Processes - Representation in the Operating System & Syscalls

David E. Culler

CS162 – Operating Systems and Systems Programming

http://cs162.eecs.berkeley.edu/

Lecture 3

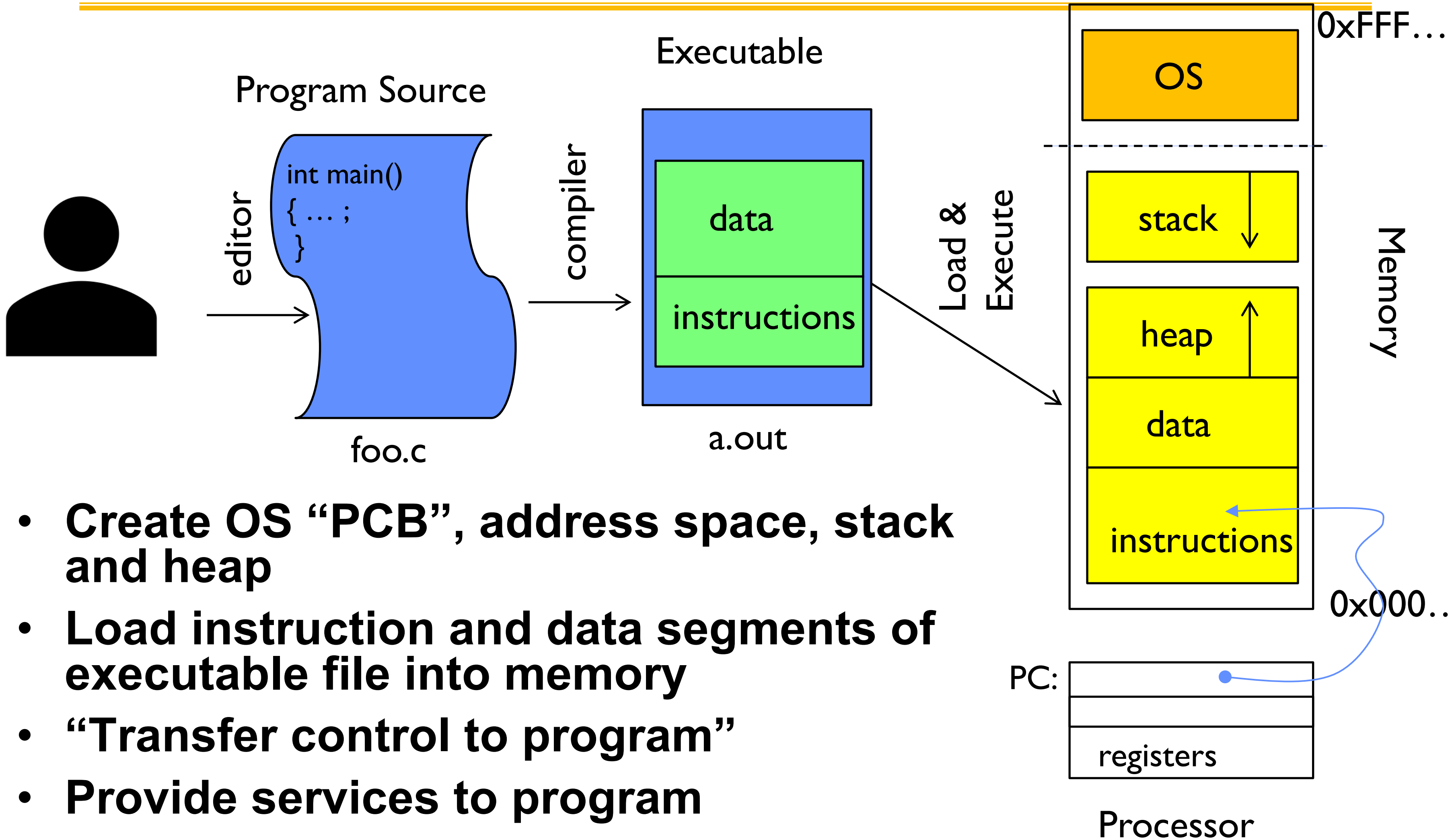
September 5, 2019

Read A&D Ch3, 4.4-6
HW0 Due 9/6
Early drop: 9/6
HW 1 out 9/7, Due 9/18

Recall: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Virtual Address space**
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads + …
- **Dual mode operation / Protection**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other

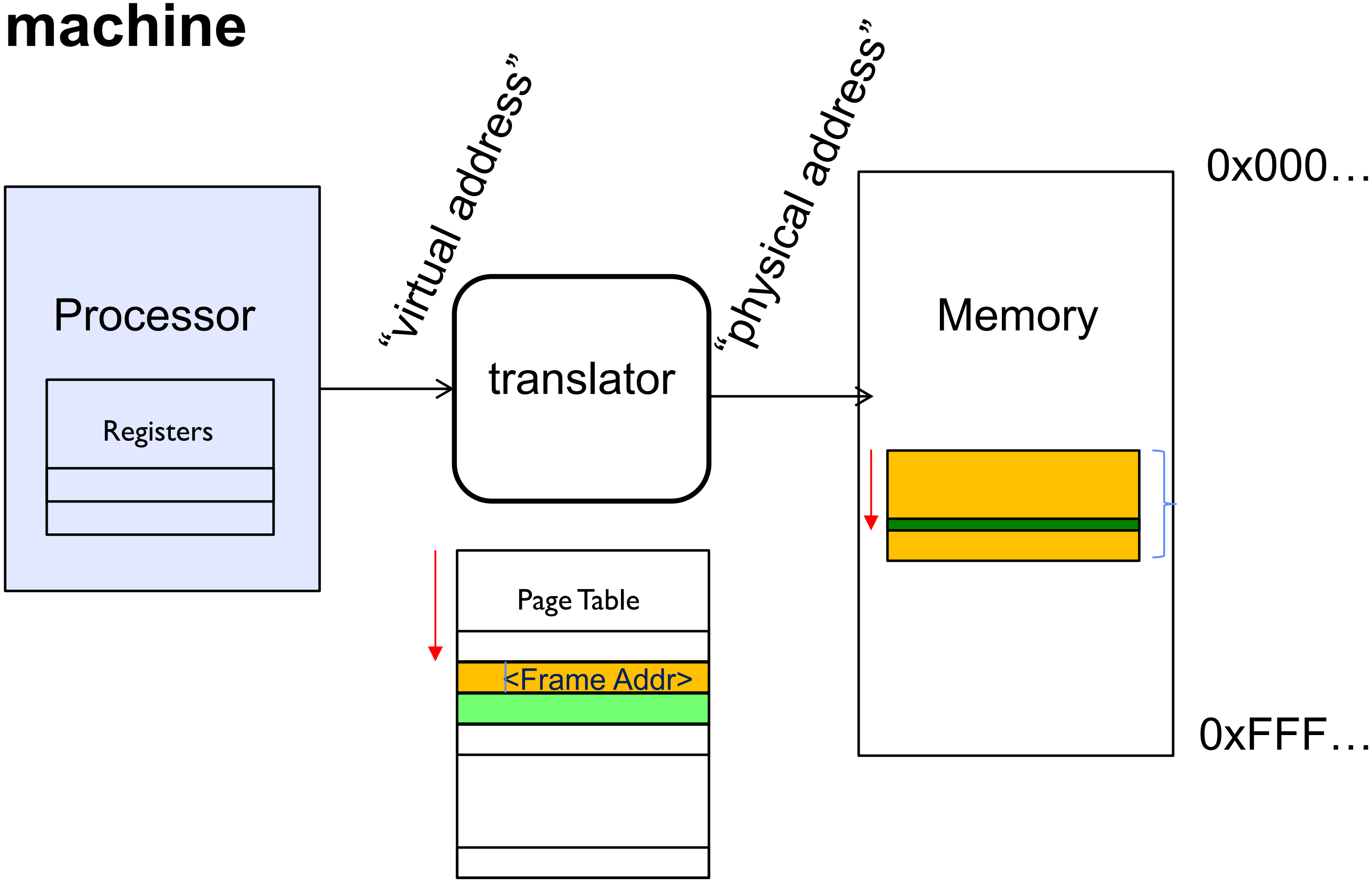
Recall: OS Bottom Line: Run Programs



- **Create OS "PCB", address space, stack and heap**
- **Load instruction and data segments of executable file into memory**
- **"Transfer control to program"**
- **Provide services to program**
- **While protecting OS and program**

Key OS Concept: Address Space

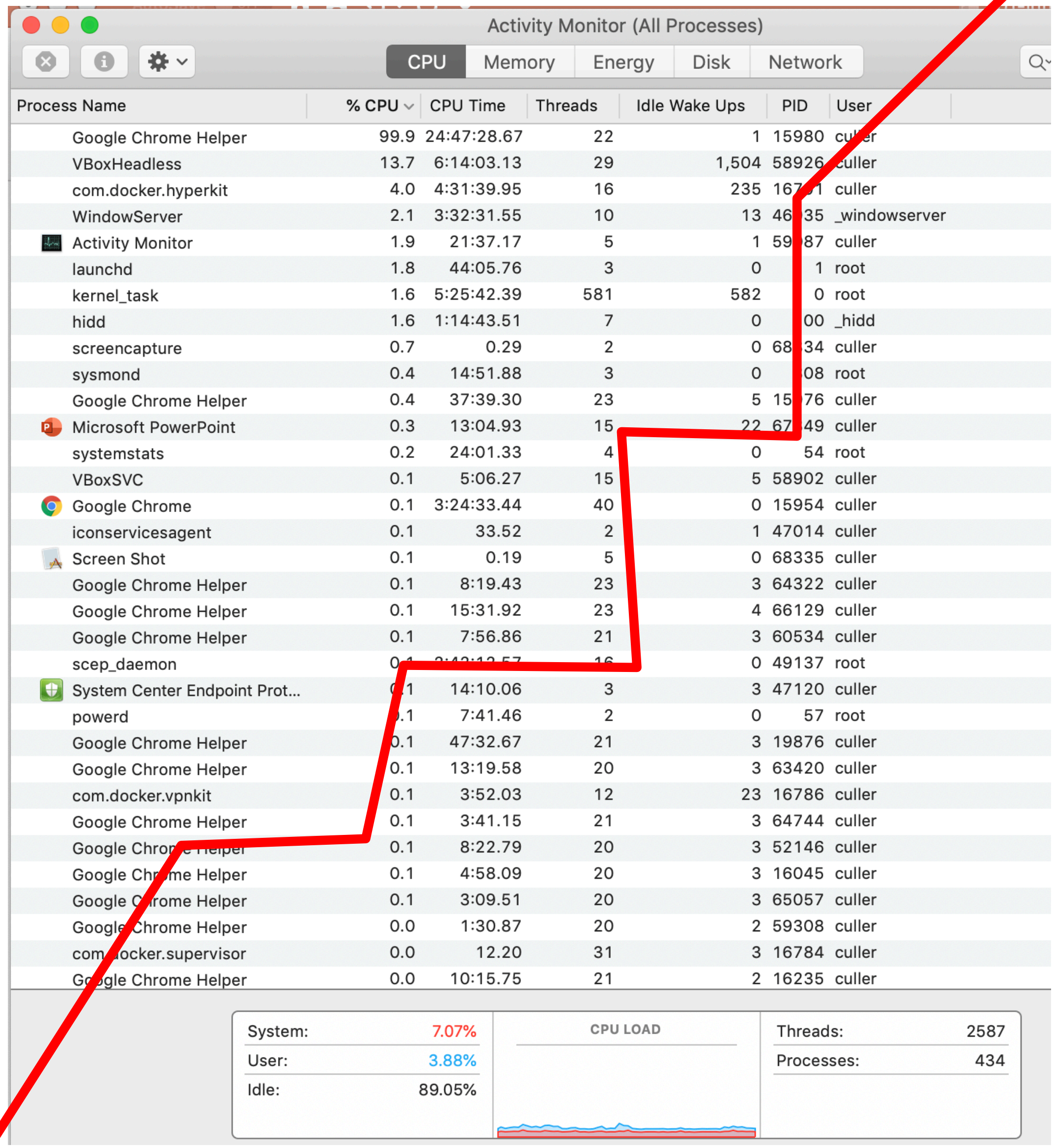
- **Program operates in an address space that is distinct from the physical memory space of the machine**



Recall: The Process

- **Definition:** execution environment with restricted rights
  - Address Space with One or More Threads
    - » *Page table per process!*
  - Owns memory (mapped pages)
  - Owns file descriptors, file system context, …
  - Encapsulates one or more threads sharing process resources
- Application program executes as a process
  - Complex applications can fork/exec child processes [later]
- Why processes?
  - Protected from each other. OS Protected from them.
  - Execute concurrently [trade-offs with threads? later]
  - Basic unit OS deals with

What's beneath the Illusion?



