CS162
Operating Systems and Systems Programming
Lecture 3

Processes (con't), Fork, System Calls

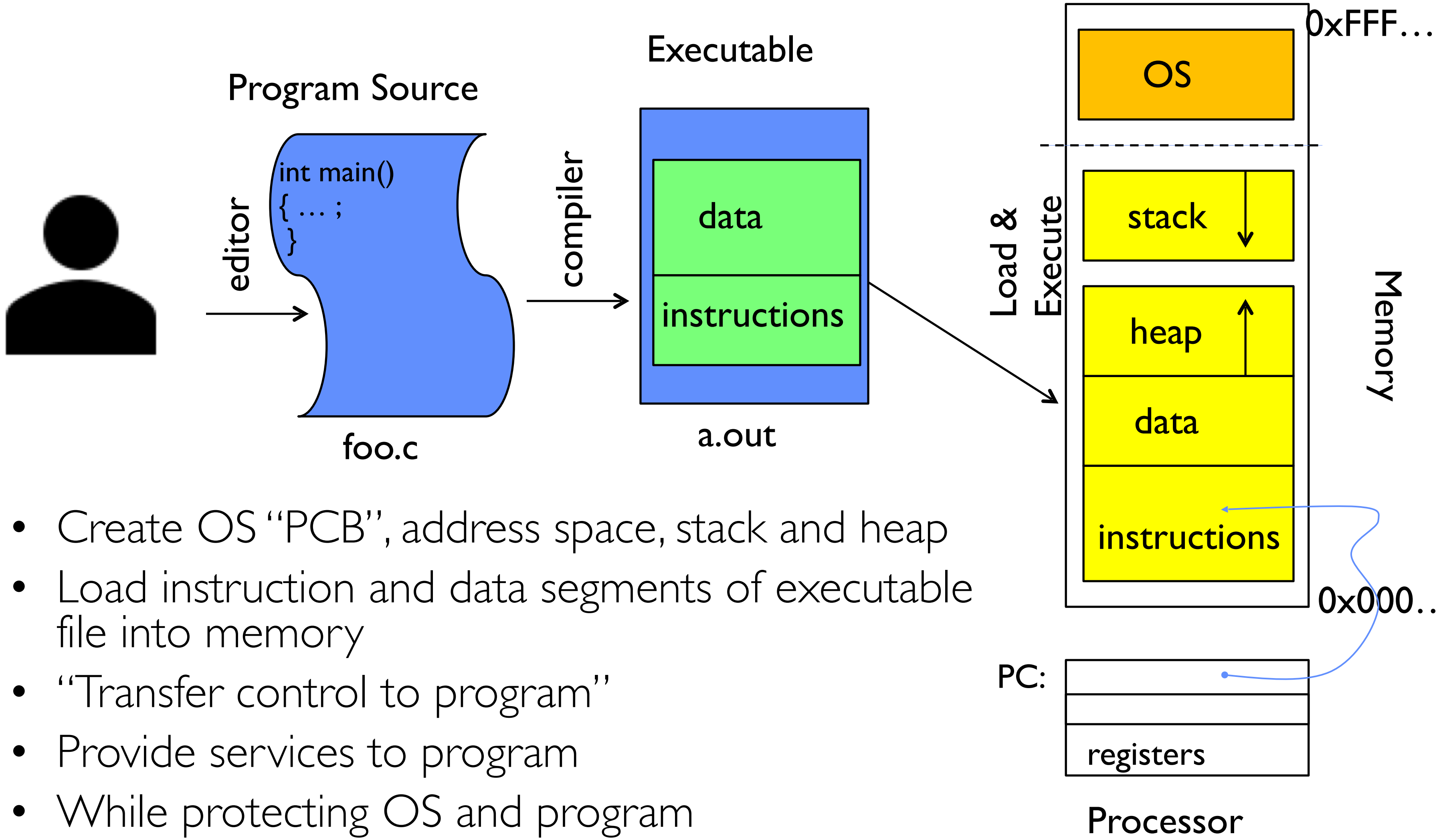
January 28th, 2020
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes*. When slides are obtained from other sources, a a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

Recall: Four Fundamental OS Concepts

- Thread: Execution Context
  - Fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- Address space (with or w/o translation)
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- Process: an instance of a running program
  - Protected Address Space + One or more Threads
- Dual mode operation / Protection
  - Only the "system" has the ability to access certain resources
  - Combined with translation, isolates programs from each other and the OS from programs

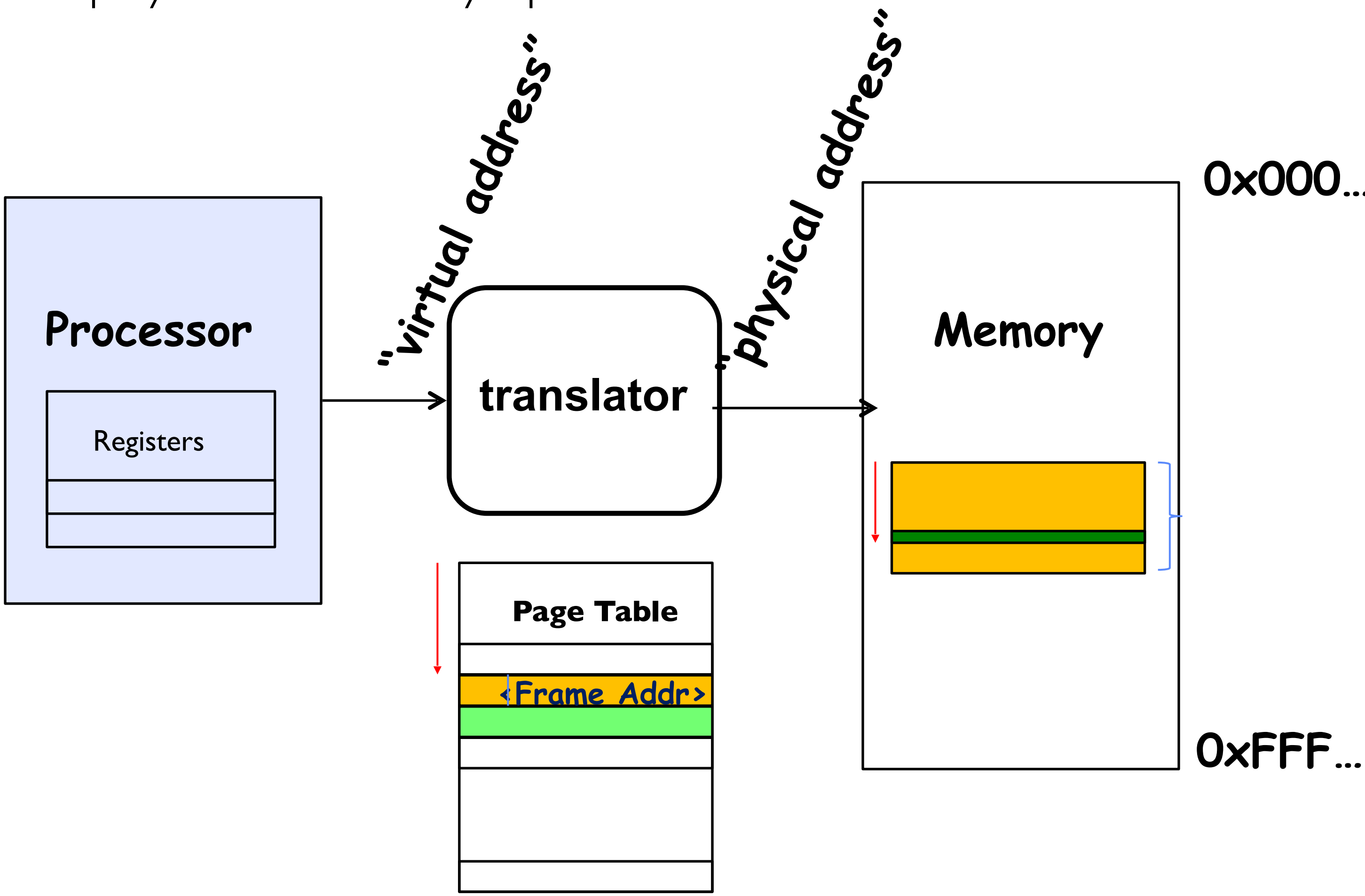
Recall: OS Bottom Line: Run Programs



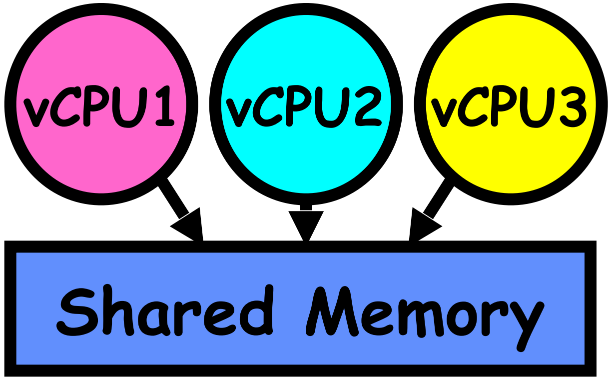
- Create OS "PCB", address space, stack and heap
- Load instruction and data segments of executable file into memory
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

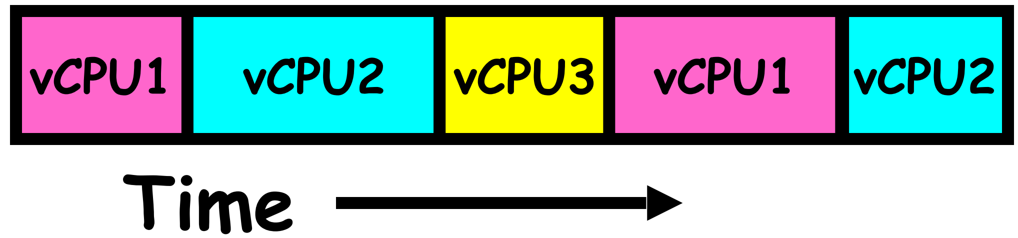
Recall: Protected Address Space

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



Recall: give the illusion of multiple processors?



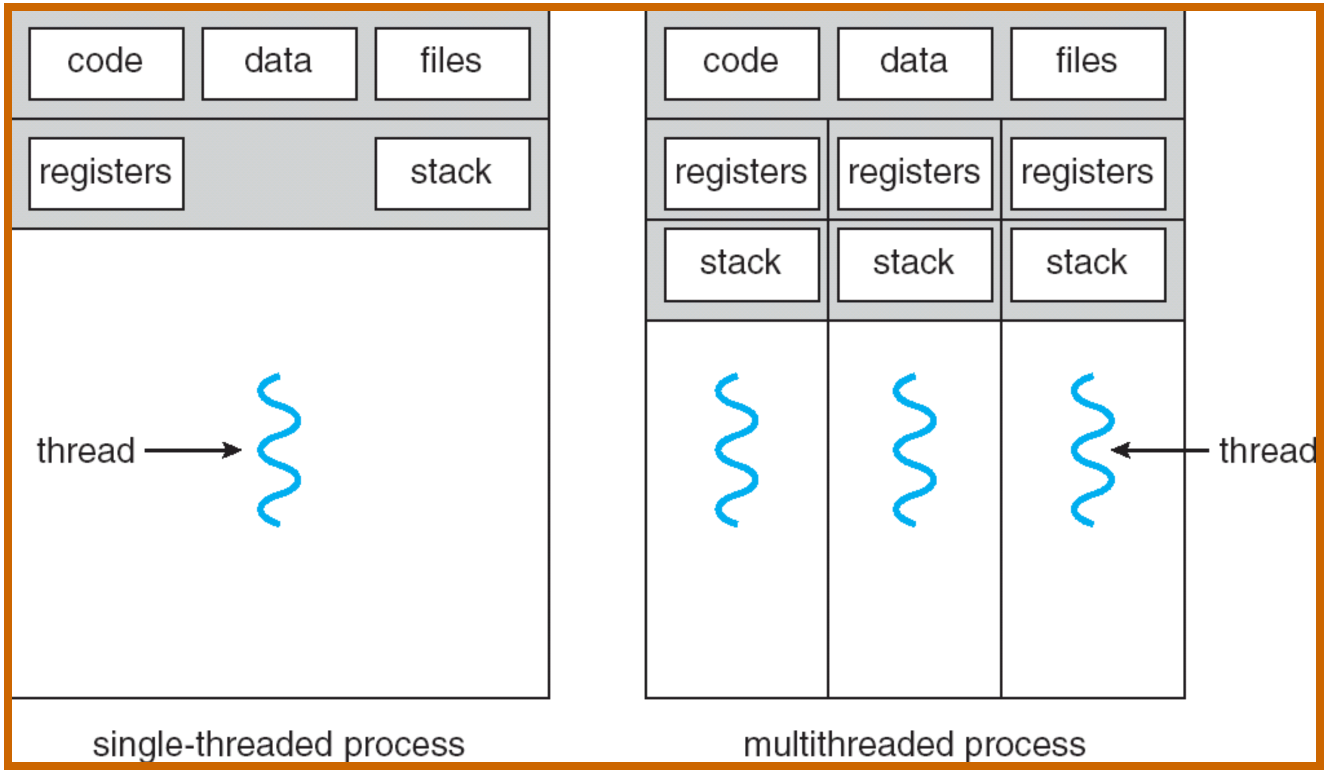


- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
  - Multiple "virtual CPUs"
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one virtual CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

