CS 162: Operating Systems and Systems Programming

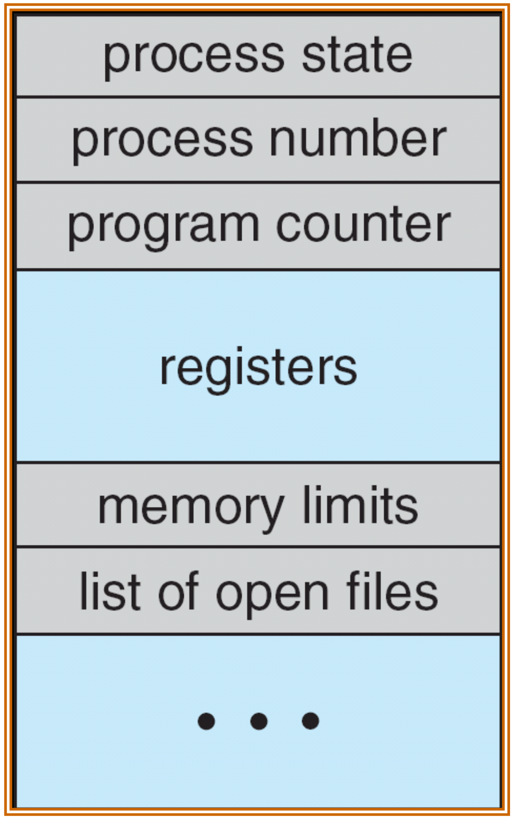
Lecture 4: Threads and Concurrency

Sept 10, 2019
Instructor: David E. Culler
https://cs162.eecs.berkeley.edu

Read: A&D 5.1-3, 5.7.1
HW 1 due 9/18
Groups and Section Pref Wed
Proj 1 released, Design doc Tues

Recall: Multiplexing Processes

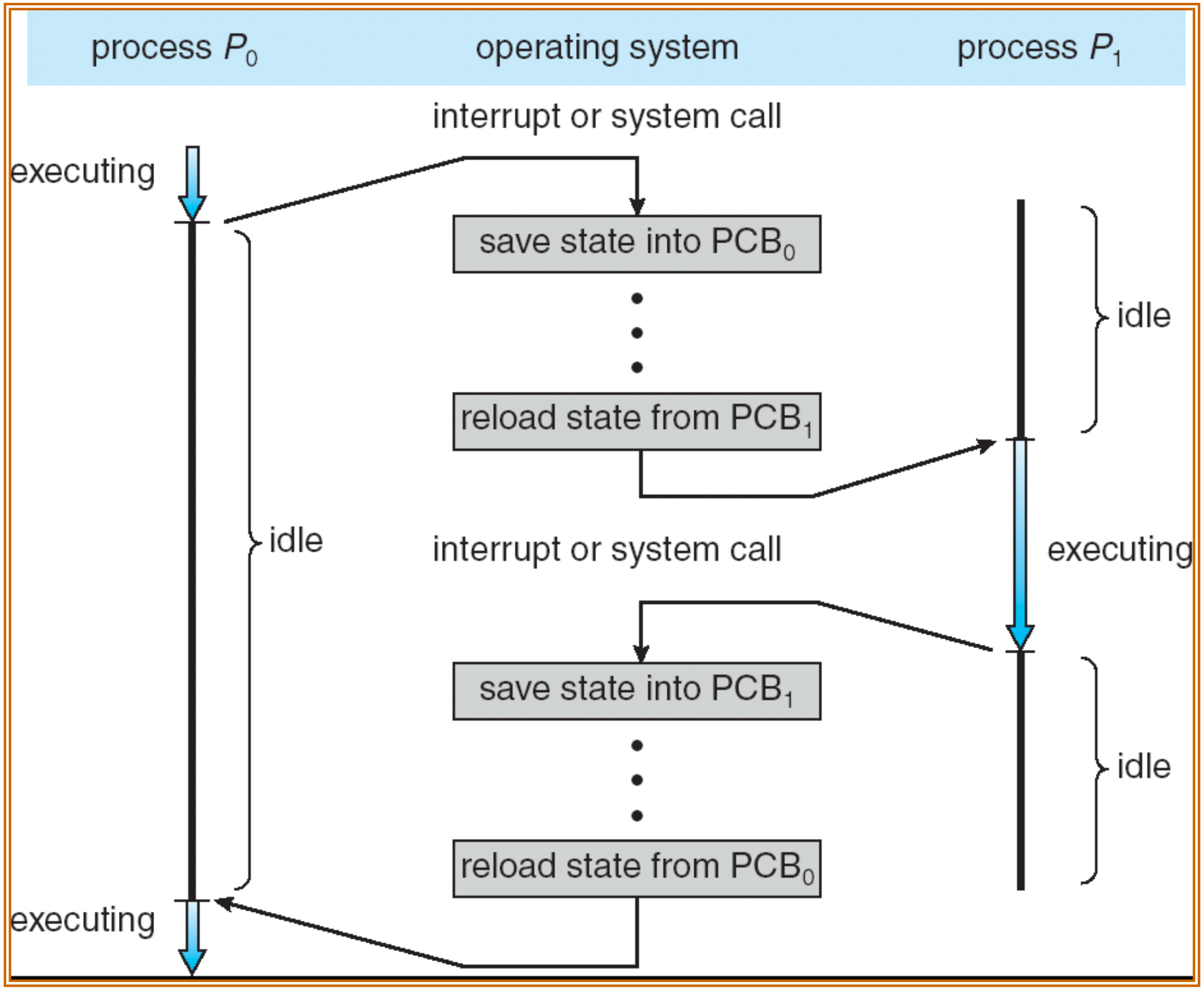
- Snapshot of each process in its PCB
  - Only one thread active at a time per core…
- Give out CPU to different processes
  - **Scheduling**
  - **Policy Decision**
- Give out non-CPU resources
  - Memory/IO
  - Another **policy decision**



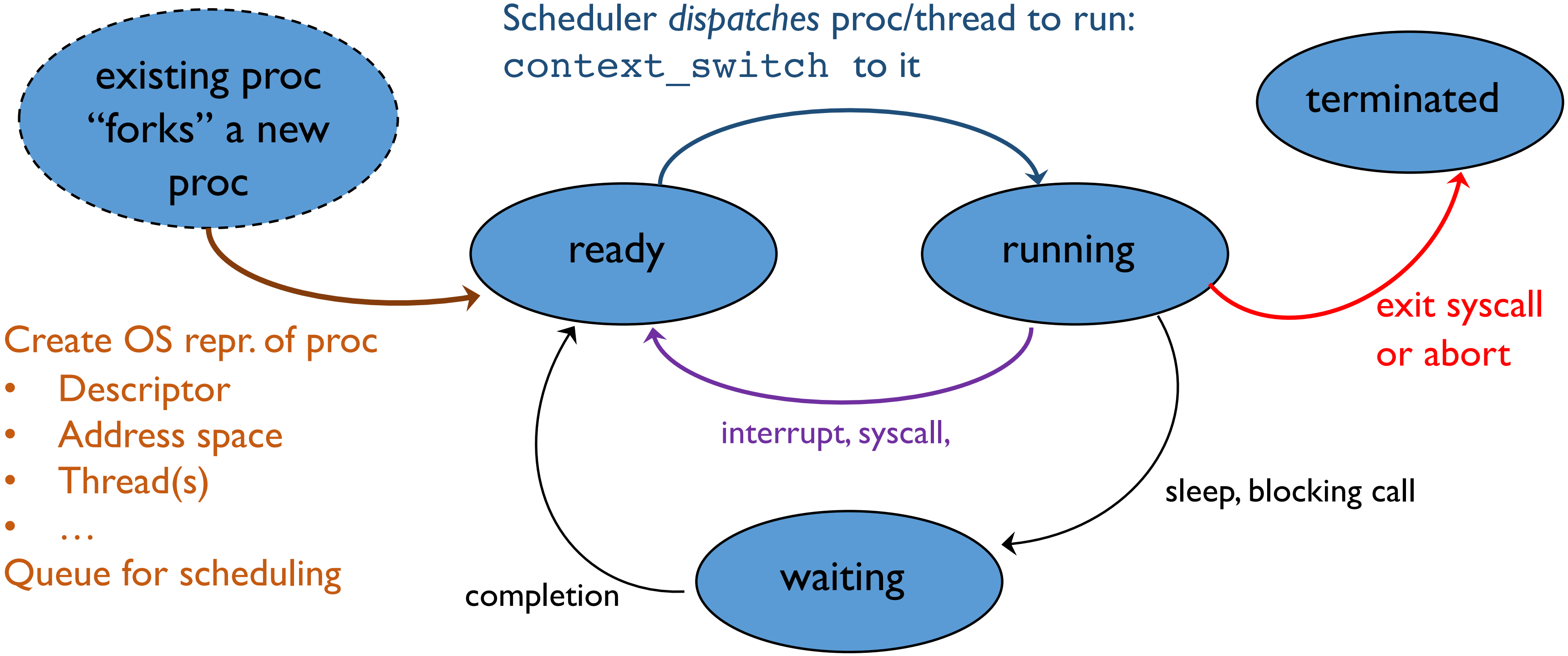
Process Control Block

Recall: Context Switch

TCB, Stacks and Register Mgmt



Recall: Lifecycle of a process / thread



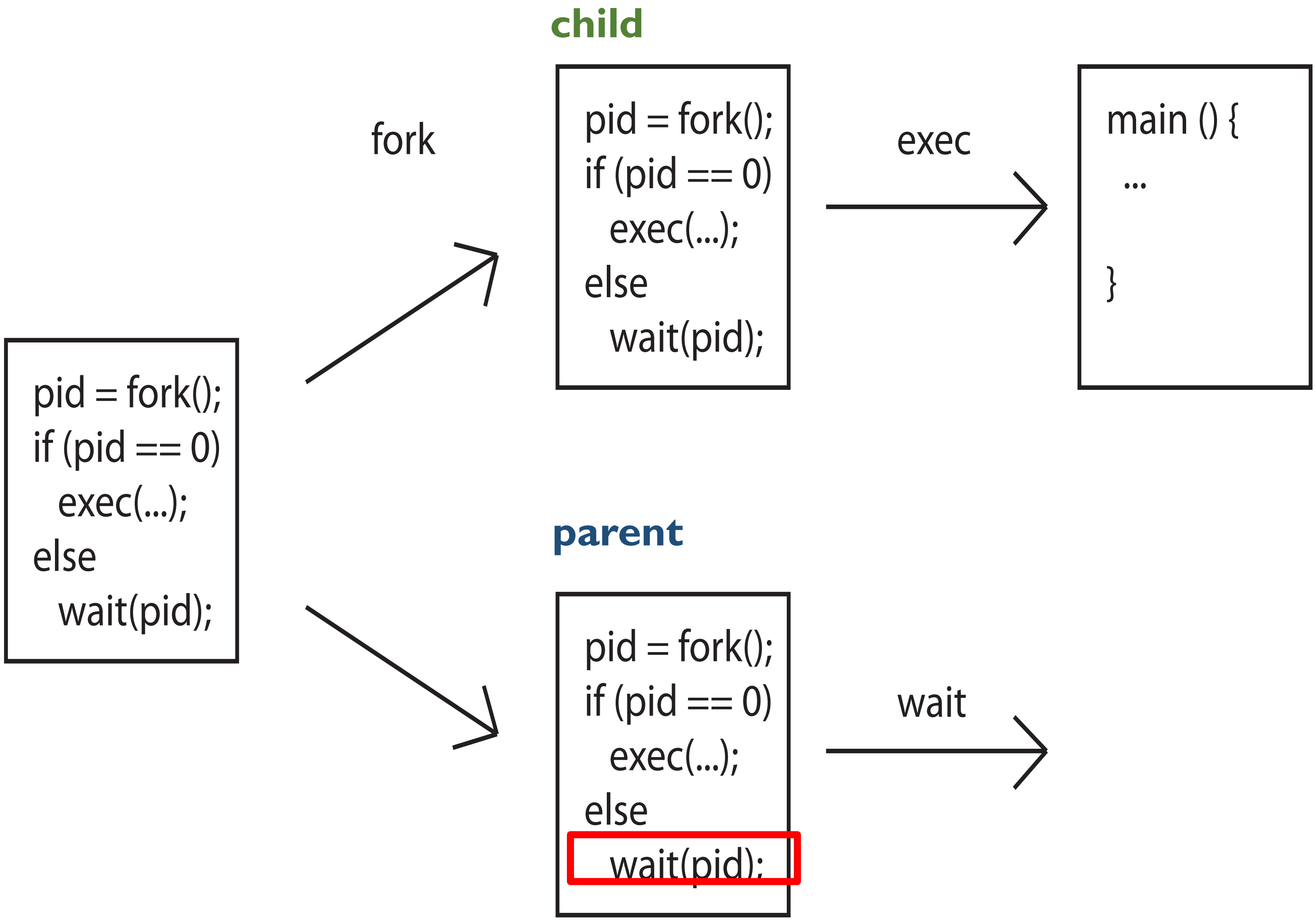
- OS juggles many process/threads using kernel data structures
- Proc's may create other process (fork/exec)
  - All starts with init process at boot

Pintos: process.c

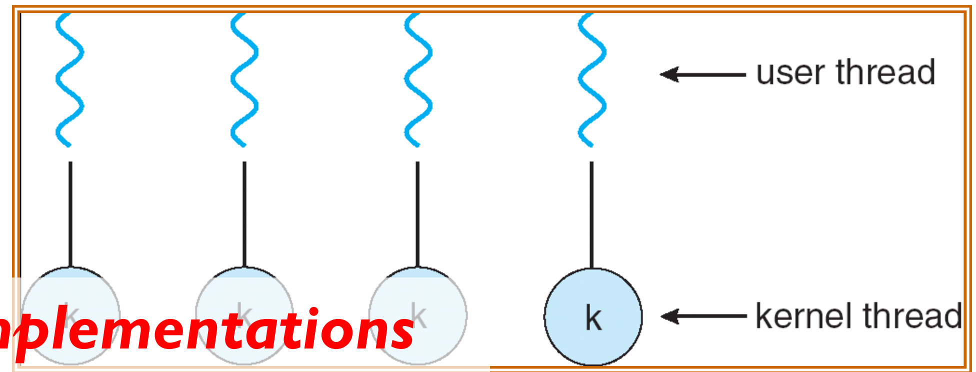
Recall: Process Management

- `exit – terminate a process`
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

