Lecture 2

Jan 4, 2024

Trivia Time

Which of the following are likely components of an operating system ? Write all that apply

A) File editor

B) File System

C) Device Driver

D) Cache Memory

E) Web Browser

F) Scheduler

Processes and Process Management

What is a **Process**?

How are **processes represented by OS**?

How are **multiple concurrent processes are managed by OS**? [Particularly when multiple physical processes share a **single physical platform**]

Process abstraction

When you run an exe file, the OS creates a process = a running program

- OS timeshares CPU across multiple processes: virtualizes CPU
- OS has a CPU scheduler that picks one of the many active processes to execute on a CPU

– Policy: which process to run

– Mechanism: how to “context switch” between processes

Process Definition [From Program to Processes]

Instance of an executing program [Synonymous with "task" or "Job"]

Recall : OS manages H/W on behalf of applications

Application = Program on a disk/flash memory/cloud.. **[Static Entity]**

Process = State of the program when executing [Gets loaded in memory & starts execution] **[Active Entity] (Program in Execution)**

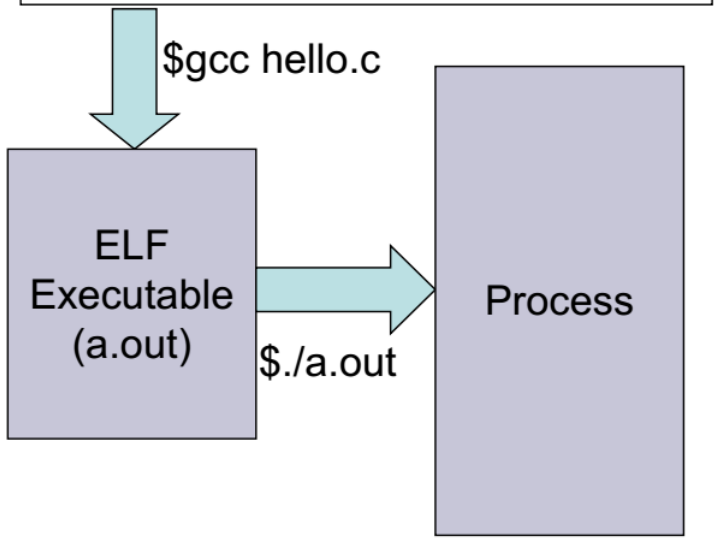
Same program launched more than once => Multiple processes will get created => But each one of them may be in a different state {??}

Program	Process
code + static and global data	Dynamic instantiation of code + data + heap + stack + process state
One program can create several processes	A process is unique isolated entity

What does a Process Contain?

```c
#include <stdio.h>

int main(){
  char str[] = "Hello World\n";
  printf("%s", str);
}
```



- Process
  - A program in execution
  - Most important abstraction in an OS
  - Comprises of — A unique identifier (PID)
    - Code
    - Data
    - Stack
    - Heap
    - State in the OS
    - Kernel stack

  From executable: Code, Data — from ELF

  During execution: Stack, Heap — In the user space of process

  State in the OS, Kernel stack — In the kernel space

  - State contains : registers, list of open files, related processes, etc.

CPU context – Program counter, current operands, Stack pointer

What does a Process Contain?

Code – Instructions from compiled executable

Data – Static global data from executable

Stack – Local variables, Function arguments, Function return address

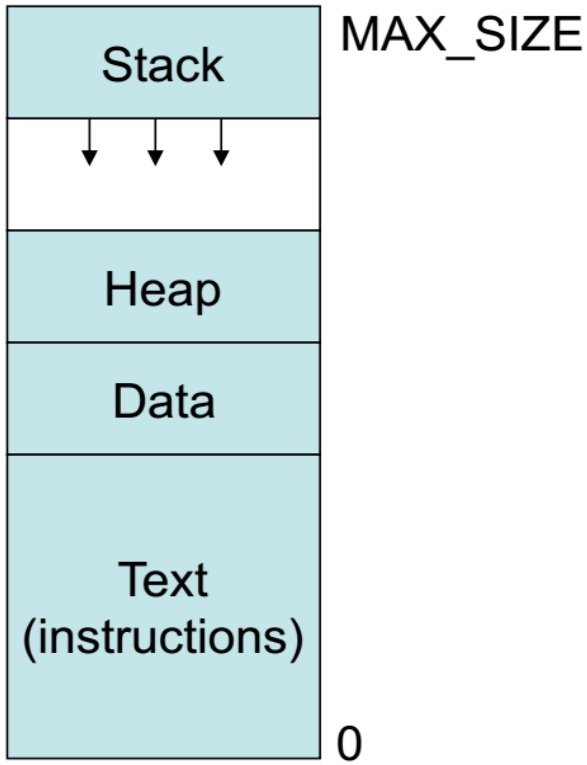
Heap – Variables allocated by malloc()

