Exceptional Control Flow and Processes

How the applications interact with the OS

Motivation

- **ECF will help you understand important system concepts:**
  - I/O, processes
  - How applications interact with OS using system calls

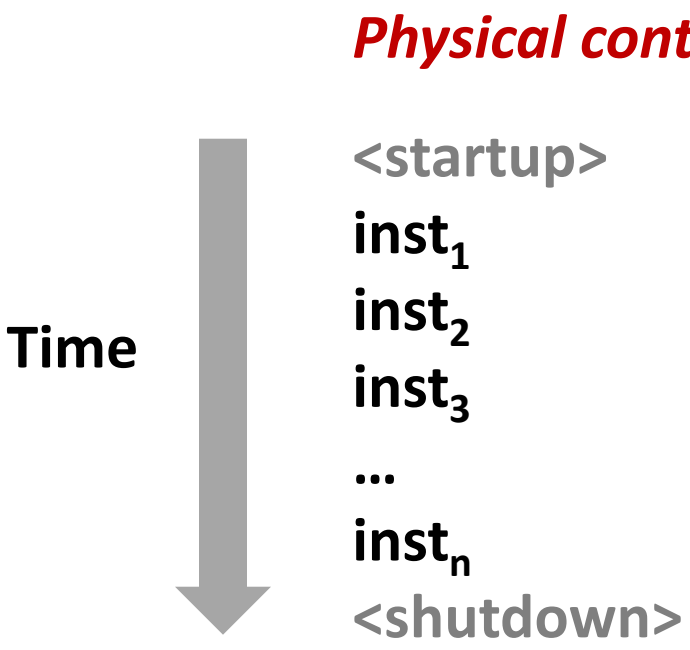
- **You will get an idea**
  - on how web servers and Unix Shell is written
  - about the basic mechanism to implement concurrency
  - How try, catch and throw like mechanism can be implemented in C

Today

- **Exceptional Control Flow**
- Exceptions
- Processes
- Process Control

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 mechanisms for changing control flow:**
  - Jumps and branches (e.g. if, else)
  - Call and return (e.g. function call and returns)

  React to changes in ***program state***

- **Insufficient for a useful system:**
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter(Hardware Interrupt-> OS)
  - Instruction divides by zero (CPU)
  - User hits Ctrl-C at the keyboard (User Interrupt->OS signal)
  - System timer expires (time quantum given to process, related to OS)

- **System needs mechanisms for "exceptional control flow"**
  - Abrupt change in control flow in response to some change

Exceptional Control Flow

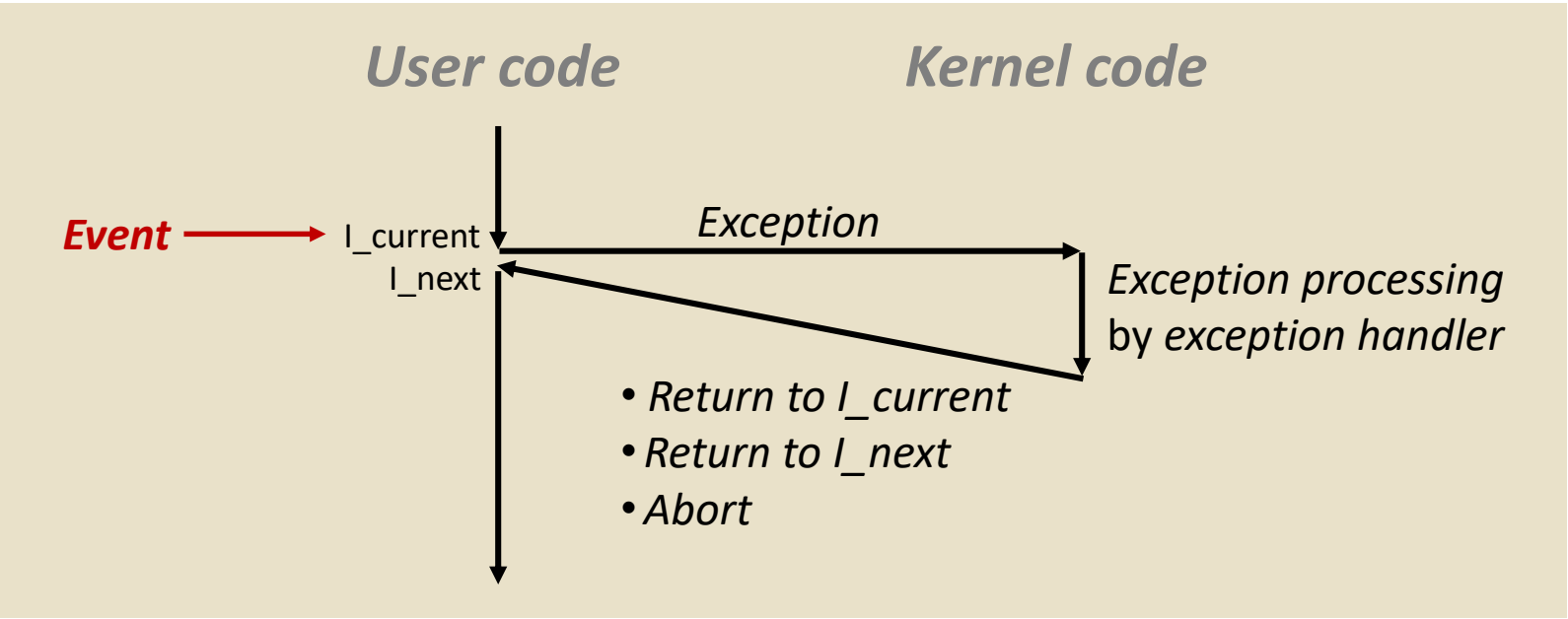
- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., CTRL-C from user, data arrives, divide by 0 error)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software

Today

- Exceptional Control Flow
- **Exceptions**
- Processes
- Process Control

Exceptions

- **An *exception* is a transfer of control to the OS *kernel* in response to some *event***
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Classes of Exceptions

- **Two classes and Four subclasses**
  - Asynchronous Exceptions
    - Interrupts
  - Synchronous Exceptions
    - Traps
    - Faults
    - Aborts

Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- **Examples:**
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Asynchronous Exceptions (Interrupts)

- **Interrupt Request (IRQ pins)**

- **In linux you can use cat proc/interrupts**

Synchronous Exceptions

- **Caused by events that occur as a result of executing current instruction:**
  - ***Traps***
    - Intentional
    - Example: ***system calls***
    - read, fork, exit
    - Returns control to “next” instruction
    - We will with this one
  - ***Faults***
    - Unintentional but possibly recoverable
    - Example: page faults (recoverable), protection fault(non-recoverable)
    - Either re-executes faulting (“current”) instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Typically hardware errors
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

Traps - System Calls

- **Each x86-64 system call has a unique ID number**
- **Examples:**

Number	*Name*	*Description*
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

System Calls

- C program can invoke any system call directly using syscall function.
- However that is not necessary for most of the times,
  - There are available wrappers implemented
- We will use wrappers, instead of using syscall.

```
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

…

int rc;

rc = syscall(SYS_chmod, "/etc/passwd", 0444);
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

…

int rc;

rc = chmod("/etc/passwd", 0444);
```

