
}
```

threadhello – Example Output


```
[tyagi]@linux2 ~/csce313/sp17/thread> (16:10:14 03/05/17)
:: ./threadHello
Hello from thread 0
Hello from thread 4
Hello from thread 2
Hello from thread 3
Hello from thread 1
Hello from thread 5
Hello from thread 8
Hello from thread 7
Hello from thread 6
Hello from thread 9
Thread 0 returned with 100
Thread 1 returned with 101
Thread 2 returned with 102
Thread 3 returned with 103
Thread 4 returned with 104
Thread 5 returned with 105
Thread 6 returned with 106
Thread 7 returned with 107
Thread 8 returned with 108
Thread 9 returned with 109
Main thread done.
```

threadhello – Example Output


```
[tyagi]@linux2 ~/csce313/sp17/thread> (16:33:36 03/05/17)
:: ./threadHello
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 4
Hello from thread 5
Hello from thread 6
Hello from thread 7
Hello from thread 8
Thread 0 returned with 100
Thread 1 returned with 101
Thread 2 returned with 102
Thread 3 returned with 103
Hello from thread 9
Thread 4 returned with 104
Thread 5 returned with 105
Thread 6 returned with 106
Thread 7 returned with 107
Thread 8 returned with 108
Thread 9 returned with 109
Main thread done.
```

threadhello: Example Output

- Why might the "Hello" message from thread 2 print after the "Hello" message from thread 5 even though thread 5 was created after thread 2?
- Why must "thread returned" print in order?
- What is maximum # of threads that could exist when thread 5 prints hello?
- Minimum?

Fork/Join Concurrency

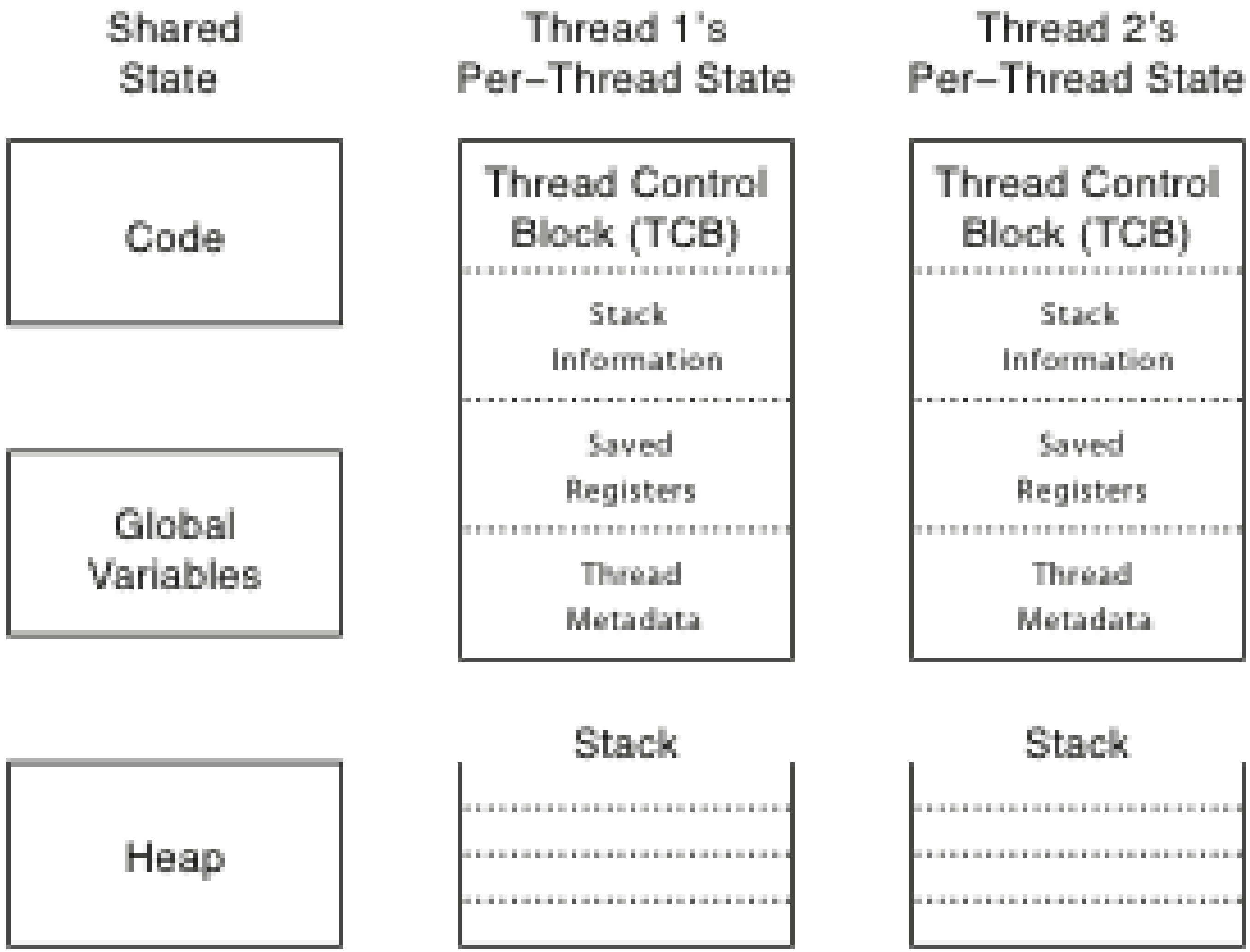
- Threads can create children, and wait for their completion
- Data only shared before fork (from parent) and after join (from child)
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

bzero with fork/join concurrency

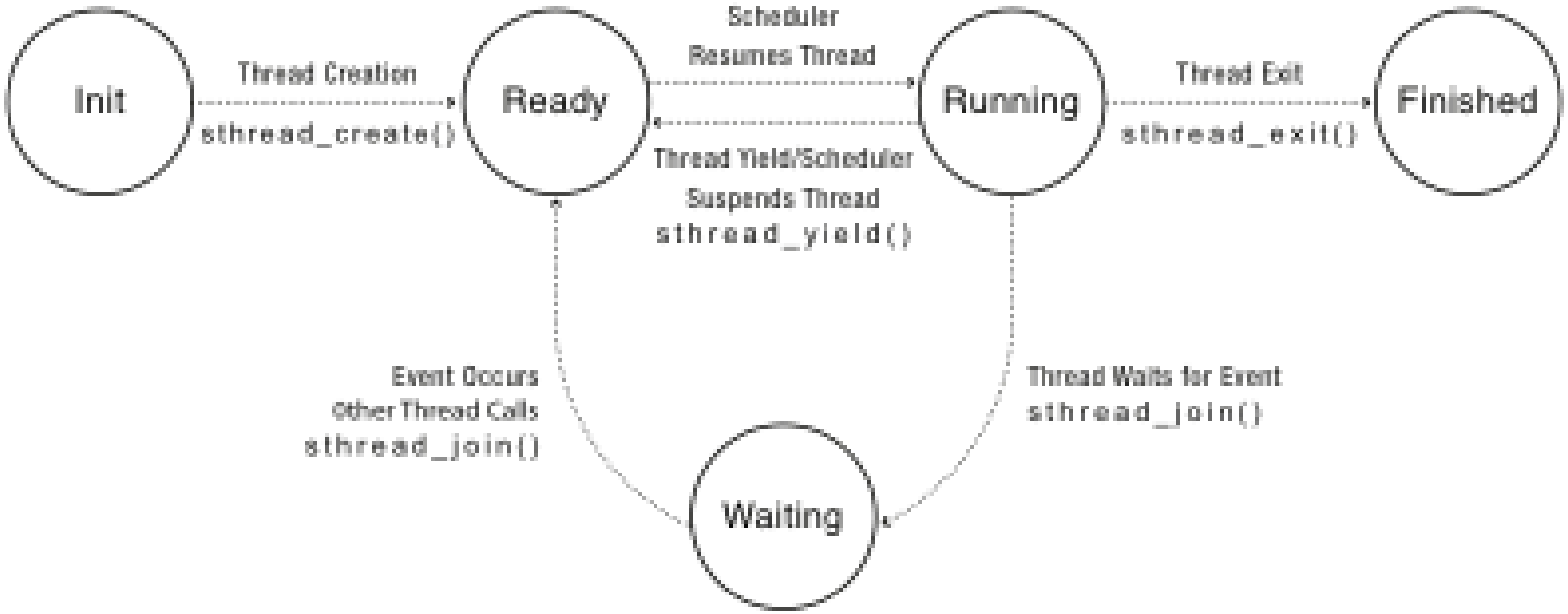

```
void blockzero (unsigned char *p, int length) {
   int i, j;
   thread_t threads[NTHREADS];
   struct bzeroparams params[NTHREADS];

// For simplicity, assumes length is divisible by NTHREADS.
