Processes & Threads

CS 4410, Operating Systems

Fall 2016
Cornell University

Rachit Agarwal
Anne Bracy

See: Ch 3&4 in OSPP textbook

The slides are the product of many rounds of teaching CS 4410 by Professors Sirer, Bracy, Agarwal, George, and Van Renesse. Some content from Markus Püschel at CMU.

What is a Process?

- An instance of a program
- An abstraction of a computer:

Address Space + Execution Context + Environment

A good abstraction:

- is portable and hides implementation details
- has an intuitive and easy-to-use interface
- can be instantiated many times
- is efficient and reasonably easy to implement

Process Management

Can a program…

- Create an instance of another program?
- Wait for it to complete?
- Stop or resume another running program?
- Send it an asynchronous event?

Who should be allowed to start a process?

Possibility #1: Only the kernel may start a process

Possibility #2: User-level processes may start processes

System Call Interface

Why so skinny?

Example:

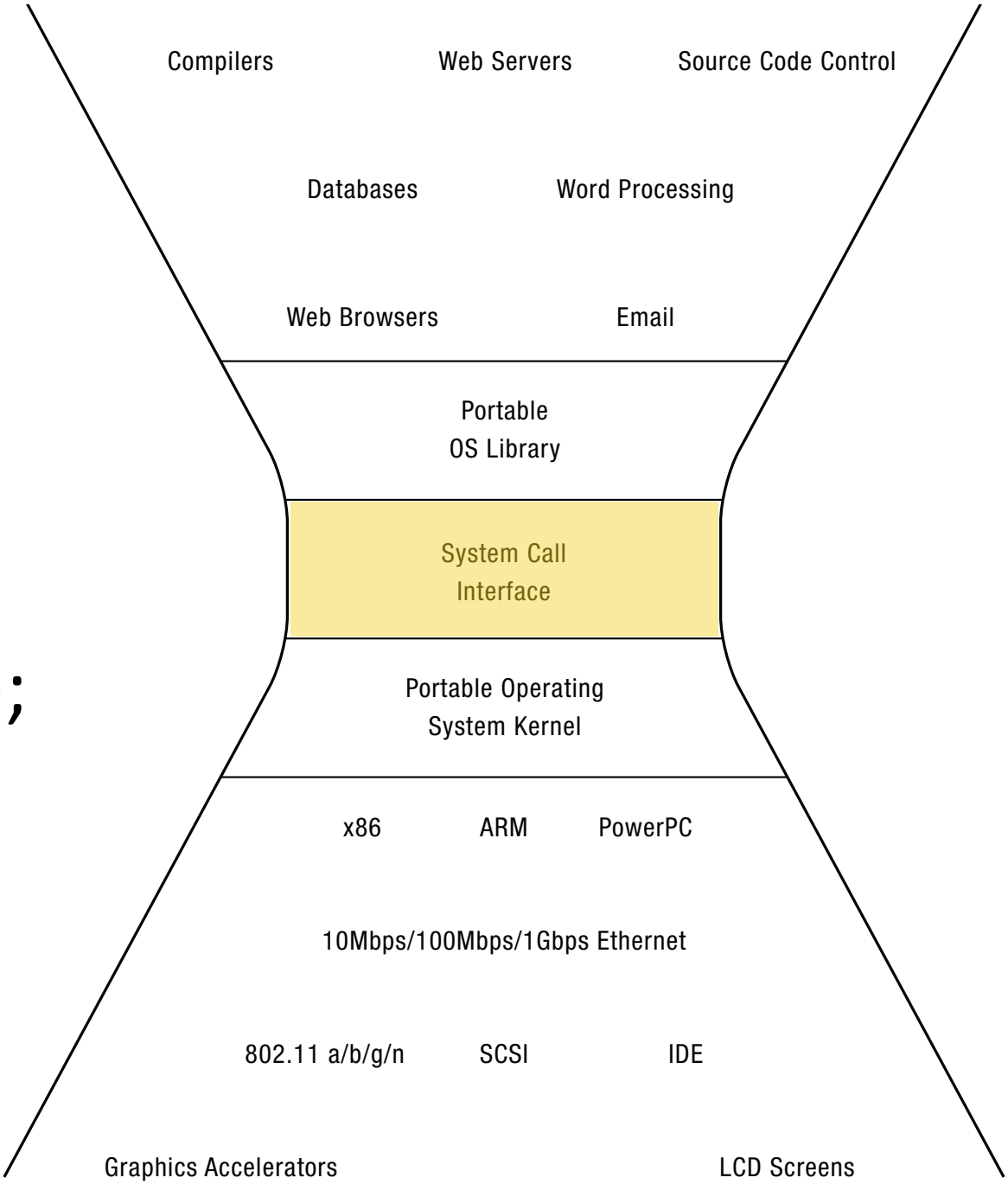
Creating a Process

Windows:

CreateProcess(…);