Recall: The Process

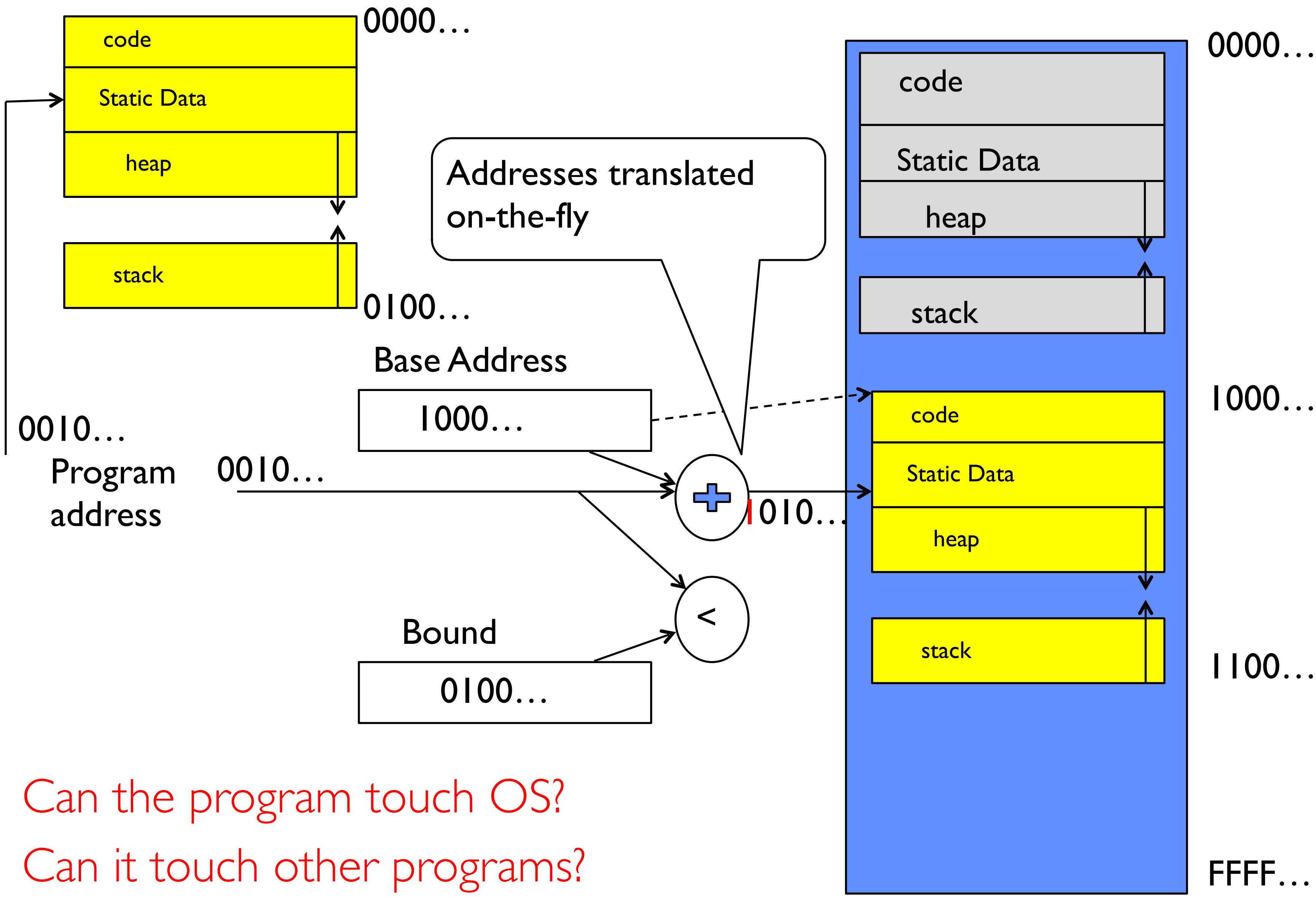
- **Definition:** execution environment with restricted rights
  - Address Space with One or More Threads
    - » *Page table per process!*
  - Owns memory (mapped pages)
  - Owns file descriptors, file system context, …
  - Encapsulates one or more threads sharing process resources
- Application program 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

Recall: Single and Multithreaded Processes



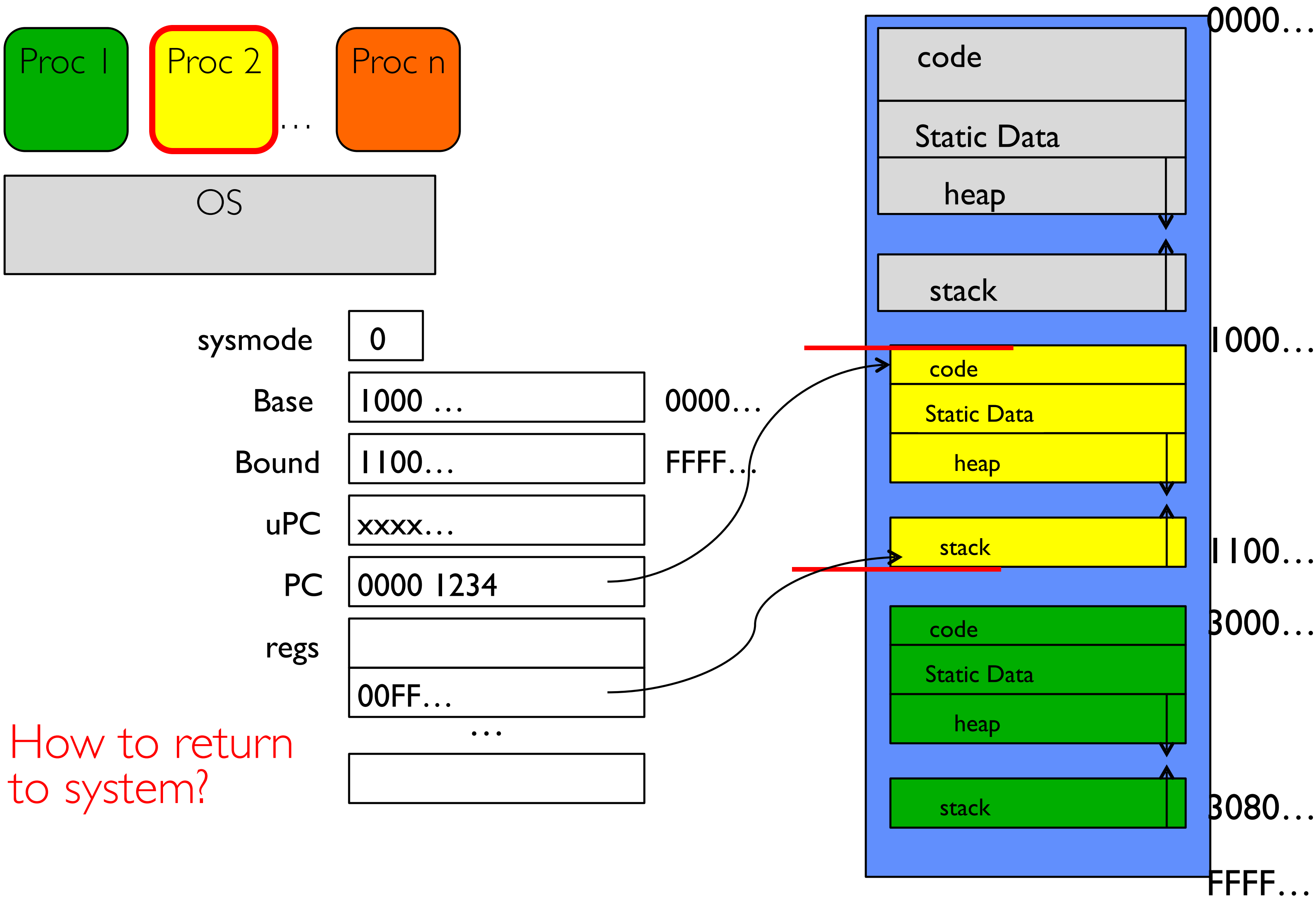
- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Recall: Simple address translation with Base and Bound



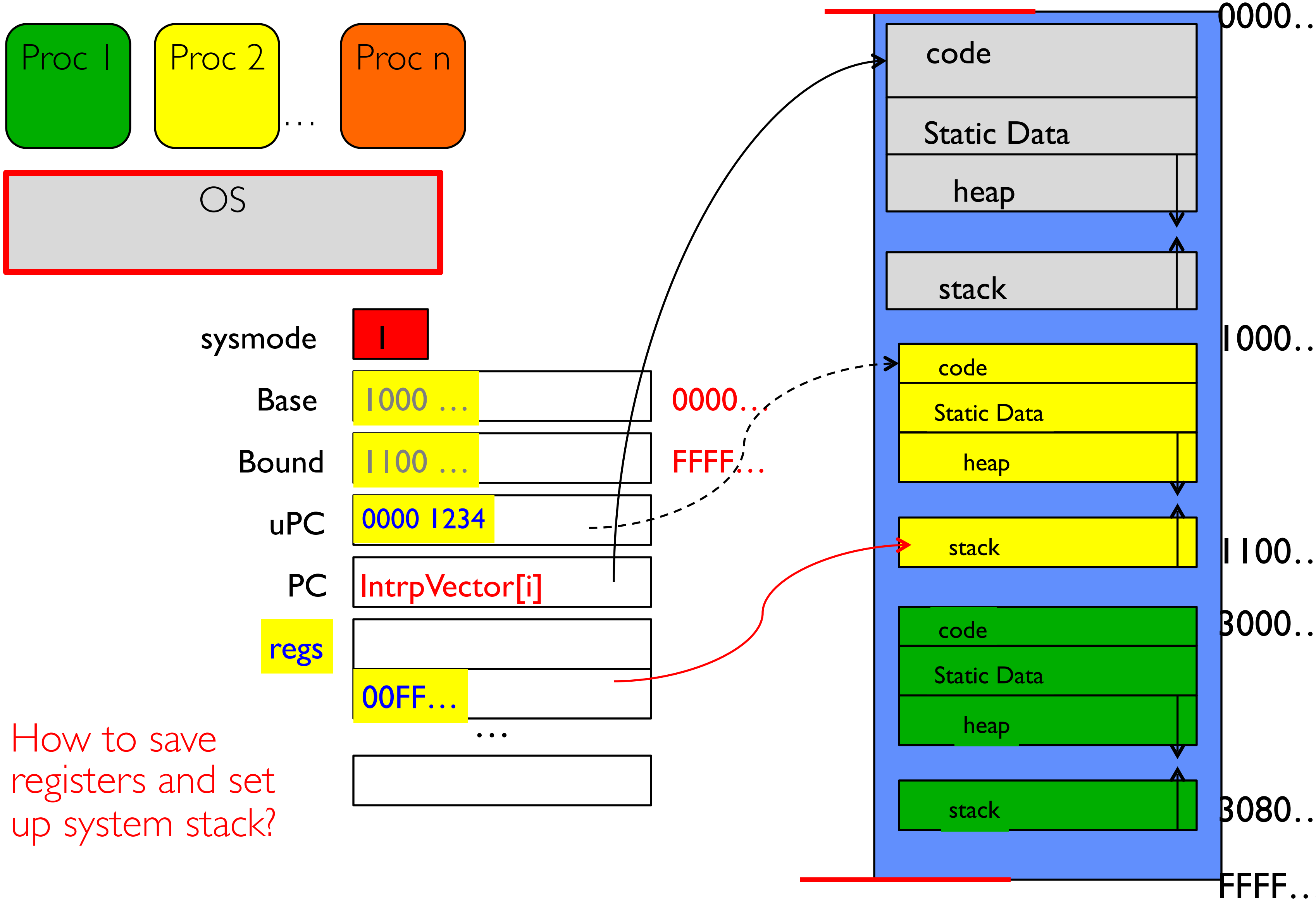
- Can the program touch OS?
- Can it touch other programs?

