

# Processes

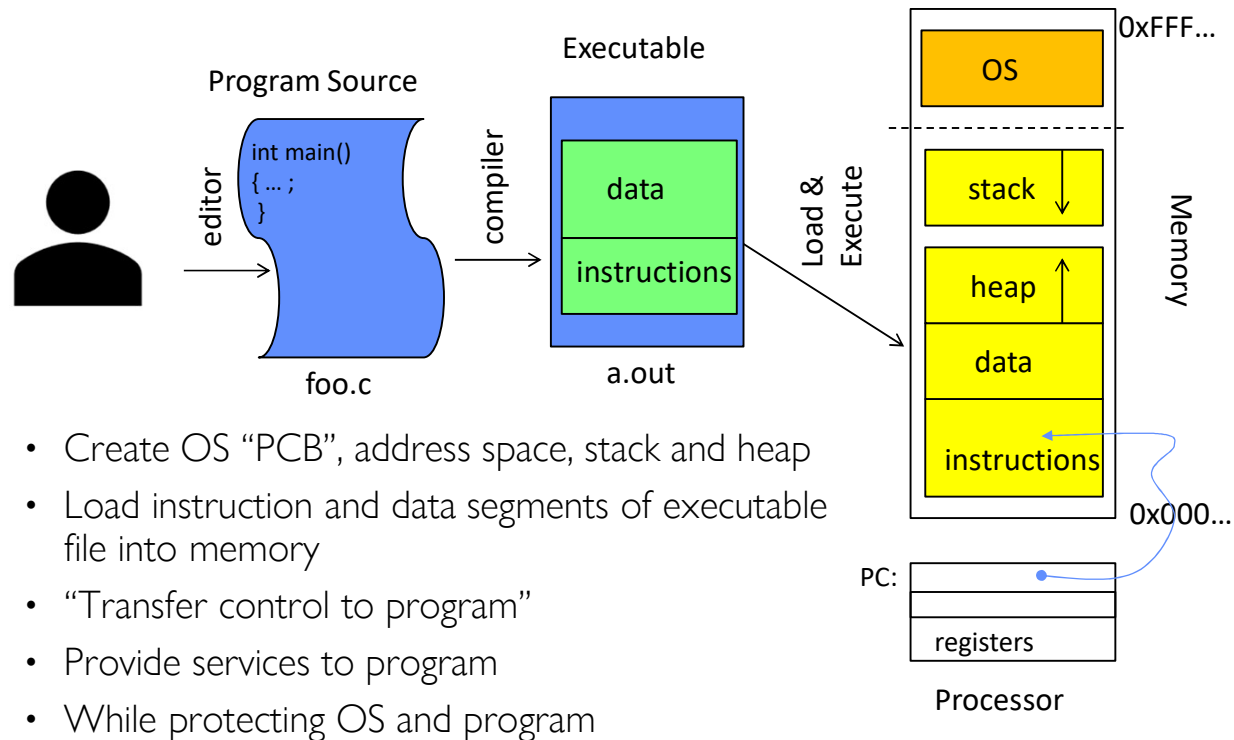
Some slides were taken from John Kubiawicz's Notes, UC Berkely

## Four Fundamental OS Concepts

---

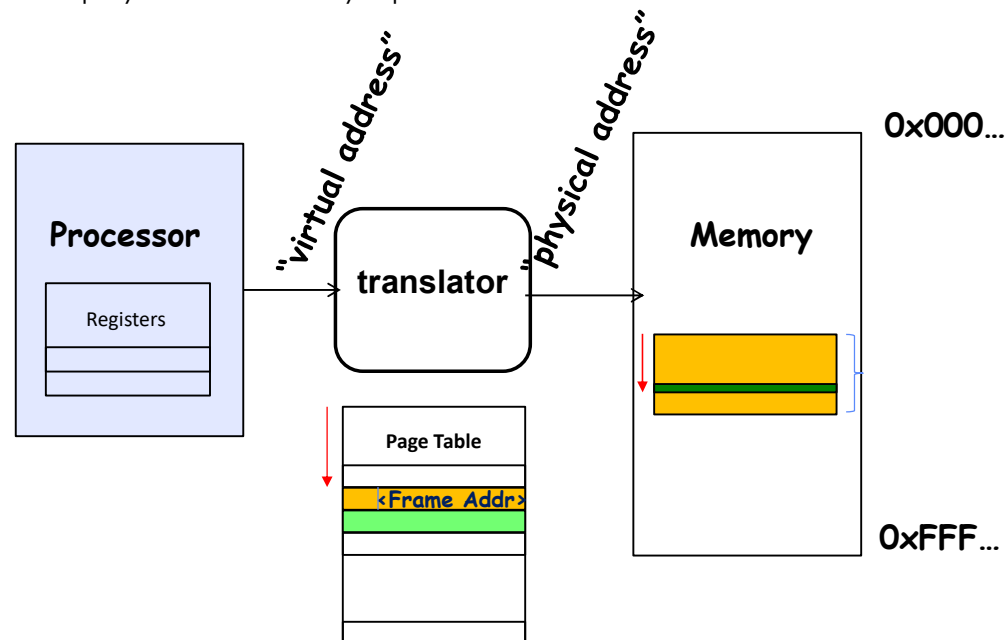
- Thread: Execution Context
  - Fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- Address space (with or w/o translation)
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine (in which 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

## OS Bottom Line: Run Programs



## Protected Address Space

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



## How to provide the illusion of many CPUs?

---

- CPU virtualizing
  - The OS can promote the illusion that many virtual CPUs exist.
  - **Time sharing**: Running one process, then stopping it and running another
    - » The potential cost is **performance**.

## A Process

---

**A process is a running program.**

- Comprising of a process:
  - Memory (address space)
    - » Instructions
    - » Data section
  - Registers
    - » Program counter
    - » Stack pointer

## Process API

---

- These APIs are available on any modern OS.
  - **Create**
    - » Create a new process to run a program
  - **Destroy**
    - » Halt a runaway process
  - **Wait**
    - » Wait for a process to stop running
  - **Miscellaneous Control**
    - » Some kind of method to suspend a process and then resume it
  - **Status**
    - » Get some status info about a process

## Process Creation

---

1. **Load** a program code into memory, into the address space of the process.
  - Programs initially reside on disk in *executable format*.
  - OS perform the loading process *lazily*.
    - » Loading pieces of code or data only as they are needed during program execution.
2. The program's run-time **stack** is allocated.
  - Use the stack for *local variables*, *function parameters*, and *return address*.
  - Initialize the stack with arguments → `argc` and the `argv` array of `main()` function