UNIX

fork + exec



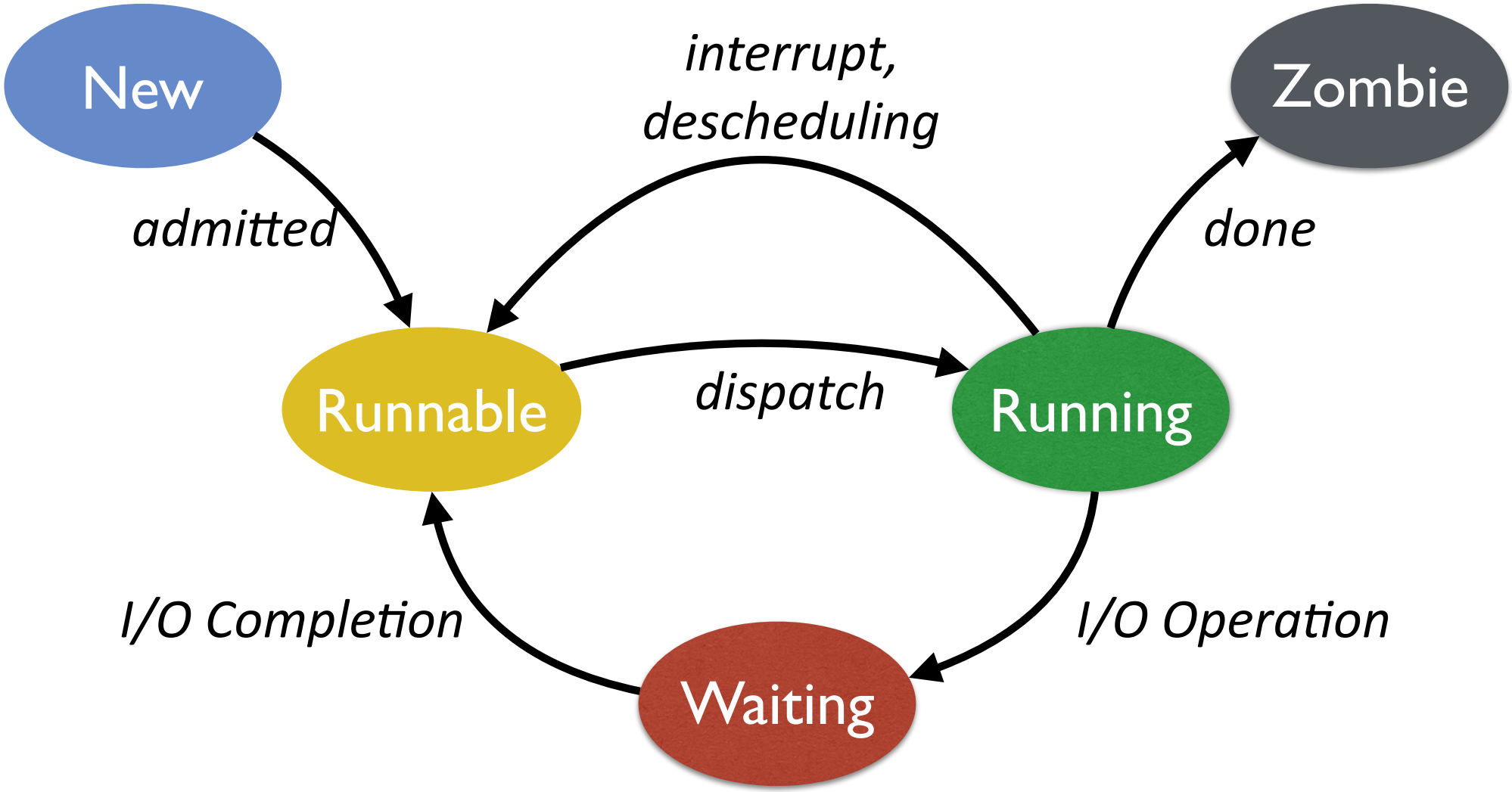
Beginning a Process via CreateProcess

Kernel has to:

- Create & initialize PCB in the kernel
- Create and initialize a new address space
- Load the program into the address space
- Copy arguments into memory in address space
- Initialize hw context to start execution at “start”
- Inform scheduler that new process is ready to run

[Windows]

Abstract Life of a Process



Details on Thursday.

CreateProcess (Simplified)

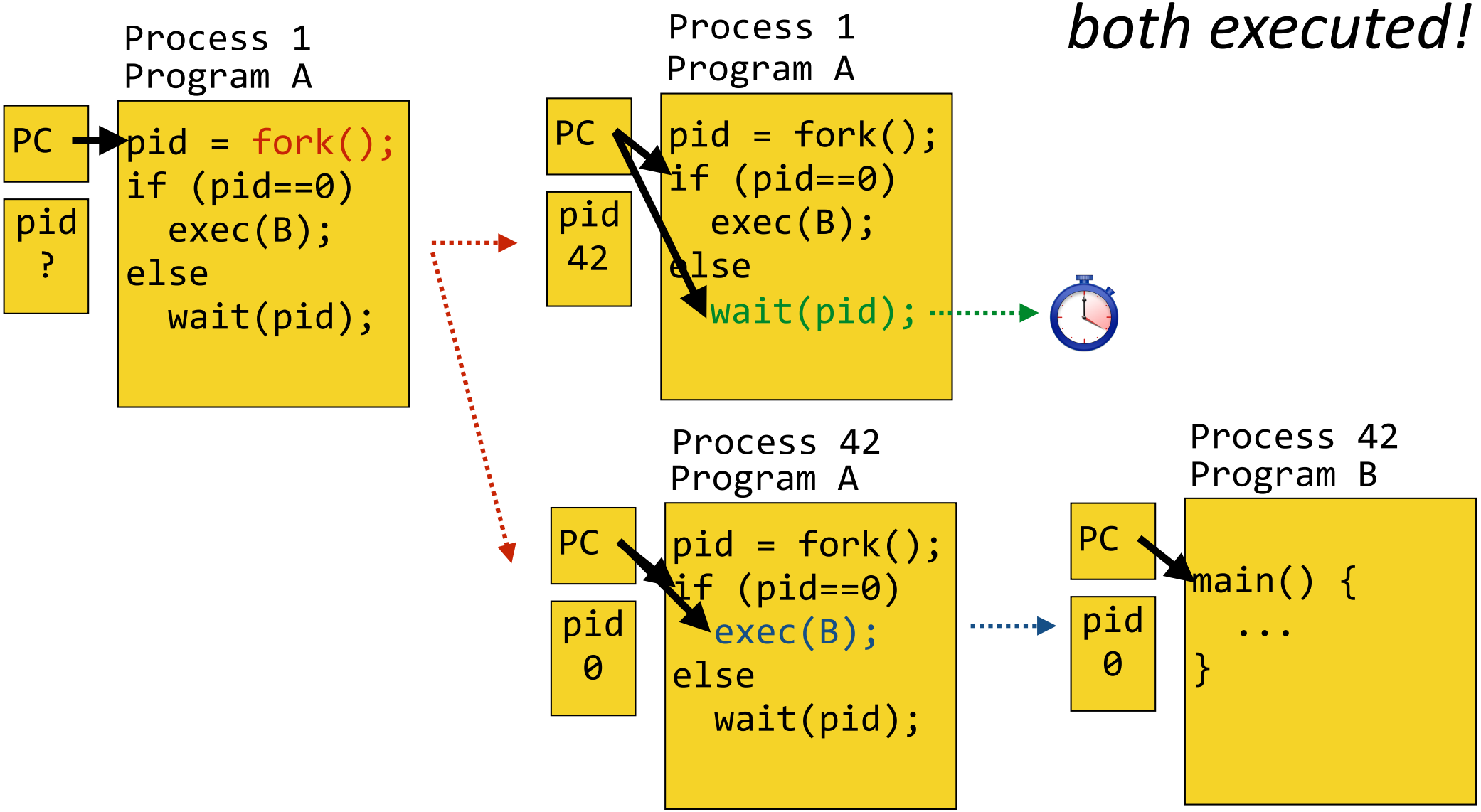
System Call

```
if (!CreateProcess(
    NULL,           // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,            // Pointer to STARTUPINFO structure
    &pi )           // Ptr to PROCESS_INFORMATION structure
)
```

[Windows]

Fork + Exec

***if** and **else** both executed!*



[UNIX]

Beginning a Process via ~~CreateProcess~~ Fork

Kernel has to:

- Create & initialize PCB in the kernel
- Create ~~and initialize~~ a new address space

- ~~Load the program into the address space~~
- ~~Copy arguments into memory in address space~~
- *Initialize the address space with a copy of the entire contents of the address space of the parent*

- ~~Initialize hw context to start execution at "start"~~
- *Inherit execution context of parent (e.g. open files)*

- Inform scheduler that new process is ready to run

[UNIX]

Code example

```
/*
 * Corresponds to Figure 3.5 in the textbook
 *
 */

#include <stdio.h>
#include <unistd.h>

int main() {

   int child_pid = fork();

   if (child_pid == 0) {          // child process
        printf("I am process #%d\n", getpid());
        return 0;
   } else {                       // parent process.
        printf("I am the parent of process #%d\n", child_pid);
        return 0;
   }
}
```

Possible outputs?