Activity Monitor (All Processes)

CPU | Memory | Energy | Disk | Network

Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
Google Chrome Helper	99.9	24:47:28.67	22	1	15980	culler
VBoxHeadless	13.7	6:14:03.13	29	1,504	58926	culler
com.docker.hyperkit	4.0	4:31:39.95	16	235	[illegible]	culler
WindowServer	2.1	3:32:31.55	10	13	[illegible]	_windowserver
Activity Monitor	1.9	21:37.17	5	1	[illegible]	culler
launchd	1.8	44:05.76	3	0	1	root
kernel_task	1.6	5:25:42.39	581	582	0	root
hidd	1.6	1:14:43.51	7	0	[illegible]	_hidd
screencapture	0.7	0.29	2	0	[illegible]	culler
sysmond	0.4	14:51.88	3	0	[illegible]	root
Google Chrome Helper	0.4	37:39.30	23	5	[illegible]	culler
Microsoft PowerPoint	0.3	13:04.93	15	22	[illegible]	culler
systemstats	0.2	24:01.33	4	0	54	root
VBoxSVC	0.1	5:06.27	15	5	58902	culler
Google Chrome	0.1	3:24:33.44	40	0	15954	culler
iconservicesagent	0.1	33.52	2	1	47014	culler
Screen Shot	0.1	0.19	5	0	68335	culler
Google Chrome Helper	0.1	8:19.43	23	3	64322	culler
Google Chrome Helper	0.1	15:31.92	23	4	66129	culler
Google Chrome Helper	0.1	7:56.86	21	3	60534	culler
scep_daemon	[illegible]	[illegible]	[illegible]	0	49137	root
System Center Endpoint Prot...	0.1	14:10.06	3	3	47120	culler
powerd	0.1	7:41.46	2	0	57	root
Google Chrome Helper	0.1	47:32.67	21	3	19876	culler
Google Chrome Helper	0.1	13:19.58	20	3	63420	culler
com.docker.vpnkit	0.1	3:52.03	12	23	16786	culler
Google Chrome Helper	0.1	3:41.15	21	3	64744	culler
Google Chrome Helper	0.1	8:22.79	20	3	52146	culler
Google Chrome Helper	0.1	4:58.09	20	3	16045	culler
Google Chrome Helper	0.1	3:09.51	20	3	65057	culler
Google Chrome Helper	0.0	1:30.87	20	2	59308	culler
com.docker.supervisor	0.0	12.20	31	3	16784	culler
Google Chrome Helper	0.0	10:15.75	21	2	16235	culler

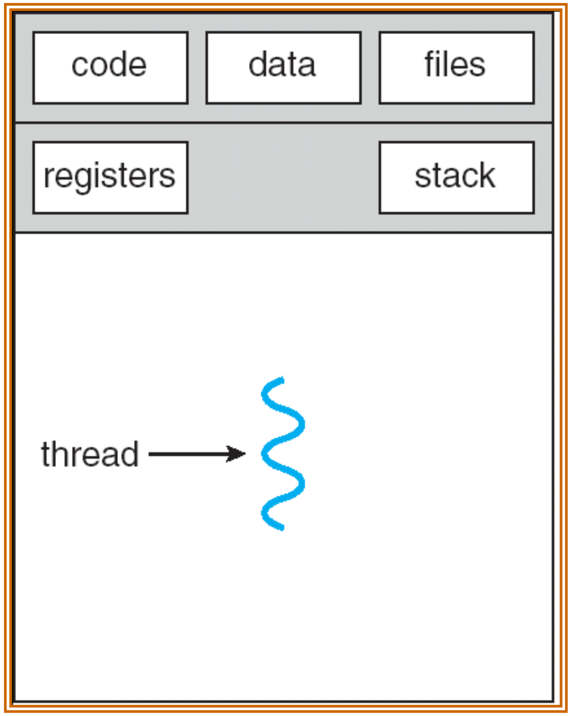
System:	7.07%	CPU LOAD	Threads:	2587
User:	3.88%		Processes:	434
Idle:	89.05%			

Today: How does the Operating System create the Process Abstraction

- What data structures are used?
- What machine structures are employed?
  – Focus on x86, since will use in projects (and everywhere)

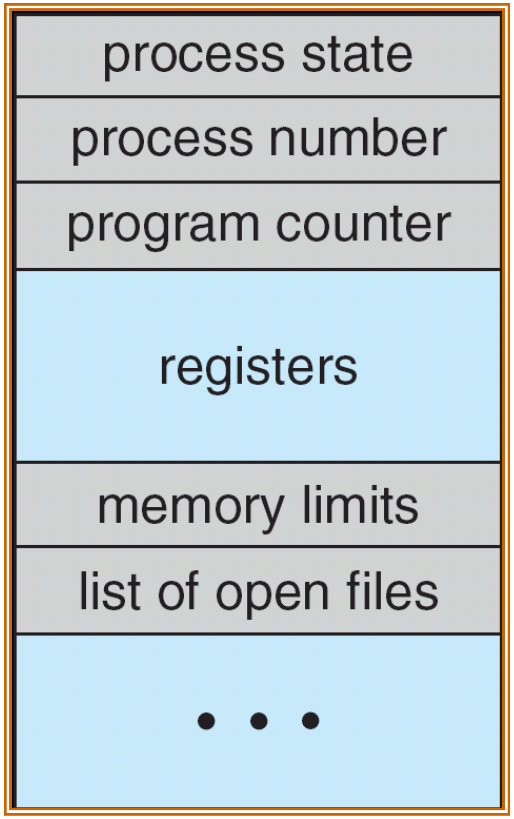
Starting Point: Single Threaded Process

- **Process: OS abstraction of what is needed to run a single program**
  1. **Sequential program execution stream**
     - » **Sequential stream of execution (thread)**
     - » **State of CPU registers**
  2. **Protected resources**
     - » **Contents of Address Space**
     - » **I/O state (more on this later)**



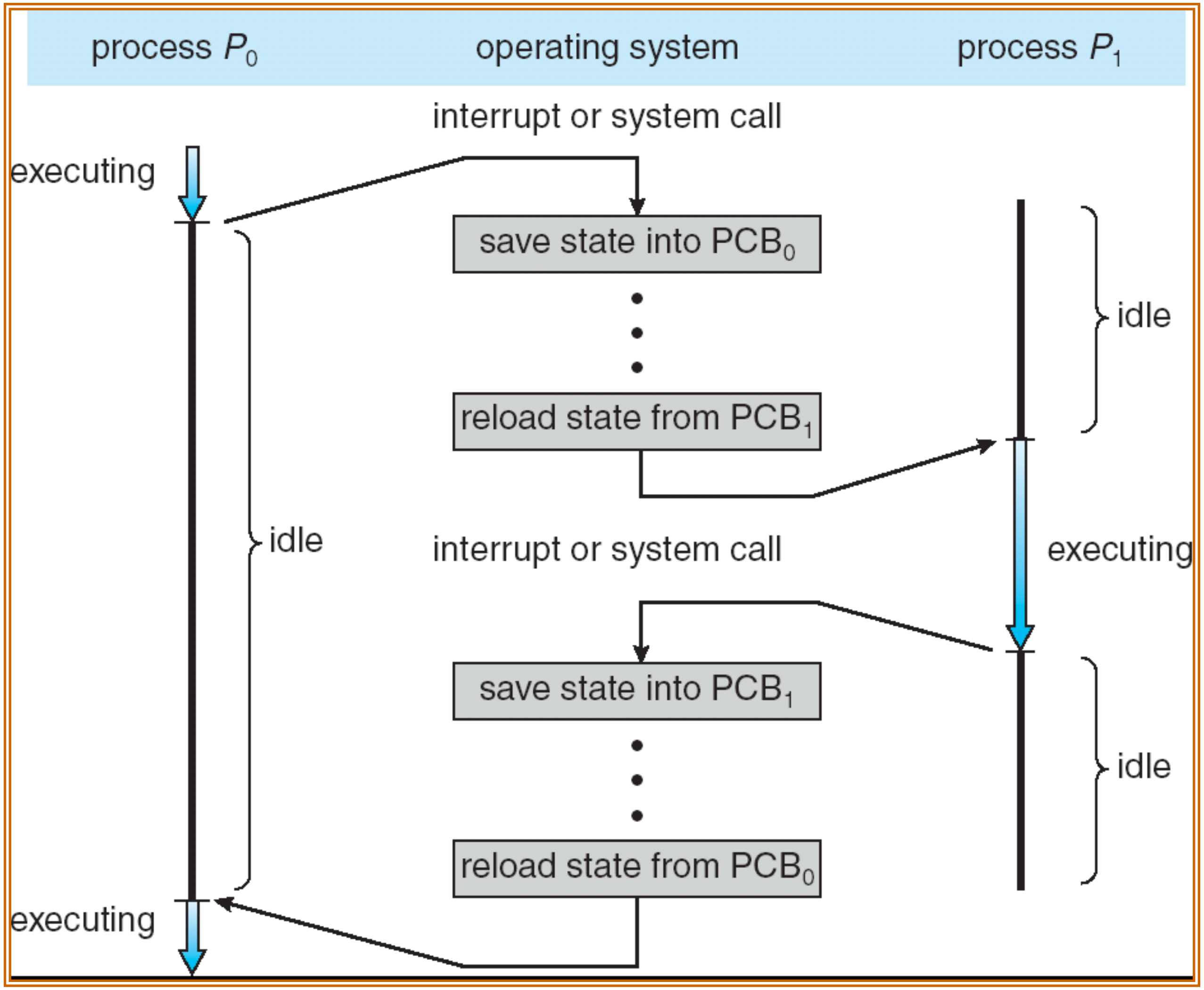
Multiplexing Processes

- **Snapshot of each process in its PCB**
  - **Only one 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**

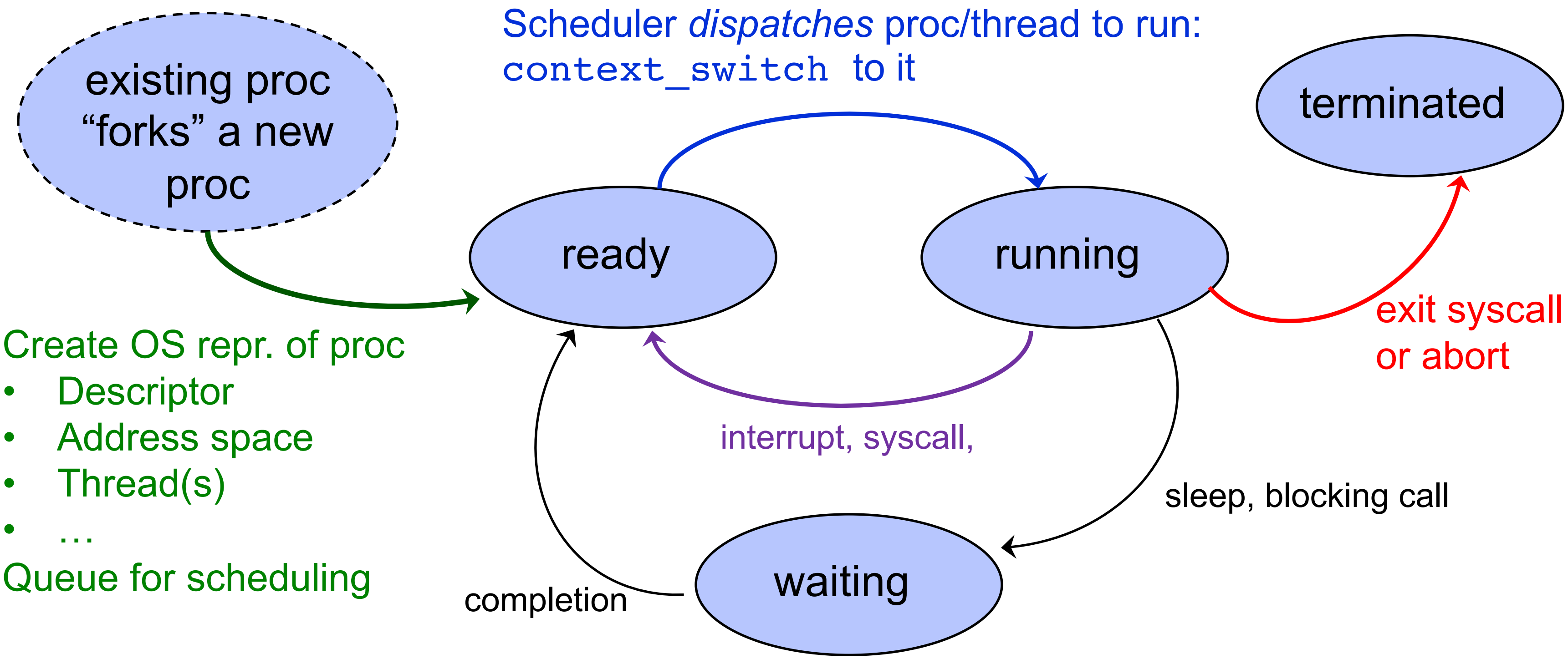


Process Control Block

Context Switch



Lifecycle of a process / thread



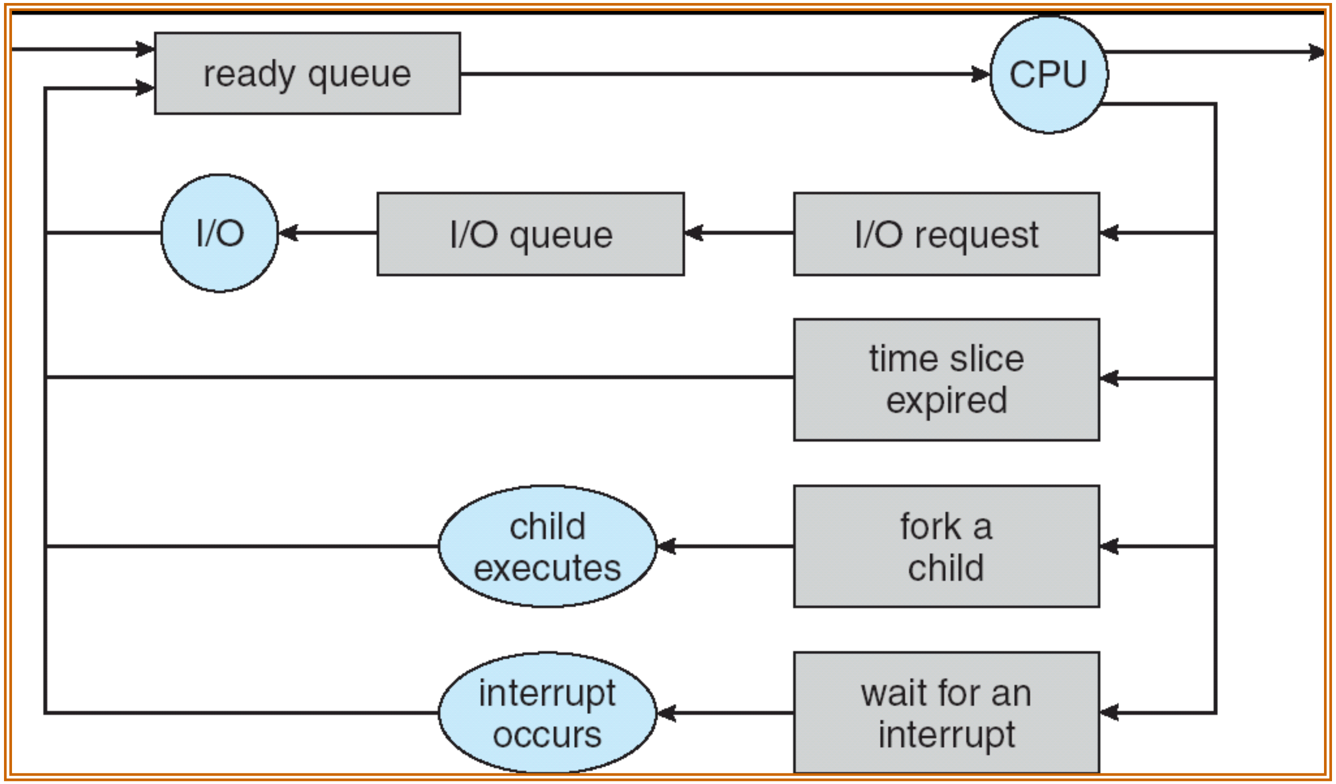
Create OS repr. of proc
- Descriptor
- Address space
- Thread(s)
- …