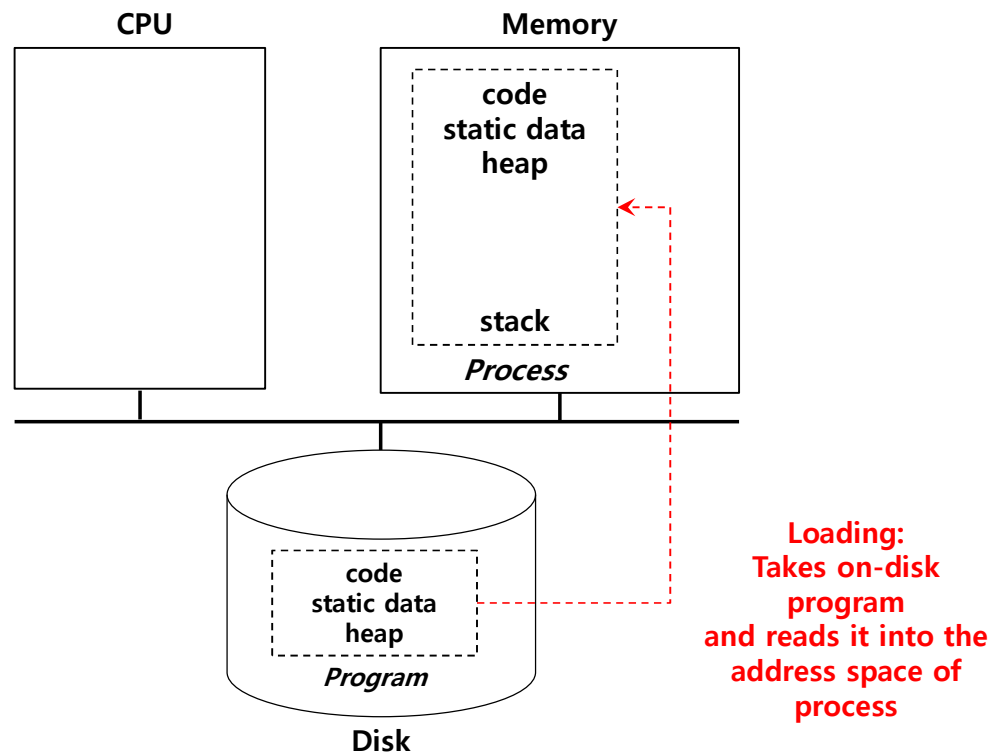


## Process Creation (Cont.)

---

3. The program's **heap** is created.
  - Used for explicitly requested dynamically allocated data.
  - Program request such space by calling `malloc()` and free it by calling `free()`.
4. The OS do some other initialization tasks.
  - input/output (I/O) setup
    - » Each process by default has three open file descriptors.
    - » Standard input, output and error
5. **Start the program** running at the entry point, namely `main()`.
  - The OS *transfers control* of the CPU to the newly-created process.

## Loading: From Program To Process



10

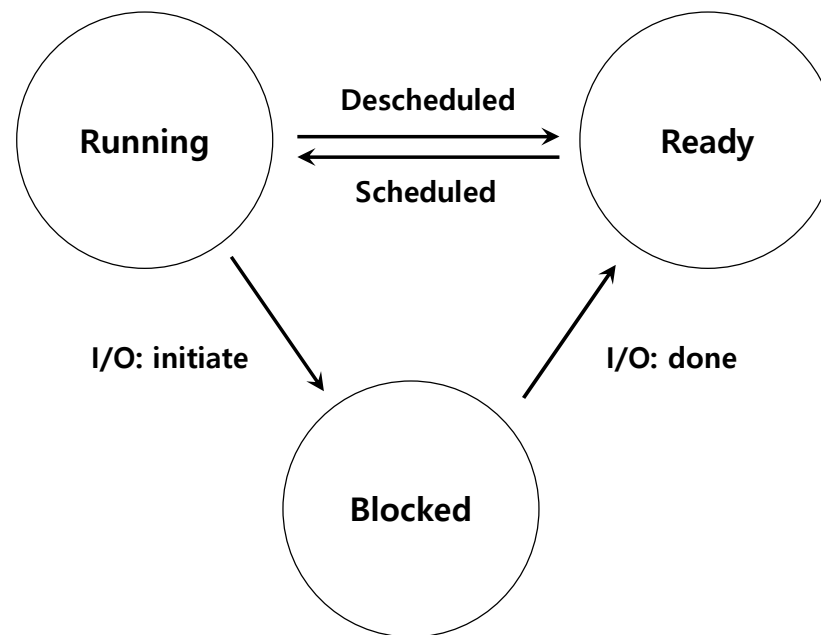
## Process States

---

- A process can be one of three states.
  - **Running**
    - » A process is running on a processor.
  - **Ready**
    - » A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
  - **Blocked**
    - » A process has performed some kind of operation.
    - » When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

## Process State Transition

---



12

## Data structures

---

- The OS has some key data structures that track various relevant pieces of information.
  - Process list
    - » Ready processes
    - » Blocked processes
    - » Current running process
  - Register context
- PCB(Process Control Block)
  - A C-structure that contains information about each process.

## Example: The xv6 kernel Proc Structure

---

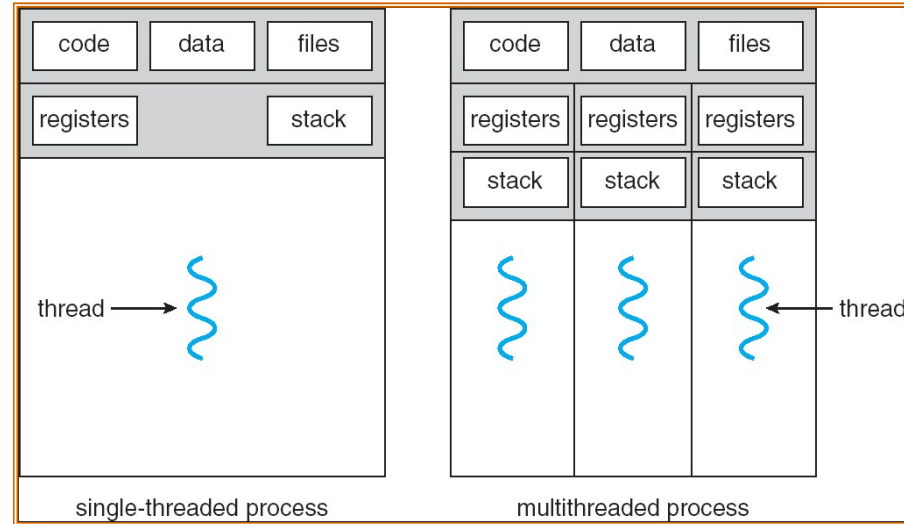
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