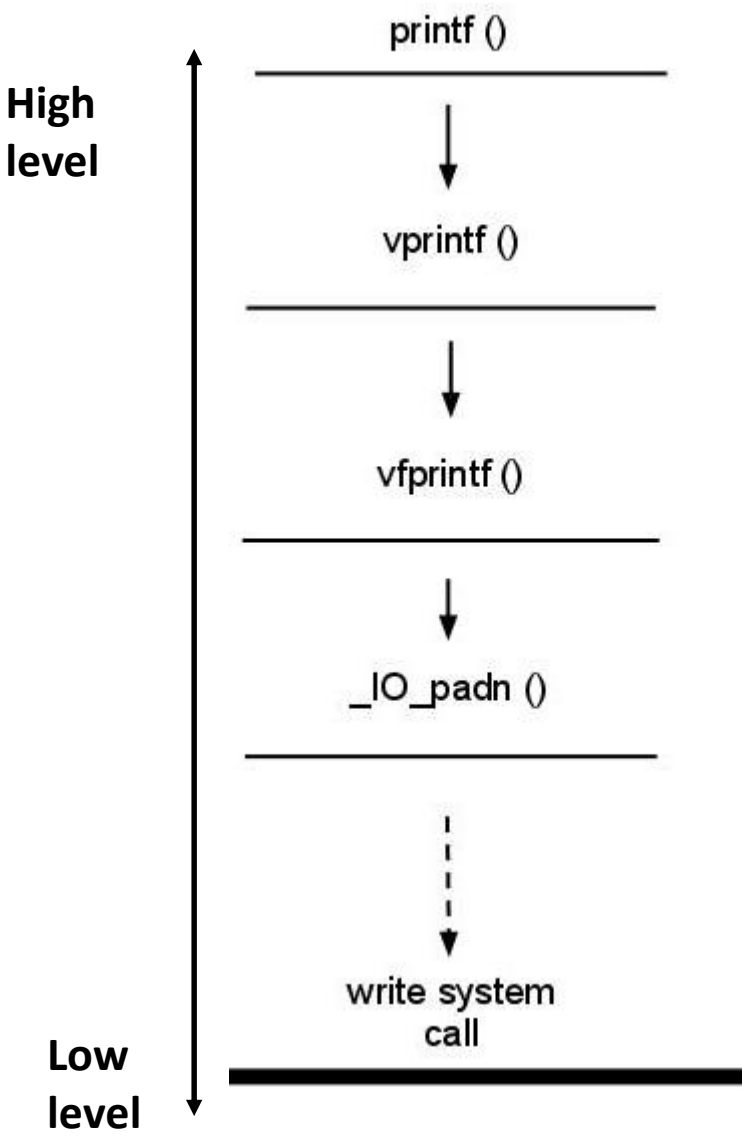
https://www.gnu.org/software/libc/manual/html_node/System-Calls.html

System Calls

```c
#include <unistd.h>
#include <stdio.h>

int main()
{
  write(1, "hello, world\n", 13);
  return 0;
}
```

```
hello, world
```

- First argument sends output to stdout
- Second argument is sequence of bytes to write
- Third argument is number of bytes to write
- There is equivalent system call : sys_write



Today

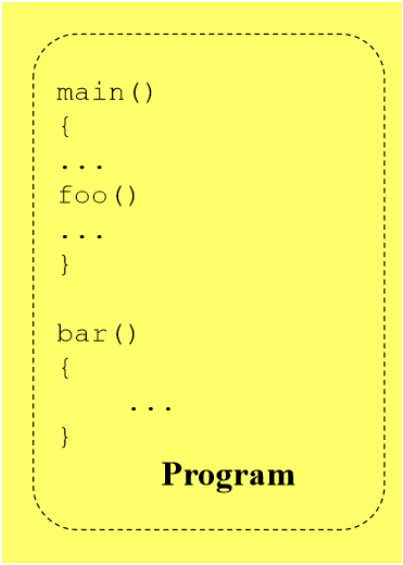
- Exceptional Control Flow
- Exceptions/ System calls intro
- **Processes**
- Process Control

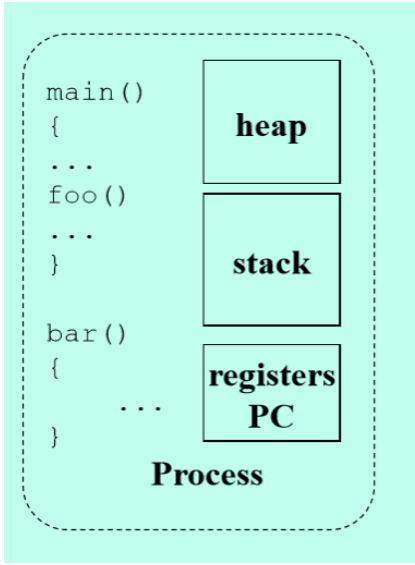
Today

- Exceptional Control Flow
- Exceptions
- **Processes**
  - Management
  - Multiprocessing and Concurrency
  - Creation using fork()
  - Reaping using wait()
  - Example server application
- Process Control

Process

- **Process: a program in execution**
  - One or more threads (units of work)
  - Associated system resources
- **Program vs. process**
  - program: a passive entity
  - process: an active entity
  - Analogy:
    - Program: Recipe
    - Process: Activity of a cook

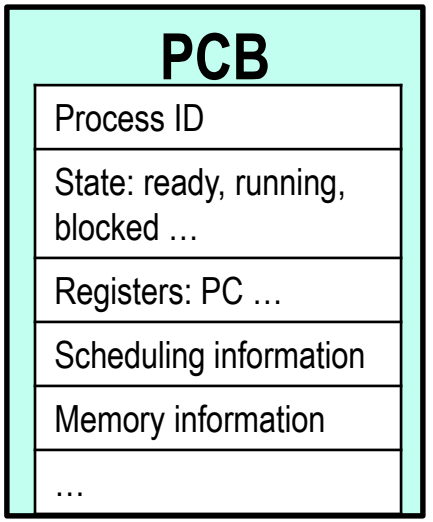




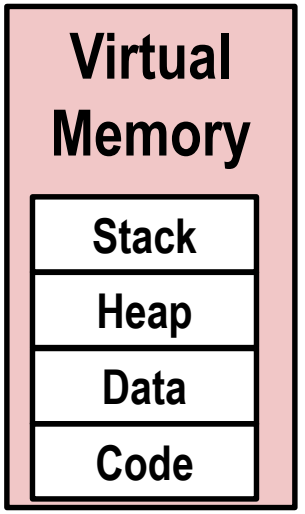
- **For a program to execute, a process is created for that program**
  - A program can invoke more than one process at the same time

Process Management

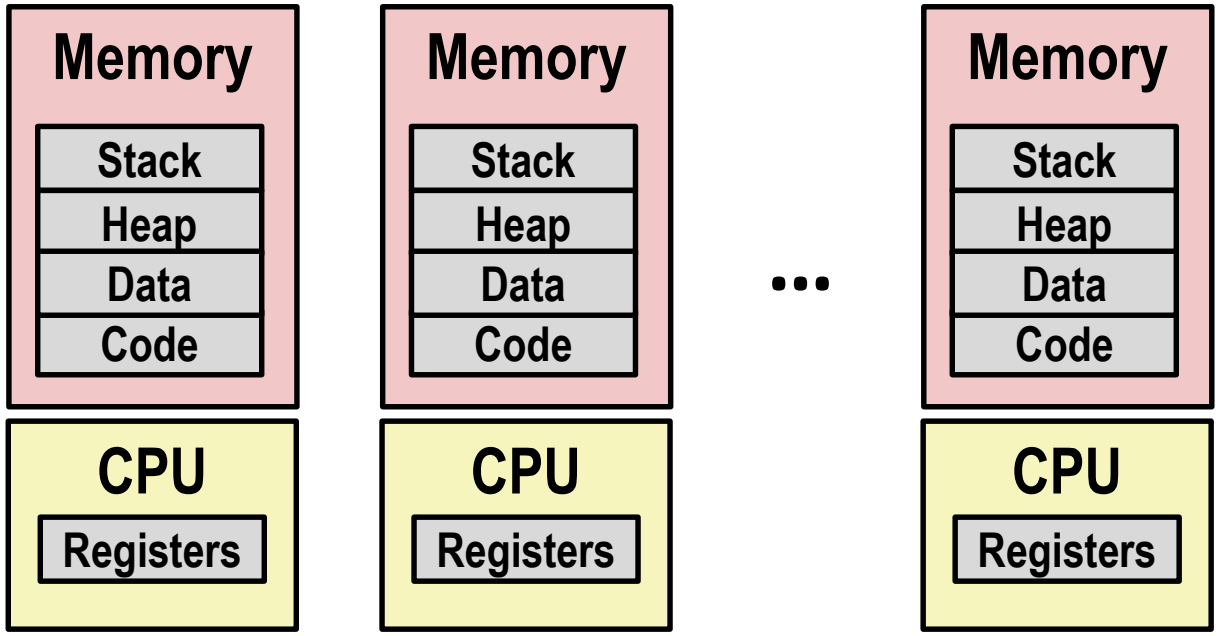
- **Fundamental tasks of Operating System(OS)**
  - Allocation of resources, protecting resources of each process, enabling synchronization, queuing…
  - Process is a unit of scheduling



