Process life cycle

Dr. Naser Al Madi

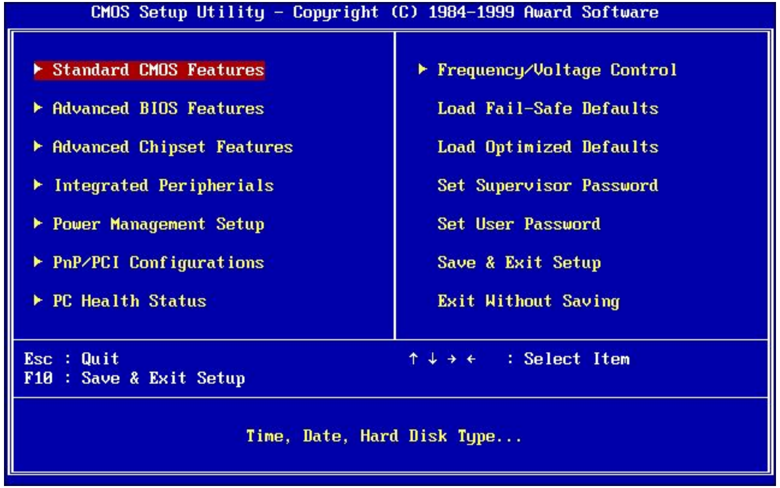
Learning objectives

- Storage structure and caching
- Go over running a process in details + context switching
- Kernel structure and system calls

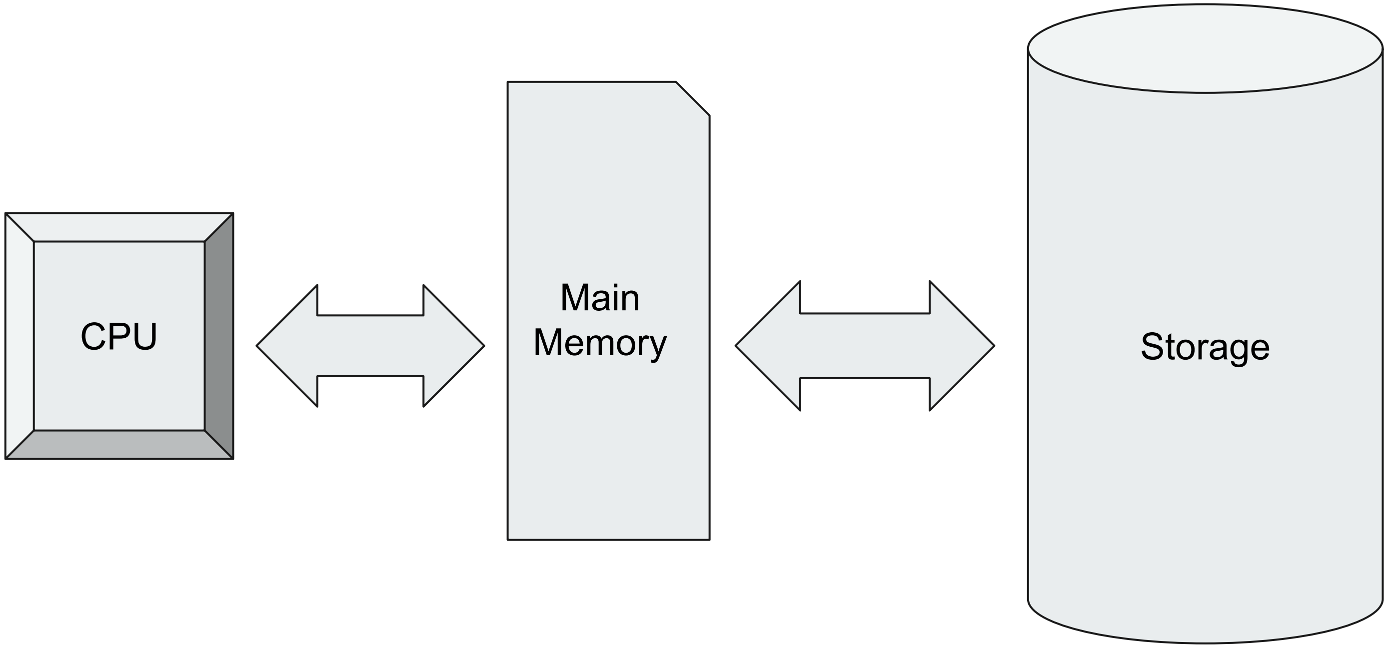
Note to self: record lecture

How BIOS works

http://flint.cs.yale.edu/feng/cos/resources/BIOS/



Storage structure


CPU
Main Memory
Storage

Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - Random access
  - Typically volatile

Secondary storage – extension of main memory that provides large nonvolatile storage capacity

- Hard disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into tracks, which are subdivided into sectors
  - The disk controller determines the logical interaction between the device and the computer
- Solid-state disks – faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular

Storage Structure

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
  - Size

Seagate Portable 1TB External Hard Drive HDD – USB 3.0 for PC, Mac, PlayStation, & Xbox, 1-Year Rescue Service (STGX1000400) , Black

Visit the Seagate Store

177,039 ratings | 1000+ answered questions

Amazon's Choice for "1tb hd"

Price: $49.99 prime & FREE Returns

Thank you for being a Prime member. Get a $100 Gift Card: Pay $0.00 upon approval for the Amazon Prime Rewards Visa Card. No annual fee.

May be available at a lower price from other sellers, potentially without free Prime shipping.

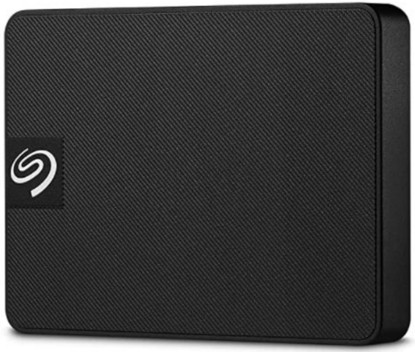
Capacity: **1TB**

500 GB | **1TB** | 2TB | 4TB | 5TB | 6 TB | 8TB | 10TB | 12TB | 14TB | 16TB | 18TB

Style: **Portable HDD**

Portable SSD | **Portable HDD** | Desktop HDD

Color: **Black**



Seagate Expansion SSD 1TB Solid State Drive – USB 3.0 for PC, Laptop and Mac, 3-Year Rescue Service (STJD1000400) , Black

Visit the Seagate Store

177,039 ratings | 1000+ answered questions

$219.99

& FREE Returns

Pay $36.67/month for 6 months, interest-free upon approval for the Amazon Prime Rewards Visa Card

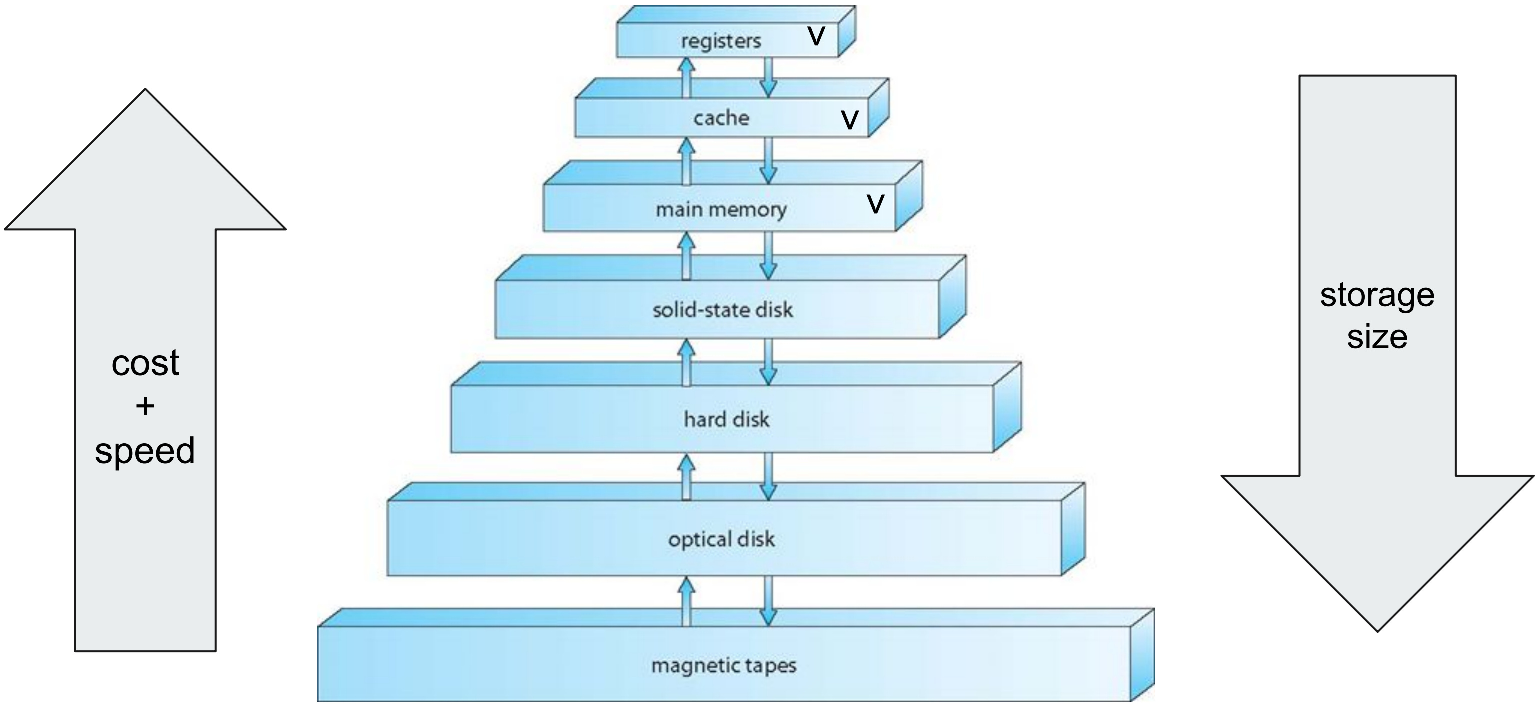
Not eligible for Amazon Prime. Available with free Prime shipping from other sellers on Amazon.

Capacity: **1TB**

500 GB | **1TB** | 2TB | 4TB | 5TB | 6 TB | 8TB | 10TB | 12TB | 14TB | 16TB | 18TB

Style: **Portable SSD**

Portable SSD | Portable HDD | Desktop HDD


cost
+
speed
registers
V
cache
V
main memory
V
solid-state disk
hard disk
optical disk
magnetic tapes
storage
size

*v: volatile

Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
    - If it is, information used directly from the cache (fast)
    - If not, data copied to cache and used there
- Cache smaller than storage being cached
    - Cache management important design problem
    - Cache size and replacement policy

Running a process

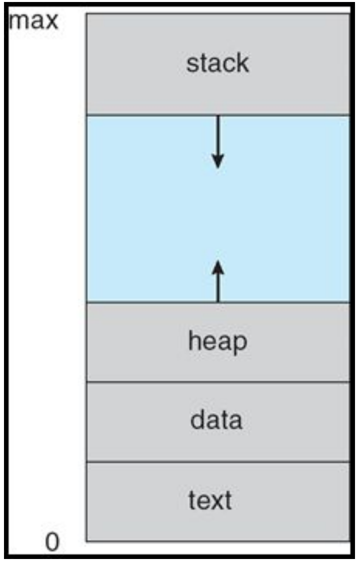
In more detail

Process Concept

- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- **Program** is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of a program starts via GUI mouse clicks, command line entry of its name, system startup, by another process, etc
- One program can be several processes
  - Consider multiple users executing the same program
- Process has multiple parts – next slides

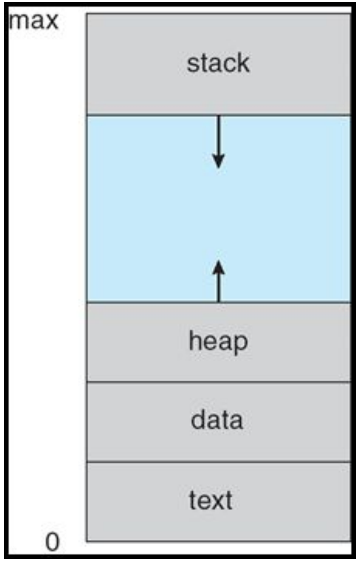
Process in Memory

- **context** - the entire state of the process at any instant
  - program code
  - Data section (global variables)
  - heap (dynamic data) When does data get placed in heap?
  - procedure call stack - contains temporary data - subroutine parameters, return addresses, local variables
  - register contents
    - general purpose registers
    - program counter (PC) — address of next instruction to be executed
    - stack pointer (SP)
  - OS resources in use - open files, connections to other programs accounting information



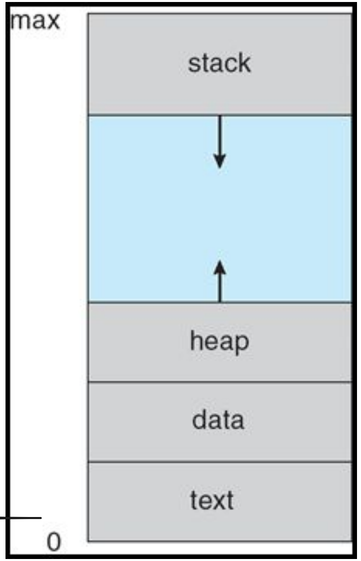
Process memory map

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

int calls;

void fact(int a, int *b){
    calls++;
    if (a == 1) return;
    *b = *b * a;
    fact(a - 1, b);
}

int main(){
    int n, *m;

    scanf("%d", &n);
    m = malloc(sizeof(int));
    *m = 1;
    fact(n, m);
    print("factorial (%d) is %d\n", n ,*m);
    free(m);
}
```



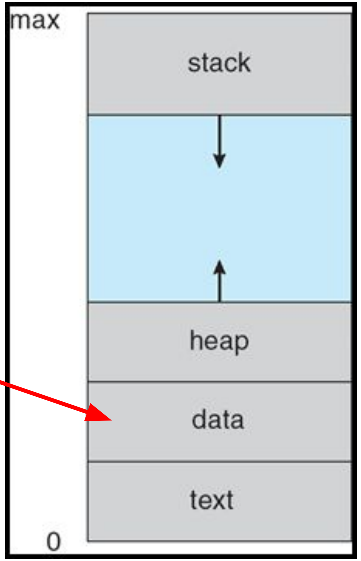
Process memory map

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

int calls;

void fact(int a, int *b){
    calls++;
    if (a == 1) return;
    *b = *b * a;
    fact(a - 1, b);
}

int main(){
    int n, *m;

    scanf("%d", &n);
    m = malloc(sizeof(int));
    *m = 1;
    fact(n, m);
    print("factorial (%d) is %d\n", n ,*m);
    free(m);
}
```



