Process

CSE 421/521: Operating Systems

Karthik Dantu

Slides adopted from CS162 class at Berkeley, CSE 451 at U-Washington and CSE 421 by Prof Kosar at UB

Logistics – Prior Action Items

- Join Piazza
- Set up development environment: VirtualBox + Ubuntu 16.04
- Implement assignment#0 and test in the environment
- Form groups

Logistics - II

- Assignment 1 out
- Recitations start this week
  - Wed 10-10:50 (NSC 210)
  - Fri 8-8:50 (Park 250)
- Recitation: Basic Pintos discussion as well as C/git/Unix tools
- Schedule up on website – check for conflicts!

Logistics – New action Items

- Assignment 1 out
  - Read the code
  - Compile/test
  - Learn structure – use *printf*s where you can to understand flow
- Test sample programs from class

Recall: Four fundamental OS concepts

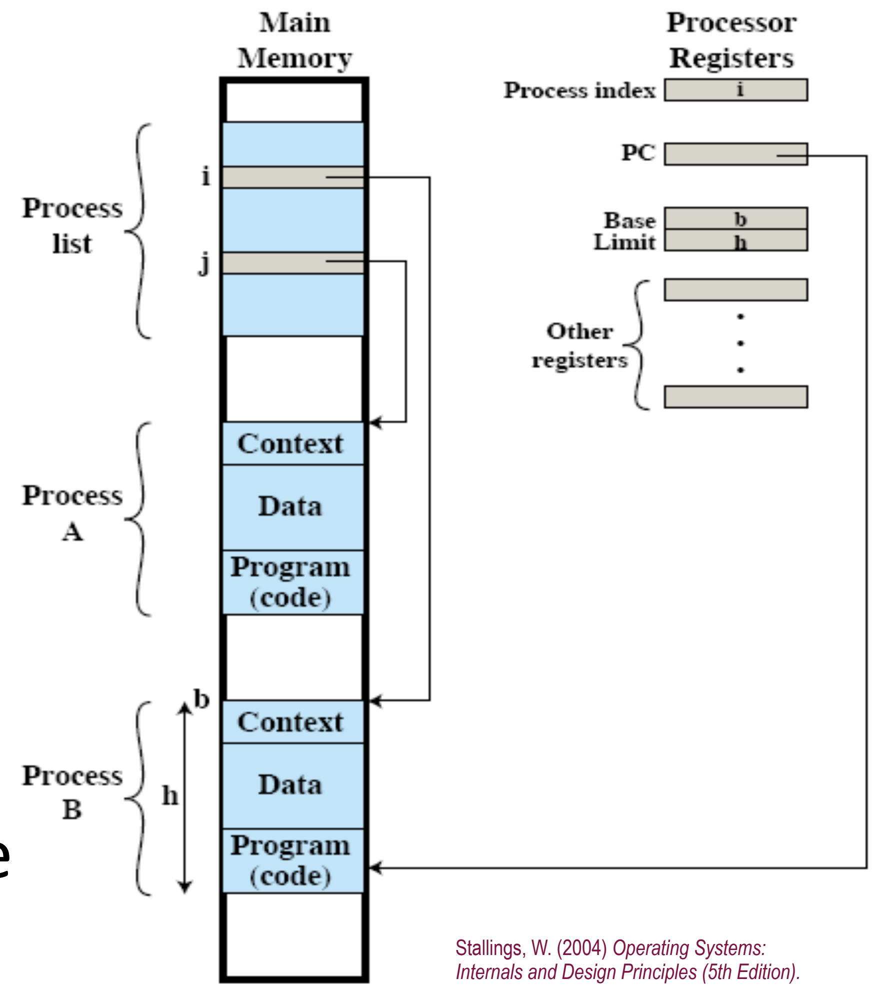
- Thread
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- Address Space with Translation
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- Dual Mode operation/Protection
  - Only the "system" has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

Process Concept

- Process is a program in execution
  - A process image consists of three components
  - an executable program
  - the associated data needed by the program

user address space

- the execution context of the process, which contains all information the O/S needs to manage the process (ID, state CPU registers, stack, etc.)



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

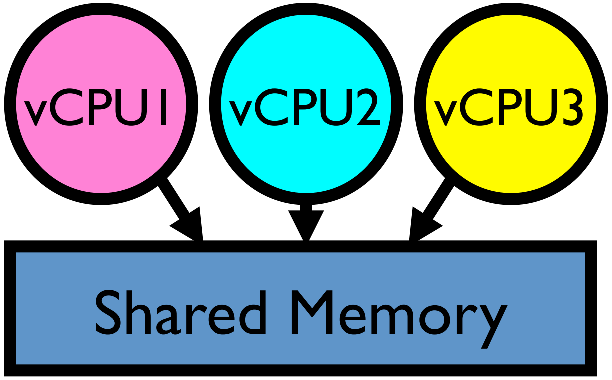
Typical process image implementation

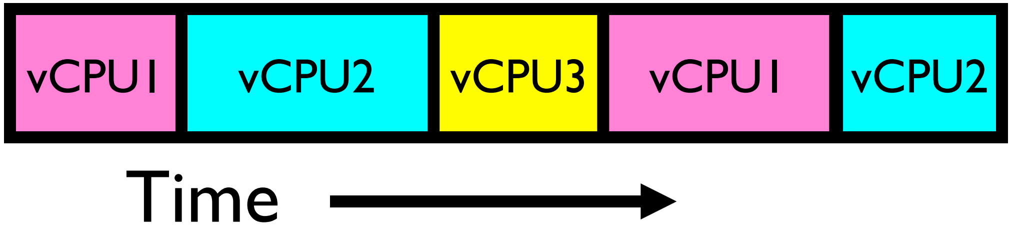
Process Control Block

(*Assume single threaded processes for now*)

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Registers, SP, … (when not running)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation tables, …
- Kernel Scheduler maintains a data structure containing the PCBs
- Scheduling algorithm selects the next one to run

Recall: give the illusion of multiple processors?