Simple B&B: User => Kernel



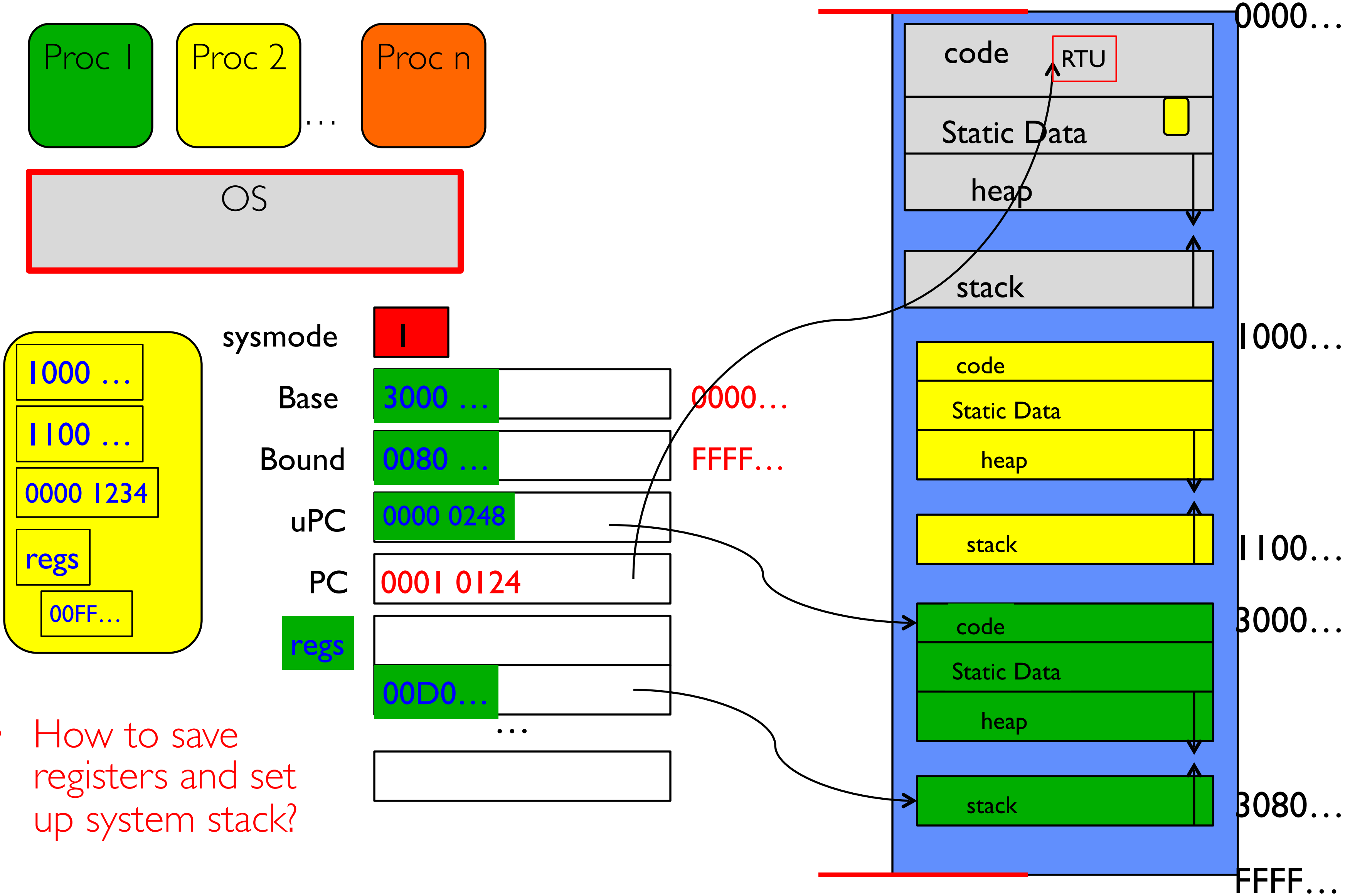
- How to return to system?

Simple B&B: Interrupt



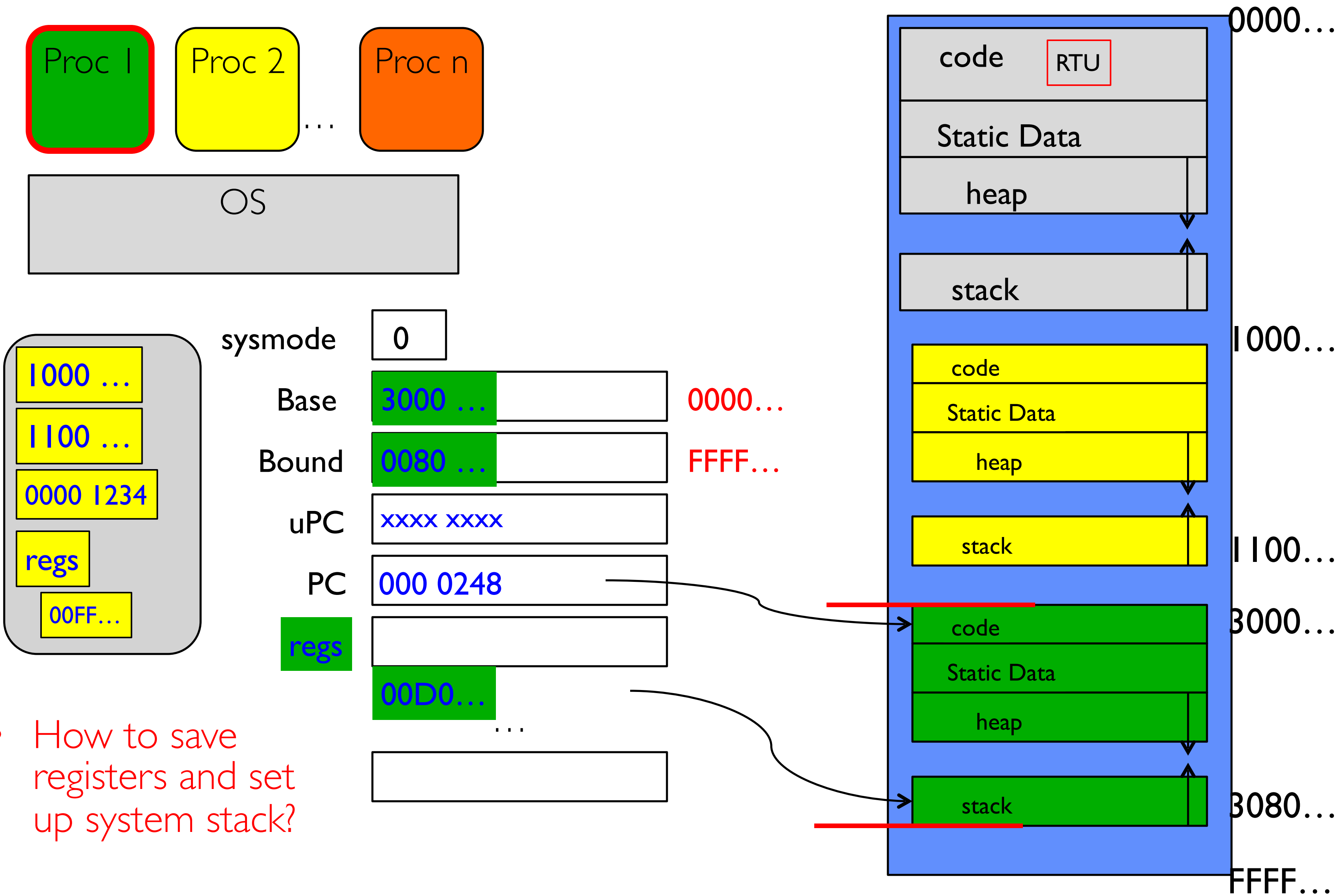
- How to save registers and set up system stack?

Simple B&B: Switch User Process



- How to save registers and set up system stack?

Simple B&B: "resume"

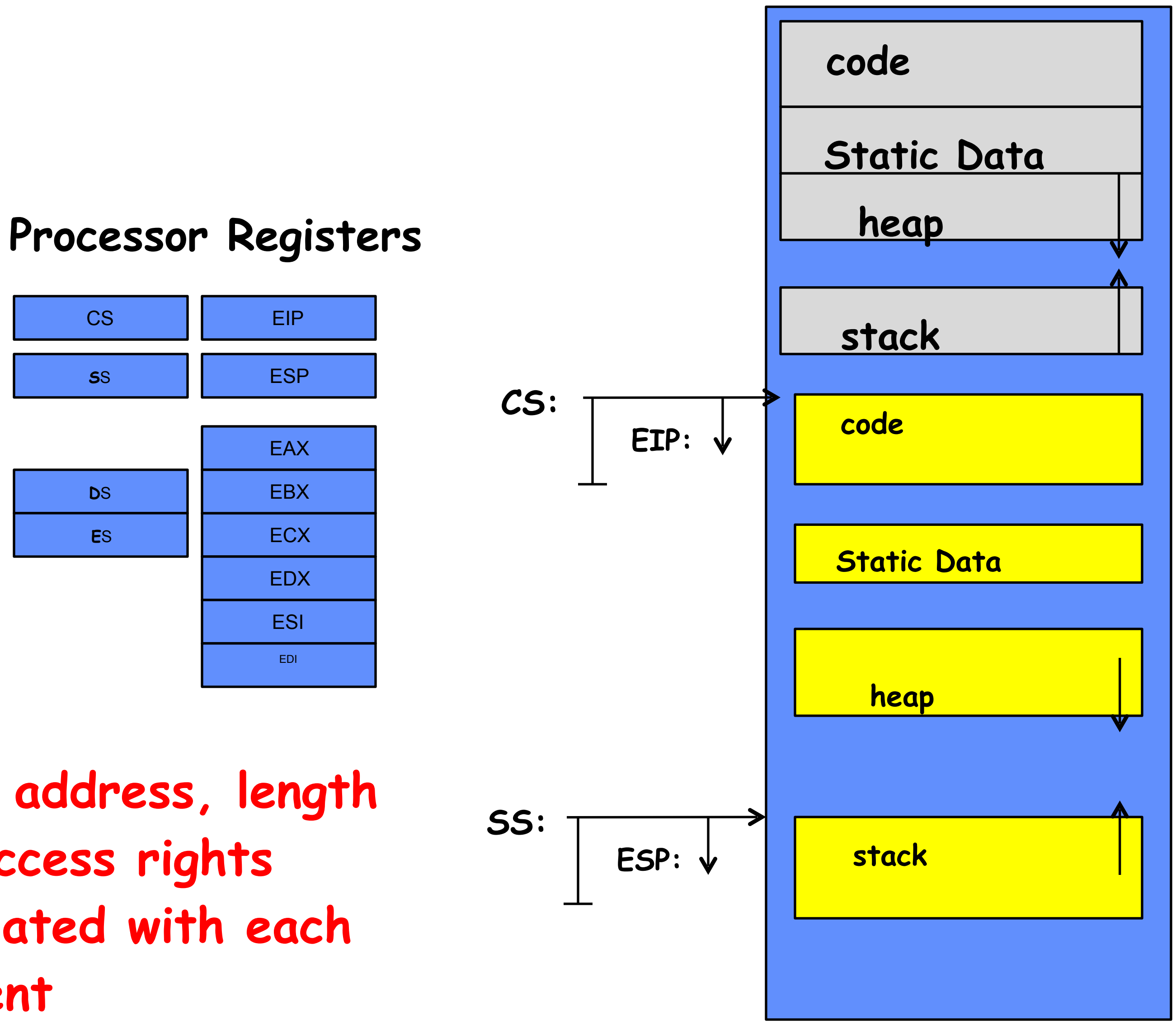


- How to save registers and set up system stack?

Is Branch and Bound a Good-Enough Protection Mechanism?

