CS310 Operating Systems

Lecture 6: System Calls

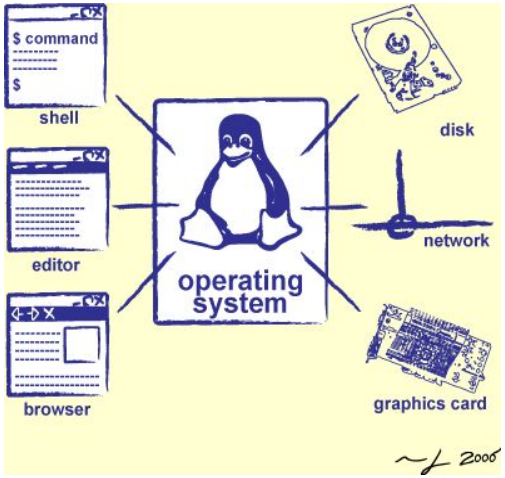
Ravi Mittal
IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, Profs. Natacha Crooks and Anthony D. Joseph, Univesity of California, Berkeley
  - Book: Operating System Concepts, 10th Edition, by Silberschatz, Galvin, and Gagne

We will study..

- Kernel Stack – another system data structure
- System Calls
- Application Programming Interfaces
- A tree of processes in Linux

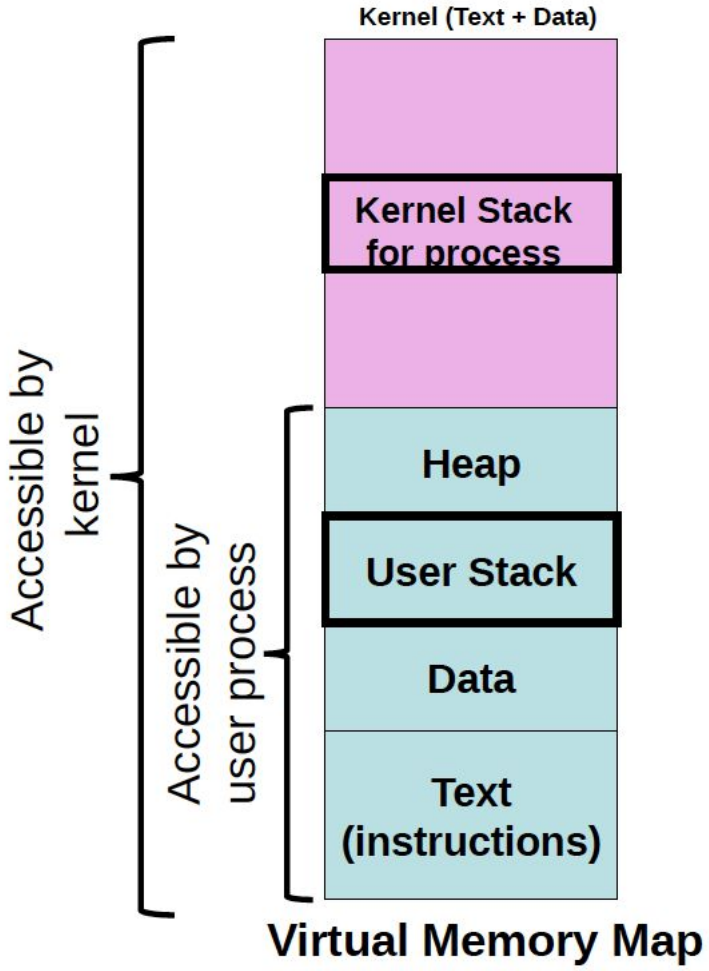


Read the following:

- Book: Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Volume 1, Kernel and Processes
    - Chapter 2.4: About Kernel Stack
- Book: Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau
  - Chapter 5: Process APIs