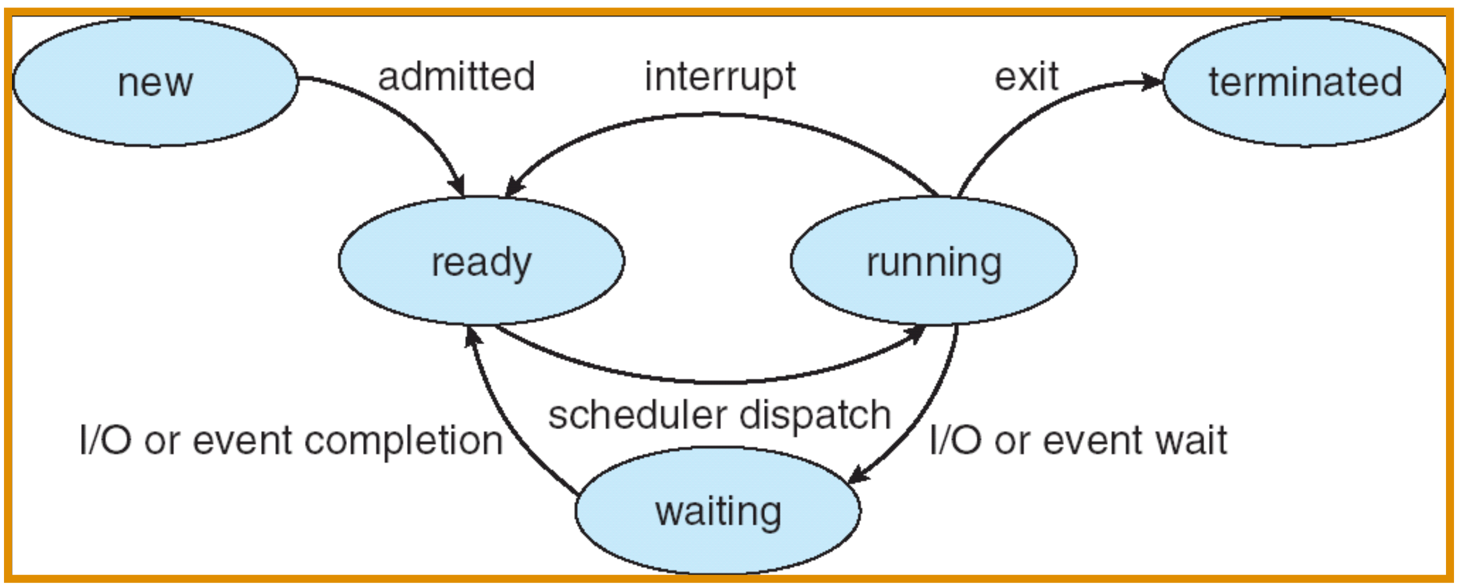




- Assume a single processor. How do we provide the *illusion* of multiple processors?
  - Multiplex in time!
  - Multiple "virtual CPUs"
- Each virtual "CPU" needs a structure to hold, i.e., PCB:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch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

Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution



Scheduler

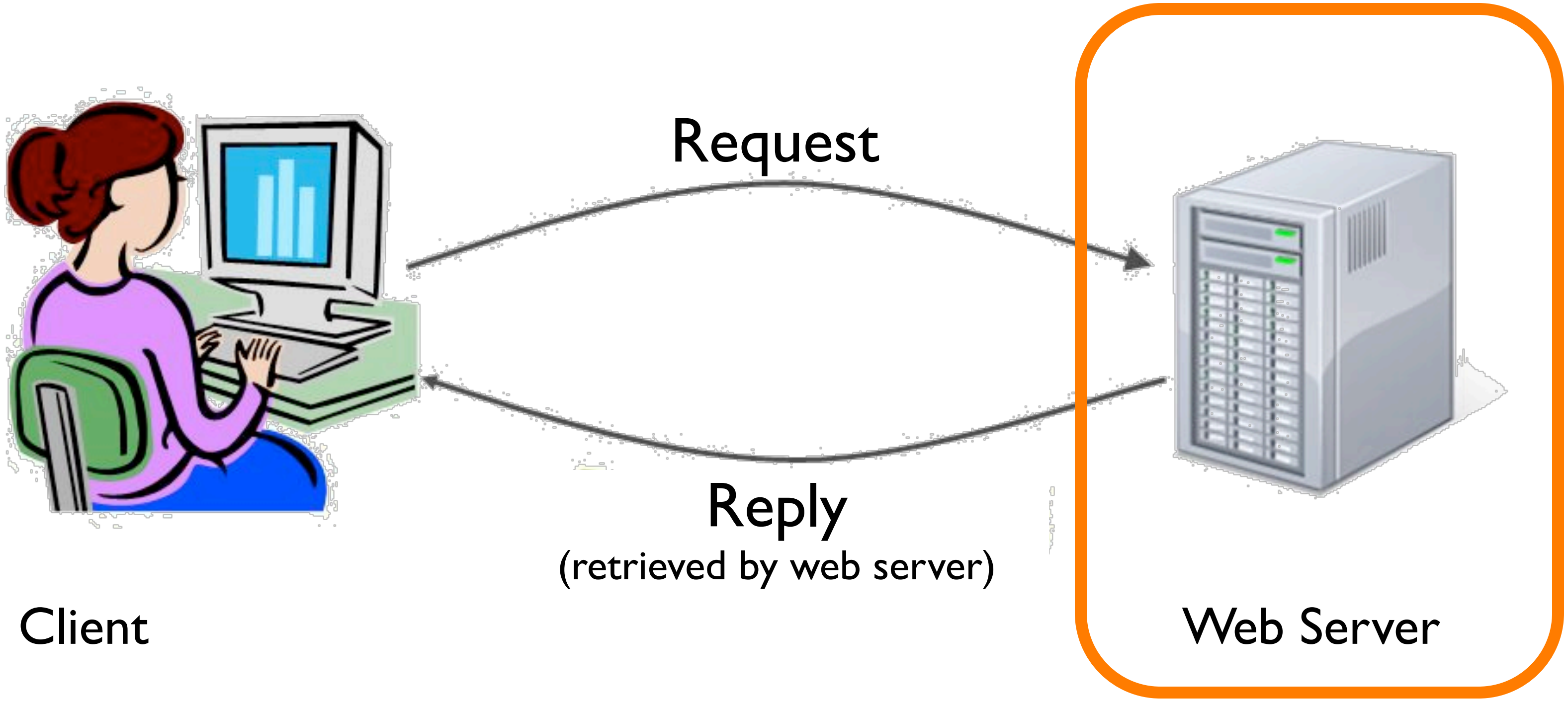
```
if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    run( nextPCB );
} else {
    run_idle_process();
}
```