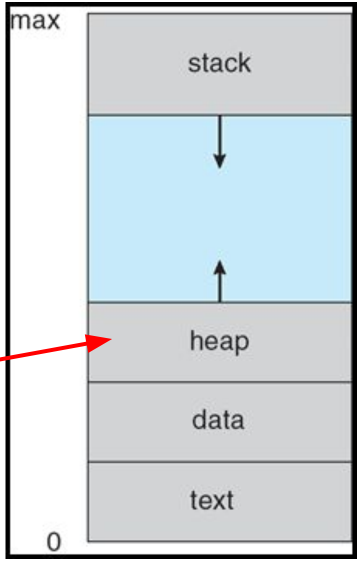
Process memory map

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

int calls;

void fact(int a, int *b){
    calls++;
    if (a == 1) return;
    *b = *b * a;
    fact(a - 1, b);
}

int main(){
    int n, *m;

    scanf("%d", &n);
    m = malloc(sizeof(int));
    *m = 1;
    fact(n, m);
    print("factorial (%d) is %d\n", n ,*m);
    free(m);
}
```



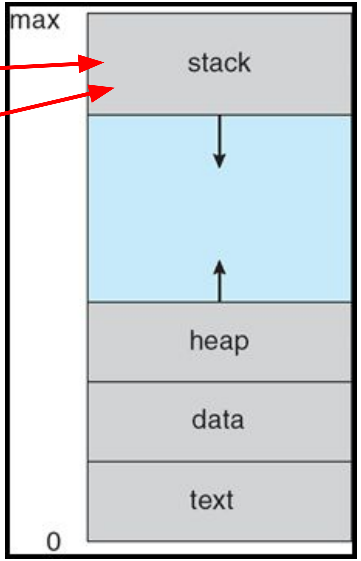
Process memory map

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

int calls;

void fact(int a, int *b){
    calls++;
    if (a == 1) return;
    *b = *b * a;
    fact(a - 1, b);
}

int main(){
    int n, *m;

    scanf("%d", &n);
    m = malloc(sizeof(int));
    *m = 1;
    fact(n, m);
    print("factorial (%d) is %d\n", n ,*m);
    free(m);
}
```



Process memory map

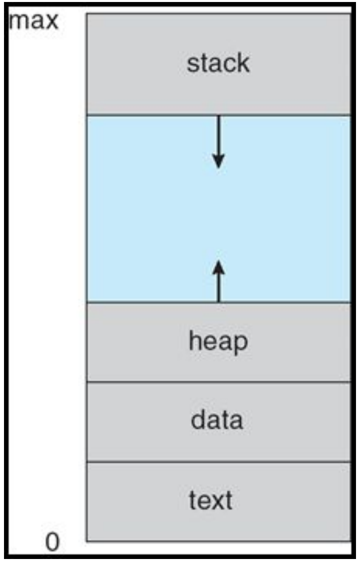
```c
#include <stdio.h>
#include <stdlib.h>

int calls;

void fact(int a, int *b){
    calls++;
    if (a == 1) return;
    *b = *b * a;
    fact(a - 1, b);
}

int main(){
    int n, *m;

    scanf("%d", &n);
    m = malloc(sizeof(int));
    *m = 1;
    fact(n, m);
    print("factorial (%d) is %d\n", n ,*m);
    free(m);
}
```



In-class Practice: Process memory map

For this Java code, where does the highlighted code go?

```
public class Example{
    public static int[] createArray(int n){
        int[] array = new int[n];
        for(int i = 0; i < n; i++){
            array[i] = 0;
        }
        return array;
    }

    public static void main(String [] args){
        int num = 10;
        int[] array = createArray(num);
    }
}
```



Compiling code

- In reality, readable code never makes it to memory or CPU
- Compilers/interpreters convert code into a language closer to hardware, a common choice is assembly language
- Java and Python are a little too complicated for the scope of this example, so I will focus on our new favorite language C.

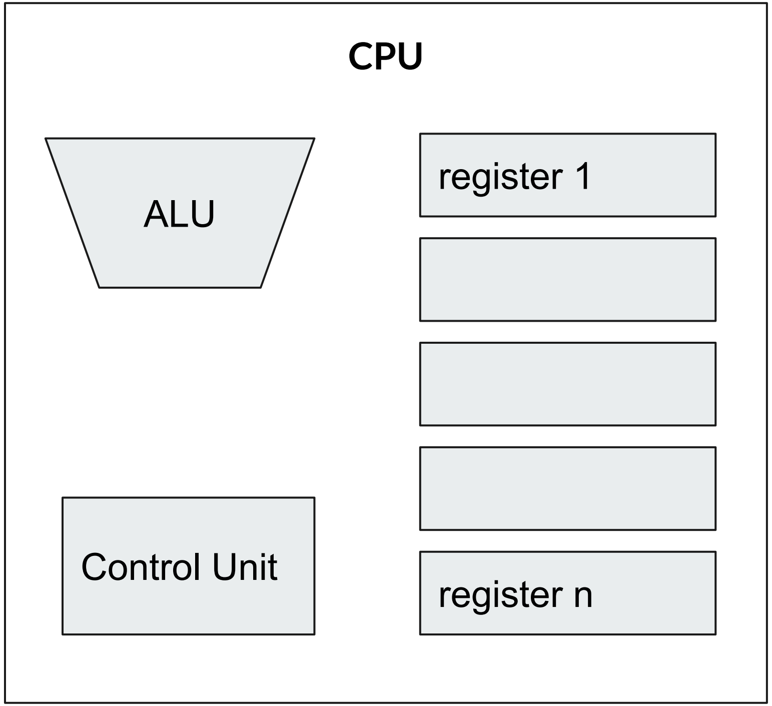
Compiling C code

```c
int main() {

    int number1, number2, sum;

    number1 = 10;
    number2 = 20;

    // calculating sum
    sum = number1 + number2;

    return 0;
}
```

compile

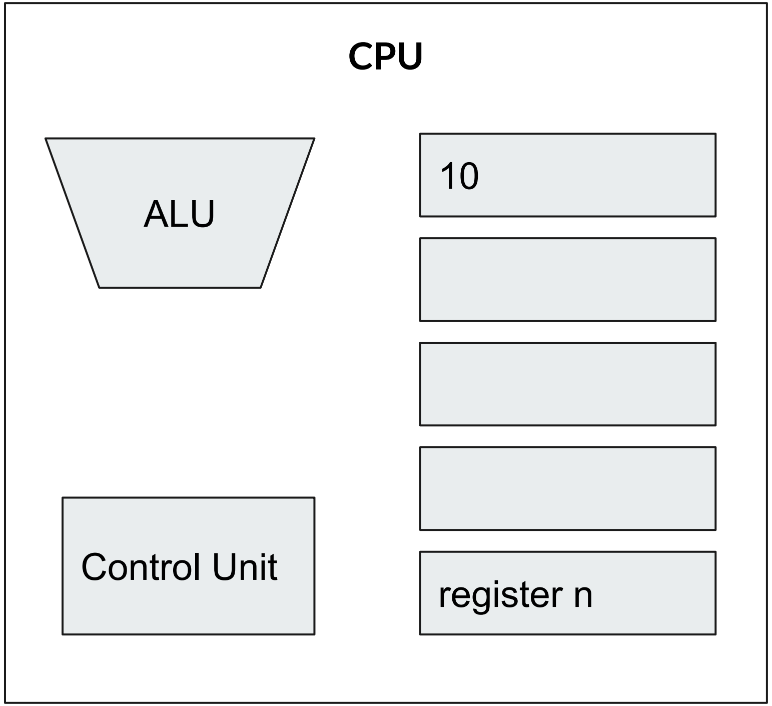
pseudo assembly

LD	number1	register1
LD	number 2	register2
AD	register1	register2
ST	sum	



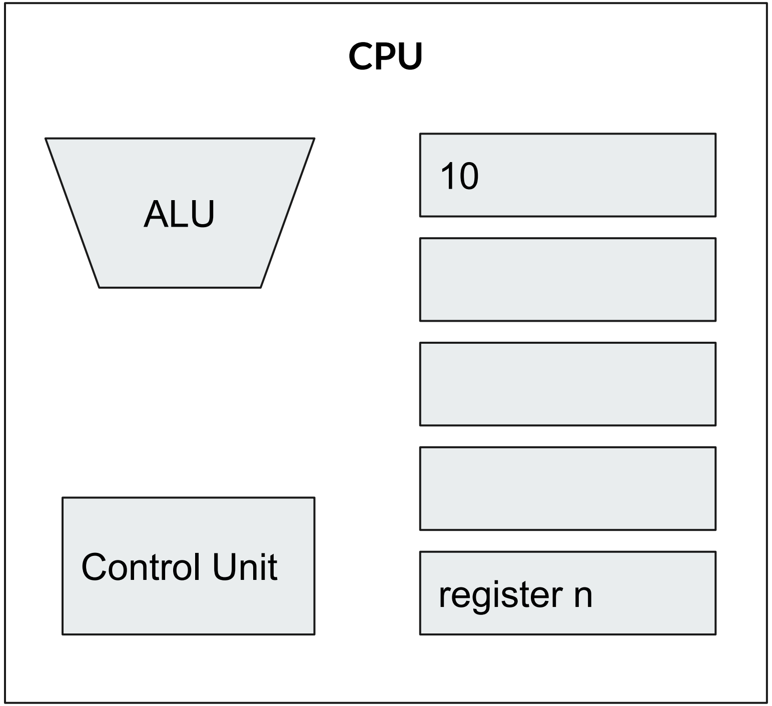
pseudo assembly

```
LD    number1     register1
LD    number 2    register2
AD    register1   register2
ST    sum
```



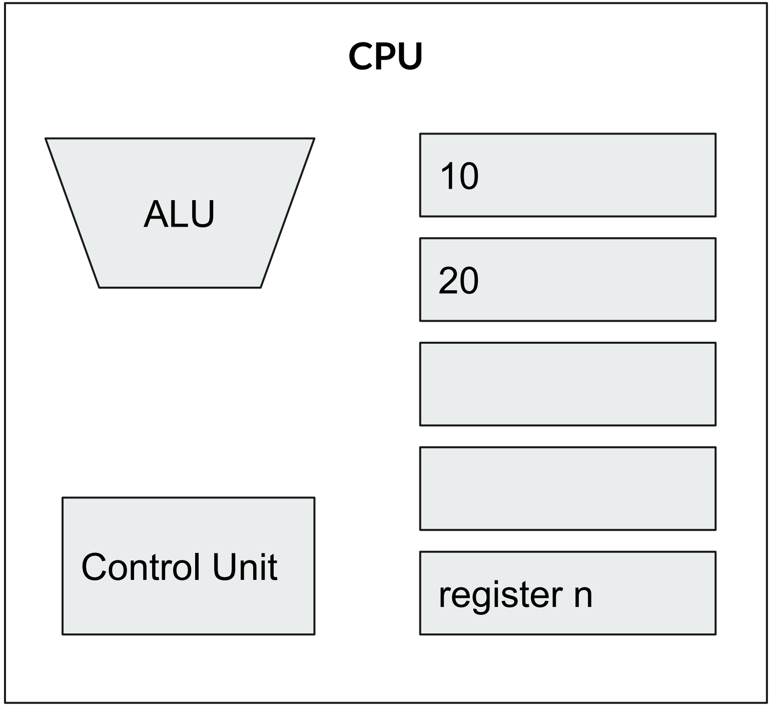
pseudo assembly

```
LD    number1     register1
LD    number 2    register2
AD    register1   register2
ST    sum
```



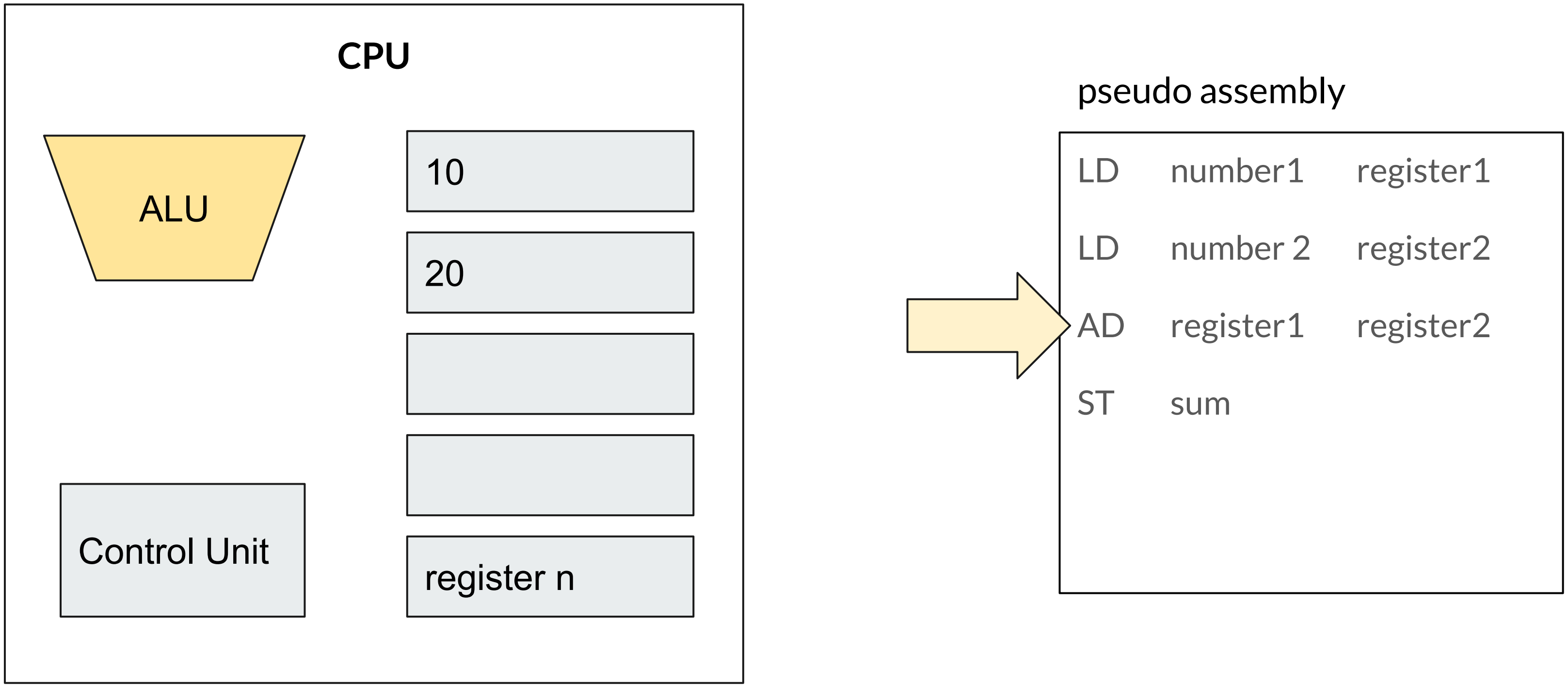
pseudo assembly

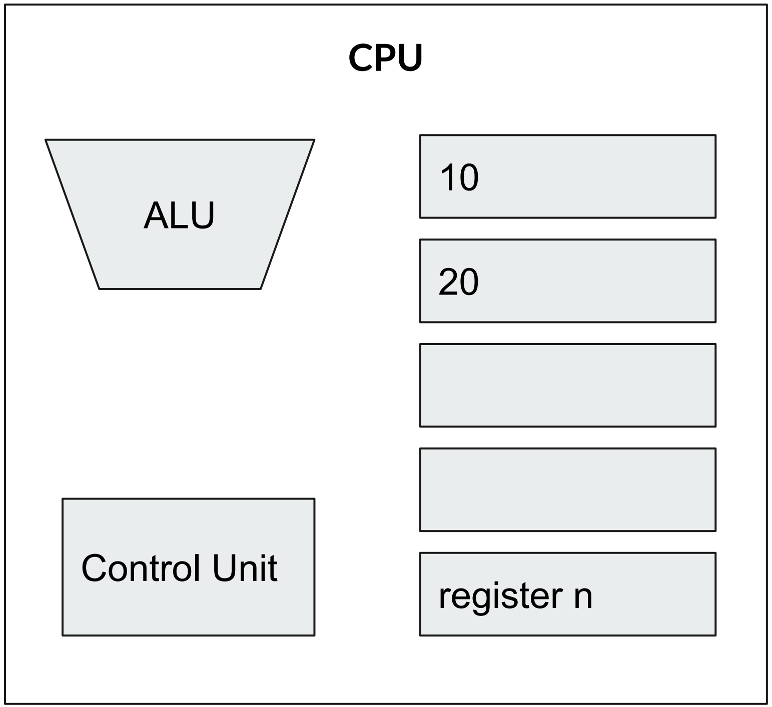
```
LD    number1     register1
LD    number 2    register2
AD    register1   register2
ST    sum
```



pseudo assembly

```
LD      number1      register1
LD      number 2     register2
AD      register1    register2
ST      sum
```