Kernel Stack

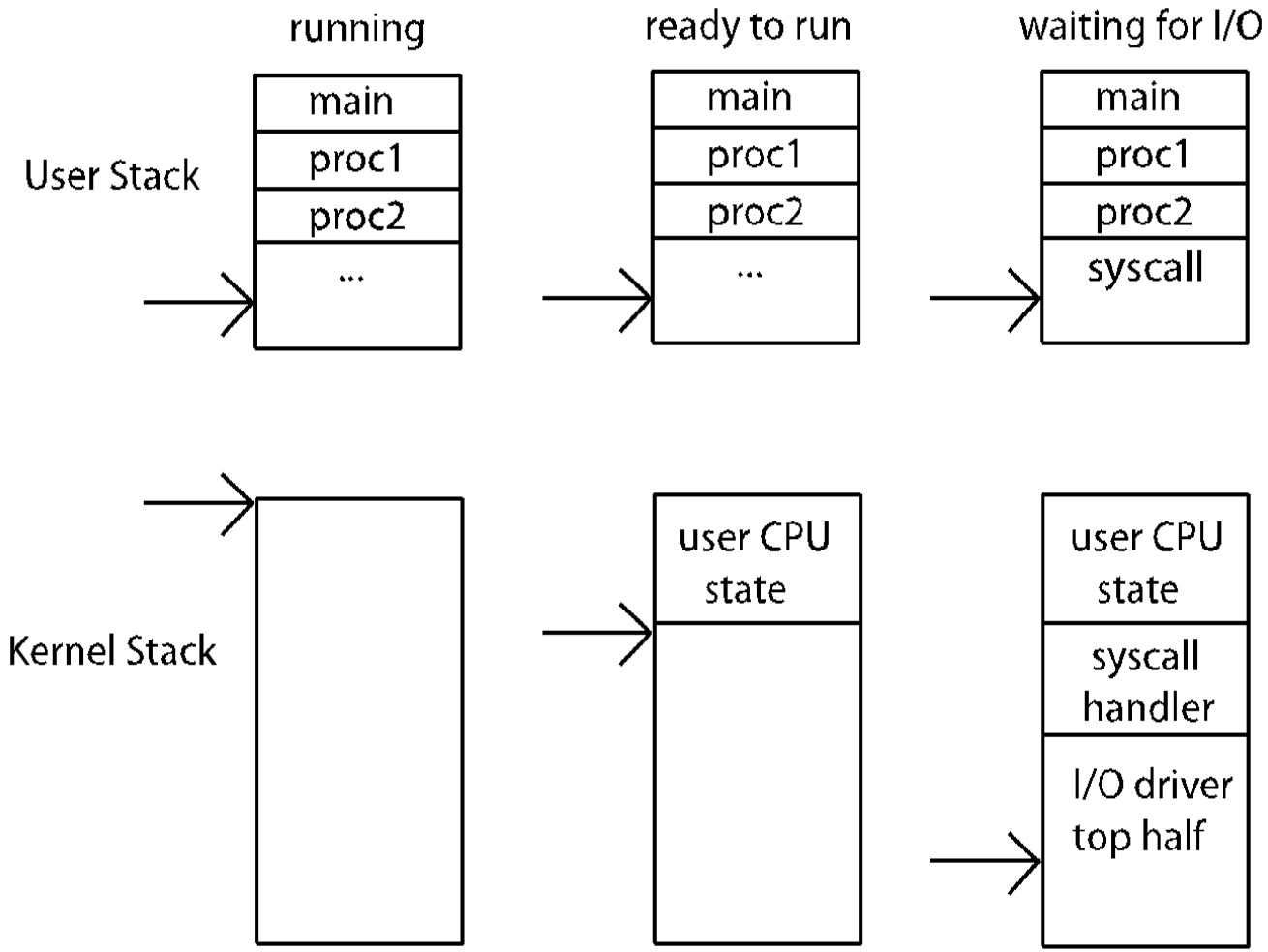
- For each user level process, most OSs allocate two stacks for
  - User stack
  - Kernel stack
- Kernel needs space to work
  - Can't put anything on the user stack (?)



Kernel Stack

- This simplifies switching to new process
- When a process is running in user mode, the kernel stack is empty
- The pointer to process's kernel stack is stored in PCB
- If the process is running on the processor on kernel mode, due to an interrupt or system call, it's kernel stack contains
  - Registers (saved from the suspended process)
  - PC, Stack pointer
  - Flags (Processor status word in x86)
- A special privileged hardware register (stack pointer) points to a region of kernel memory called the Kernel stack

Kernel Stack



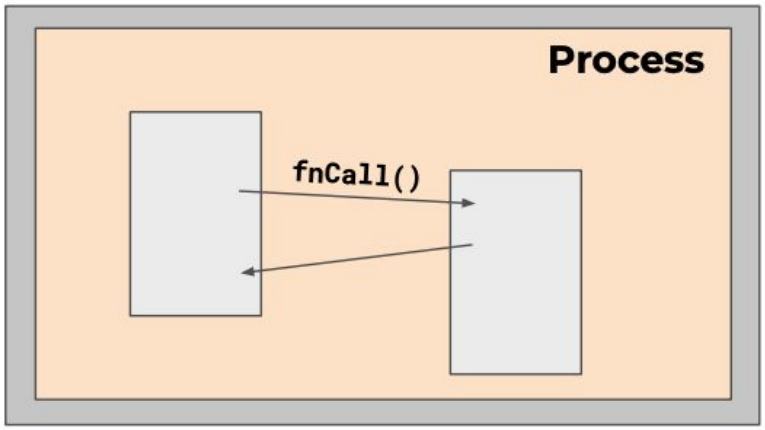
Kernel Stack contents

- When the process is running, does the kernel stack have anything useful on it?
  - You might think yes – it called into the user program.
  - Actually no
- When a process is ready to run, but not running
  - It has its user stack as before,
  - We need a place to store the state that had been in the CPU when it stopped running
- When a process is waiting for I/O
  - The kernel stack contains the suspended computations
    - Eg Syscall handler state, I/O driver state etc
  - User CPU state: registers, PC, SP etc

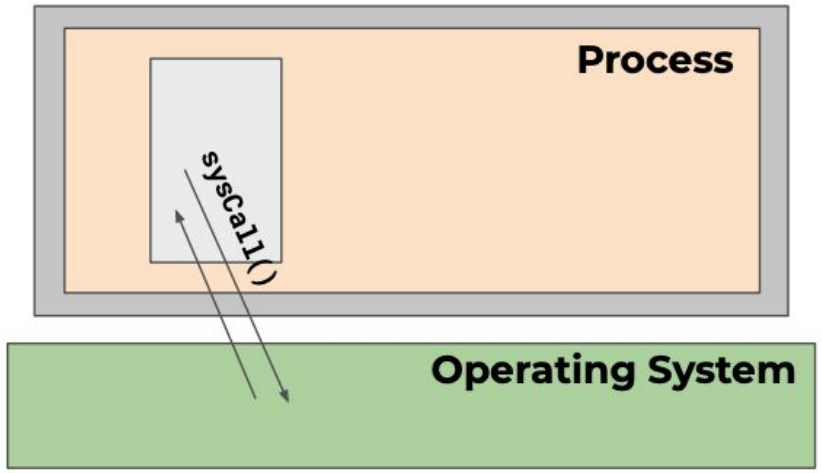
System Calls

Function Calls vs System Calls

Function Call:



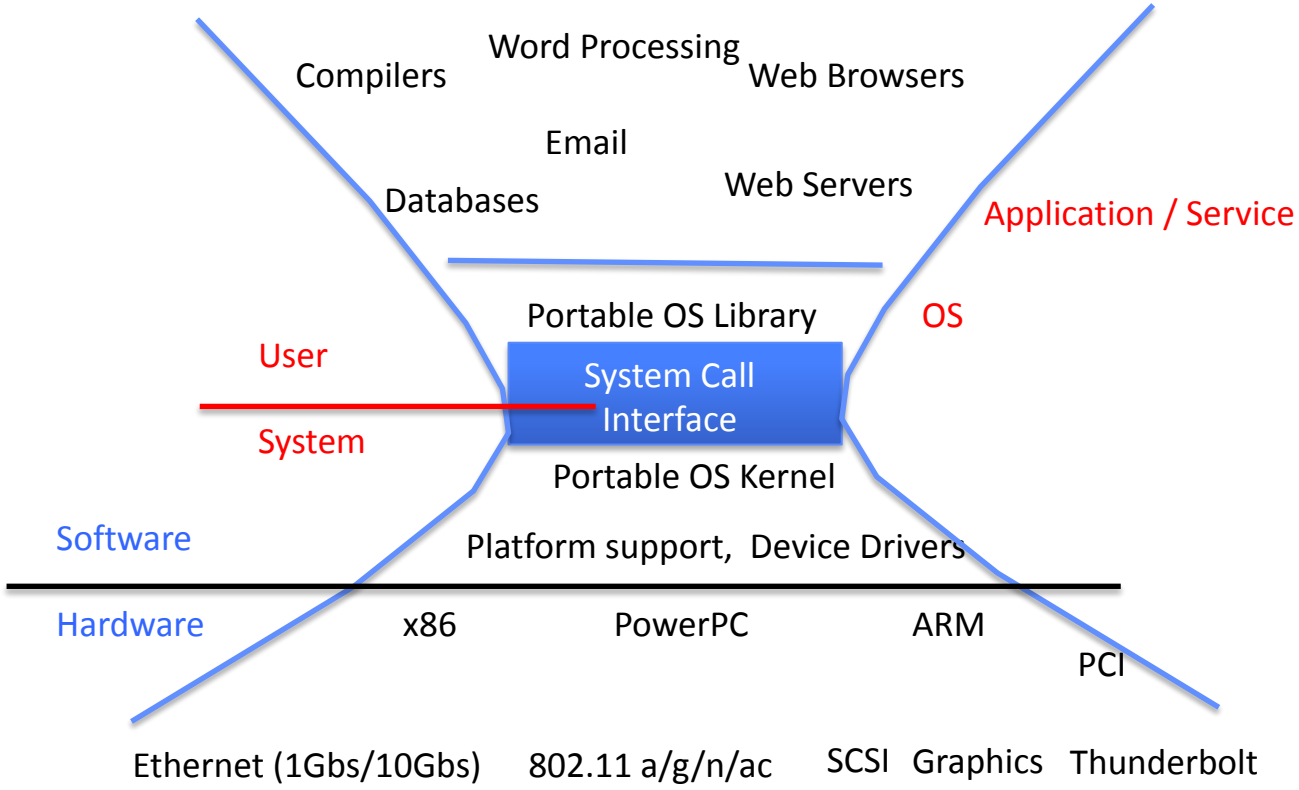
System Call



- Caller and callee in the same process
  - Same user
  - Same "domain of trust"

- OS is trusted; User process is not
- OS code runs privileged with complete access to all system resources

System Calls ("Syscalls")



System Calls

- API : Application Programming Interface
  - Function available to write user programs
- System calls are the interface of the OS for
  - Processes
    - Creating, existing, waiting, and terminating
  - Memory
    - Allocation and deallocation
  - Files and Folders
    - Opening, reading, writing, closing
  - Inter Process Communication