- **How?**
  - Data Structure called PCB (Process Control Block)
    - Describing state, resource and ownership
    - Process ID and Process Stacks
  - OS provides each process the illusion it has the whole machine for itself.
  - Each process has a dedicated address space within a virtual memory
    - That memory is not accessible by other processes

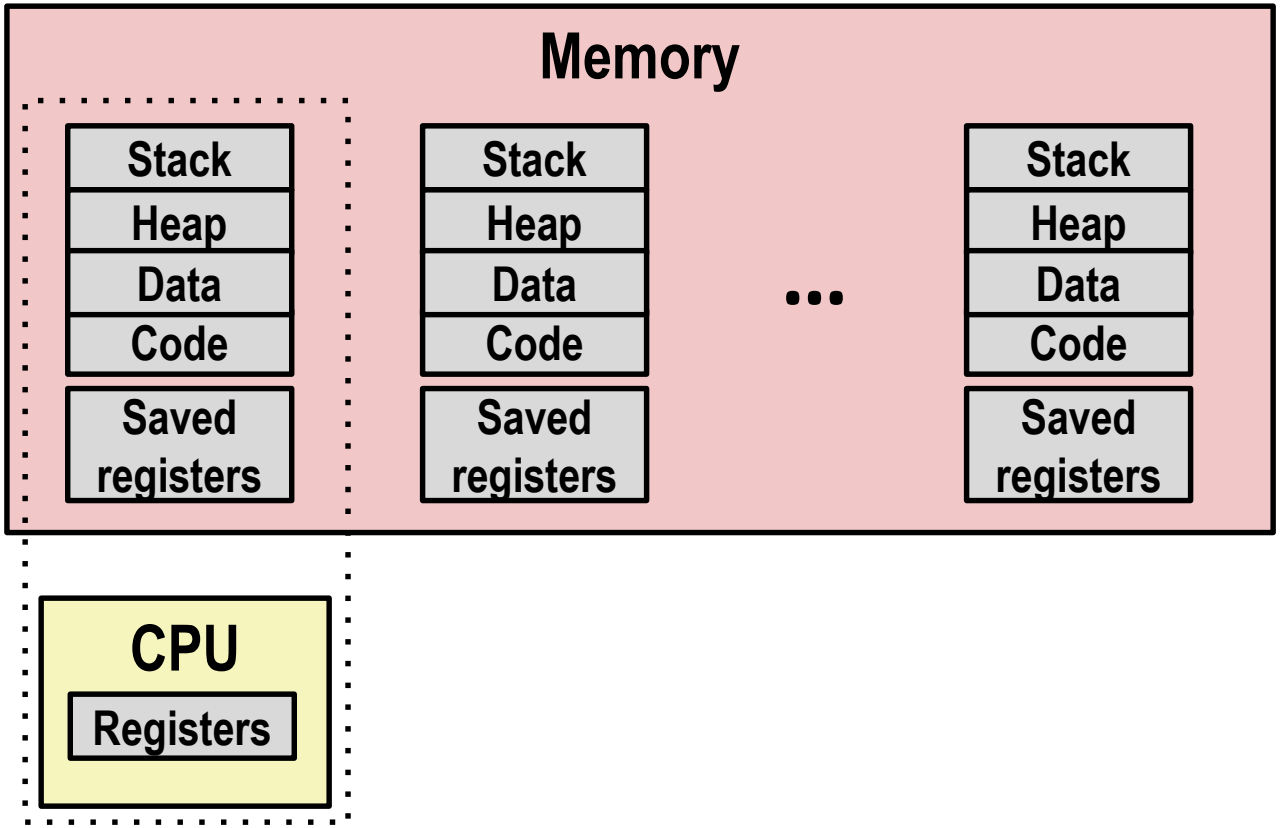


Multiprocessing: The Illusion



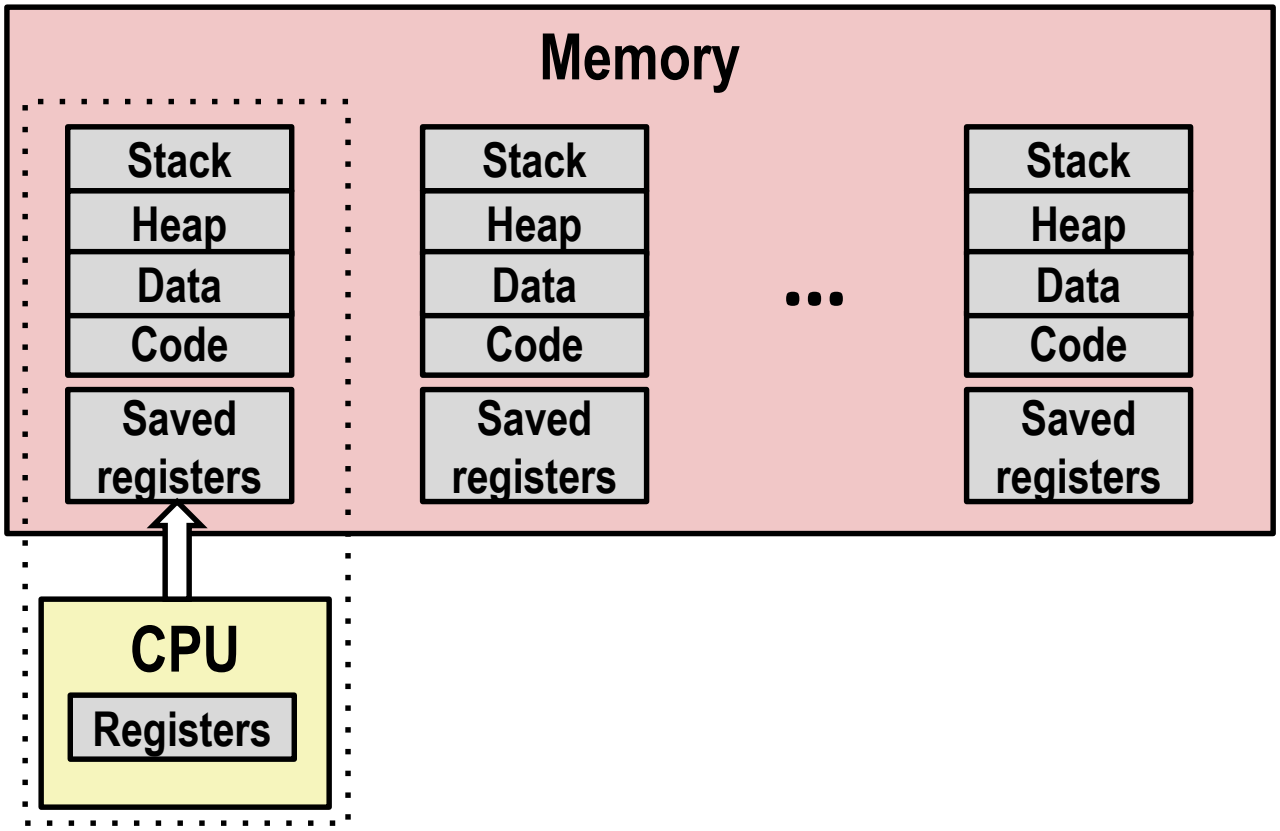
- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing with single CPU: The (Traditional) Reality