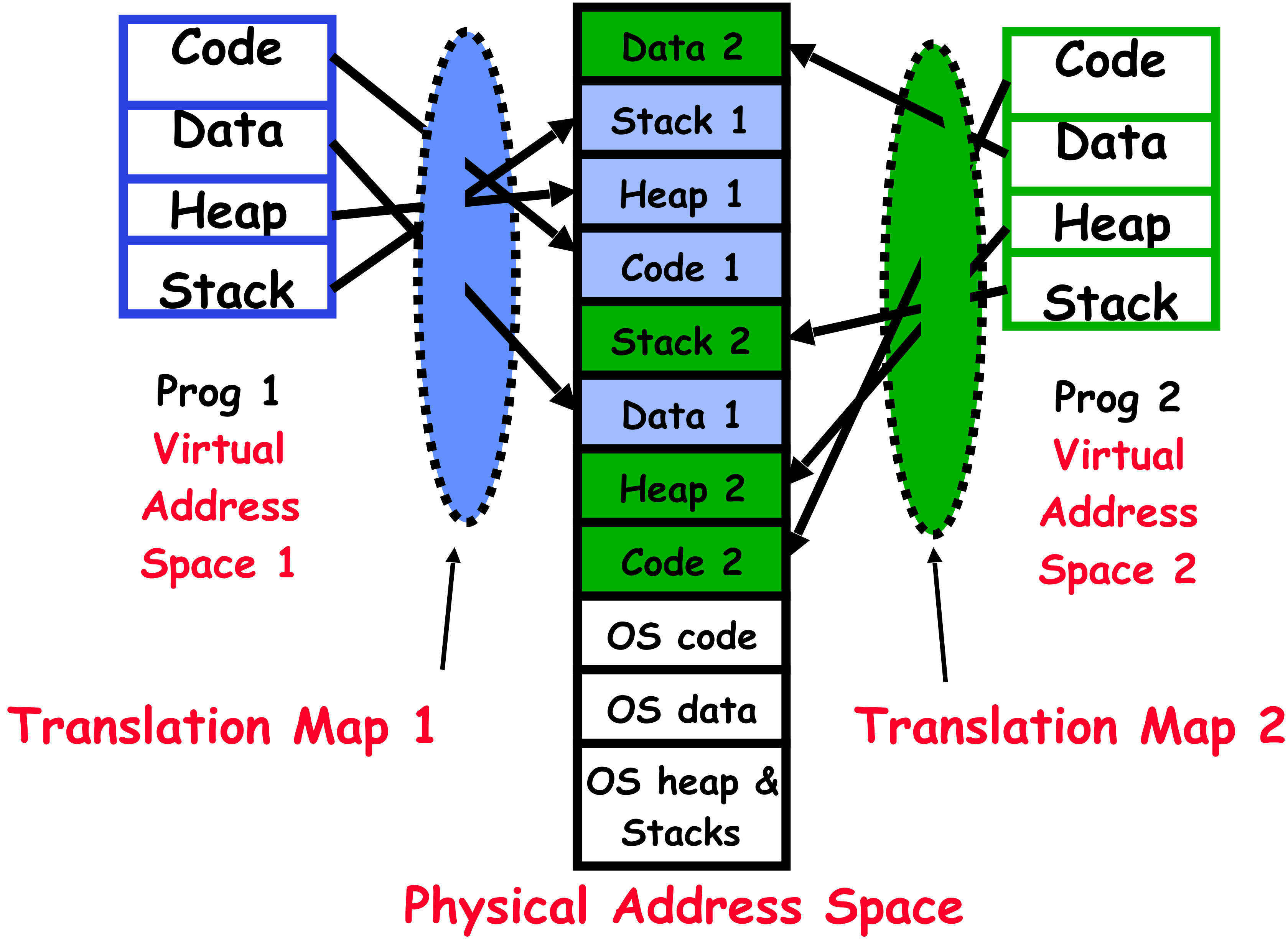
- NO: Too simplistic for real systems
- Inflexible/Wasteful:
  - Must dedicate physical memory for *potential* future use
  - (Think stack and heap!)
- Fragmentation:
  - Kernel has to somehow fit whole processes into contiguous block of memory
  - After a while, memory becomes fragmented!
- Sharing:
  - Very hard to share any data between Processes or between Process and Kernel
  - Need to communicate indirectly through the kernel…

Better: x86 – segments and stacks

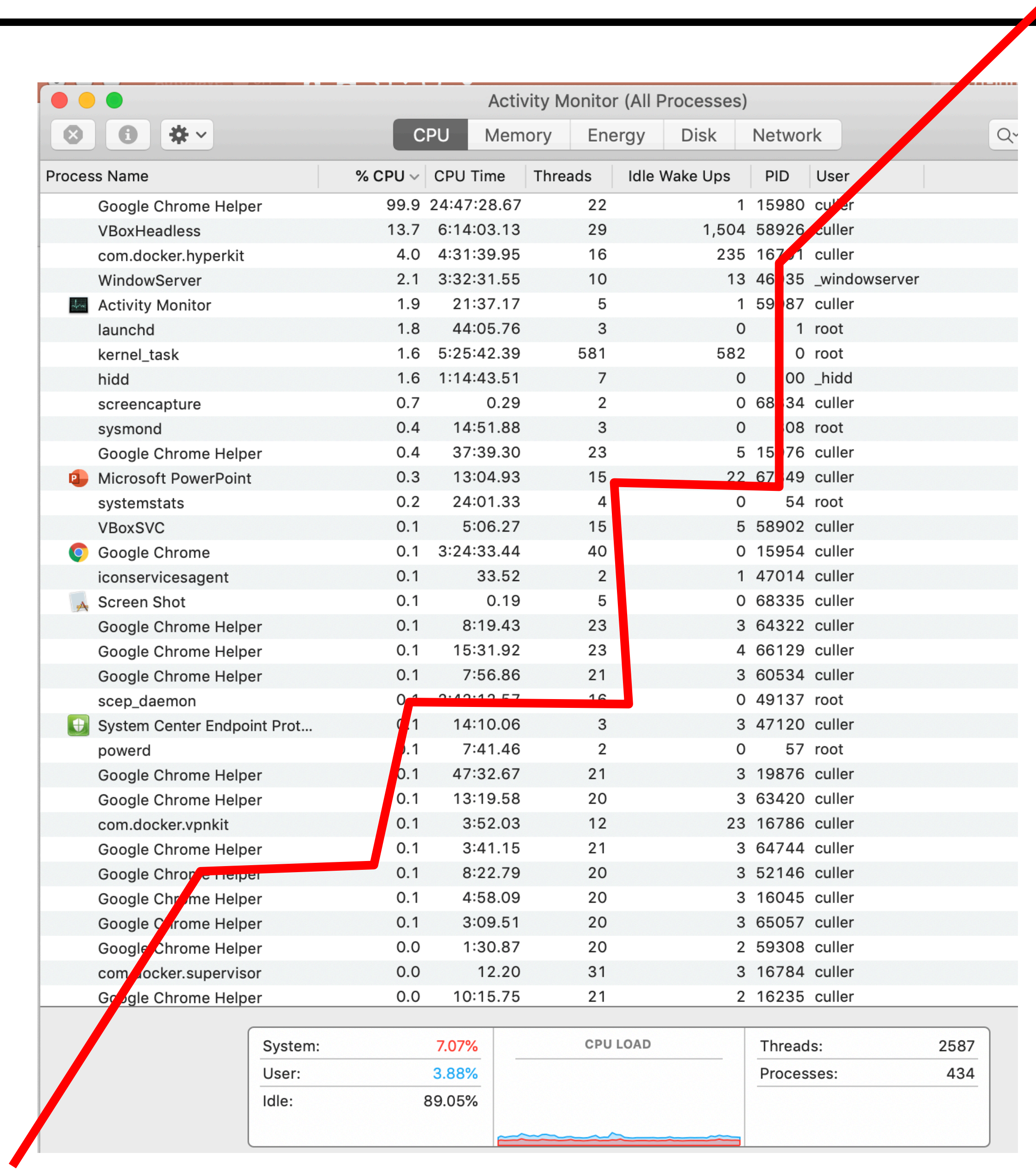


Start address, length and access rights associated with each segment

Alternative: Page Table Mapping (More soon!)



What's beneath the Illusion?



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
Google Chrome Helper	99.9	24:47:28.67	22	1	15980	cu[illegible]er
VBoxHeadless	13.7	6:14:03.13	29	1,504	58926	culler
com.docker.hyperkit	4.0	4:31:39.95	16	235	167[illegible]1	culler
WindowServer	2.1	3:32:31.55	10	13	46[illegible]35	_windowserver
Activity Monitor	1.9	21:37.17	5	1	59[illegible]87	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]00	_hidd
screencapture	0.7	0.29	2	0	68[illegible]34	culler
sysmond	0.4	14:51.88	3	0	[illegible]08	root
Google Chrome Helper	0.4	37:39.30	23	5	15[illegible]76	culler
Microsoft PowerPoint	0.3	13:04.93	15	22	67[illegible]49	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	0.1	[illegible]	16	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 Chro[illegible]	0.1	8:22.79	20	3	52146	culler
Google Chr[illegible]me Helper	0.1	4:58.09	20	3	16045	culler
Google C[illegible]rome Helper	0.1	3:09.51	20	3	65057	culler
Googl[illegible] Chrome Helper	0.0	1:30.87	20	2	59308	culler
com[illegible]ocker.supervisor	0.0	12.20	31	3	16784	culler
Go[illegible]gle Chrome Helper	0.0	10:15.75	21	2	16235	culler

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

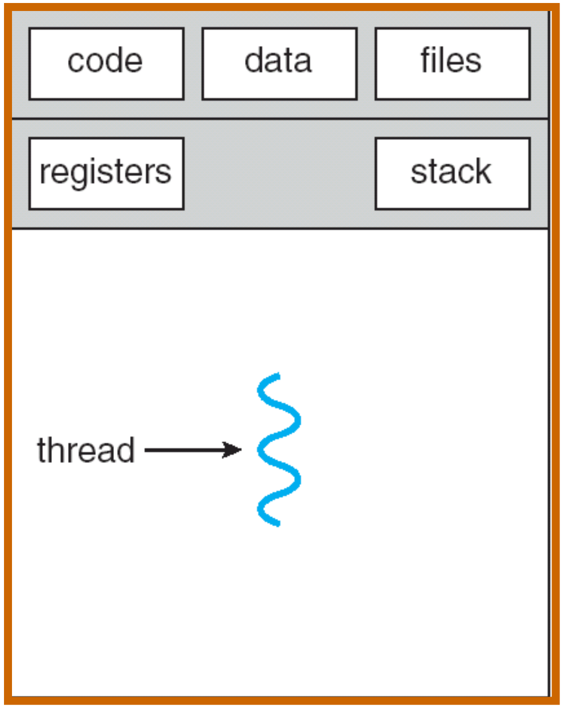
CPU LOAD

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)

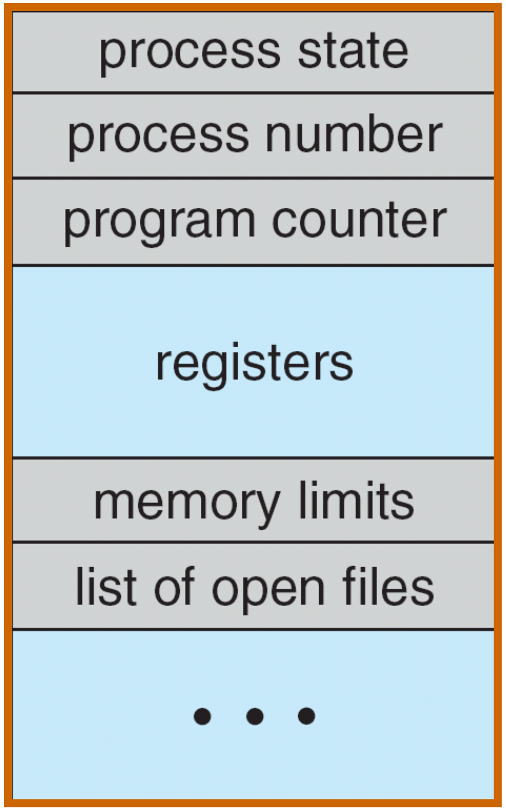


Running Many Programs

- We have the basic mechanism to
  - switch between user processes and the kernel,
  - the kernel can switch among user processes,
  - Protect OS from user processes and processes from each other
- Questions ???
  - How do we represent each process in the kernel?
  - How do we decide which user process to run?
  - How do we pack up the process and set it aside?
  - How do we get a stack and heap for the kernel?
  - Aren't we wasting are lot of memory?