Recall: Process Management

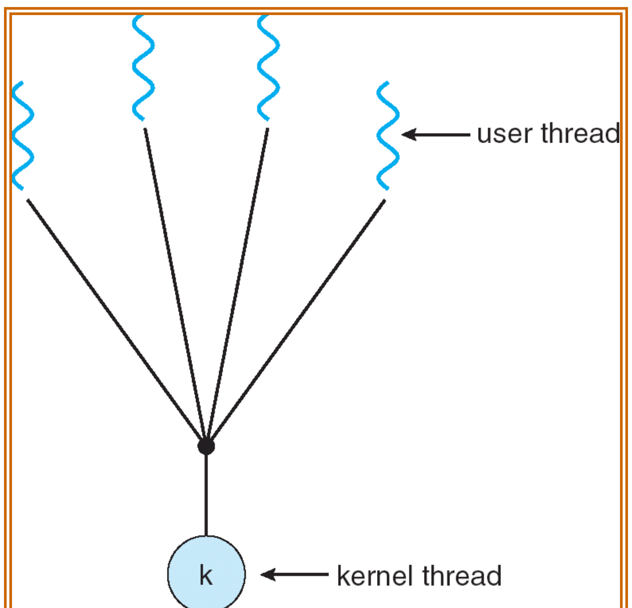


User/OS Threading Models

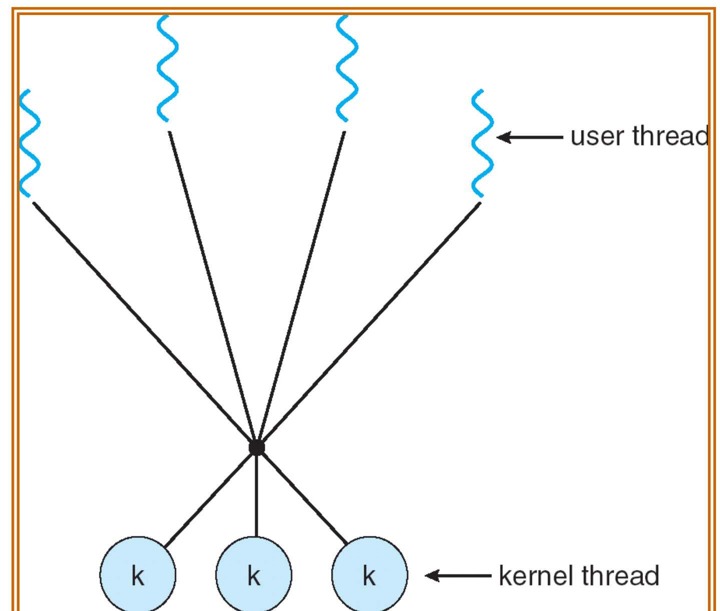


Simple One-to-One Threading Model

Almost all current implementations

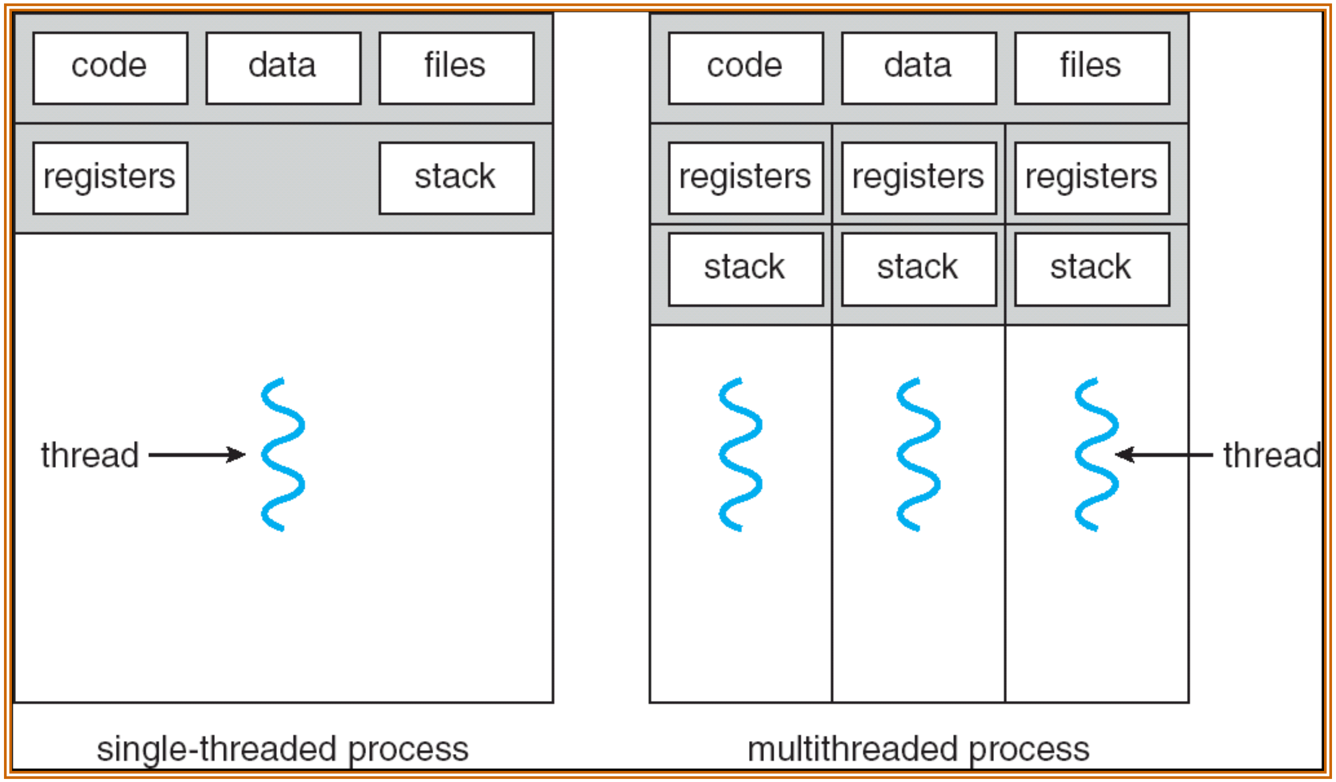


Many-to-One



Many-to-Many

Single vs. Multithreaded Processes



Today

- What, Why, and How of Threads
- Kernel-Supported User Threads
- Coordination among Threads
  - Synchronization
- Implementing Synchronization
- User-level Threads

Definitions

- A *thread* is a single execution sequence that represents a separately schedulable task
- Protection is an orthogonal concept
  - Can have one or many threads per protection domain
  - Single threaded user program: one thread, one protection domain
  - Multi-threaded user program: multiple threads, sharing same data structures, isolated from other user programs
  - Multi-threaded kernel: multiple threads, sharing kernel data structures, capable of using privileged instructions

Threads Motivation

- 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
  - Sequence steps in access or communication

Silly Example for Threads

Imagine the following program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```

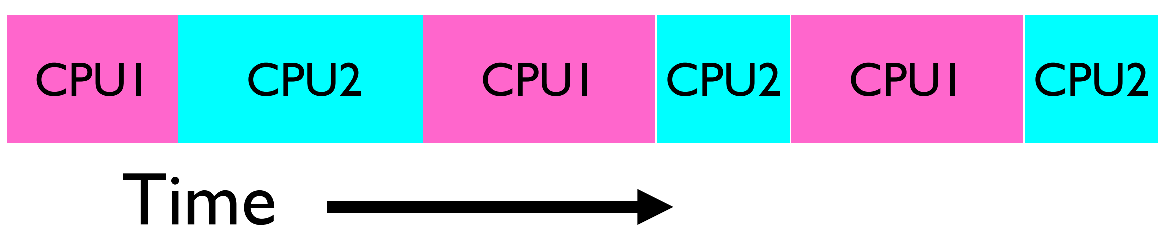
- What is the behavior here?
  - Program would never print out class list
  - Why? `ComputePI` would never finish

Adding Threads

- Version of program with Threads (loose syntax):

```
main() {
    thread_fork(ComputePI, "pi.txt" ));
    thread_fork(PrintClassList, "classlist.txt"));
}
```

- `thread_fork`: Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



More Practical Motivation

Back to Jeff Dean's "Numbers everyone should know"

Handle I/O in separate thread, avoid blocking other progress

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Little Better Example for Threads?

Imagine the following program:

```
main() {
    …
    ReadLargeFile(“pi.txt”);
    RenderUserInterface();
}
```

- What is the behavior here?
  - Still respond to user input
  - While reading file in the background

Voluntarily Giving Up Control

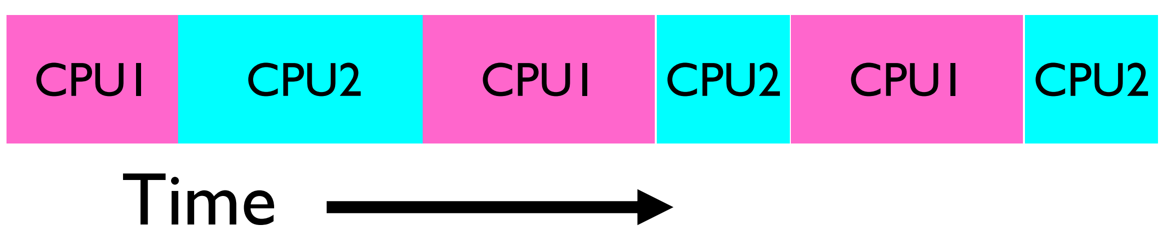
- I/O – e.g. keypress
- Waiting for a signal from another thread
  - Thread makes system call to *wait*
- Thread executes `thread_yield()`
  - Relinquishes CPU but puts calling thread back on ready queue

Adding Threads

- Version of program with Threads (loose syntax):

```
main() {
    thread_fork(ReadLargeFile, “pi.txt” );
    thread_fork(RenderUserInterface, “classlist.txt”);
}
```

- `thread_fork`: Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs

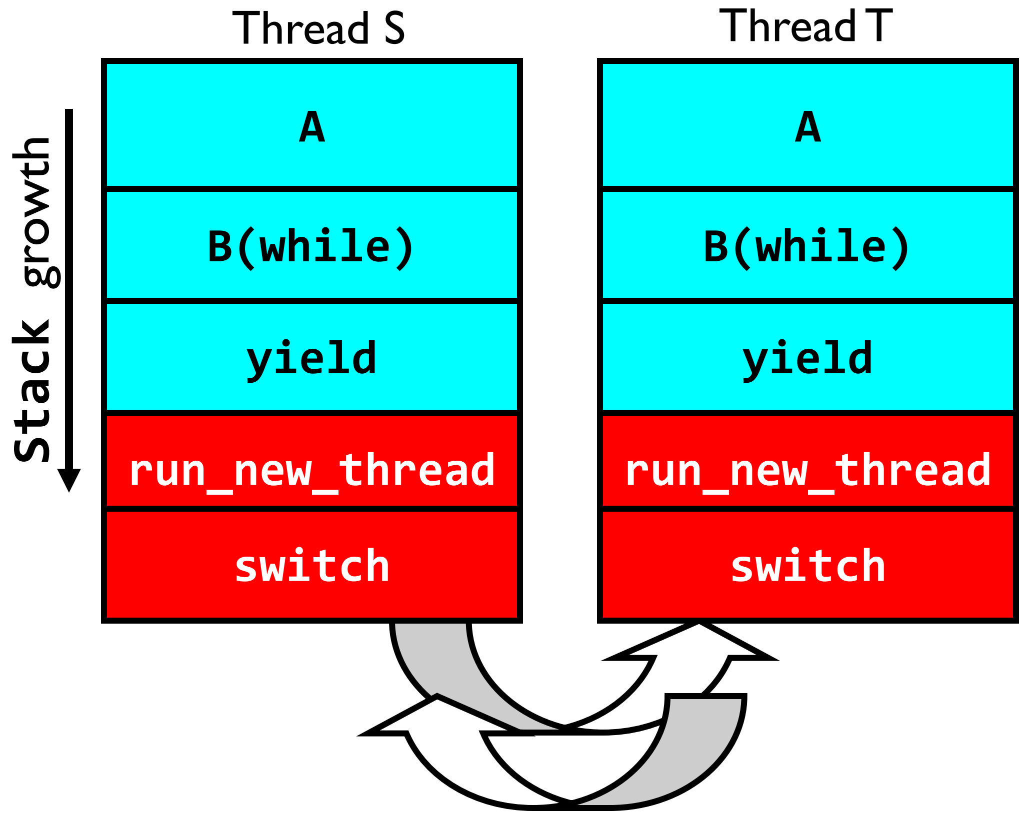


Switching Threads

- Consider the following code blocks:

```
func A() {
  B();
}
func B() {
  while(TRUE) {
    yield();
  }
}
```

- Two threads, S and T, each run A

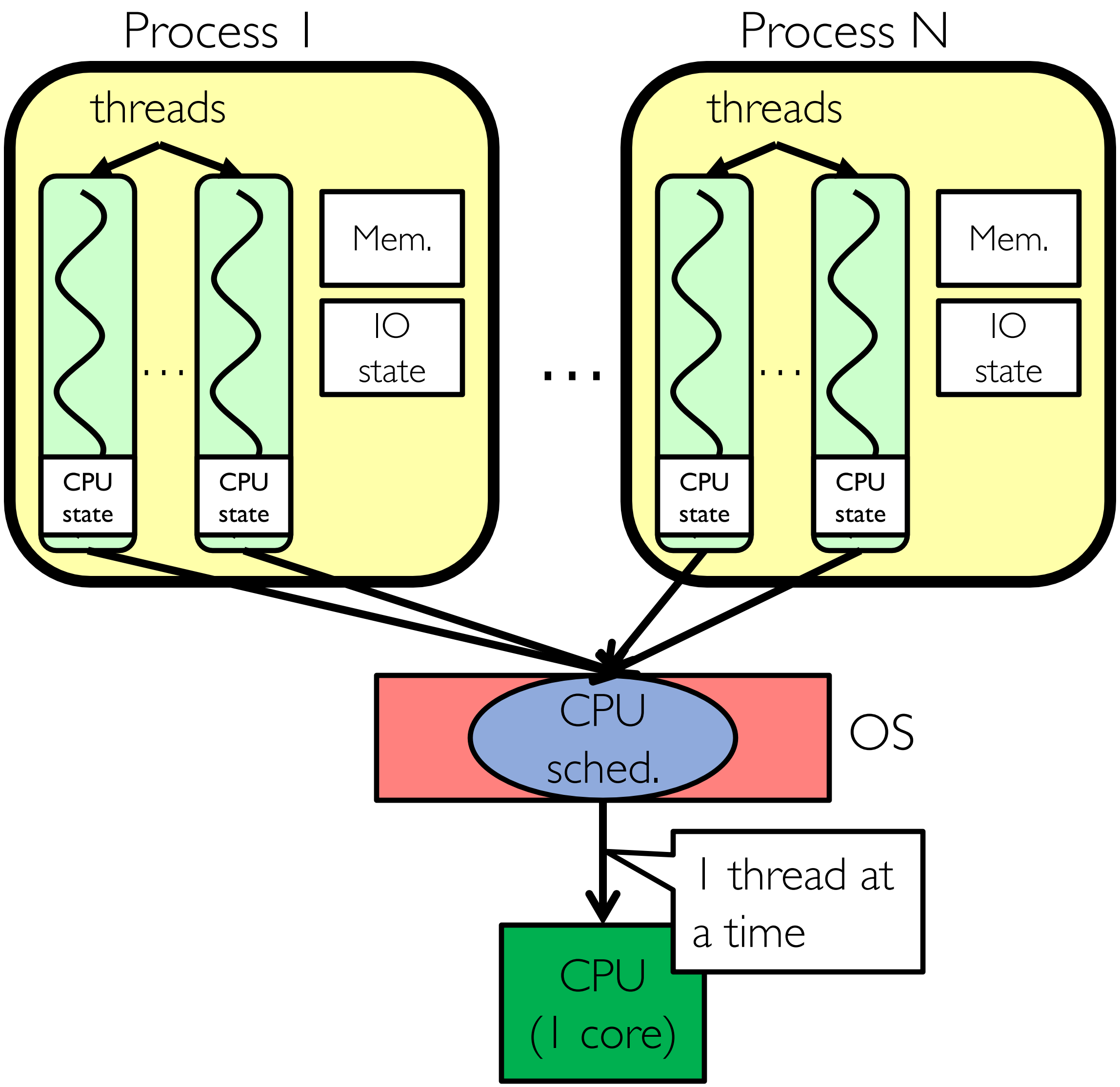


Thread S's `switch` returns to Thread T's (and vice versa)

Aren't we still switching contexts?

