

Processes II

CSE 351 Winter 2022

Instructor:

Sam Wolfson

Teaching Assistants:

Angela Xu

Anirudh Kumar

Catherine Guevara

Dara Stotland

Harrison Bay

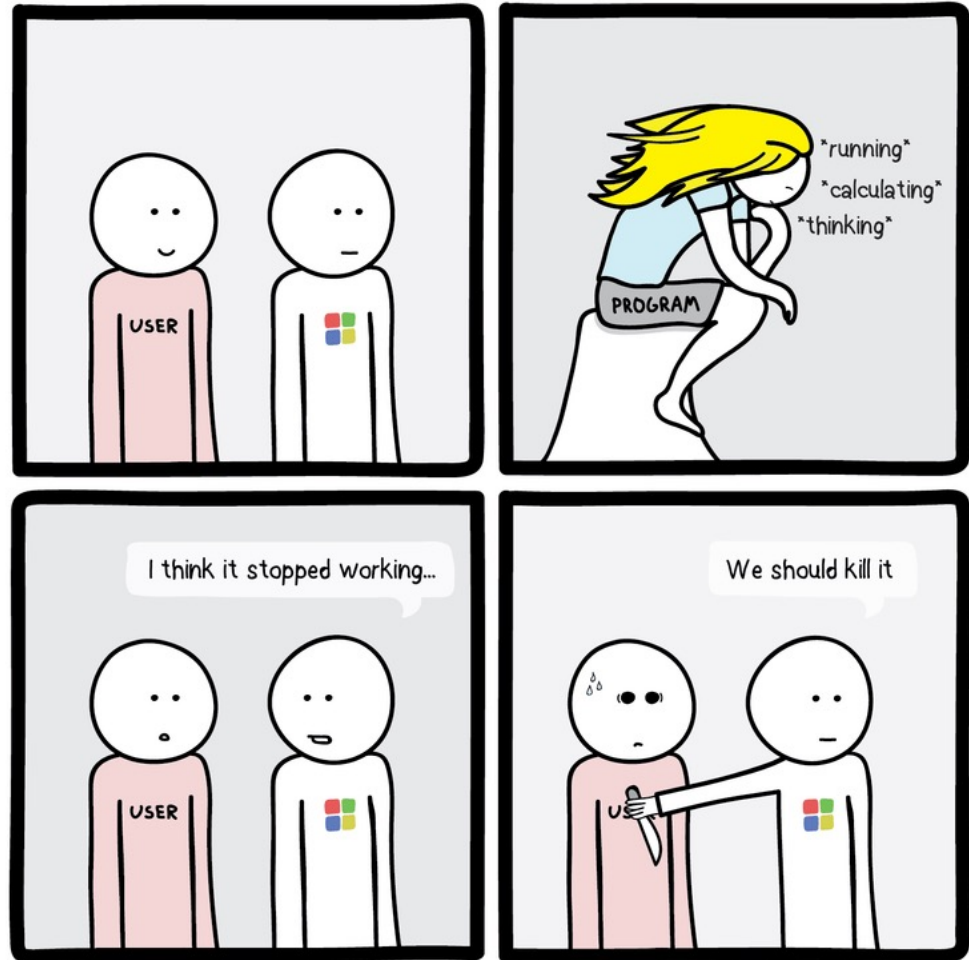
Ian Hsiao

Kevin Wang

Mara Kirdani-Ryan

Nick Durand

Sanjana Sridhar



PRETENDS TO BE DRAWING | PTBD.JWELS.BERLIN

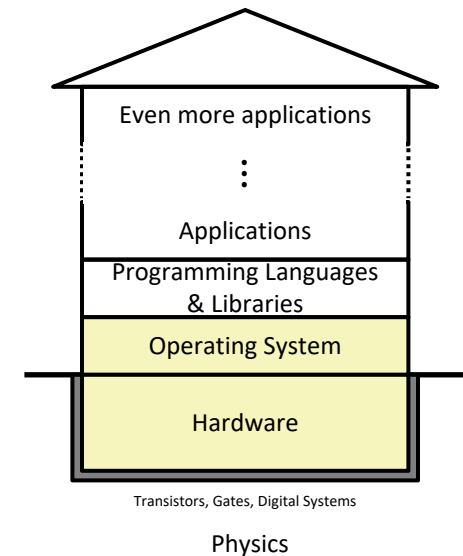
Relevant Course Information

- ❖ Lab 4 due next Monday
 - hw18 (lab 4 prep) due tonight
- ❖ Lab 3 grading in progress
- ❖ Midterm grading in progress

The Hardware/Software Interface

❖ Topic Group 3: **Scale & Coherence**

- Caches, **Processes**, Virtual Memory, Memory Allocation



- ❖ How do we maintain logical consistency in the face of more data and more processes?
 - How do we support control flow both within many processes and things external to the computer?
 - How do we support data access, including dynamic requests, across multiple processes?

Reading Review

❖ Terminology:

- Exceptional control flow, event handlers
- Operating system kernel
- Exceptions: interrupts, traps, faults, aborts
- Processes: concurrency, context switching, fork-exec model, process ID
- `exec*()`, `exit()`, `wait()`, `waitpid()`
- `init/systemd`, reaping, zombie processes

❖ Questions from the Reading?

Leading Up to Processes

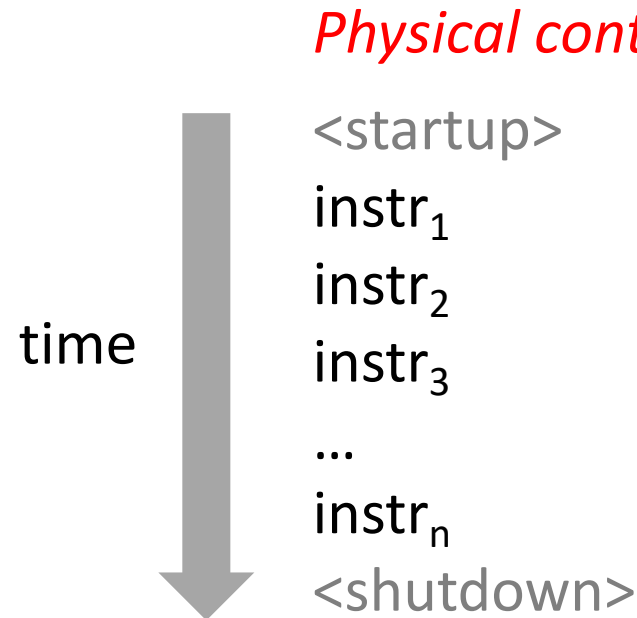
- ❖ System Control Flow
 - **Control flow**
 - **Exceptional control flow**
 - Asynchronous exceptions (interrupts)
 - Synchronous exceptions (traps & faults)

Control Flow

- ❖ **So far:** we've seen how the flow of control changes as a *single program* executes
- ❖ **Reality:** multiple programs running *concurrently*
 - How does control flow across the many components of the system?
 - In particular: More programs running than CPUs
- ❖ **Exceptional control flow** is basic mechanism used for:
 - Transferring control between *processes* and OS
 - Handling *I/O* and *virtual memory* within the OS
 - Implementing multi-process apps like shells and web servers
 - Implementing concurrency

Control Flow

- ❖ Processors do only one thing:
 - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
 - This sequence is the CPU's *control flow* (or *flow of control*)



Altering the Control Flow

- ❖ Up to now, two ways to change control flow:
 - Jumps (conditional and unconditional)
 - Call and return
 - Both react to changes in *program state*
- ❖ Processor also needs to react to changes in *system state*
 - Unix/Linux user hits “Ctrl-C” at the keyboard
 - User clicks on a different application’s window on the screen
 - Data arrives from a disk or a network adapter
 - Instruction divides by zero
 - System timer expires
- ❖ Can jumps and procedure calls achieve this?
 - No – the system needs mechanisms for *“exceptional”* control flow!

Exceptional Control Flow

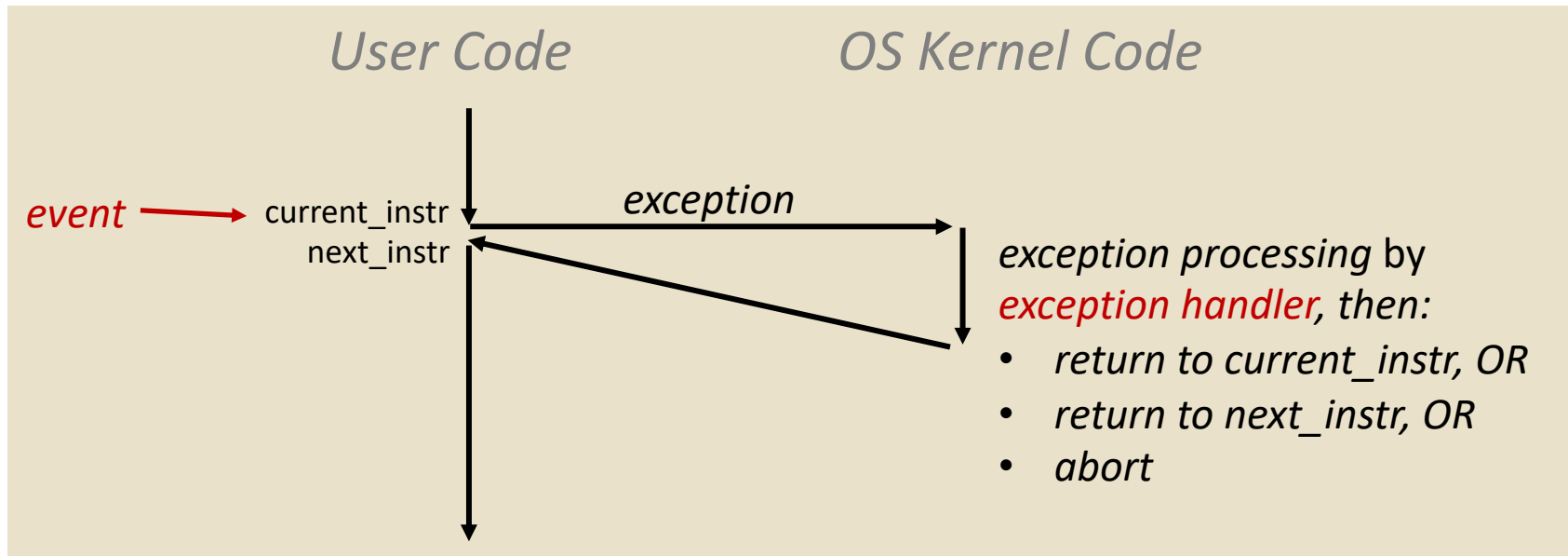
- ❖ Exists at all levels of a computer system
- ❖ Low level mechanisms
 - **Exceptions**
 - Change in processor's control flow in response to a system event (*i.e.*, change in system state, user-generated interrupt)
 - Implemented using a combination of hardware and OS software
- ❖ Higher level mechanisms
 - **Process context switch**
 - Implemented by OS software and hardware timer
 - **Signals**
 - Implemented by OS software
 - We won't cover these in detail – see CSE 451 and EE/CSE 474

These are
unrelated to Java's
exceptions! (woof!)