CPU
ALU
10
20
Control Unit
register n
pseudo assembly
LD number1 register1
LD number 2 register2
AD register1 register2
ST sum



pseudo assembly

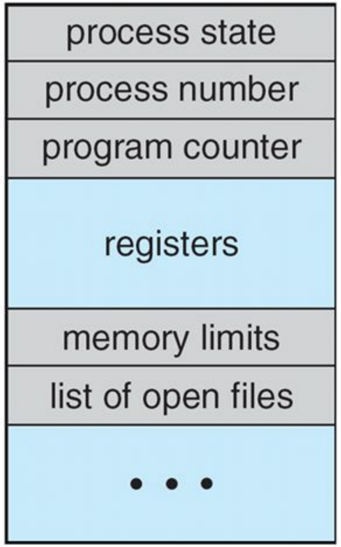
LD	number1	register1
LD	number 2	register2
AD	register1	register2
ST	sum	

Stores sum = 30 in cache near CPU or in main memory

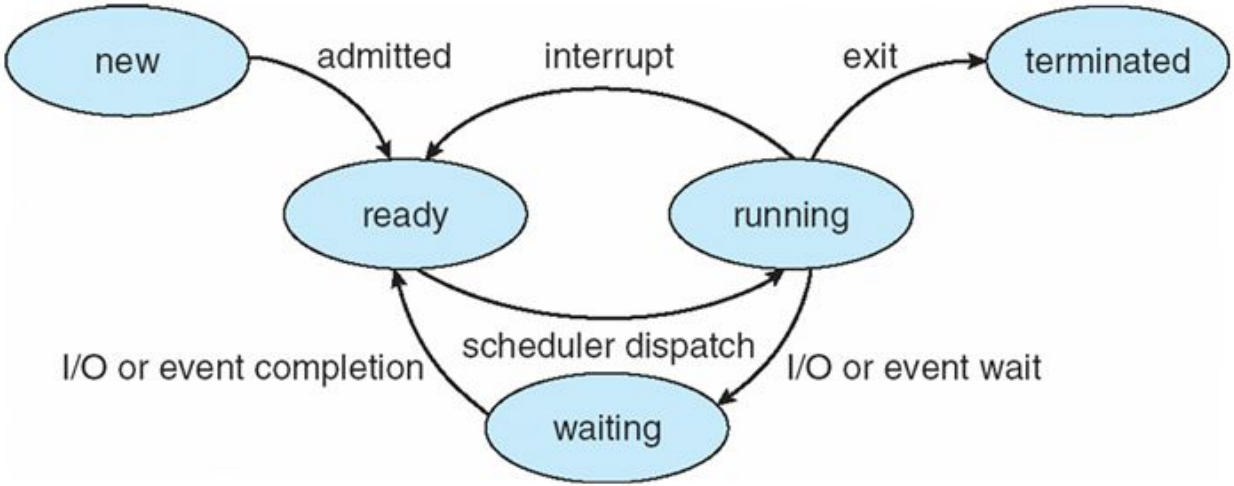
Process life cycle and context switching

Process Control Block (PCB)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

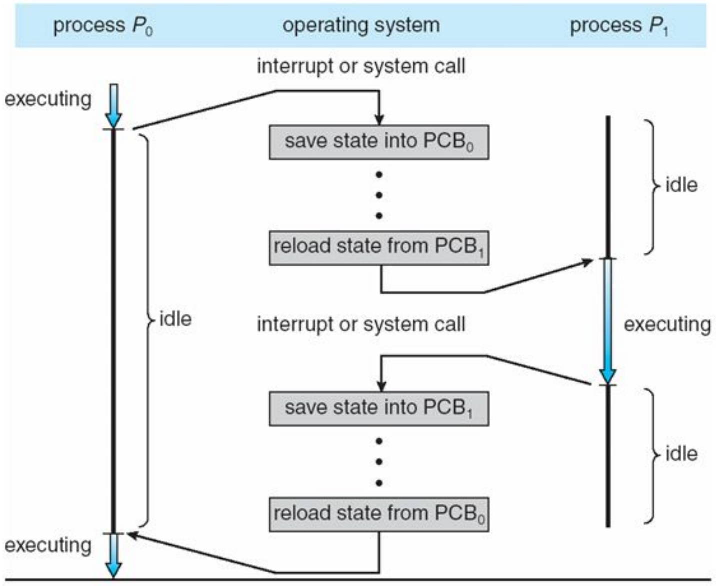


Process States



Context Switching: CPU Switch from one processor to another

- Time sharing OSs allow for multiple processes to run at the same time (concurrency).
- This means that processes will be swapped in and out of the CPU very quickly (remember juggling).
- This quick swapping process is called Context Switching.
- The context or a process is saved, another process context is loaded, then the CPU continues executing.
- This increases CPU utilization!

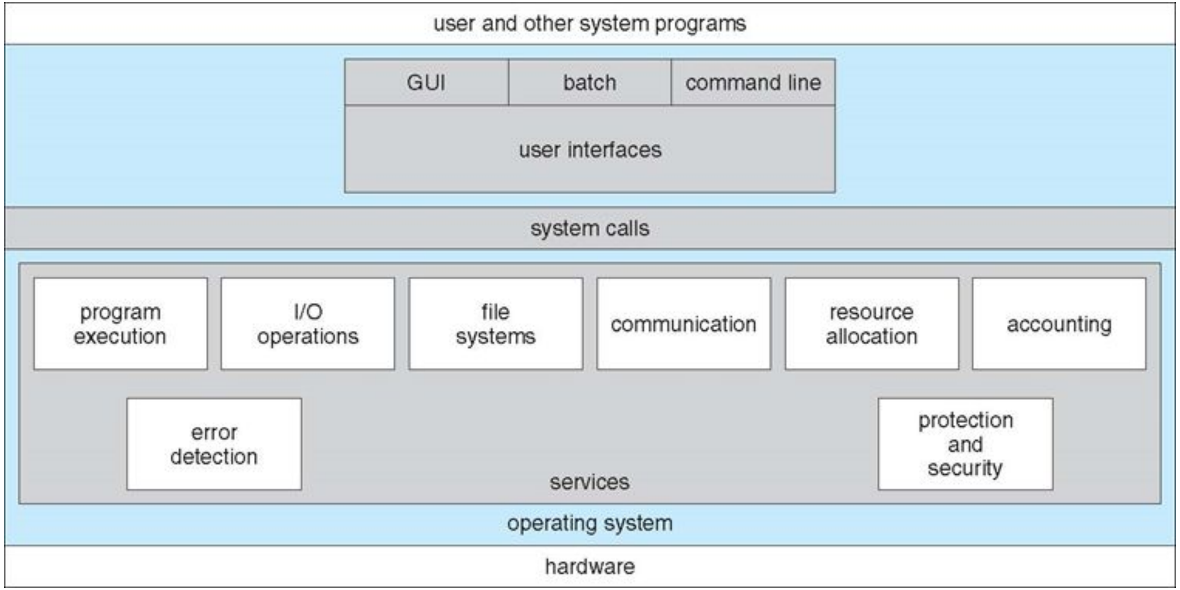


Python turtle example to show concurrency

Solar system example

System calls

A View of Operating System Services



System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

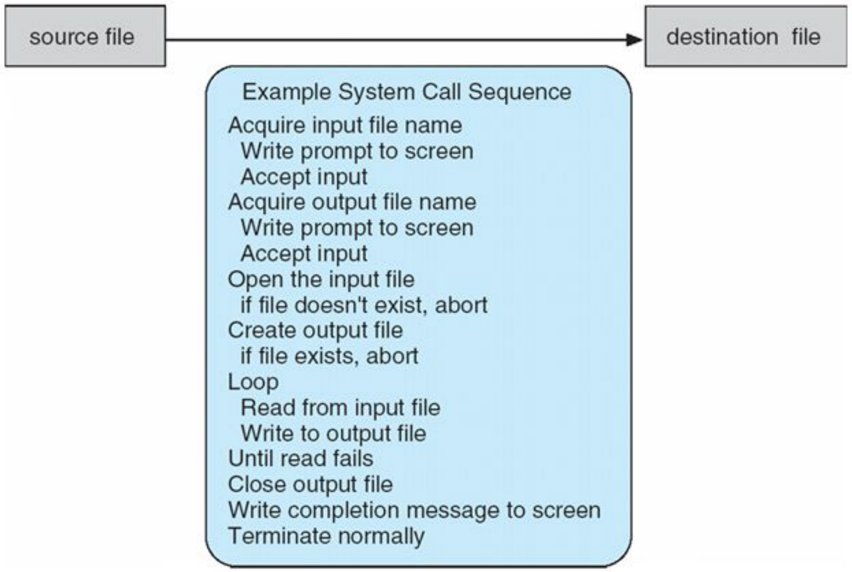
Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Example of System Calls

- System call sequence to copy the contents of one file to another file.

```
>>cp    \path\source    \path\destination
```

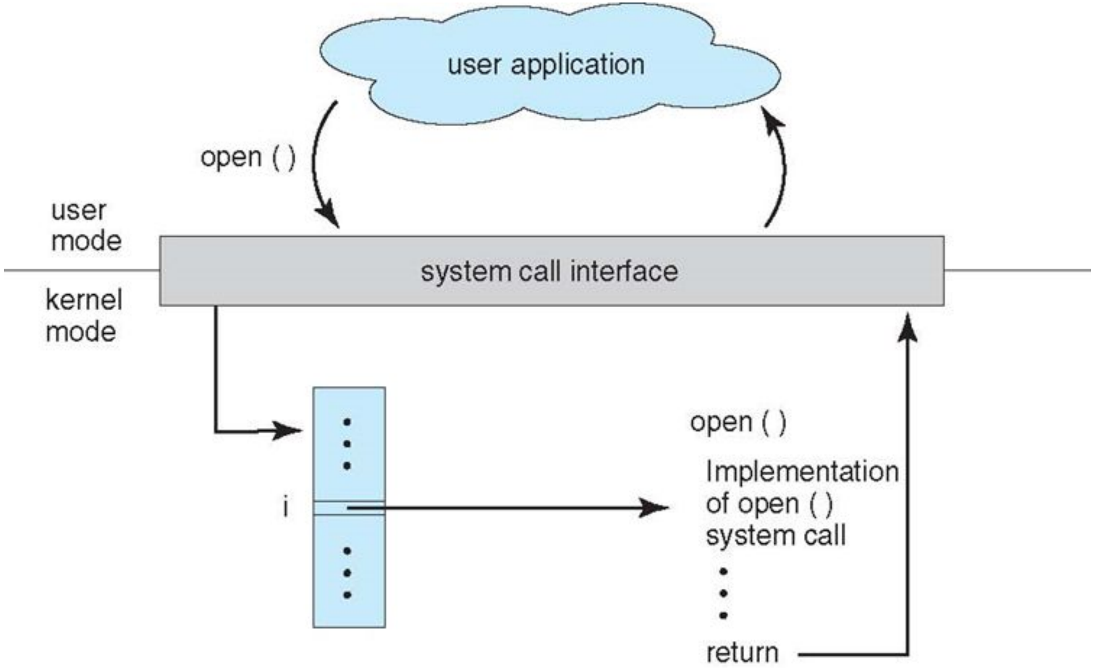


System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API

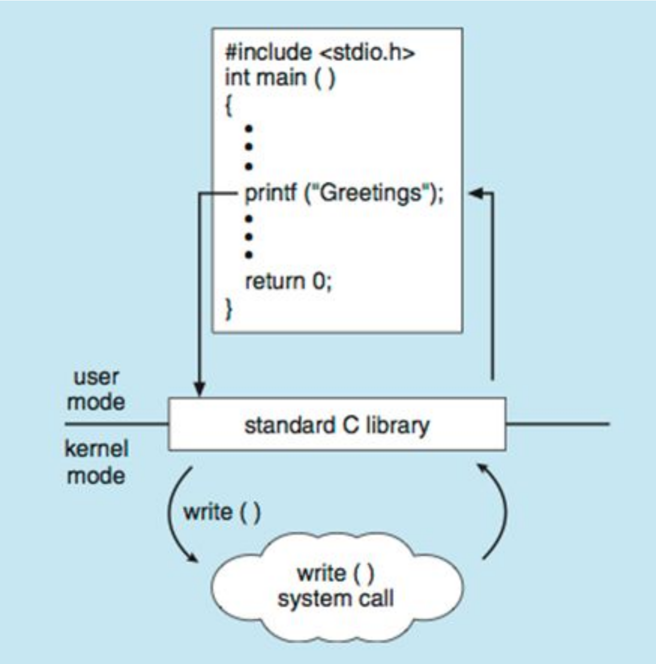
Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