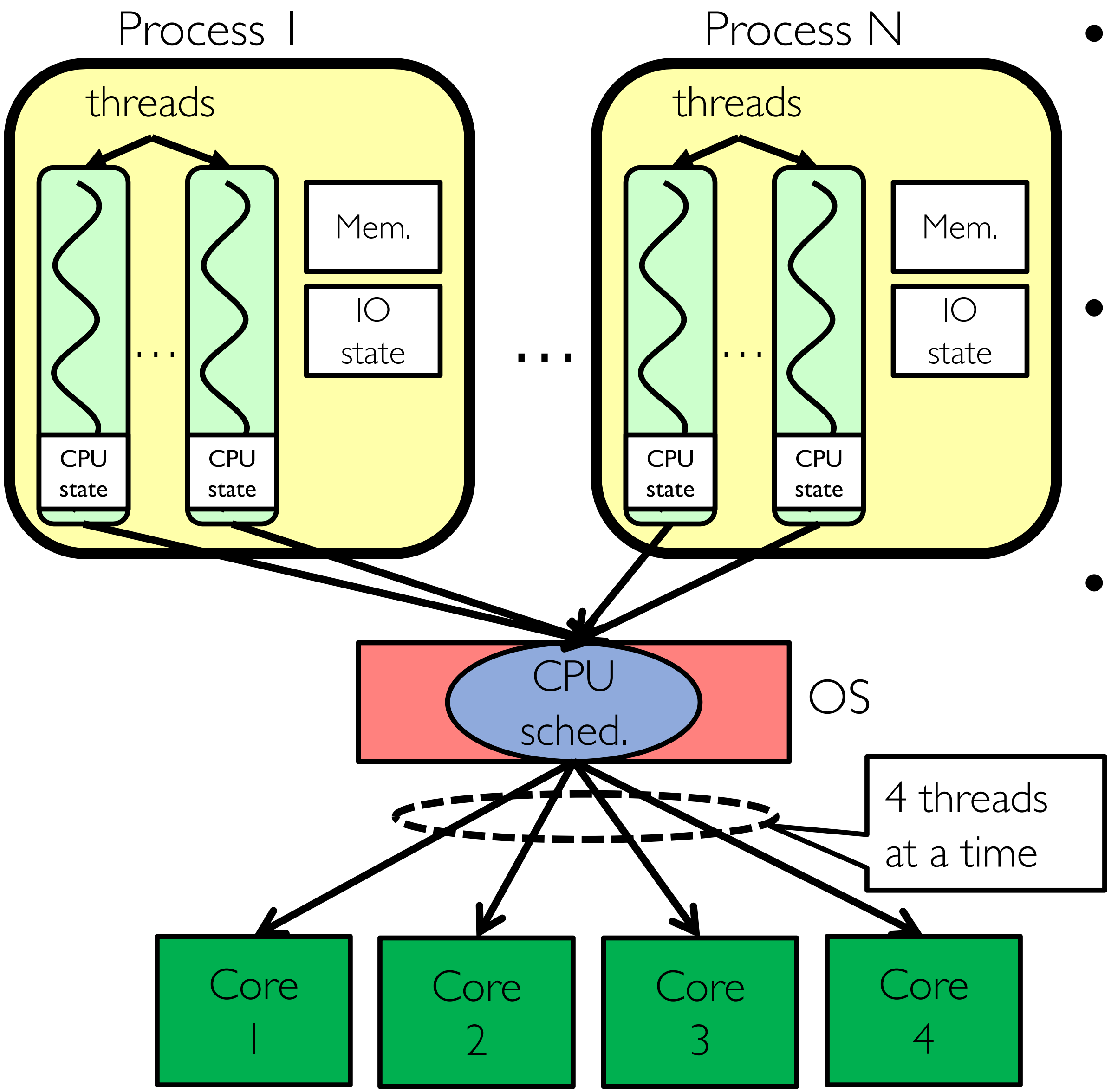
- Yes, but **much cheaper** than switching processes
  - No need to change address space
- Some numbers from Linux:
  - Frequency of context switch: 10-100ms
  - Switching between processes: 3-4 μsec.
  - Switching between threads: 100 ns

Processes vs. Threads



- Switch overhead:
  - Same process: **low**
  - Different proc.: **high**
- Protection
  - Same proc: **low**
  - Different proc: **high**
- Sharing overhead
  - Same proc: **low**
  - Different proc: **high**

Processes vs. Threads



- Switch overhead:
  - Same process: **low**
  - Different proc.: **high**
- Protection
  - Same proc: **low**
  - Different proc: **high**
- Sharing overhead
  - Same proc: **low**
  - Different proc: **high**

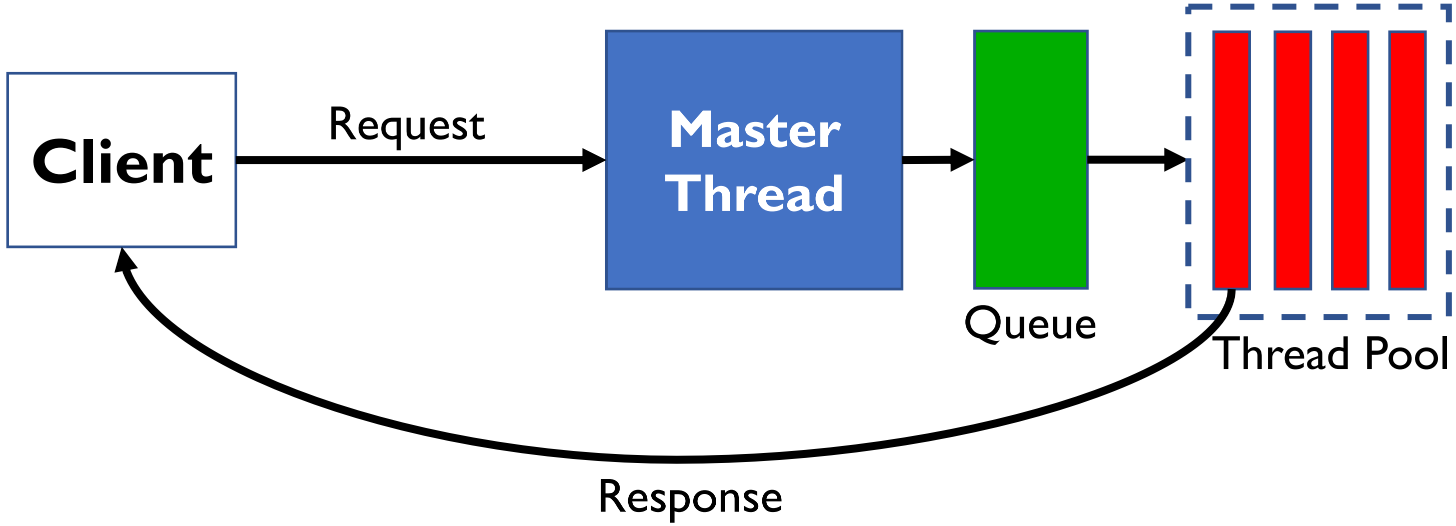
Example: Multithreaded Server

```
serverLoop() {
  connection = AcceptNewConnection();
  (thread_)fork(ServiceWebPage, connection);
}
```

- One process/thread per connection, many concurrent connections
- Process (isolation) vs Thread (performance)
- How fast is creating threads?
  - Better than `fork()`, but still overhead
- Problem: What if we get a lot of requests?
  - Might run out of memory (thread stacks)
  - Schedulers usually have trouble with too many threads

Web Server: Thread Pools

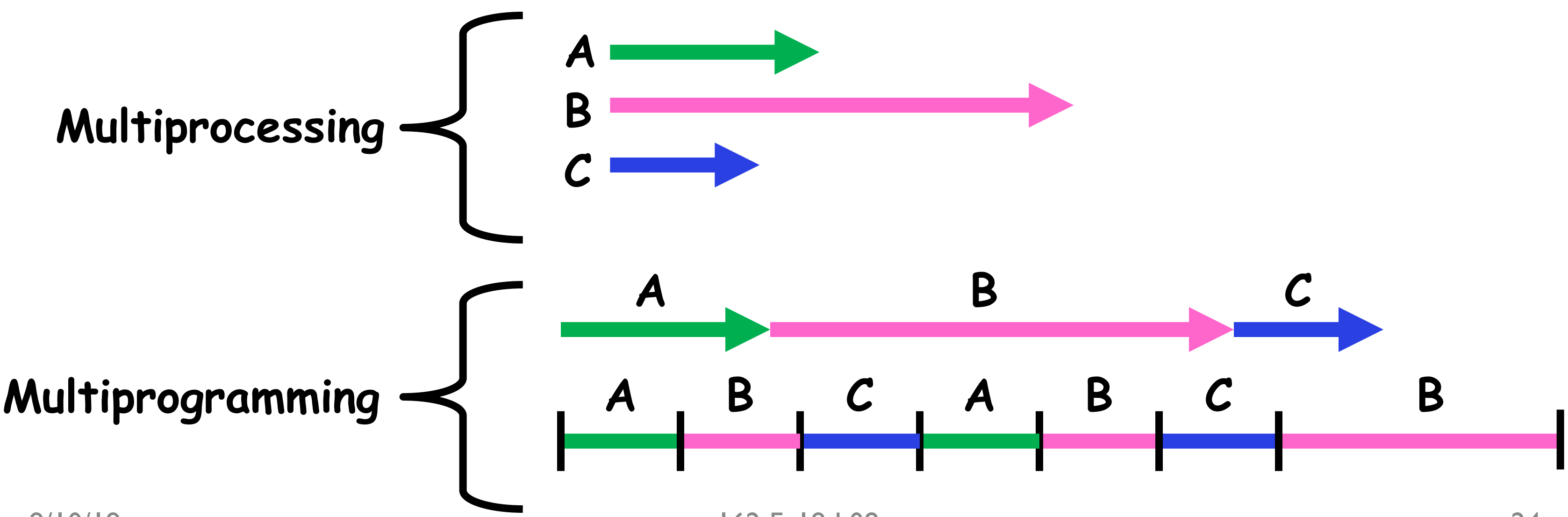
- **Bounded** pool of worker threads
  - Allocated in **advance:** no thread creation overhead
  - **Queue** of pending requests
  - **Limited number** of requests in progress



Multiprocessing vs Multiprogramming

- Multiprocessing: Multiple cores
- Multiprogramming: Multiple Jobs/Processes
- Multithreading: Multiple threads/processes

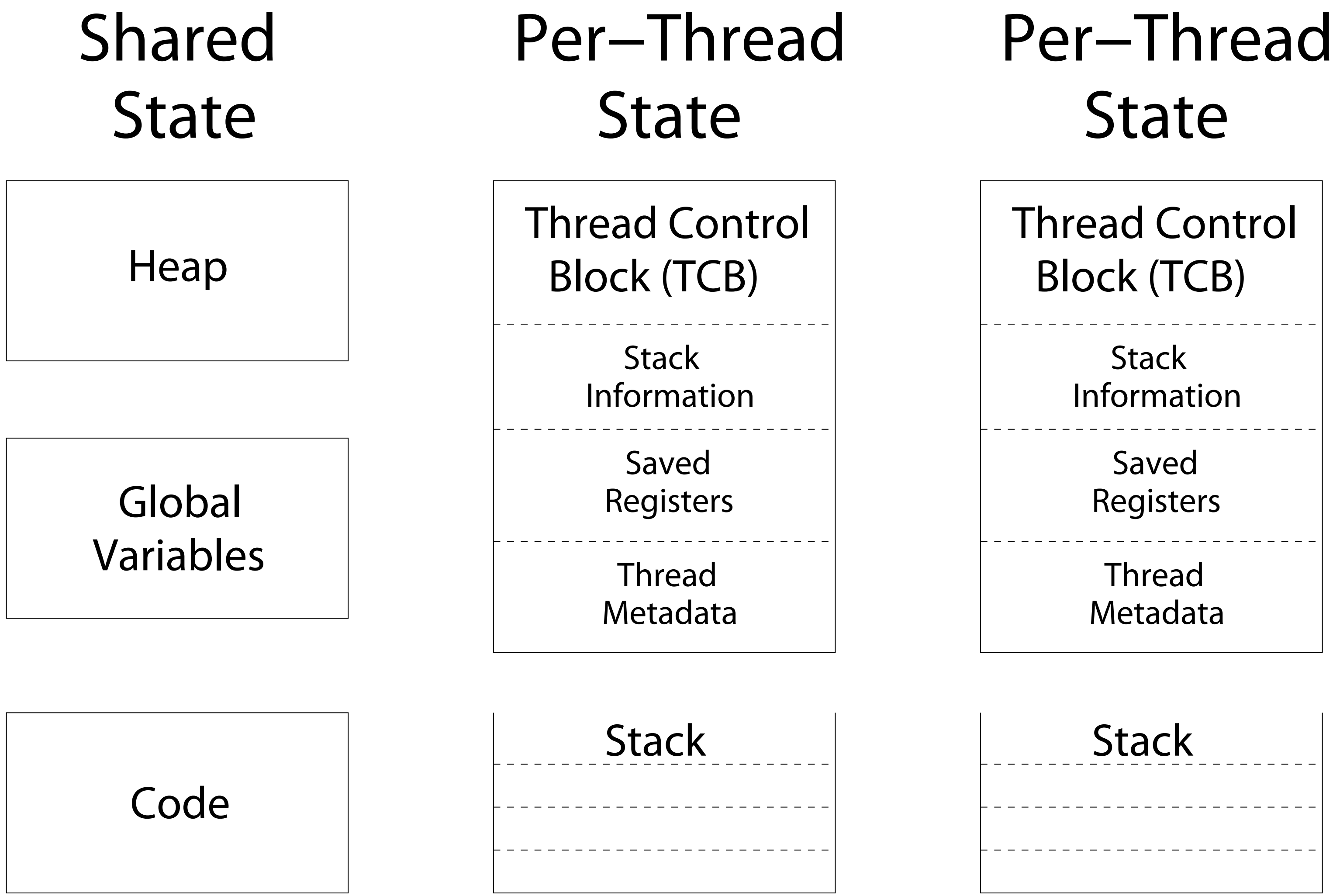
- What does it mean to run two threads concurrently?
  - Scheduler is free to run threads in any order and interleaving