Machine State – CPU registers, PC, SP

Kernel Stack – Open files, process trees

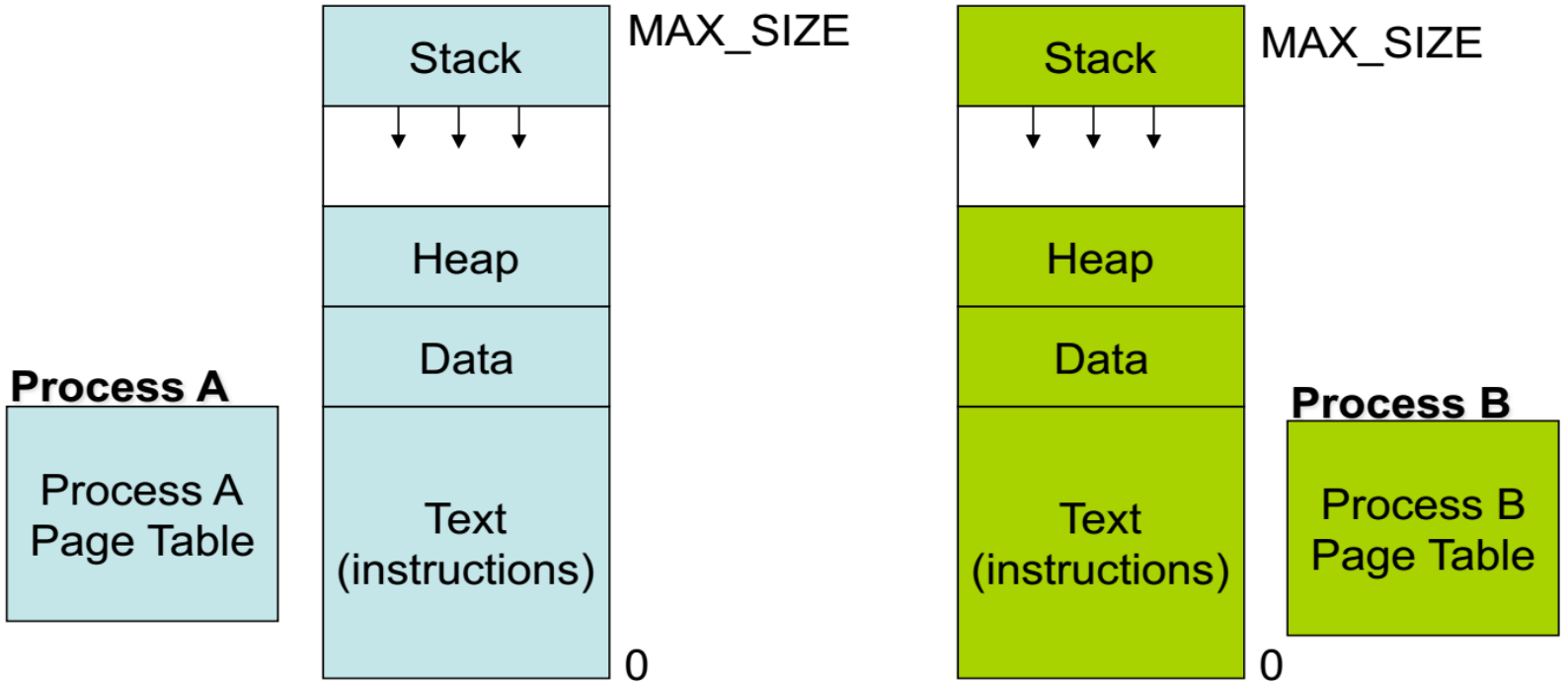
Process Memory Image (Address Space)

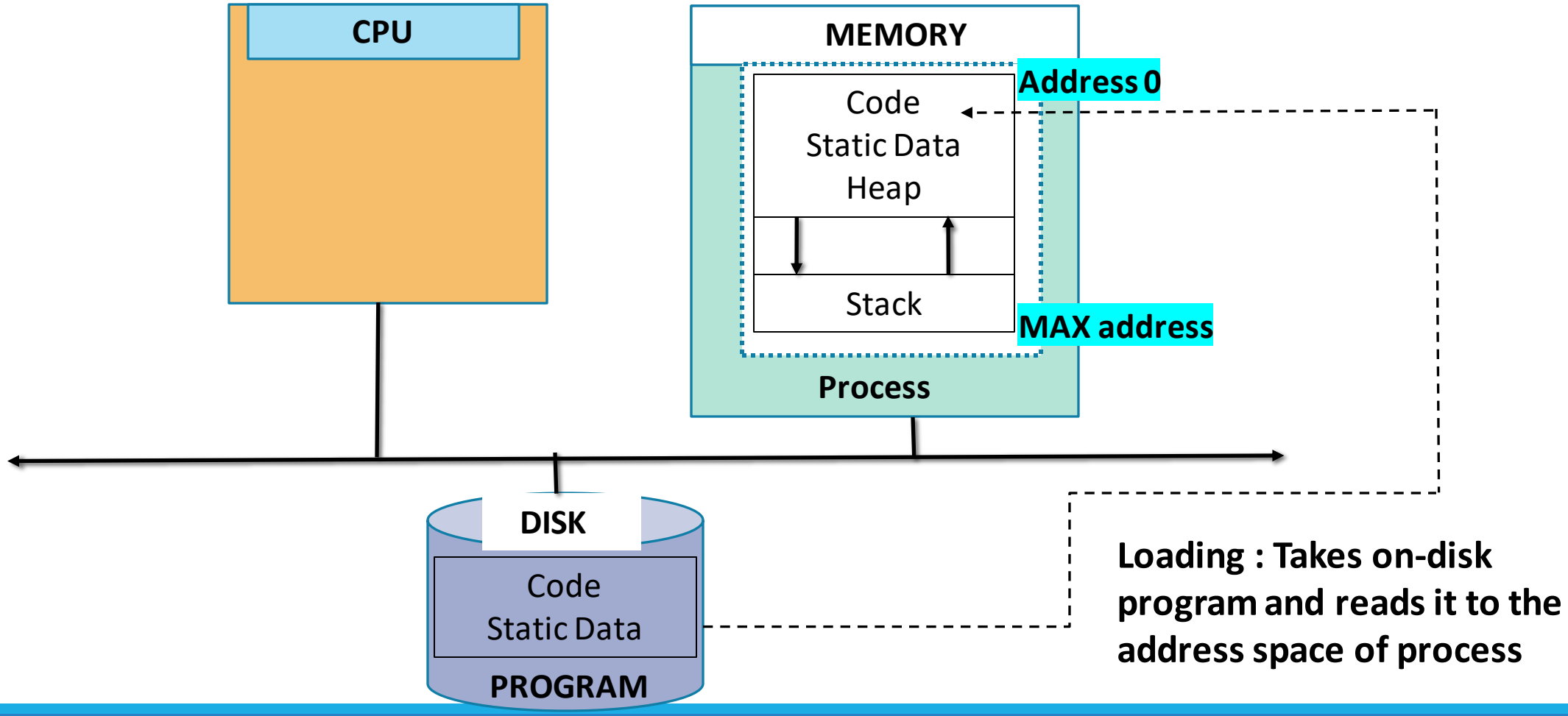
- Virtual Address Map
  - All memory a process can address
  - Large contiguous array of addresses from 0 to MAX_SIZE