Exceptions (Review)

- ❖ An *exception* is transfer of control to the operating system (OS) kernel in response to some *event* (i.e., change in processor state)
 - Kernel is the memory-resident part of the OS
 - Examples: division by 0, page fault, I/O request completes, Ctrl-C

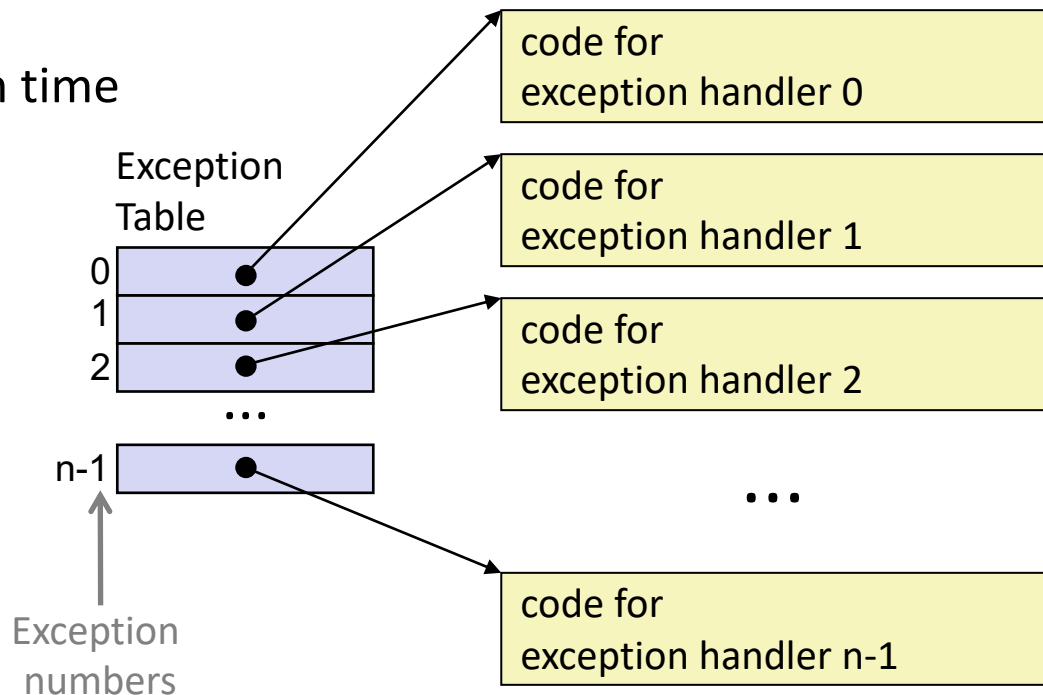


- ❖ *How does the system know where to jump to in the OS?*

Exception Table

This is extra
(non-testable)
material

- ❖ A jump table for exceptions (also called *Interrupt Vector Table*)
 - Each type of event has a unique exception number k
 - k = index into exception table (a.k.a. interrupt vector)
 - Handler k is called each time exception k occurs



Exception Table (Excerpt)

This is extra
(non-testable)
material

<i>Exception Number</i>	<i>Description</i>	<i>Exception Class</i>
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-255	OS-defined	Interrupt or trap

Leading Up to Processes

- ❖ System Control Flow
 - Control flow
 - Exceptional control flow
 - **Asynchronous exceptions (interrupts)**
 - **Synchronous exceptions (traps & faults)**

Asynchronous Exceptions (Review)

- ❖ **Interrupts**: caused by events external to the processor
 - Indicated by setting the processor's interrupt pin(s) (wire into CPU)
 - After interrupt handler runs, the handler returns to “next” instruction
- ❖ Examples:
 - I/O interrupts
 - Hitting Ctrl-C on the keyboard
 - Clicking a mouse button or tapping a touchscreen
 - Arrival of a packet from a network
 - Arrival of data from a disk
 - Timer interrupt
 - Every few milliseconds, an external timer chip triggers an interrupt
 - Used by the OS kernel to take back control from user programs

Synchronous Exceptions (Review)



- ❖ Caused by events that occur as a result of executing an instruction:
 - **Traps**
 - **Intentional**: transfer control to OS to perform some function
 - Examples: *system calls*, breakpoint traps, special instructions
 - Returns control to “next” instruction
 - **Faults**
 - **Unintentional** but possibly recoverable
 - Examples: *page faults*, segment protection faults, integer divide-by-zero exceptions
 - Either re-executes faulting (“current”) instruction or aborts
 - **Aborts**
 - **Unintentional** and unrecoverable
 - Examples: parity error, machine check (hardware failure detected)
 - Aborts current program

System Calls

- ❖ Each system call has a unique ID number
- ❖ Examples for Linux on x86-64:

<i>Number</i>	<i>Name</i>	<i>Description</i>
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

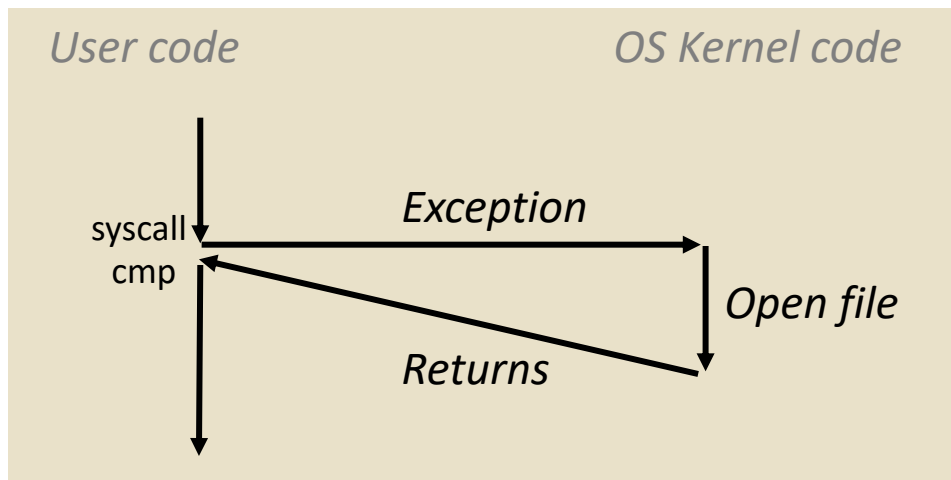
These are
not the same
as exception
numbers!



Traps Example: Opening File

- ❖ User calls `open(filename, options)`
- ❖ Calls `__open` function, which invokes system call instruction `syscall`

```
000000000000e5d70 <__open>:  
...  
e5d79:  b8 02 00 00 00      mov  $0x2,%eax    # open is syscall 2  
e5d7e:  0f 05               syscall           # return value in %rax  
e5d80:  48 3d 01 f0 ff ff    cmp  $0xffffffffffffffff001,%rax  
...  
e5dfa:  c3                 retq
```



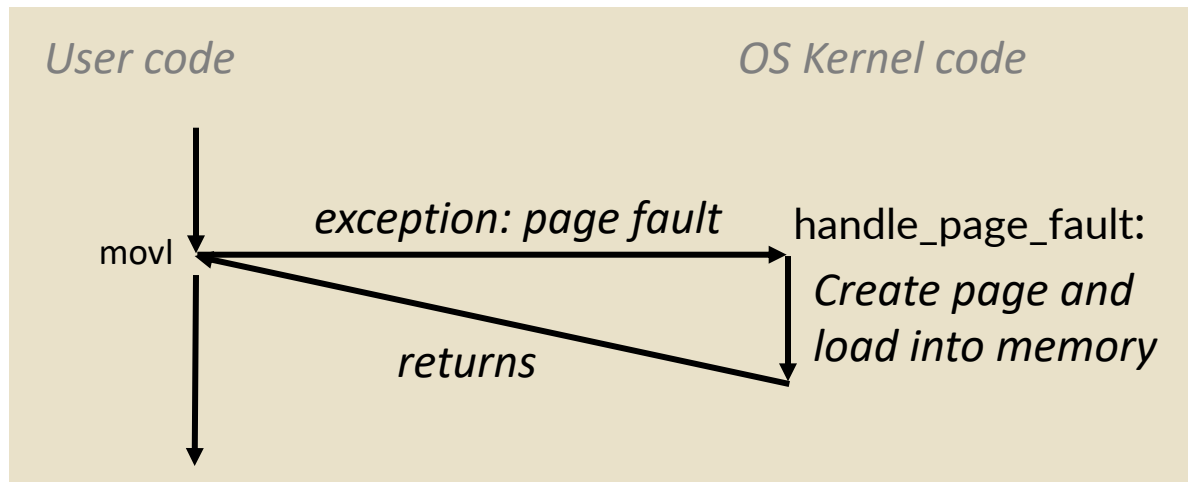
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

Fault Example: Page Fault

- ❖ User writes to memory location
- ❖ That portion (page) of user's memory is currently on disk

```
int a[1000];  
int main () {  
    a[500] = 13;  
}
```

```
80483b7:      c7 05 10 9d 04 08 0d  movl    $0xd,0x8049d10
```