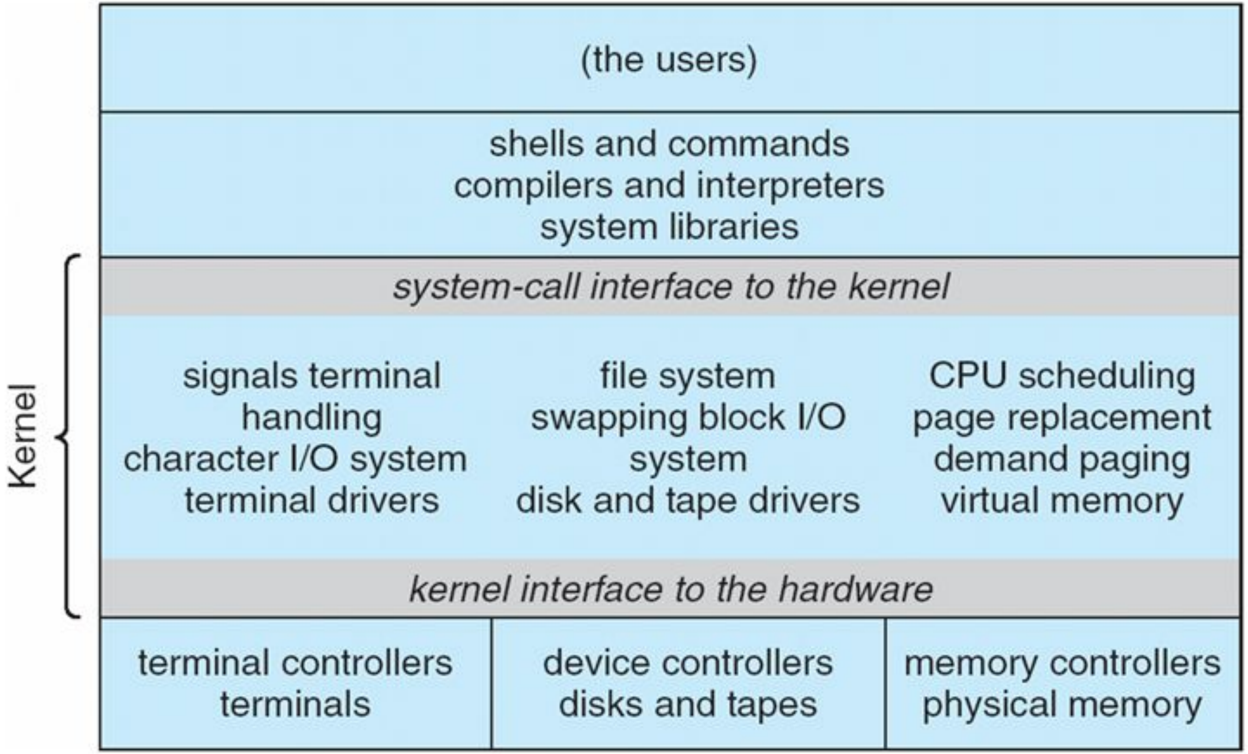
Standard C Library Example

C program invoking printf() library call, which calls write() system call



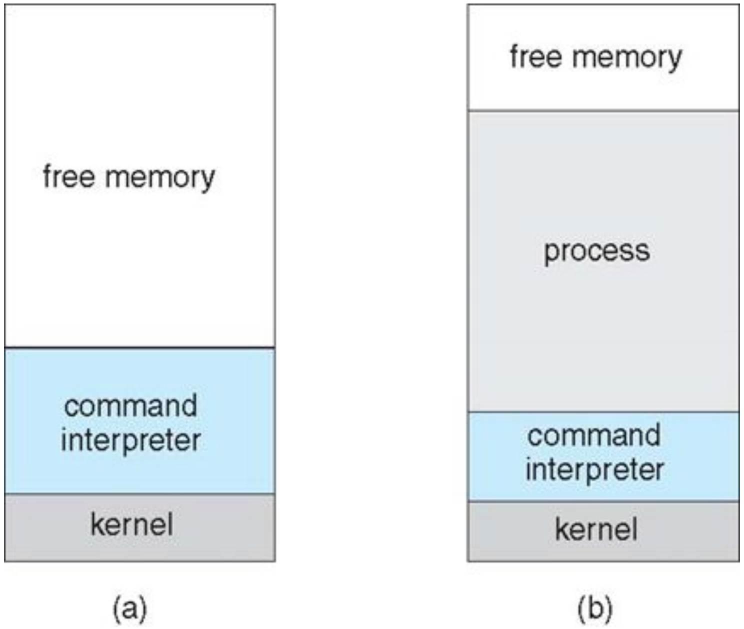
User mode vs. Kernel mode

Traditional UNIX System Structure

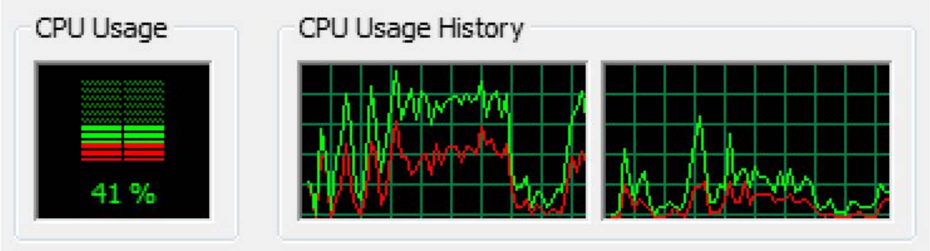


Example: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
  - Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



Understanding User and Kernel Mode

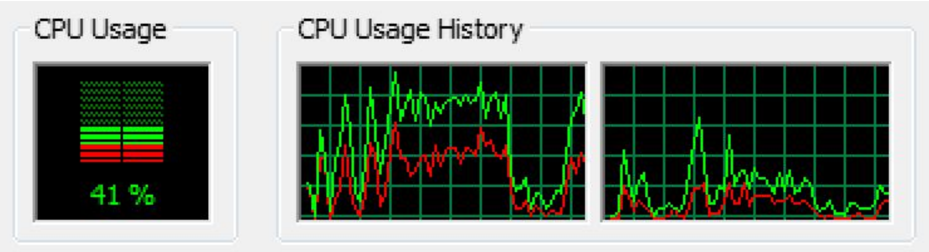


Understanding User and Kernel Mode

CPU usage is generally represented as a simple percentage of CPU time spent on non-idle tasks. But this is a bit of a simplification. In any modern operating system, the CPU is actually spending time in two very distinct modes:

1- Kernel Mode

2- User Mode



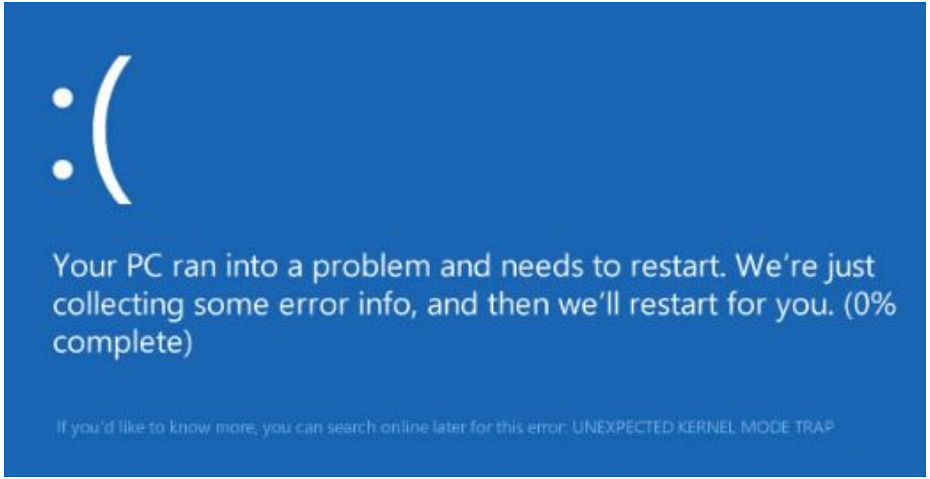
It's possible to enable display of Kernel time in Task Manager, as I have in the above screenshot. The green line is total CPU time; the red line is Kernel time. The gap between the two is User time.

Kernel Mode

- Executing code has complete and unrestricted access to the underlying hardware.
- It can execute any CPU instruction and reference any memory address.
- Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system.
- Crashes in kernel mode are catastrophic; they will halt the entire PC.

Blue Screen of Death (BSoD)

UNEXPECTED_KERNEL_MODE_TRAP (error code 0x0000007F) is one of the Windows 10 errors that often show Blue Screen of Death (BSoD) and cause PC crashes and freezes.



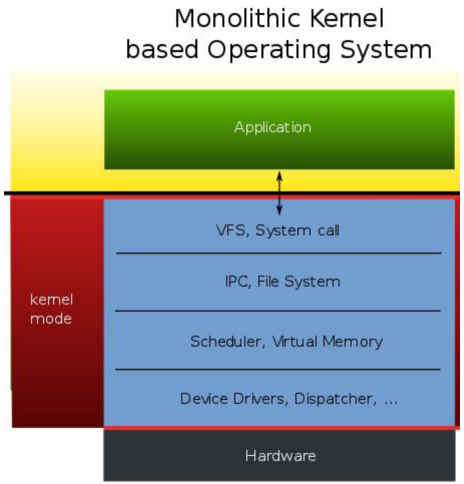
User Mode

- Executing code has no ability to directly access hardware or reference memory.
- Code running in user mode must delegate to system APIs to access hardware or memory.
- Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable.
- Most of the code running on your computer will execute in user mode.

Kernel design

Monolithic Kernel

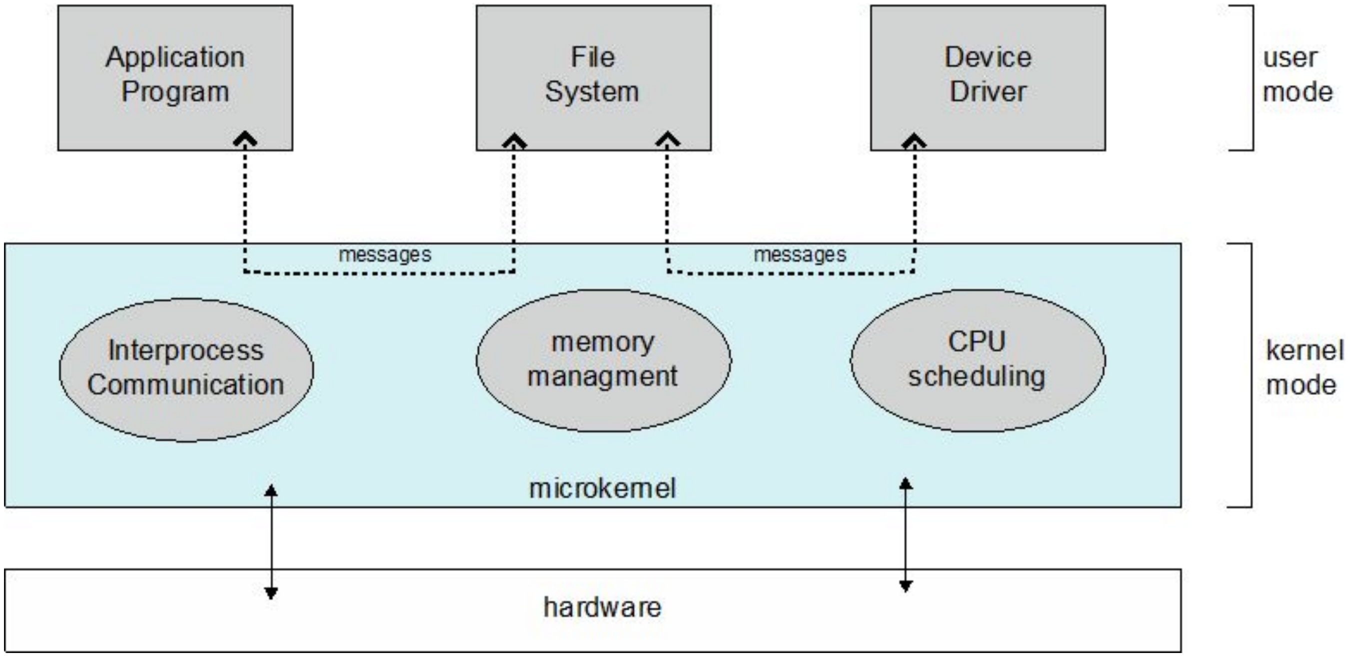
A monolithic kernel is an operating system architecture where the entire operating system is working in kernel space.

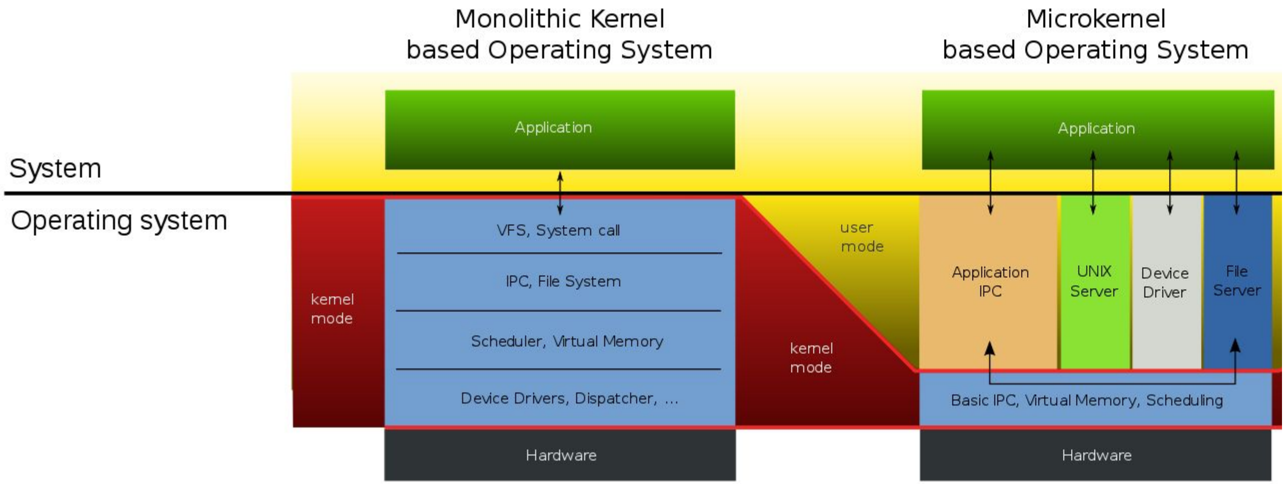


Microkernel System Structure

- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure

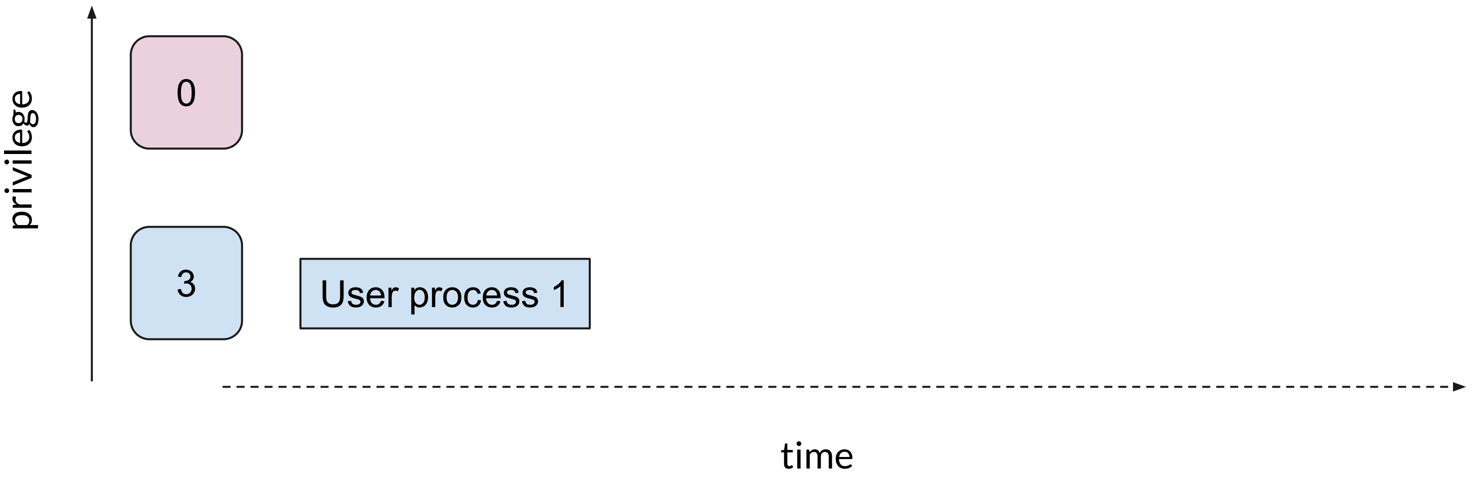



Monolithic Kernel
based Operating System
Microkernel
based Operating System
System
Operating system
Application
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
kernel
mode
user
mode
kernel
mode
Application
IPC
UNIX
Server
Device
Driver
File
Server
Basic IPC, Virtual Memory, Scheduling
Hardware
Hardware