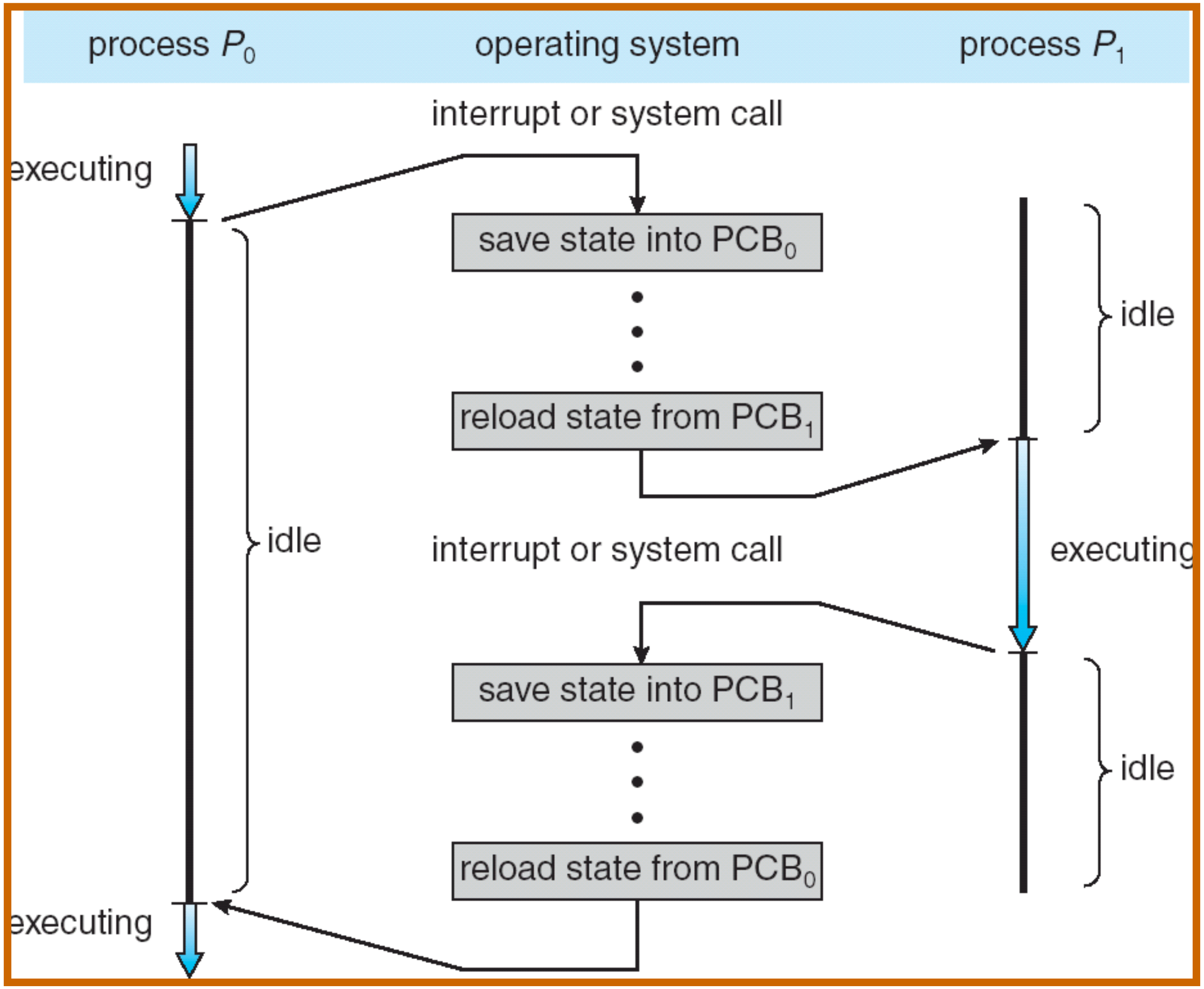
Multiplexing Processes: The Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel *Scheduler* maintains a data structure containing the PCBs
  - Give out CPU to different processes
  - This is a Policy Decision
- Give out non-CPU resources
  - Memory/IO
  - Another policy decision