Process Address Space

- Each process has a different address space
- This is achieved by the use of virtual memory
- Ie. 0 to MAX_SIZE are virtual memory addresses




CPU
MEMORY
Address 0
Code
Static Data
Heap
Stack
MAX address
Process
DISK
Code
Static Data
PROGRAM
Loading : Takes on-disk program and reads it to the address space of process

What should the OS do?

Allocates Memory and Creates Memory Image

Load executable from disk to main memory

Setup Virtual Memory

- Allocate memory to be used as stack
- Fill in the arguments (eg in main(…))
- Allocate memory to be used as heap

Files

- Open file descriptions
- Standard files – Stdin, stdout, stderr

Set PC, Jump

How does OS create a process?

Allocates memory and creates memory image

– Loads code, static data from disk [exe] in to memory [Addr. Space]

– Creates runtime stack[local var, func.paramaters, ret.Addr], heap[malloc]

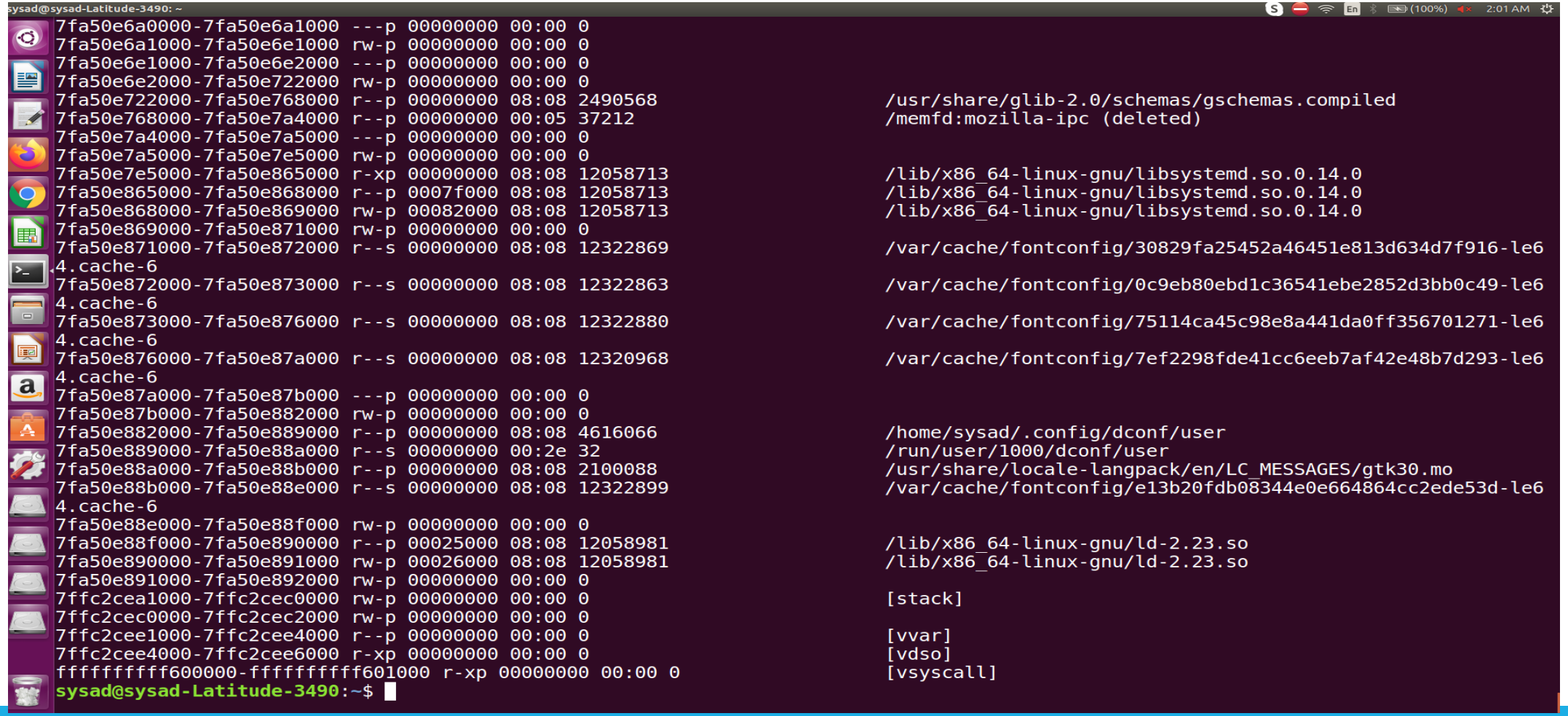
Opens basic files [Initializes basic I/O]

– STD IN, OUT, ERR [Three open file descriptors] (read from terminal, print output to screen]

Initializes CPU registers [main() - entry point]

– PC points to first instruction

On Linux : /proc/pid/maps



Why Multiple Entries ??

What are these ??