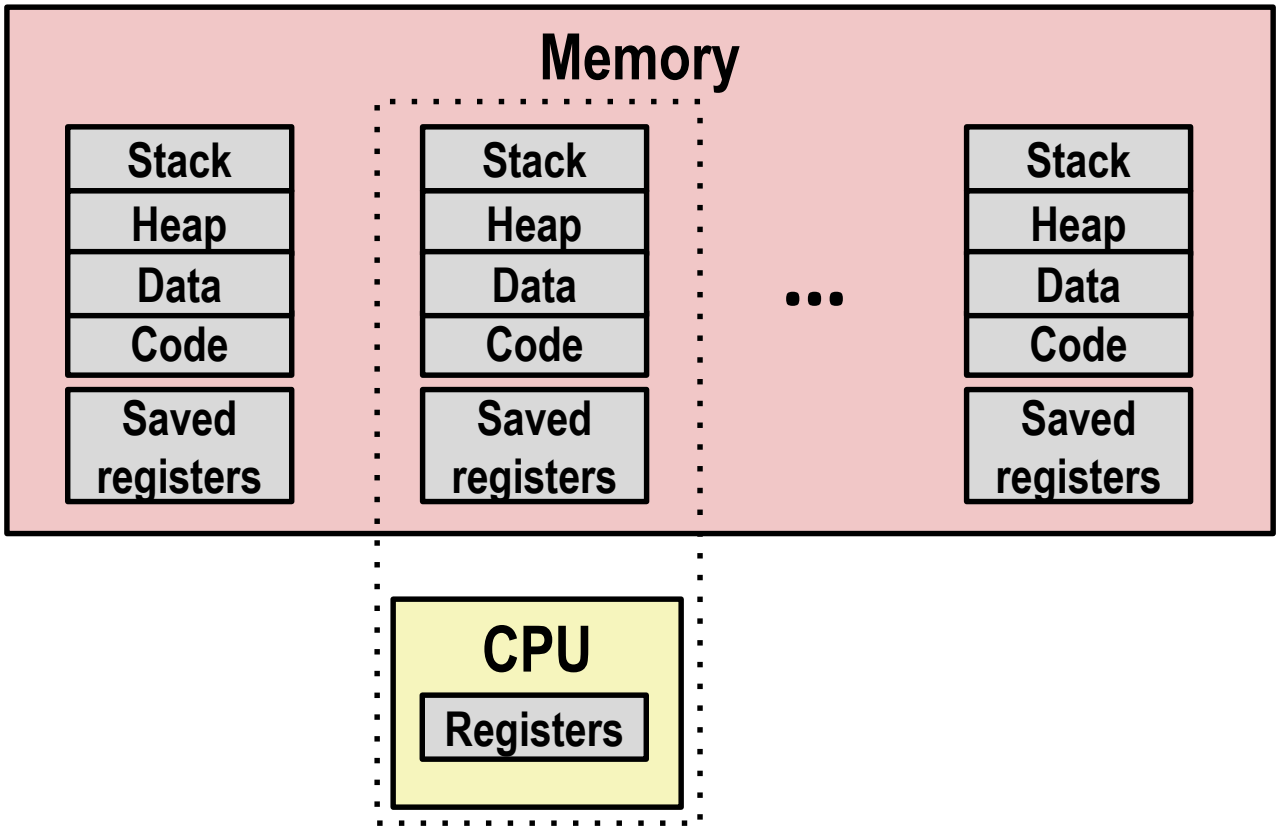
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



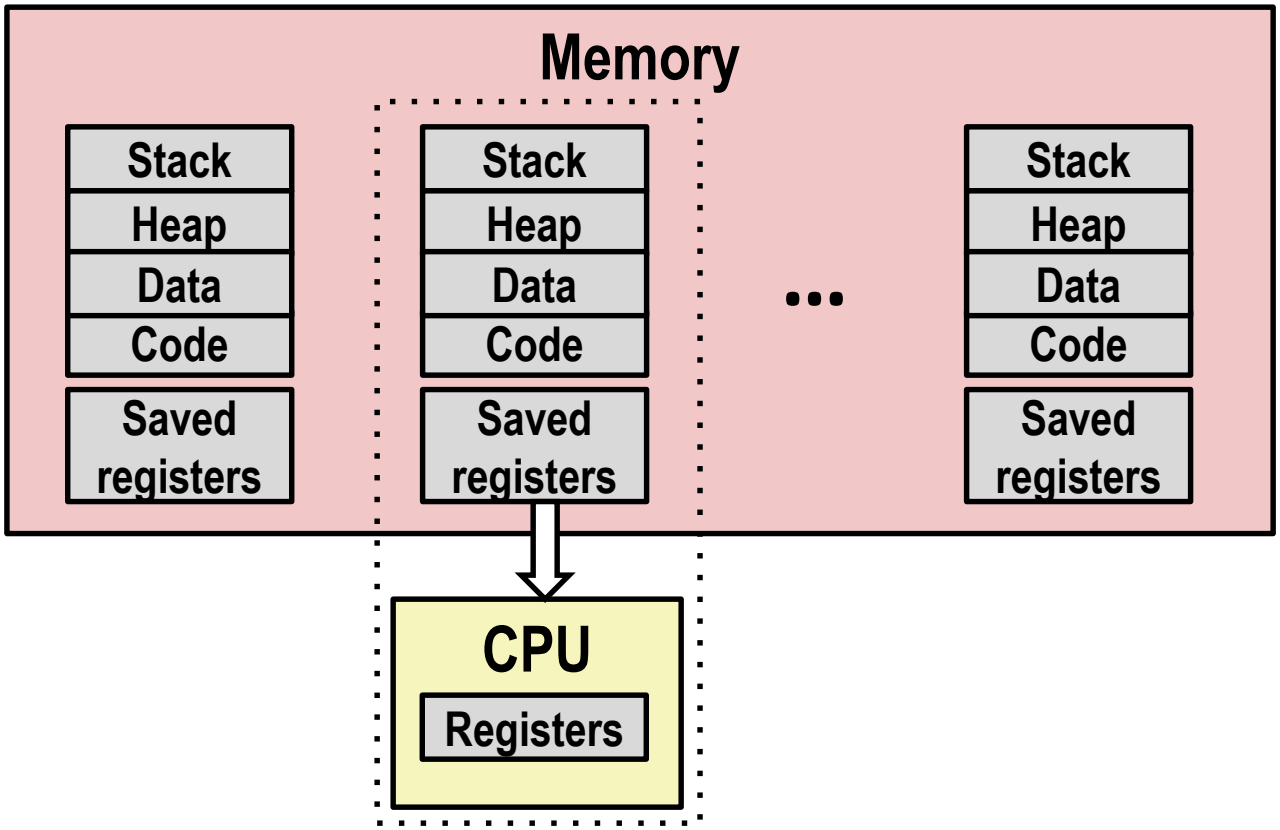
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



- Schedule next process for execution

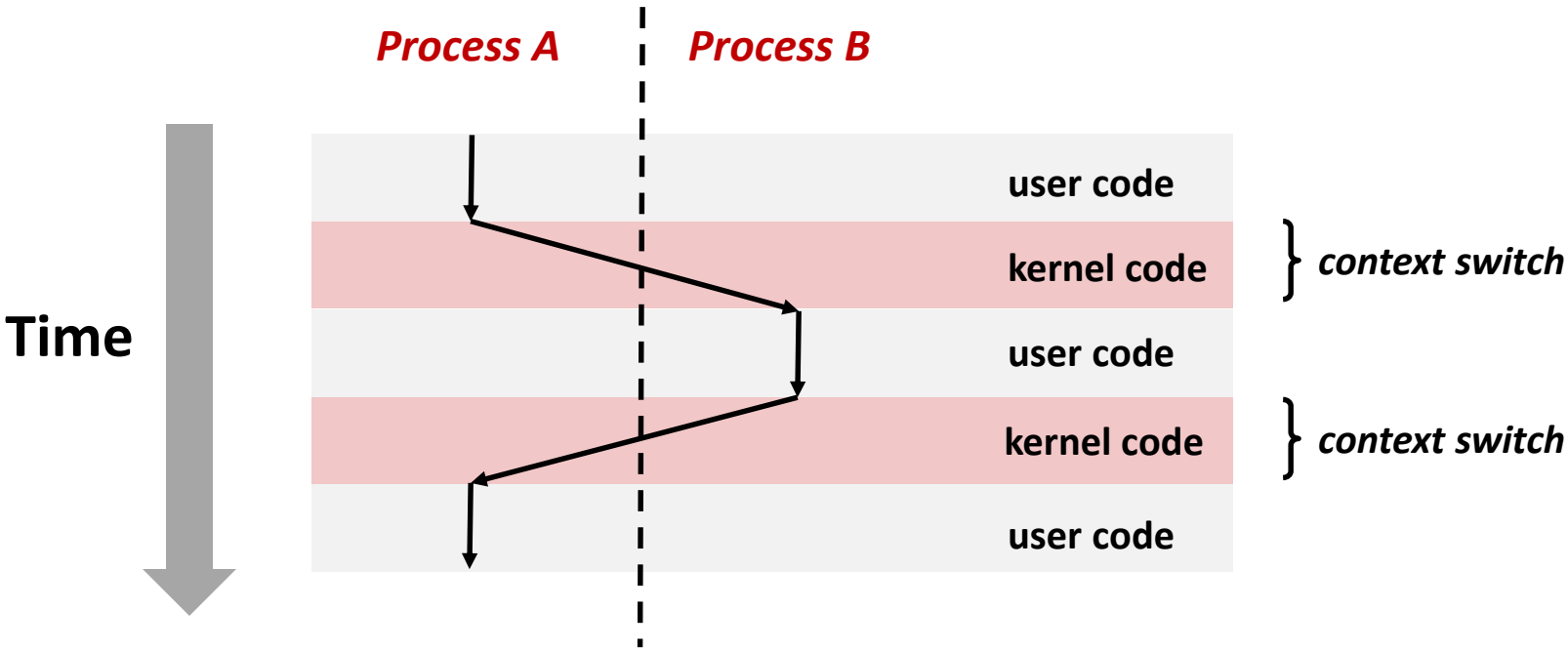
Multiprocessing: The (Traditional) Reality