Interrupts

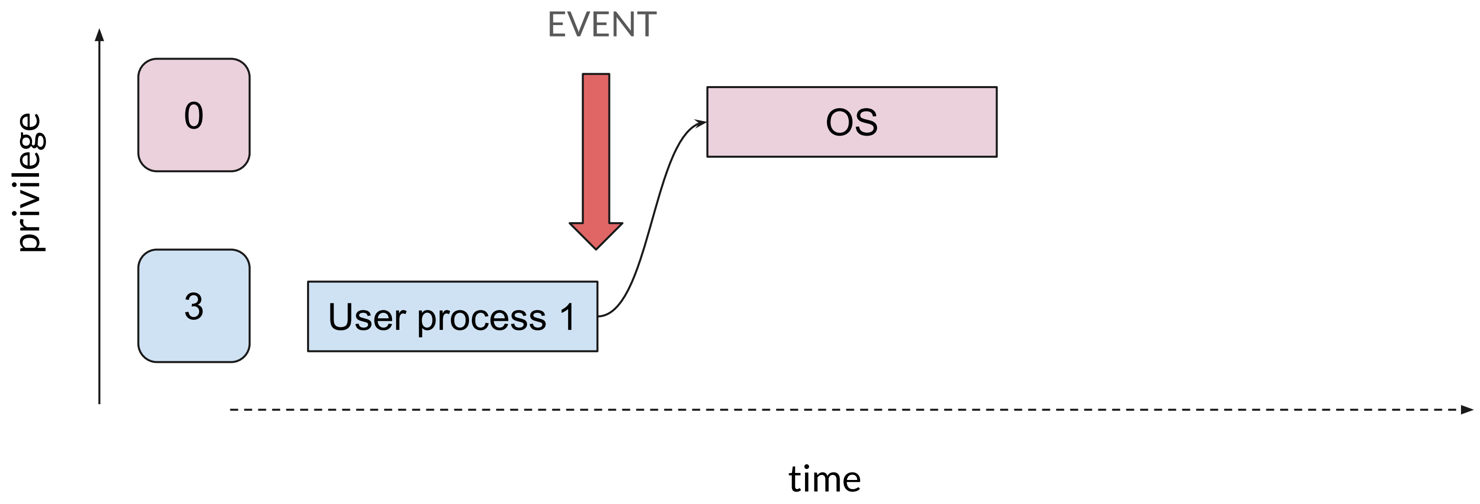
OS Events

- OS is event driven



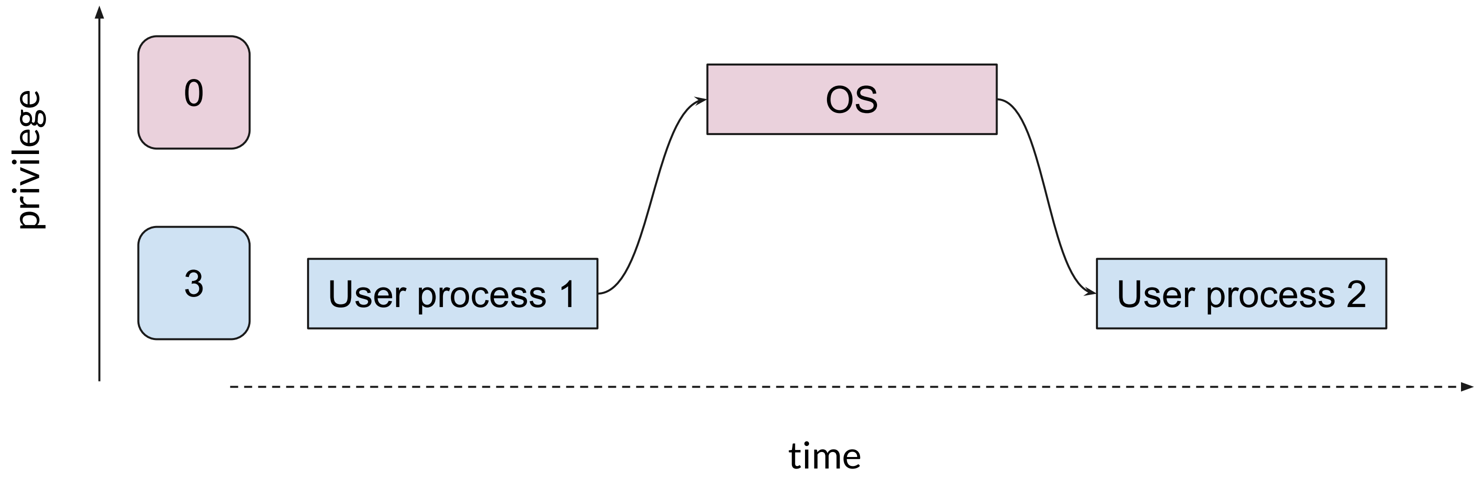
OS Events

- OS is event driven



OS Events

- OS is event driven



Events

Hardware interrupts:

- Raised by hardware devices (keyboard, mouse, disk, ...)
- Asynchronous (may occur at anytime)

Trap:

- Software interrupts
- Raised by user programs to invoke an OS functionality

Exceptions:

- Generated automatically by the processor itself as a result of an illegal instruction.
- Faults: recoverable errors (can you give me an example of a fault?)

Events

Hardware interrupts:

- Raised by hardware devices (keyboard, mouse, disk, ...)
- Asynchronous (may occur at anytime)

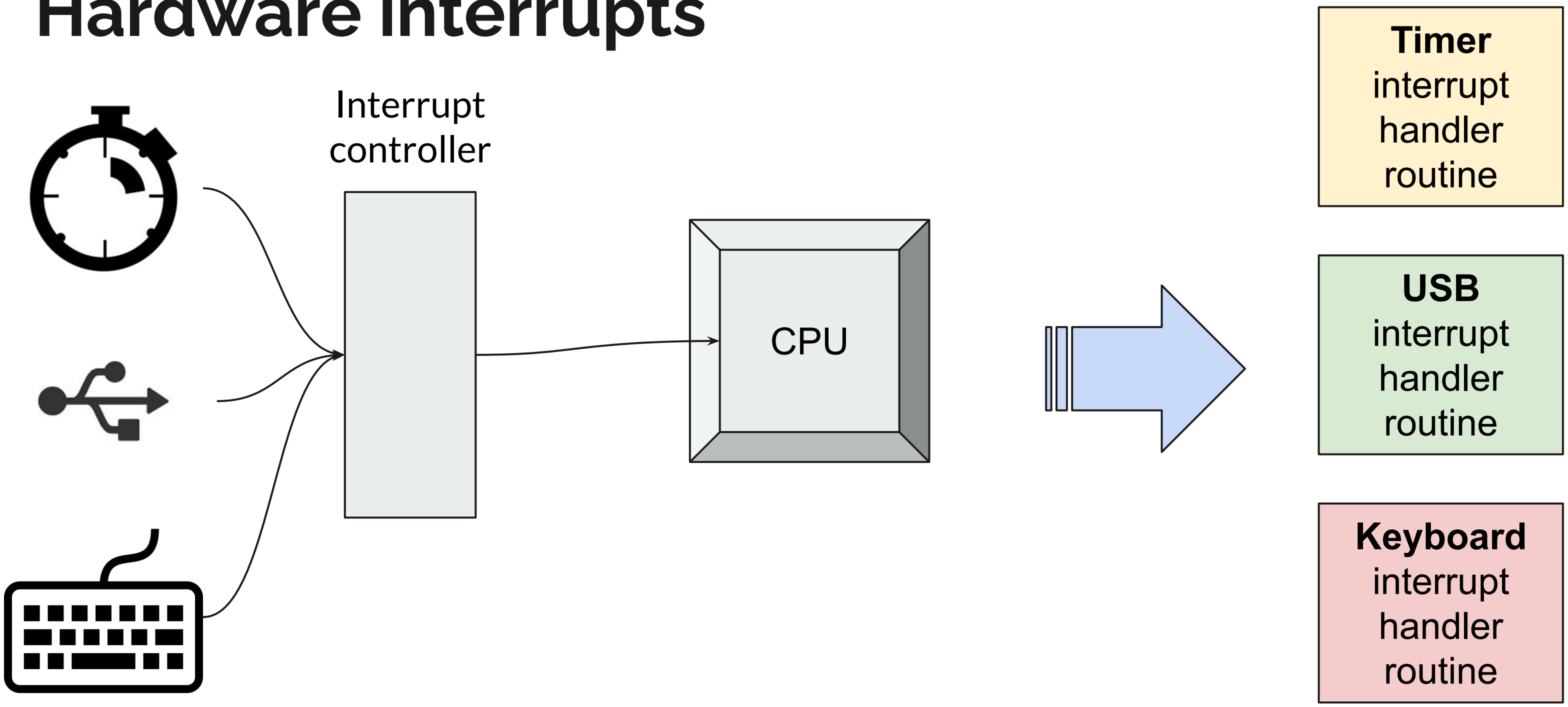
Trap:

- Software interrupts
- Raised by user programs to invoke an OS functionality

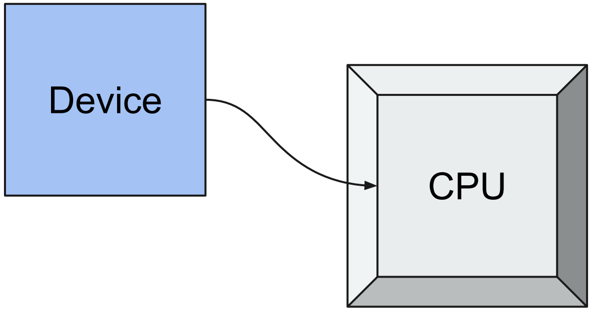
Exceptions:

- Generated automatically by the processor itself as a result of an illegal instruction.
- Faults: recoverable errors (page fault in memory management)
- Aborts: difficult to recover (such as divide by 0)

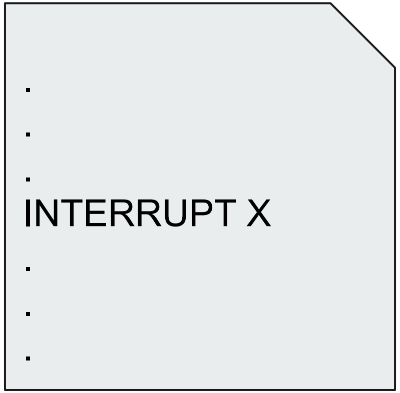
Hardware interrupts



Hardware interrupts vs Software interrupts

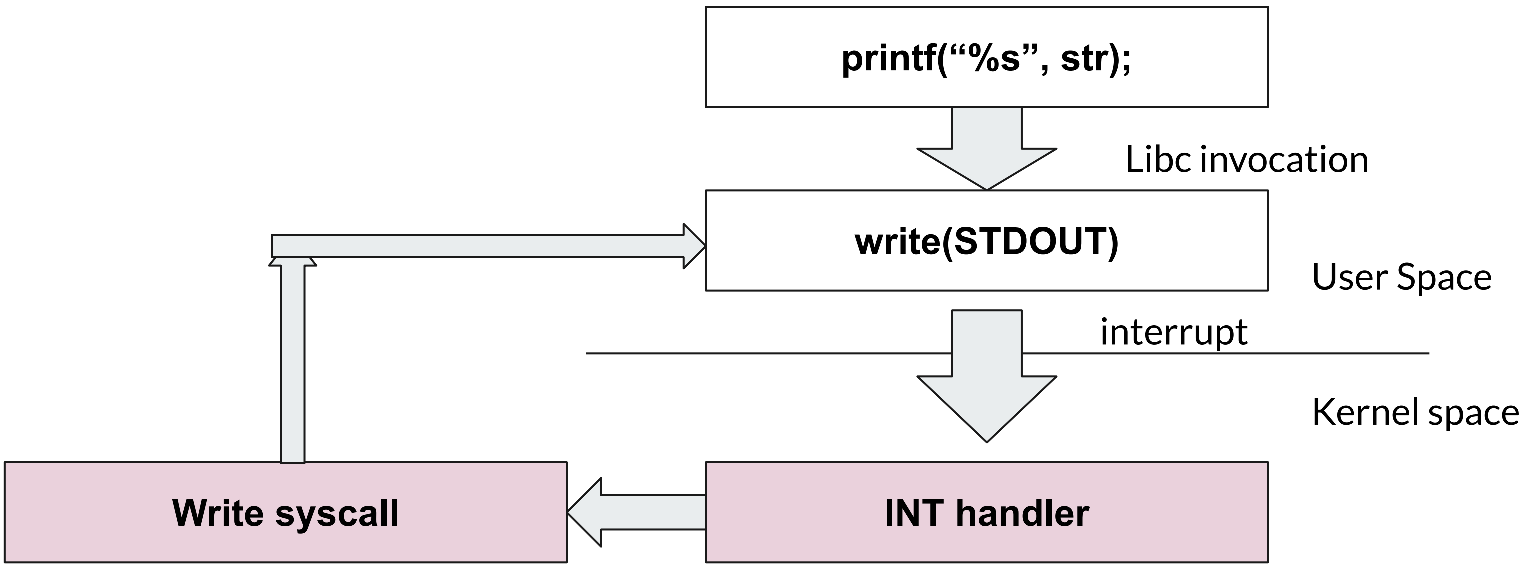


A device asserts a pin in the CPU (through PIC)