Queue for scheduling

- **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

Scheduling: All About Queues



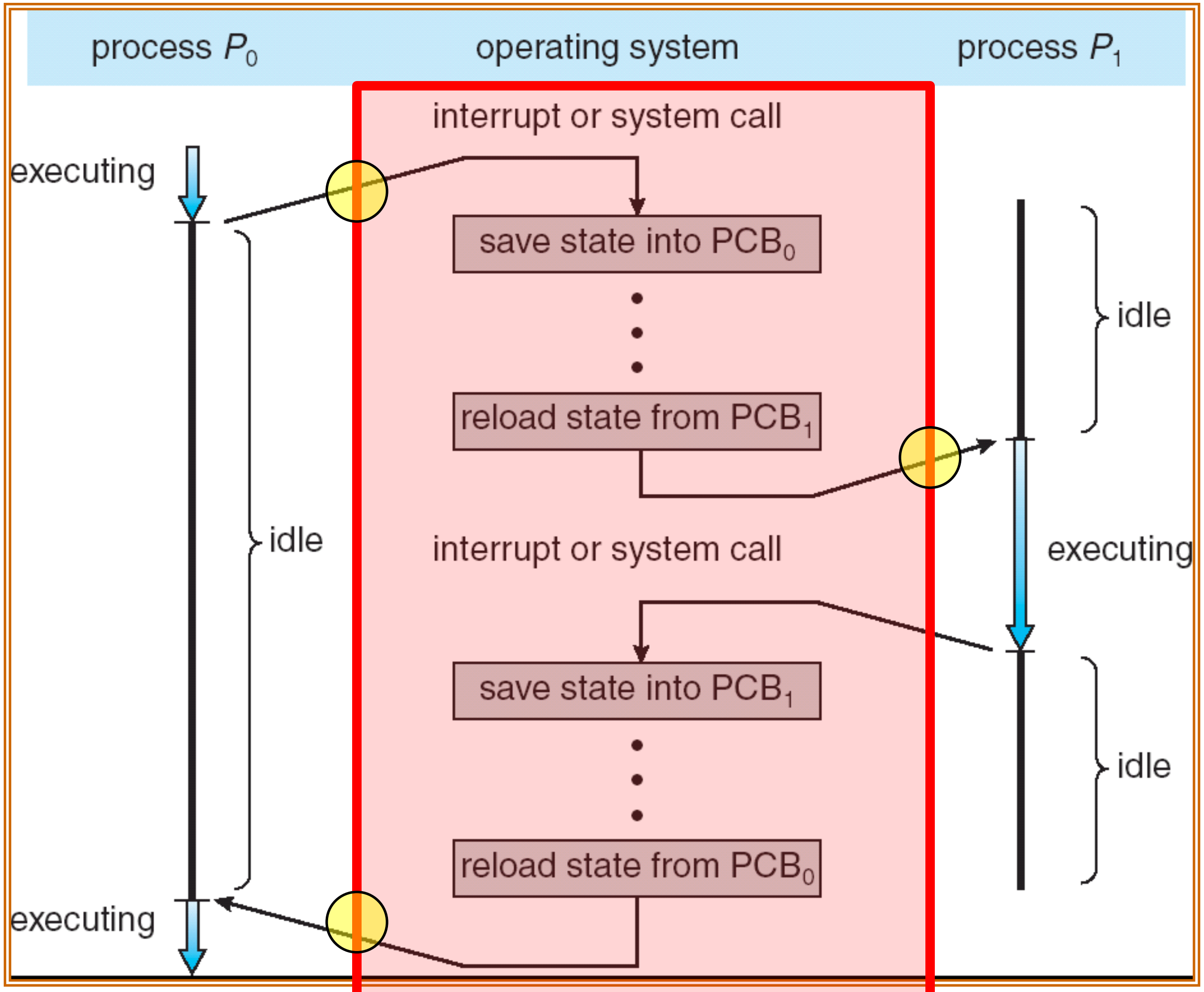
- **PCBs move from queue to queue**
- **Scheduling: which order to remove from queue**
  - **Much more on this soon**