On Linux : /proc/pid/maps

Each row in /proc/$PID/maps describes a region of contiguous virtual memory in a process or thread. Each row has the following fields:

---

address - This is the starting and ending address of the region in the process's address space

permissions - This describes how pages in the region can be accessed. There are four different permissions: read, write, execute, and shared. If read/write/execute are disabled, a - will appear instead of the r/w/x. If a region is not *shared*, it is *private*, so a p will appear instead of an s. If the process attempts to access memory in a way that is not permitted, a segmentation fault is generated. Permissions can be changed using the mprotect system call.

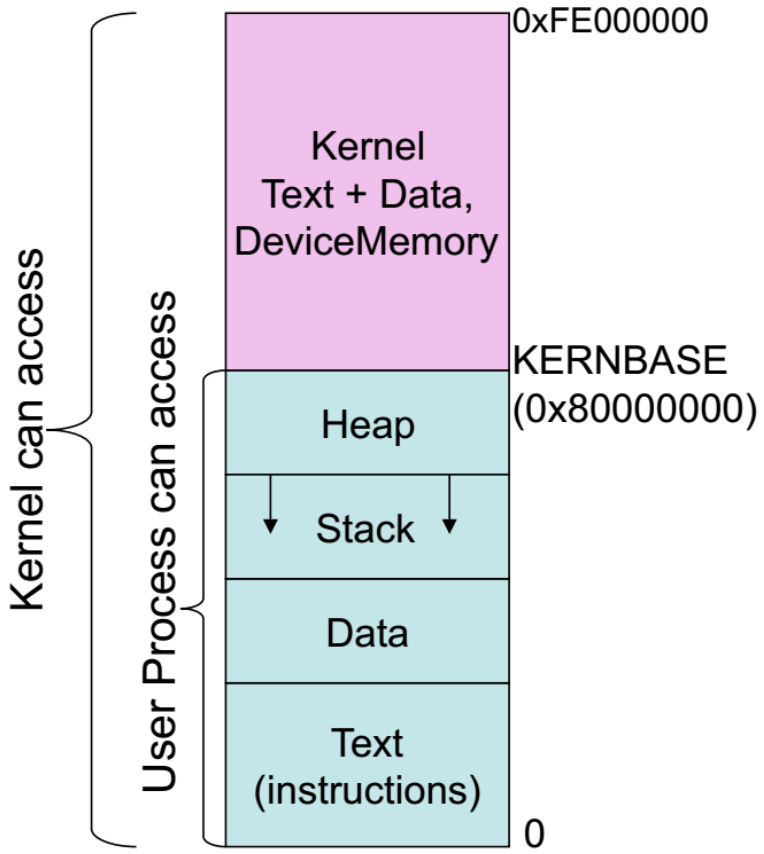
offset - If the region was mapped from a file (using mmap), this is the offset in the file where the mapping begins. If the memory was not mapped from a file, it's just 0.

device - If the region was mapped from a file, this is the major and minor device number (in hex) where the file lives.

inode - If the region was mapped from a file, this is the file number.

pathname - If the region was mapped from a file, this is the name of the file. This field is blank for anonymous mapped regions. There are also special regions with names like [heap], [stack], or [vdso]. [vdso] stands for virtual dynamic shared object. It's used by system calls to switch to kernel mode.