Creating and Managing Processes

fork	Create a child process as a clone of the current process. Returns to both parent and child.
exec (prog, args)	Run the application **`prog`** in the current process.
exit	Tell the kernel the current process is complete, and its data structures (stack, heap, code) should be garbage collected. Why not necessarily PCB?
wait(pid)	Pause until the child process has exited.
kill (pid, type)	Send an interrupt of a specified type to a process.

[UNIX]

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

What is a Shell?

Job control system

- runs programs on behalf of the user
- allows programmer to create/manage set of programs

  - sh — Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - csh — BSD Unix C shell (tcsh: enhanced csh at CMU and elsewhere)
  - bash — "Bourne-Again" Shell

Runs at user-level. What system calls does it use?

Built-In UNIX Shell Commands

jobs	List all jobs running in the background + all stopped jobs.
bg <job>	Run the application **prog** in the current process.
fg <job>	Change a stopped or running background job to a running in the foreground.
kill <job>	Terminate a job.

[UNIX]

show in action, + exec

Signals

A virtualized interrupt. Allow applications to behave like operating systems.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctrl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal (e.g. ctrl-z from keyboard)

[UNIX]

youtube?

Sending a Signal

Kernel delivers a signal to a destination process

For one of the following reasons:

- Kernel detected a system event (e.g. div-by-zero (SIGFPE) or termination of a child (SIGCHLD))
- A process invoked the **kill system call** requesting kernel to send signal to another process
    - debugging
    - suspension
    - resumption
    - timer expiration

Receiving a Signal

A destination process **receives** a signal when it is forced by the kernel to react in some way to the delivery of the signal

Three possible ways to react:

1. Ignore the signal (do nothing)
2. Terminate process (+ optional core dump)
3. Catch the signal by executing a user-level function called signal handler
   - Like a hardware exception handler being called in response to an asynchronous interrupt

show handler.c

Signal Example

```
void int_handler(int sig) {
   printf("Process %d received signal %d\n", getpid(), sig);
   exit(0);
}

int main() {
   pid_t pid[N];
   int i, child_status;
   signal(SIGINT, int_handler);
   for (i = 0; i < N; i++) // N forks
       if ((pid[i] = fork()) == 0) {
          while(1); //child infinite loop
       }

   for (i = 0; i < N; i++) {    // parent continues executing
       printf("Killing proc. %d\n", pid[i]);
       kill(pid[i], SIGINT);
   }
   for (i = 0; i < N; i++) {
       pid_t wpid = wait(&child_status);
       if (WIFEXITED(child_status)) // parent checks for each child's exit
           printf("Child %d terminated w exit status %d\n",
                  wpid, WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
   }
   exit(0);
}
```

Blocked Signals

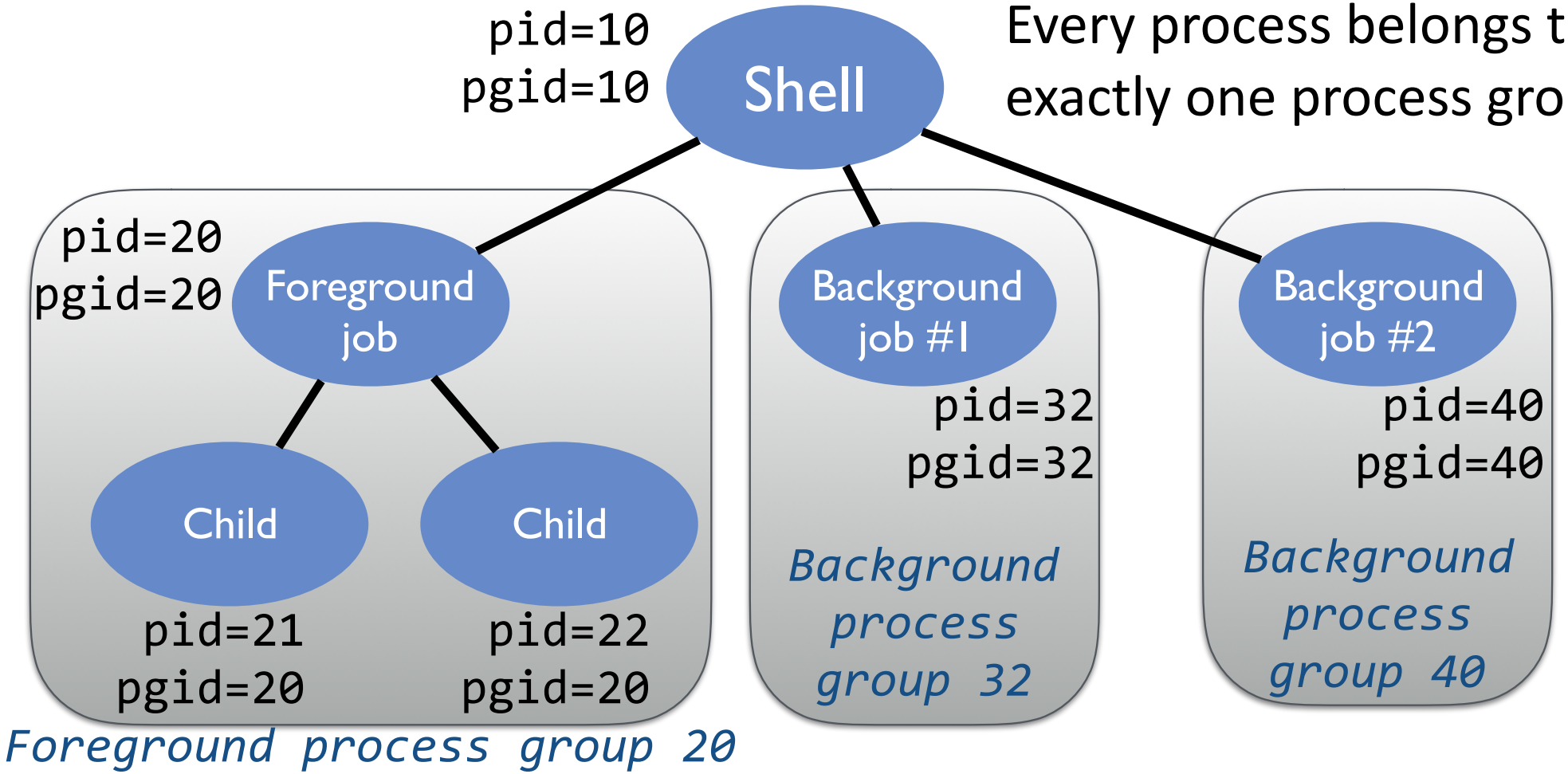
A process can block the receipt of certain signals

- Blocked signals can be delivered, but will not be received until the signal is unblocked

Kernel maintains pending and blocked bit vectors in the context of each process

- **blocked**: represents the set of blocked signals
  Can be set and cleared by using the **`sigprocmask`** function

Process Groups



`getpgrp()` Return process group of current process

`setpgid()` Change process group of a process

`/bin/kill -9 21` Send SIGKILL to process 24818

`/bin/kill -9 -20` Send SIGKILL to every process in process group 20

Implementing a (really, really simple) Shell

```
void eval(char *cmdline) {
  char *argv[MAXARGS]; /* argv for execve() */
  int bg;              /* should the job run in bg or fg? */
  pid_t pid;           /* process id */

  bg = parseline(cmdline, argv);
  if (!builtin_command(argv)) {
    if ((pid = Fork()) == 0) {   /* child runs user job */
      if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
      }
    }

   if (!bg) {   /* parent waits for fg job to terminate */
      int status;
      if (waitpid(pid, &status, 0) < 0)
        unix_error("waitfg: waitpid error");
   }
   else         /* otherwise, don't wait for bg job */
      printf("%d %s", pid, cmdline);
  }
}
```

And Now… Threads!