---

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                // Start of process memory
    uint sz;                  // Size of process memory
    char *kstack;             // Bottom of kernel stack
                                // for this process
    enum proc_state state;    // Process state
    int pid;                   // Process ID
    struct proc *parent;      // Parent process
    void *chan;               // If non-zero, sleeping on chan
    int killed;               // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;         // Current directory
    struct context context;    // Switch here to run process
    struct trapframe *tf;      // Trap frame for the
                                // current interrupt
};
```

## 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?



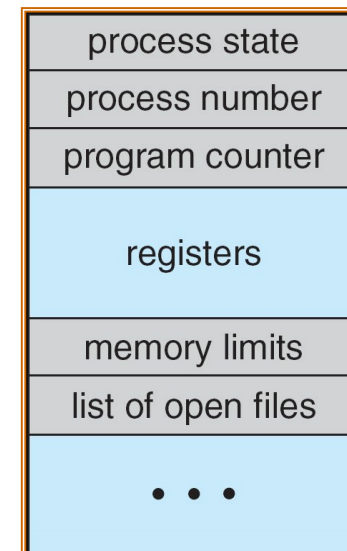
## Running Many Programs

---

- 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 user processes in the OS?
  - 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 a lot of memory?

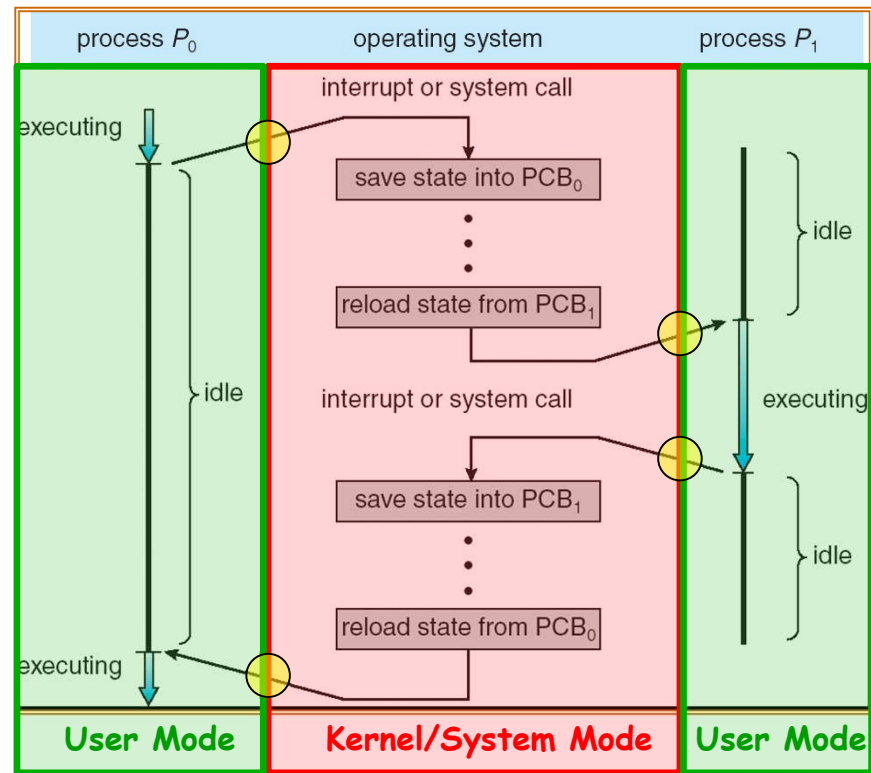
## Multiplexing Processes: The Process Control Block

- 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



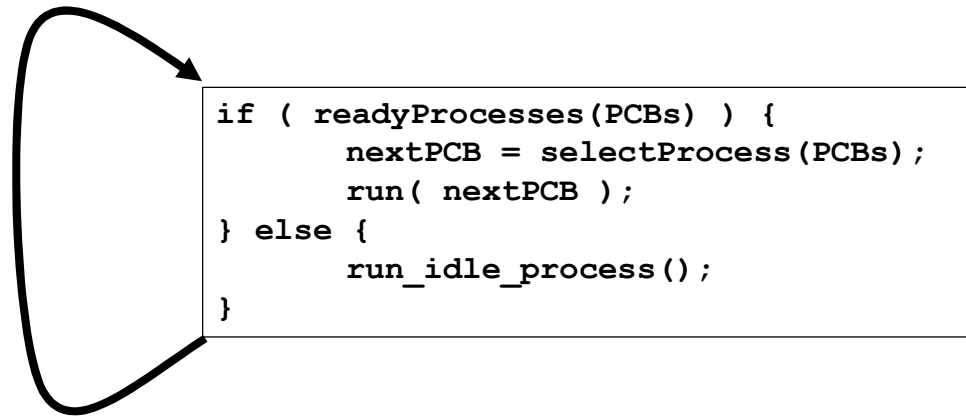
Process  
Control  
Block

## CPU Switch From Process A to Process B



## Scheduler

---



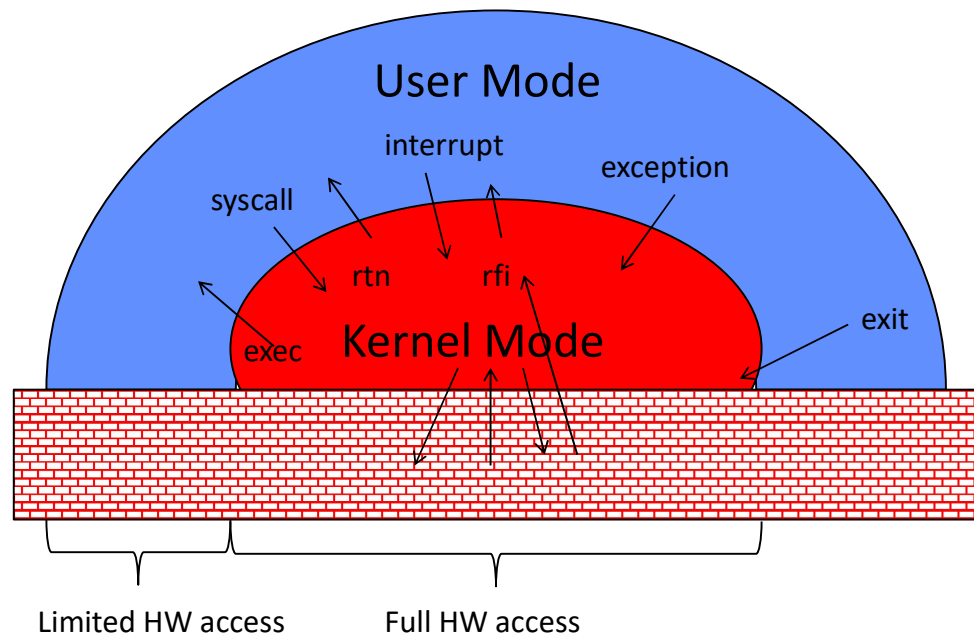
- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide ...
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

## 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 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, ...

## User/Kernel (Privileged) Mode



## Implementing Safe Kernel Mode Transfers

---

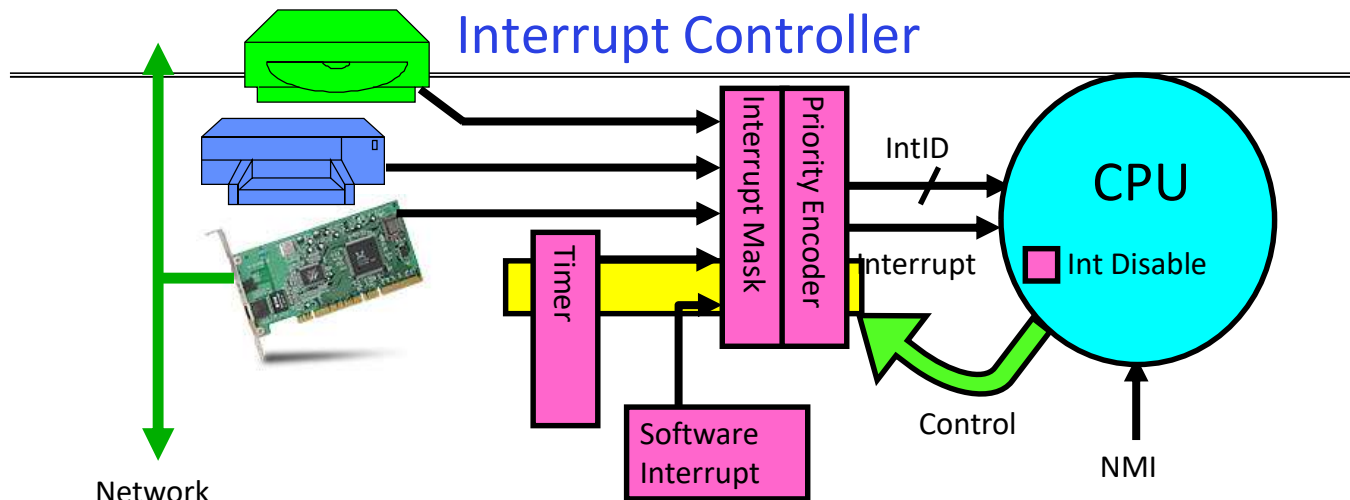
- Important aspects:
  - Controlled transfer into kernel (e.g., syscall table)
  - Separate kernel stack!
- Carefully constructed kernel code packs up the user process state and sets it aside
  - Details depend on the machine architecture
  - More on this next time
- Should be impossible for buggy or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

## Hardware support: Interrupt Control

---

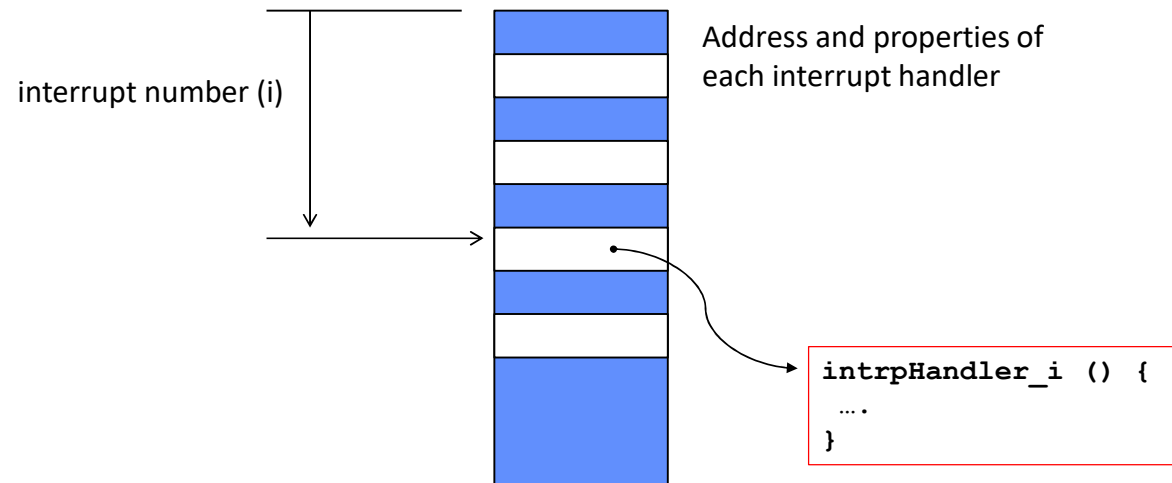
- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?
- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (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





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

## Interrupt Vector



- Where else do you see this dispatch pattern?
  - System Call
  - Exceptions

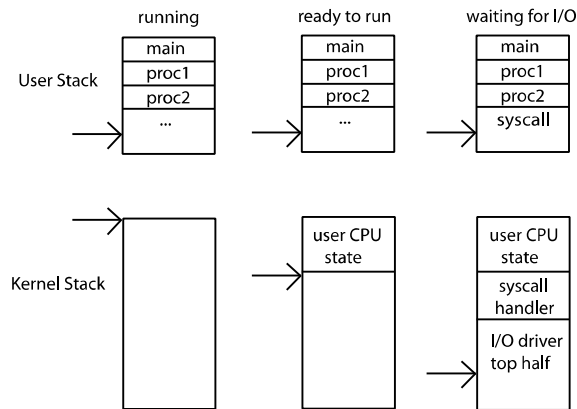
## How do we take interrupts safely?

---

- 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

## 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

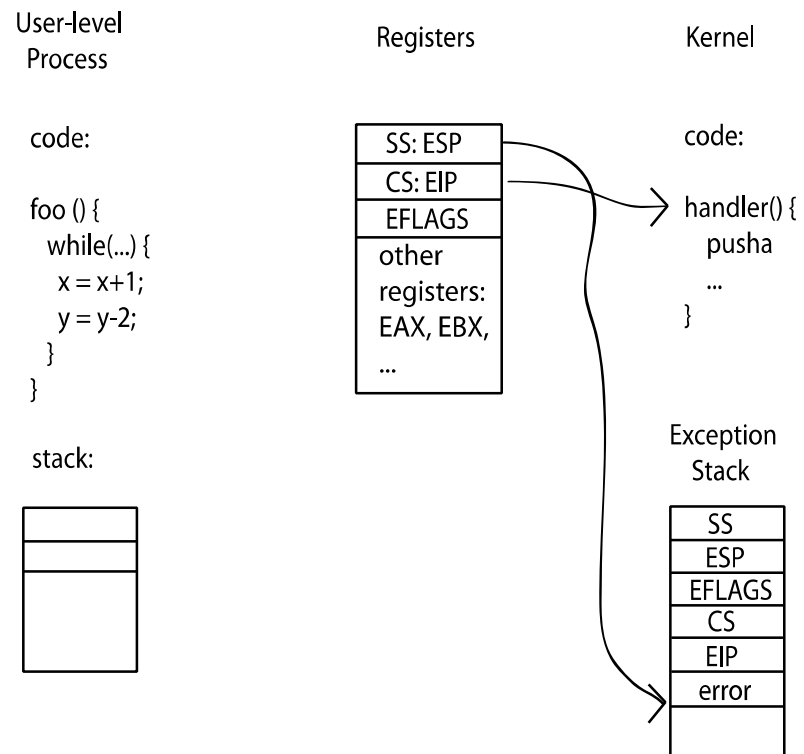
code:

```
handler() {  
  pusha  
  ...  
}
```

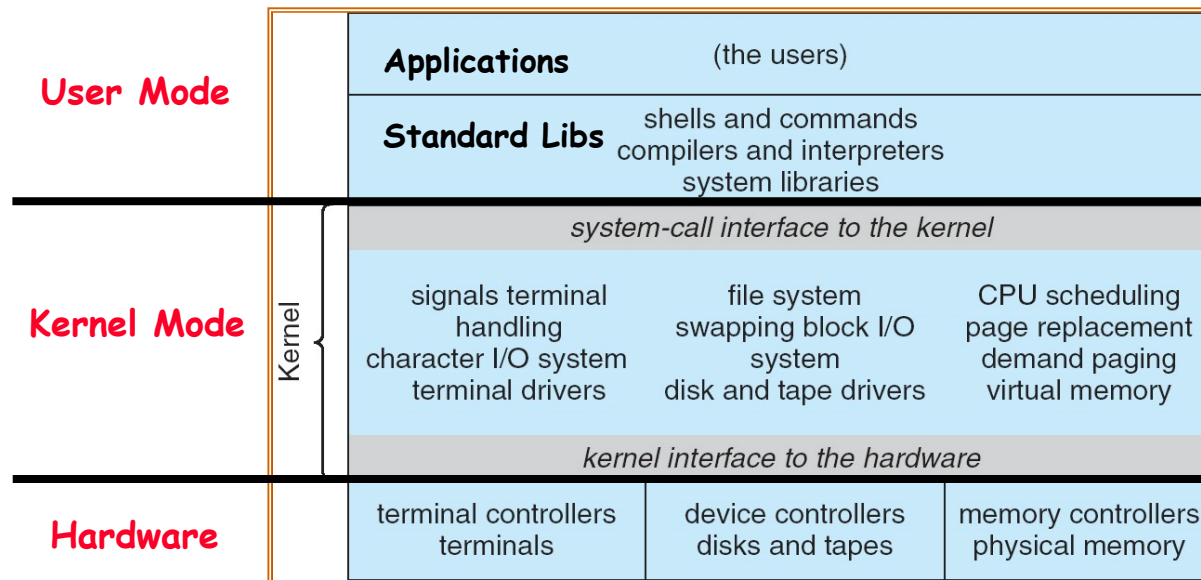
Exception  
Stack



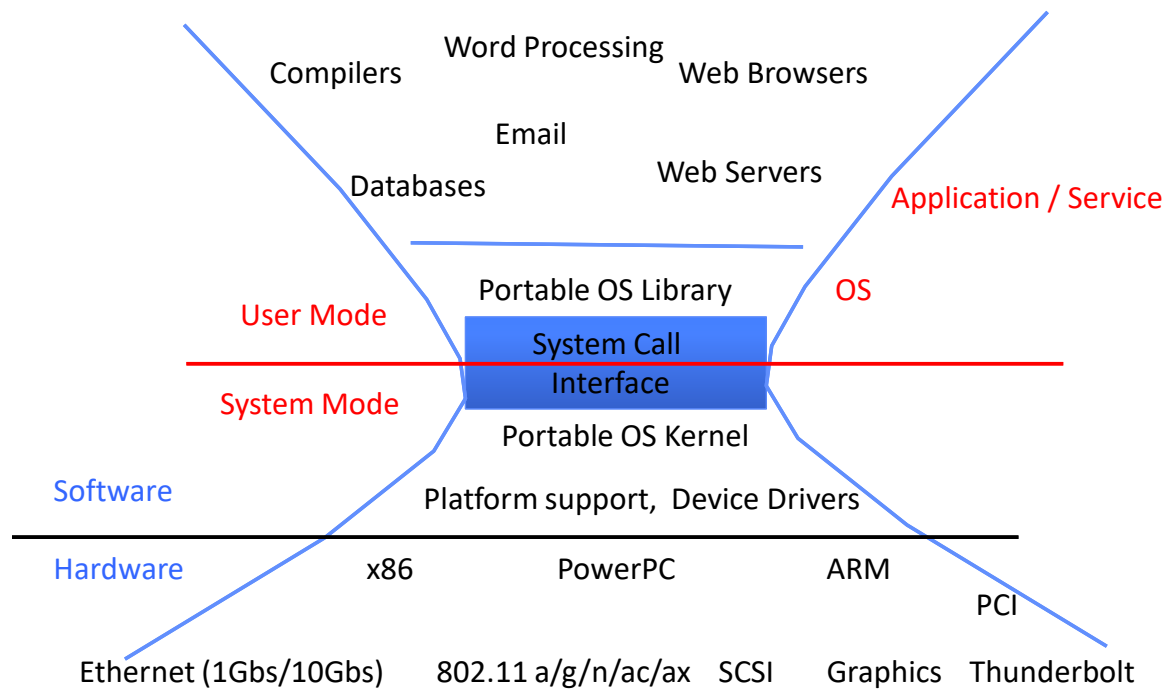
## During Interrupt/System Call



## UNIX System Structure



## A Narrow Waist





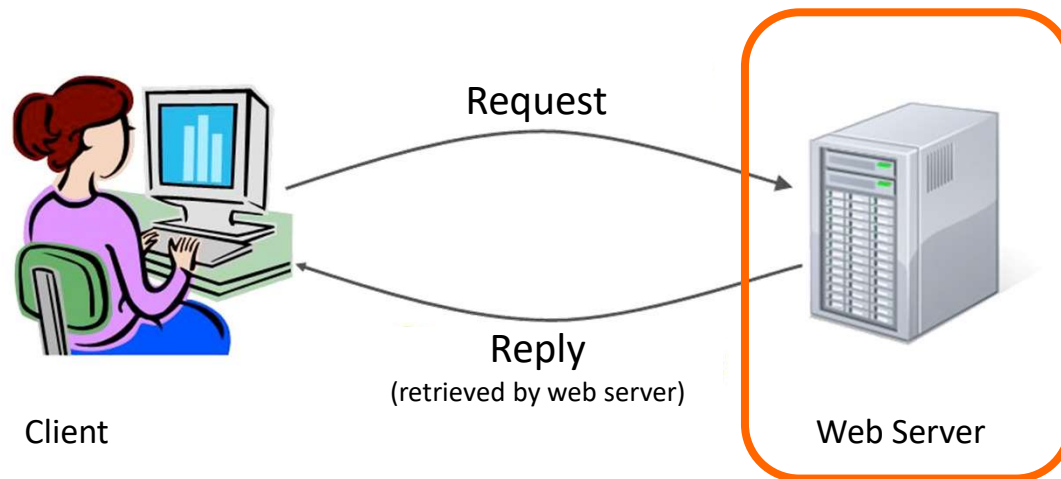
## Kernel System Call Handler

---

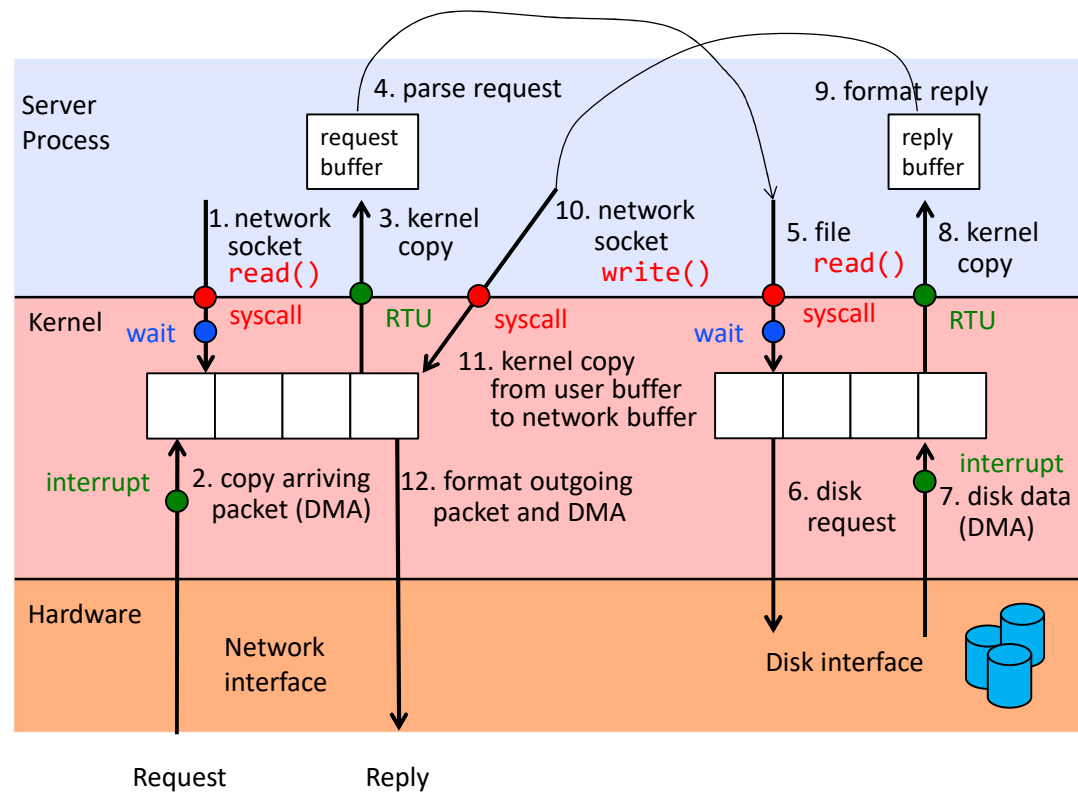
- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory

## Putting it together: web server

---



## Putting it together: web server



## Bootstrapping

---

- If processes are created by other processes, how does the first process start?
- First process is started by the kernel
  - Often configured as an argument to the kernel *before* the kernel boots
  - Often called the “init” process
- After this, all processes on the system are created by other processes

## Process Management API

---

- **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

## What API does the OS provide to user programs

---

- API = Application Programming Interface
  - = functions available to write user programs
- API provided by OS is a set of “system calls”
  - System call is a function call into OS code that runs at a higher privilege level of the CPU
  - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level
  - Some “blocking” system calls cause the process to be blocked and descheduled (e.g., `read` from disk)

## So, Should we rewrite programs for each OS?

---

- POSIX API: a standard set of system calls that an OS must implement
  - Programs written to the POSIX API can run on any POSIX compliant OS
  - Most modern OSes are POSIX compliant
  - Ensures program portability
- Program language libraries hide the details of invoking system calls
  - The `printf` function in the C library calls the `write` system call to write to screen
  - User programs usually do not need to worry about invoking system calls

## Process Management API

---

- **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



## pid.c

---

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

    exit(0);
}
```