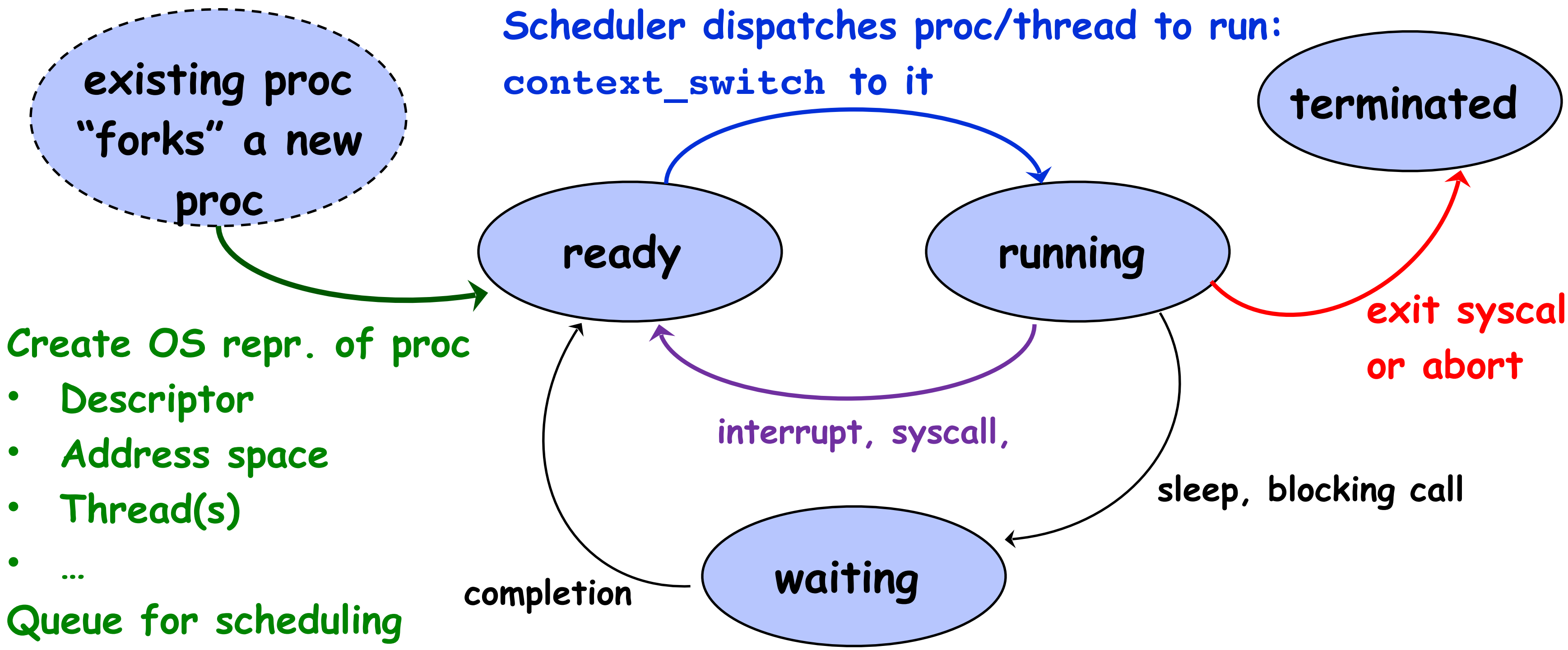


Process Control Block

Context Switch



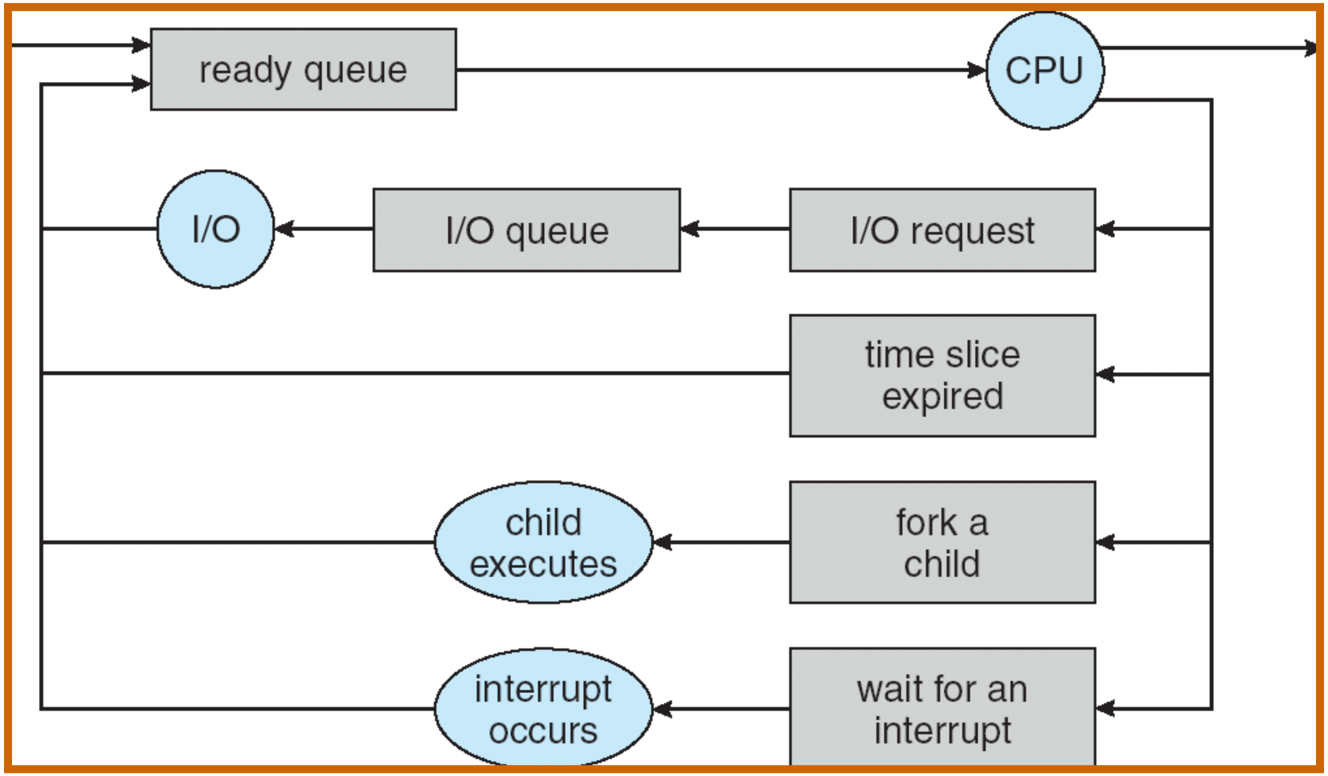
Lifecycle of a process / thread



- 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

Scheduling: All About Queues



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

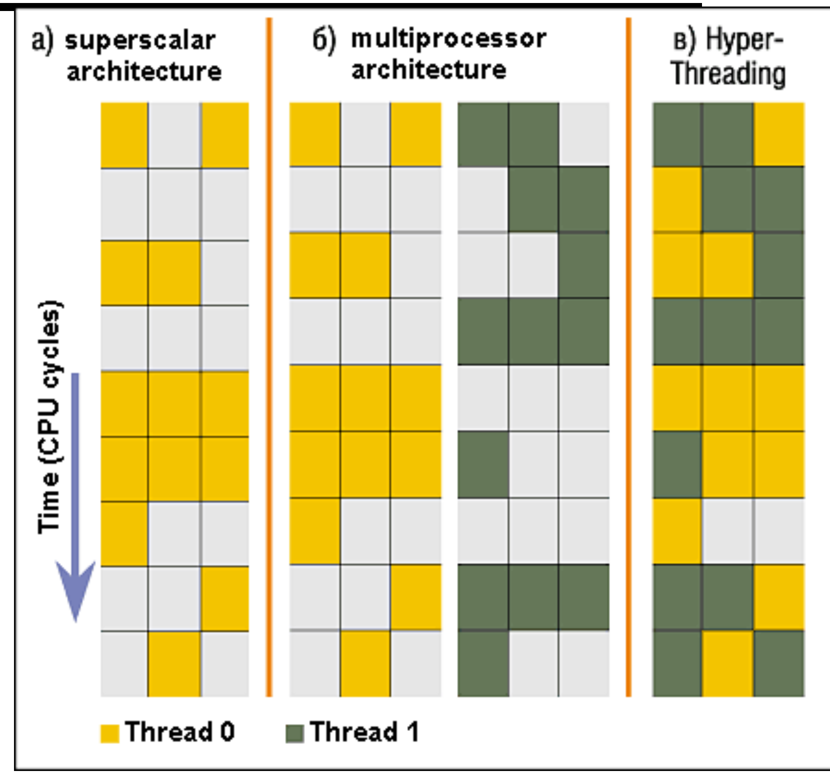
Scheduler

```
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 ..

Simultaneous MultiThreading/Hyperthreading

- Hardware scheduling technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading 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!

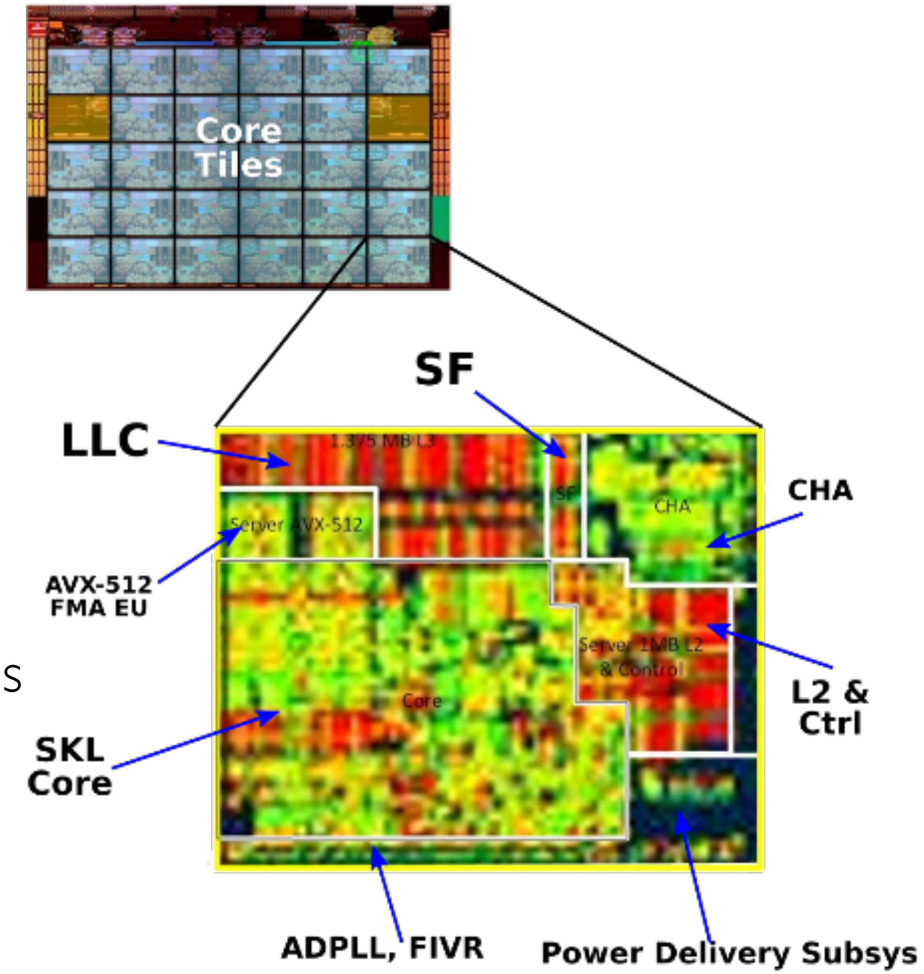


Colored blocks show instructions executed

- Original technique called "Simultaneous Multithreading"
  - http://www.cs.washington.edu/research/smt/index.html
  - SPARC, Pentium 4/Xeon ("Hyperthreading"), Power 5

Also Recall: The World Is Parallel

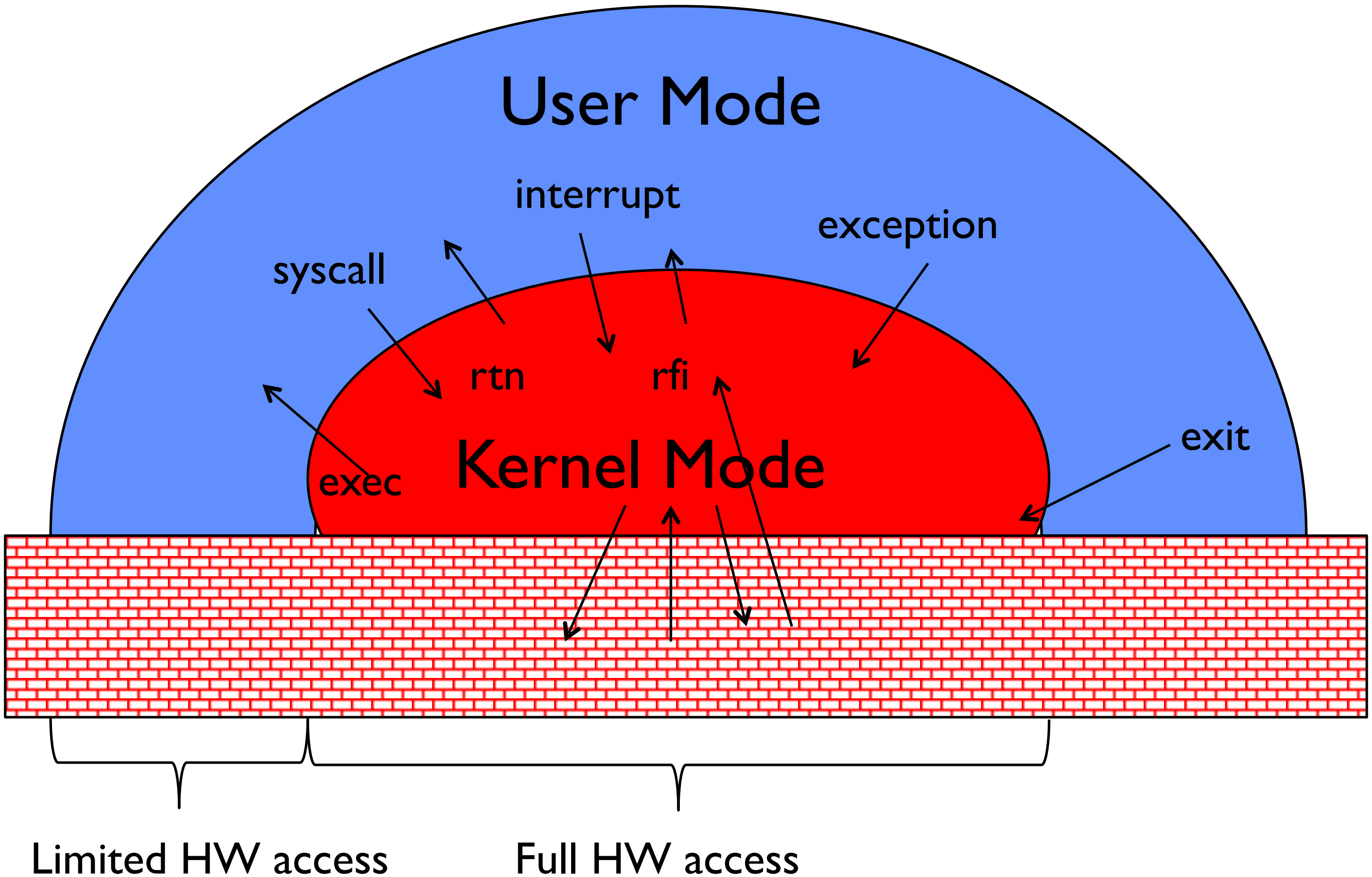
- Intel Skylake (2017)
  - 28 Cores
  - Each core has two hyperthreads!
  - So: 54 Program Counters(PCs)
- Scheduling here means:
  - Pick which core
  - Pick which thread
- Space of possible scheduling much more interesting
  - Can afford to dedicate certain cores to housekeeping tasks
  - Or, can devote cores to services (e.g. Filesystem)



Administrivia: Getting started

- Homework 0 Due Monday!
  - Get familiar with the tools
  - configure your VM, submit via git
  - Practice finding out information:
    - » How to use GDB? How to understand output of unix tools?
    - » We don't assume that you already know everything!
    - » Learn to use "man" (command line), "help" (in gdb, etc), google
- HW1 released today
- Group sign up form
- HW/GHW Schedule/Deadlines
- THIS Monday is Drop Deadline!
  - Given the assignments, this is a highly rewarding but time consuming course
  - If you are not serious about putting in the time, please drop early

Recall: User/Kernel (Privileged) Mode



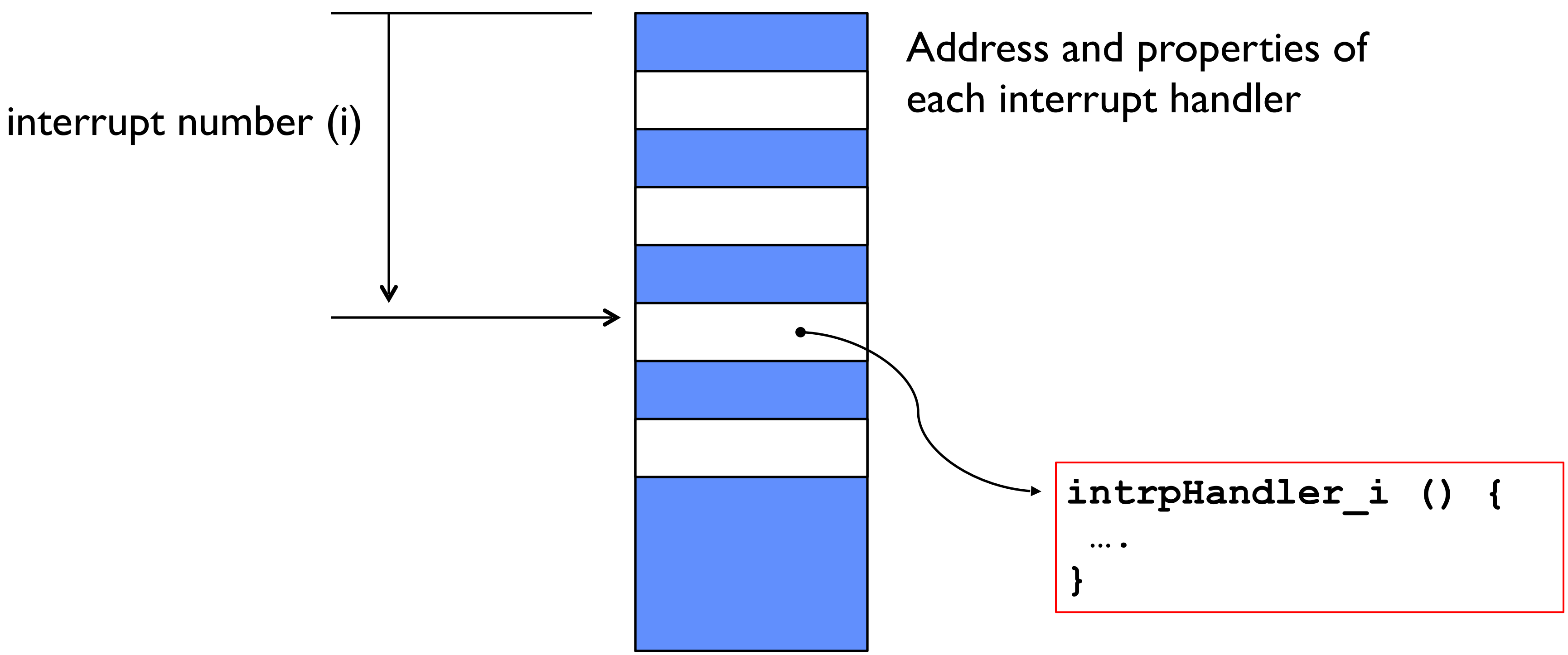
Three types of Kernel Mode Transfer

- 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 or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …

Implementing Safe Kernel Mode Transfers

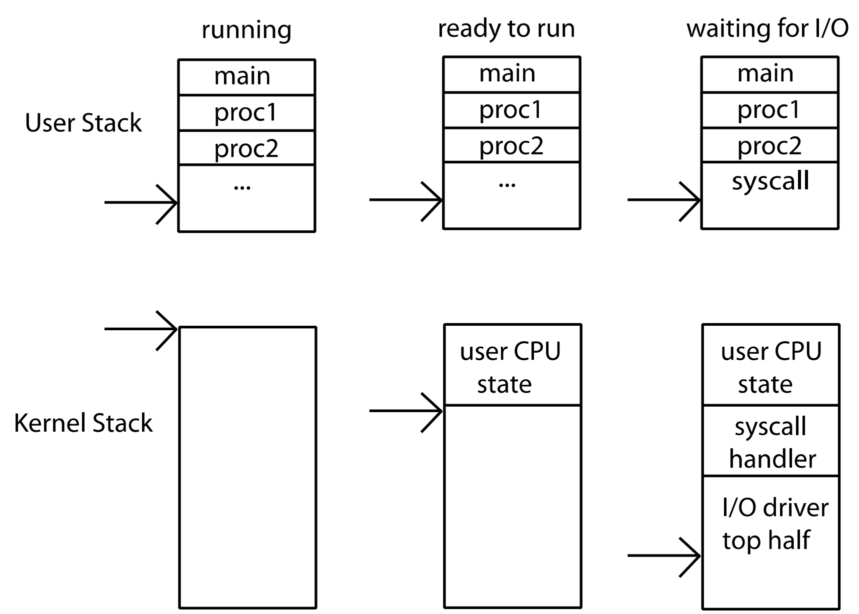
- Important aspects:
  - Controlled transfer into kernel (e.g., syscall table)
  - Separate kernel stack
- Carefully constructed kernel code packs up the user process state and sets it aside
  - Details depend on the machine architecture
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself

Interrupt Vector

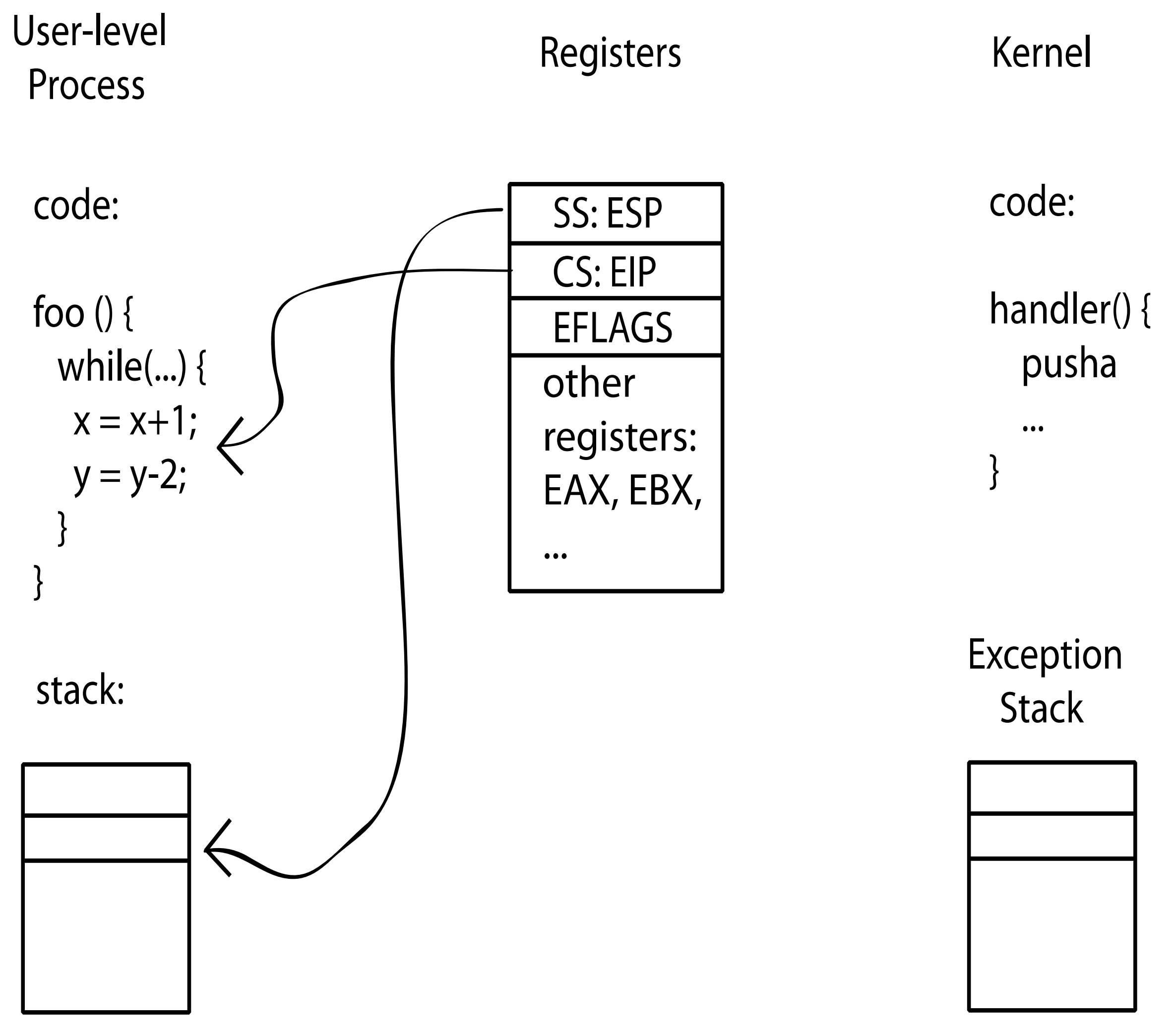


Need for Separate Kernel Stacks

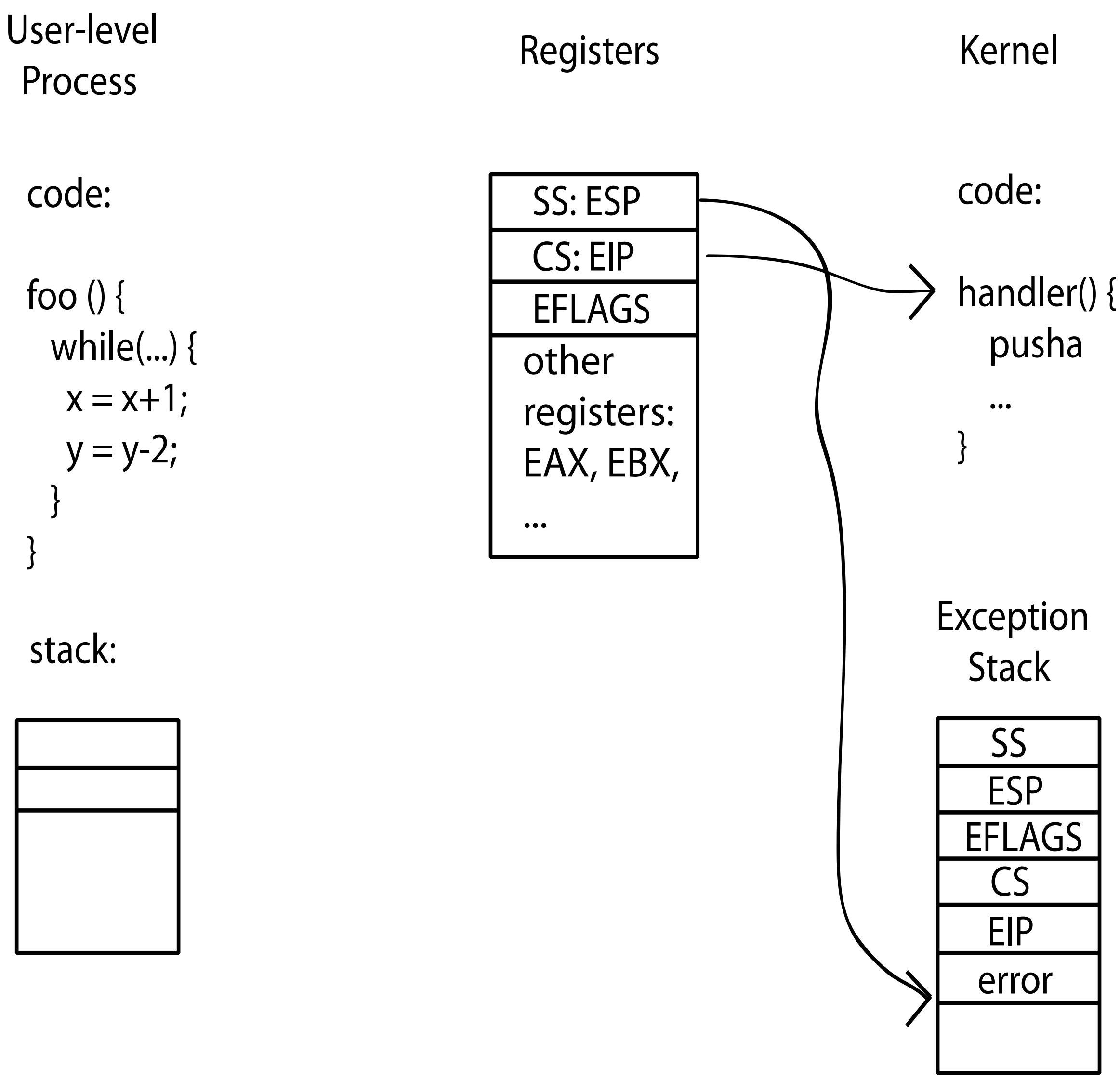
- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)



Before


User-level Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...
Kernel
code:
handler() {
pusha
...
}
Exception Stack

During



User-level Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...
Kernel
code:
handler() {
pusha
...
}
Exception Stack
SS
ESP
EFLAGS
CS
EIP
error

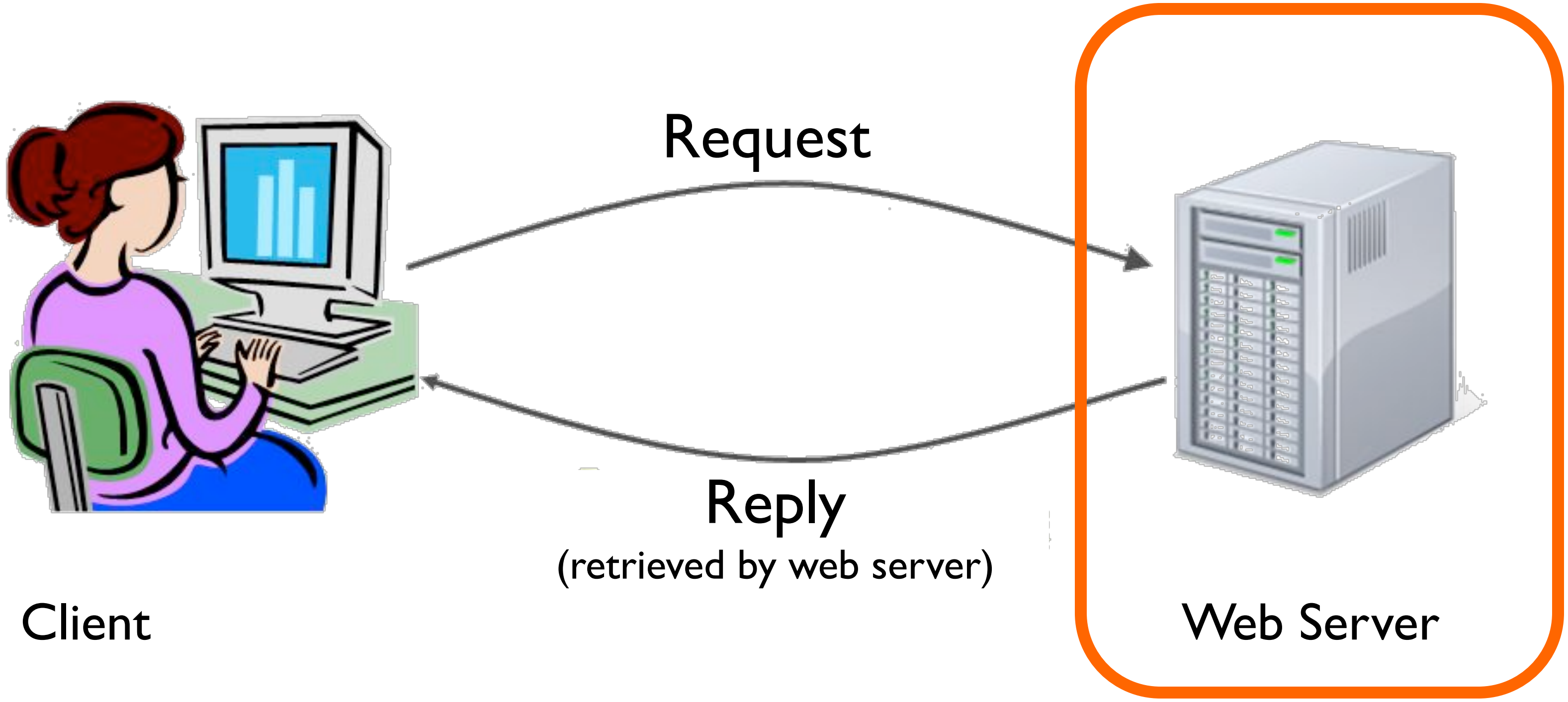
Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to 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

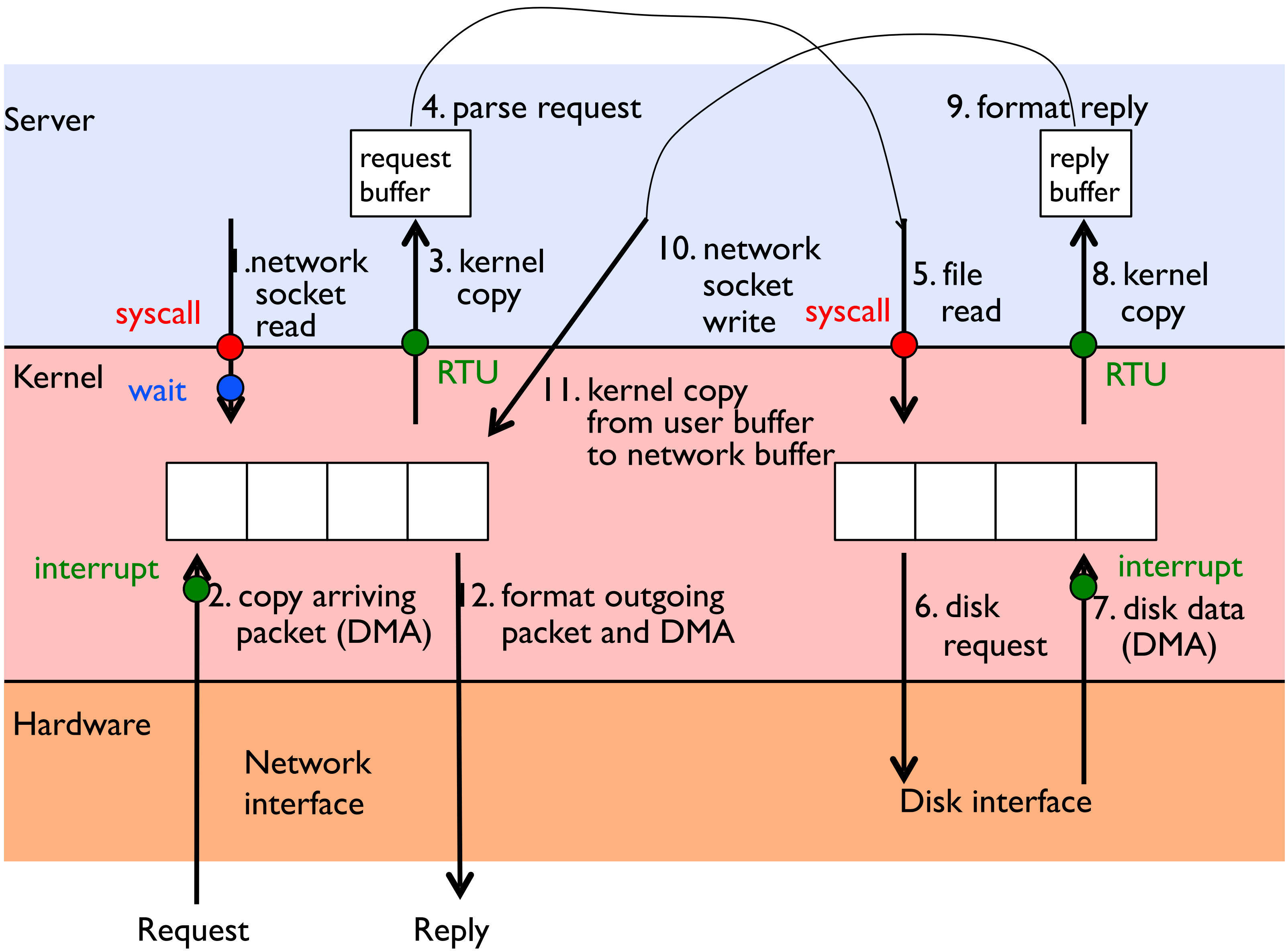
Hardware support: Interrupt Control

- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?
- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    - » wake up an existing OS thread

Putting it together: web server



Putting it together: web server



Meta-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

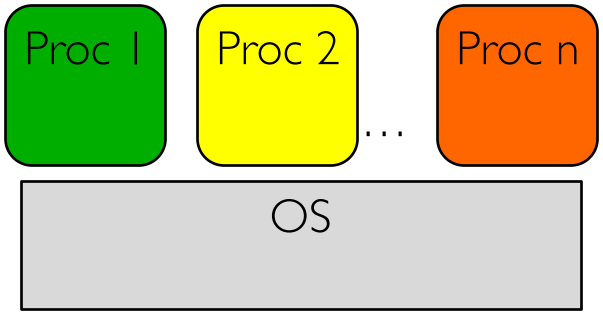
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
  pid_t pid = getpid();   /* get current processes PID */

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

  exit(0);
}
```