Recall: Scheduler

```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

Context Switch



Privilege Level: 3 - user

Privilege Level: 0 - sys

Privilege Level: 3 - user

Process Control Block

- **Kernel representation of each process**
  - **Status (running, ready, blocked)**
  - **Register state (if not running)**
  - **Thread control block(s)**
  - **Process ID**
  - **Execution time**
  - **Address space**
  - **Open files, etc**

How is this represented?

- **Scheduler maintains a data structure of PCBs**
- ***Scheduling algorithm:* Which process should the OS run next?**

Process Creation

- Allocate and initialize Process object
- Allocate and initialize kernel thread mini-stack and associated Thread object
- Allocate and initialize page table for process
- Load code and static data into user pages
- Build initial User Stack
  - Initial register contents
- Schedule (post) process thread for execution
- …
- Eventually switch to user thread …

Project 1

- Several lists of various types

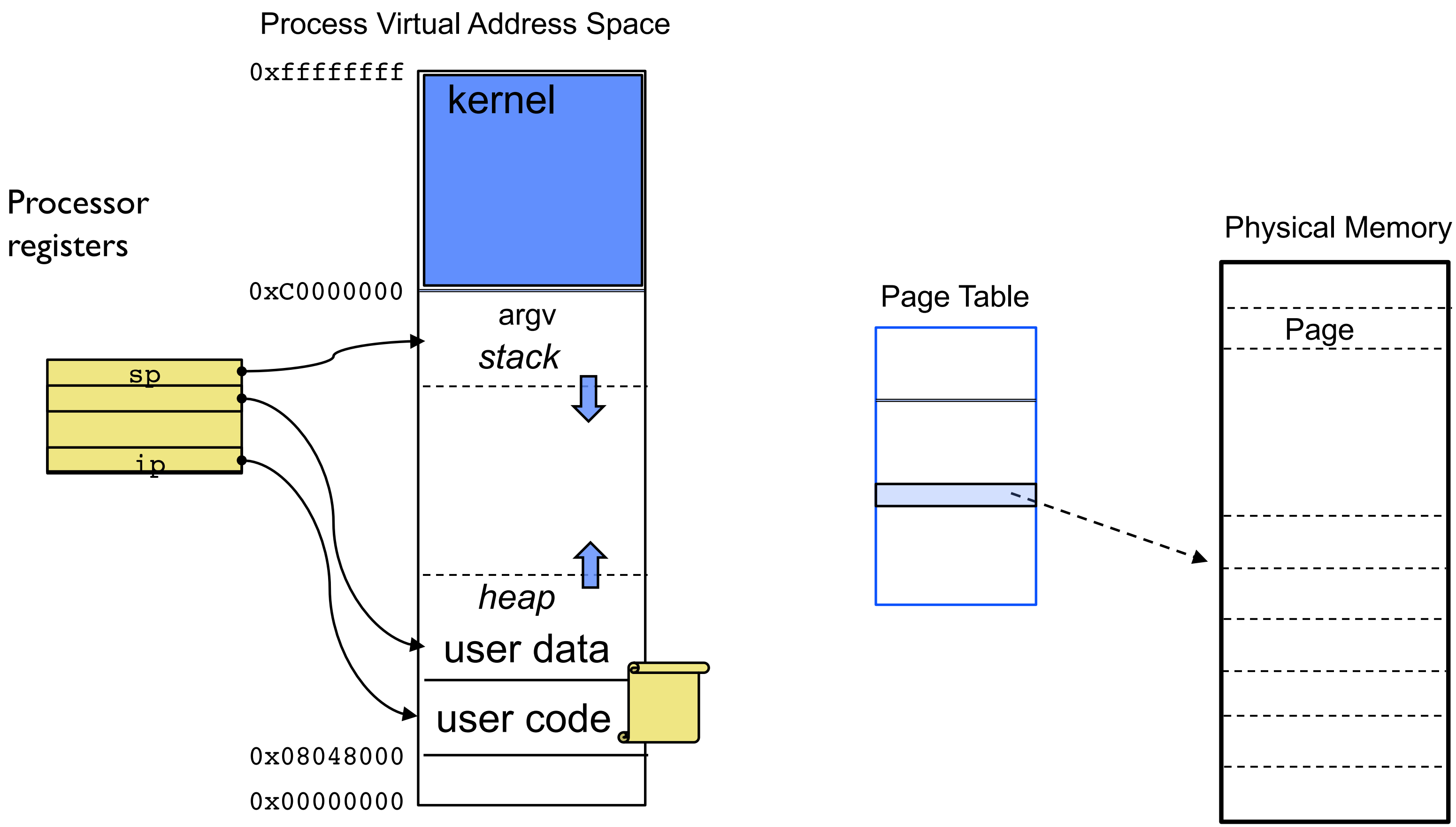
Pintos: process.c, thread.c

Understanding "Address Space”

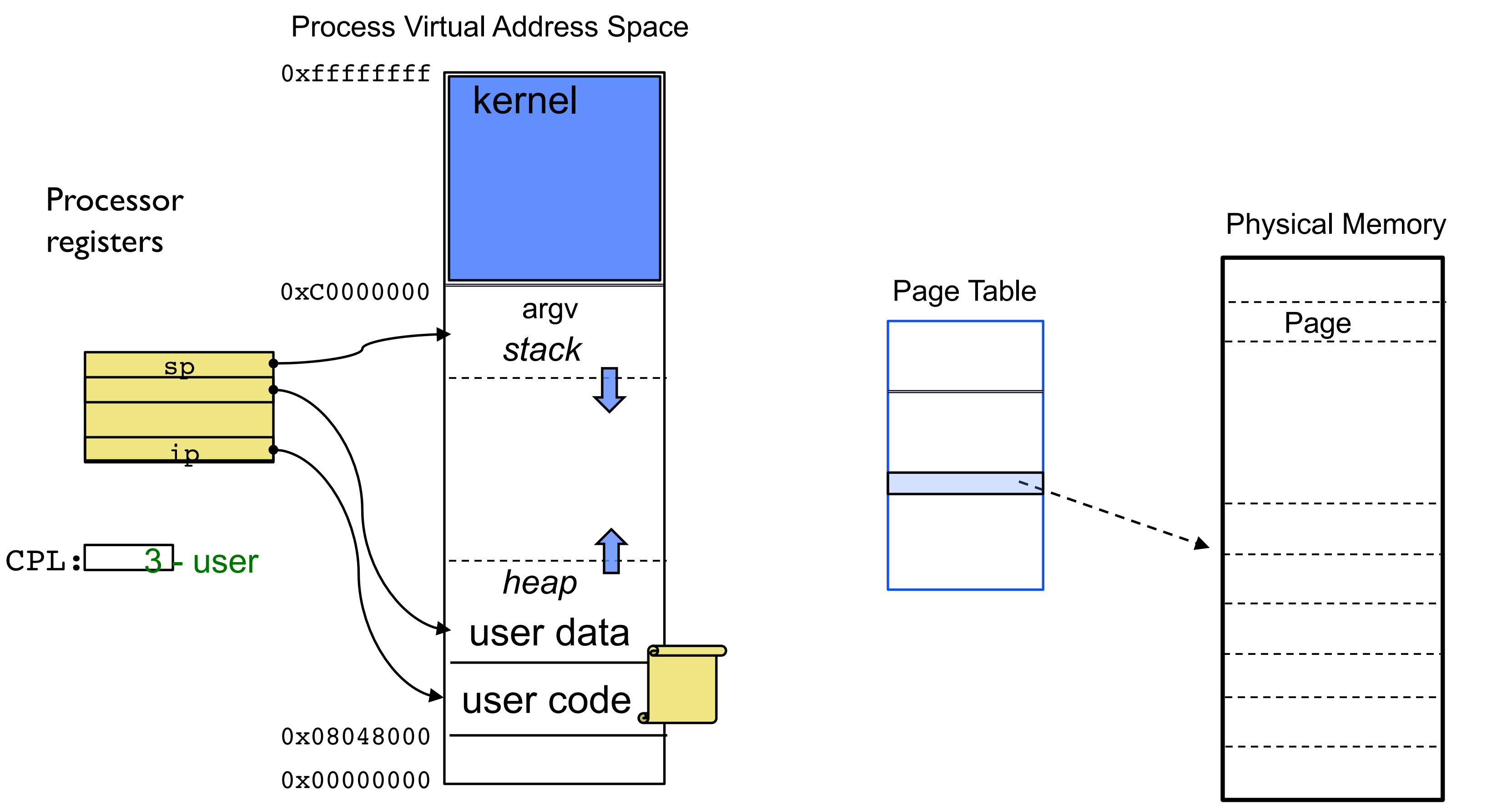
- Page table is the primary mechanism
- Privilege Level determines which regions can be accessed
  – Which entries can be used
- System (PL=0) can access all, User (PL=3) only part
- Each process has its own address space
- The “System” part of all of them is the same

=> All system threads share the same system address space and same memory

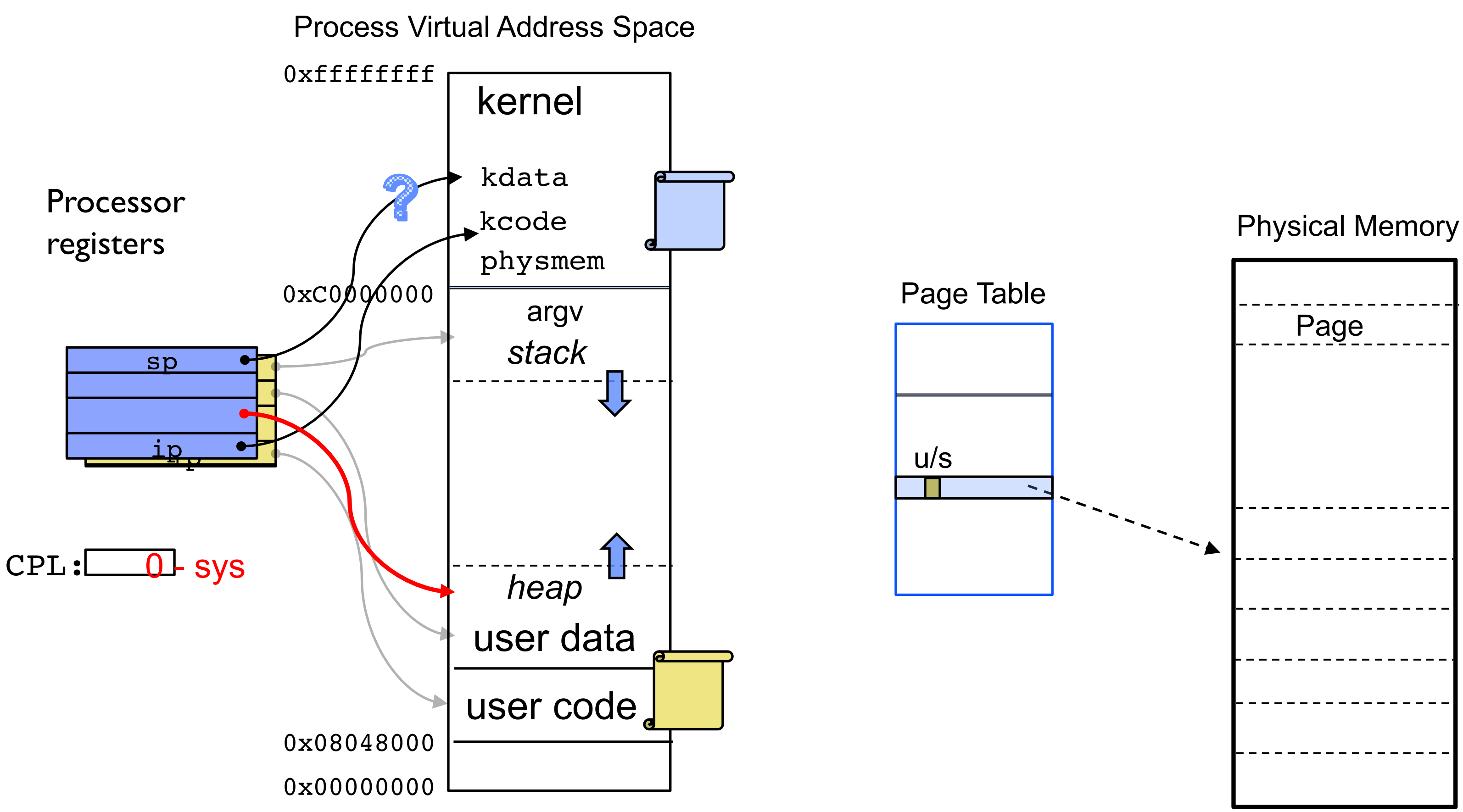
User Process View



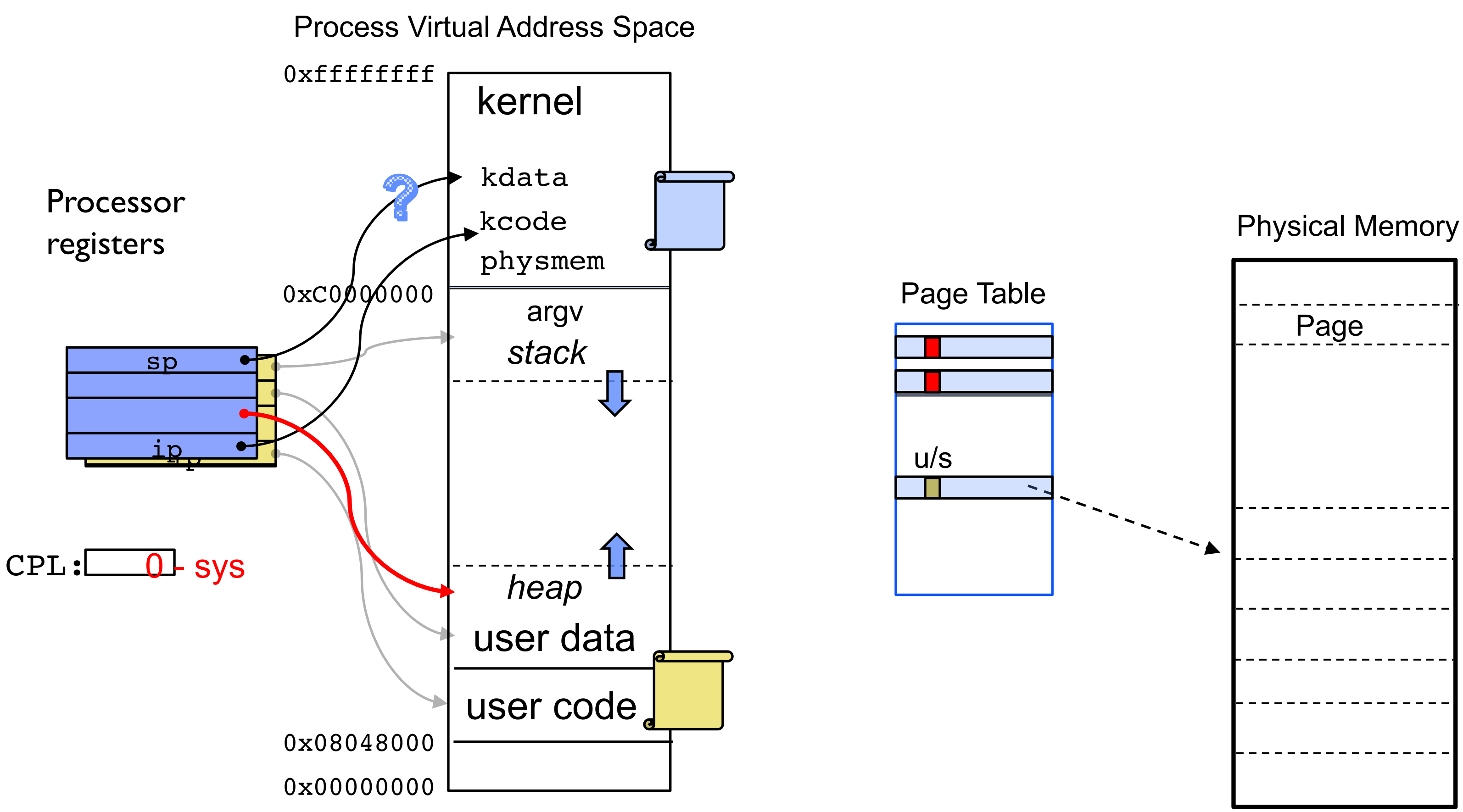
Processor Mode (Privilege Level)



User → Kernel: PL = 0

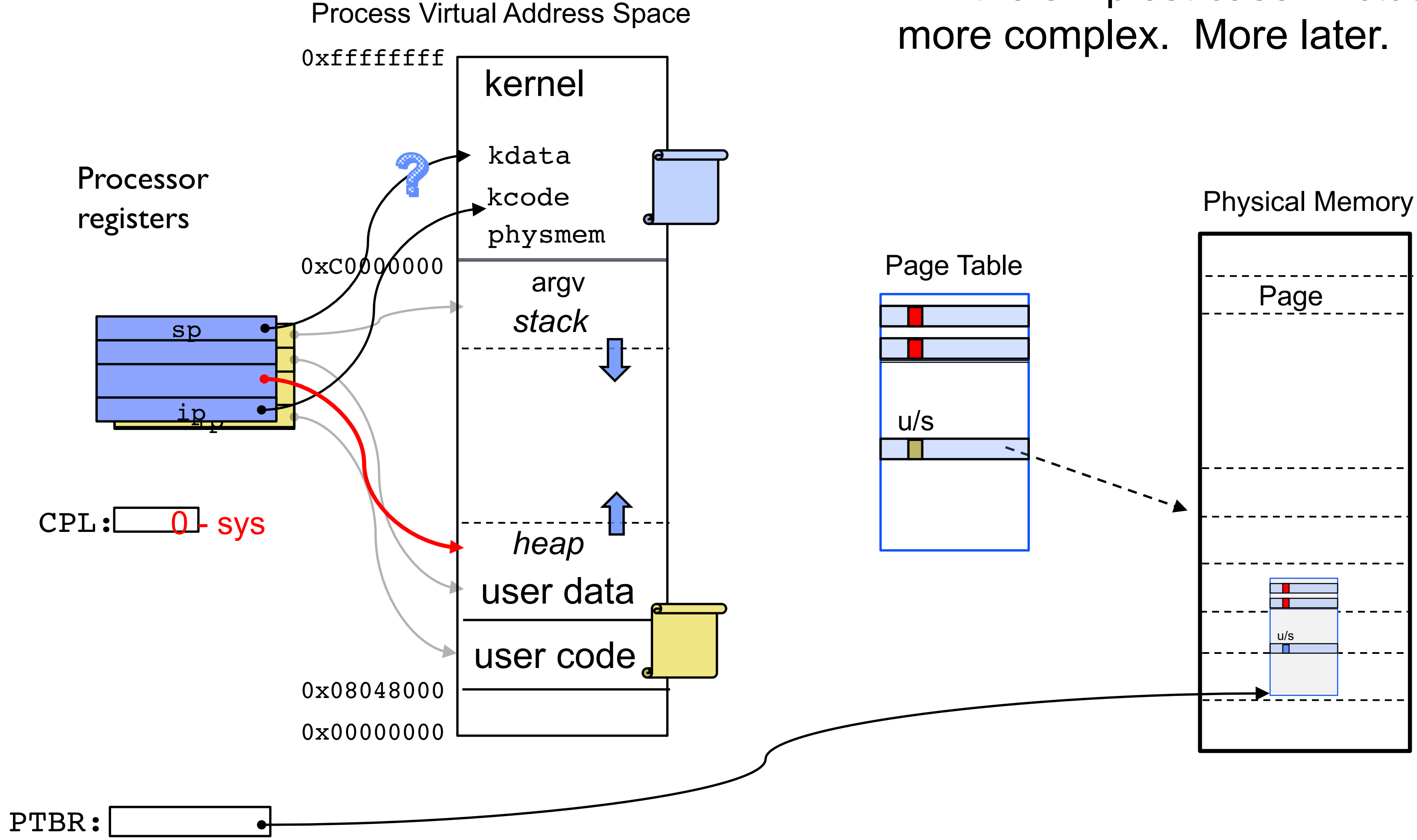


Page Table enforces PL

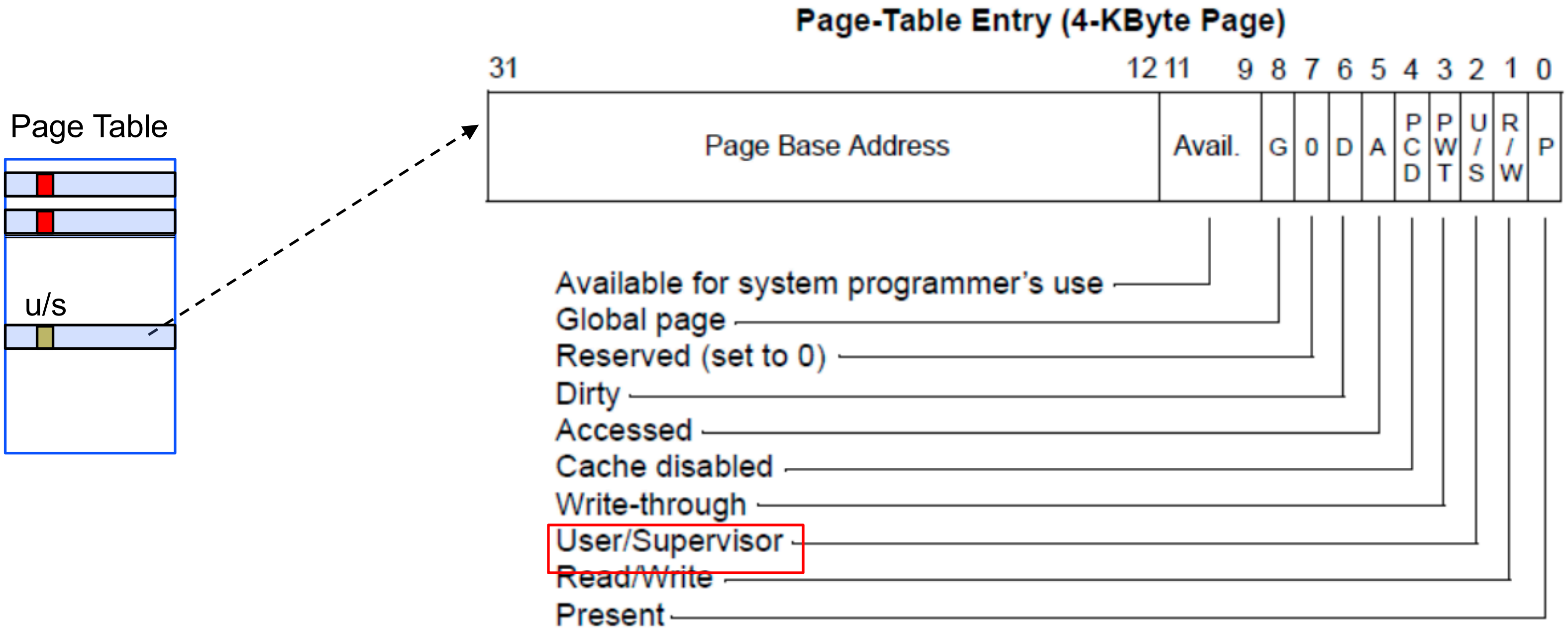


Page Table resides in memory*

* In the simplest case. Actually more complex. More later.



x86 (32-bit) Page Table Entry



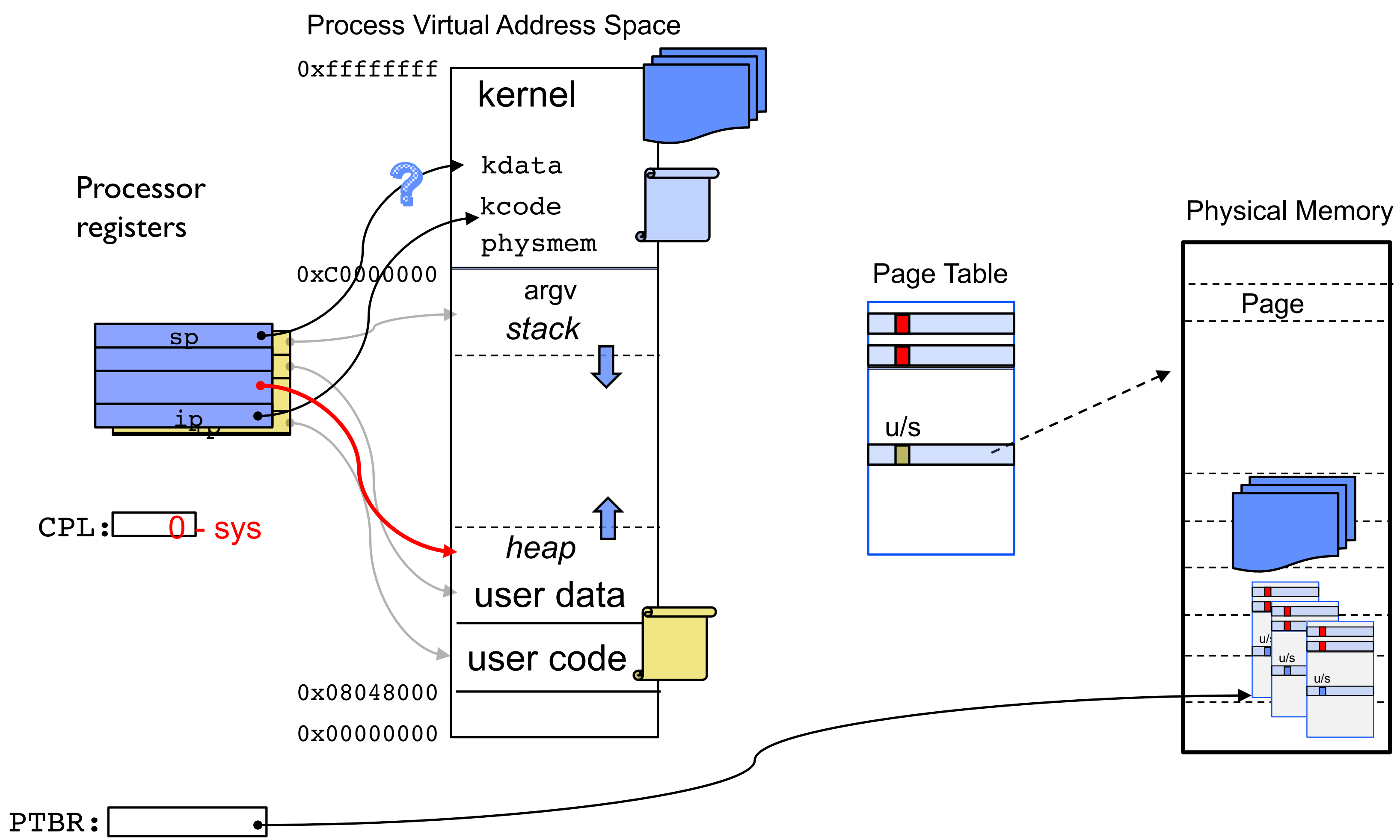
- **Controls many aspects of access**
- **Later – discuss page table organization**
  - **For 32 (64?) bit VAS, how large? vs size of memory?**
  - **Use sparsely. Very very fast HW access**

Pintos: page_dir.c

Kernel Portion of Address Space

- Contains the kernel code
  - Loaded when the machine booted
- Explicitly mapped to physical memory
  - OS creates the page table
- Used to contain all kernel data structures
  - List of all the processes and threads
  - The page tables for those processes
  - Other system resources (files, sockets, ttys, …)
- Also contains (little) stacks for "kernel threads"
  - Early OS design serviced all processes on a single execution thread