Q: What if we let main return without ever calling exit?

- The OS Library calls exit() for us!
- The entrypoint of the executable is in the OS library
- OS library calls main
- If main returns, OS library calls exit
- You'll see this in Project 0: init.c

## Process Management API

---

- **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
  - New process contains a single thread
- 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


---

```
#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

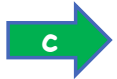
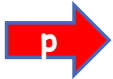
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

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");
    }
}
```

## 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

---

```
int i;
pid_t 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?

Recall: a process consists of one or more threads executing in an address space

- Here, each process has a single thread
- These threads execute concurrently

## Running Another Program

---

- With threads, we could call **pthread\_create** to create a new thread executing a separate function
- With processes, the equivalent would be spawning a new process executing a different program
- How can we do this?



## Process Management API

---

- **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

## fork3.c

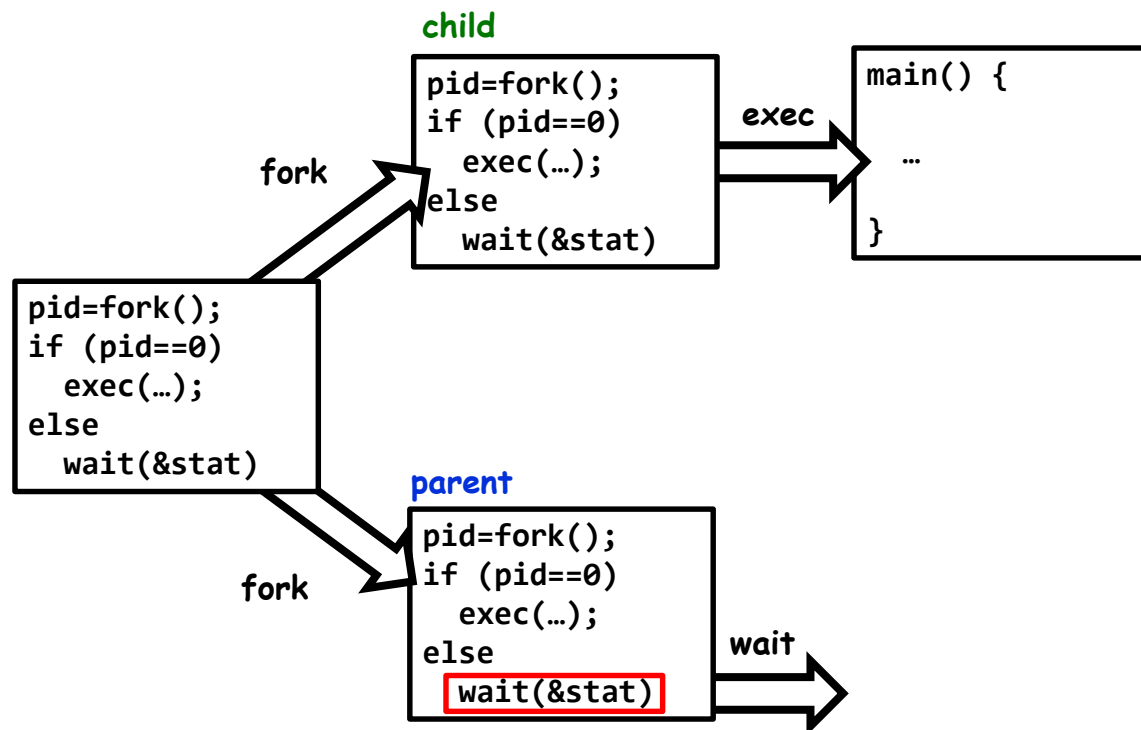
---