Thread vs. Process State

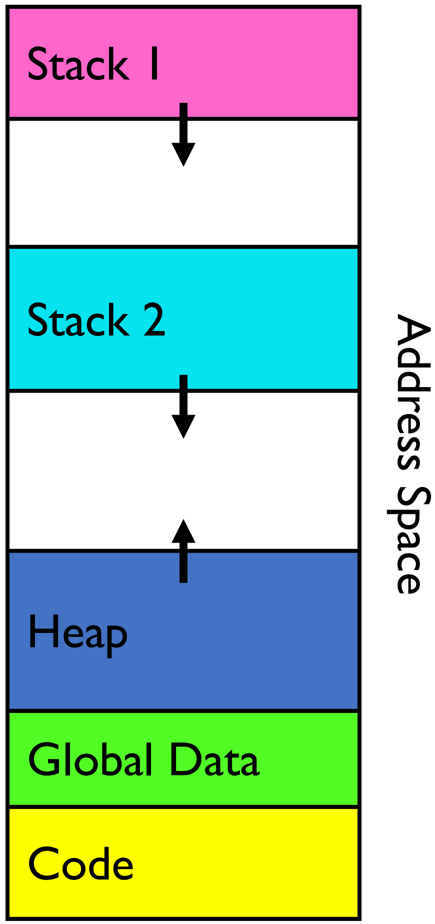
- Process-wide state:
  - Memory contents (global variables, heap)
  - I/O bookkeeping
- Thread-”local" state:
  - CPU registers including program counter
  - Execution stack
  - Kept in **Thread Control Block**

Shared vs. Per-Thread State

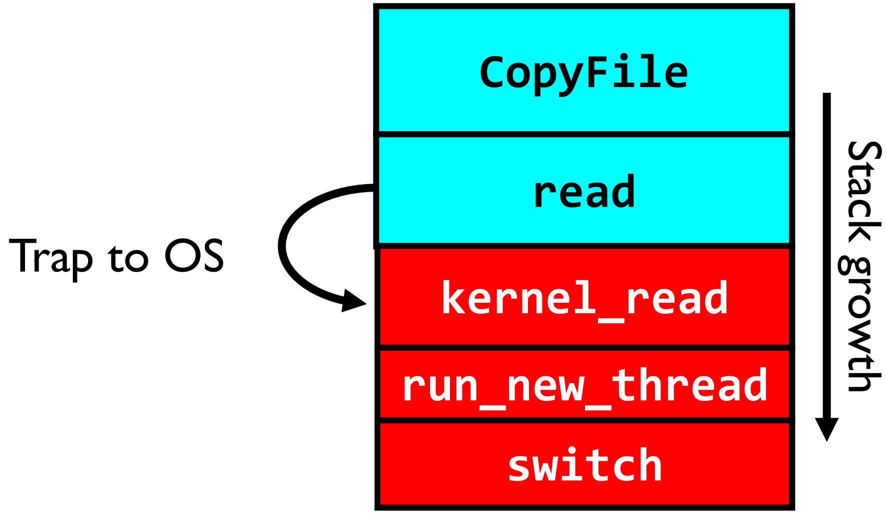


Memory Footprint: Two Threads

- Two sets of CPU registers
- Two sets of Stacks
- Issues:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?
- User threads need 'proper' stacks
  - System threads may be very constrained



Yield is covered, what about I/O?



- User code invokes syscall
- IO operation initiated (more later)
- Run new thread, switch
- Really, same thing as before
  - Just put the thread on a different queue

Preempting a Thread

- What happens if thread never does any I/O, never waits, and never yields control?
  - Must find way that dispatcher can regain control!

- **Interrupts**: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every some milliseconds

- Interrupt is a hardware-invoked mode switch
  - Handled immediately, no scheduling required

Example: Network Interrupt

External Interrupt

```
    ...
add     $r1,$r2,$r3
subi    $r4,$r1,#4
slli    $r4,$r4,#2
    ...
```

Pipeline Flush

```
    ...
lw  $r2,0($r4)
lw  $r3,4($r4)
add $r2,$r2,$r3
sw  8($r4),$r2
        ...
```

PC saved
Disable All Ints
Kernel Mode

```
Raise priority
     (set mask)
Save registers
Reenable Ints
Dispatch to Handler
     ...
Transfer Network Packet
     from hardware
to Kernel Buffers
     ...
Restore registers
Clear current Int
Disable All Ints
Restore priority
     (clear Mask)
RTI
```

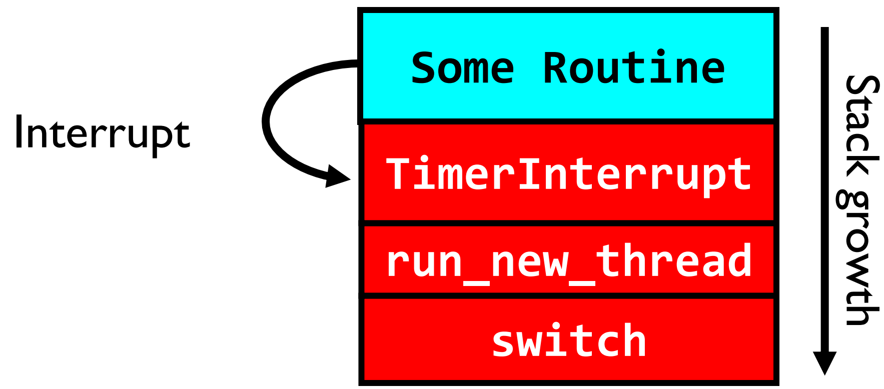
"Interrupt Handler"

Restore PC
Enable all Ints
User Mode

- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

Switching Threads from Interrupts

- Prevent thread from running forever with **timer interrupt**



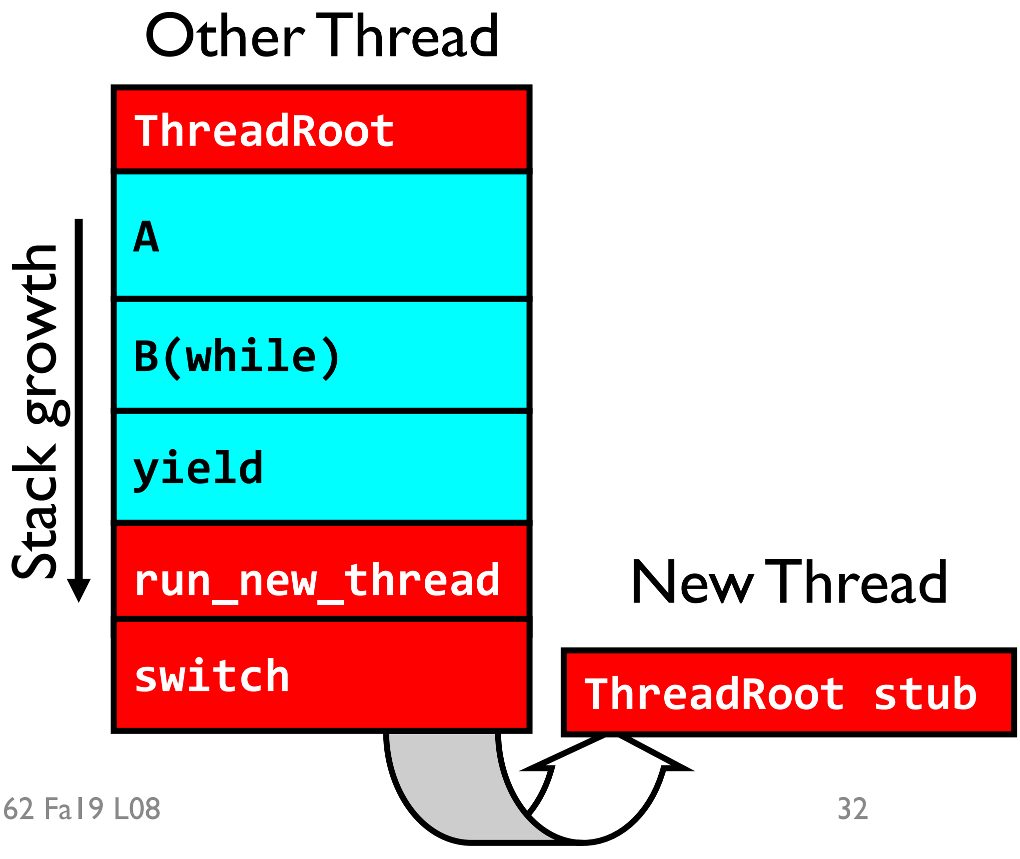

```
TimerInterrupt() {
   DoPeriodicHouseKeeping();
   run_new_thread();
}
```

- Same thing from IO interrupts
  - Example: immediately start process waiting for keypress

How does a thread get started?

- Can't call `switch()` without starting a thread
- How do we make a ***new*** thread?

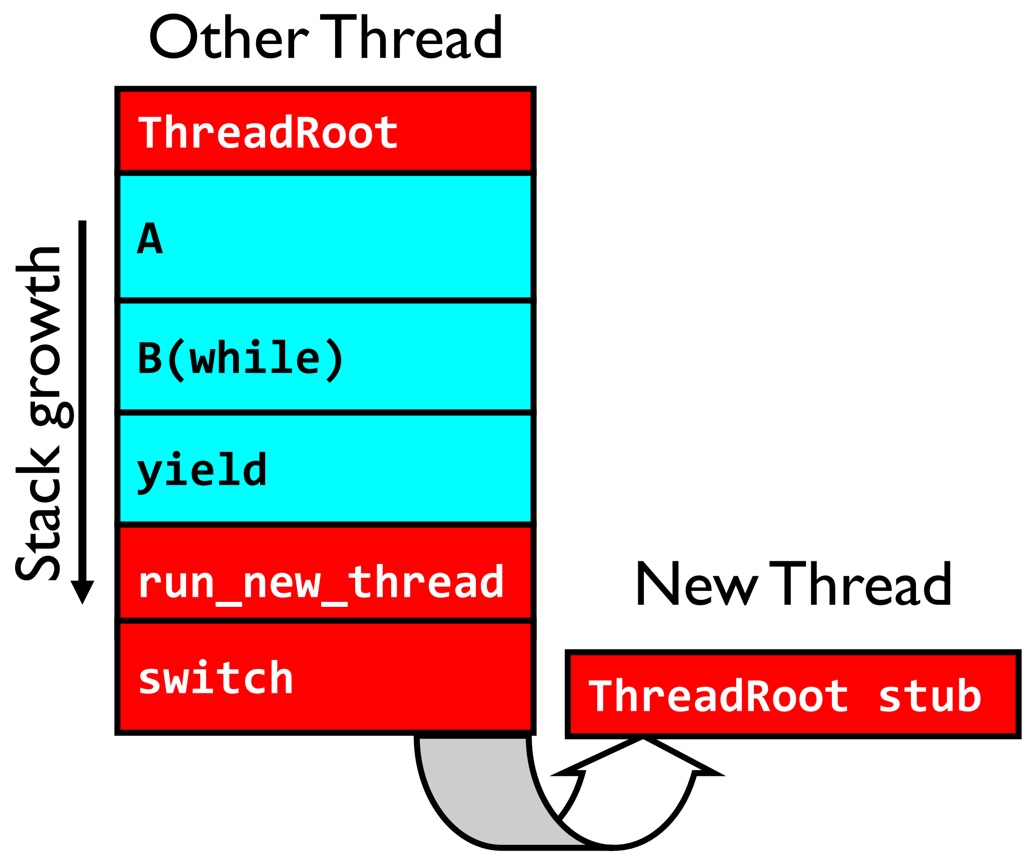
```
SetupNewThread(tNew) {
  …
  TCB[tNew].regs.sp =
    newStack;
  TCB[tNew].regs.retpc =
    &ThreadRoot;
}
```



How does a thread get started?

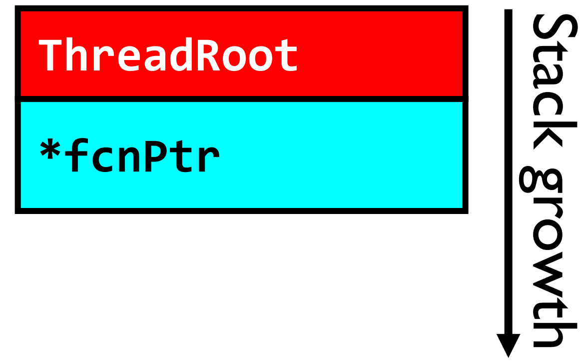
- So when does the new thread really start executing?