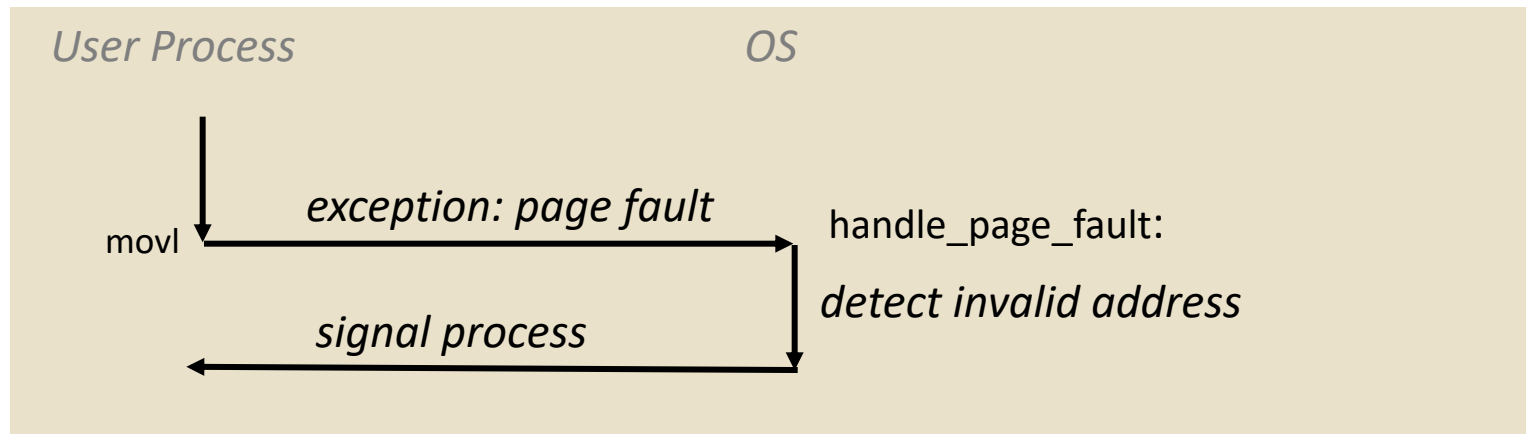


- ❖ Page fault handler must load page into physical memory
- ❖ Returns to faulting instruction: `mov` is executed again!
 - Successful on second try

Fault Example: Invalid Memory Reference

```
int a[1000];  
int main() {  
    a[5000] = 13;  
}
```

80483b7: c7 05 60 e3 04 08 0d **movl** \$0xd,0x804e360



- ❖ Page fault handler detects invalid address
- ❖ Sends `SIGSEGV` signal to user process
- ❖ User process exits with “segmentation fault”

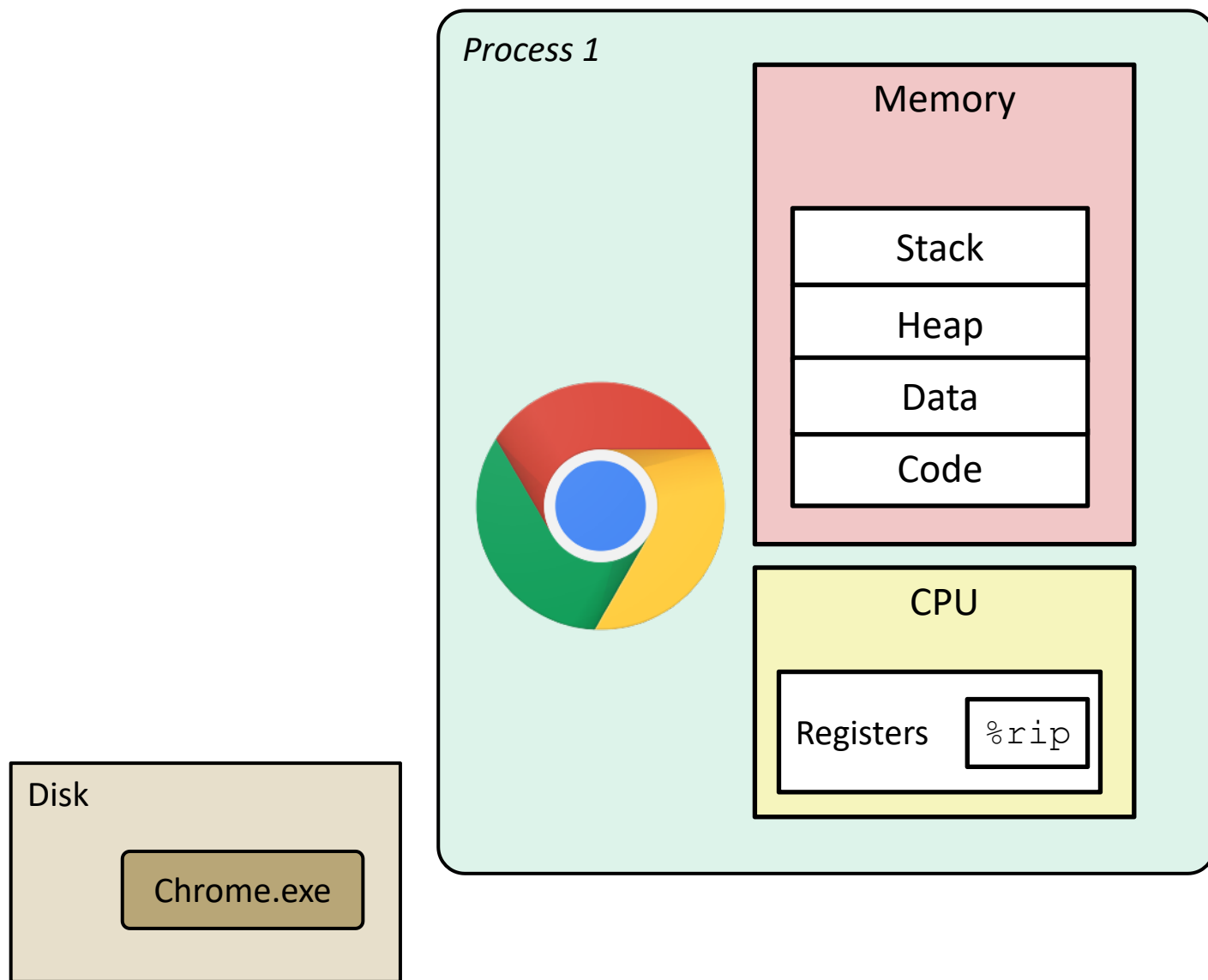


Processes

- ❖ **Processes and context switching**
- ❖ Creating new processes
 - `fork()`, `exec*()`, and `wait()`
- ❖ Zombies

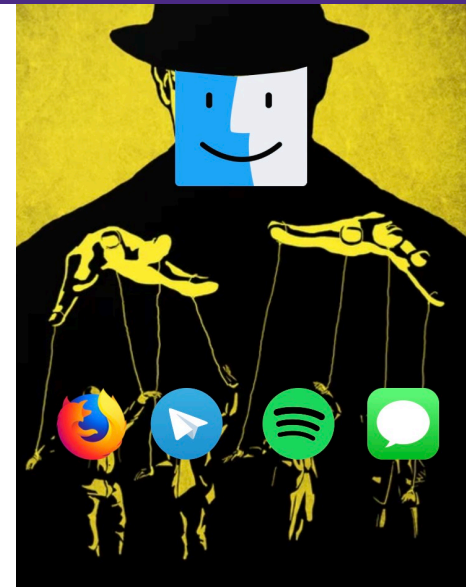
What is a process? (Review)

It's an *illusion*!



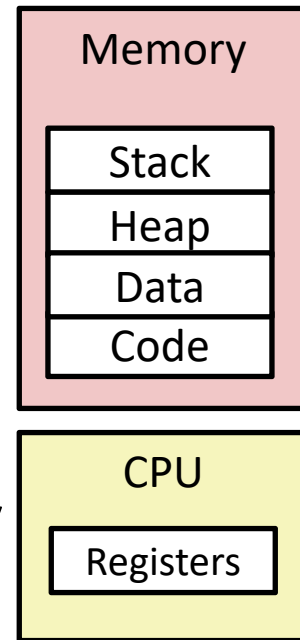
What is a process? (Review)

- ❖ Another *abstraction* in our computer
 - Provided by the OS
 - OS uses a data structure to represent each process (contains process ID (PID), etc.)
 - Maintains the *interface* between the program and the underlying hardware (CPU + memory)
- ❖ What do *processes* have to do with *exceptional control flow*?
 - Exceptional control flow is the *mechanism* the OS uses to enable **multiple processes** to run on the same system
- ❖ What is the difference between:
 - A processor? A program? A process?



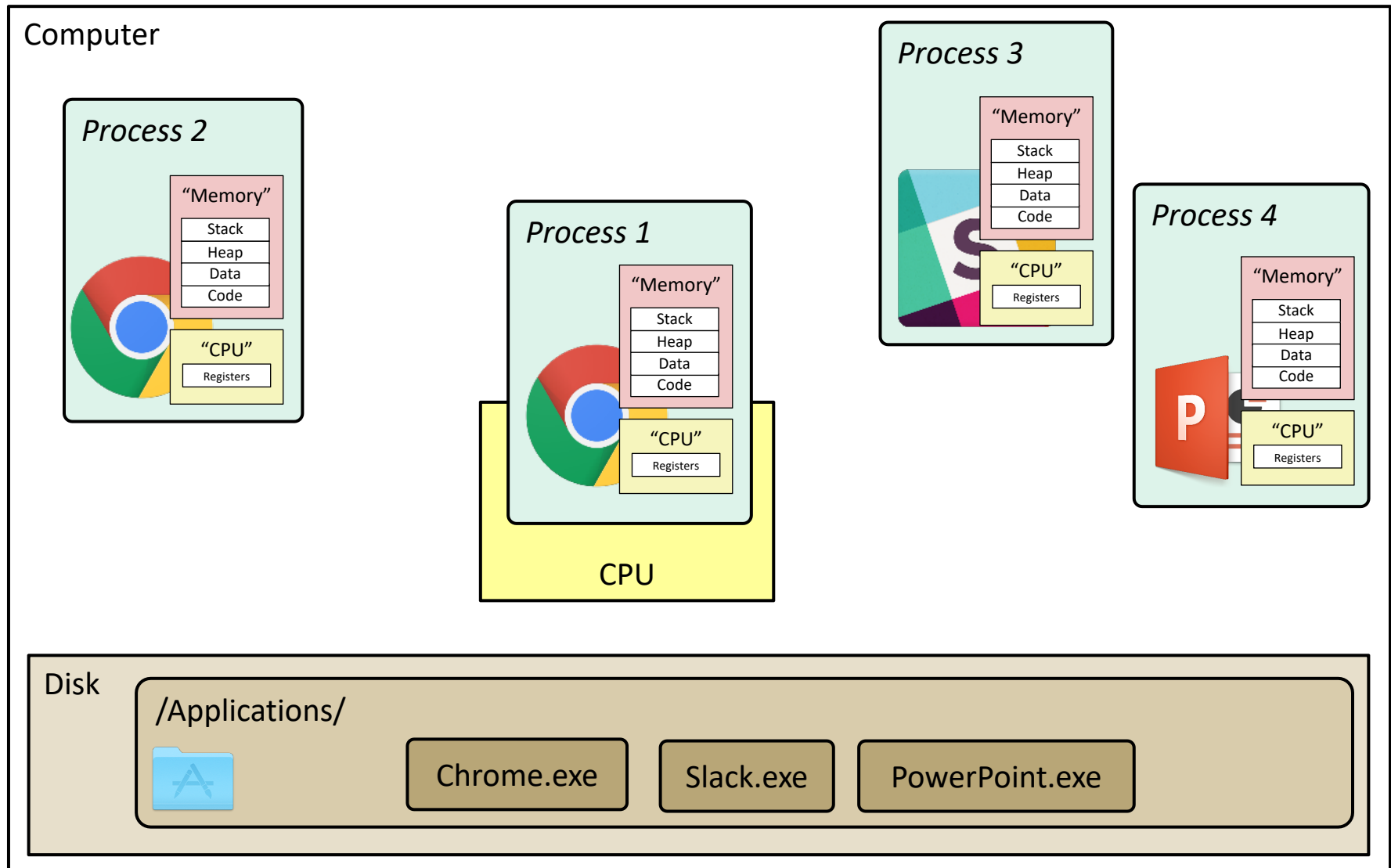
Processes (Review)