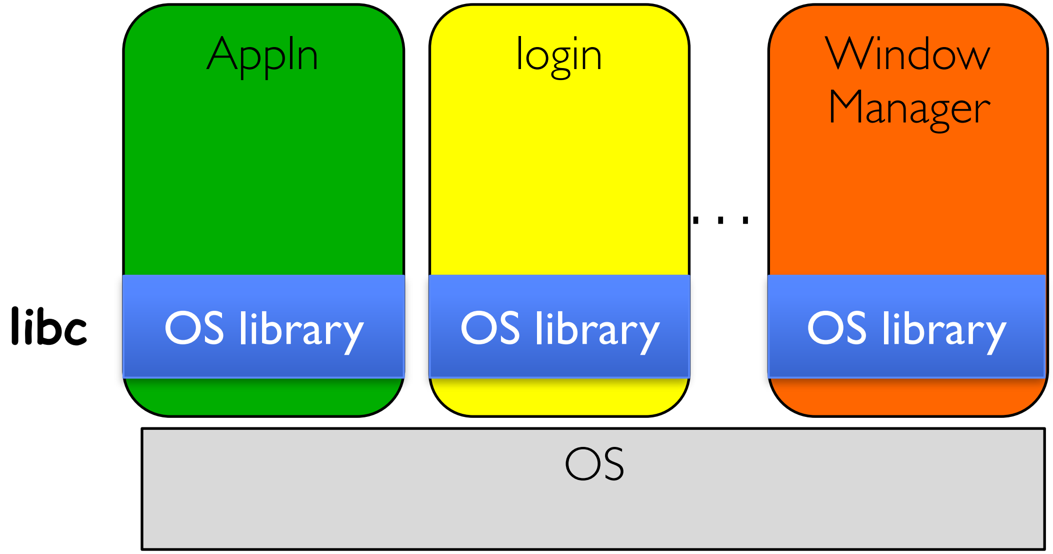
ps anyone?