Why Threads?

- Performance: exploiting multiple processors

Does multi-threading only make sense on multicore?

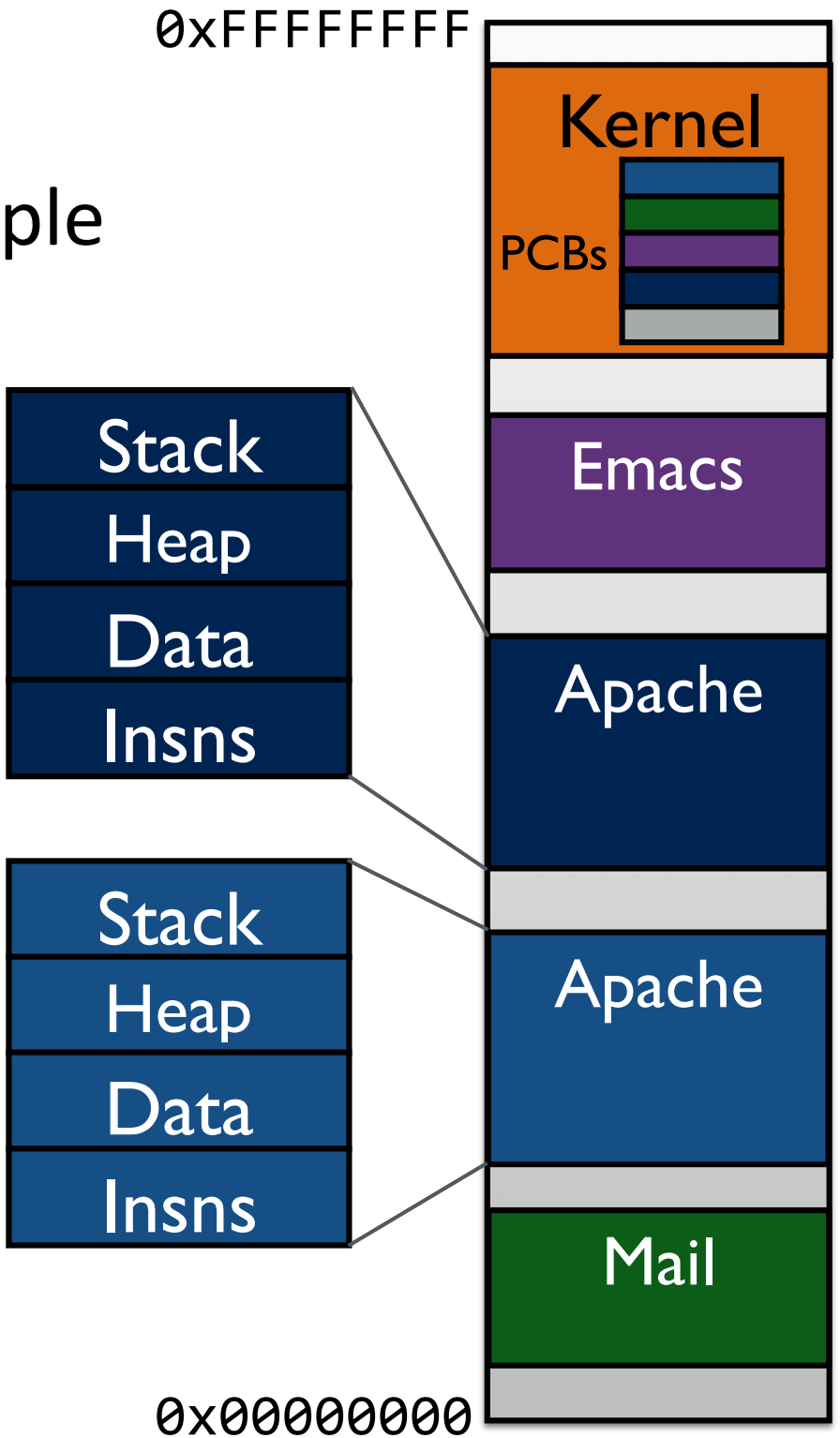
- Program structure: expressing logically concurrent tasks
- Responsiveness: shifting work to run in the background
- Performance: managing I/O devices

What happens when…

Apache wants to run multiple concurrent computations?

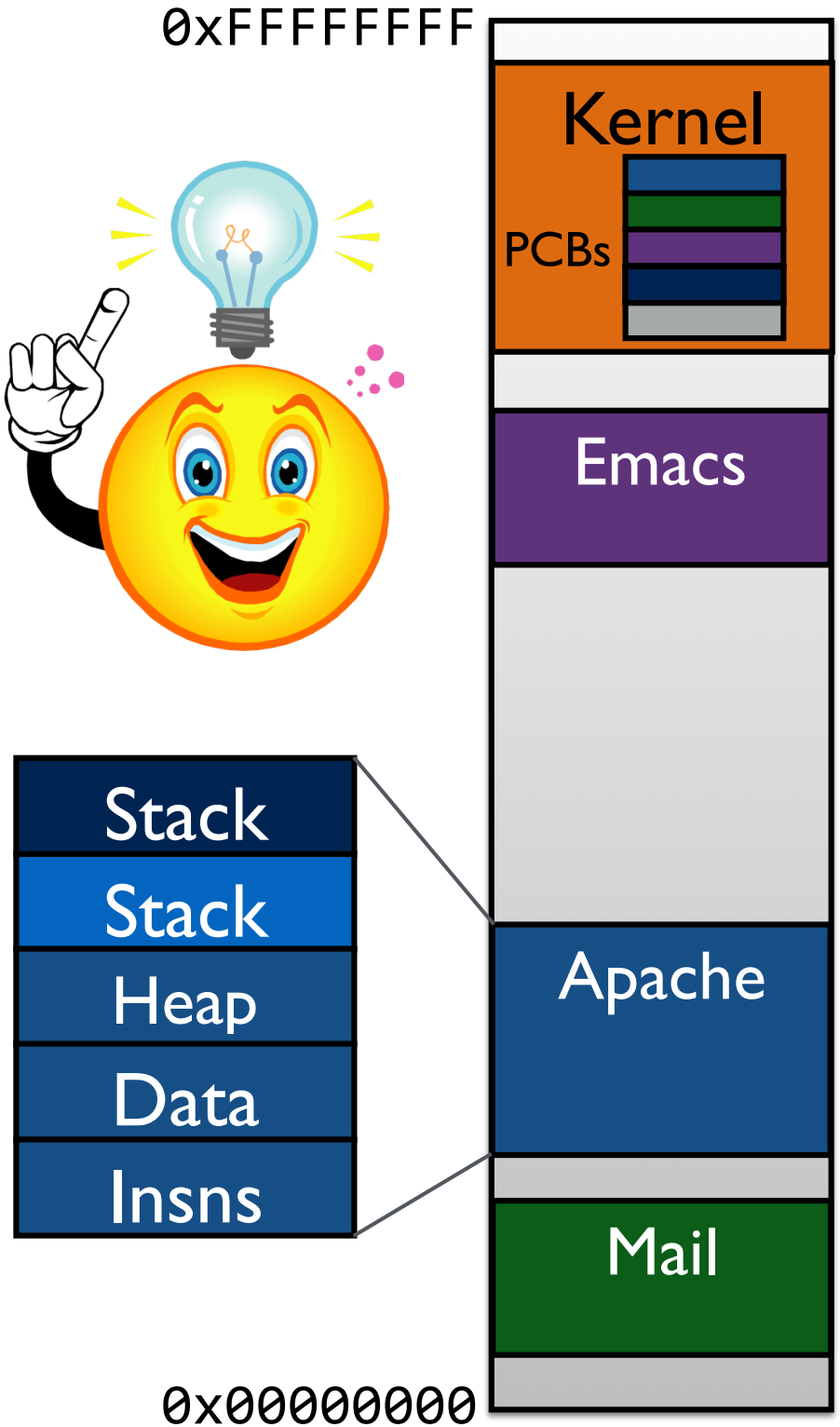
Two heavyweight address spaces for two concurrent computations?