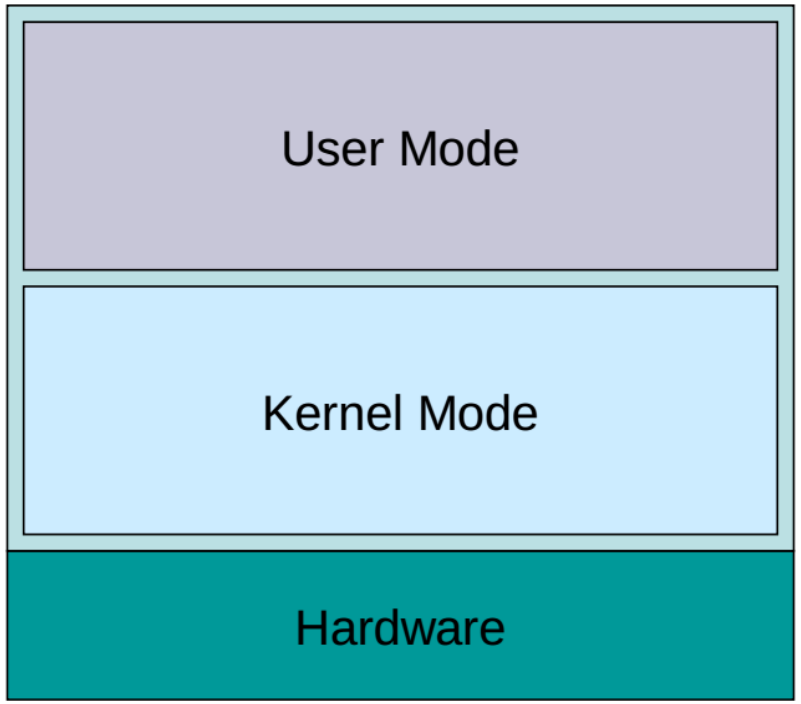
Beyond the process's memory Map



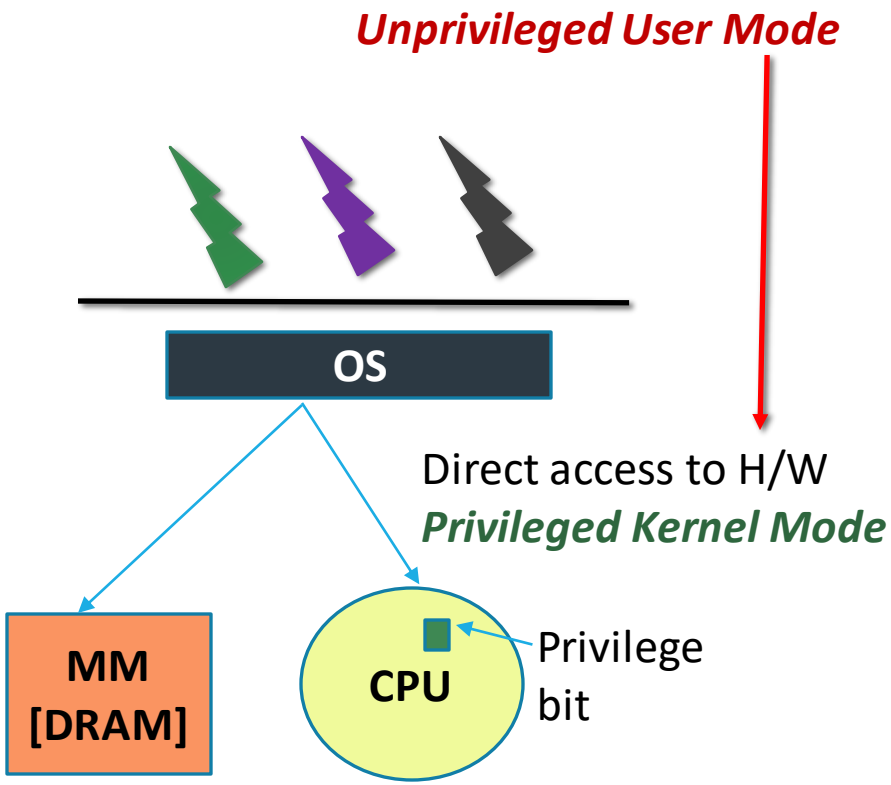
- Entire kernel mapped into every process address space
  - This allows easy switching from user code to kernel code (ie. during system calls)
    - No change of page tables needed
  - Easy access of user data from kernel space

Operating Modes

- User Mode
  - Where processes run
  - Restricted access to resources
  - Restricted capabilities
- Kernel mode a.k.a. Privileged mode
  - Where the OS runs
  - Privileged (can do anything)



User/Kernel Protection Boundary



User to Kernel level switching is supported by H/W

In Kernel mode; **a special bit is set in CPU**; If this bit is set; any instruction that manipulates H/W is permitted to execute

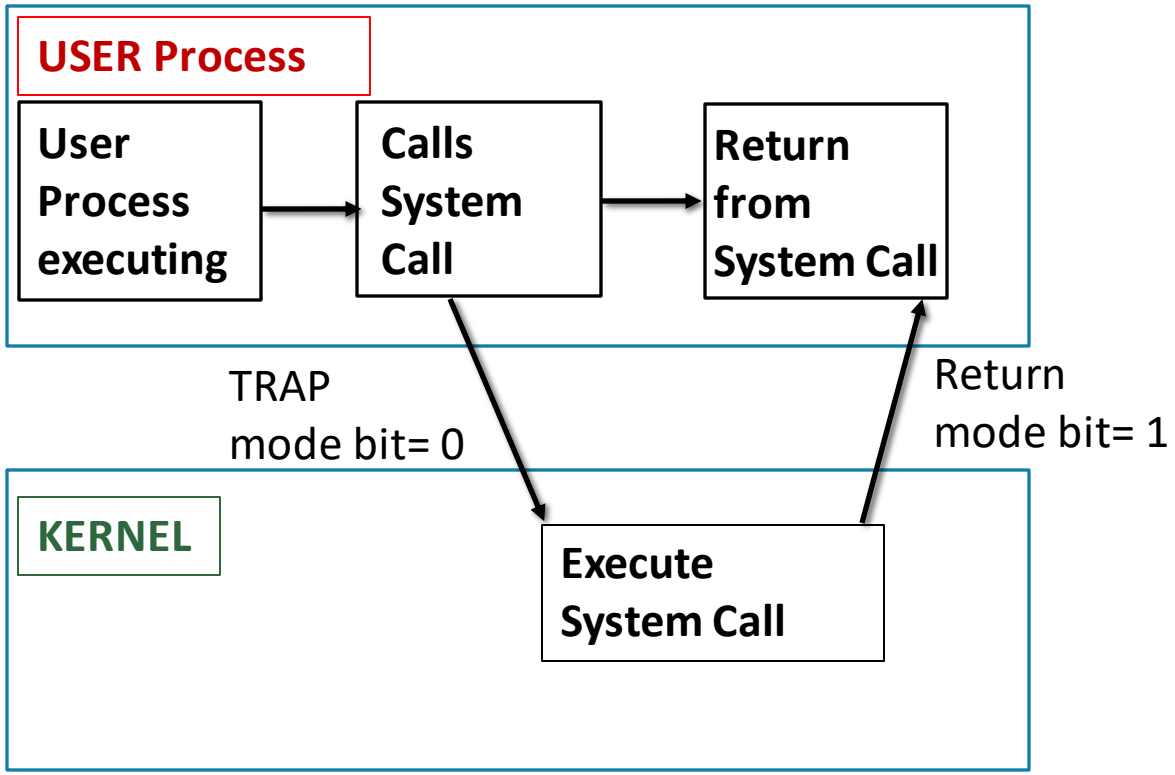
In User mode; privilege bit is not set and so instructions would be forbidden to manipulate H/W