    - » Event driven programming
  - Today: Each Process Thread supported by (little) Kernel Thread

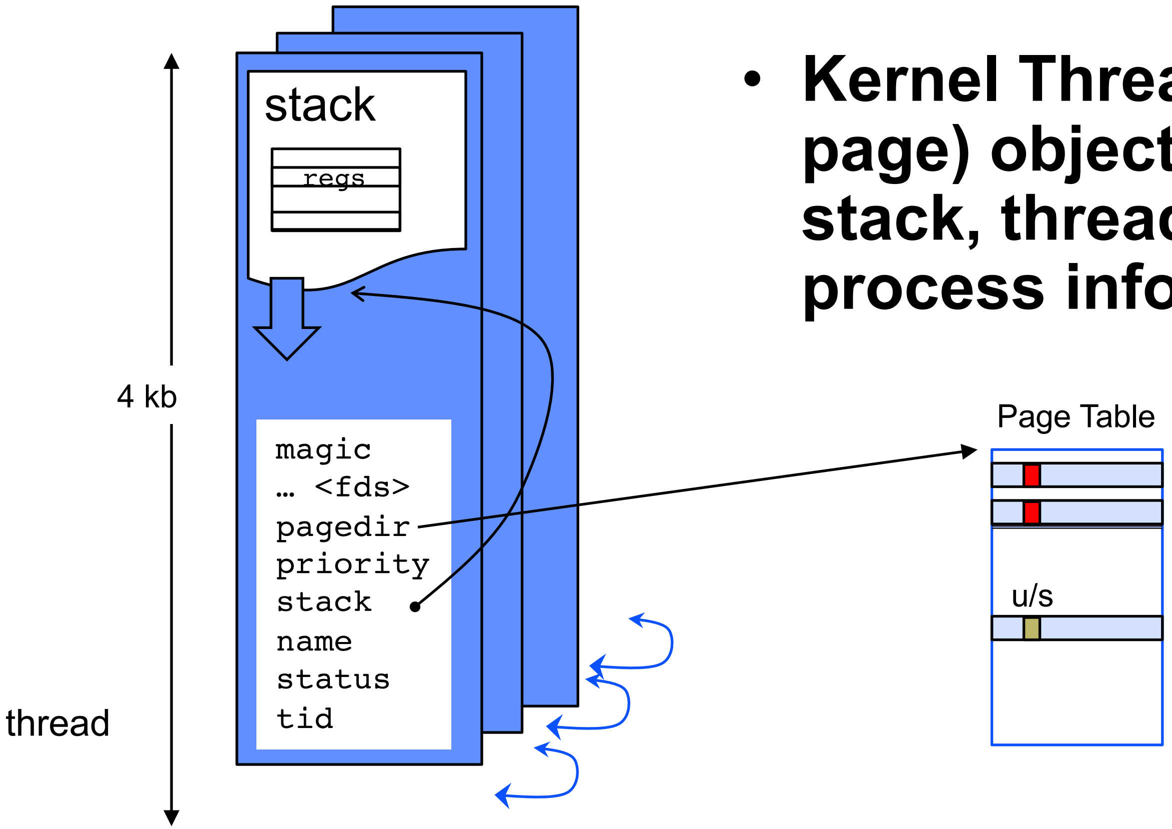
1 Kernel Code, many Kernel "stacks"



From Machine Structure to OS Data Structures

- Traditional Unix, etc. design maintains a Process Control Block (PCB) per process
- Each with a Thread Control Block (TCB) per thread of that process
- Today, assume single thread per process
  - PINTOS model
  - Linux also organized around threads with “groups of threads” associated with a process

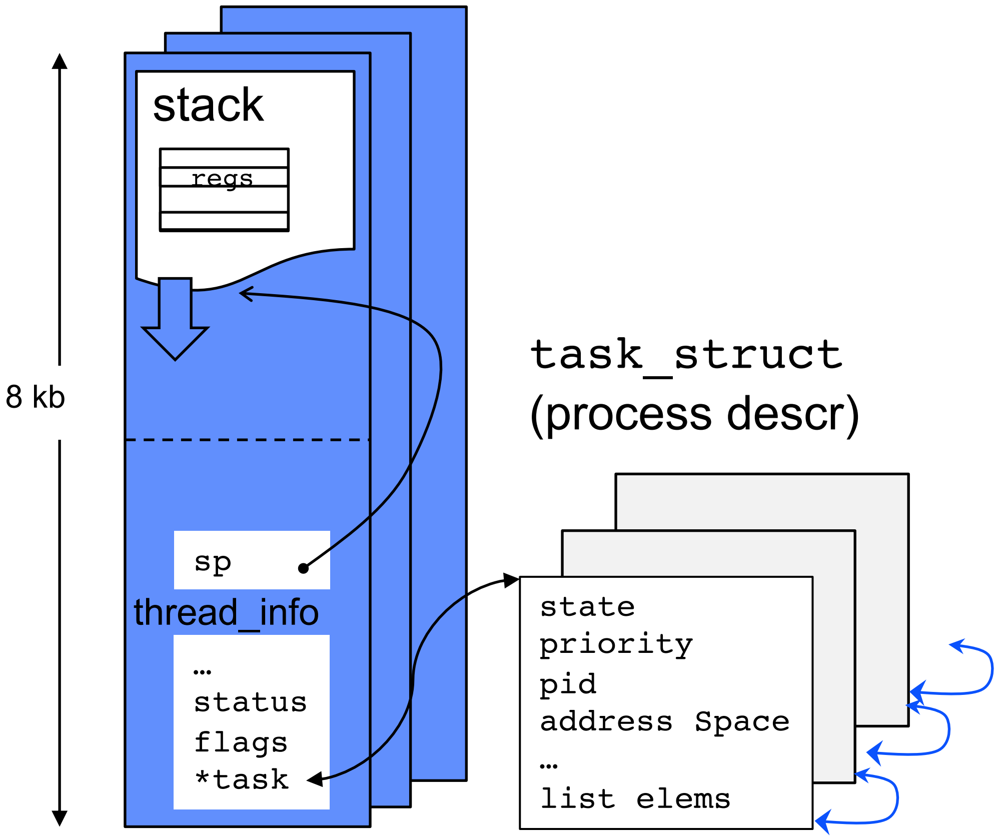
PINTOS Thread



- **Kernel Thread as 4 kb (2 page) object containing stack, thread info, process info**

Pintos: thread.c

Linux "Task"



- **Kernel Thread as 8 kb (2 page) object containing stack and thread information + process decriptor**

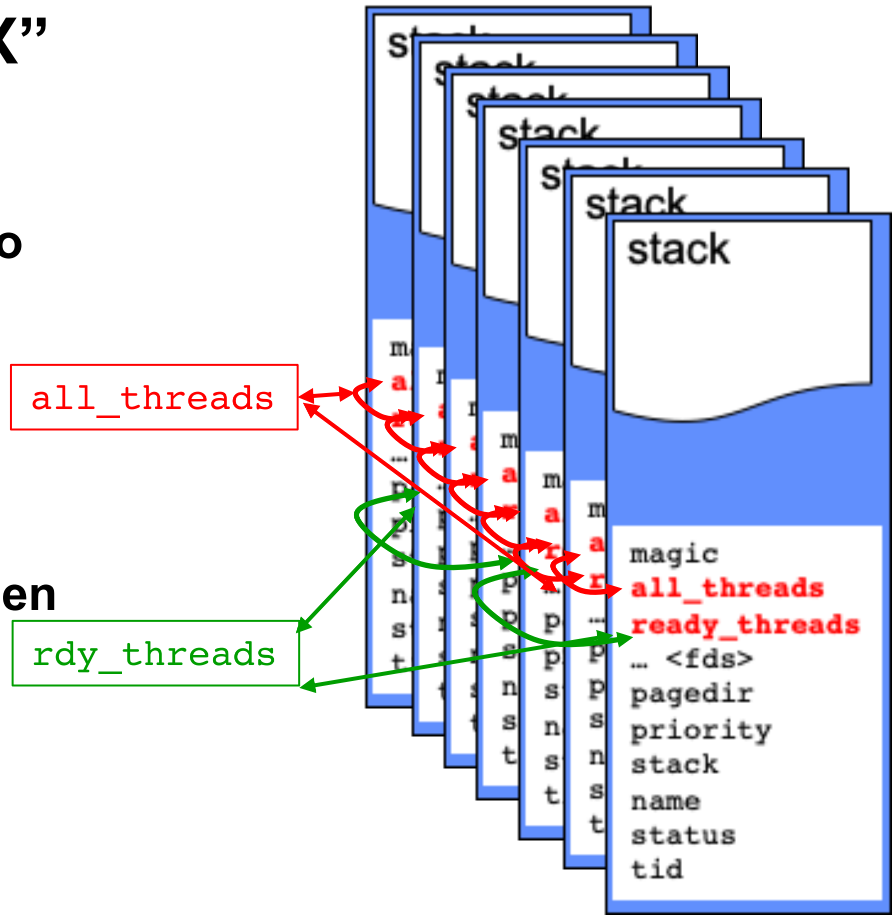
Process Creation

- Allocate and initialize Process object
- Allocate and initialize kernel thread mini-stack and associated Thread object
- Allocate and initialize page table for process
  - Referenced by process object
- Load code and static data into user pages
- Build initial User Stack
  - Initial register contents, argv, …
- Schedule (post) process thread for execution
- …
- Eventually *switch* to user thread …

- Several lists of various types

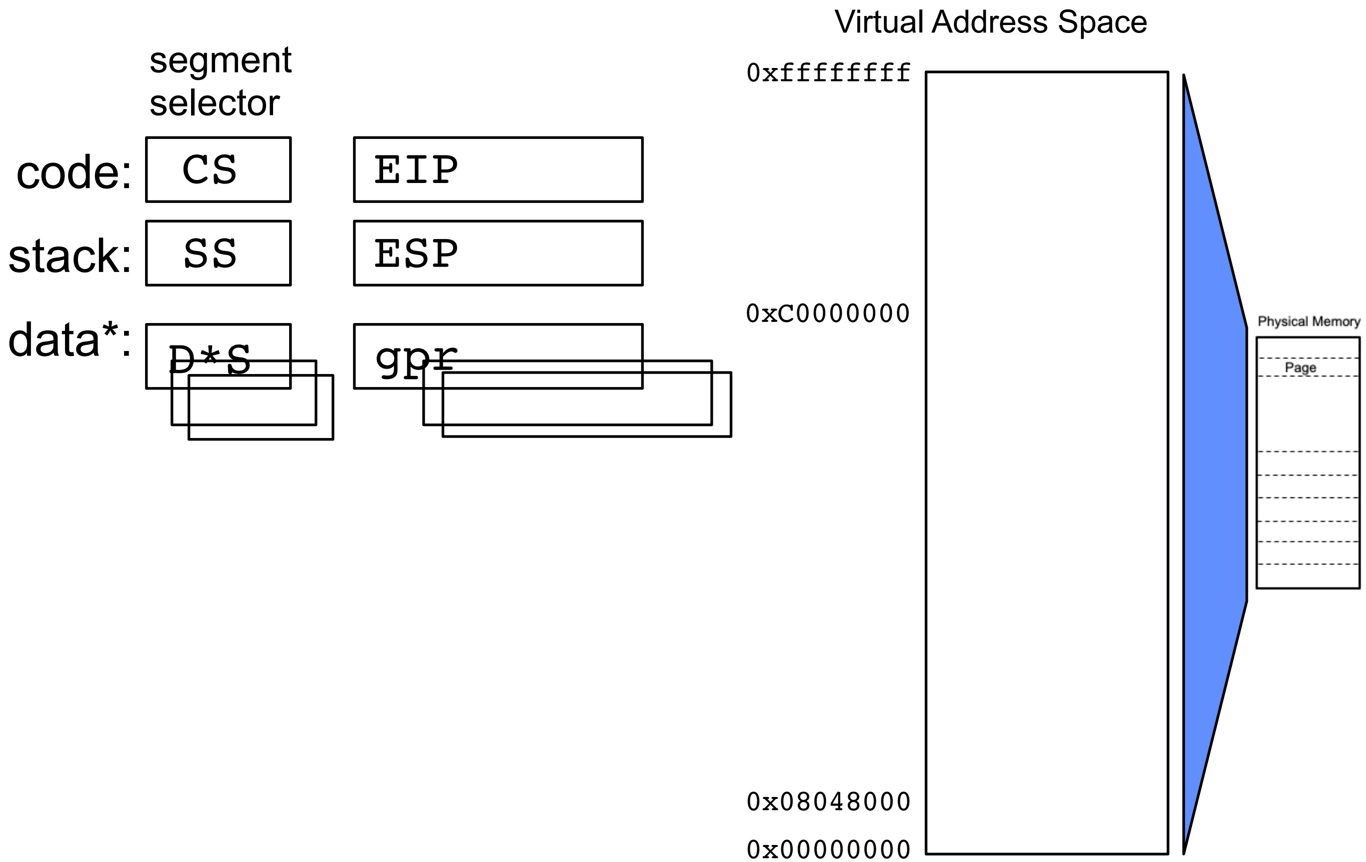
Aside: Polymorphic lists in C

- **Many places in the kernel need to maintain a "list of X"**
  - **This is tricky in C, which has no polymorphism**
  - **Essentially adding an *interface* to a package (ala Go)**
- **In Linux and Pintos this is done by embedding a `list_elem` in the struct**
  - **Macros allow shift of view between object and list**
  - **You'll practice in HW1 – before getting into PINTOS**

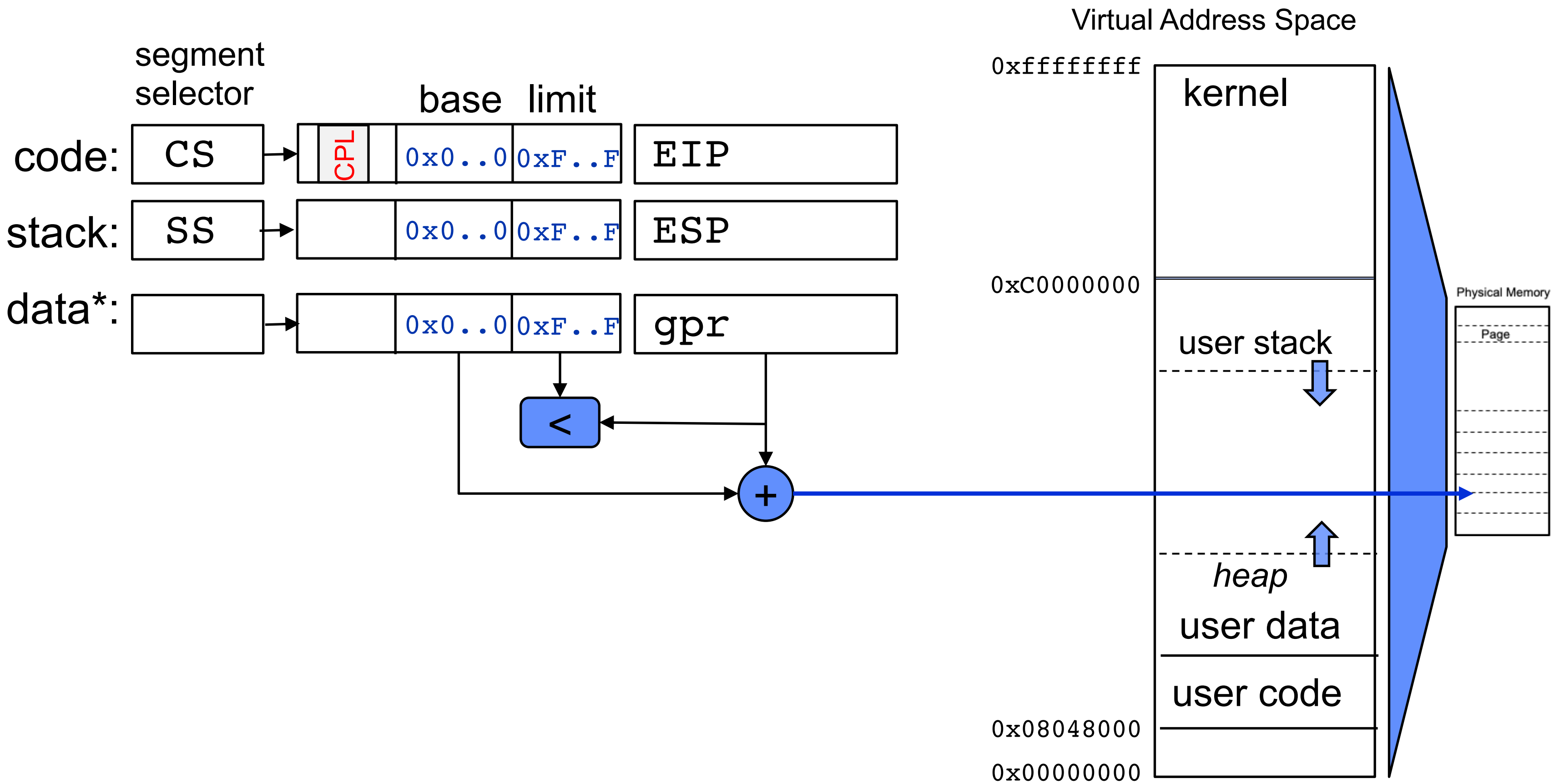


Pintos: list.c

Bit of x86 thread/process/VAS management



Bit of x86 thread/process/VAS management

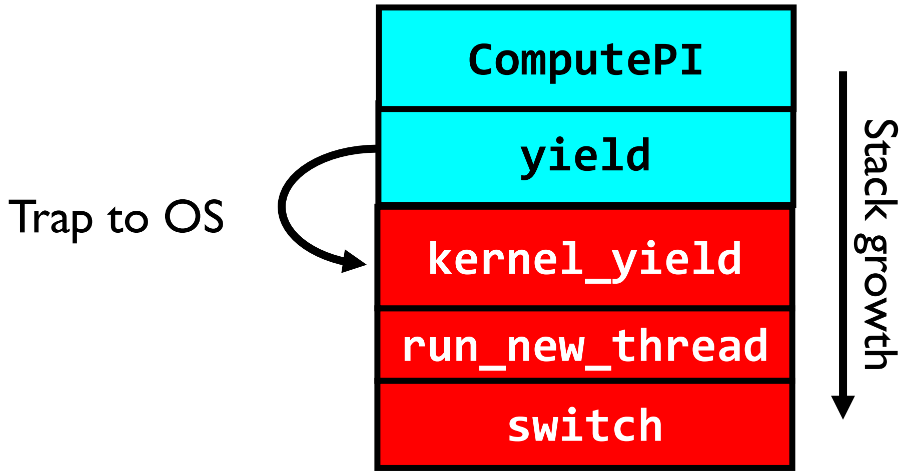


Pintos: loader.h

Recall: 3 types of U→K Mode Transfer

- **Syscall**
  - Process requests a system service, e.g., exit
  - Like a function call, but "outside" the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- **Interrupt**
  - External asynchronous event triggers context switch
  - eg. Timer, I/O device
  - Independent of user process
- **Trap**
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …
- **All 3 *exceptions* are an UNPROGRAMMED CONTROL TRANSFER**
  - Where does it go? (To handler specified in interrupt vector)
  - Are interrupts enabled or disabled when get there?

Stack for Thread Transition



Cyan = User Stack; Red = Kernel Stack