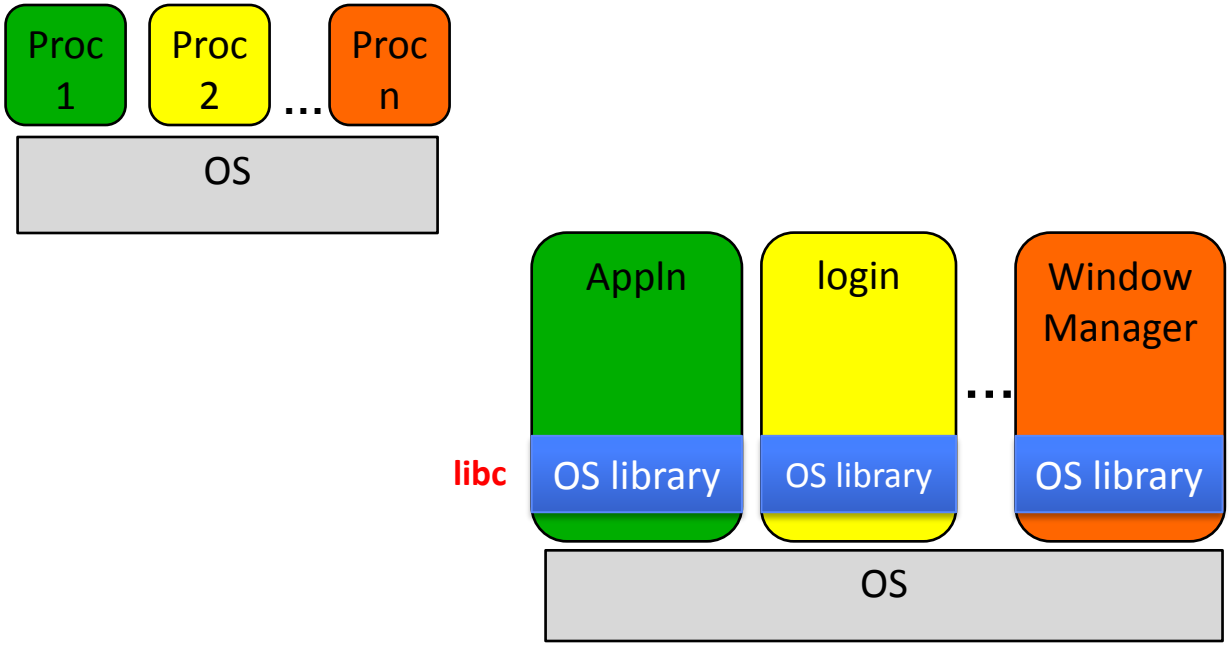
- We will study process system calls in later lectures

Should we rewrite programs for each OS?

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

C □ libc □ syscall

OS Library Issues Syscalls



APIs

- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use

Multiple Sys Calls are needed ..

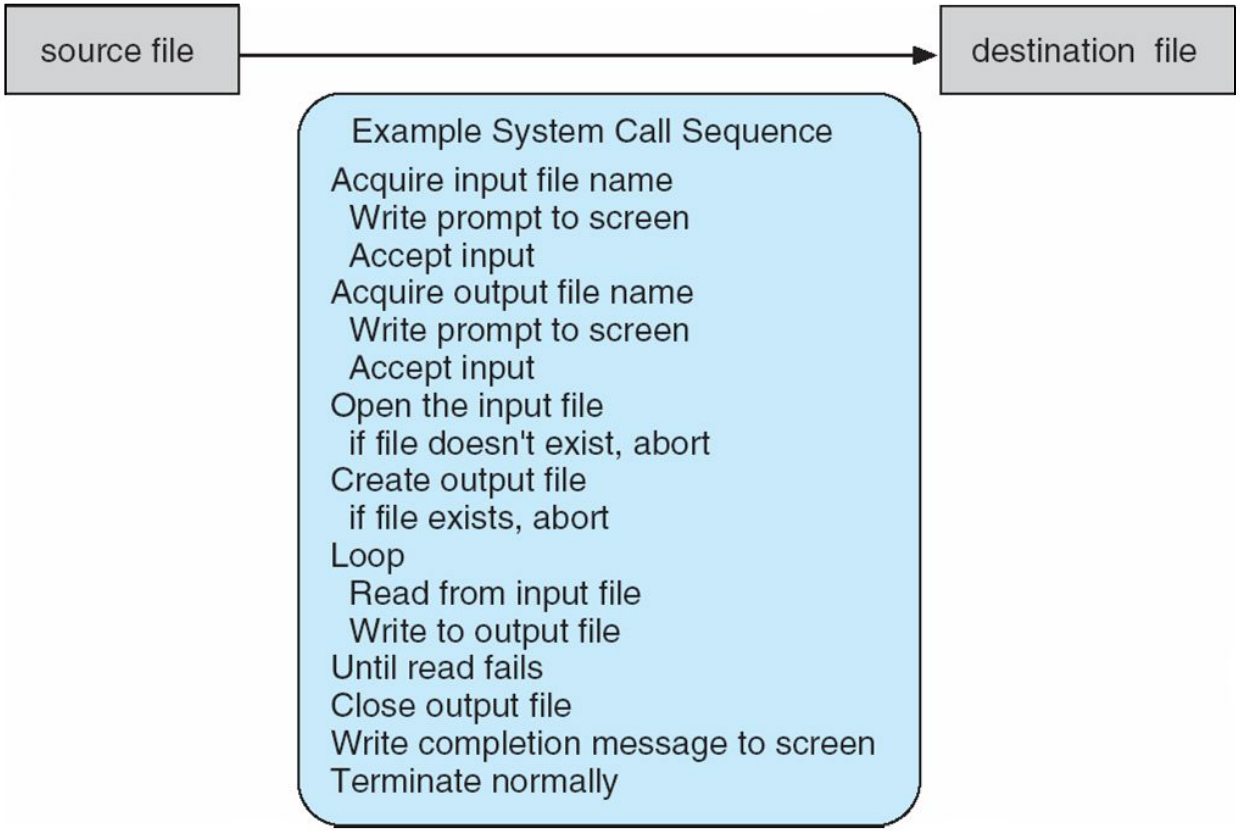
- Consider following Unix command

```
cp in.txt out.txt
```