`run_new_thread` selects this thread's TCB, "returns" into beginning of **`ThreadRoot`**



Bootstrapping Threads: ThreadRoot

```
ThreadRoot() {
  DoStartupHousekeeping();
  UserModeSwitch(); /* enter user mode */
  call fcnPtr(fcnArgPtr);
  ThreadFinish();
}
```

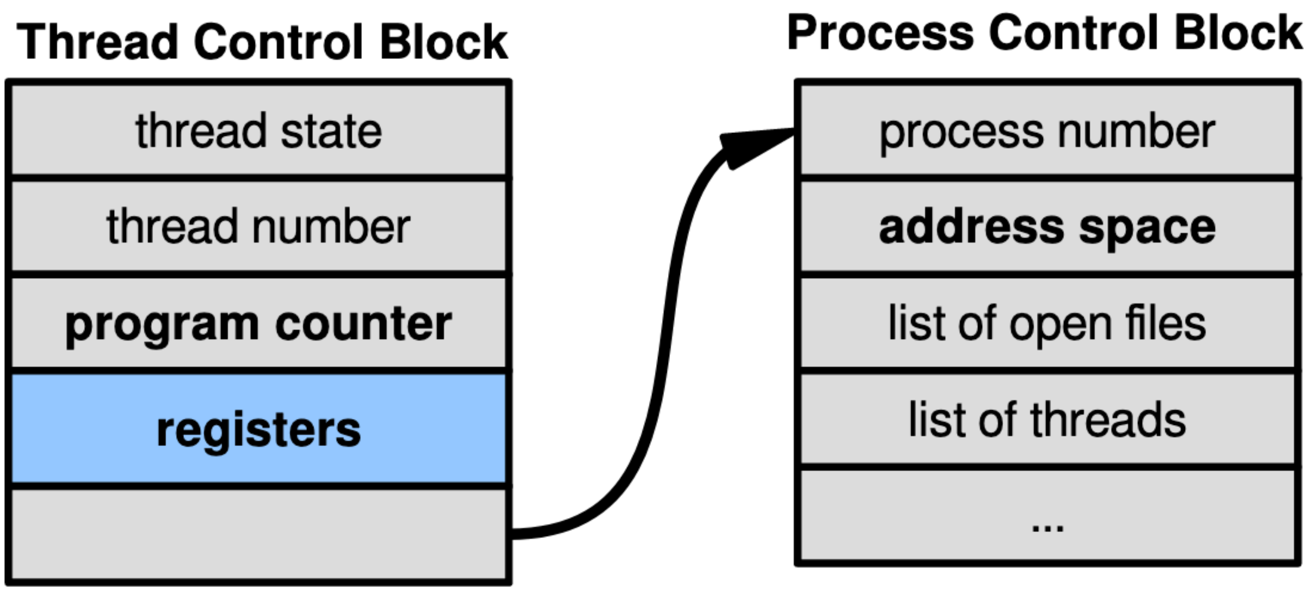


Running Stack

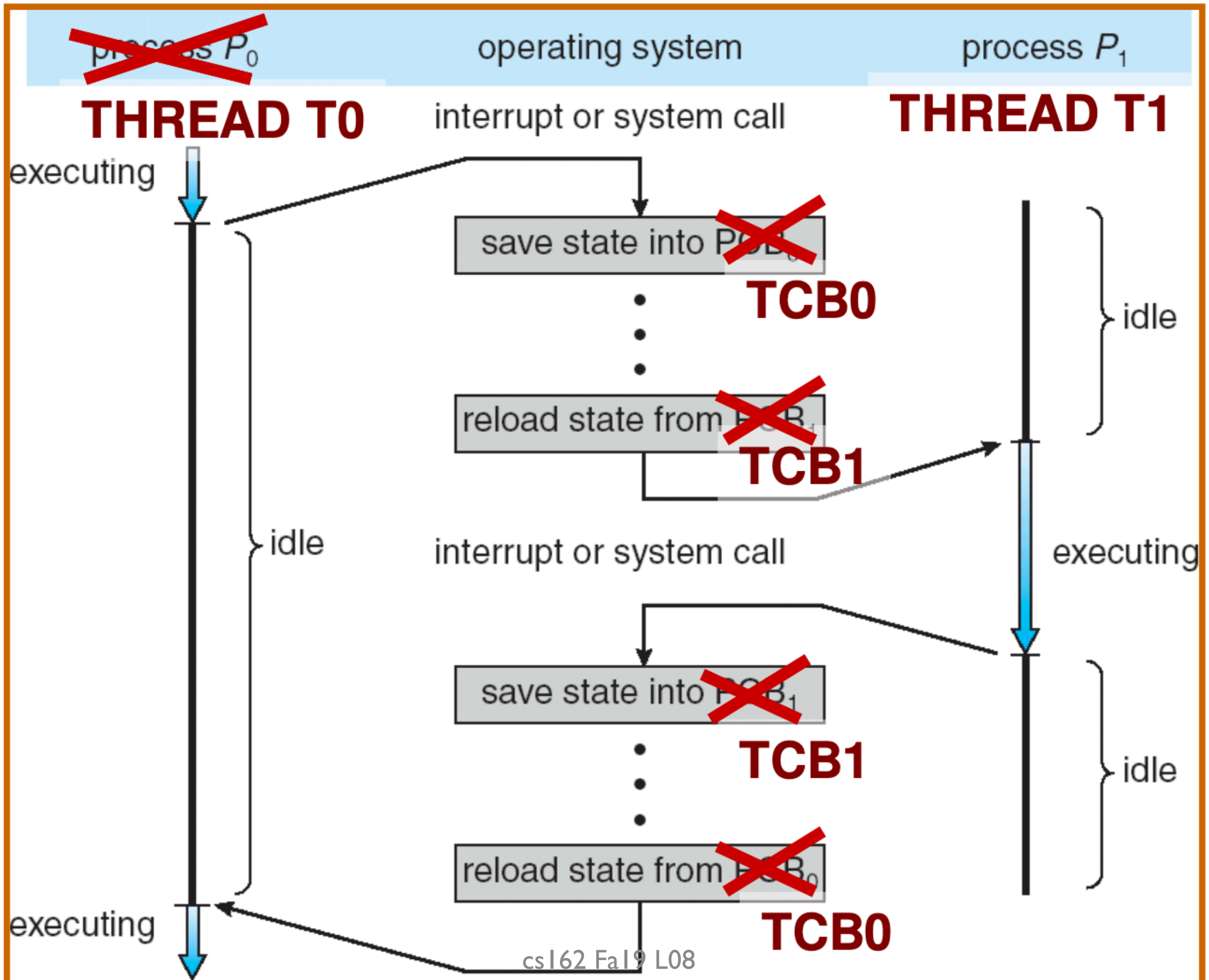
- Stack will grow and shrink with execution of thread
- **`ThreadRoot()`** never returns
  - **`ThreadFinish()`** destroys thread, invokes scheduler

Kernel-Supported Threads

- Each thread has a **thread control block**
  - CPU registers, including PC, pointer to stack
  - Scheduling info: priority, etc.
  - Pointer to **Process control block**
- OS scheduler uses TCBs, not PCBs



Kernel-Supported User Threads



User-level Multithreading: *pthreads*

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);`
  - thread is created executing *start_routine* with *arg* as its sole argument. (return is implicit call to pthread_exit)
- `void pthread_exit(void *value_ptr);`
  - terminates and makes *value_ptr* available to any successful join
- `int pthread_join(pthread_t thread, void **value_ptr);`
  - suspends execution of the calling thread until the target *thread* terminates.
  - On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in the location referenced by *value_ptr*.

man pthread

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

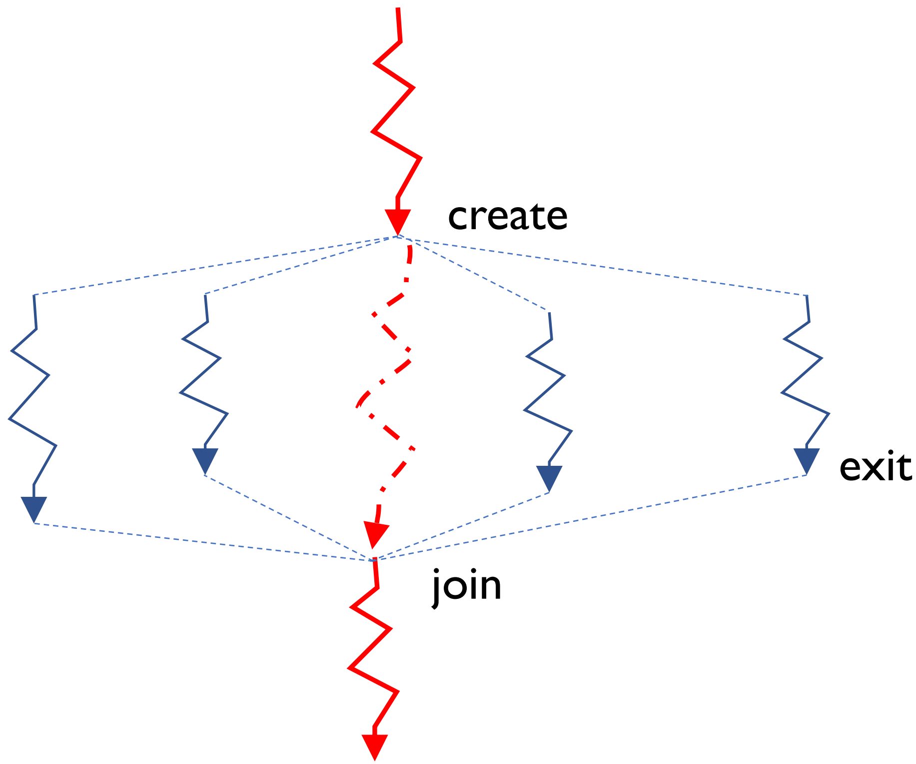
Little Example

```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);           /* last thing in the main thread */
}
```

How to tell if something is done?
Really done?
OK to reclaim its resources?

Fork-Join Pattern



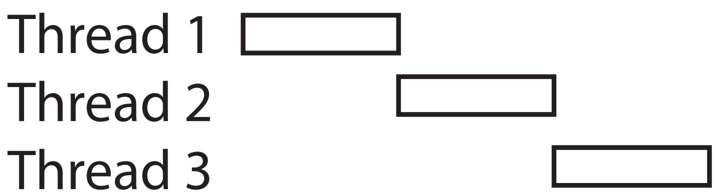
- Main thread *creates* (forks) collection of sub-threads passing them args to work on, *joins* with them, collecting results.

Interleaving & Nondeterminism

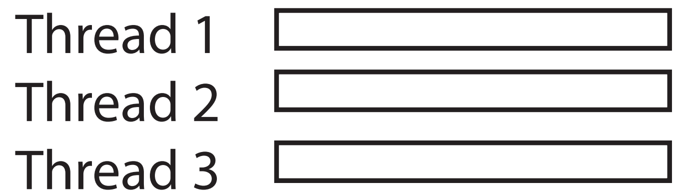
Programmer vs. Processor View

Programmer's View	Possible Execution #1
.	.
.	.
.	.
x = x + 1;	x = x + 1;
y = y + x;	y = y + x;
z = x +5y;	z = x + 5y;
.	.
.	.
.	.

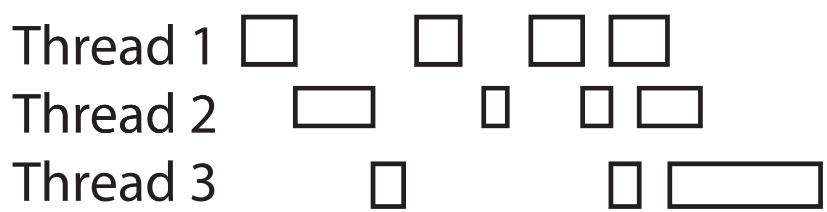
Possible Executions



a) One execution



b) Another execution

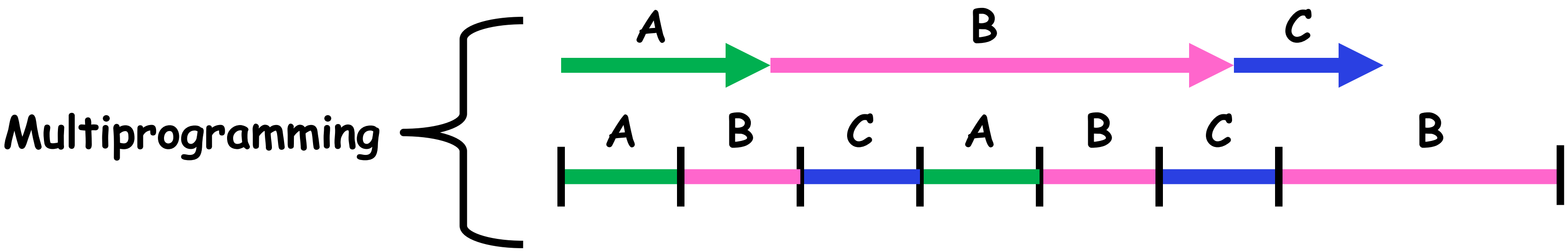


c) Another execution

Correctness with Concurrent Threads

- Non-determinism:
  - Scheduler can run threads in **any order**
  - Scheduler can switch threads **at any time**
  - This can make testing very difficult
- *Independent Threads*
  - No state shared with other threads
  - Deterministic, reproducible conditions
- *Cooperating Threads*
  - Shared state between multiple threads
- **Goal: Correctness by Design**

Remember: Multiprogramming



- Scheduler can run threads in any order
- And with multiple cores:
  - Even more interleaving
  - **Could truly be running at the same time**

Race Conditions

- What are the possible values of **`x`** below?
- Initially **`x = y = 0;`**

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

- Must be **`1`**. Thread B cannot interfere.

Race Conditions

- What are the possible values of **`x`** below?
- Initially **`x = y = 0;`**

<u>Thread A</u>

```
x = y + 1;
```

<u>Thread B</u>

```
y = 2;
y = y * 2;
```

- 1 or 3 or 5 (non-deterministic)
- Race Condition: Thread A races against Thread B

Atomic Operations

- Definition: **An operation that runs to completion or not at all**
  - Need some to allow threads to work together
- `counter++;`   **// atomic?**
  - x86 has memory-to-memory instructions, but that still doesn't make them atomic
- Some store instructions are not atomic
  - Ex: double-precision floating point store

Real-Life Analogy: Too Much Milk

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

See "Additional Materials" and text…

Break

Relevant Definitions

- **Mutual Exclusion:** Ensuring only one thread does a particular thing at a time (one thread *excludes* the others)

- **Critical Section:** Code exactly one thread can execute at once
  - Result of mutual exclusion

Relevant Definitions

- **Lock:** An object only one thread can hold at a time
  - **Provides** mutual exclusion

- Offers two **atomic** operations:
  - `Lock.Acquire()` – wait until lock is free; then grab
  - `Lock.Release()` – Unlock, wake up waiters

Using Locks

```
MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()
```

But how do we implement this?

First, how do we use it?