- ❖ A **process** is an **instance of a running program**
 - One of the most profound ideas in computer science
- ❖ Process provides each program with two key abstractions:
 - *Logical control flow*
 - Each program *seems to* have exclusive use of the CPU
 - Provided by kernel mechanism called **context switching**
 - *Private address space*
 - Each program *seems to* have exclusive use of main memory
 - Provided by kernel mechanism called **virtual memory**



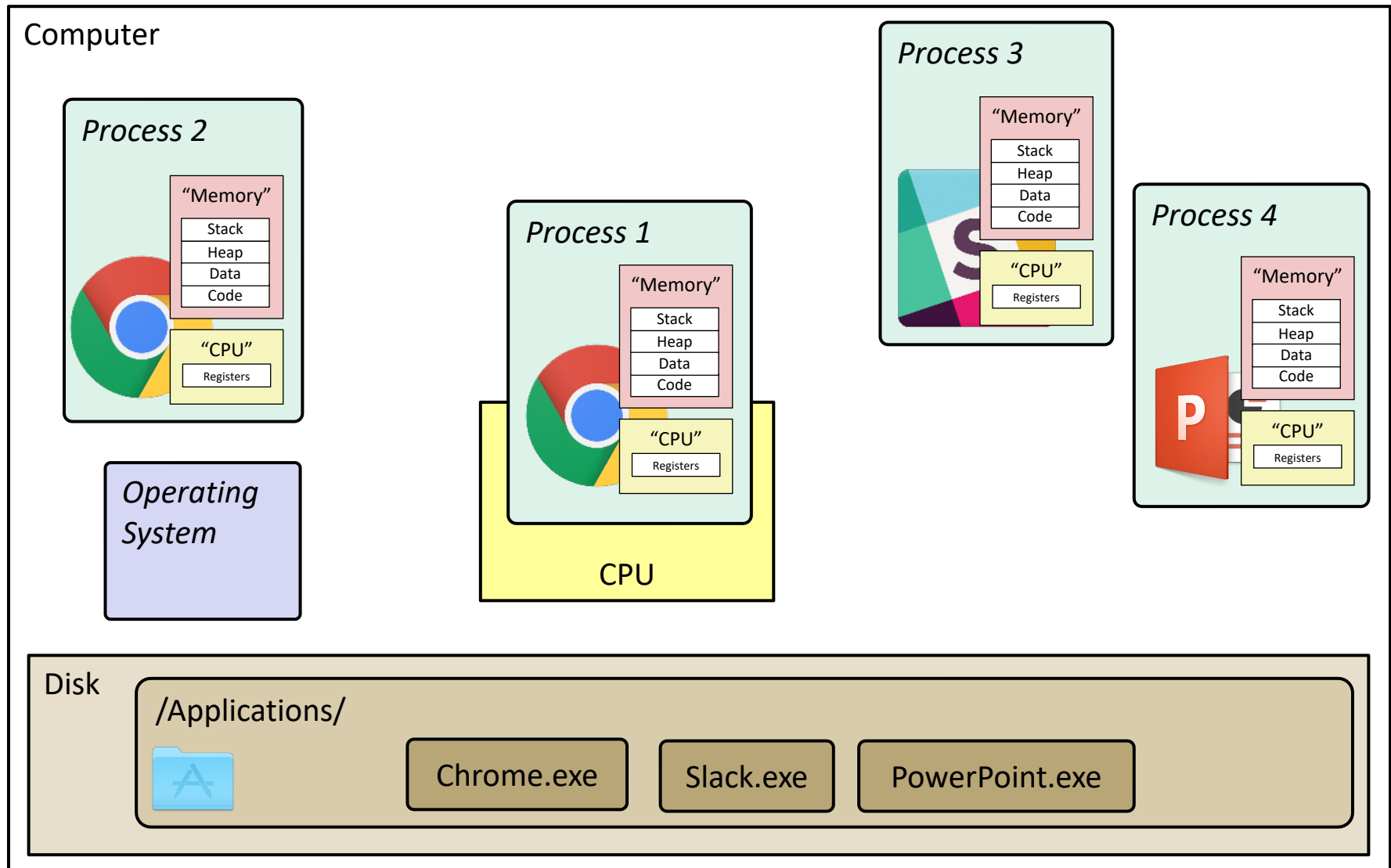
What is a process?

It's an *illusion*!

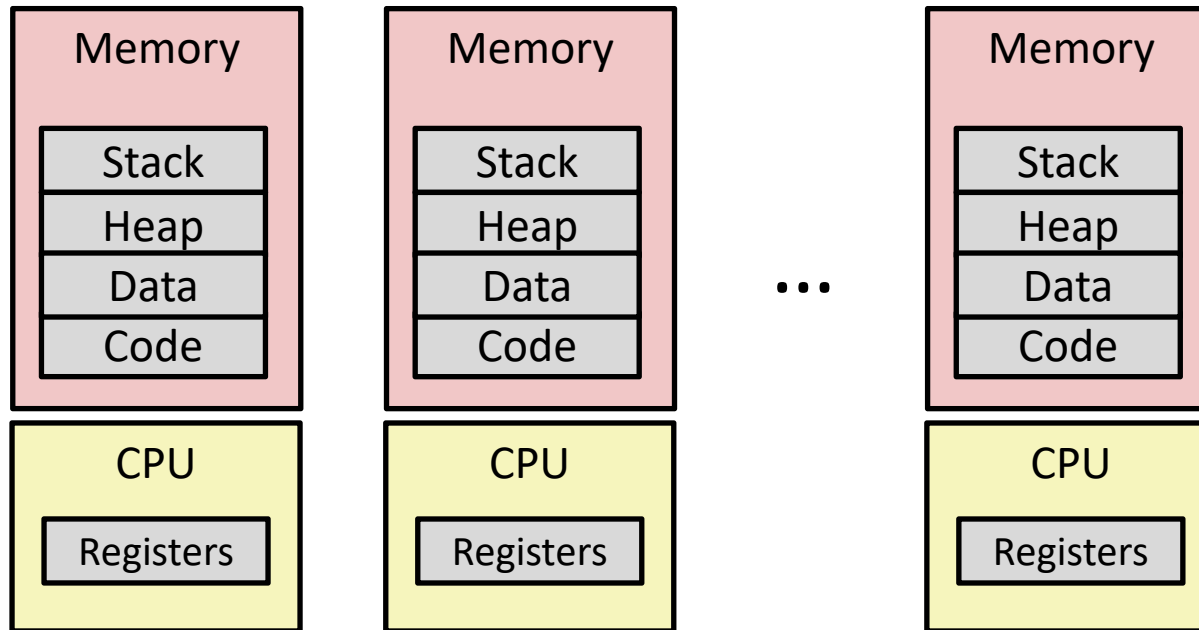


What is a process?

It's an *illusion*!