for (i = 0, j = 0; i < NTHREADS; i++, j += length/NTHREADS) {
      params[i].buffer = p + i * length/NTHREADS;
      params[i].length = length/NTHREADS;
      thread_create_p(&(threads[i]), &go, &params[i]);
   }
   for (i = 0; i < NTHREADS; i++) {
      thread_join(threads[i]);
   }
}
```

Thread Data Structures



Thread Lifecycle



Location of per thread state for different life cycle

State of Thread	Location of TCB	Location of Thread Registers
INIT	Being created	TCB
READY	Ready Queue	TCB
RUNNING	Running Queue (may be single or multiple)	CPU
WAITING	Synch Variable Waiting List	TCB
FINISHED	Finished Queue and then deleted	TCB or deleted

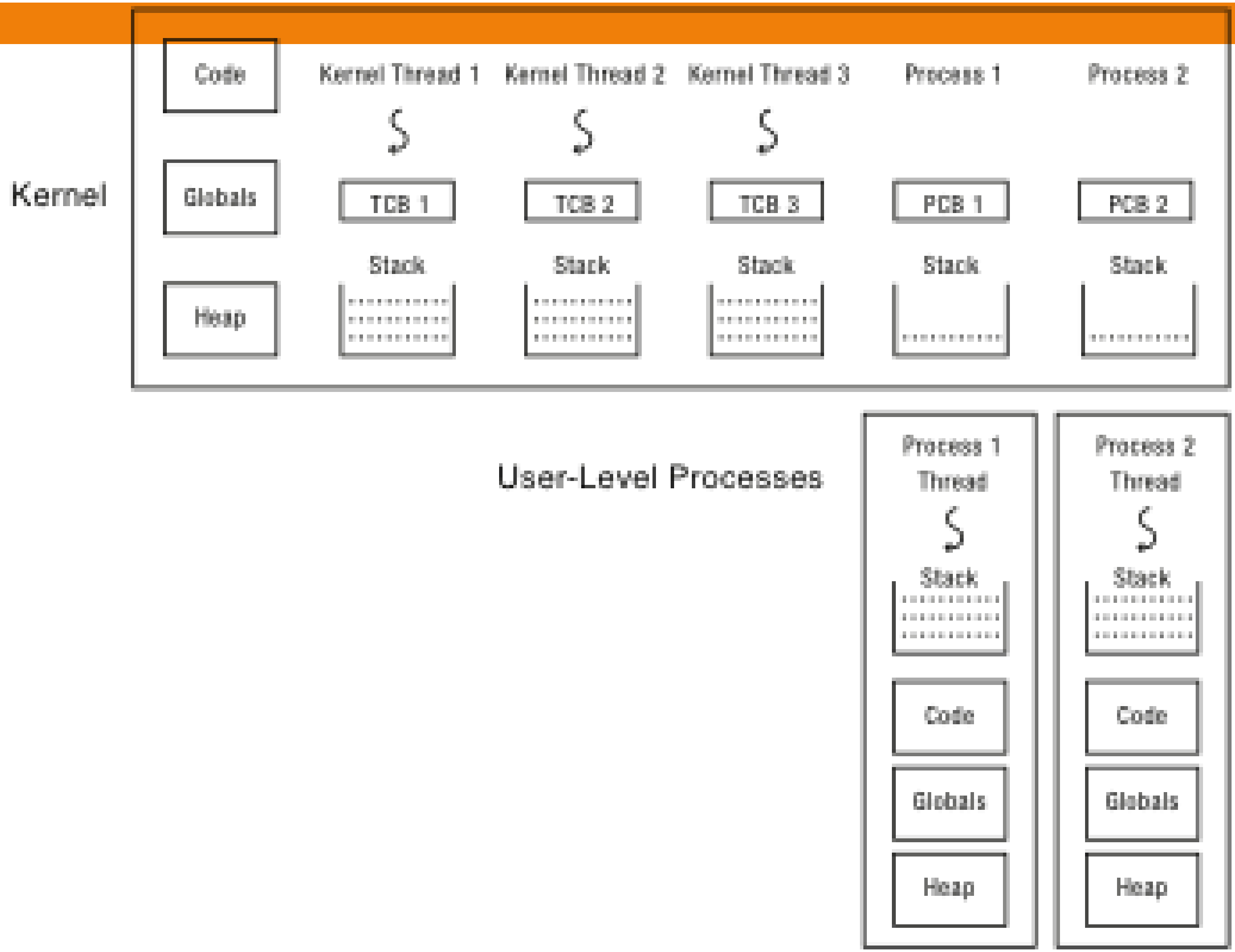
Discussion

- Question1: For the threadhello program, when thread_join returns for thread i, what is thread i's thread state?
- Question2: For the threadhello program, what is the minimum and maximum number of times that the main thread enters the READY state on a Uniprocessor?