- This command copies the input file `in.txt` to the output file `out.txt`
- This seemingly simple command requires a large number of system calls
  - Write prompting message on terminal
  - To read keyboard input
  - To write text onto the terminal
  - Open input file and open output file
  - Copy text from input file to output file – in loop
  - Possible error message if files don't exist
  - Closing both files on completion
  - And so on

Example of System Calls

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



Application APIs

Application Programming Interface (API)

- A simple program uses hundreds of system calls in a short time
- Application Programmers use APIs
  - API specifies
    - a set of functions that are available to an application programmer
    - Parameters passed to each function and return values it can expect
- 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)
- A programmer accesses an API via a library of code provided by the operating system
- `libc`: For Linux and Unix, library for programs written in C

APIs vs System Call

- System Call is an explicit request to the kernel made via a software interrupt
- API is a function definition that specifies how to obtain a given service
  - An OS has several libraries of functions that provide APIs to programmers:
    - Some APIs provide services to users in User Mode
    - Other APIs use system calls (in turn use kernel mode)
      - Wrapper over system call with added functionality
- Example:
  - In Linux, the malloc( ) , calloc( ) , and free( ) APIs are implemented in the libc library
    - The code in this library keeps track of the allocation and deallocation requests and uses the brk( ) system call to enlarge or shrink the process heap

Examples of API calls and System Calls

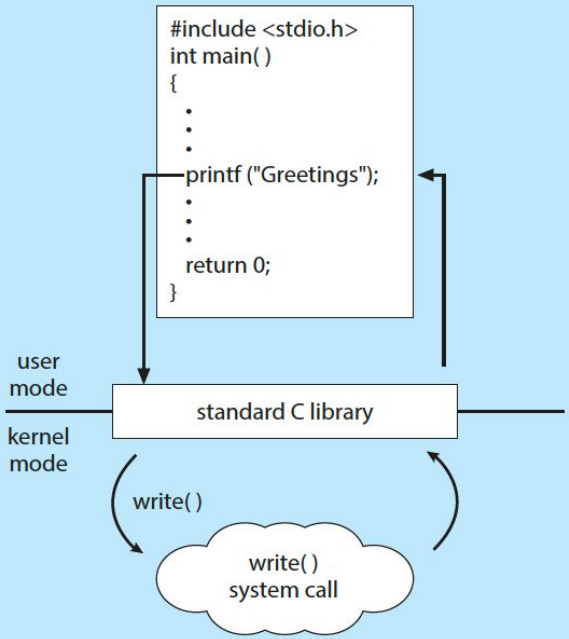
C Library Call:	Linux System Call:	Win32 System Call:
fopen	open	CreateFileA
fclose	close	CloseHandle
getc/putc	read/write	ReadFile / WriteFile
fread/fwrite		
scanf/printf		
fprintf		
fseek	lseek	SetFilePointer
rand		

Standard C Library Example

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

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:

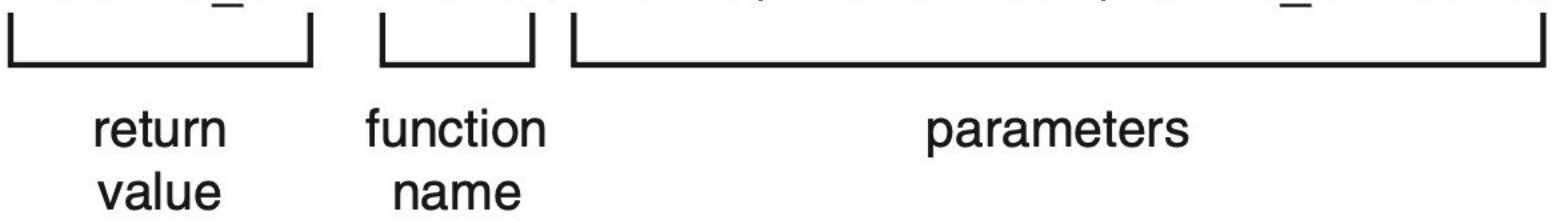


Advantages of using APIs instead of System Calls

- Program Portability
  - Application programmer need not worry about processor and OS?
- Simple to use compared to system calls
- Standard `read` API is described as