```
...
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);
}
...
```

## Process Management



## Process Management API

---

- **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

## 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);
    exit(42);
}
...
```

## Process Management API

---

- **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

## 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) {}
}
```

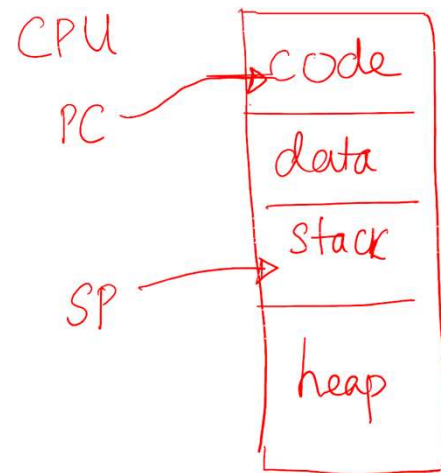
Q: What would happen if the process receives a SIGINT signal, but does not register a signal handler?

A: The process dies!

For each signal, there is a default handler defined by the system

## Process Execution

- **OS allocates memory and creates memory image**
  - Code and data (from exe)
  - Stack and heap
- **Points CPU program counter to current instruction**
  - Other registers may store operands, return values etc.
- **After setup, OS is out of the way and process executes directly on CPU**



5  
6



## How to efficiently virtualize the CPU with control?

---

- The OS needs to share the physical CPU by **time sharing**.
- Issue
  - **Performance:** How can we implement virtualization without adding excessive overhead to the system?
  - **Control:** How can we run processes efficiently while retaining control over the CPU?

## Direct Execution

- Just run the program directly on the CPU.

OS	Program
<ul style="list-style-type: none"><li>1. Create entry for process list</li><li>2. Allocate memory for program</li><li>3. Load program into memory</li><li>4. Set up stack with <code>argc / argv</code></li><li>5. Clear registers</li><li>6. Execute call <code>main()</code></li></ul>	<ul style="list-style-type: none"><li>7. Run <code>main()</code></li><li>8. Execute <code>return from main()</code></li></ul>
<ul style="list-style-type: none"><li>9. Free memory of process</li><li>10. Remove from process list</li></ul>	

**Without *limits* on running programs,  
the OS wouldn't be in control of anything and  
thus would be "just a library"**

## Problem 1: Restricted Operation

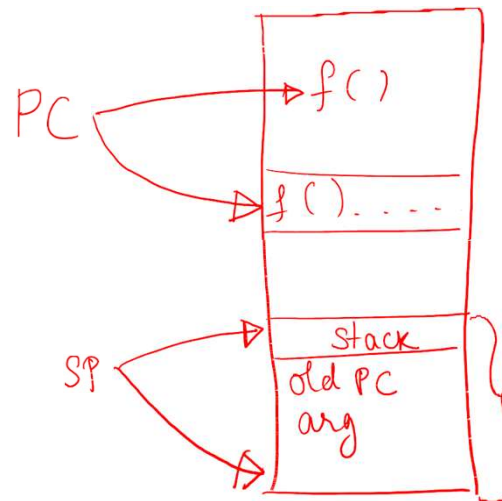
---

- What if a process wishes to perform some kind of restricted operation such as ...
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory
- **Solution:** Using protected control transfer
  - **User mode:** Applications do not have full access to hardware resources.
  - **Kernel mode:** The OS has access to the full resources of the machine

---

## A simple function call

- A function call translates to a jump instruction
- A new stack frame pushed to stack and stack pointer (SP) updated
- Old value of PC (return value) pushed to stack and PC updated
- Stack frame contains return value, function arguments etc.



4

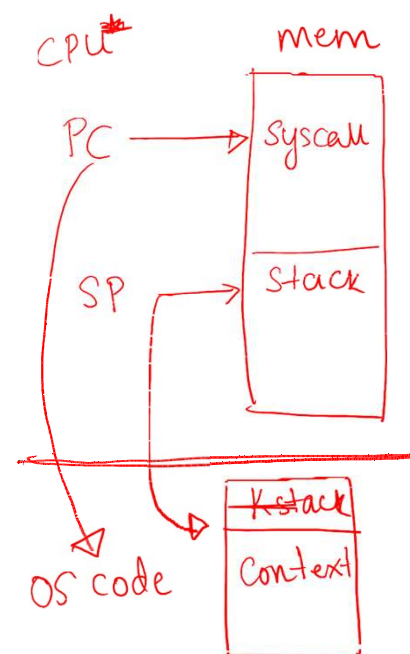
---

## How is a system call different?

- CPU hardware has multiple privilege levels
  - One to run user code: user mode
  - One to run OS code like system calls: kernel mode
  - Some instructions execute only in kernel mode
- Kernel does not trust user stack
  - Uses a separate kernel stack when in kernel mode
- Kernel does not trust user provided addresses to jump to
  - Kernel sets up Interrupt Descriptor Table (IDT) at boot time
  - IDT has addresses of kernel functions to run for system calls and other events

## Mechanism of system call: trap instruction

- When system call must be made, a special trap instruction is run (usually hidden from user by libc)
- Trap instruction execution
  - Move CPU to higher privilege level
  - Switch to kernel stack
  - Save context (old PC, registers) on kernel stack
  - Look up address in IDT and jump to trap handler function in OS code



---

## More on the trap instruction

- Trap instruction is executed on hardware in following cases:
  - System call (program needs OS service)
  - Program fault (program does something illegal, e.g., access memory it doesn't have access to)
  - Interrupt (external device needs attention of OS, e.g., a network packet has arrived on network card)
- Across all cases, the mechanism is: save context on kernel stack and switch to OS address in IDT
- IDT has many entries: which to use?
  - System calls/interrupts store a number in a CPU register before calling trap, to identify which IDT entry to use



## Return from trap

---

- When OS is done handling syscall or interrupt, it calls a special instruction return-from-trap
  - Restore context of CPU registers from kernel stack
  - Change CPU privilege from kernel mode to user mode
  - Restore PC and jump to user code after trap
- User process unaware that it was suspended, resumes execution as always
- Must you always return to the same user process from kernel mode? No
- Before returning to user mode, OS checks if it must switch to another process





## System Call

---

- Allow the kernel to **carefully expose** certain key pieces of functionality to user program, such as ...
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

## System Call (Cont.)

---

- **Trap** instruction
  - Jump into the kernel
  - Raise the privilege level to kernel mode
- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode

## Limited Direction Execution Protocol

OS @ boot  
(kernel mode)

Hardware

initialize trap table

remember address of ...  
syscall handler

OS @ run  
(kernel mode)

Hardware

Program  
(user mode)

Create entry for process list  
Allocate memory for  
program  
Load program into memory  
Setup user stack with argv  
Fill kernel stack with reg/PC  
return-from -trap

restore regs from kernel  
stack  
move to user mode  
jump to main

Run main()  
...  
Call system  
trap into OS

67

## Limited Direction Execution Protocol (Cont.)

OS @ run  
(kernel mode)

Hardware

Program  
(user mode)

*(Cont.)*

Handle trap  
Do work of syscall  
return-from-trap

save regs to kernel stack  
move to kernel mode  
jump to trap handler

restore regs from kernel  
stack  
move to user mode  
jump to PC after trap

...  
return from main  
trap (via `exit()`)

Free memory of process  
Remove from process  
list

68

## Problem 2: Switching Between Processes

---

- How can the OS **regain control** of the CPU so that it can switch between *processes*?
  - A cooperative Approach: **Wait** for system calls
  - A Non-Cooperative Approach: The OS takes control

## A cooperative Approach: Wait for system calls

---

- Processes **periodically give up the CPU** by making **system calls** such as `yield`.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - » Divide by zero
    - » Try to access memory that it shouldn't be able to access
  - Ex) Early versions of the Macintosh OS, The old Xerox Alto system

**A process gets stuck in an infinite loop.  
→ Reboot the machine**

## A Non-Cooperative Approach: OS Takes Control

---

- A timer interrupt
  - During the boot sequence, the OS start the timer.
  - The timer raise an interrupt every so many milliseconds.
  - When the interrupt is raised :
    - » The currently running process is halted.
    - » Save enough of the state of the program
    - » A pre-configured interrupt handler in the OS runs.

**A timer interrupt gives OS the ability to run again on a CPU.**

## Saving and Restoring Context

---

- Scheduler makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS executes context switch.



## Context Switch

---

- A low-level piece of assembly code
  - **Save a few register values** for the current process onto its kernel stack
    - » General purpose registers
    - » PC
    - » kernel stack pointer
  - **Restore a few** for the soon-to-be-executing process from its kernel stack
  - **Switch to the kernel stack** for the soon-to-be-executing process

## Limited Direction Execution Protocol (Timer interrupt)

**OS @ boot  
(kernel mode)**

**Hardware**

**initialize trap table**

**remember address of ...  
syscall handler  
timer handler**

**start interrupt timer**

**start timer  
interrupt CPU in X ms**

**OS @ run  
(kernel mode)**

**Hardware**

**Program  
(user mode)**

**Process A**

...

**timer interrupt  
save regs(A) to k-  
stack(A)  
move to kernel mode  
jump to trap handler**

74

### Limited Direction Execution Protocol (Timer interrupt)

OS @ run  
(kernel mode)

Hardware

Program  
(user mode)

*(Cont.)*

Handle the trap  
Call switch() routine  
  save regs(A) to proc-struct(A)  
  restore regs(B) from proc-struct(B)  
  switch to k-stack(B)  
return-from-trap (into B)

restore regs(B) from k-stack(B)  
move to user mode  
jump to B's PC

Process B  
...

75

## The xv6 Context Switch Code