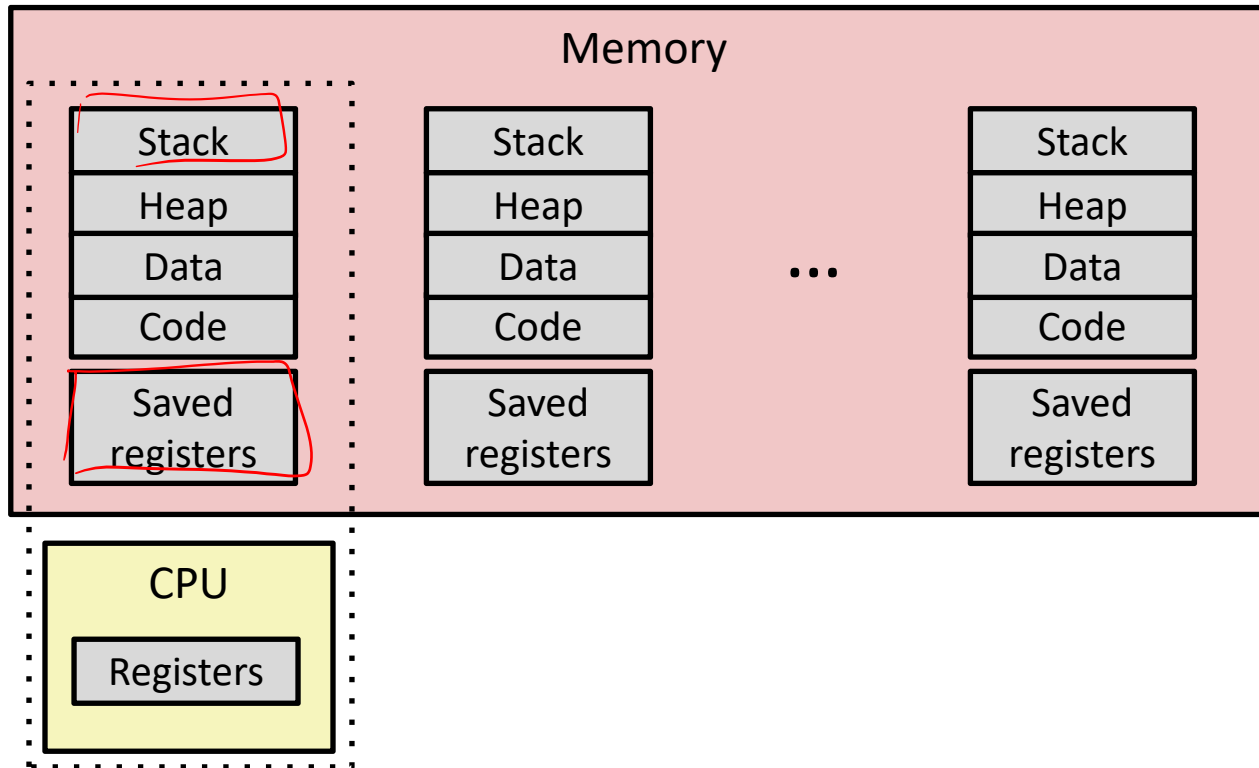


Multiprocessing: The Illusion



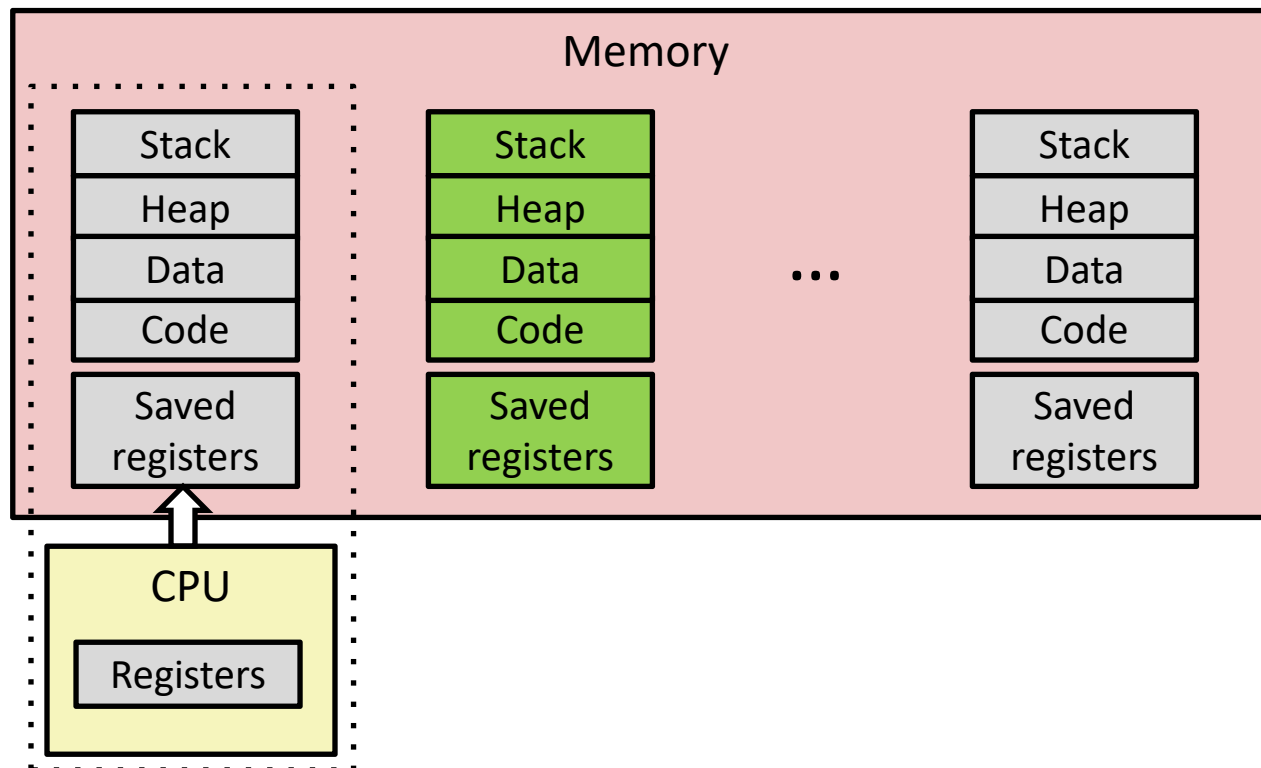
- ❖ Computer runs many processes simultaneously
 - Applications for one or more users
 - Web browsers, email clients, editors, ...
 - Background tasks
 - Monitoring network & I/O devices

Multiprocessing: The Reality



- ❖ Single processor executes multiple processes *concurrently*
 - Process executions interleaved, CPU runs *one at a time*
 - Address spaces managed by virtual memory system (later in course)
 - *Execution context* (register values, stack, ...) for other processes saved in memory

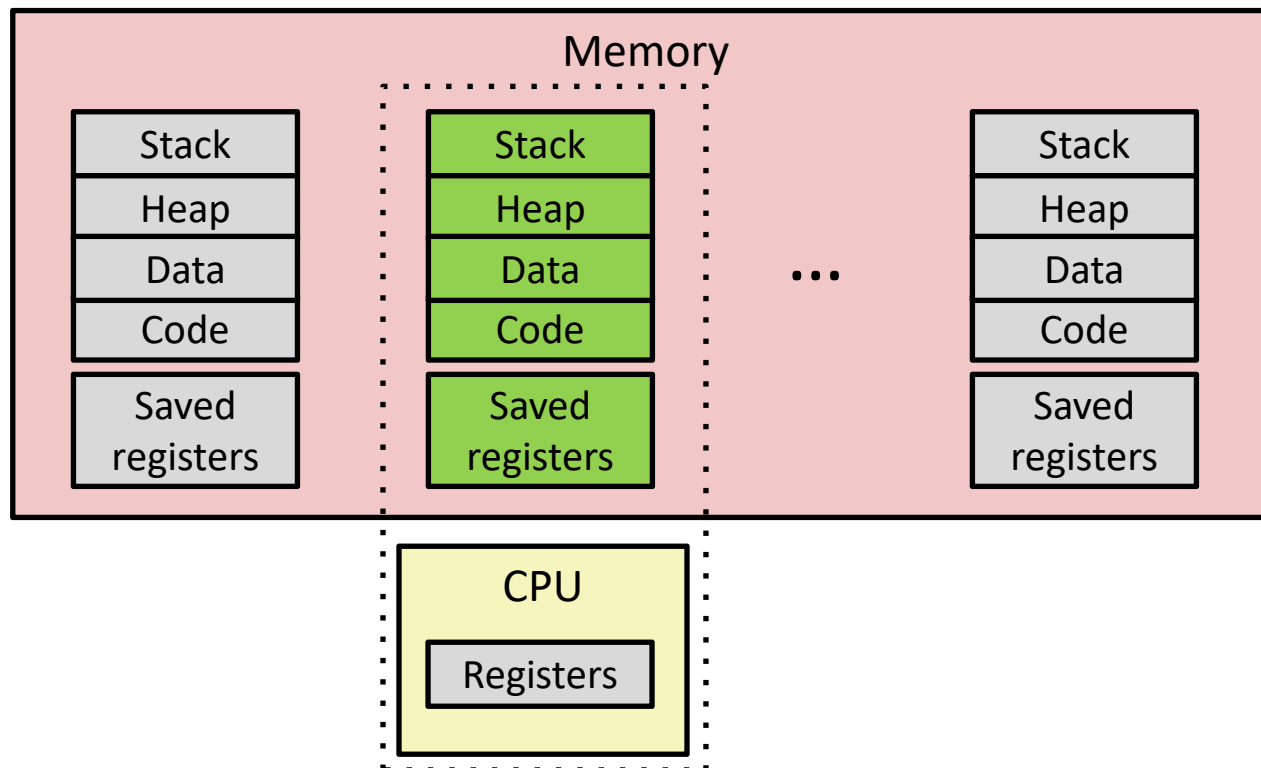
Multiprocessing (Review)



❖ Context switch

- 1) Save current registers in memory

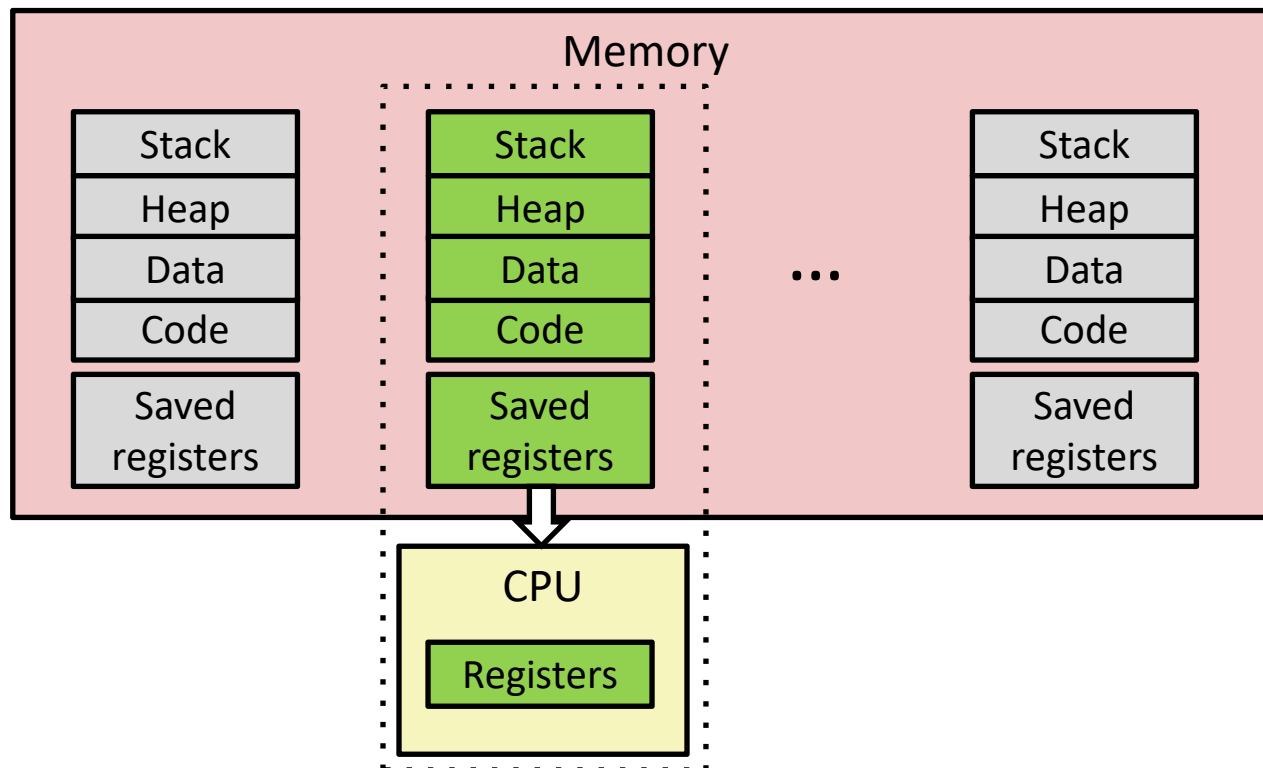
Multiprocessing (Review)