What is distinct about these address spaces?



Idea!

Eliminate duplicate address spaces and place concurrent computations in the same address space.



Process vs. Thread

Process:

- Address Space
- Shared I/O resources
- One or more **Threads**:

Not Shared:

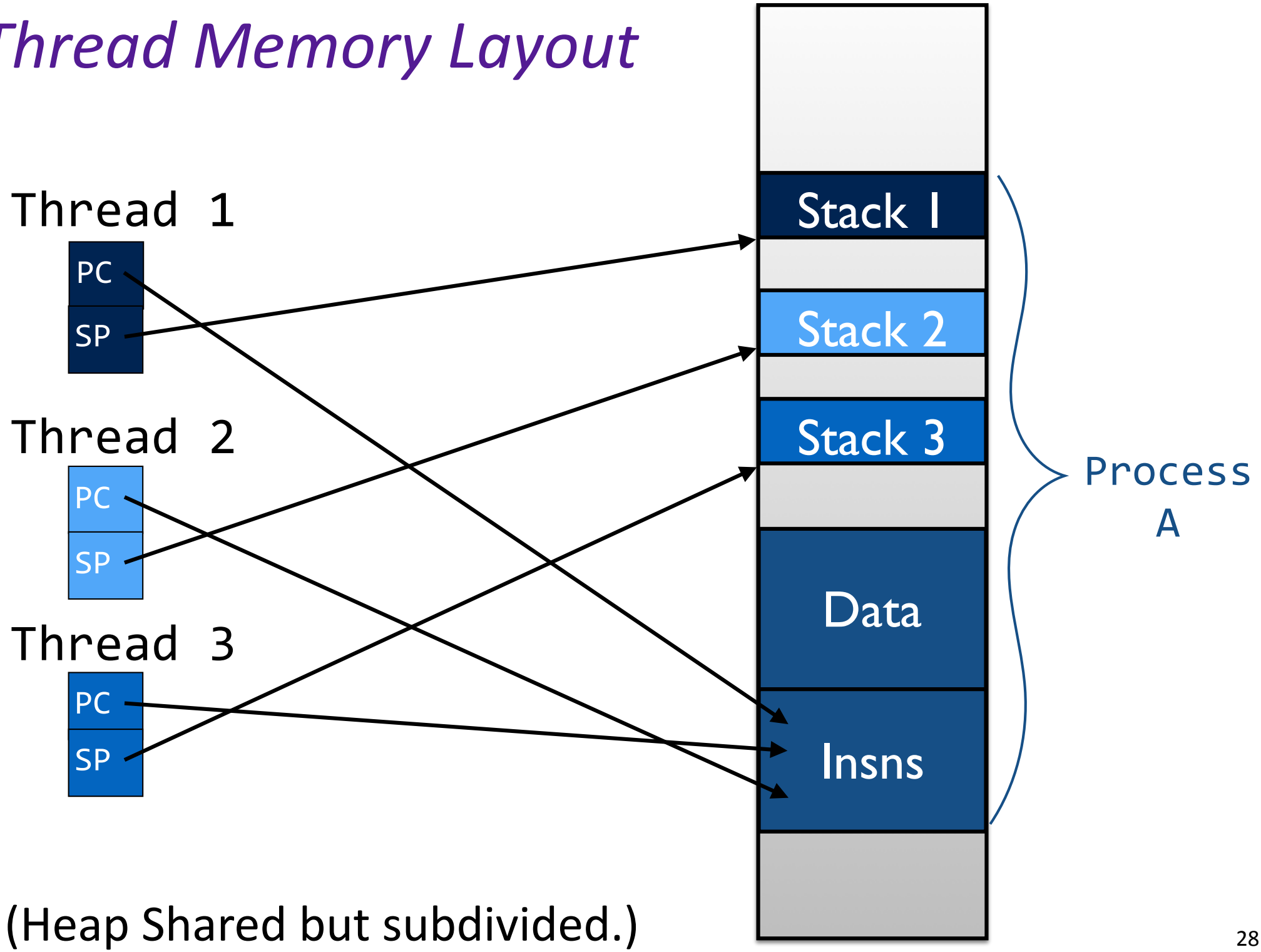
- Registers, PC, SP
- Stack

Shared:

- Code
- Data
- Privileges

Other terms for threads: Lightweight Process, Thread of Control, Task

Thread Memory Layout



(Heap Shared but subdivided.)

Simple Thread API

`void thread_create (thread,func,arg)`	Create a new thread, storing information about it in thread. Concurrently with the calling thread, thread executes the function **func** with the argument **arg**.
`void thread_yield ()`	Calling thread voluntarily gives up processor to let other thread(s) run. Scheduler can resume running the calling thread whenever it chooses to do so.
`int thread_join (thread)`	Wait for thread to finish if it has not already done so; then return the value passed to thread_exit by that thread. Note that thread_join may be called only once for each thread.
`void thread_exit (ret)`	Finish the current thread. Store the value ret in the current thread's data structure. If another thread is already waiting in a call to thread_join, resume it.