Pthreads - mutex

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);


int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Our example

Critical section

```
int common = 162;
pthread_mutex_t common_lock = PTHREAD_MUTEX_INITIALIZER;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  pthread_mutex_lock(&common_lock);
  int my_common = common++;
  pthread_mutex_unlock(&common_lock);

  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid,
         (unsigned long) &common, my_common);
  pthread_exit(NULL);
}
```

Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX (& Pintos)
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P() or down(): atomic operation that waits for semaphore to become positive, then decrements it by 1
  - V() or up(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

Two Important Semaphore Patterns

- **Mutual Exclusion:** (Like lock)
  - Called a "binary semaphore"

```
 initial value of semaphore = 1;
semaphore.down();
 // Critical section goes here
semaphore.up();
```

- **Signaling** other threads, e.g. **`ThreadJoin`**

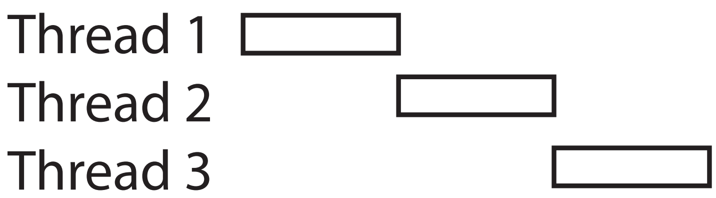
```
Initial value of semaphore = 0
```

```
ThreadFinish {
    semaphore.up();
}
```

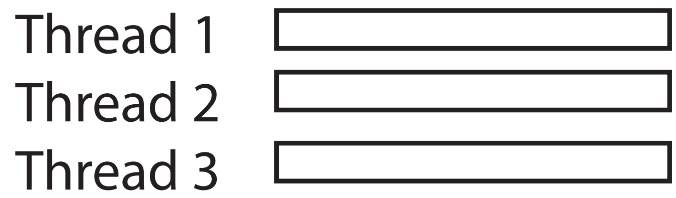
```
ThreadJoin {
    semaphore.down();
}
```

Think of *down* as wait() operation

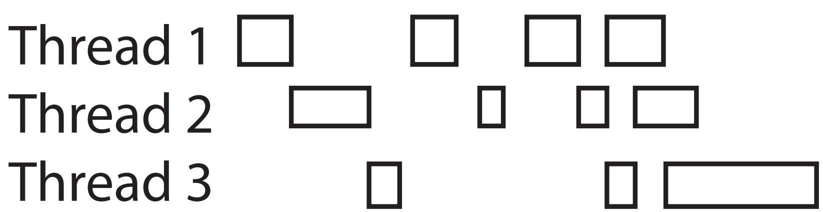
What can we conclude… over "All Possible Executions" ?



a) One execution



b) Another execution



c) Another execution

Implementing Locks: Single Core

- Idea: A context switch can only happen (assuming threads don't yield) if there's an **interrupt**
- "Solution": **Disable interrupts** while holding lock
- x86 has `cli` and `sti` instructions that only operate in system mode (PL=0)
  - Interrupts enabled bit in FLAGS register

Naïve Interrupt Enable/Disable

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

- Problem: can stall the entire system

```
Lock.Acquire()
While (1) {}
```

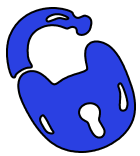
- Problem: What if we want to do I/O?

```
Lock.Acquire()
Read from disk
/* OS waits for (disabled) interrupt)! */
```

Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```



```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

Discussion

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Critical Section

- Note: unlike previous solution, the critical section (inside `Acquire()`) is very short
  - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior
  - Critical interrupts taken in time!

Implementing Locks: Single Core

- Idea: Disable interrupts for **mutual exclusion** on accesses to `value` indicating lock status

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

```
Release() {
  disable interrupts;
  if (anyone waiting) {
    take a thread off queue;
  } else {
    Value = FREE;
}
  enable interrupts;
}
```

Reenabling Interrupts When Waiting

enable interrupts

enable interrupts

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
} else {
    value = BUSY;
  }
  enable interrupts;
}
```

- Before on the queue?
  - Release might not wake up this thread!
- After putting the thread on the queue?
  - Gets woken up, but immediately switches away

Reenabling Interrupts When Waiting

enable interrupts

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

- Best solution: after the current thread suspends
- How?
  - `run_new_thread()` should do it!
  - Part of returning from `switch()`

How to Re-enable After Sleep()?

- In scheduler, since interrupts are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

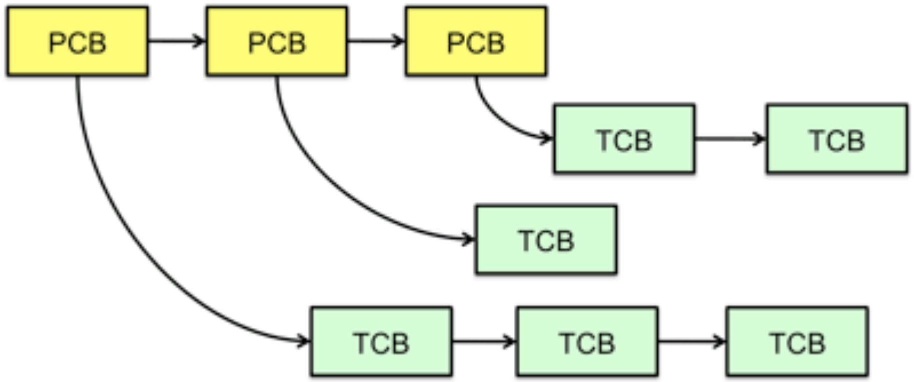
Thread A		Thread B
.		
.		
`disable ints`		
`sleep`	context switch →	
		`sleep return`
		`enable ints`
		.
		.
		.
		`disable int`
	← context switch	`sleep`
`sleep return`		
`enable ints`		
.		
.		

Recall: 61c

- Hardware provides certain atomic operations
  - Swap, Compare&Swap, Test&Set, Fetch&Add, LoadLocked/StoreConditional
  - More on optimized synchronization ops later
- System threads need more than the atomic operation
  - May need to manipulate scheduling queues too
  - Requires combination of HW and SW to do it right
- Pintos implements "semaphores"
  - Builds locks and CVs on top of them

Multithreaded Processes

- PCB may be associated with multiple TCBs:



- Switching threads within a process is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables.

So does the OS schedule processes or threads?

- We've been talking about processes assuming the "old model" -> one thread per process
  - And many textbooks say this as well
- Usually it's really: **threads** (e.g., in Linux)
- More on some of these issues later

- One point to notice: switching threads vs. switching processes incurs different costs:
  - Switch threads: Save/restore registers
  - Switch processes: Change active address space too!
    - Expensive
    - Disrupts caching

User-level threads?

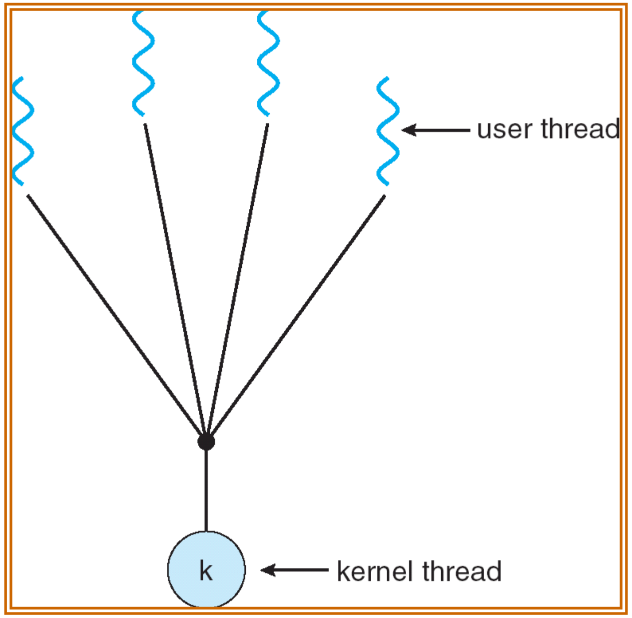
- Can multiple threads be implemented entirely at user level?
- Most other aspects of system virtualize.

Kernel-Supported Threads

- Threads run and block (e.g., on I/O) independently
- One process may have multiple threads waiting on different things
- Two mode switches for every context switch (expensive)
- Create threads with syscalls

- Alternative: multiplex several streams of execution (at user level) on top of a single OS thread
  - E.g., Java, Go, … (and many many user-level threads libraries before it)

User-Mode Threads



- User program contains its own scheduler
- Several user threads per kernel thd.
- User threads may be scheduled non-preemptively
  - Only switch on `yield`
- Context switches cheaper
  - Copy registers and jump (`switch` in userspace)

User-Mode Threads: Problems

- One user-level thread blocks on I/O: they all do
  - Kernel cannot adjust scheduling among threads it doesn't know about
- Multiple Cores?
- Can't completely avoid blocking (syscalls, page fault)
- One Solution: *Scheduler Activations*
  - Have kernel inform user-level scheduler when a thread blocks

- Evolving the contract between OS and application.

Classification

# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, HP-UX, Win NT to 8, Solaris, OS X, Android, iOS

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space

Summary

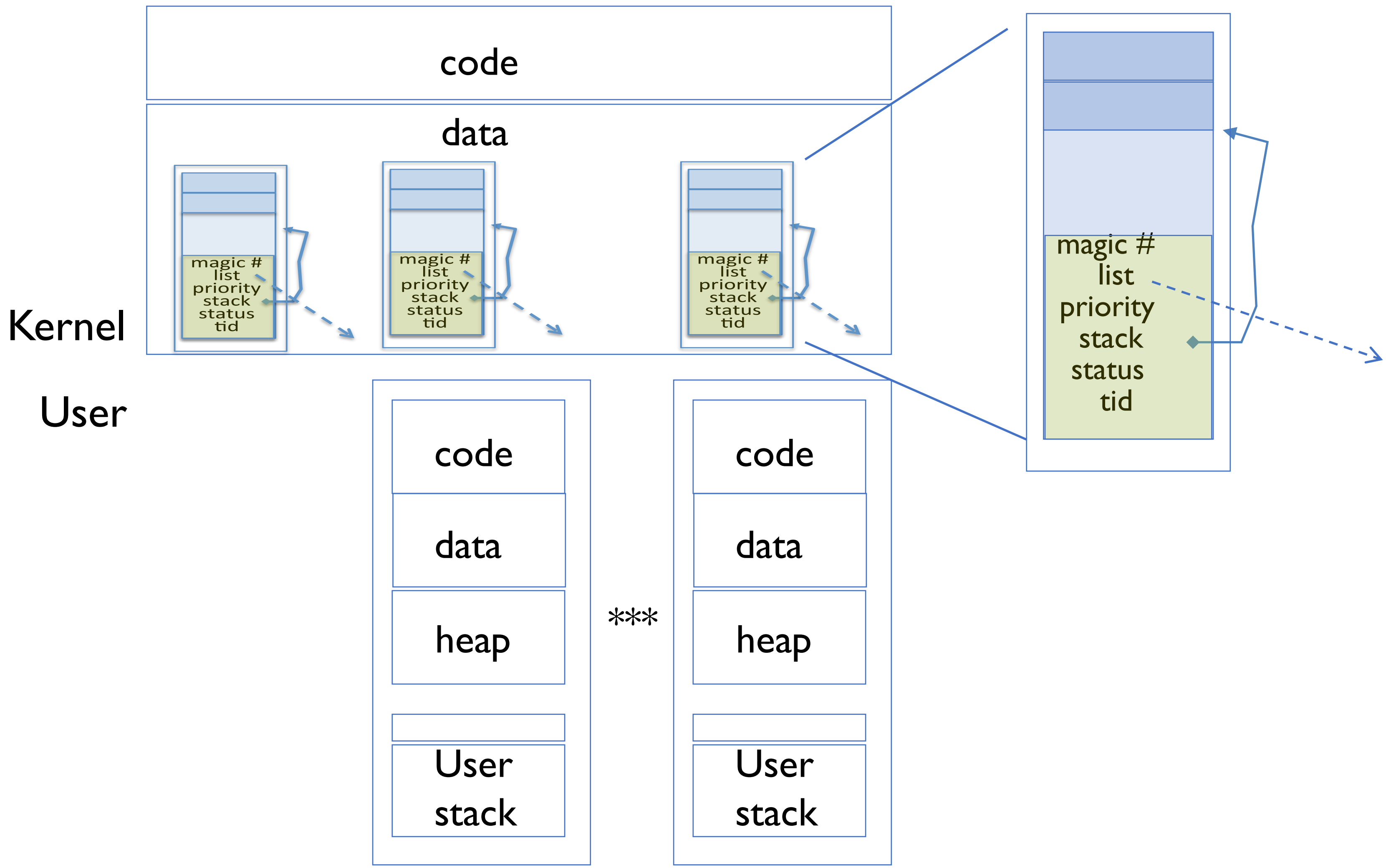
- Process consists of two components
    1. Address Space (Protection)
    2. One or more threads (Concurrency)
- Threads: unit of concurrent execution
    - Useful for parallelism, overlapping computation and IO, organizing sequences of interactions (protocols)
    - Require: multiple stacks per address space
    - Thread switch:
        - Save/Restore registers, "return" from new thread's `switch` routine
    - Challenging to write correct concurrent code:
        - **Arbitrary interleavings**
        - Could access shared resources while in bad state
    - Kernel threads, Kernel-supported User Threads, User-mode Threads
- Synchronization
    - Building block: atomic operations
    - Mutual exclusion (locks) & Signaling (exit->join, semaphore)
- Scheduling: Threads move between queues
    - Synchronization and scheduler deeply interrelated