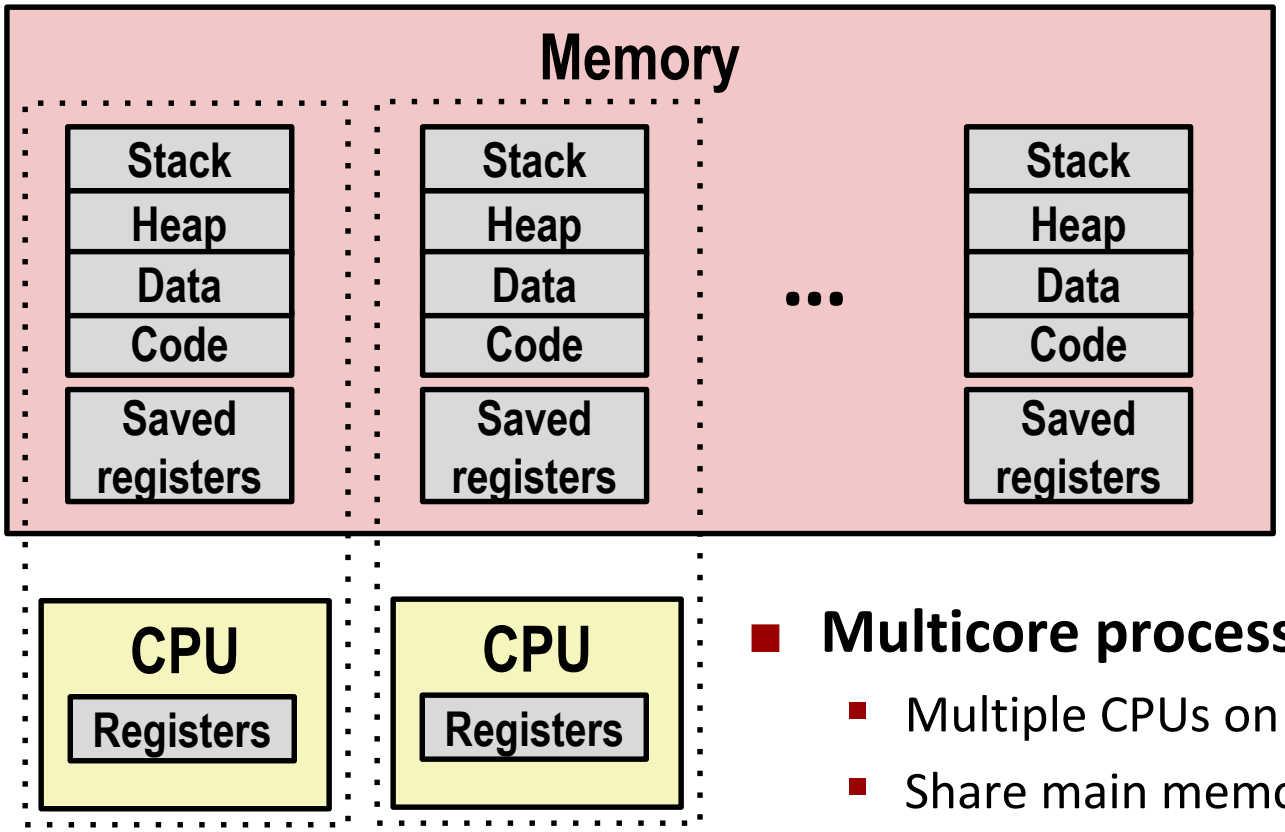
- Load saved registers and switch address space (context switch)

Context Switching

- **Processes are managed by a shared chunk of memory-resident OS code called the *kernel***
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- **Control flow passes from one process to another via a *context switch***



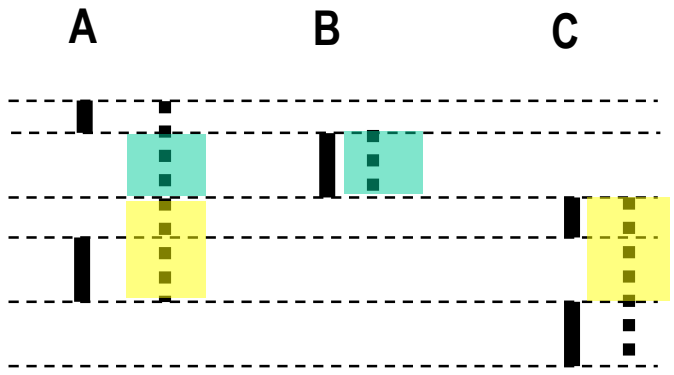
Multiprocessing: The (Modern) Reality



- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

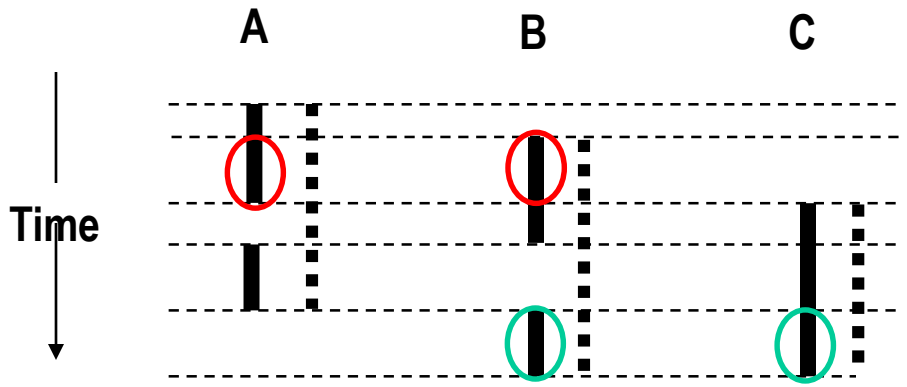
Concurrency and Parallelism

- **Single Core Processor**
  - Simulate parallelism by time slicing



Run 3 processes or threads on 1 core

- **Multi-Core Processor**
  - Can have true parallelism



Run 3 processes or threads on 2 cores

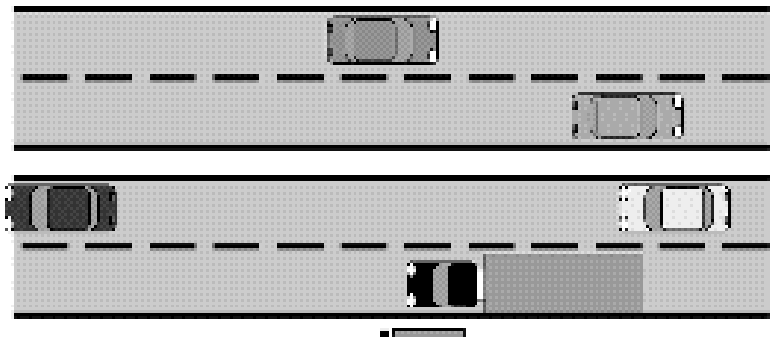
User view

Actual run

Multi-processing
Multi-threading

Concurrency – the easy and the hard

- **Concurrency is easy if there is no interaction**
- **Challenges arise when there is an interaction**
  - Race Conditions
  - Deadlock
  - Starvation
- **We will start with the easy part**