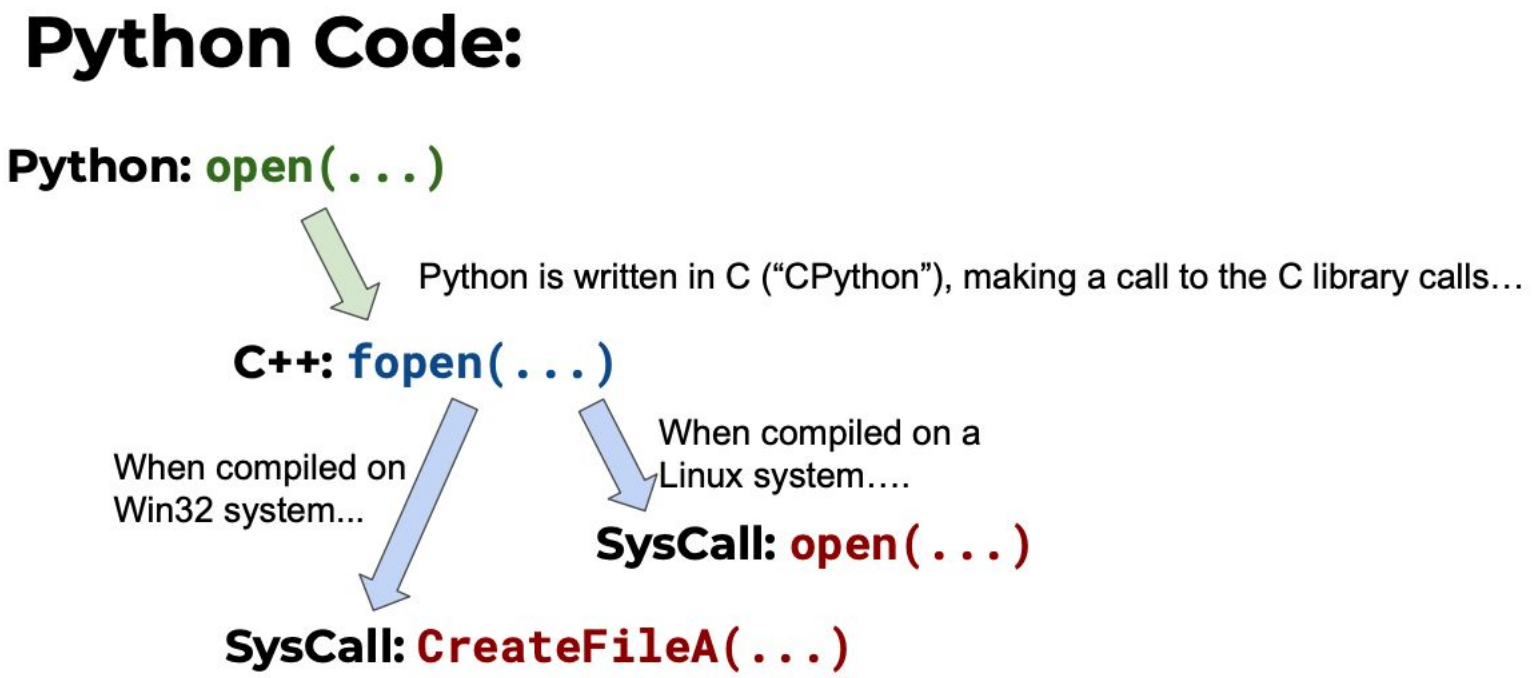
```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```



- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `Size_t count`—the maximum number of bytes to be read into the buffer

Python Code


Python Code:
Python: open(...)
Python is written in C ("CPython"), making a call to the C library calls…
C++: fopen(...)
When compiled on a Linux system….
When compiled on Win32 system...
SysCall: open(...)
SysCall: CreateFileA(...)

System Call Implementation

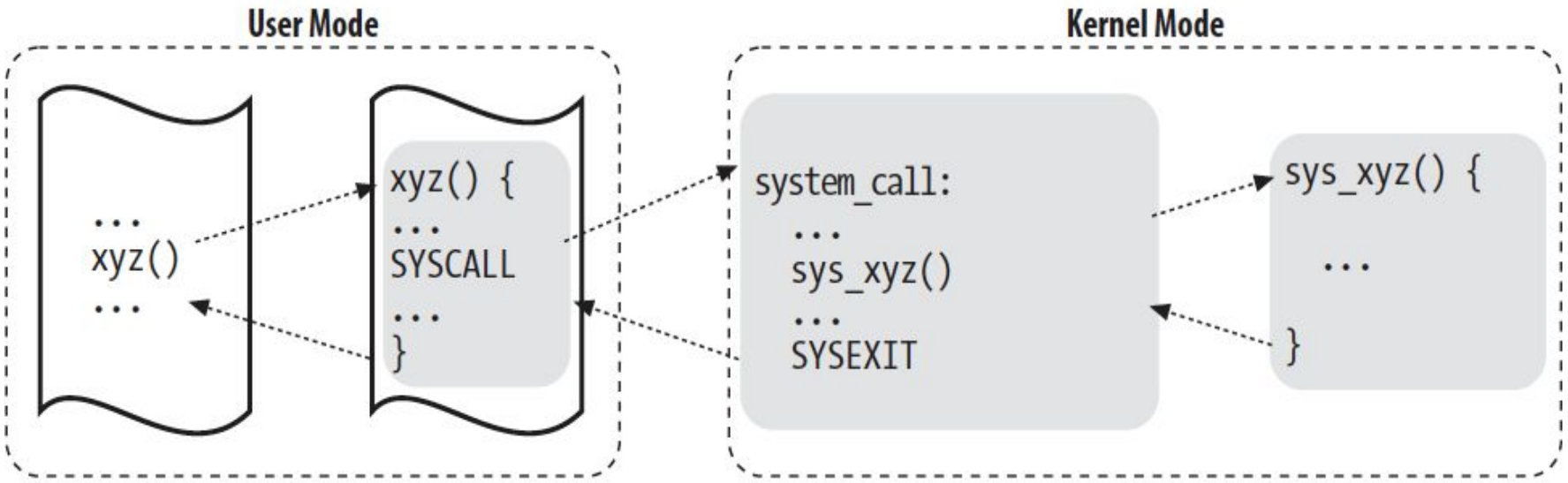
System Call Implementation

- Typically, a number is 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)

System Call Implementation

- When a process makes a system call, the processor switches to kernel mode
- In Kernel mode it runs System Call handler
- User process passes a parameter called system call number and other parameters
- System call handler is similar to other exception handler (will study it later). It performs the following actions:
  - Saves contents of registers in Kernel Mode Stack
  - Handles the system call by invoking a function called system call service routine
  - Exit from handler: Regs are loaded with values saved in Kernel stack ; CPU is switched back to user mode

Invoking a system call from Applications



Application program

Wrapper Routine

System Call handler

System Call Service Routine

Types of System Calls

Types of System Calls

- Process control
- File Management
- Device Management
- Information Management
- Communications