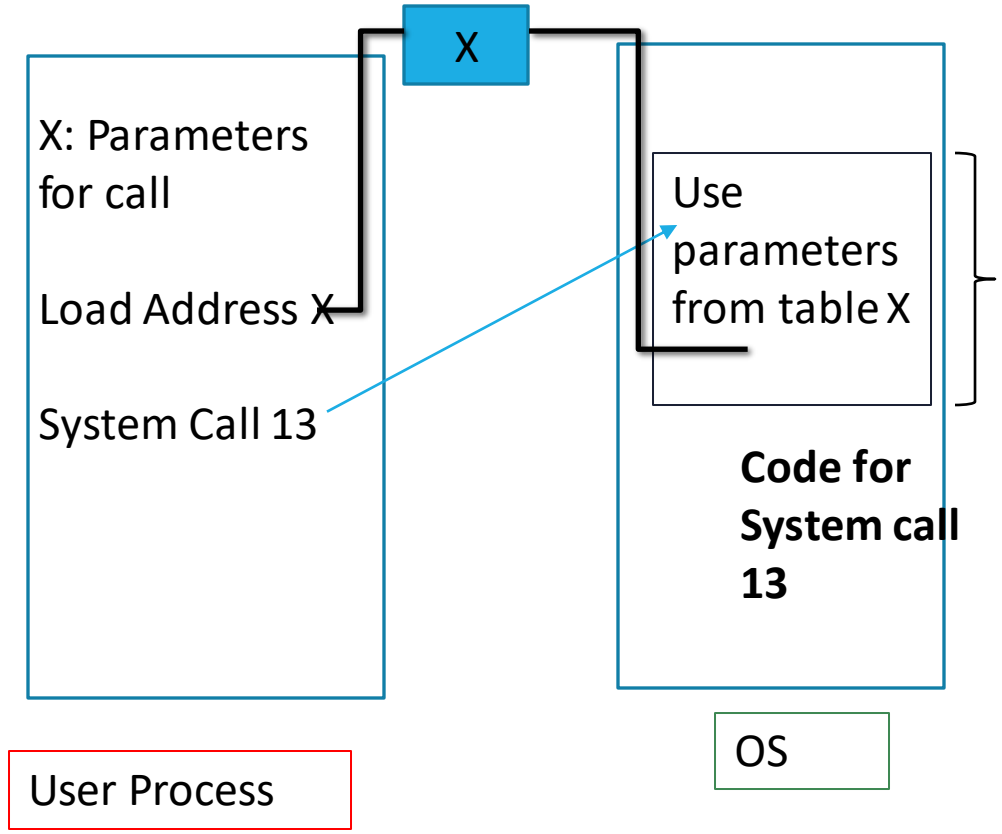
TRAP instruction is generated when user mode tries to perform privileged operations [Application execution interrupted, control transfer to OS at a specific location; OS will check what caused the trap to occur [verify to grant access or NOT]]

SYSTEM CALLS –Privileged operations will be performed by OS on behalf of applications {open[file], send[socket], mmap [memory]}

SIGNALS – OS notifications to Applications [kill]