Additional Materials

Deeper Review: User/Kernel Threads in Pintos

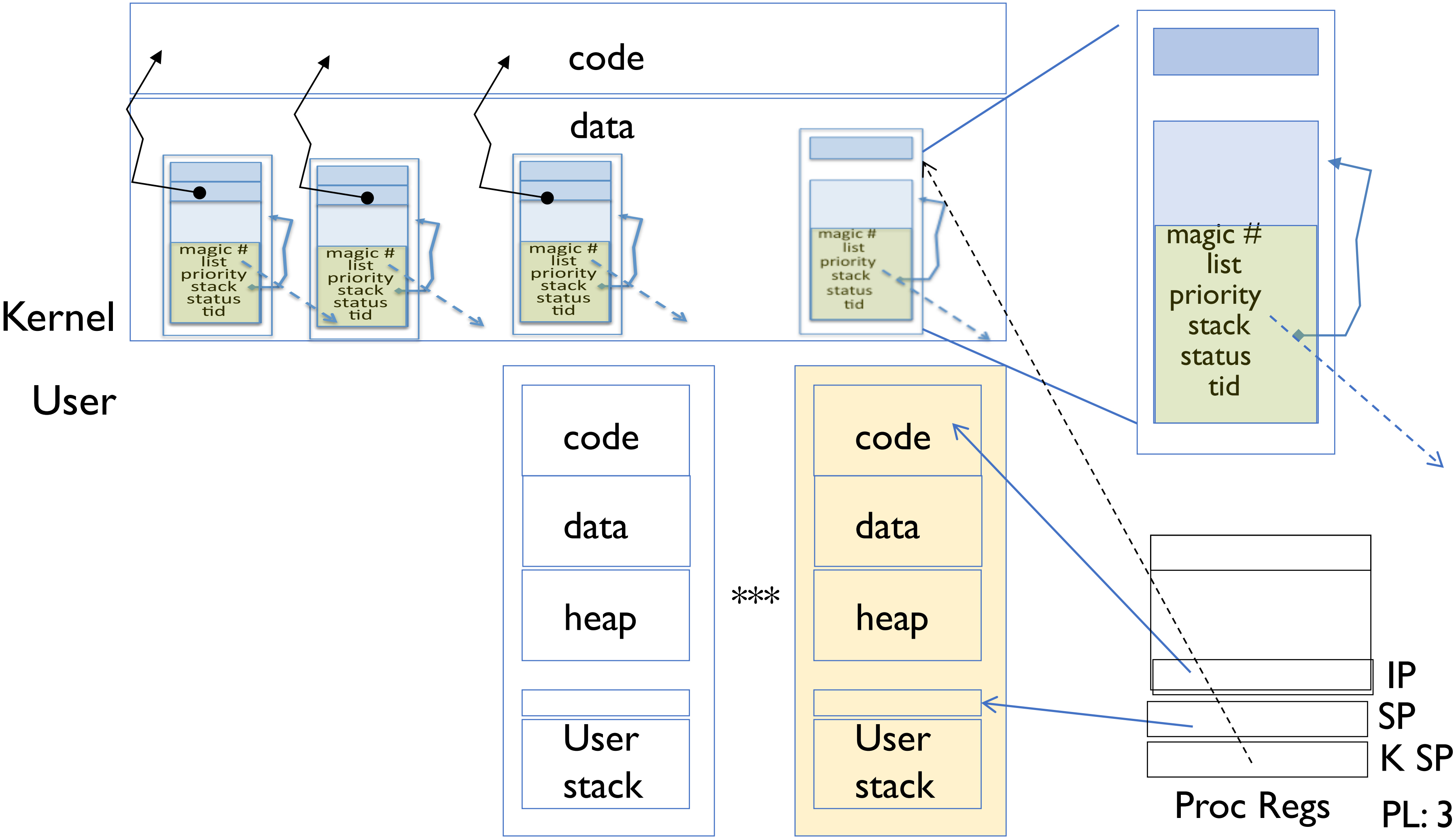
- Now that you're reading the code, let's do a quick picture of what's going on

MT Kernel 1T Process ala Pintos/x86



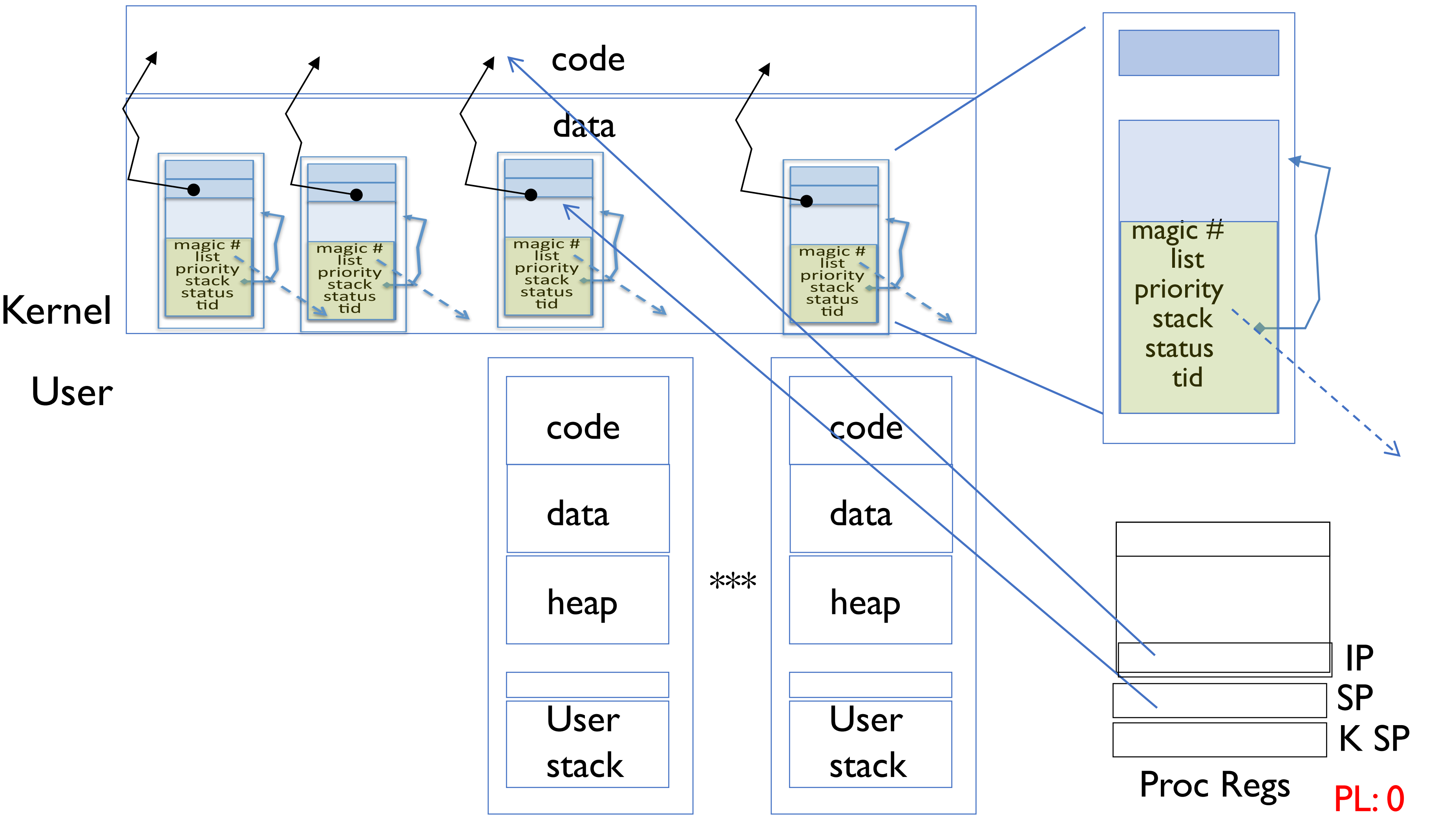
- Each user process/thread associated with a kernel thread, described by a 4kb Page object containing TCB and kernel stack for the kernel thread

In User thread, w/ k-thread waiting



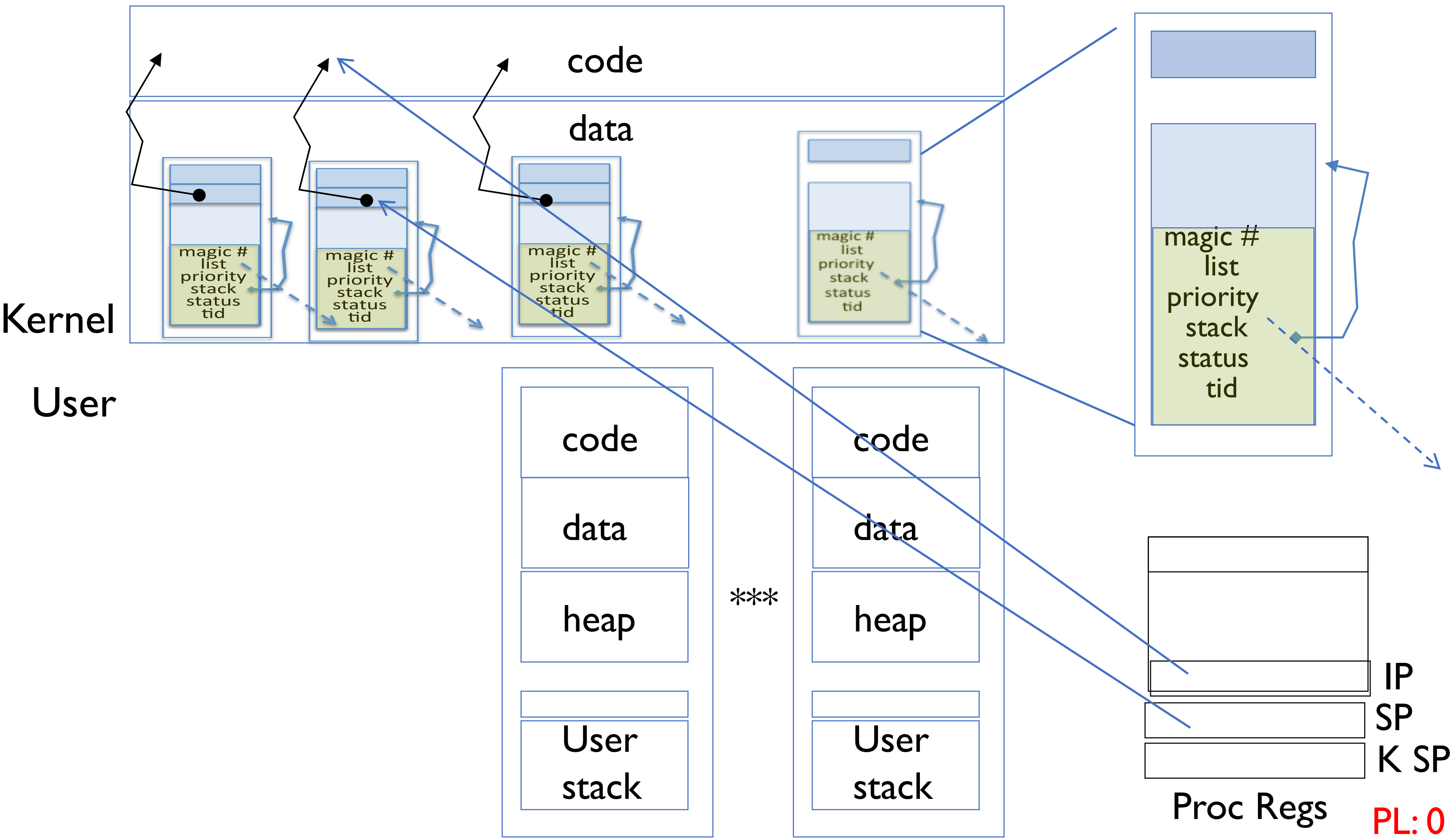
- x86 proc holds interrupt SP high system level
- During user thread exec, associated kernel thread is “standing by”

In Kernel thread



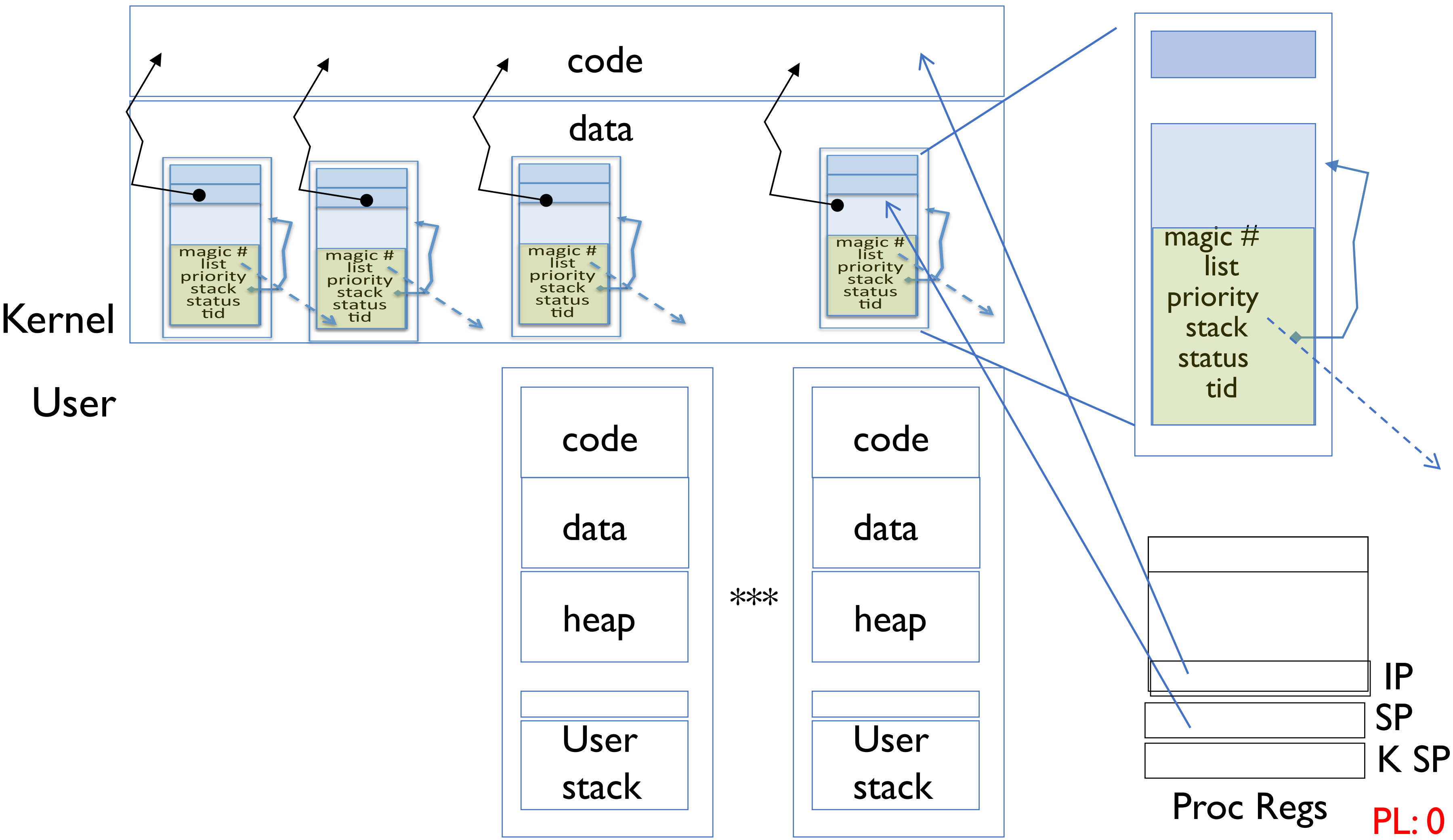
- Kernel threads execute with small stack in thread struct
- Scheduler selects among ready kernel and user threads