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

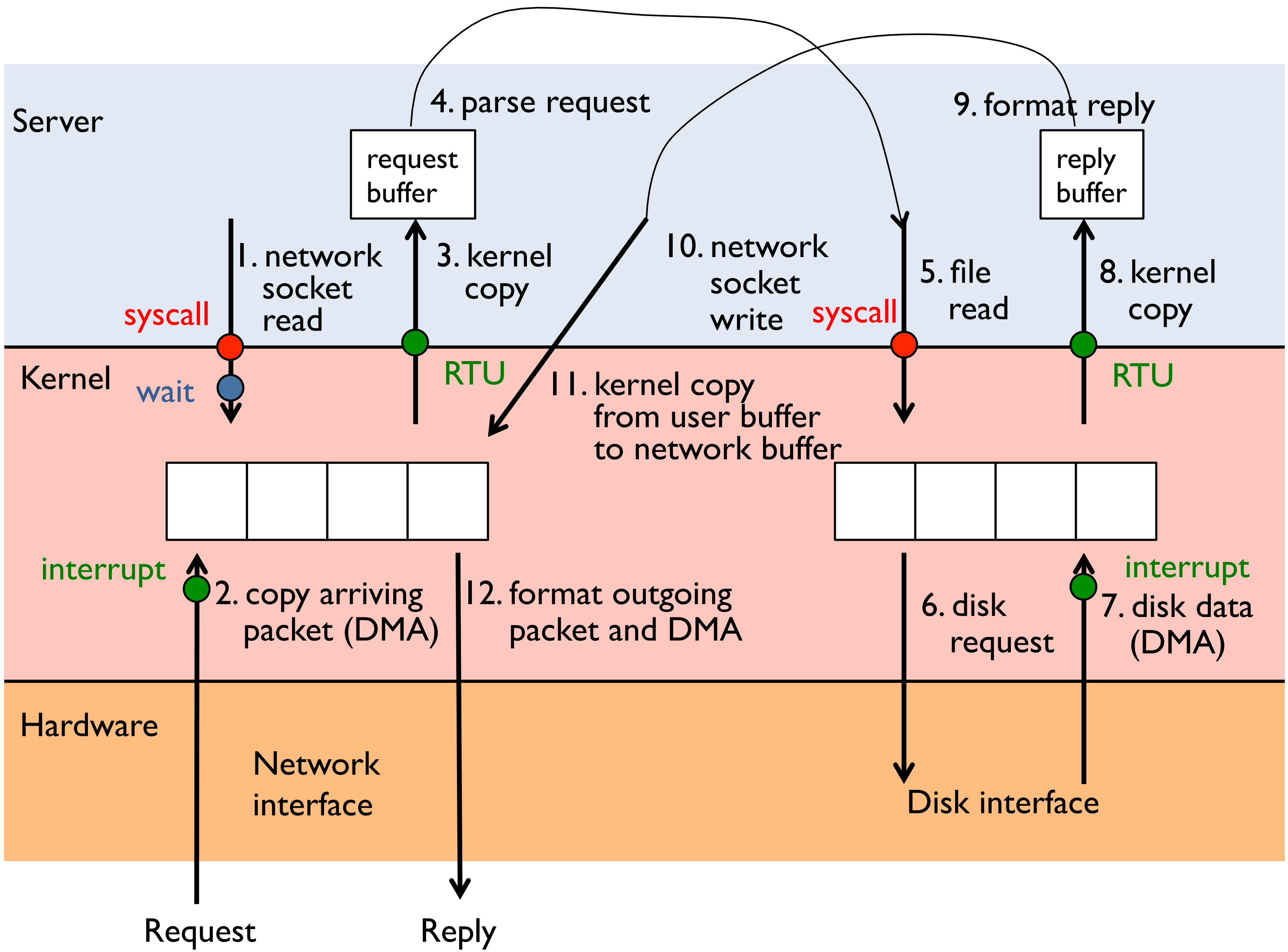
Process Creation

- Some events that lead to process creation
  - the system boots
    - when a system is initialized, several background processes or “daemons” are started (email, logon, etc.)
  - a user requests to run an application
    - by typing a command in the CLI shell or double-clicking in the GUI shell, the user can launch a new process
  - an existing process spawns a child process
    - for example, a server process (i.e. web server, file server) may create a new process for each request it handles
  - the init daemon waits for user login and spawns a shell