```
run_new_thread() {
  newThread = PickNewThread();
  switch(curThread, newThread);
  ThreadHouseKeeping(); /* Do any cleanup */
}
```

Scheduling: Policy Decision

Hardware context switch support

- **Syscall/Intr (U → K)**
  - **PL 3 → 0;**
  - **TSS ← EFLAGS, CS:EIP;**
  - **SS:SP ← k-thread stack (TSS PL 0);**
  - **push (old) SS:ESP onto (new) k-stack**
  - **push (old) eflags, cs:eip, <err>**
  - **CS:EIP ← <k target handler>**
- **Then**
  - ***Handler then saves other regs, etc***
  - ***Does all its works, possibly choosing other threads, changing PTBR (CR3)***

  - **kernel thread has set up user GPRs**
- **iret (K → U)**
  - **PL 0 → 3;**
  - **Eflags, CS:EIP ← popped off k-stack**
  - **SS:SP ← user thread stack (TSS PL 3);**

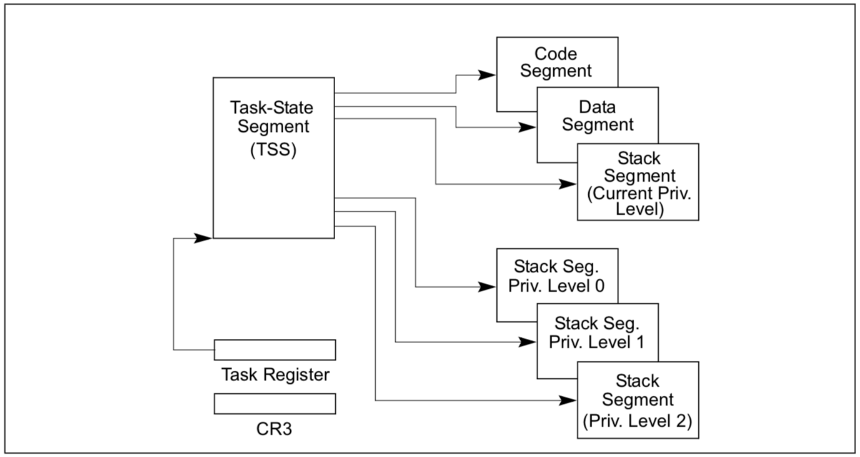
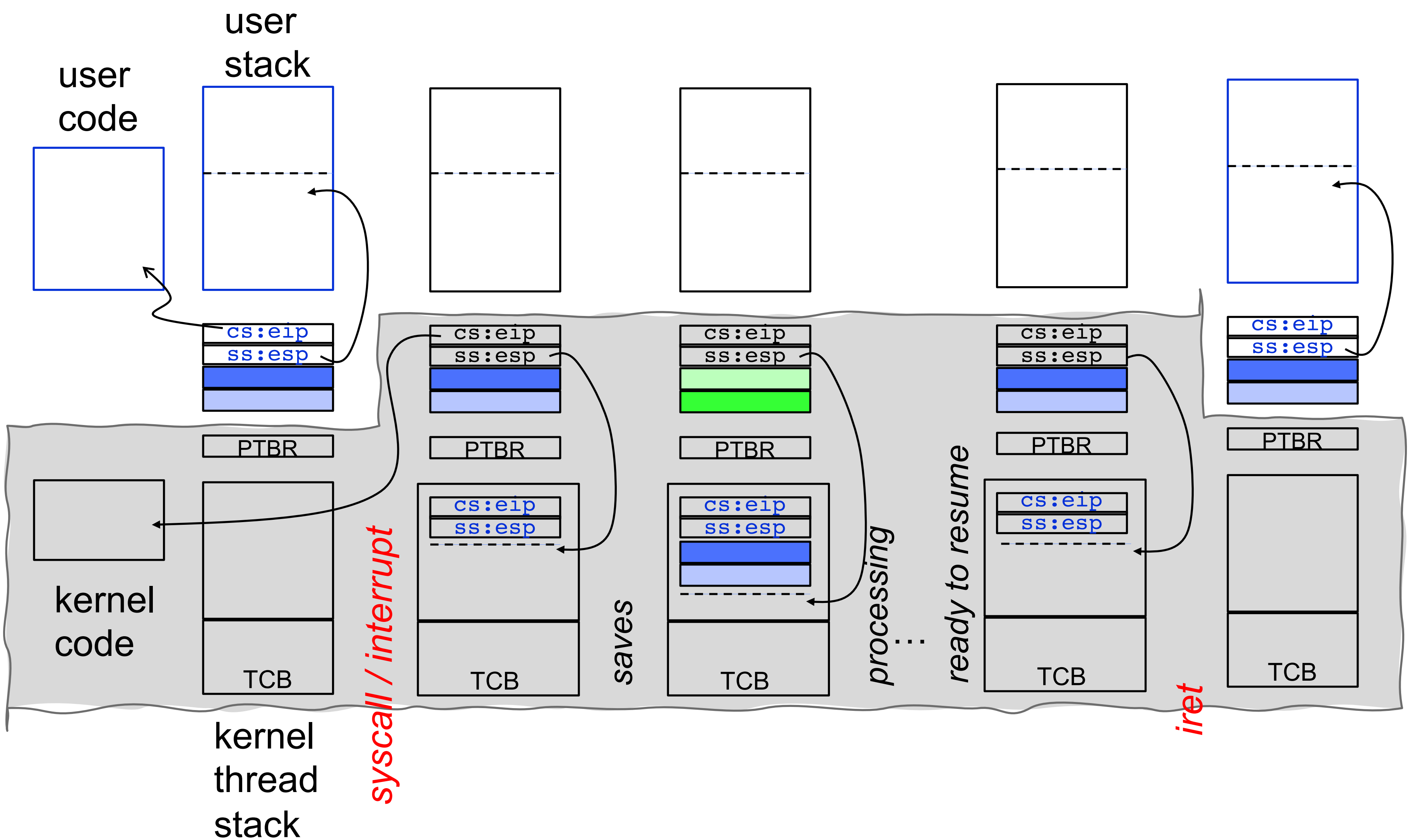


Figure 7-1. Structure of a Task

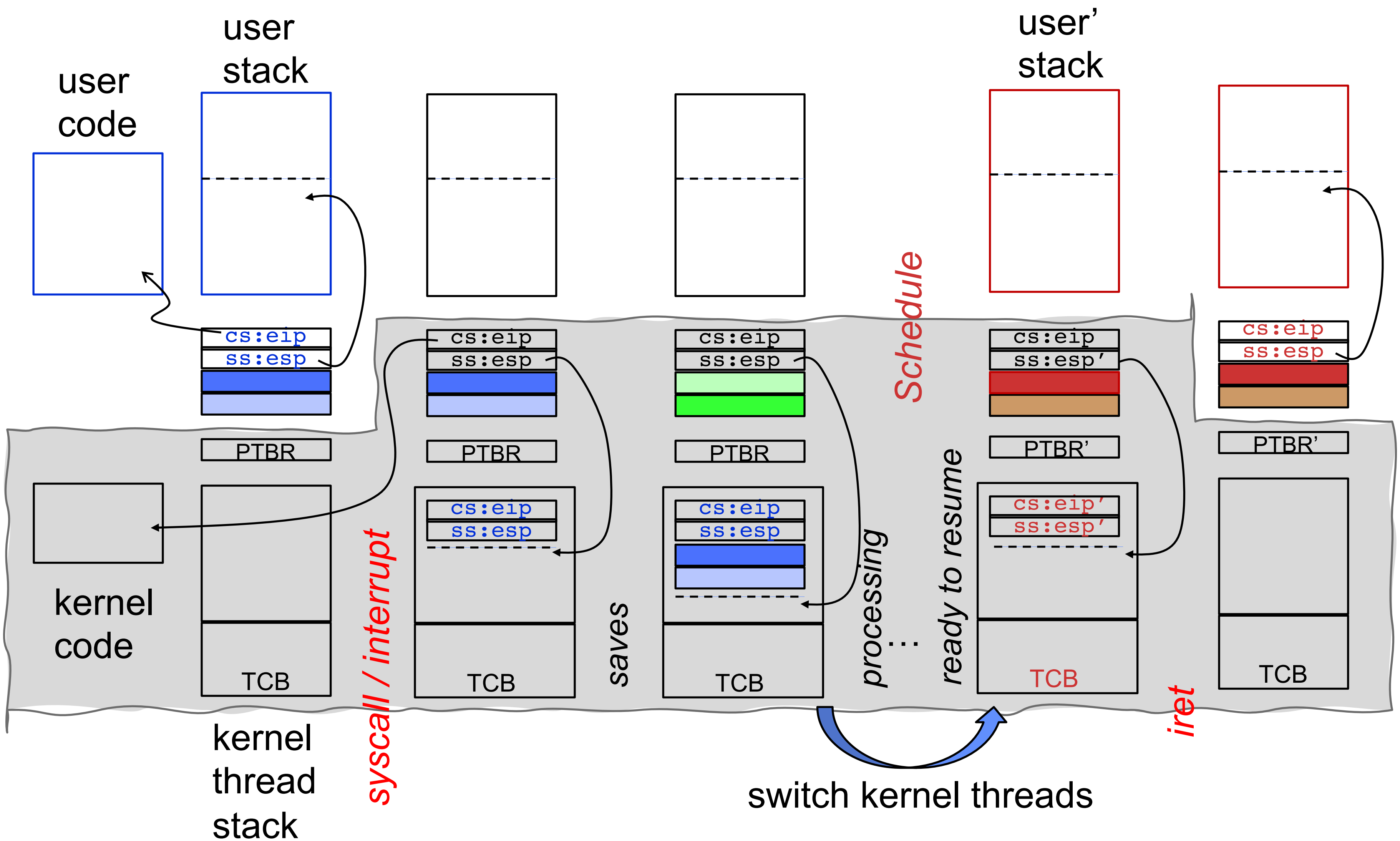
Pintos: tss.c, intr-stubs.S

pg 2,942 of 4,922 of x86 reference manual

Context Switch – in pictures



Context Switch – Scheduling



Pintos: switch.S

Context Switch between K-threads