Thread Switch (switch.S)

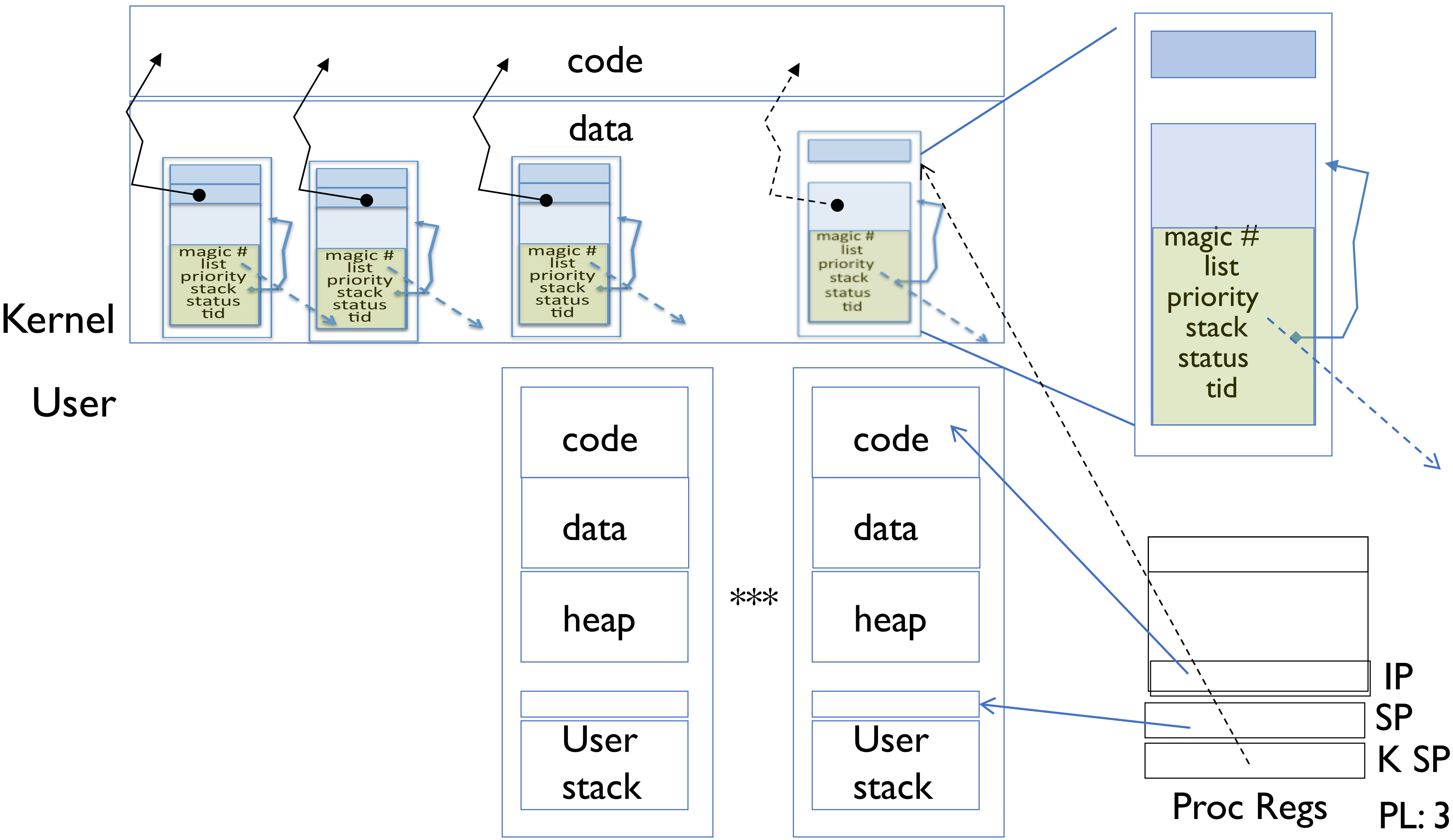


- switch_threads: save regs on current small stack, change SP, return from destination threads call to switch_threads

Switch to Kernel Thread for Process

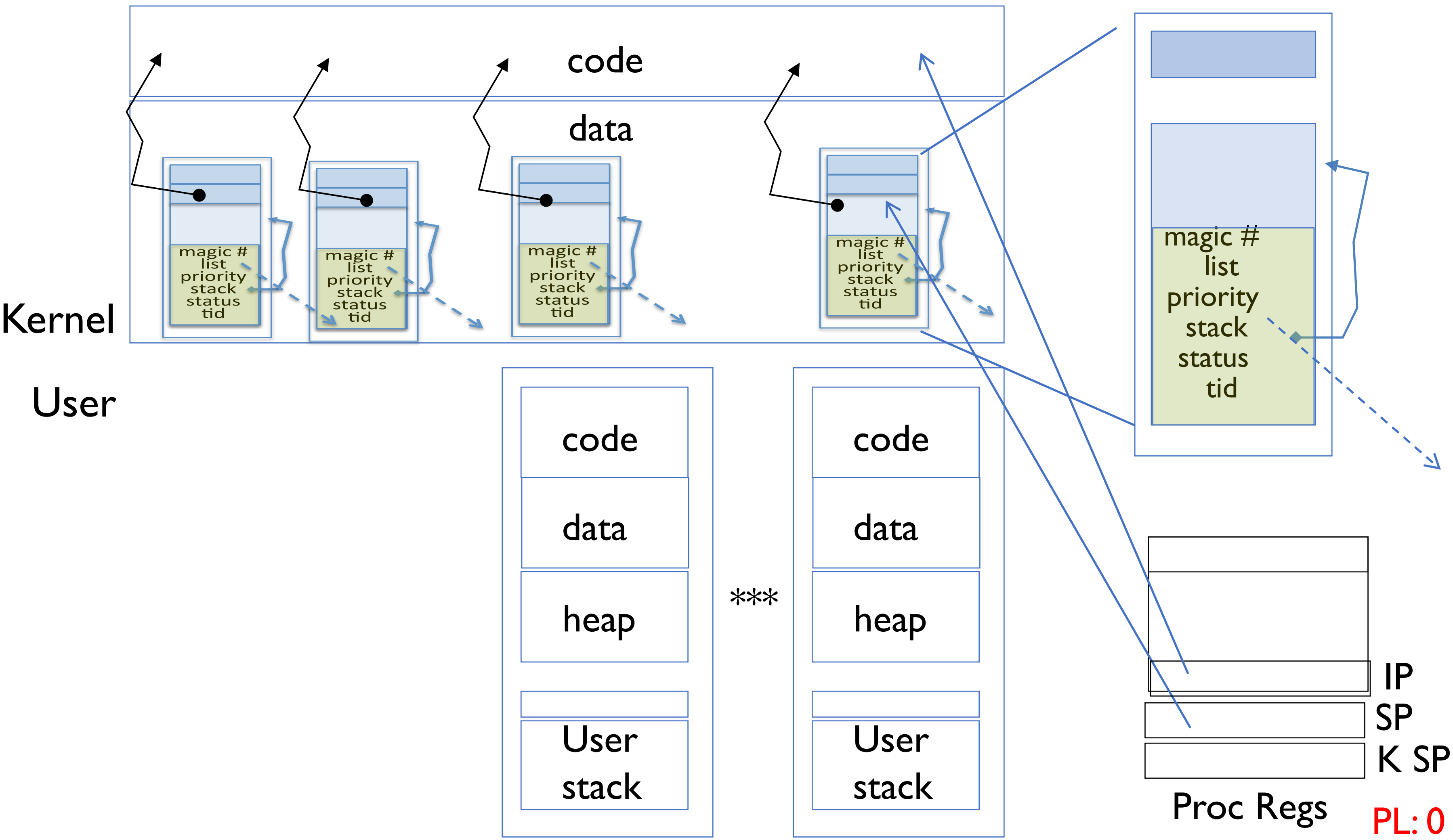


Kernel->User



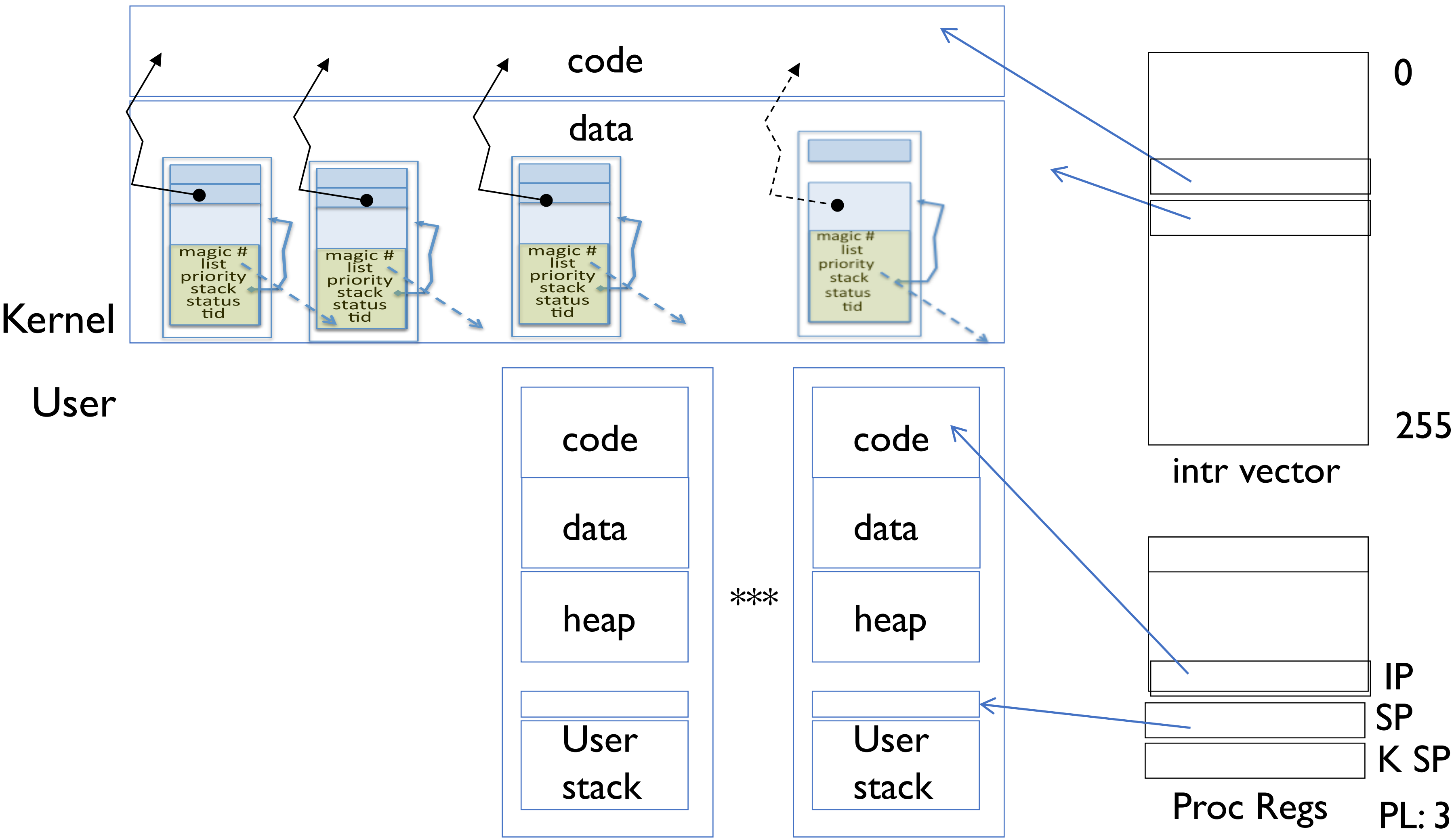
- iret restores user stack and PL

User->Kernel



- Mechanism to resume k-thread goes through interrupt vector

User->Kernel via interrupt vector



- Interrupt transfers control through the IV (IDT in x86)
- iret restores user stack and PL

Too Much Milk: Correctness

1. At most one person buys milk

2. At least one person buys milk if needed

Solution Attempt #1

- Leave a note
  - Place on fridge before buying
  - Remove after buying
  - Don't go to store if there's already a note

- Leaving/checking a note is atomic (word load/store)

```
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note;
  }
}
```

Attempt #1 in Action

Alice

```
if (noMilk) {
  if (noNote) {


    leave Note;
    buy milk;
    remove Note;
  }
}
```

Bob

```


if (noMilk) {
  if (noNote) {





    leave Note;
    buy milk;
    remove note;
  }
}
```

Solution Attempt #2

```
leave Note;
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
   }
}
remove Note;
```

But there's always a note – you just left one!

At least you don't buy milk twice…

Solution Attempt #3

- Leave a named note – each person ignores their own

Alice

```
leave note Alice
if (noMilk) {
  if (noNote Bob) {
    buy milk
  }
}
remove note Alice;
```

Bob

```
leave note Bob
if (noMilk) {
  if (noNote Alice) {
    buy milk
  }
}
remove note Bob;
```

Attempt #3 in Action

Alice

```
leave note Alice
if (noMilk) {

  if (noNote Bob) {
    buy milk
  }
}







remove note Alice
```

Bob

```


leave note Bob





if (noMilk) {
  if (noNote Alice) {
    buy milk
  }
remove note Bob
```

Solution Attempt #4

Alice

```
leave note Alice
while (note Bob) {
  do nothing
}
if (noMilk) {
  buy milk
}
remove note Alice;
```

Bob

```
leave note Bob
if (noNote Alice) {
  if (noMilk) {
    buy milk
  }
}
remove note Bob;
```

- This is a correct solution, but …

Issues with Solution 4

- Complexity
  - Proving that it works is hard
  - How do you add another thread?

- Busy-waiting
  - Alice **consumes CPU time to wait**
- Fairness
  - Who is more likely to buy milk?

OS Archaeology

- Because of the cost of developing an OS from scratch, most modern OSes have a long lineage:
- Multics → AT&T Unix → BSD Unix → Ultrix, SunOS, NetBSD,…
- Mach (micro-kernel) + BSD → NextStep → XNU → Apple OSX, iphone iOS
- Linux → Android OS
- CP/M → QDOS → MS-DOS → Windows 3.1 → NT → 95 → 98 → 2000 → XP → Vista → 7 → 8 → phone → …
- Linux → RedHat, Ubuntu, Fedora, Debian, Suse,…