- Protection

System Calls in Windows and Unix

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	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()`

Book: The Operating System Concepts: Silberschatz, Galvin, Gagne

Process Control - System Calls

- create process, terminate process
- end, abort
- load, execute
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory
- Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
- **Locks** for managing access to shared data between processes – `acquire_lock()` and `release_lock()`

File and Device Management - System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Information and Communication - System Calls

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - **message passing model -** send, receive messages to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions

Protection - System Calls

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Linux System Call

http://www.cheat-sheets.org/saved-copy/Linux_Syscall_quickref.pdf

- Linux Kernel 3.7 has 393 system calls

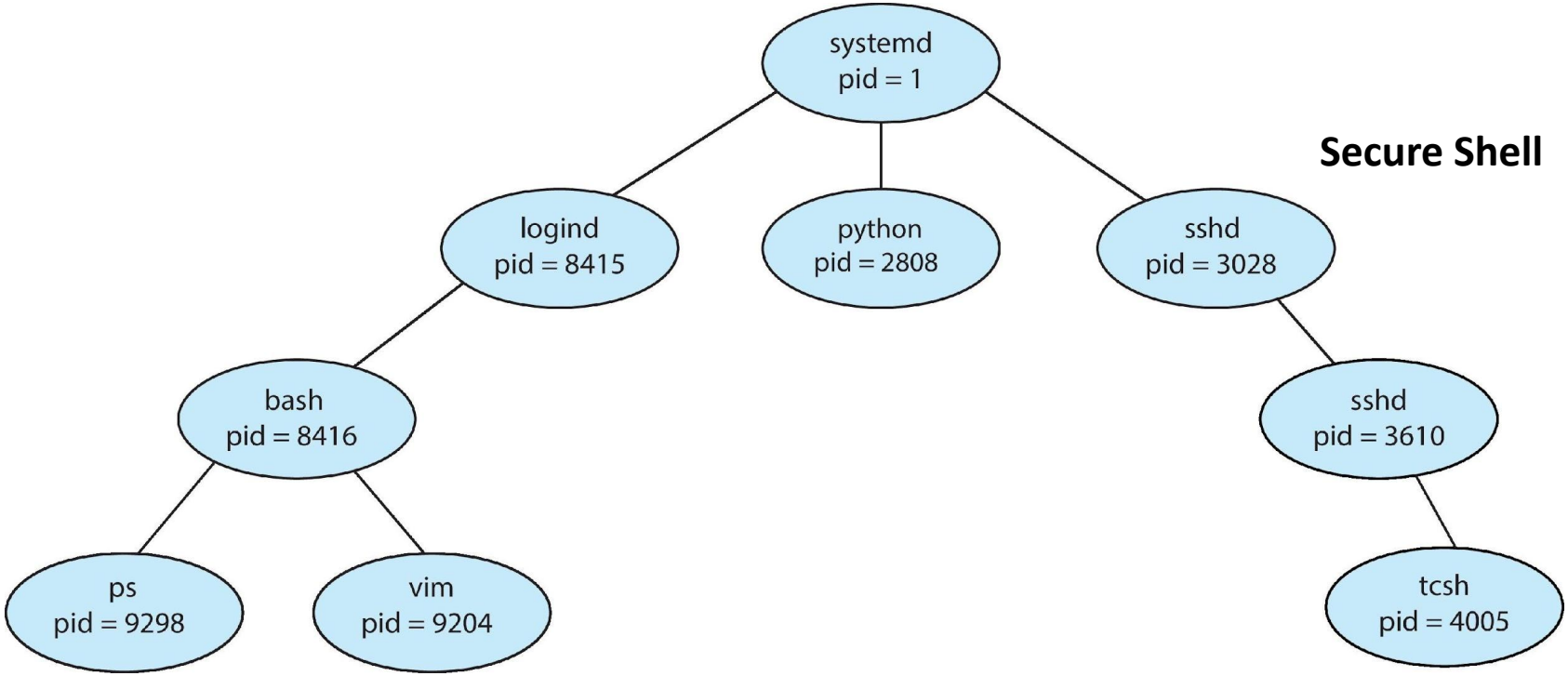
No	Func Name	Description	Source
1	exit	terminate the current process	*kernel/exit.c*
2	fork	create a child process	*arch/i386/kernel/process.c*
3	read	read from a file descriptor	*fs/read_write.c*
4	write	write to a file descriptor	*fs/read_write.c*
5	open	open a file or device	*fs/open.c*
6	close	close a file descriptor	*fs/open.c*
7	waitpid	wait for process termination	*kernel/exit.c*
8	creat	create a file or device ("man 2 open" for information)	*fs/open.c*