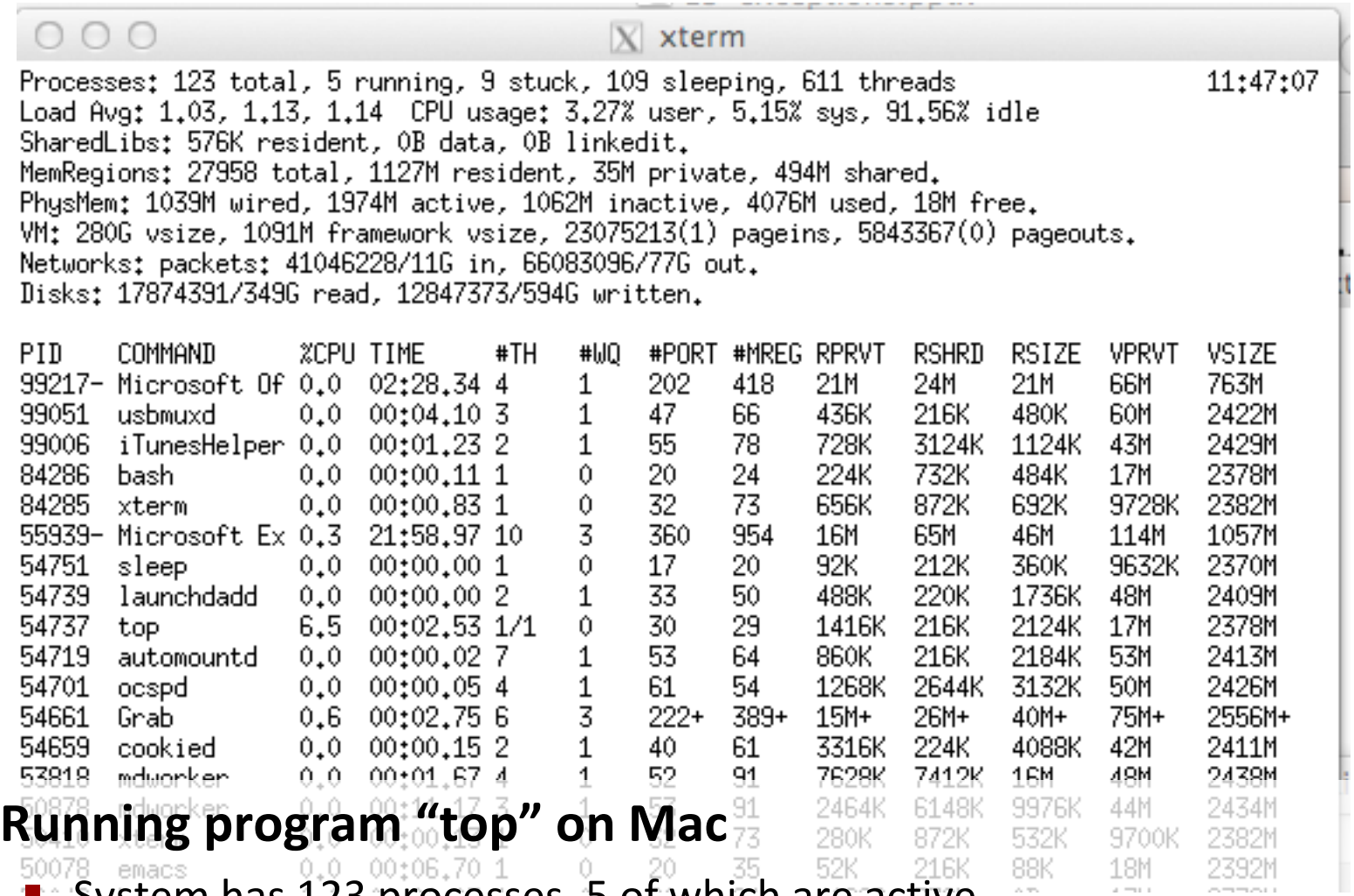
Without interaction

Easy!

With interaction- requires synchronization, mutual exclusion

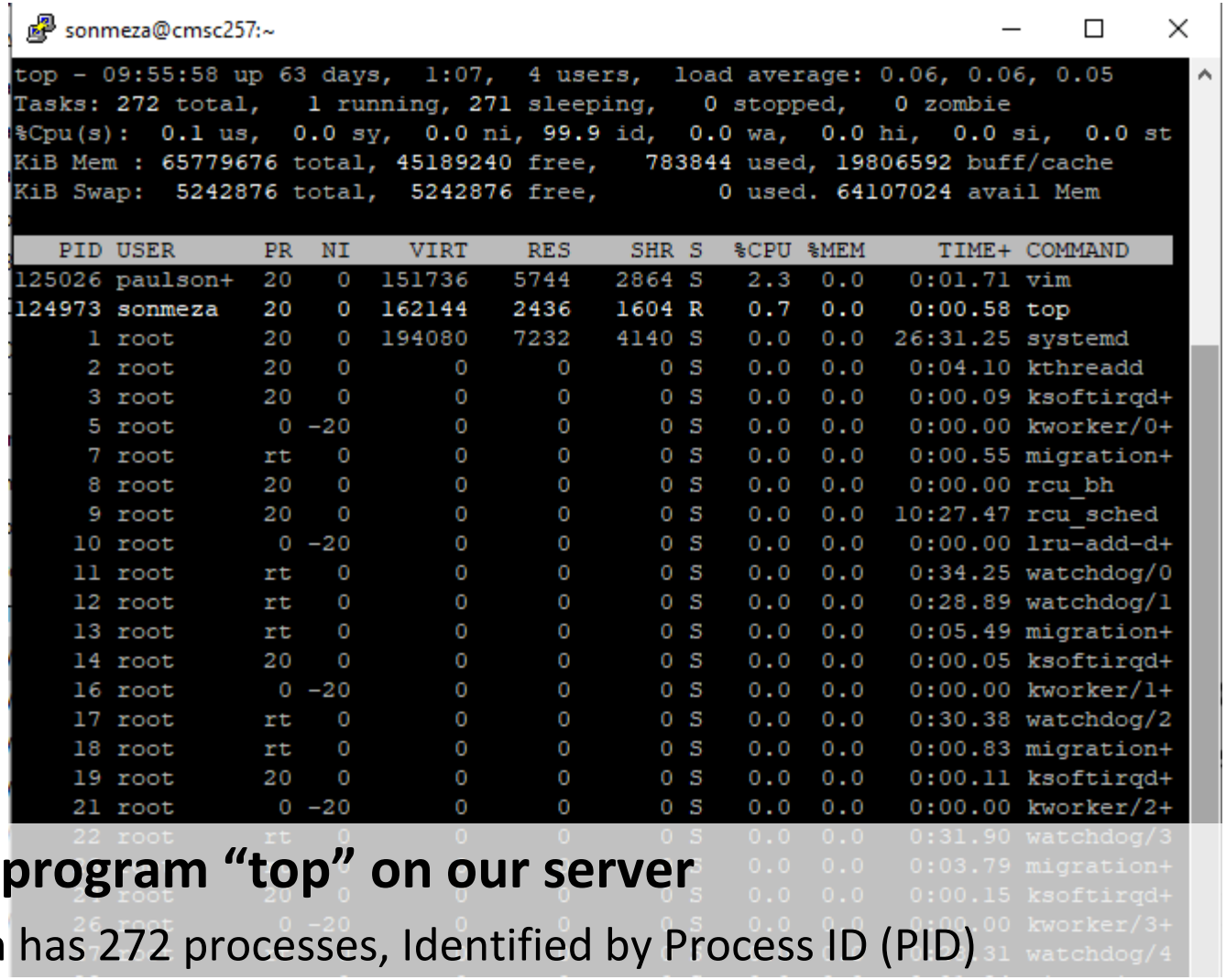
Hard!