An instruction causes an interrupt

Software interrupt



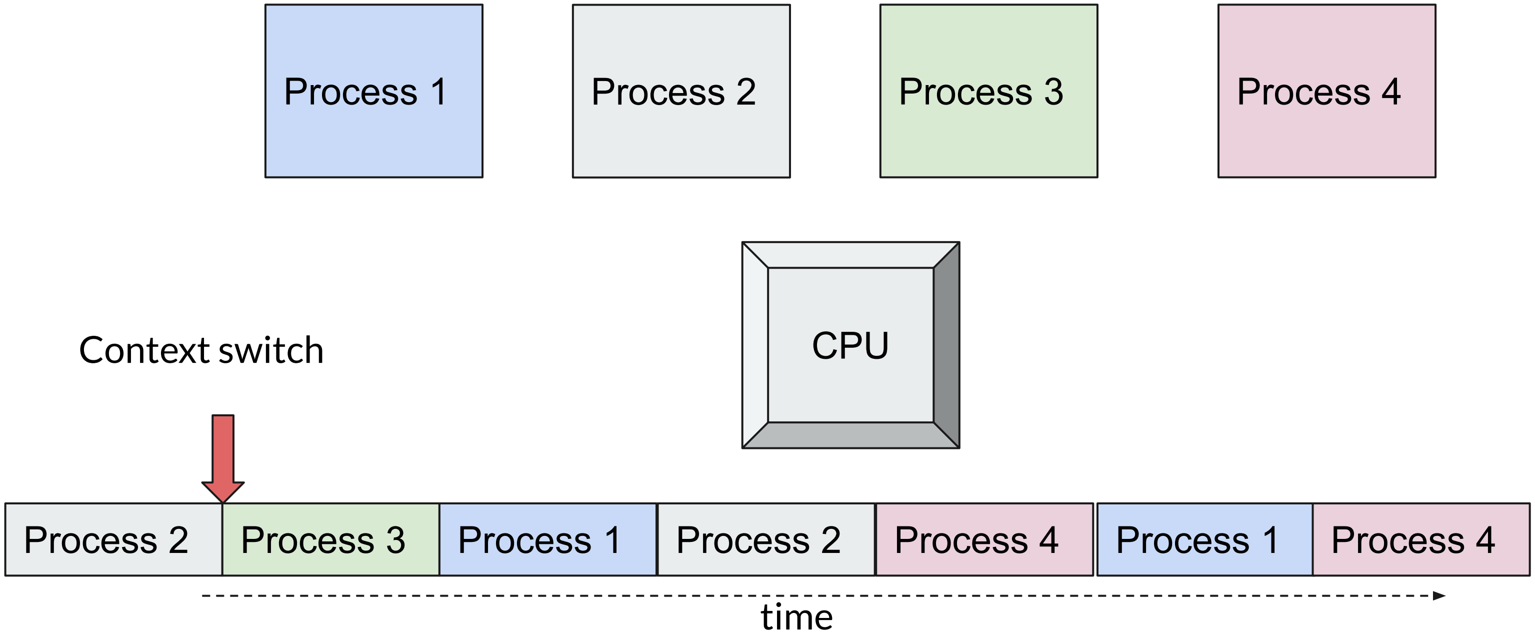
System calls in Xv6

https://github.com/nalmadi/xv6-public/blob/master/syscall.c

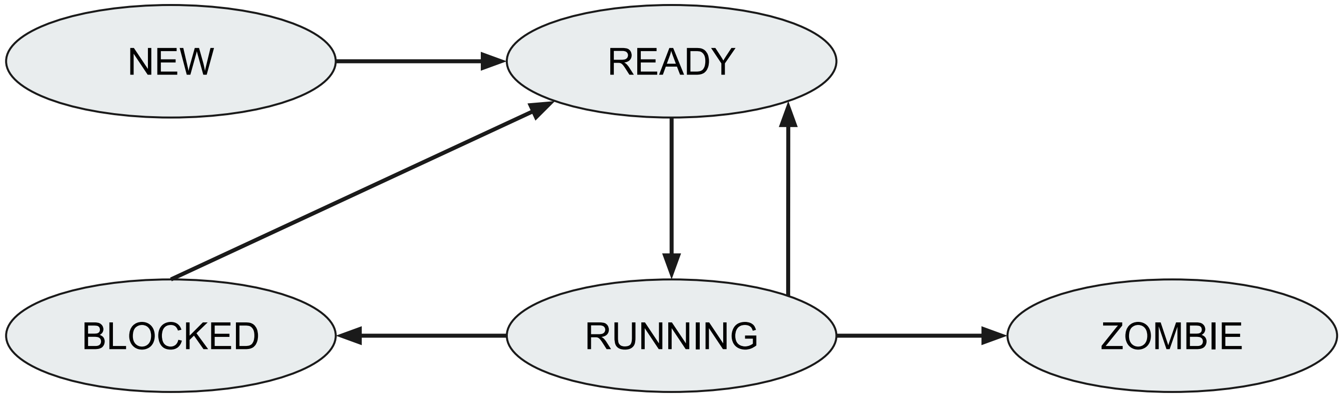
```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};
```

Scheduling

Context switching

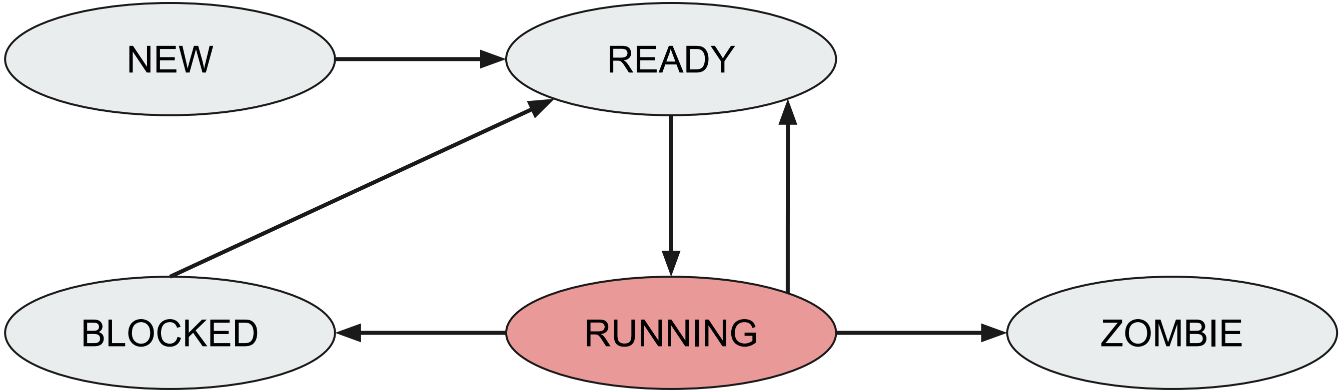


Process lifecycle



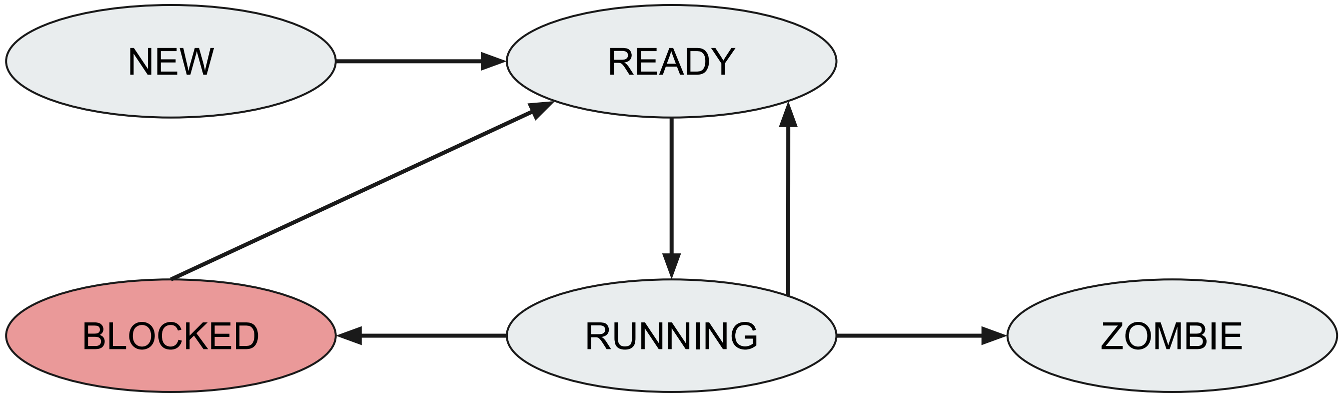
Process lifecycle

```
int main(){
   char str[10];
   scanf("%s", str);
}
```



Process lifecycle

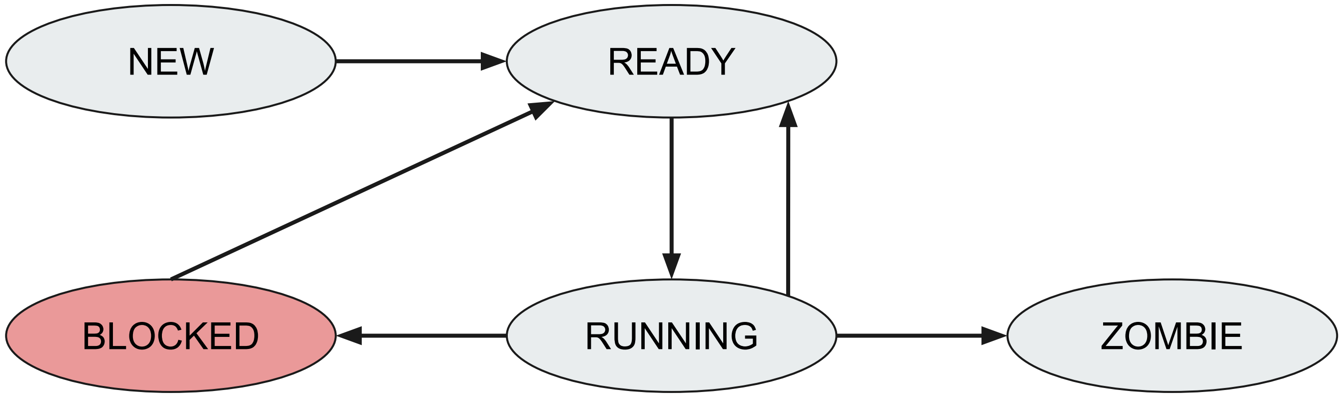
```
int main(){
    char str[10];
    scanf("%s", str);
}
```



Process lifecycle

```
int main(){
    char str[10];
    scanf("%s", str);
}
```

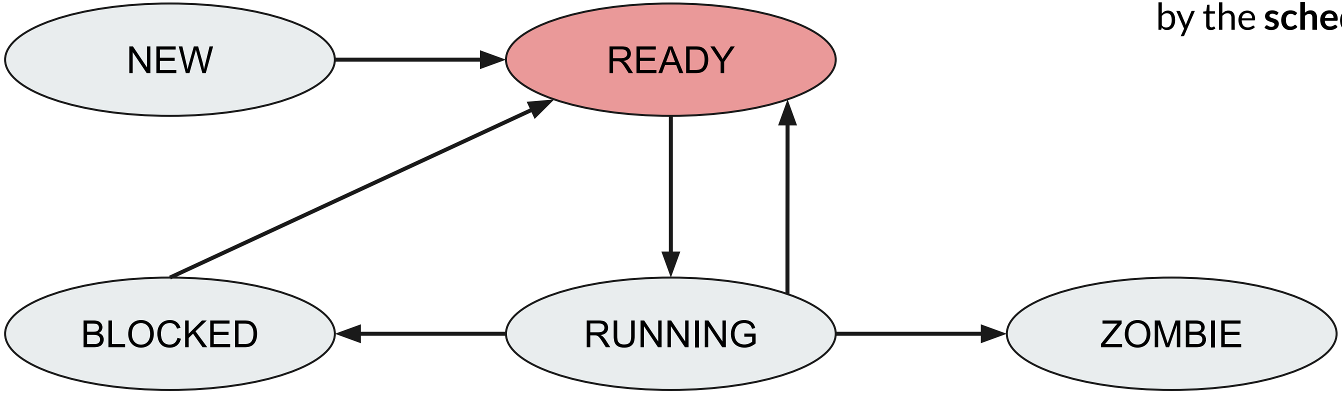
User enters some string



Process lifecycle

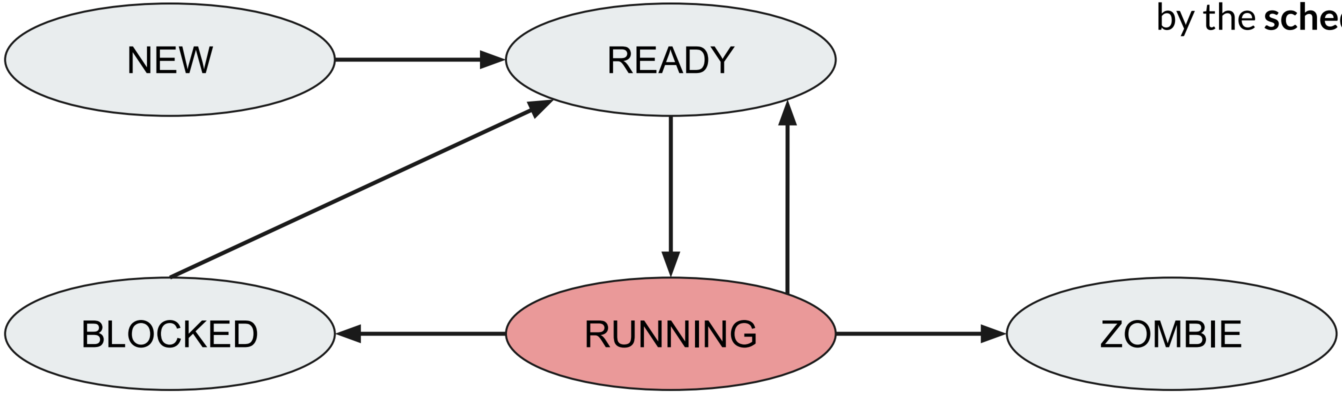
```
int main(){
   char str[10];
   scanf("%s", str);
}
```