```
Switch(tCur,tNew) {
   /* Unload old thread */
   TCB[tCur].regs.r7 = CPU.r7;
           …
   TCB[tCur].regs.r0 = CPU.r0;
   TCB[tCur].regs.sp = CPU.sp;
   TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

   /* Load and execute new thread */
   CPU.r7 = TCB[tNew].regs.r7;
           …
   CPU.r0 = TCB[tNew].regs.r0;
   CPU.sp = TCB[tNew].regs.sp;
   CPU.retpc = TCB[tNew].regs.retpc;
   return; /* Return to CPU.retpc */
}
```

Concurrency

- But, … ???
- With all these threads in the kernel, won't they step on each other?
  – For example, while one is loading a program, other threads should run …
  – Processes are isolated from each other, but all the threads in the kernel share the kernel address space, memory, data structures
- We will study synchronization soon
- The kernel controls whether hardware interrupts are enabled or not
  – Disabled on entry, selectively enable
  – Atomic operations, …

Dispatch Loop

```
Loop {
  RunThread();
  ChooseNextThread();
  SaveStateOfCPU(curTCB);
  LoadStateOfCPU(newTCB);
}
```

- *Conceptually* all the OS executes
- Infinite Loop
  - When would we ever "exit?"
  - Can we assume some thread is always ready?

Dispatch Loop

```
Loop {
  RunThread();
  ChooseNextThread();
  SaveStateOfCPU(curTCB);
  LoadStateOfCPU(newTCB);
}
```

How to run a new thread?

- Load thread's registers into CPU
- Load its environment (address space, if in different process)
- Jump to thread's PC

How does dispatch loop get control again?

- Thread returns control voluntarily – `yield`, I/O
- External events: thread is *preempted*

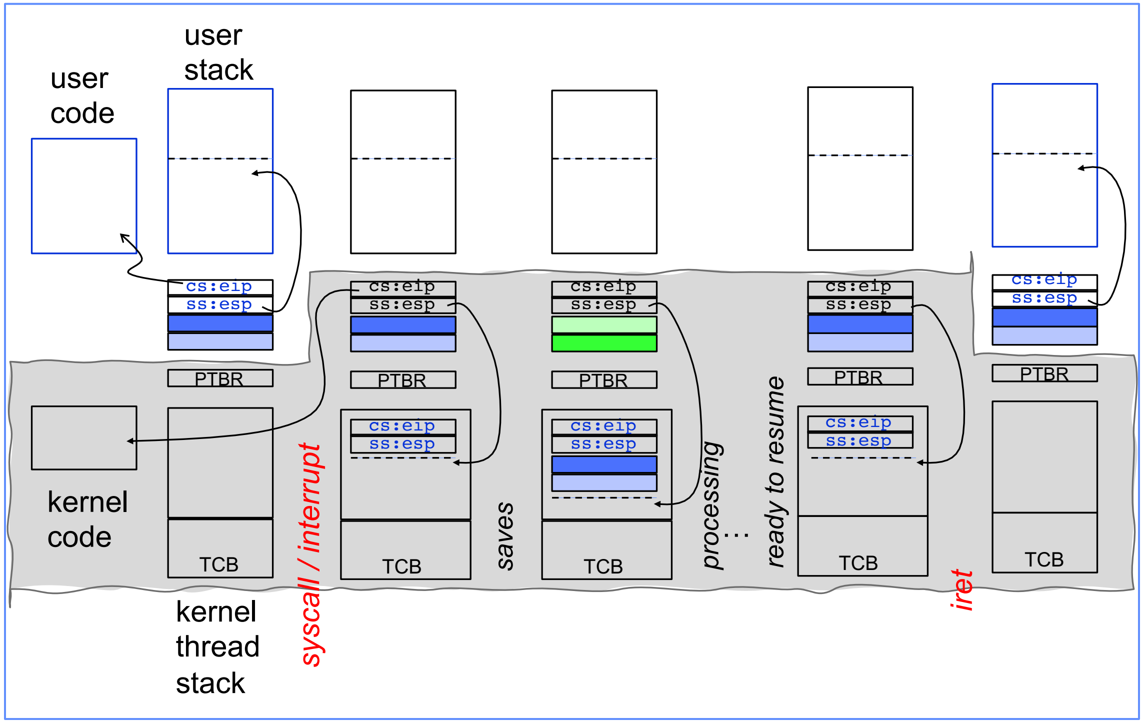
Thread Operations in Pintos

- `thread_create(name, priority, func, args)`
  – Create a new thread to run func(args)
- `thread_yield()`
  – Relinquish processor voluntarily
- `thread_join(thread)`
  – Wait (put in queue) until thread exits, then return
- `thread_exit`
  – Quit thread and clean up, wake up joiner if any

More later, incl. synch ops

Peer question

- **Which kind of thread is performing these operations?**
  - **"user level thread" with its full stack and user address space?**
  - **"system proxy thread" for a "user level thread"**



Tout Question

- **Process is an instance of a program executing.**
  - **The fundamental OS responsibility**
- **Processes do their work by processing and calling file system operations**

- **Are their any operations on processes themselves?**

- **exit ?**

pid.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
  pid_t pid = getpid();   /* get current processes PID */

  printf("My pid: %d\n", pid);

  exit(0);
}
```

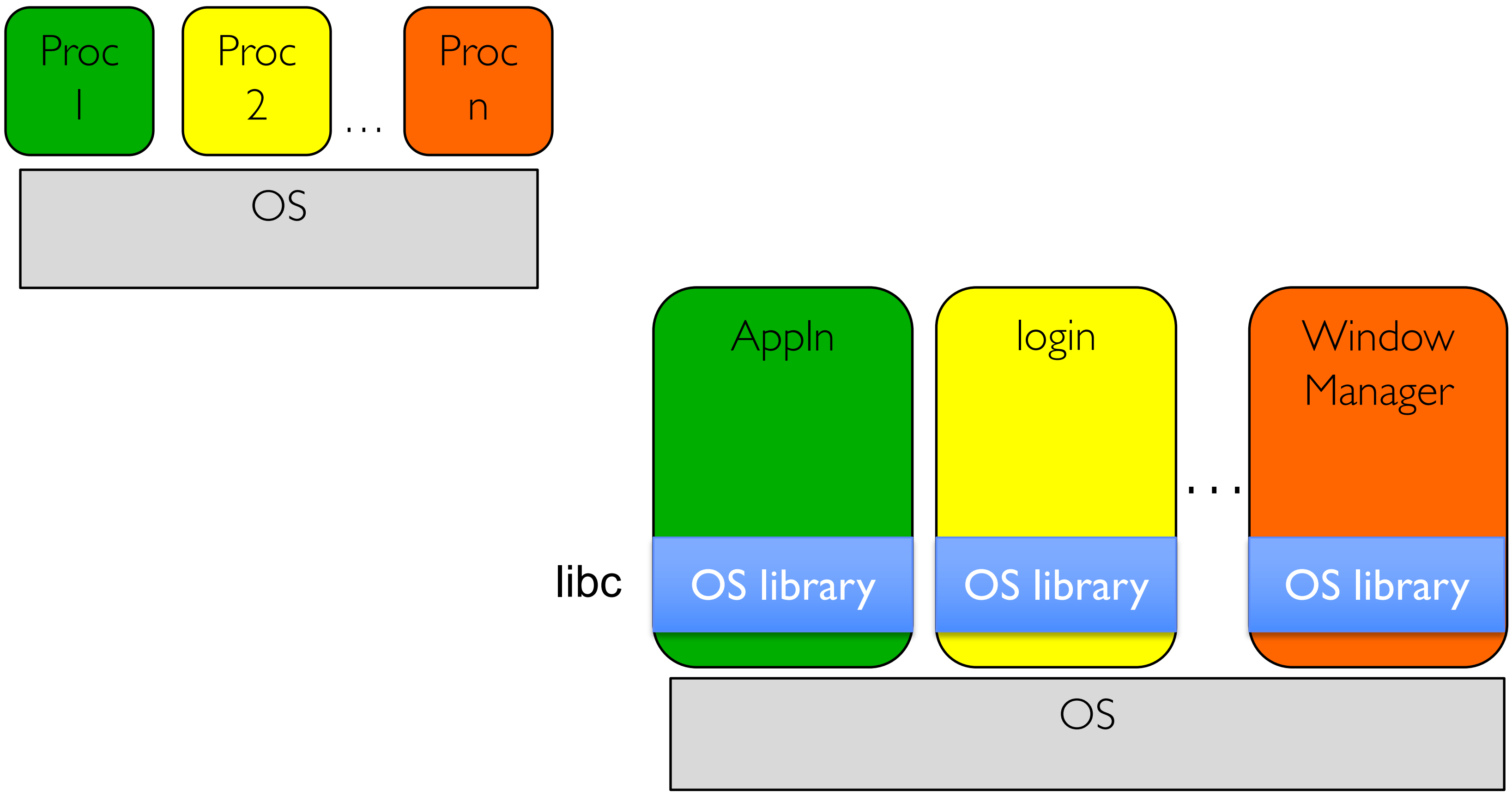
ps anyone?

Can a process create a process ?

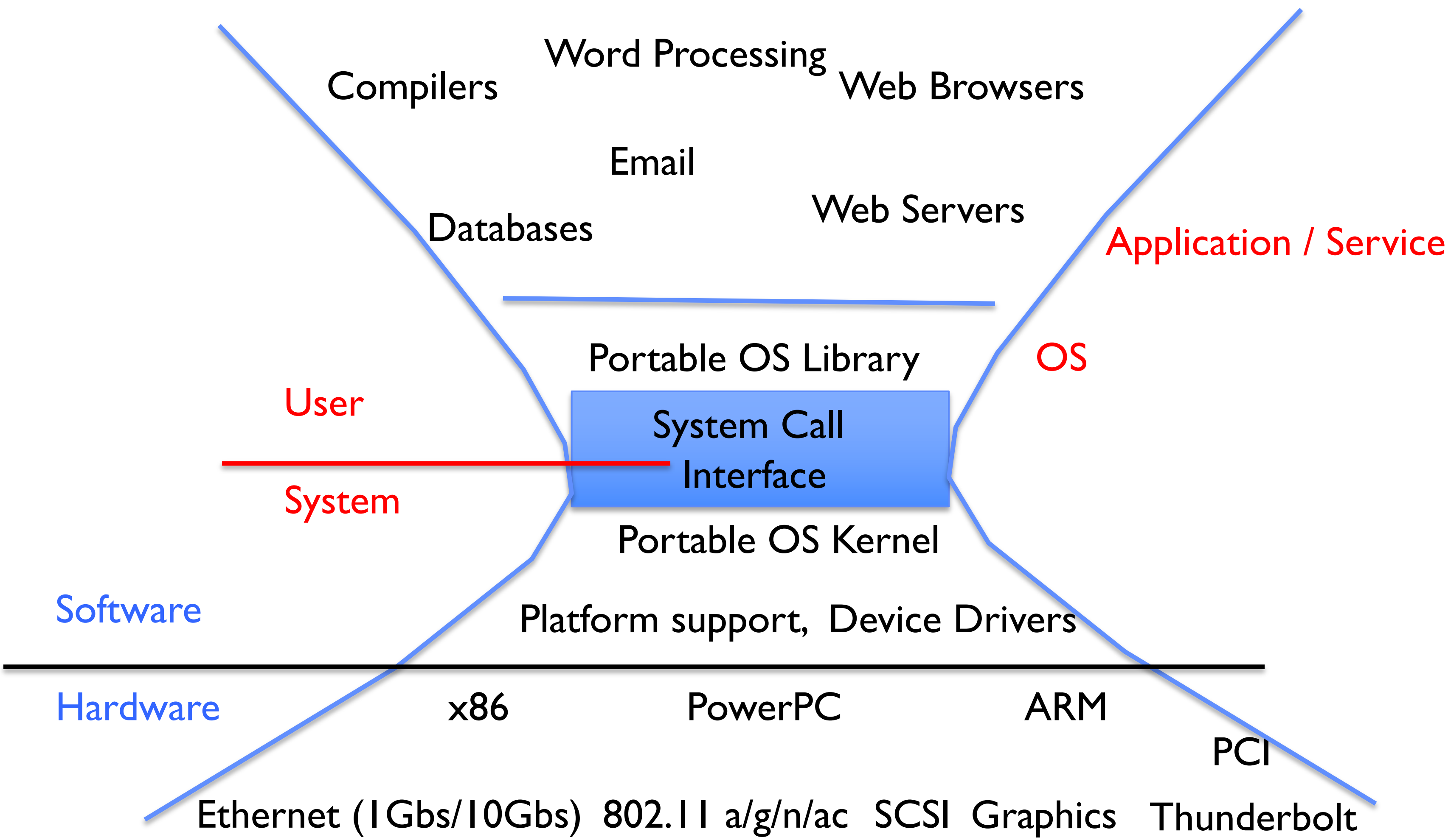
- Yes
- Fork creates a copy of process
- What about the program you want to run?