Multiprocessing Example



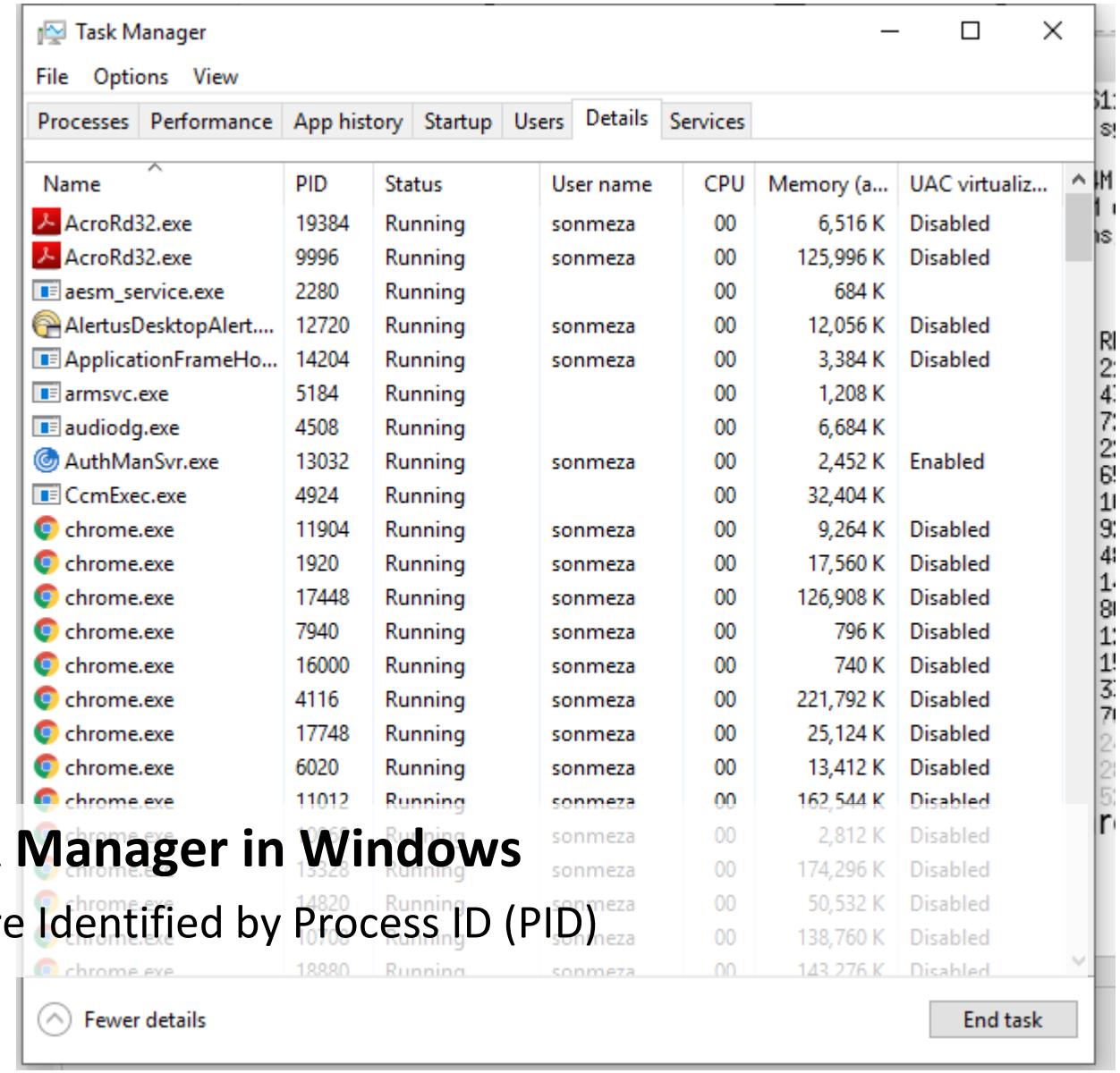
- **Running program "top" on Mac**
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing Example



- **Running program "top" on our server**
  - System has 272 processes, Identified by Process ID (PID)

Multiprocessing Example



- **Running Task Manager in Windows**
  - Processes are Identified by Process ID (PID)

Obtaining Process IDs in C

- The getpid and getppid routines return an integer value of type pid_t, which on Linux systems is defined in types.h as an int.

- **`pid_t getpid(void)`**
  - Returns PID of current process

- **`pid_t getppid(void)`**
  - Returns PID of parent process

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    printf("%d\n", getpid());
    printf("%d\n", getppid());
}
```

```
cs257@cs257-VirtualBox:~/Desktop/processes$ gcc getpid.c -o gpid
cs257@cs257-VirtualBox:~/Desktop/processes$ ./gpid
1251
32163
```

Parent process is bash with PID 32163