❖ Context switch

- 1) Save current registers in memory
- 2) **Schedule next process for execution**

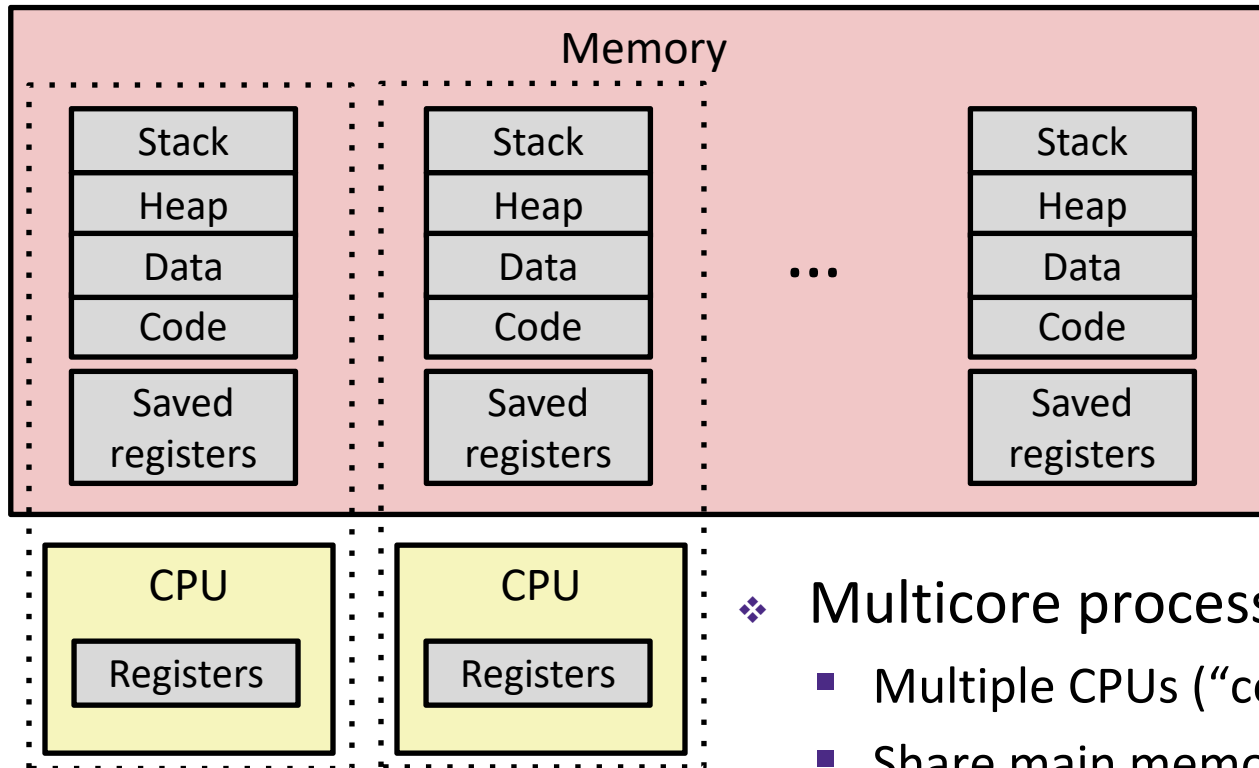
Multiprocessing (Review)



❖ Context switch

- 1) Save current registers in memory
- 2) Schedule next process for execution
- 3) **Load saved registers and switch address space**

Multiprocessing: The (Modern) Reality



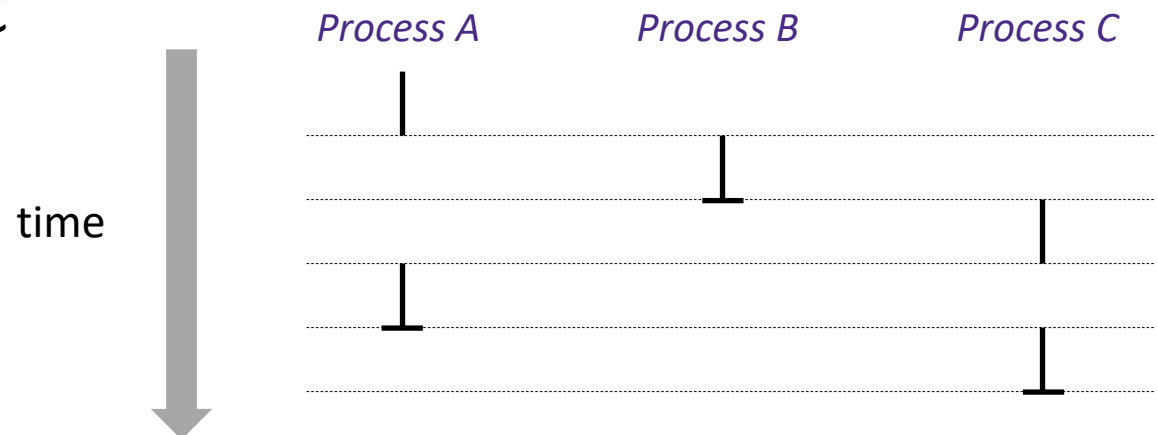
❖ Multicore processors

- Multiple CPUs (“cores”) on single chip
- Share main memory (and some of the caches)
- Each can execute a separate process
 - Kernel schedules processes to cores
 - ***Still* constantly swapping processes**

Concurrent Processes

Assume only one CPU

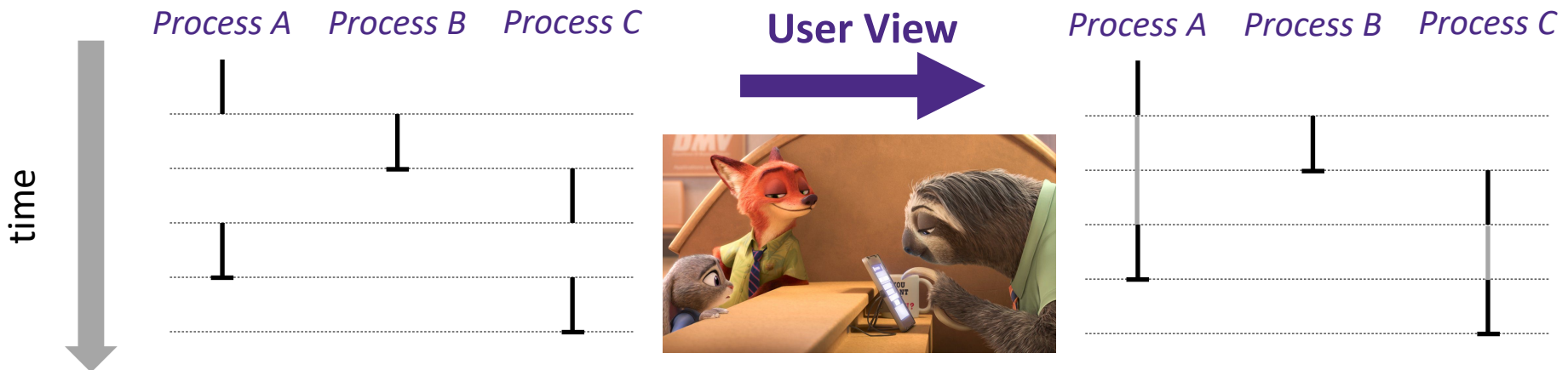
- ❖ Each process is a logical control flow
- ❖ Two processes *run concurrently* (are concurrent) if their instruction executions (flows) overlap in time
 - Otherwise, they are *sequential*
- ❖ Example: (running on single core)
 - Concurrent: A & B, A & C
 - Sequential: B & C



User's View of Concurrency

Assume only one CPU

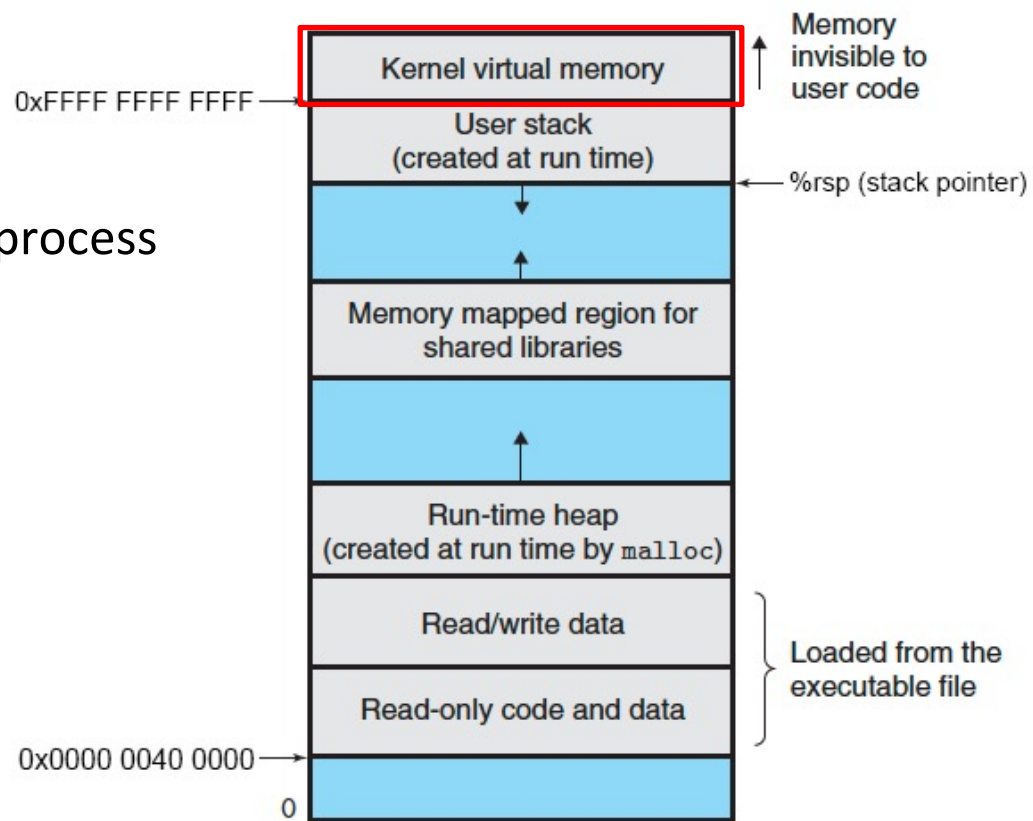
- ❖ Control flows for concurrent processes are physically disjoint in time
 - CPU only executes instructions for one process at a time
- ❖ However, the user can *think of* concurrent processes as executing at the same time, in *parallel*