```
1 # void swtch(struct context **old, struct context *new);
2 #
3 # Save current register context in old
4 # and then load register context from new.
5 .globl swtch
6 swtch:
7     # Save old registers
8     movl 4(%esp), %eax           # put old ptr into eax
9     popl 0(%eax)                # save the old IP
10    movl %esp, 4(%eax)          # and stack
11    movl %ebx, 8(%eax)          # and other registers
12    movl %ecx, 12(%eax)
13    movl %edx, 16(%eax)
14    movl %esi, 20(%eax)
15    movl %edi, 24(%eax)
16    movl %ebp, 28(%eax)
17
18    # Load new registers
19    movl 4(%esp), %eax           # put new ptr into eax
20    movl 28(%eax), %ebp         # restore other registers
21    movl 24(%eax), %edi
22    movl 20(%eax), %esi
23    movl 16(%eax), %edx
24    movl 12(%eax), %ecx
25    movl 8(%eax), %ebx
26    movl 4(%eax), %esp         # stack is switched here
27    pushl 0(%eax)              # return addr put in place
28    ret                        # finally return into new ctxt
```

76

## Worried About Concurrency?

---

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.

## 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?

## How does a Shell work?

---

- In a basic OS, the `init` process is created after initialization of hardware
- The `init` process spawns a shell like `bash`
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like `ls` are all executables that are simply exec'ed by the shell
- `prompt>ls`
- `a.txt b.txt c.txt`

## More funky things about the shell

---

- Shell can manipulate the child in strange ways
- Suppose you want to redirect output from a command to a file
- `prompt>ls > foo.txt`
- Shell spawns a child, rewires its standard output to a file, then calls `exec` on the child



## Shell Assignment

---

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
- You will build your own shell in Assignment 3...
  - ... using **fork** and **exec** system calls to create new processes...
  - ... and the File I/O system calls we'll see next time to link them together

## Process vs. Thread APIs

---

- Why have **fork()** and **exec()** system calls for processes, but just a **pthread\_create()** function for threads?
  - Convenient to **fork** without **exec**: put code for parent and child in one executable instead of multiple
  - It will allow us to programmatically control child process' state
    - » By executing code before calling **exec()** in the child
  - We'll see this in the case of File I/O next time
- Windows uses **CreateProcess()** instead of **fork()**
  - Also works, but a more complicated interface

## Threads vs. Processes

---

- If we have two tasks to run concurrently, do we run them in separate threads, or do we run them in separate processes?
- Depends on how much isolation we want
  - Threads are lighter weight [why?]
  - Processes are more strongly isolated

## Conclusion

---

- 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