Break

OS Run-Time Library



A Narrow Waist



POSIX/Unix

- Portable Operating System Interface [X?]
- Defines "Unix", derived from AT&T Unix
  - Created to bring order to many Unix-derived OSs
- Interface for application programmers (mostly)

System Calls

Application:

```
fd = open(pathname);
```

Library:

```
File *open(pathname) {
    asm code … syscall # into ax
    put args into registers bx, …
    special trap instruction
```

Operating System:

```
get args from regs
dispatch to system func
process, schedule, …
complete, resume process
```

```
    get results from regs
  };
```

```
Continue with results
```

Pintos: userprog/syscall.c, lib/user/syscall.c

SYSCALLs (of over 300)

%eax	Name	Source	%ebx	%ecx	%edx	%esi	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-
23	sys_setuid	kernel/sys.c	uid_t	-	-	-	-
24	sys_getuid	kernel/sched.c	-	-	-	-	-
25	sys_stime	kernel/time.c	int *	-	-	-	-
26	sys_ptrace	arch/i386/kernel/ptrace.c	long	long	long	long	-
27	sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
28	sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
29	sys_pause	arch/i386/kernel/sys_i386.c	-	-	-	-	-
30	sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-

Pintos: syscall-nr.h

Recall: Kernel System Call Handler

- Locate arguments
  - In registers or on user(!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - into user memory

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

Creating Processes

- pid_t fork(); -- copy the current process
  - New process has different pid
- Return value from fork(): pid (like an integer)
  - When > 0:
    - » Running in (original) Parent process
    - » return value is pid of new child
  - When = 0:
    - » Running in new Child process
  - When < 0:
    - » Error! Must handle somehow
    - » Running in original process
- State of original process duplicated in *both* Parent and Child!
  - Address Space (Memory), File Descriptors (covered later), etc…

fork1.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();            /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

fork1.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();            /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();            /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

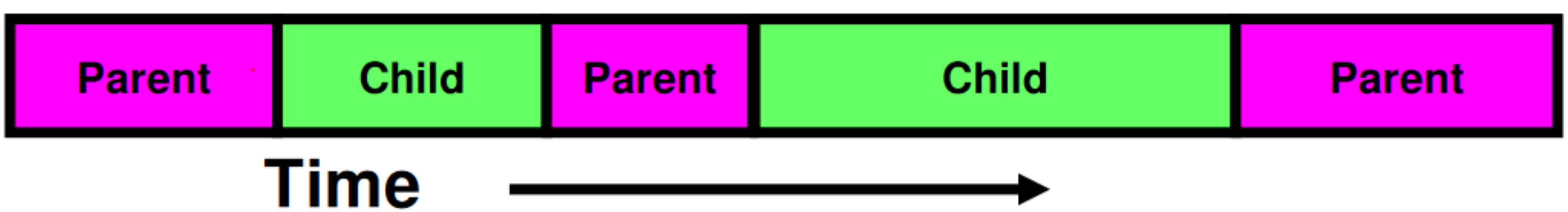
fork_race.c

```c
int i;
cpid = fork();
if (cpid > 0) {
  for (i = 0; i < 10; i++) {
    printf("Parent: %d\n", i);
    // sleep(1);
  }
} else if (cpid == 0) {
  for (i = 0; i > -10; i--) {
    printf("Child: %d\n", i);
    // sleep(1);
  }
}
```

- **What does this print?**
- **Would adding the calls to `sleep` matter?**

Fork “race”

```
int i;
cpid = fork();
if (cpid > 0) {
  for (i = 0; i < 10; i++) {
    printf("Parent: %d\n", i);
    // sleep(1);
  }
} else if (cpid == 0) {
  for (i = 0; i > -10; i--) {
    printf("Child: %d\n", i);
    // sleep(1);
  }
}
```



Process Management

- `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

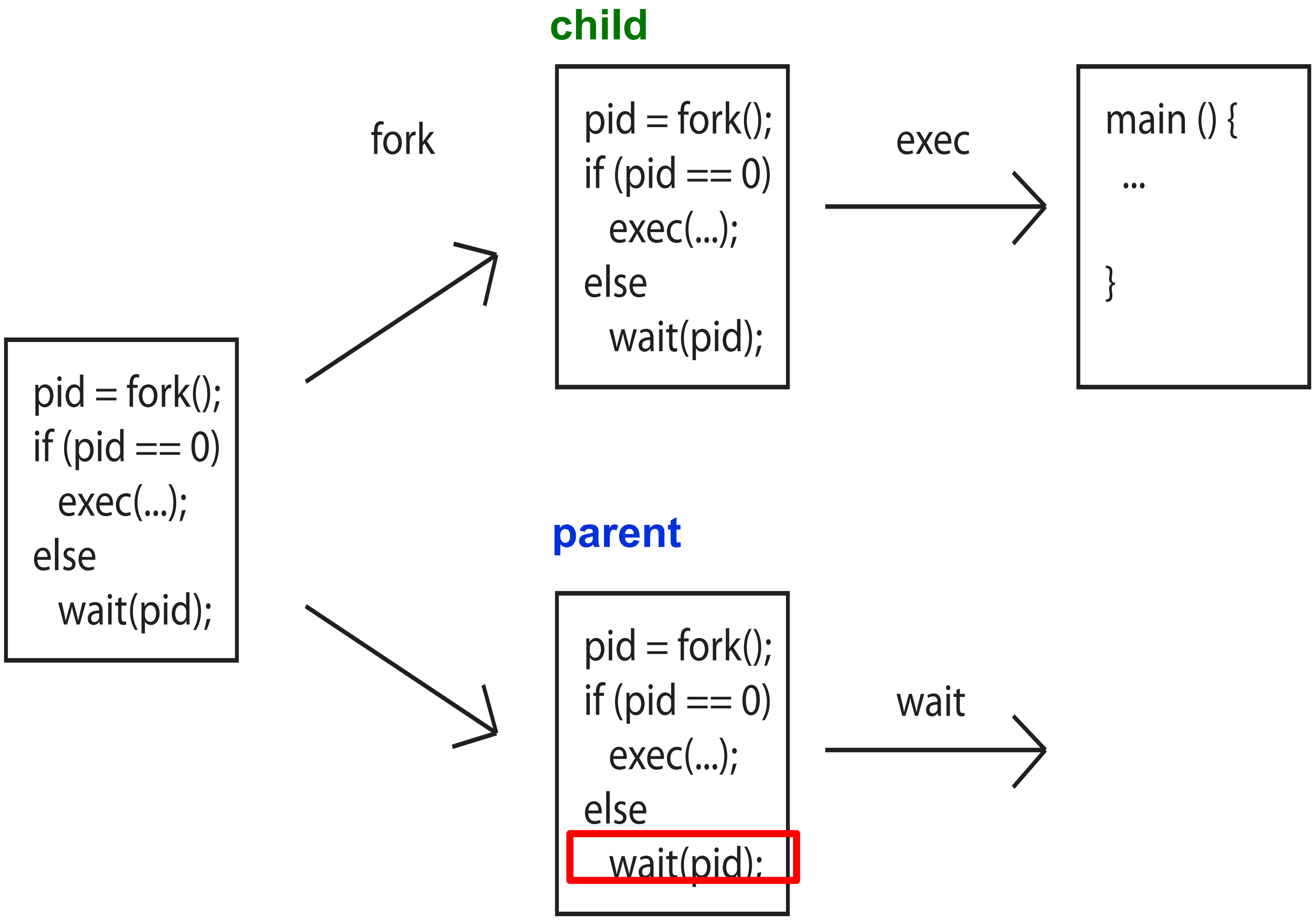
fork2.c – parent waits for child to finish

```
int status;
pid_t tcpid;
…
cpid = fork();
if (cpid > 0) {                /* Parent Process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
} else if (cpid == 0) {        /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
}
…
```

Process Management

- `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

Process Management



fork3.c

```
…
cpid = fork();
if (cpid > 0) {                    /* Parent Process */
  tcpid = wait(&status);
} else if (cpid == 0) {      /* Child Process */
  char *args[] = {“ls”, “-l”, NULL};
  execv(“/bin/ls”, args);
  /* execv doesn’t return when it works.
     So, if we got here, it failed! */
  perror(“execv”);
  exit(1);
}
…
```

Shell

- **A shell is a job control system**
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells

- **Example: to compile a C program**

```
cc –c sourcefile1.c
cc –c sourcefile2.c
In –o program sourcefile1.o sourcefile2.o
./program
```

HW3

Process Management

- `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

inf_loop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
  printf("Caught signal!\n");
  exit(1);
}
int main() {
  struct sigaction sa;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = signal_callback_handler;
  sigaction(SIGINT, &sa, NULL);
  while (1) {}
}
```

Common POSIX Signals

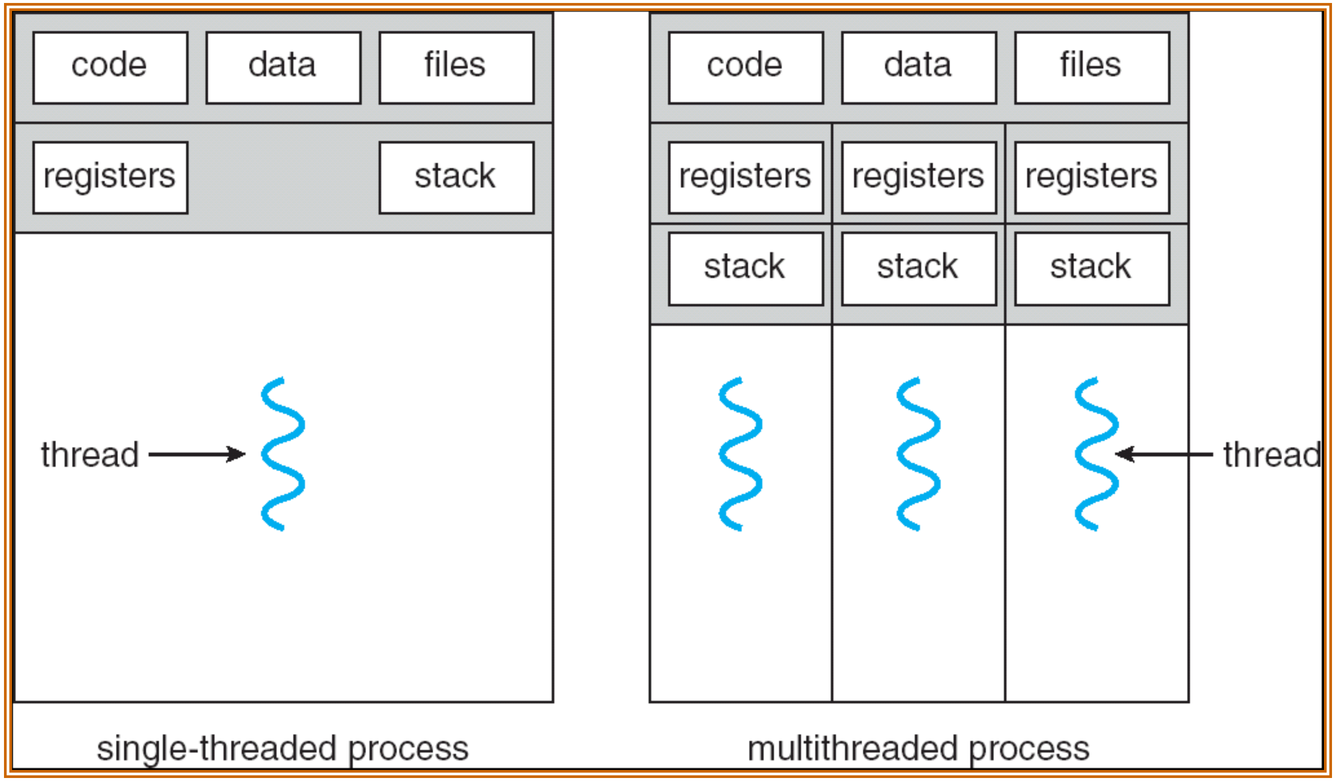
- `SIGINT` – control-C
- `SIGTERM` – default for `kill` shell command
- `SIGSTP` – control-Z (default action: stop process)

- `SIGKILL`, `SIGSTOP` – terminate/stop process
  - Can't be changed with `sigaction`
  - Why?

Modern Processes: Multiple Threads

- Thread: execution stream within a process
  - Used to be called "lightweight processes"
  - Shares address space with other threads belonging to the same process
- Why separate concepts of threads and processes?
  - Threads: Concurrency
  - Processes: Protection

Single vs. Multithreaded Processes



Summary

- **Process consists of two pieces**
  1. **Address Space (Memory & Protection)**
  2. **One or more threads (Concurrency)**
- **Represented in kernel as**
  - **Process object (resources associated with process)**
  - **Thread object + (mini) stack**
  - **Hardware support critical in U → K → U context switch**
  - **Different privileges in different modes (CPL, Page Table)**
- **Variety of process management syscalls**
  - `fork`, `exec`, `wait`, `kill`, `sigaction`
- **Scheduling: Threads move between queues**
- **Threads: multiple stacks per address space**
  - **Context switch: Save/Restore registers, "return" from new thread's `switch` routine**
  - **So far, we've only seen kernel threads**