Context Switching

Assume only one CPU

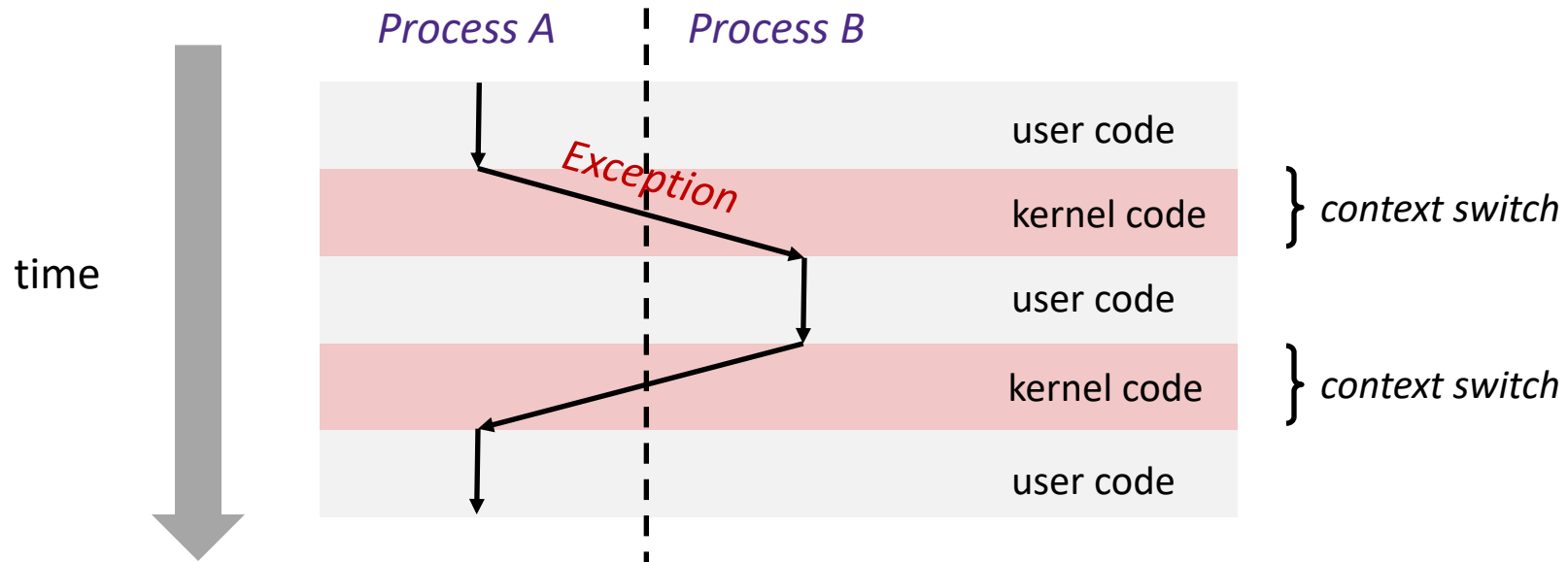
- ❖ Processes are managed by a *shared* chunk of OS code called the **kernel**
 - The kernel is not a separate process, but rather runs as part of a user process
- ❖ In x86-64 Linux:
 - Same address in each process refers to same shared memory location



Context Switching (Review)

Assume only one CPU

- ❖ Processes are managed by a *shared* chunk of OS code called the **kernel**
 - The kernel is not a separate process, but rather runs as part of a user process
- ❖ Context switch passes control flow from one process to another and is performed using kernel code



Processes & Context Switching Summary

❖ Exceptions

- Events that require non-standard control flow
- Generated asynchronously (interrupts) or synchronously (traps and faults)
- After an exception is handled, either:
 - Re-execute the current instruction
 - Resume execution with the next instruction
 - Abort the process that caused the exception

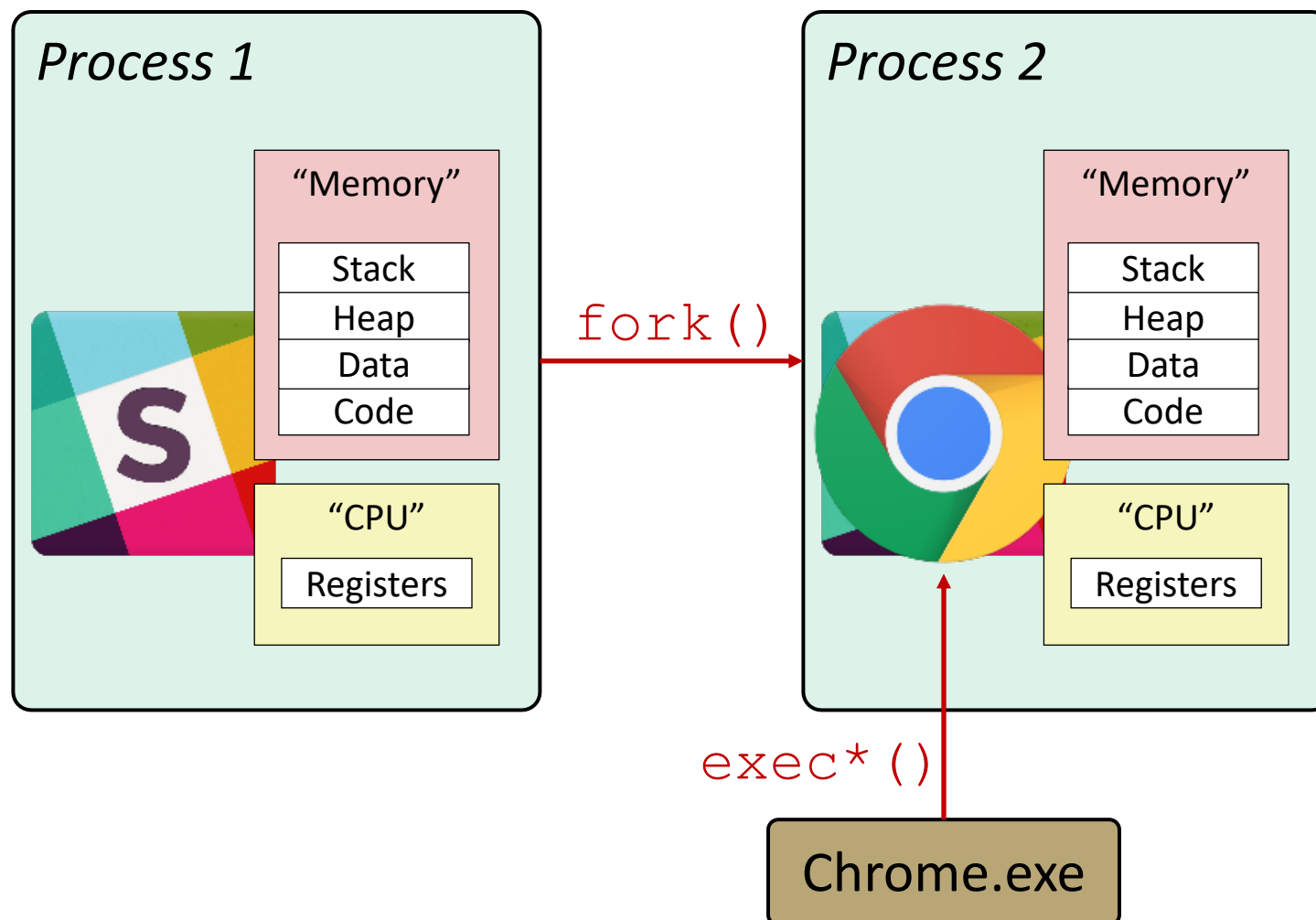
❖ Processes

- Only one of many active processes executes at a time on a CPU, but each appears to have total control of the processor
- OS periodically “context switches” between active processes

Processes

- ❖ Processes and context switching
- ❖ **Creating new processes**
 - `fork()` and `exec*()`
- ❖ Ending a process
 - `exit()`, `wait()`, `waitpid()`
 - Zombies

Creating New Processes & Programs



Creating New Processes & Programs

❖ fork-exec model (Linux):

- `fork()` creates a copy of the current process
- `exec*()` replaces the current process' code and address space with the code for a different program
 - Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
- `fork()` and `execve()` are *system calls*

❖ Other system calls for process management:

- `getpid()`
- `exit()`
- `wait()`, `waitpid()`

fork: Creating New Processes

❖ `pid_t fork(void)`

- Creates a new “**child**” process that is *identical* to the calling “**parent**” process, including all state (memory, registers, etc.)
- Returns 0 to the **child** process
- Returns child’s **process ID (PID)** to the **parent** process

❖ Child is *almost* identical to parent:

- Child gets an identical (but separate) copy of the parent’s virtual address space
- Child has a different PID than the parent

```
pid_t pid = fork();  
if (pid == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

- ❖ `fork` is unique (and often confusing) because it is called **once** but returns “**twice**”



Understanding `fork()`

Process X (parent; PID X)



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

Process Y (child; PID Y)



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

Understanding `fork()`

Process X (parent; PID X)



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

fork ret = Y



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

Process Y (child; PID Y)



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

fork ret = 0



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

Understanding `fork()`

Process X (parent; PID X)



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

fork ret = Y



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

hello from parent

Process Y (child; PID Y)



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

fork ret = 0



```
pid_t fork_ret = fork();  
if (fork_ret == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

hello from child

Which one appears first?