After a while the process is selected by the **scheduler**



Process lifecycle

```
int main(){
    char str[10];
    scanf(“%s”, str);
}
```

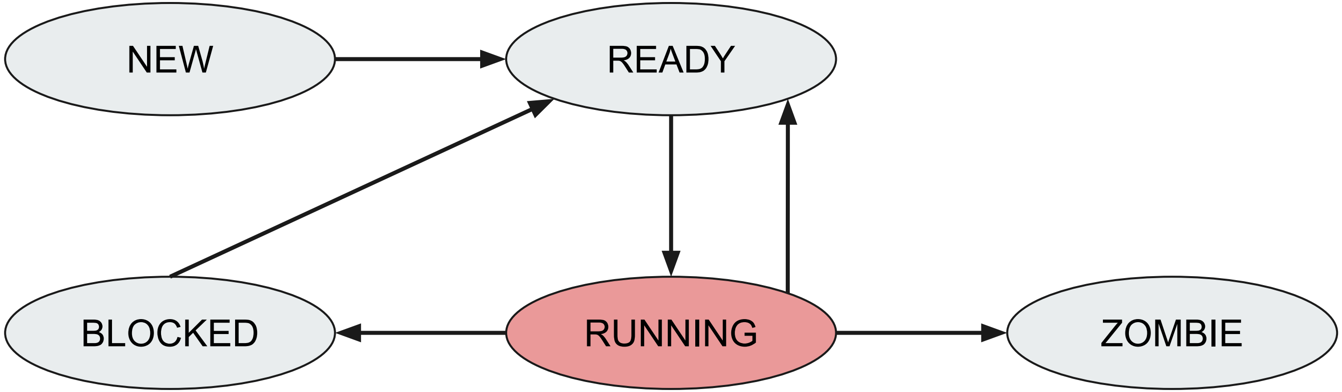


After a while the process is selected by the **scheduler**

Process lifecycle

```
int main(){
    char str[10];
    scanf("%s", str);
}
```

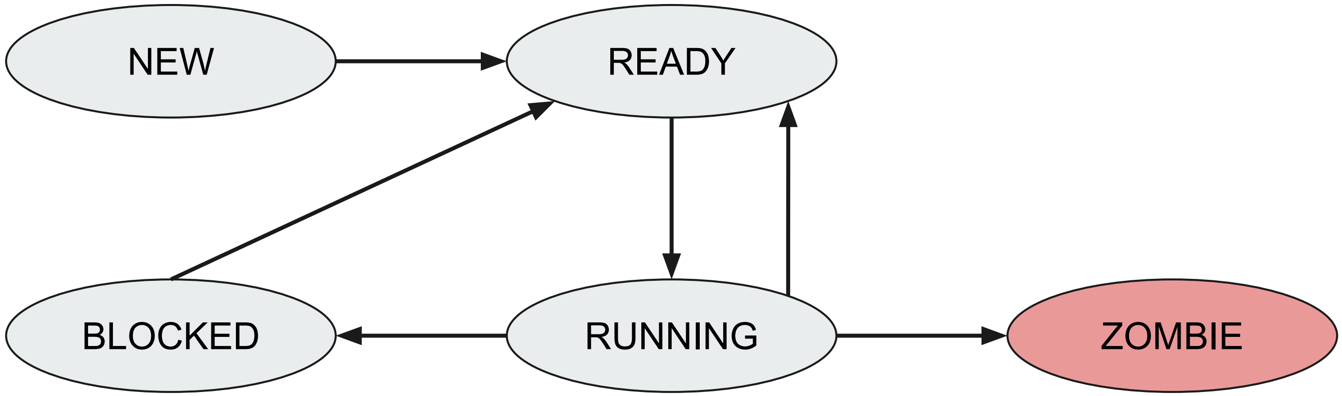
Process ends



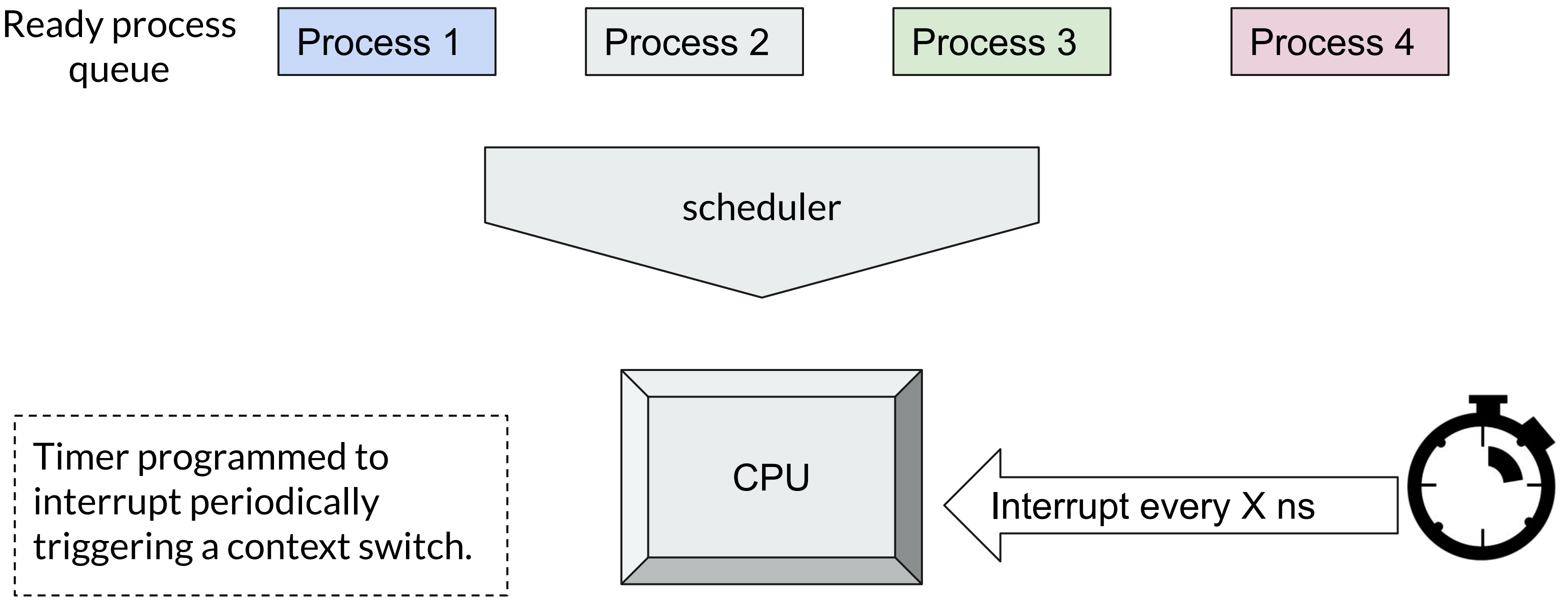
Process lifecycle

```
int main(){
    char str[10];
    scanf("%s", str);
}
```

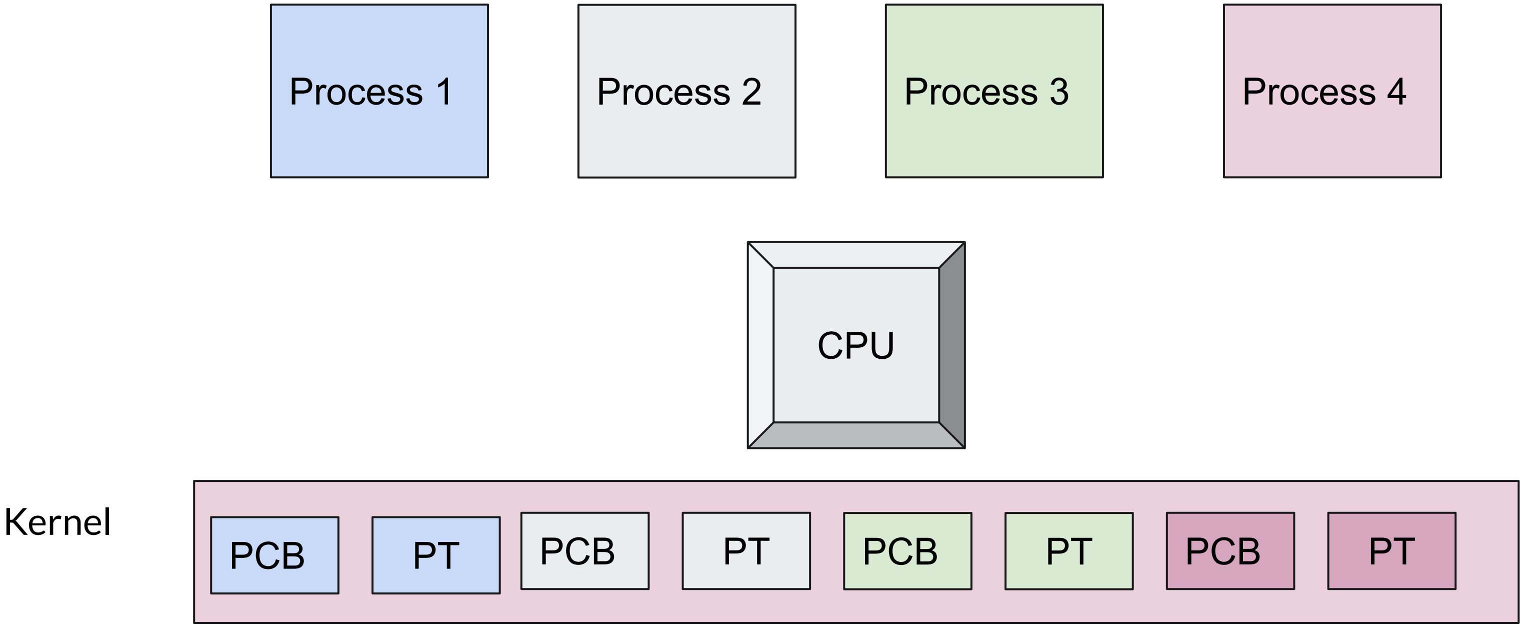
Process ends



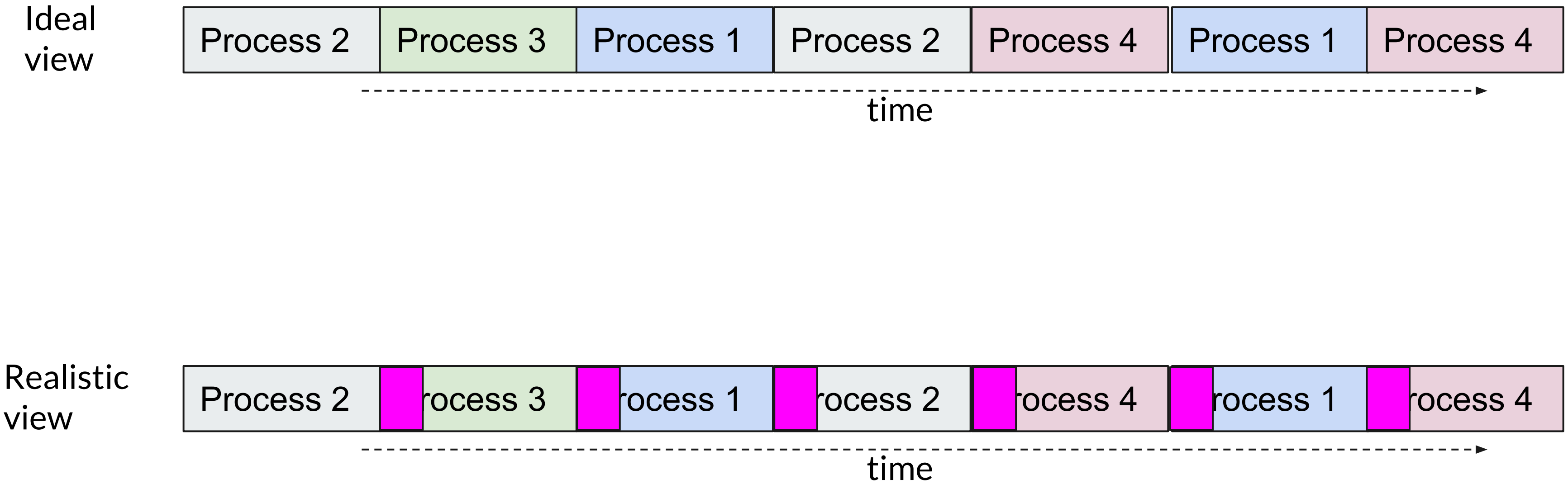
Timer interrupt



Context switch overhead



Context switch overheads

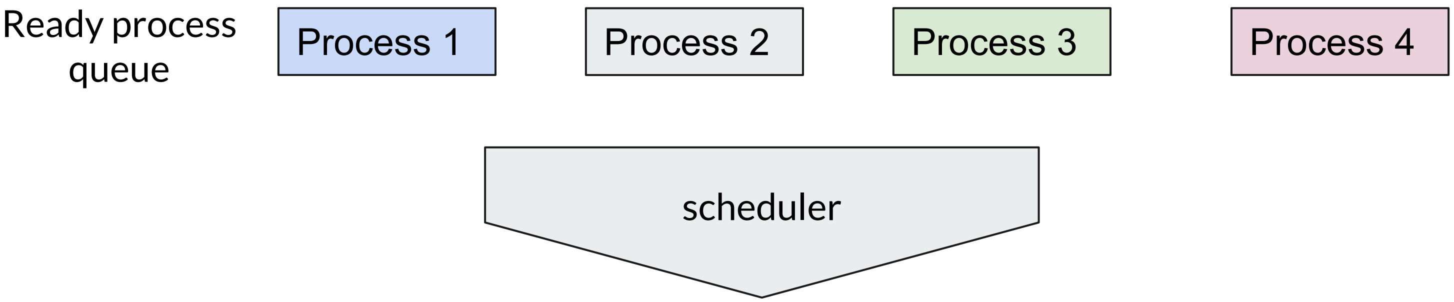


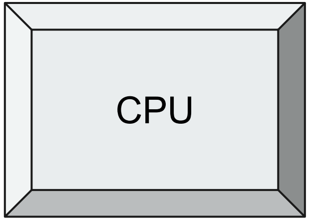
Context switch overheads

Direct factors affecting context switching time:

- Timer interrupt latency
- Saving/restoring contexts
- Finding the next process to execute (scheduling algorithm)

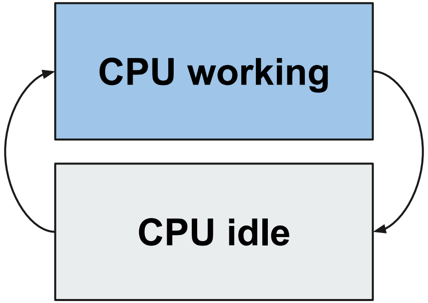
Scheduler

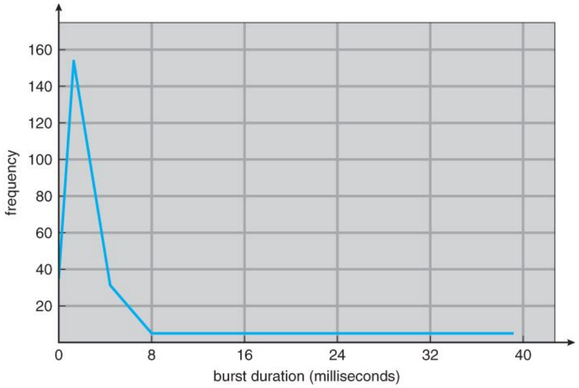


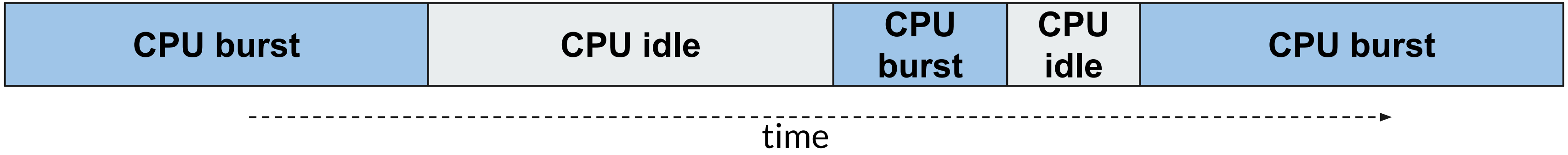


The scheduler is responsible for selecting which process should use CPU next.