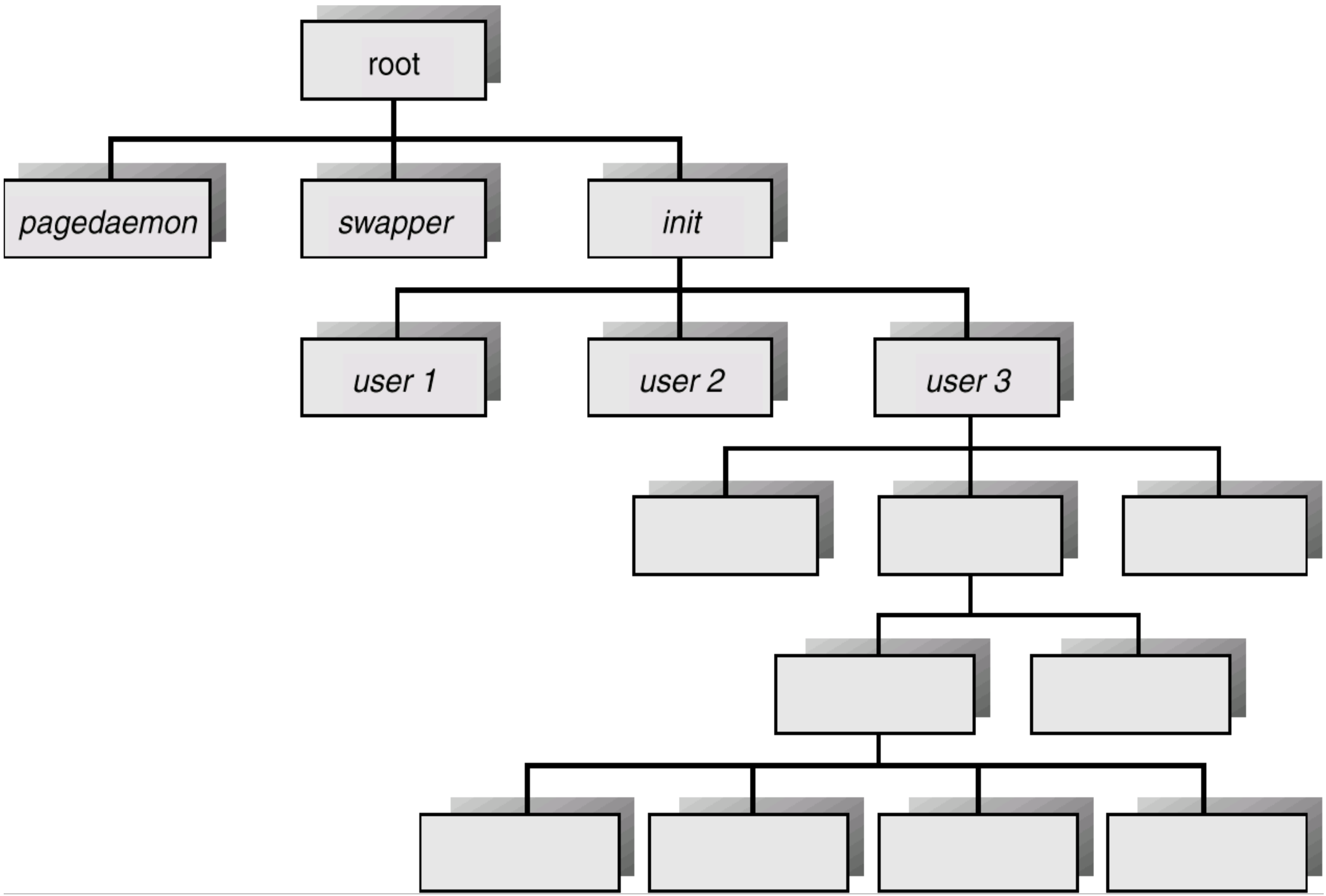
Putting it together: web server



Putting it together: web server



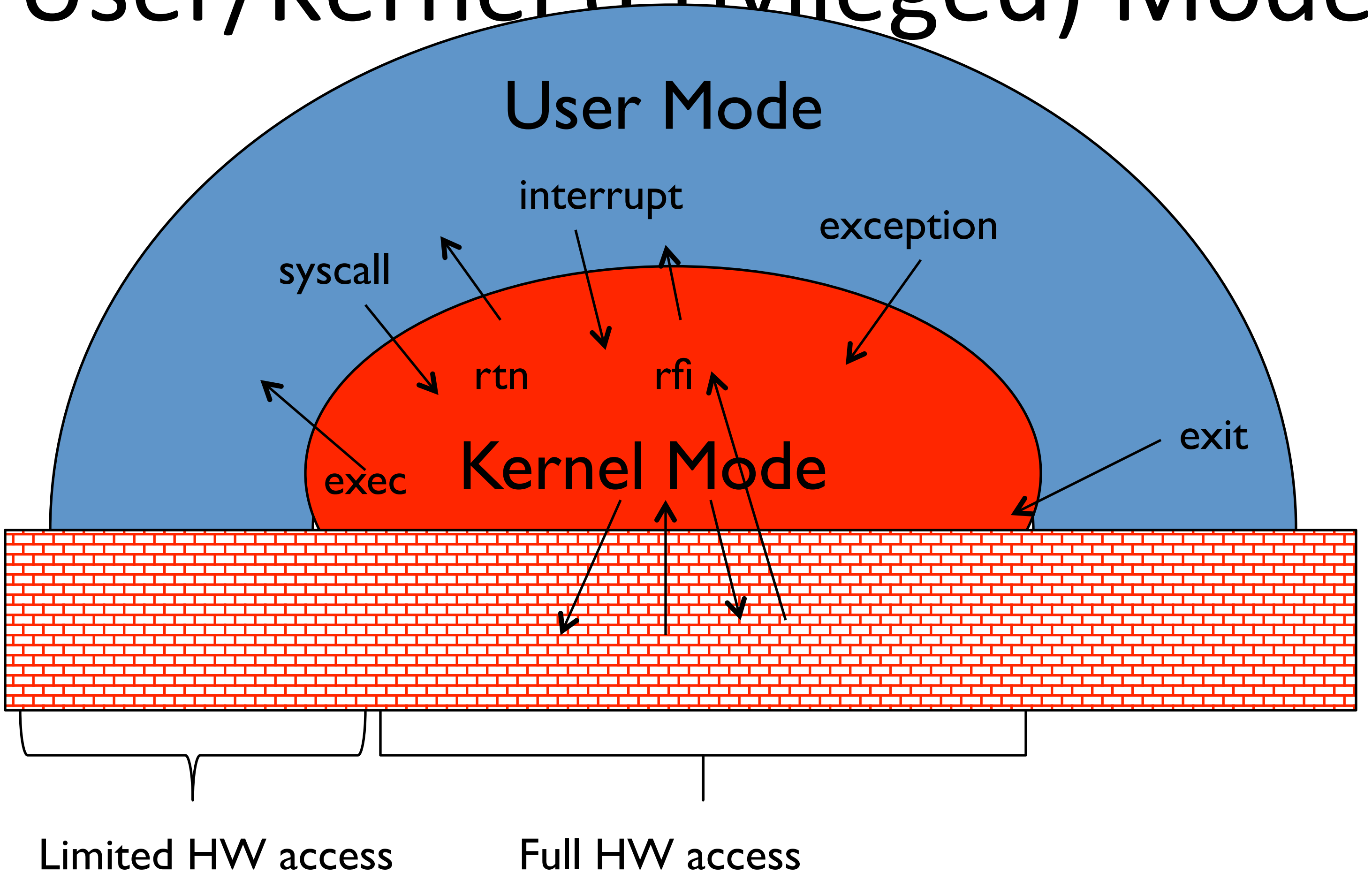
Process Tree in Linux



Recall: 3 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 arguments in registers and execute syscall
- Interrupt
  - External asynchronous event triggers context switch
  - e.g., 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, …

User/Kernel (Privileged) Mode

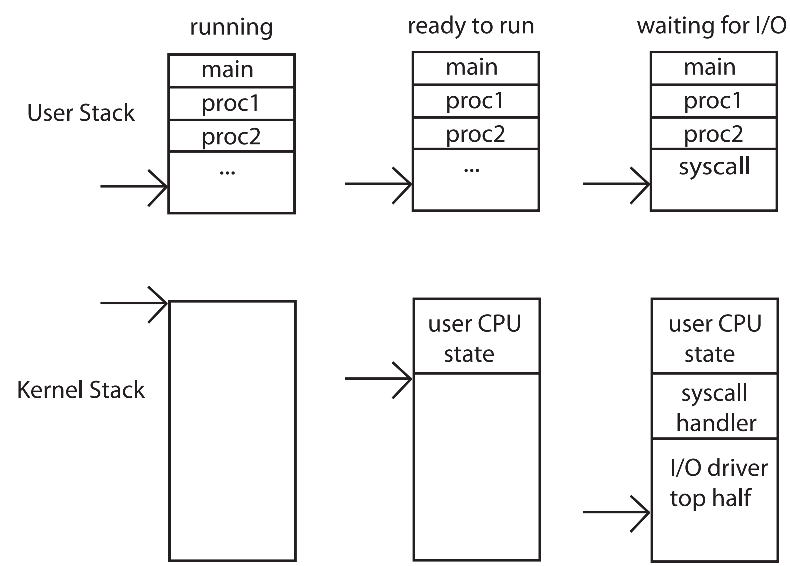


Implementing Safe Kernel Mode Transfers

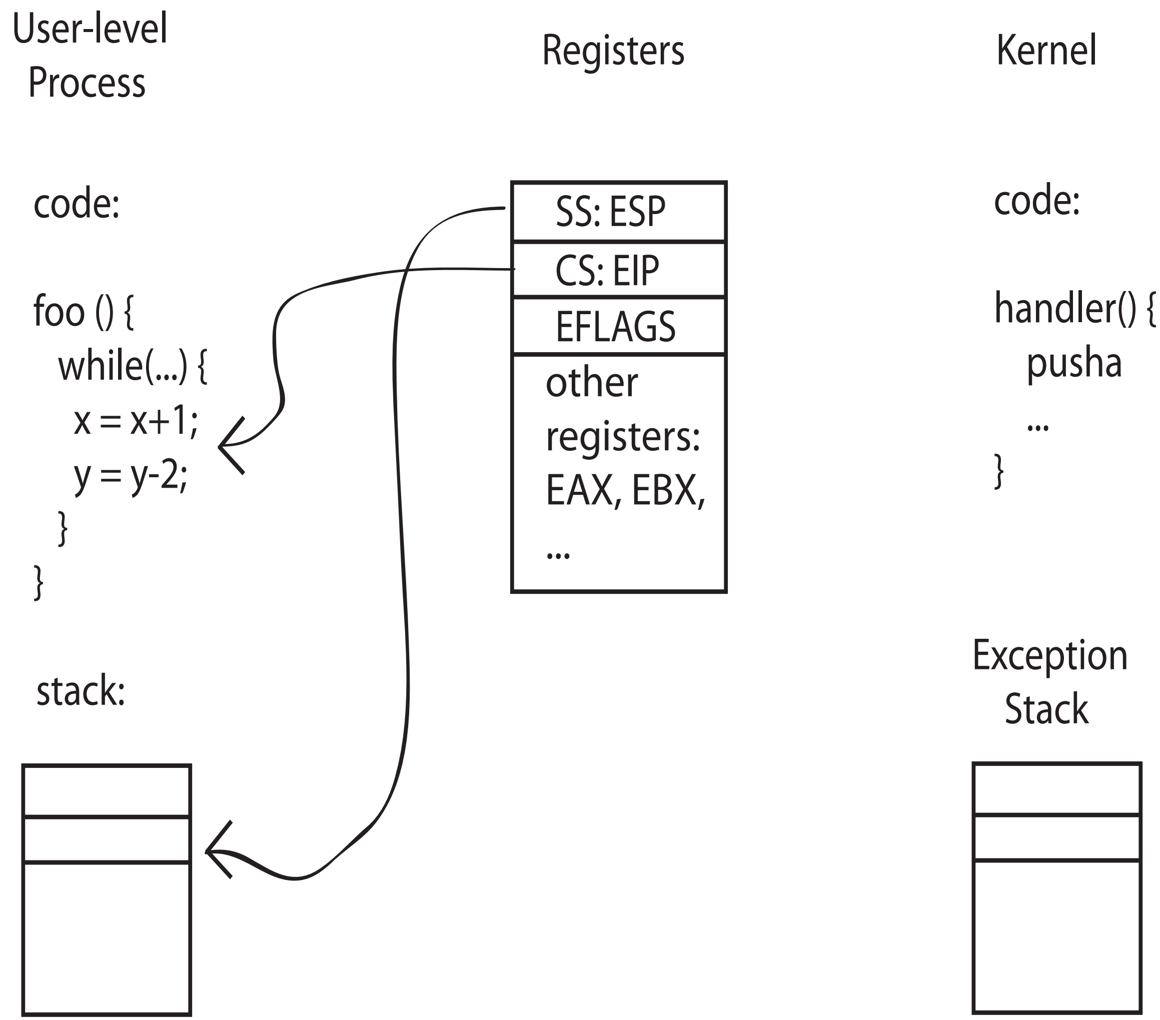
- Important aspects:
  - Separate kernel stack
  - Controlled transfer into kernel (e.g., syscall table)
- Carefully constructed kernel code packs up the user process state and sets it aside
  - Details depend on the machine architecture
- Should be impossible for buggy or malicious user program to cause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