System Calls

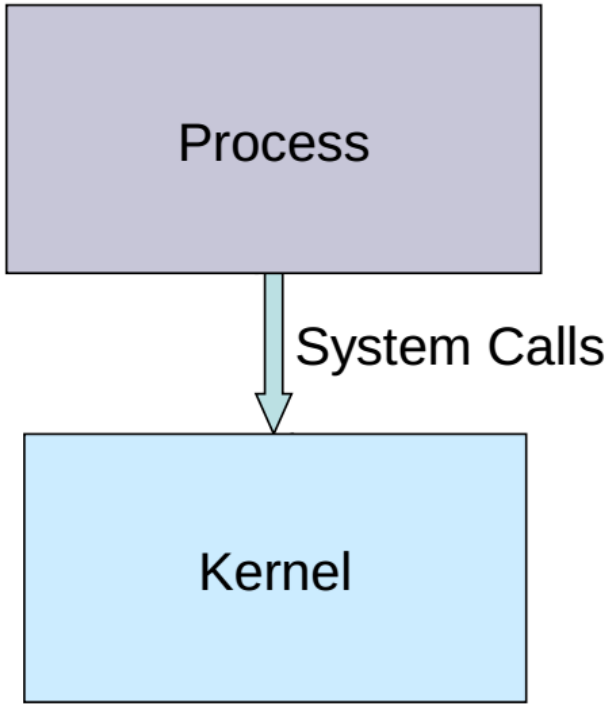




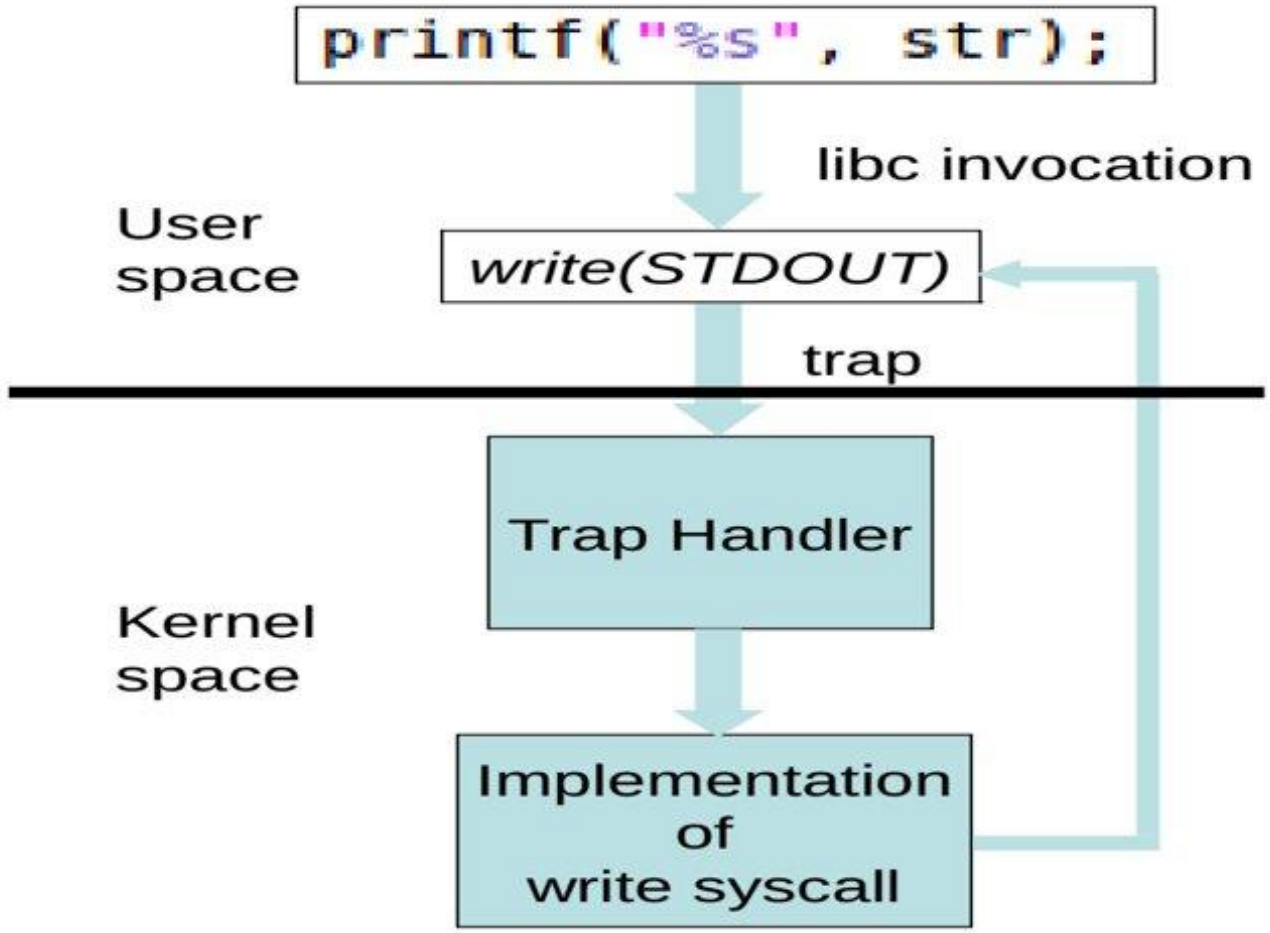
To make a system call; app must write arguments; save relevant data at well-defined location

System Calls

- System call invokes a function in the kernel using a Trap
- This causes
  - Processor to shift from user mode to privileged mode
- On completion of the system call, the execution gets transferred back to the user mode process



Example


printf("%s", str);
libc invocation
User space
write(STDOUT)
trap
Trap Handler
Kernel space
Implementation of write syscall

Consider a process that wants to write to a file

Process makes syscall to write()

- Transition to kernel mode
- Perform write operation if valid
- Return back to the user mode for continuing process

Two different set of Operations

System call vs Procedure call

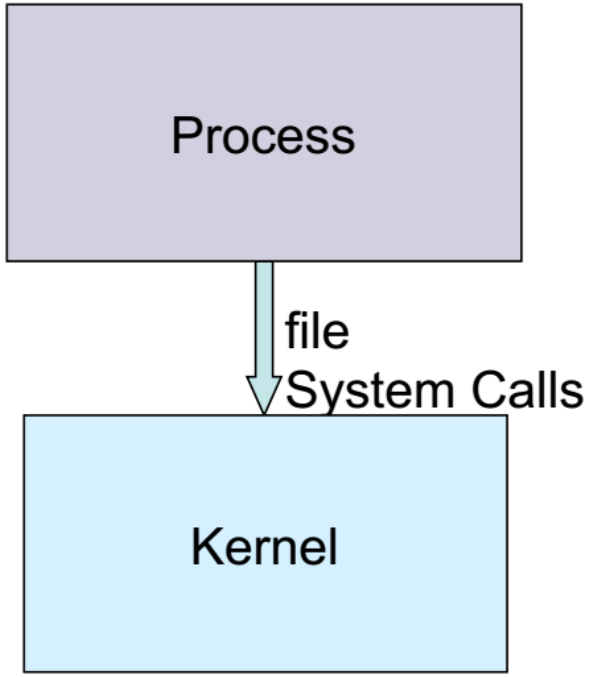
System Call	Procedure Call
Uses a TRAP instruction (such as int 0x80)	Uses a CALL instruction
System shifts from user space to kernel space	Stays in user space (or kernel space) … no shift
TRAP always jumps to a fixed addess (depending on the architecture)	Re-locatable address

System Call Interfaces

- System calls provide users with interfaces into the OS.
- What set of system calls should an OS support?
  - Offer sophisticated features
  - But yet be simple and abstract whatever is necessary
  - General design goal : rely on a few mechanisms that can be combined to provide generality

File system calls – how to design?

- Data persistent across reboot
- What should the file system calls expose?
  - Open a file, read/write file, creation date, permissions, etc.
  - More sophisticated options like seeking into a file, linking, etc.
- What should the file system calls hide?
  - Details about the storage media.
  - Exact locations in the storage media.