Fork Example

```
void fork1() {
    int x = 1;
    pid_t fork_ret = fork();
    if (fork_ret == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

- ❖ Both processes continue/start execution after `fork`
 - Child starts at instruction after the call to `fork` (storing into `pid`)
- ❖ Can't predict execution order of parent and child
- ❖ Both processes start with `x = 1`
 - Subsequent changes to `x` are independent
- ❖ Shared open files: `stdout` is the same in both parent and child

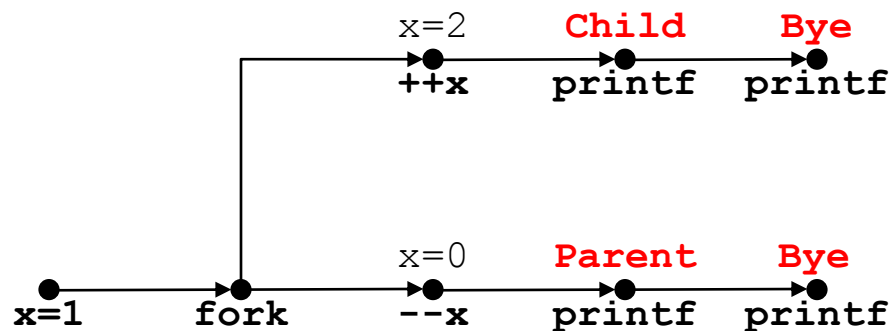
Modeling fork with Process Graphs

- ❖ A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program
 - Each vertex is the execution of a statement
 - $a \rightarrow b$ means a happens before b
 - Edges can be labeled with current value of variables
 - `printf` vertices can be labeled with output
 - Each graph begins with a vertex with no in-edges

- ❖ Any *topological sort* of the graph corresponds to a feasible total ordering
 - An ordering of nodes that contains every node, and only follows edges (lines between nodes) in the direction of the arrows

Fork Example: Possible Output

```
void fork1() {  
    int x = 1;  
    pid_t fork_ret = fork();  
    if (fork_ret == 0)  
        printf("Child has x = %d\n", ++x);  
    else  
        printf("Parent has x = %d\n", --x);  
    printf("Bye from process %d with x = %d\n", getpid(), x);  
}
```



Polling Question

❖ Are the following sequences of outputs possible?

- Vote at <https://PollEv.com/wolfson>

```
void nestedfork() {  
    printf("L0\n");  
    if (fork() == 0) {  
        printf("L1\n");  
        if (fork() == 0) {  
            printf("L2\n");  
        }  
    }  
    printf("Bye\n");  
}
```

Seq 1:

L0

L1

Bye

Bye

Bye

L2

Seq 2:

L0

Bye

L1

L2

Bye

Bye

A. No No

B. No Yes

C. Yes No

D. Yes Yes

E. We're lost...

Fork-Exec

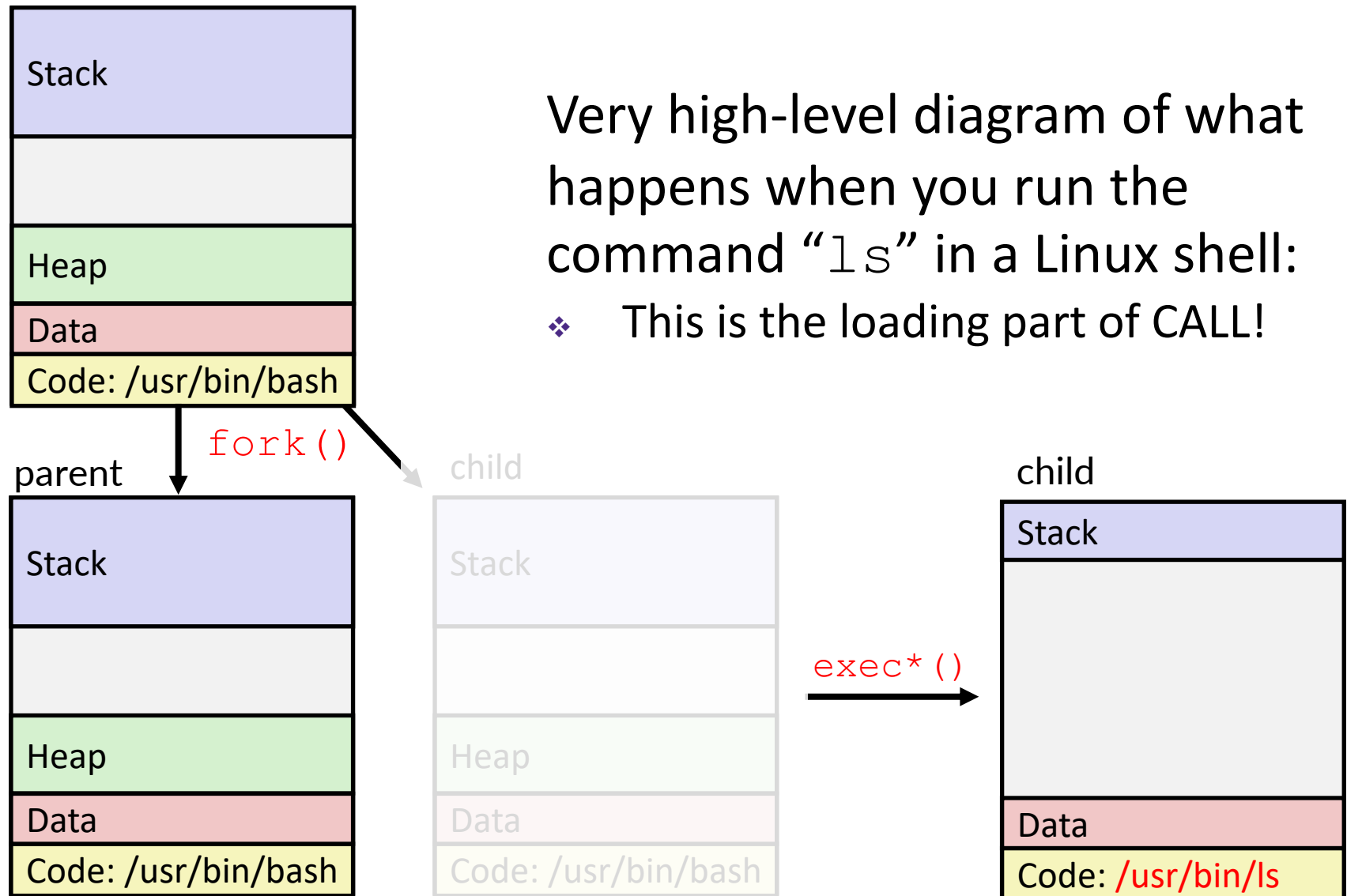
Note: the return values of `fork` and `exec*` should be checked for errors

❖ fork-exec model:

- `fork()` creates a copy of the current process
- `exec*()` replaces the current process' code and address space with the code for a different program
 - Whole family of `exec` calls – see **`exec(3)`** and **`execve(2)`**

```
// Example arguments: path="/usr/bin/ls",
//      argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char *path, char *argv[]) {
    pid_t fork_ret = fork();
    if (fork_ret != 0) {
        printf("Parent: created a child %d\n", fork_ret);
    } else {
        printf("Child: about to exec a new program\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```

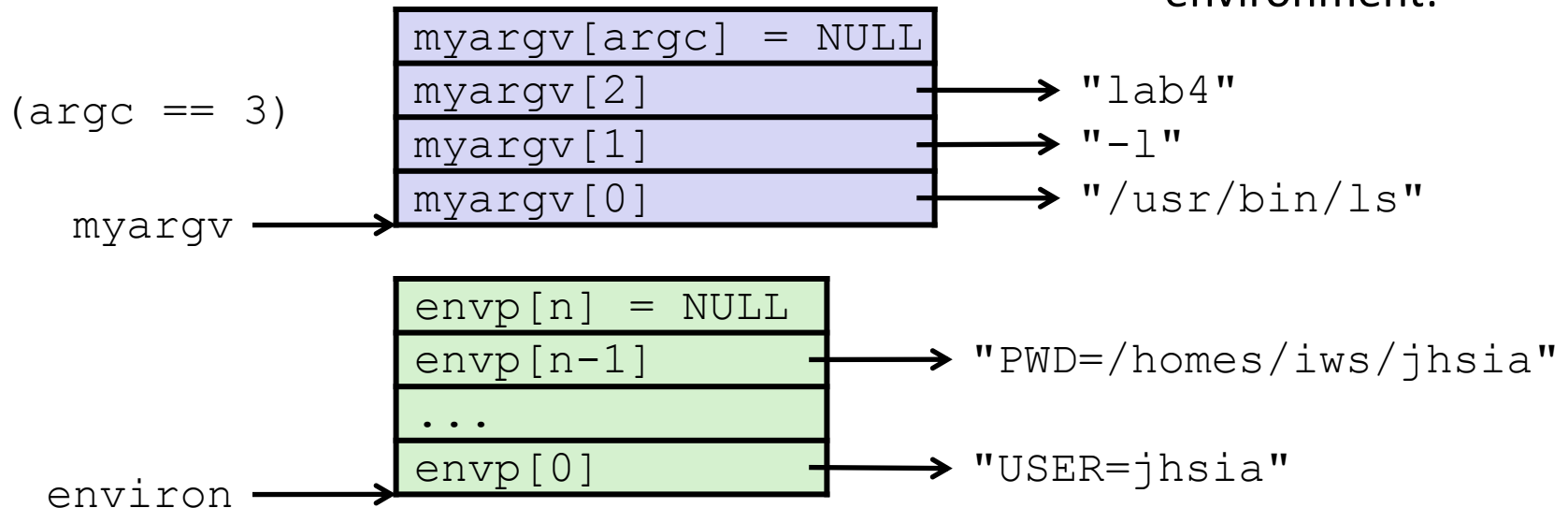

Exec-ing a new program



execve Example

This is extra
(non-testable)
material

Execute `"/usr/bin/ls -l lab4"` in child process using current environment:



```
if ((pid = fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Run the `printenv` command in a Linux shell to see your own environment variables

Stack Structure on a New Program Start