Coding Example

process_share.c

thread_share.c

Threads

Lighter-weight than processes

- **process** is an abstract *computer* (CPU, mem, devices, …)
- **thread** is an abstract *core*

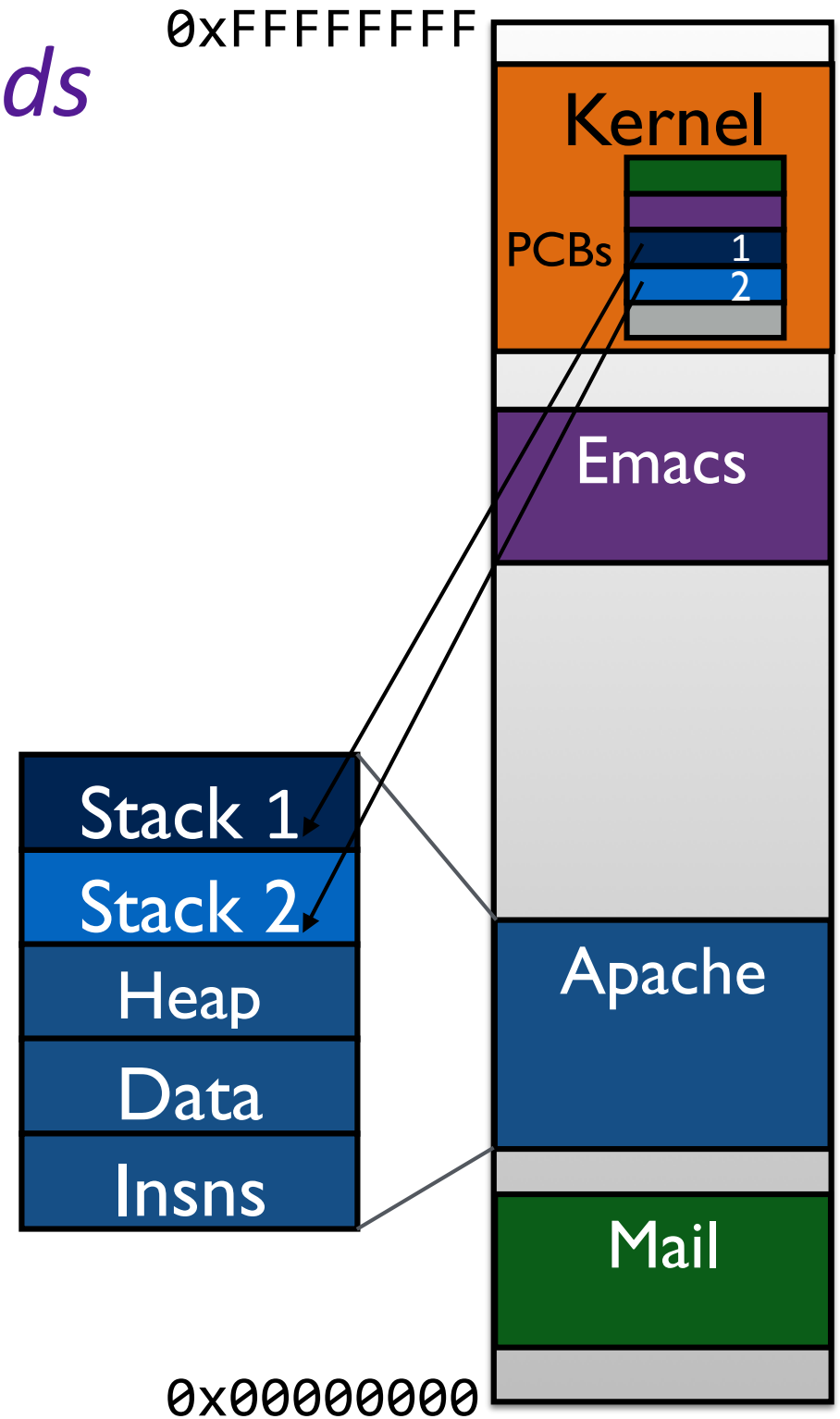
Threads need to be mutually trusting (*Why?*)

Ideal for concurrent programs where lots of code and data are shared

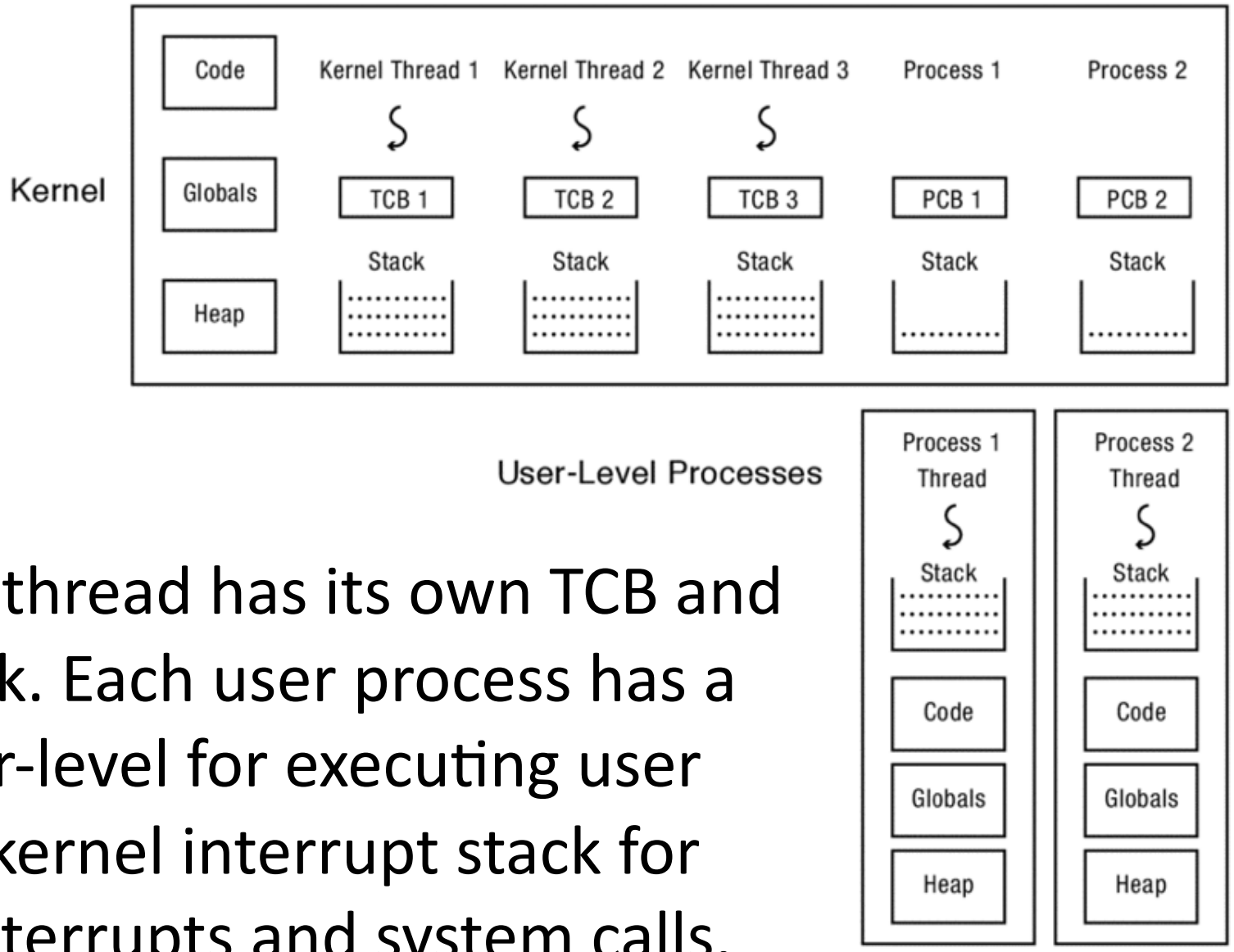
- Servers, GUI code, …

Option #1: Kernel Threads

- Threads share a single location in memory
- Separate PCBs (TCBs) for each thread
- PCBs have:
  - **same**: base & bound register values
  - **different**: PC, SP, registers