Need for Separate Kernel Stacks

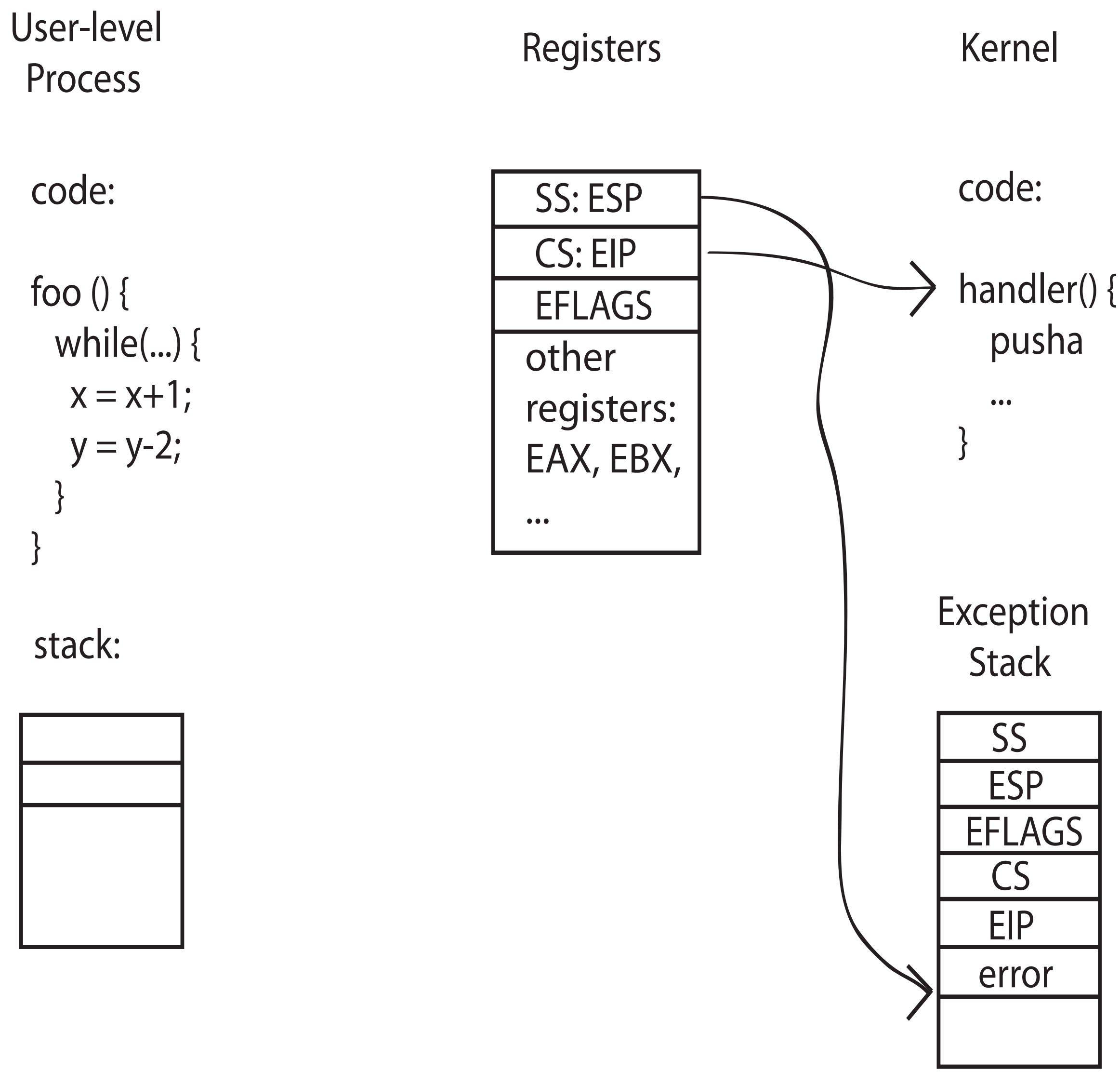
- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
  - Interrupts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



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?