System Calls

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

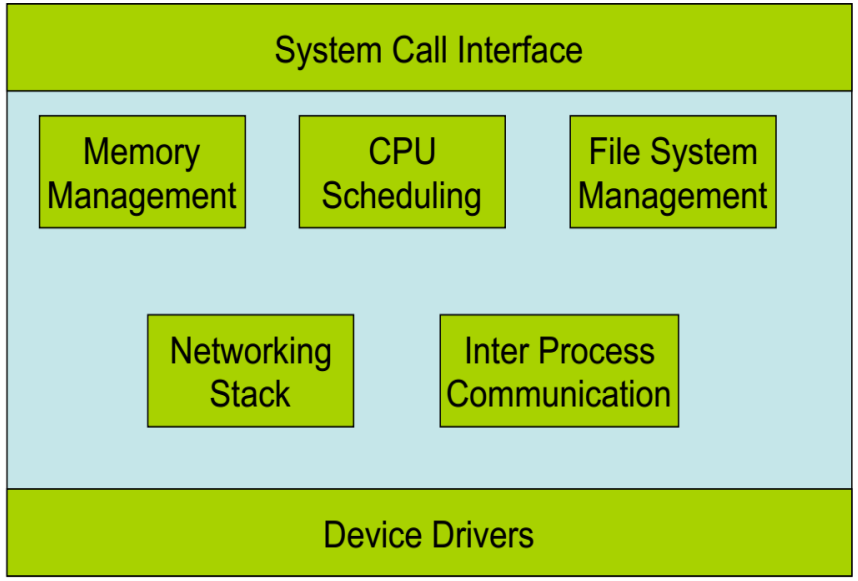
Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

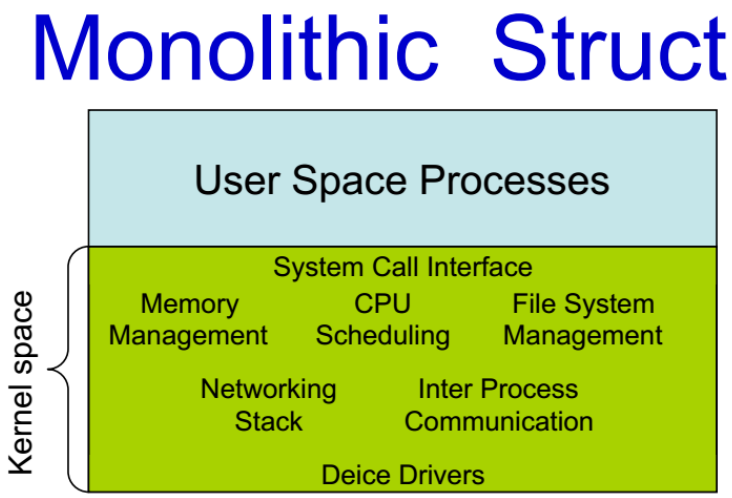
Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

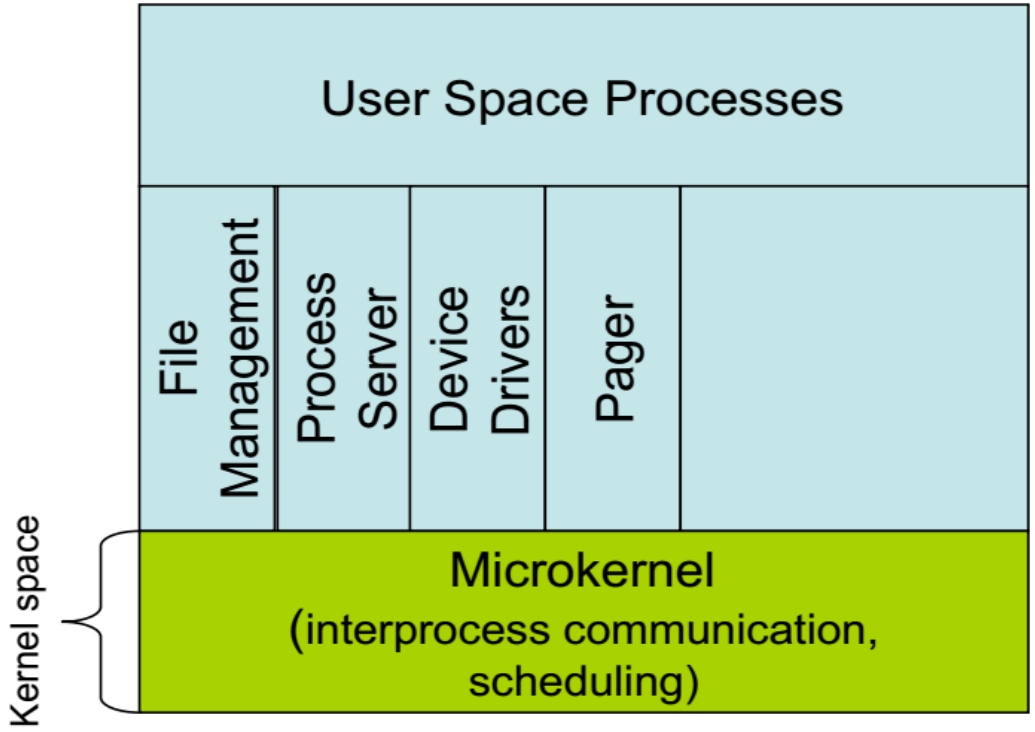
Monolithic OS





- Linux, MS-DOS, xv6
- All components of OS in kernel space
- Cons : Large size, difficult to maintain, likely to have more bugs, difficult to verify
- Pros : direct communication between modules in the kernel by procedure calls

Microkernel



Eg. QNX and L4

- Highly modular.
  - Every component has its own space.
  - Interactions between components strictly through well defined interfaces (no backdoors)
- Kernel has basic inter process communication and scheduling
  - Everything else in user space.
  - Ideally kernel is so small that it fits the first level cache

Comparison

	Monolithic	**Microkernel**
Inter process communication	Signals, sockets	Message queues
Memory management	Everything in kernel space (allocation strategies, page replacement algorithms,)	Memory management in user space, kernel controls only user rghts
Stability	Kernel more ‘crashable’ because of large code size	Smaller code size ensures kernel crashes are less likely
I/O Communication (Interrupts)	By device drivers in kernel space. Request from hardware handled by interrupts in kernel	Requests from hardware converted to messages directed to user processes
Extendibility	Adding new features requires rebuilding the entire kernel	The micro kernel can be base of an embedded system or of a server
Speed	Fast (Less communication between modules)	Slow (Everything is a message)