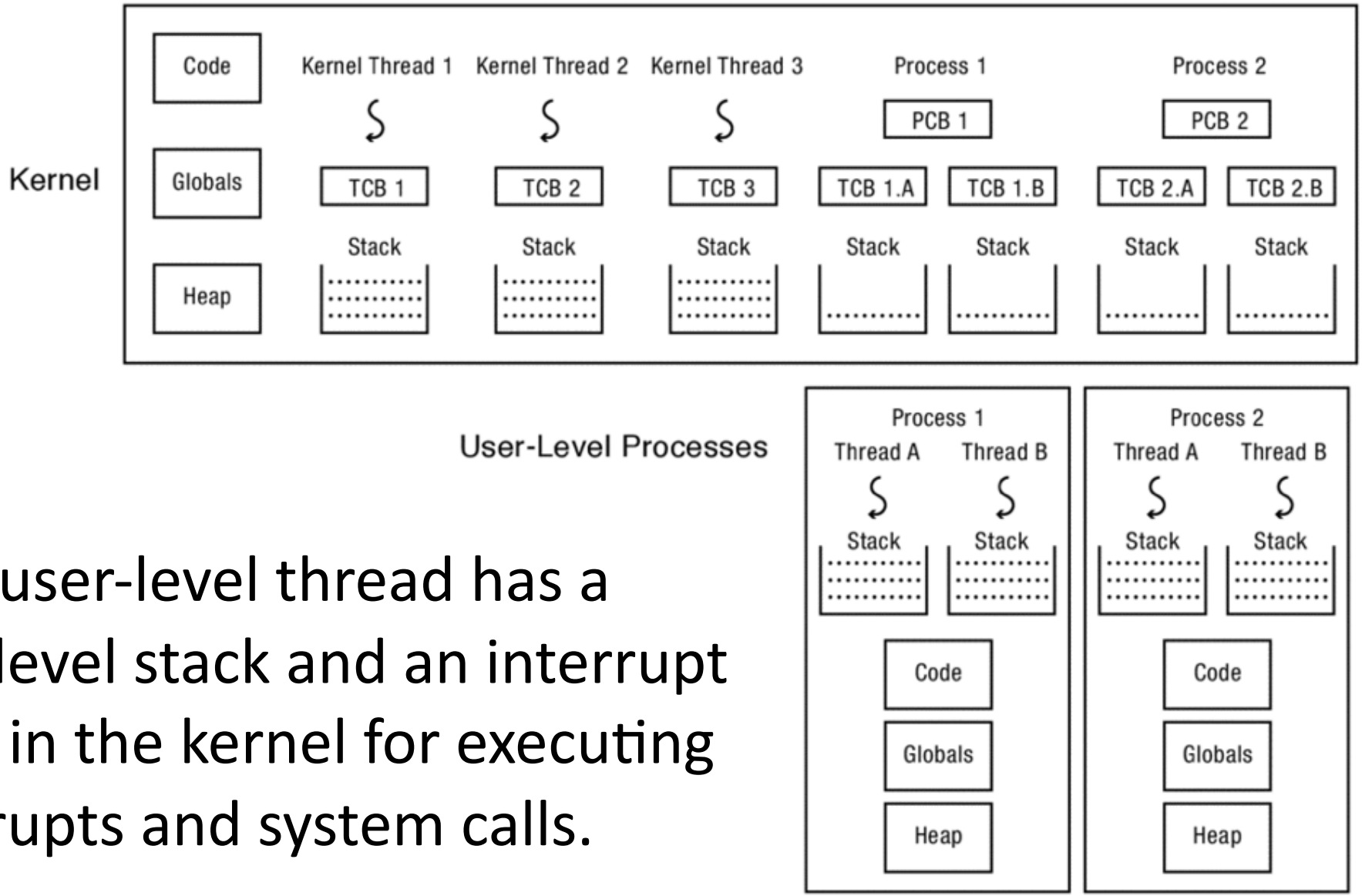


Multi-threaded kernel with three kernel threads and two single-threaded user-level processes.



Each kernel thread has its own TCB and its own stack. Each user process has a stack at user-level for executing user code and a kernel interrupt stack for executing interrupts and system calls.