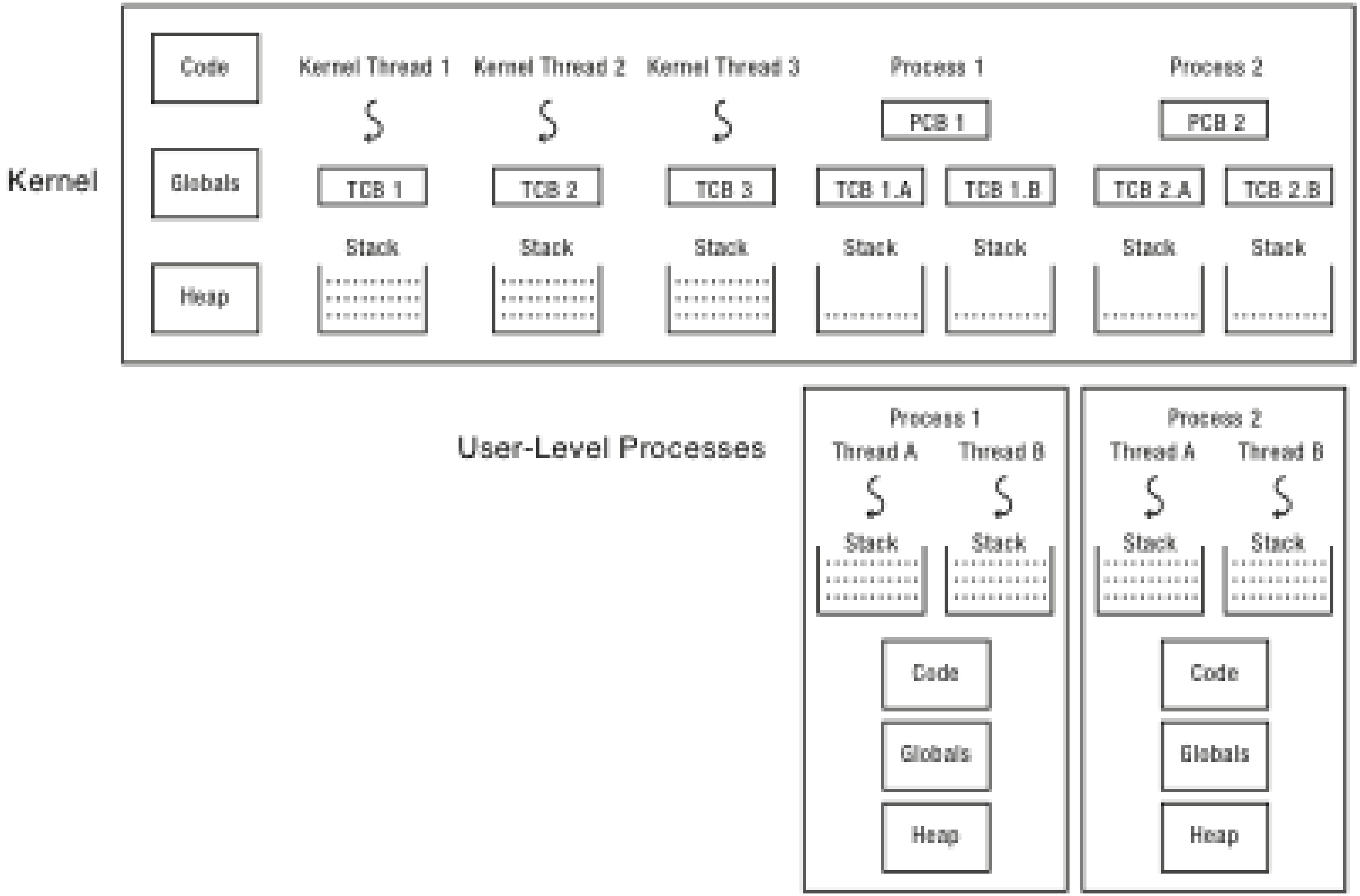
Implementing Threads

- Kernel-level 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)
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls (for efficiency purposes, the common cases can be offered as a library which can then be operated without kernel's help)

Multithreaded OS Kernel



Multithreaded User Processes



Implementing threads

- Thread_create(func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)
- stub(func, args):
  - Call (*func)(args)
  - If return, call thread_exit()

Thread Stack

- What if a thread puts too many procedures on its stack?
  - What happens in Java?
  - What happens in the Linux kernel?
  - What *should* happen?

Thread Context Switch

- Voluntary
  - Thread_yield
  - Thread_join (if child is not done yet)
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

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 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

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

Summary

- Threading is great for performance
- Threading is tricky and carry hazards!
  - After Spring Break – we will dive into hazard scenarios and prevention mechanisms
  - MP 6, 7, 8 will all build upon coding with threads