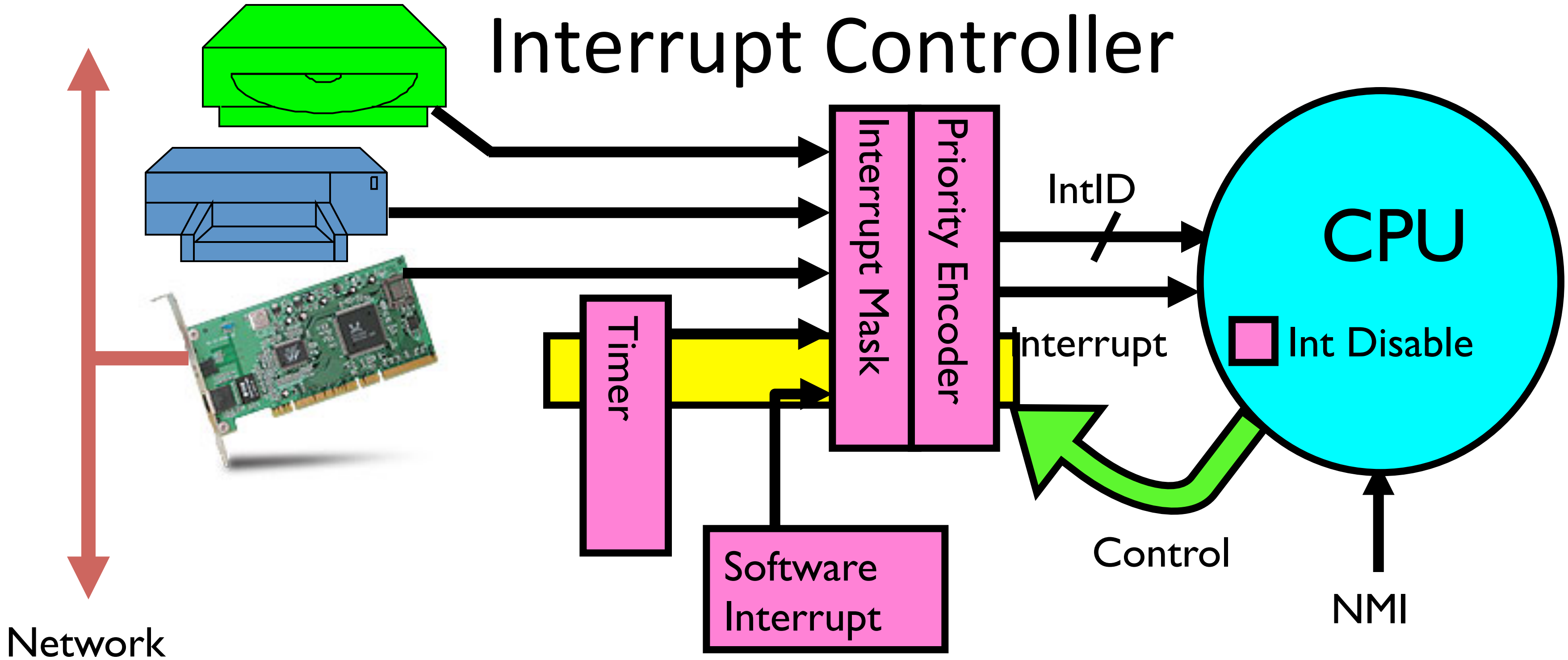
- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    - wake up an existing OS thread

Hardware support: Interrupt Control

- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall

- HW may have multiple levels of interrupt
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain Non-Maskable-Interrupts (NMI)
    - e.g., kernel segmentation fault

Interrupt Controller



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can't be disabled

How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking
- Atomic transfer of control
  - "Single instruction"-like to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

Can a process create a process ?

- Yes! Unique identity of process is the “process ID” (or PID)
- **fork()** system call creates a *copy* of current process with a new PID
- Return value from **fork()**: 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
- All state of original process duplicated in both Parent and Child!
  - Memory, File Descriptors (next topic), etc…

fork1.c

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

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  size_t readlen, writelen, slen;
  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");
    exit(1);
  }
  exit(0);
}
```

fork2.c

```
int status;
…
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 Races: fork3.c

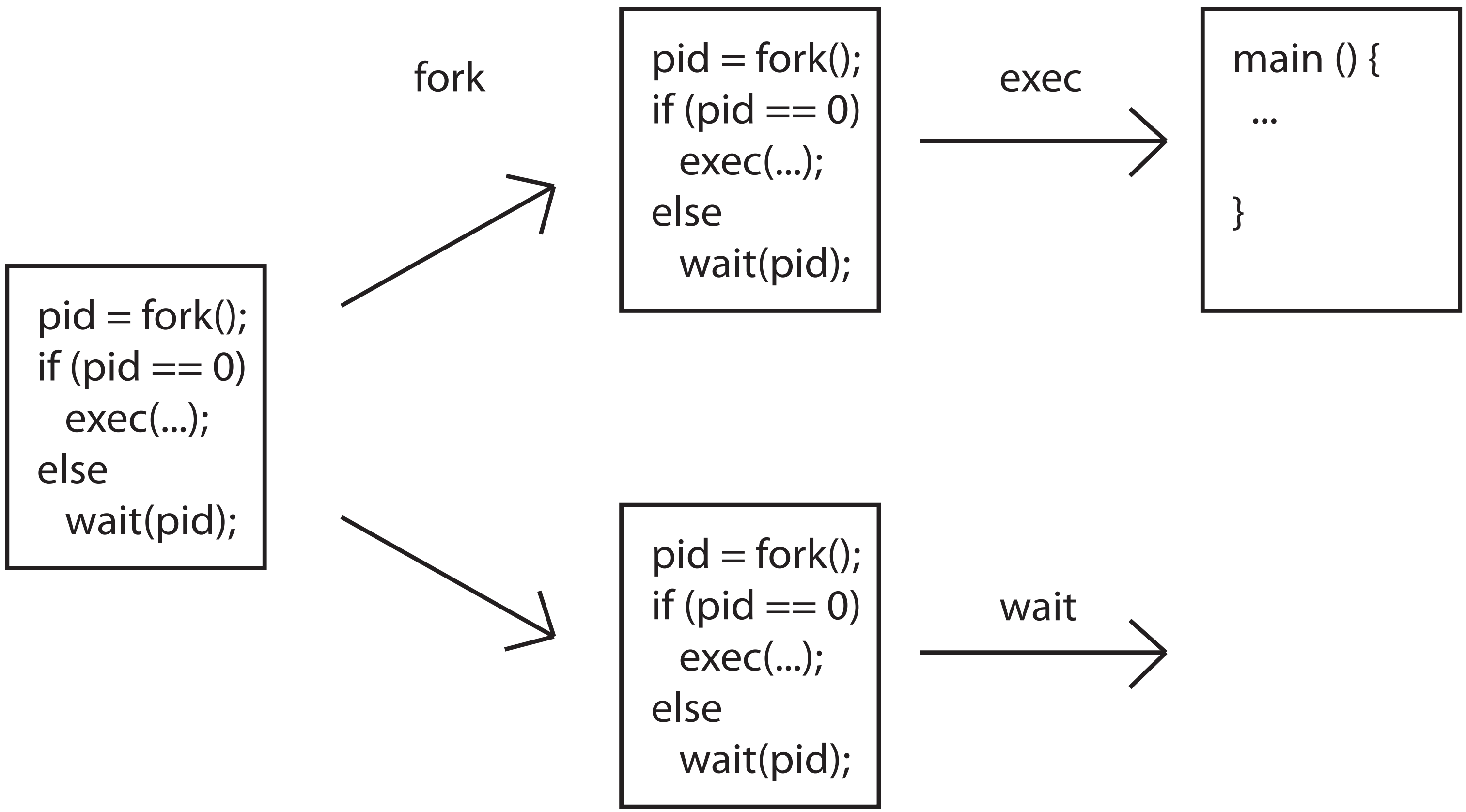
```
int i;
cpid = fork();
if (cpid > 0) {
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    for (i=0; i<10; i++) {
      printf("[%d] parent: %d\n", mypid, i);
      // sleep(1);
    }
  }  else if (cpid == 0) {
    mypid = getpid();
    printf("[%d] child\n", mypid);
    for (i=0; i>-10; i--) {
      printf("[%d] child: %d\n", mypid, i);
      // sleep(1);
    }
  }
```

- Question: What does this program print?
- Does it change if you add in one of the sleep() statements?