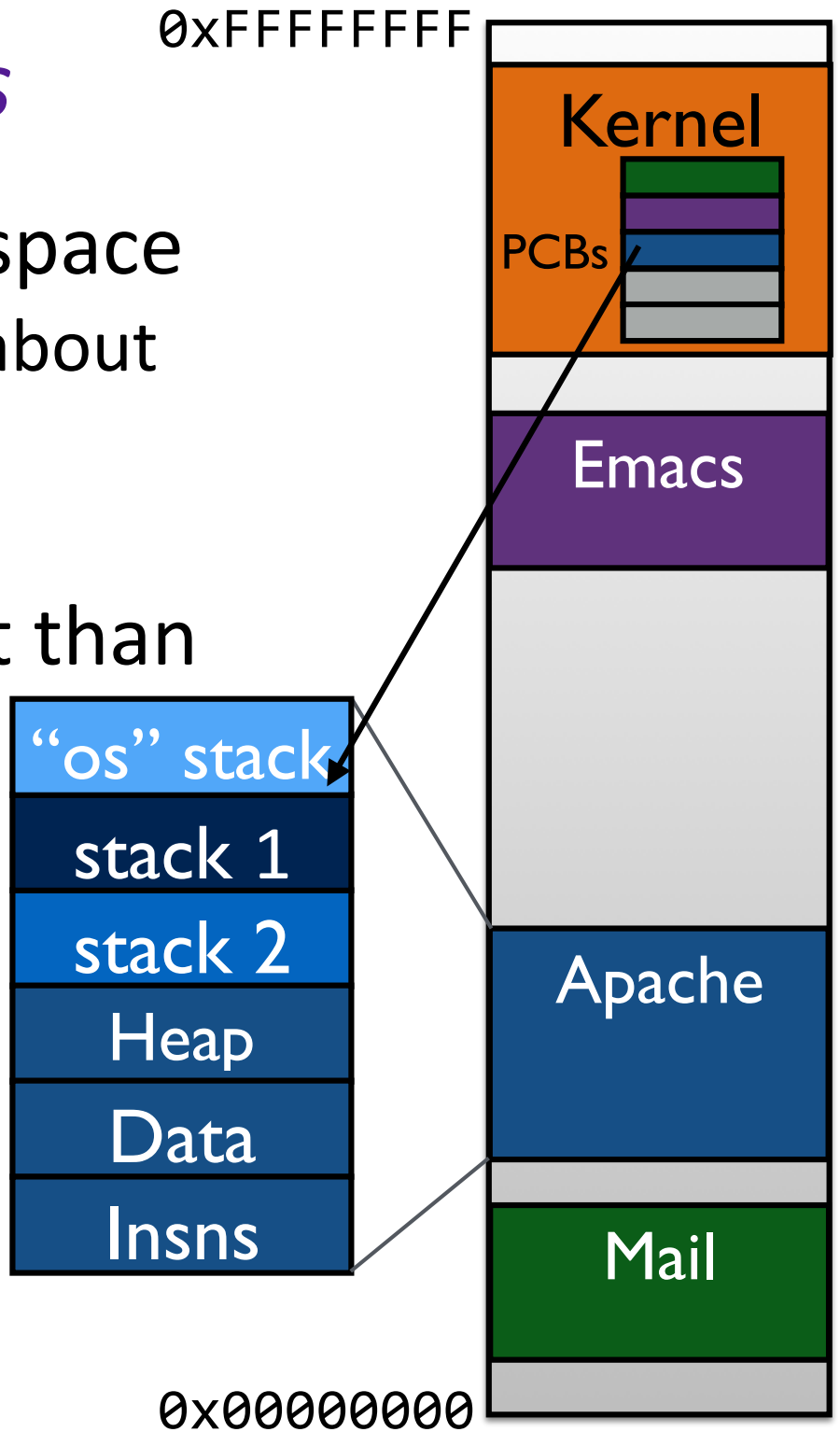
A multi-threaded kernel with 3 kernel threads and 2 user-level processes, each with 2 threads.



Each user-level thread has a user-level stack and an interrupt stack in the kernel for executing interrupts and system calls.

Option #2: User Threads

- Build a mini-OS in user space
  - Real OS doesn't know about multiple threads
  - Single PCB
- Generally more efficient than kernel threads (*Why?)*
- But kernel threads simplify system call handling and scheduling (*Why?)*



Implementing User Threads (4411-P1!)

User process supports:

- Thread Control Block (**TCB**) table
  - one entry per thread
- "**context switch**" operations
  - save/restore thread state in TCB
  - much like kernel-level context switches
- **yield()** operation:
  - thread releases core, allows another thread to use it
  - Automatic pre-emption not always supported
- **Thread scheduler**

User Threads & System Calls

- With user threads, a process may have multiple systems calls outstanding simultaneously (one per thread)
- Kernel PCB must support this (*Why?*)

Cooperative vs. Preemptive Multithreading

Cooperative: thread runs until it yields control to another thread — yield is in the code itself

+ better control of scheduling
+ simpler reasoning about shared resources
– starvation (slow interfaces)
– multi-core → *reasoning not simpler afterall!*

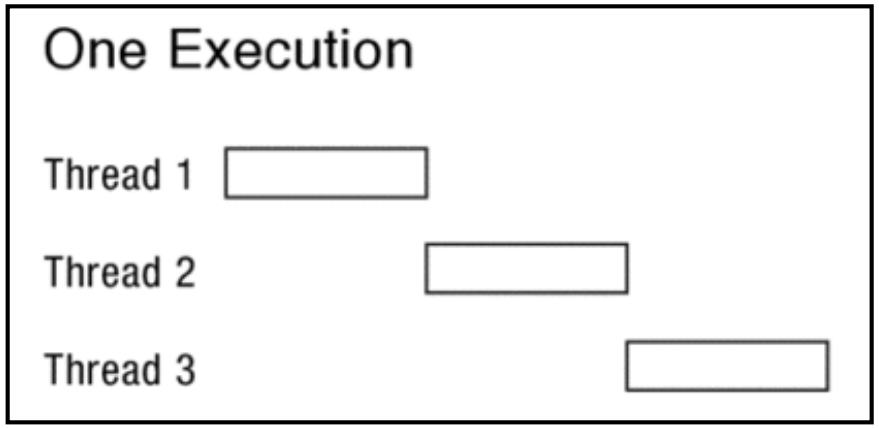
[Not common these days]

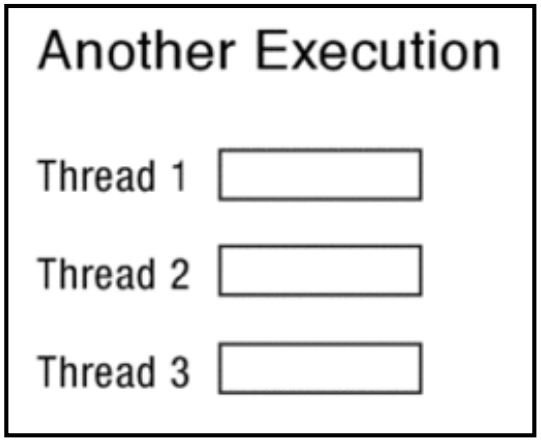
Multi-threading == preemptive multi-threading

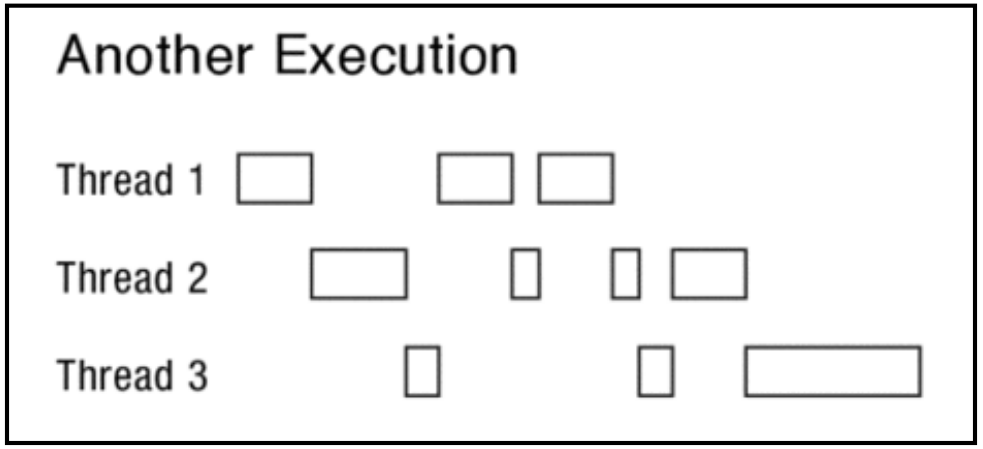
except in 11-P1 "Non-Premptive Multitasking"

Multithreading Programming Model

Rule #1: don't presume to know the schedule







Shared Resources → Synchronization Matters!

thread_share.c revisited (*Next Week!)*