Can a process create a process ?

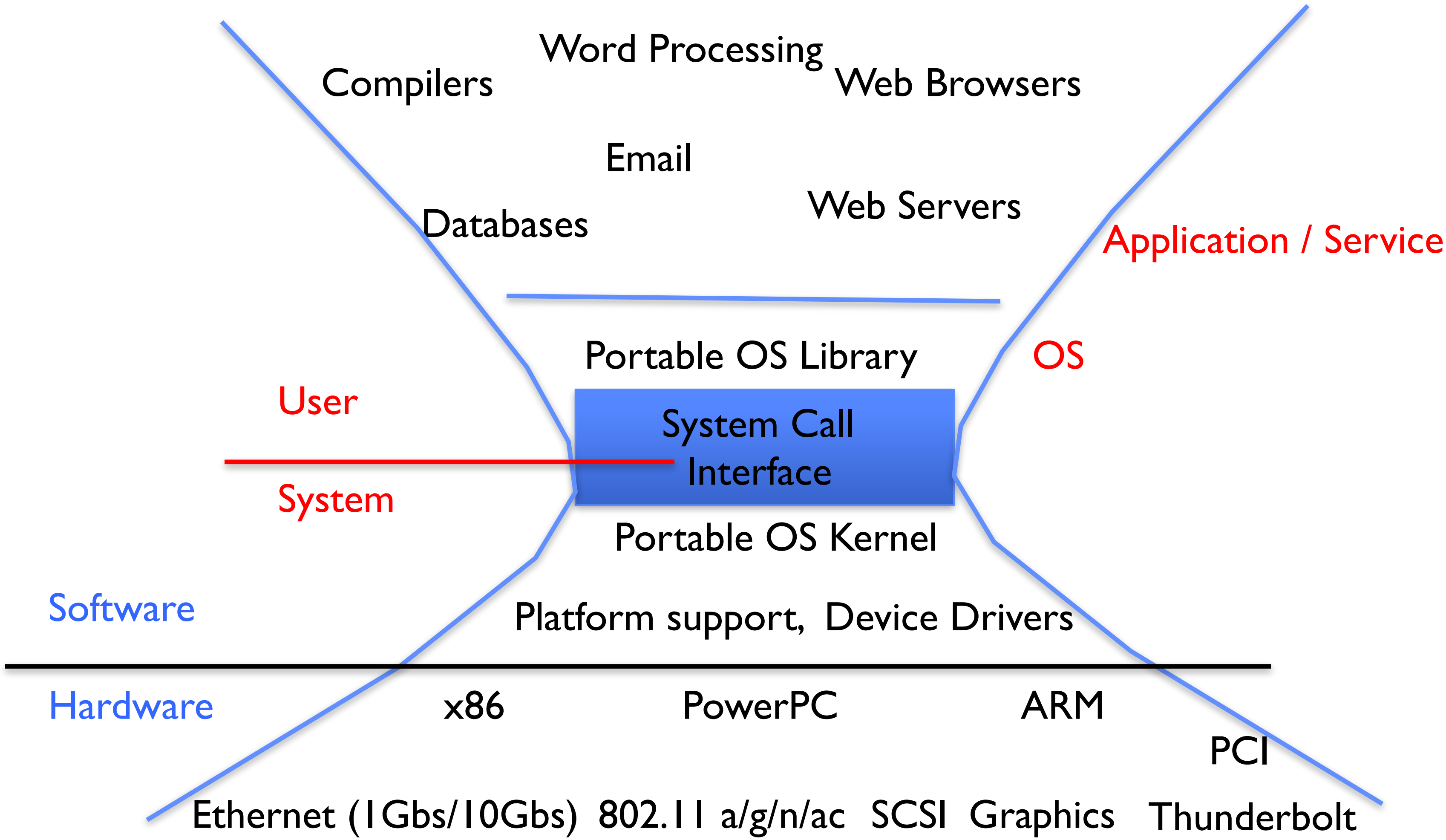
- Yes
- Fork creates a copy of process
- What about the program you want to run?

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

```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

```
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

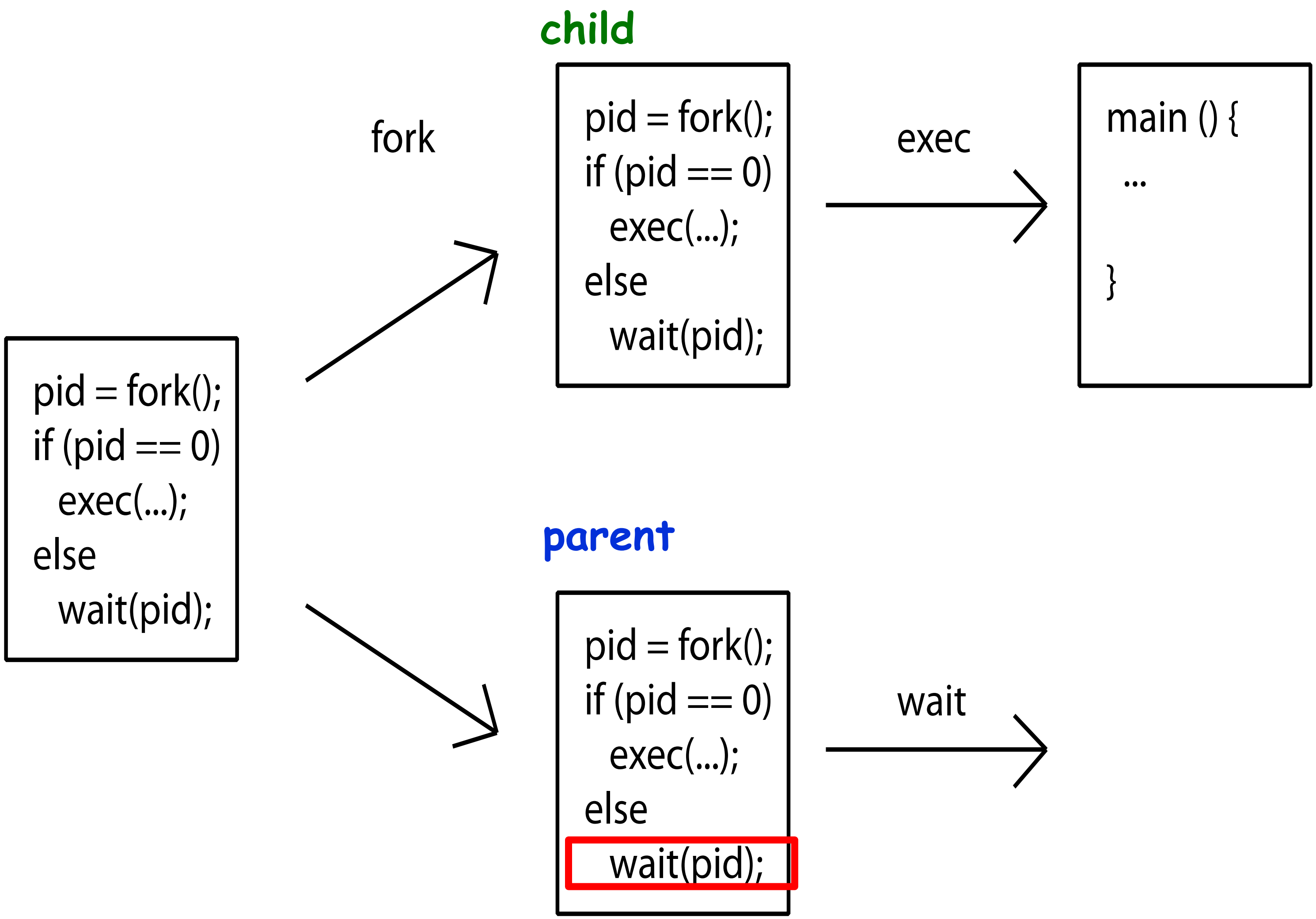
fork2.c – parent waits for child to finish

```
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

  ln –o program sourcefile1.o sourcefile2.o

  ./program

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

```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
- `SIGTSTP` – control-Z (default action: stop process)

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

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)
  – Kernel vs User stack
- Variety of process management syscalls
  – fork, exec, wait, kill, sigaction
- Scheduling: Threads move between queues