UNIX Process Management

- UNIX `fork` – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX `exec` – system call to *change the program* being run by the current process
- UNIX `wait` – system call to wait for a process to finish
- UNIX `signal` – system call to send a notification to another process
- UNIX man pages: `fork`(2), `exec`(3), `wait`(2), `signal`(3)

UNIX Process Management



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

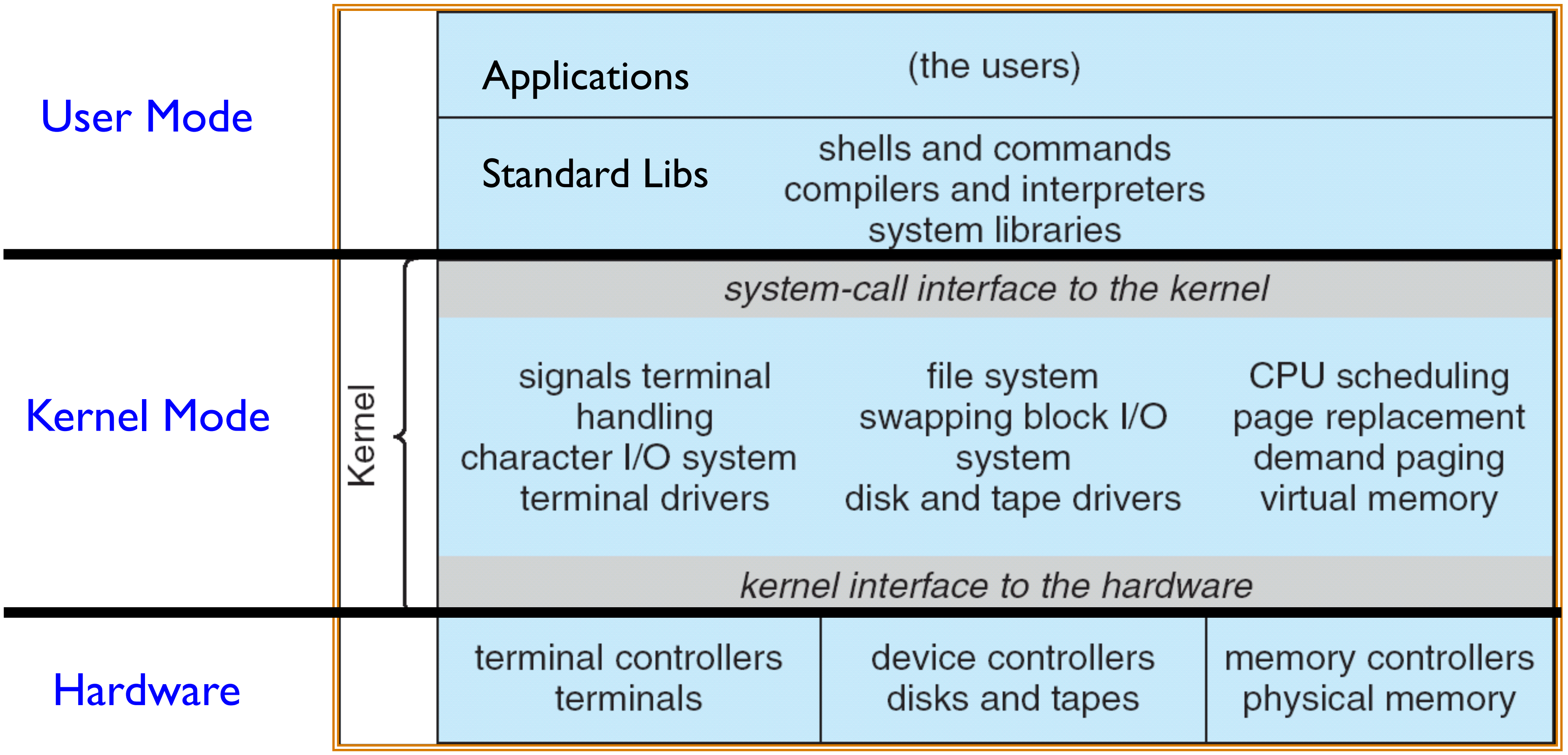
ln –o program sourcefile1.o sourcefile2.o

./program

Signals – infloop.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 %d - phew!\n",signum);
  exit(1);
}

int main() {
  signal(SIGINT, signal_callback_handler);