```
31771 cs257     20   0 2985660   9908   7708 S  0.0  1.0   0:02.15 pulse
32163 cs257     20   0    6848   4424   3400 S  0.0  0.5   0:00.18 bash
```

Frequent Operations on processes:

- **fork** (give birth to a carbon copy child, from a running parent)
- **exec** (replace contents of program with another, e.g used when shell is running a program),
- **wait** (parent waits on child),
- **exit** (self-termination),
- **kill** (process receiving kill will terminate, unless handles with signal)

Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- **Stopped**
  - Process execution is *suspended* and will not be scheduled until further notice (when we study signals)
- **Terminated**
  - Process is stopped permanently

Terminating Processes

- **Process becomes terminated for one of three reasons:**
  - Receiving a signal whose default action is to terminate (when we cover signals)
  - Returning from the `main` routine
  - Calling the `exit` function
    - Terminates program immediately

- **`void exit(int status)`**
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

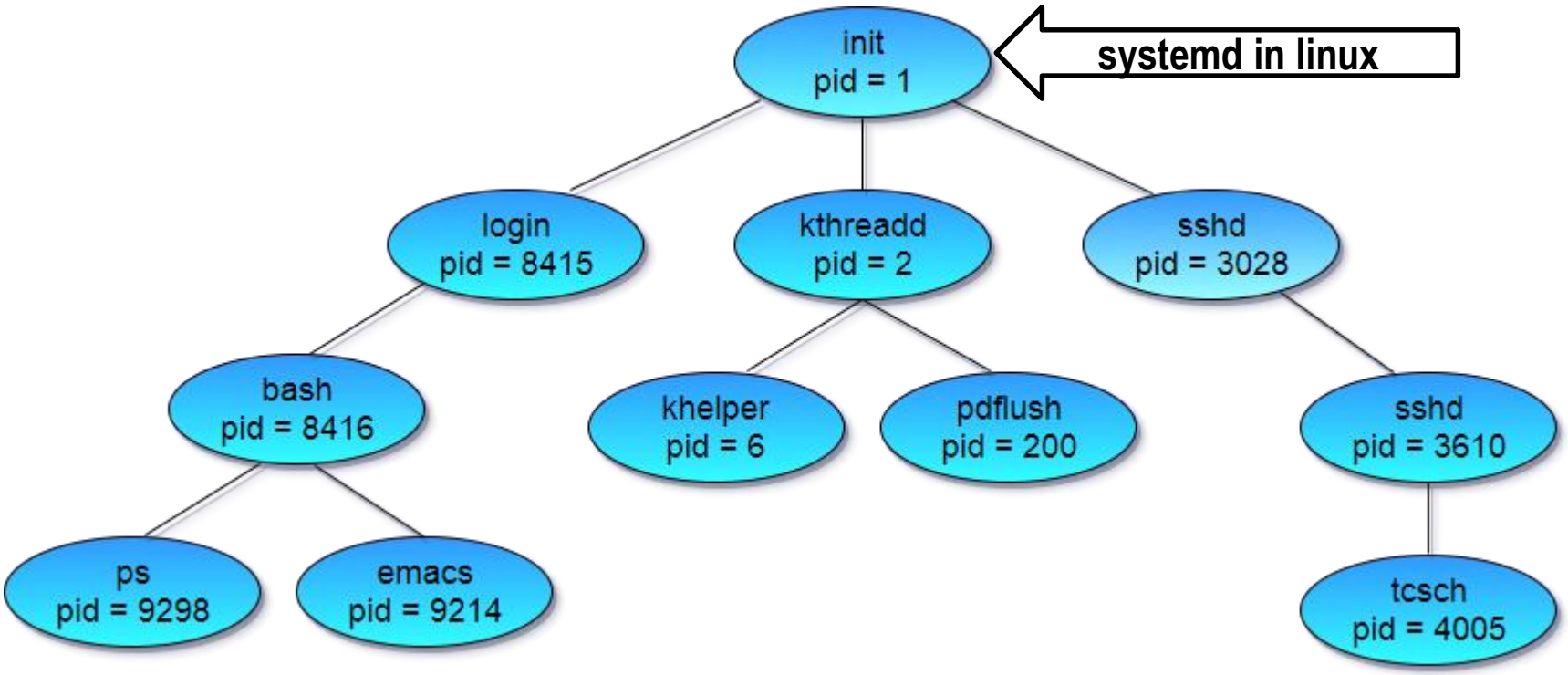
- **`exit` is called once but never returns.**

Creating Processes

- ***Parent process*** **gives birth to a new running** ***child process*** **by calling `fork`**

- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child gets
    - An identical (but separate) copy of the parent's virtual address space.
    - Identical copies of the parent's open file descriptors
    - Different PID than the parent
- **`fork` is interesting (and often confusing) because it is called *once* but returns *twice***

Process family tree in Unix

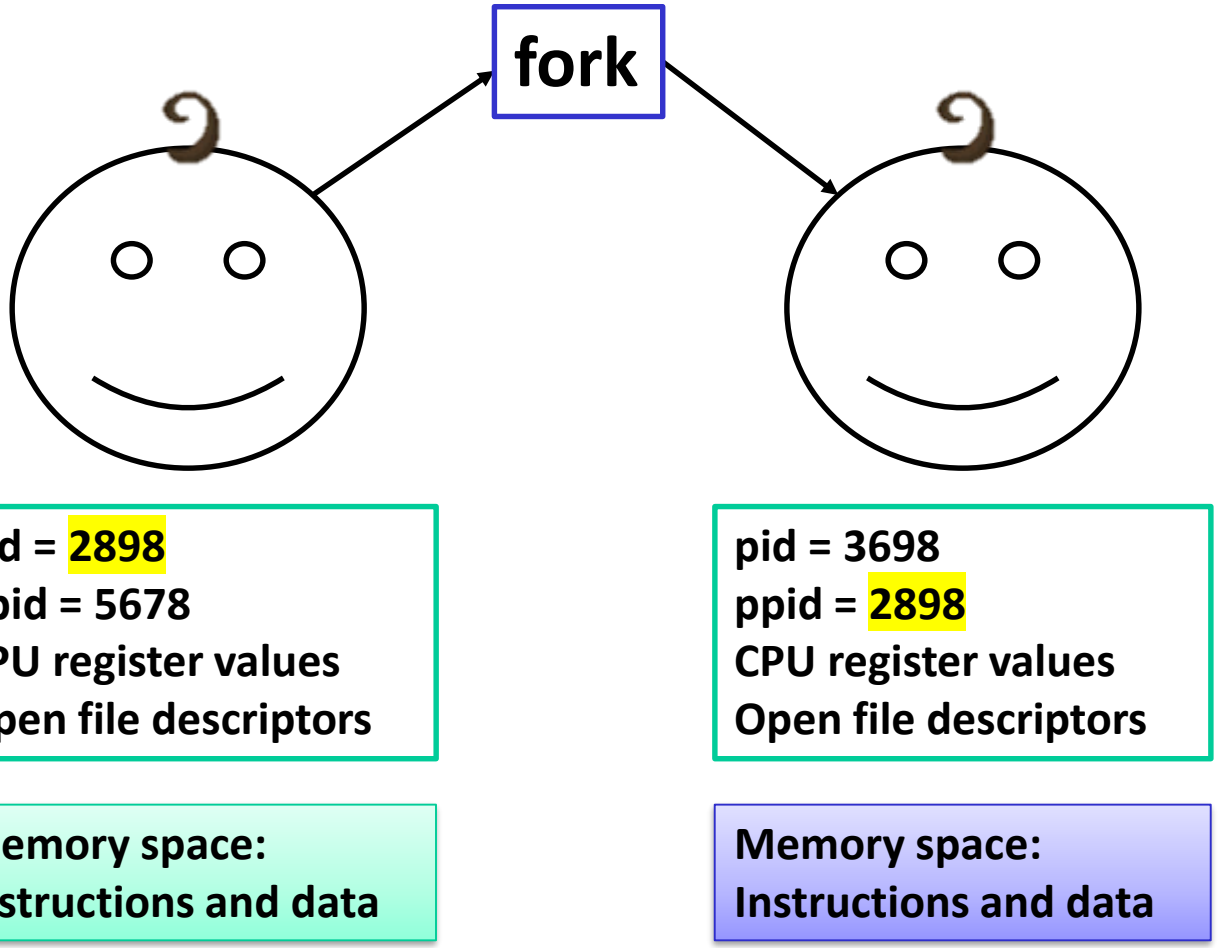


init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes using **fork + exec**

- **fork:** Clones the current process to create a new process
- **exec:** Replaces current program with another one

fork()

- fork creates a clone with same DNA, different fate



fork in action

PID = 4316
Memory int x = 1 pid_t pid =

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
}

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output

fork in action

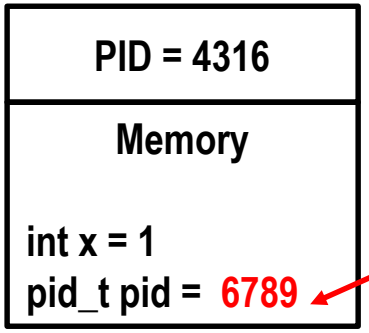
PID = 4316
Memory int x = 1 pid_t pid =

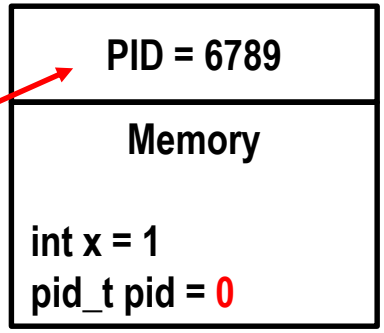
```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output
forking!

fork in action

Multiple Scenarios after this instruction!






```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output

forking!

fork in action – Scenario 1

PID = 4316
Memory int x = 1 pid_t pid = 6789

PID = 6789
Memory int x = 2 pid_t pid = 0

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

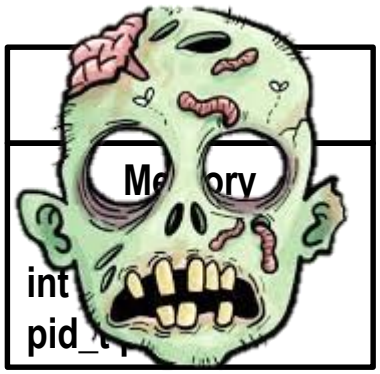
```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child *
      printf("child : x=%d\n", ++x);
      exit(0);    }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output

forking!
child: x = 2

fork in action – Scenario 1

PID = 4316
Memory int x = 1 pid_t pid = 6789



Child will become a Zombie until parent reaps!
Or init reaps(after parent die)
Still consumes some resources

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output

forking!
child: x = 2

fork in action – Scenario 1

PID = 4316
Memory int x = 0 pid_t pid = 6789

PID = 6789


```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output

forking!
child: x = 2
parent: x=0

fork in action – Scenario 1

All resources will be released after parent dies!

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output

forking!
child: x = 2
parent: x=0

fork in action – Scenario 2

PID = 4316
Memory int x = 0 pid_t pid = 6789

PID = 6789
Memory int x = 1 pid_t pid = 0

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output

forking!
parent: x=0

fork in action – Scenario 2

PID = 6789
Memory int x = 1 pid_t pid = 0

Child is orphaned. Will continue as a separate process until it dies

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

```
int main()
{
   pid_t pid;
   int x = 1;
   printf("forking!")
   pid = fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
      exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

Output

forking!
parent: x=0