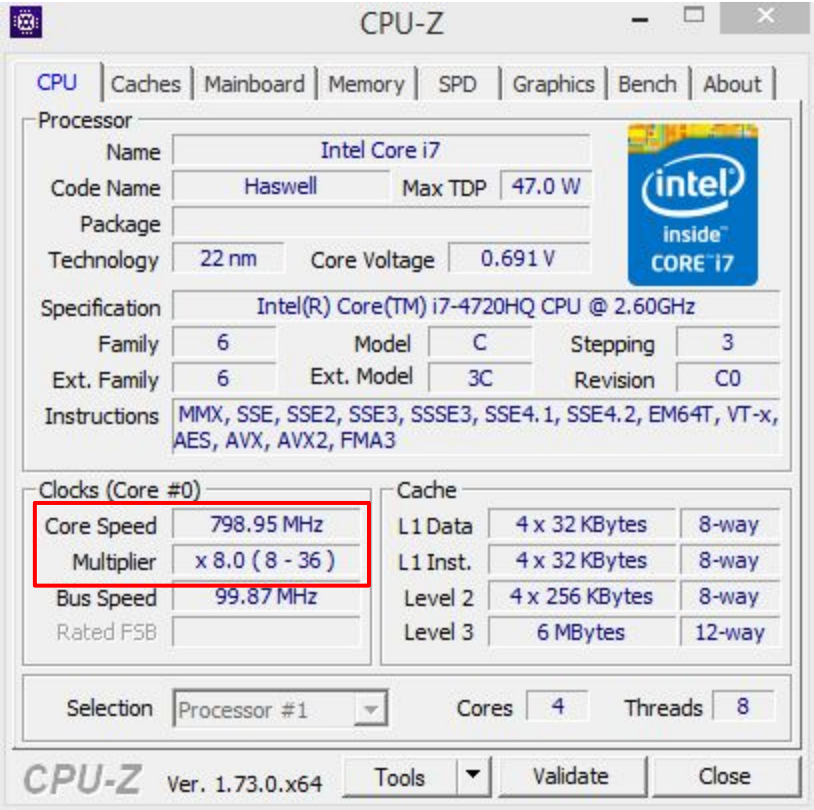
Execution phases of a process



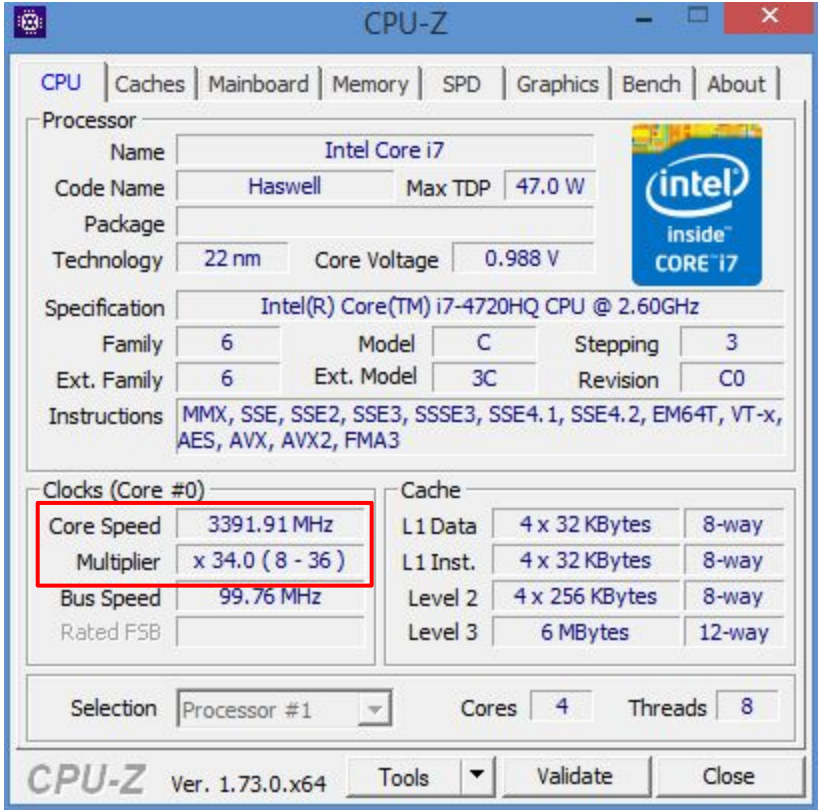




Idle


CPU-Z
CPU
Caches
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
Intel Core i7
Code Name
Haswell
Max TDP
47.0 W
Package
Technology
22 nm
Core Voltage
0.691 V
intel inside CORE i7
Specification
Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz
Family
6
Model
C
Stepping
3
Ext. Family
6
Ext. Model
3C
Revision
C0
Instructions
MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3
Clocks (Core #0)
Core Speed
798.95 MHz
Multiplier
x 8.0 (8 - 36)
Bus Speed
99.87 MHz
Rated FSB
Cache
L1 Data
4 x 32 KBytes
8-way
L1 Inst.
4 x 32 KBytes
8-way
Level 2
4 x 256 KBytes
8-way
Level 3
6 MBytes
12-way
Selection
Processor #1
Cores
4
Threads
8
CPU-Z
Ver. 1.73.0.x64
Tools
Validate
Close

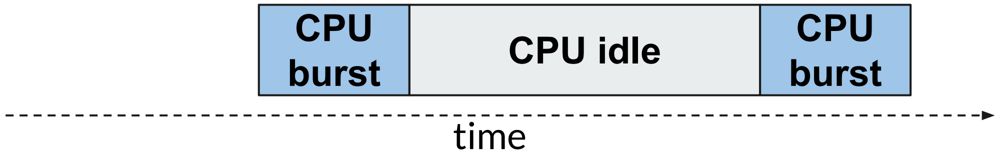
Active


CPU-Z
CPU
Caches
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
Intel Core i7
Code Name
Haswell
Max TDP
47.0 W
Package
Technology
22 nm
Core Voltage
0.988 V
intel inside CORE i7
Specification
Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz
Family
6
Model
C
Stepping
3
Ext. Family
6
Ext. Model
3C
Revision
C0
Instructions
MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3
Clocks (Core #0)
Core Speed
3391.91 MHz
Multiplier
x 34.0 (8 - 36)
Bus Speed
99.76 MHz
Rated FSB
Cache
L1 Data
4 x 32 KBytes
8-way
L1 Inst.
4 x 32 KBytes
8-way
Level 2
4 x 256 KBytes
8-way
Level 3
6 MBytes
12-way
Selection
Processor #1
Cores
4
Threads
8
CPU-Z
Ver. 1.73.0.x64
Tools
Validate
Close

Types of processes

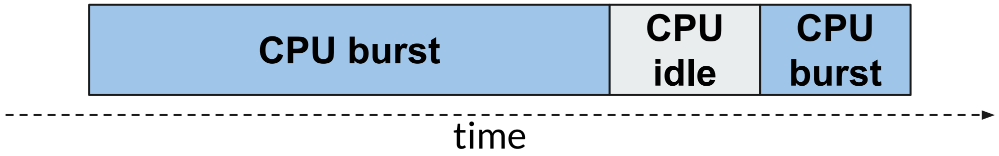
I/O bound:

- Has small bursts of CPU activity and then waits for I/O
- Eg. word processor



CPU bound:

- Hardly any I/O mostly CPU activity (scientific modeling, 3d rendering, etc)



Scheduling algorithms

Scheduling criteria

- Maximize CPU **utilization**: lowering Idle time
- Maximize **throughput**: number of processes completed per unit time
- Minimize **turnaround time**: the total amount of time spent by the process from coming in the ready state for the first time to its completion.
- Minimize **response time**: the time spent between the ready state and getting the CPU for the first time.
- Minimize **waiting time**: the total time spent by the process in the ready state waiting for CPU.
- **Fairness**: assigning CPU time to jobs such that all jobs get, on average, an equal share of resources over time.

Types of CPU Schedulers

CPU scheduler (dispatcher or short-term scheduler) selects a process from the ready queue and lets it run on the CPU

Types:

- **Non-preemptive**: simple to implement but unsuitable for time-sharing systems.
- **Preemptive** (a timer interrupt occurs): more overhead, but keeps long processes from monopolizing CPU.

Predicting the Length of CPU Burst

- some schedulers need to know CPU burst size.
- cannot know deterministically, but can estimate on the basis of previous bursts (Likelihood principle).

Initially, we will make an unrealistic assumption:

The run-time of each job is known.

Scheduling Policy: System and User Oriented

system oriented:

- maximize CPU utilization – scheduler needs to keep CPU as busy as possible. Mainly, the CPU should not be idle if there are processes ready to run
- maximize throughput – number of processes completed per unit time
- ensure fairness of CPU allocation
- should avoid **starvation** – process is never scheduled
- minimize overhead – incurred due to scheduling
    - in time or CPU utilization (e.g. due to context switches)
    - in space (e.g. data structures)

Scheduling Policy: System and User Oriented

User-oriented:

- minimize turnaround time – interval from time process becomes ready till the time it is done
- minimize average and worst-case waiting time – sum of periods spent waiting in the ready queue
- minimize average and worst-case response time – time from process entering the ready queue till it is first scheduled

CPU Scheduling Algorithms

Non-Preemptive:

- First Come First Serve (FCFS)
- Shortest Job First (SJF)
- Priority (P)

Preemptive:

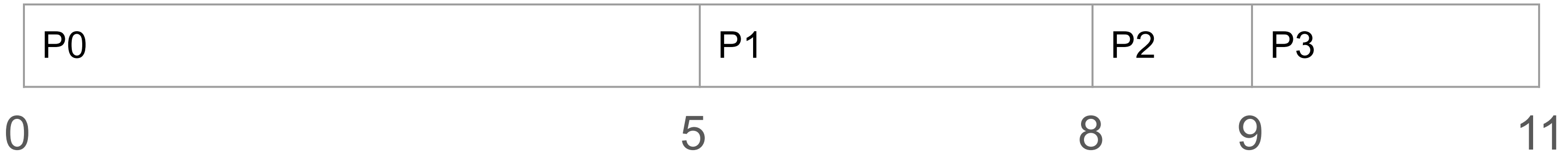
- Round Robin (RR)
- Preemptive Shortest Job First (PSJF)
- Preemptive Priority (PP)

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

First Come First Serve (FCFS)

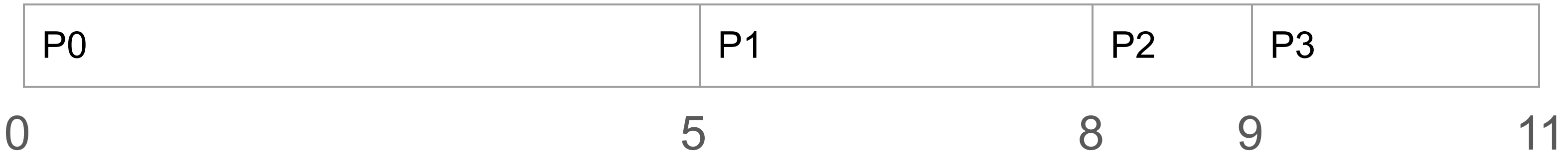
Process	CPU time
P0	5
P1	3
P2	1
P3	2



First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

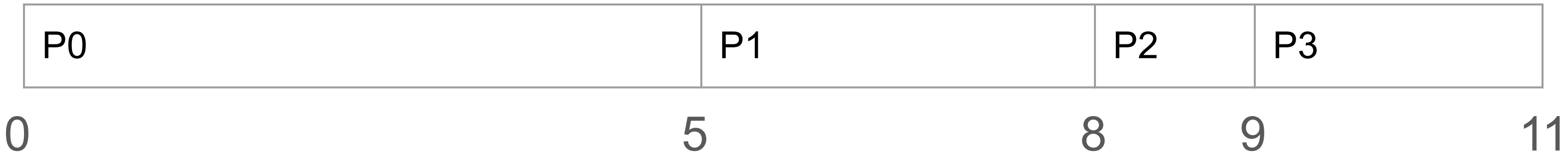
Waiting time – amount of time a process has been waiting in the ready queue (want min waiting time)



First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

Waiting time – amount of time a process has been waiting in the ready queue



First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

	Wait-time	Turnaround-time
P0	0	
P1		
P2		
P3		

P0	P1	P2	P3

0 5 8 9 11

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

	Wait-time	Turnaround-time
P0	0	
P1	5	
P2		
P3		

P0	P1	P2	P3

0 5 8 9 11

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

	Wait-time	Turnaround-time
P0	0	
P1	5	
P2	8	
P3		

P0	P1	P2	P3

0 5 8 9 11

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

	Wait-time	Turnaround-time
P0	0	
P1	5	
P2	8	
P3	9	

P0	P1	P2	P3

0 5 8 9 11

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

Turnaround time – amount of time to execute a particular process (FINISH)

P0	P1	P2	P3

0 5 8 9 11

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

	Wait-time	Turnaround-time
P0	0	5
P1	5	
P2	8	
P3	9	

P0	P1	P2	P3

0 5 8 9 11

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

	Wait-time	Turnaround-time
P0	0	5
P1	5	8
P2	8	
P3	9	

P0	P1	P2	P3

0 5 8 9 11

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

	Wait-time	Turnaround-time
P0	0	5
P1	5	8
P2	8	9
P3	9	

P0	P1	P2	P3

0 5 8 9 11

First Come First Serve (FCFS)

Process	CPU time
P0	5
P1	3
P2	1
P3	2

	Wait-time	Turnaround-time
P0	0	5
P1	5	8
P2	8	9
P3	9	11

P0	P1	P2	P3

0 5 8 9 11

FCFS Evaluation

	Wait-time	**Turnaround-time**
P0	0	5
P1	5	8
P2	8	9
P3	9	11
avg	5.5	8.25

- Non-preemptive
- response time?

FCFS Evaluation

	Wait-time	Turnaround-time
P0	0	5
P1	5	8
P2	8	9
P3	9	11
avg	5.5	8.25

- Non-preemptive
- response time — may have variance or be long
- What about fairness?

References:

Operating System Concepts, 9th Edition – Silberschatz, Galvin, and Gagne

Understanding Operating Systems, 8th edition - Ann McHoes and Ida M. Flynn

https://blog.codinghorror.com/understanding-user-and-kernel-mode/