  while (1) {}
}
```

Recall: UNIX System Structure

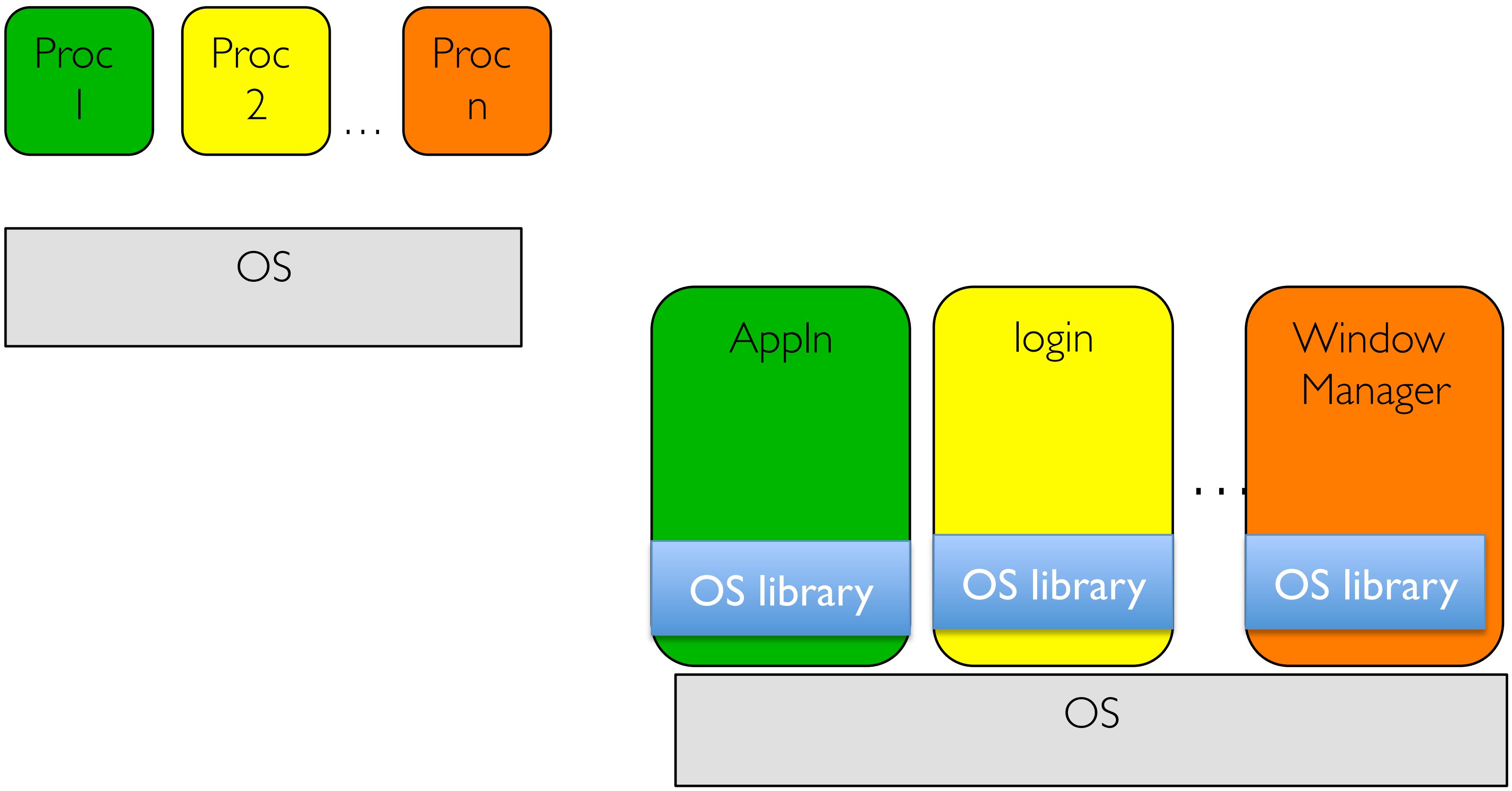


How Does the Kernel Provide Services?

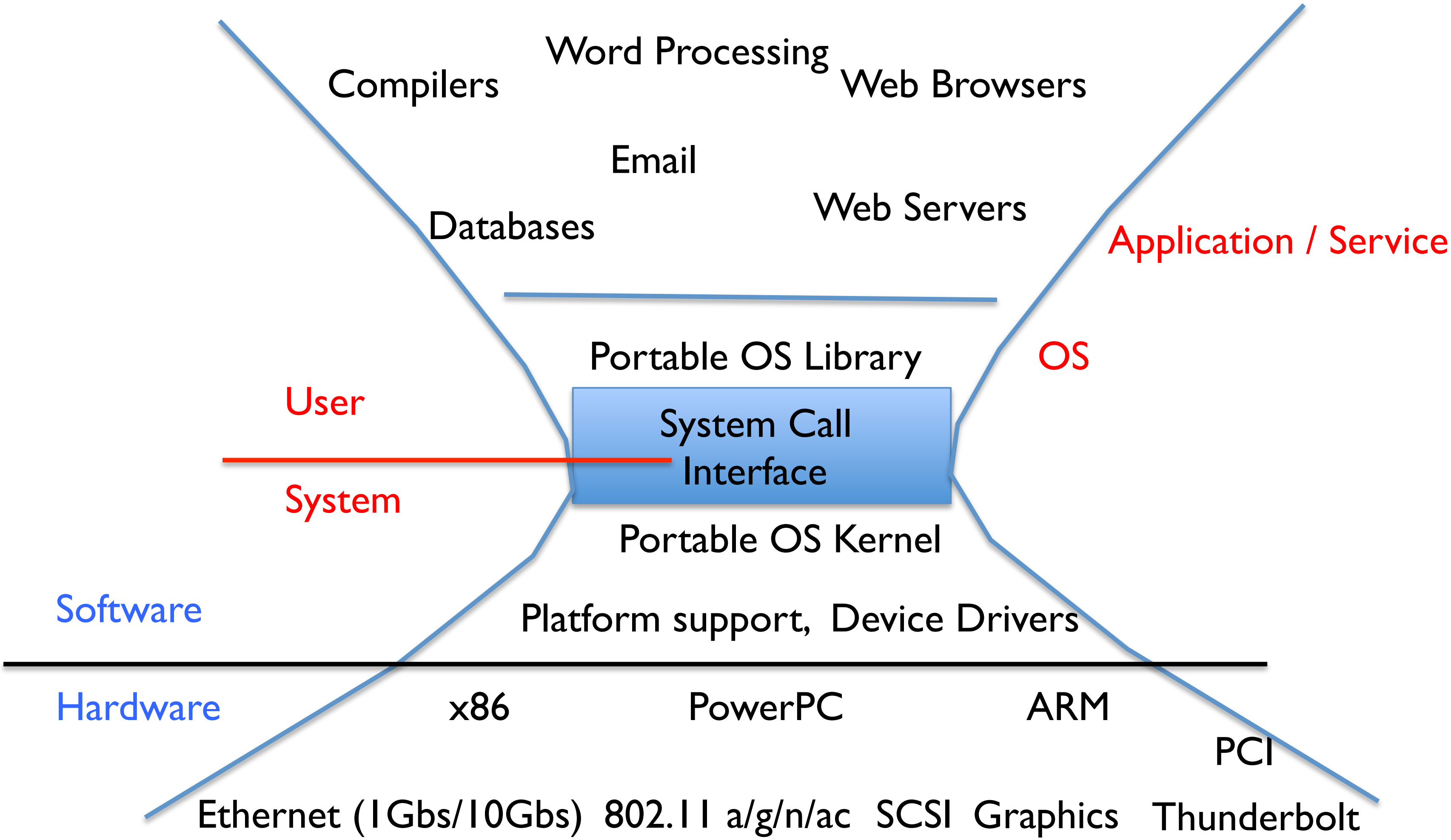
- You said that applications request services from the operating system via **syscall**, but …
- I've been writing all sort of useful applications and I never ever saw a "syscall" !!!

- That's right.
- It was buried in the programming language runtime library (e.g., libc.a)
- … Layering

OS Run-Time Library



A Kind of Narrow Waist

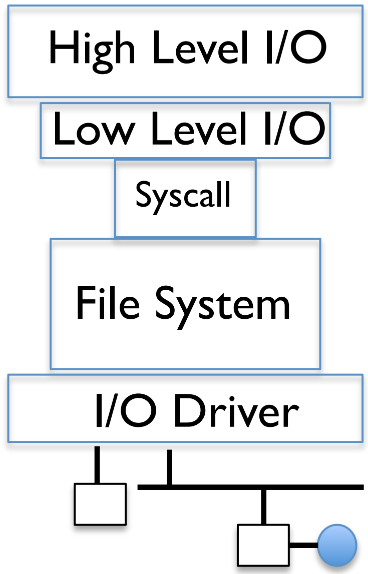


Key Unix I/O Design Concepts

- Uniformity
  - file operations, device I/O, and interprocess communication through open, read/write, close
  - Allows simple composition of programs
    - find | grep | wc …
- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  - Streaming and block devices looks the same
  - read blocks process, yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close

I/O & Storage Layers

Application / Service



streams

handles

registers

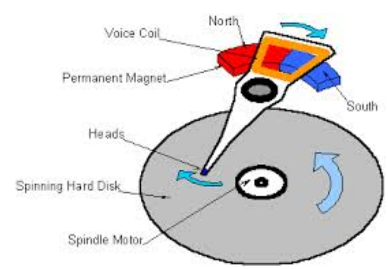
descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA







Summary

- Process: execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources
- Interrupts
  - Hardware mechanism for regaining control from user
  - Notification that events have occurred
  - User-level equivalent: Signals
- Native control of Process
  - Fork, Exec, Wait, Signal
- Basic Support for I/O
  - Standard interface: open, read, write, seek
  - Device drivers: customized interface to hardware

Logistics – New action Items

- Assignment 1 out
  - Read the code
  - Compile/test
  - Learn structure – use *printf*s where you can to understand flow
- Test sample programs from class