9	link	make a new name for a file	*fs/namei.c*
10	unlink	delete a name and possibly the file it refers to	*fs/namei.c*
11	execve	execute program	*arch/i386/kernel/process.c*
12	chdir	change working directory	*fs/open.c*
13	time	get time in seconds	*kernel/time.c*
14	mknod	create a special or ordinary file	*fs/namei.c*
15	chmod	change permissions of a file	*fs/open.c*
16	lchown	change ownership of a file	*fs/open.c*
18	stat	get file status	*fs/stat.c*
19	lseek	reposition read/write file offset	*fs/read_write.c*
20	getpid	get process identification	*kernel/sched.c*

Process Creation

Process Creation

- During execution, a process may create several new processes
  - Creating Process: Parent Process
  - New Processes: Children Processes
- Each of the children process may create other processes
  - Forming Tree of processes
- The systemd process (which always has a pid of 1) serves as the root parent process for all user processes
  - Systemd process in Unix is called init process

A Tree of Processes in Linux



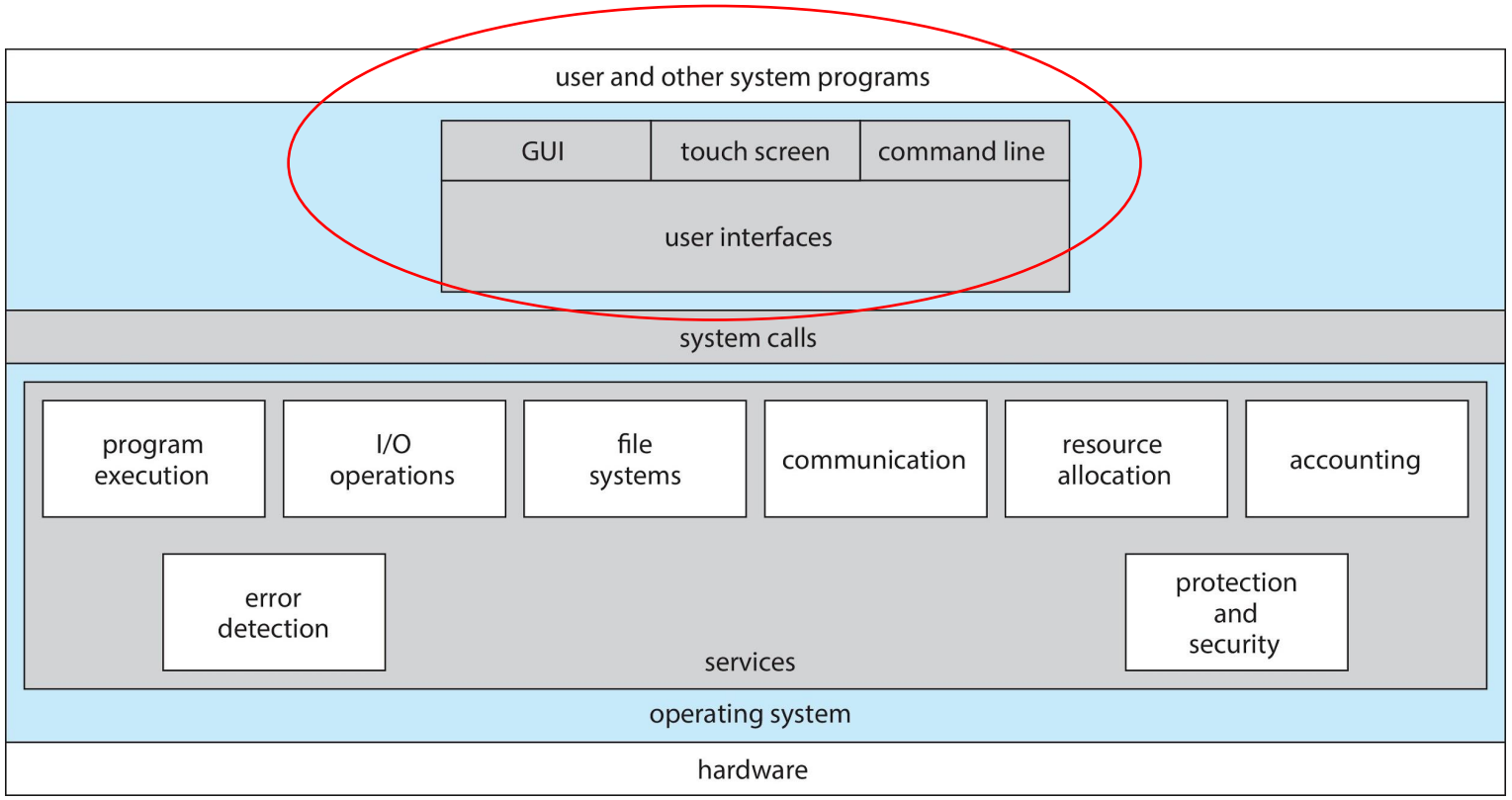
ps -el command gives details of all active processes in the system

Class Summary

- System Calls
  - System calls are interface for OS for manipulation of
    - Processes
    - Memory
    - File and folders
- Application APIs
  - A programmer usually uses higher level APIs instead of system calls
- User Interfaces: GUI, Command line, Touch Screen
- A tree of processes

User Interfaces (For self reading)

User Interfaces



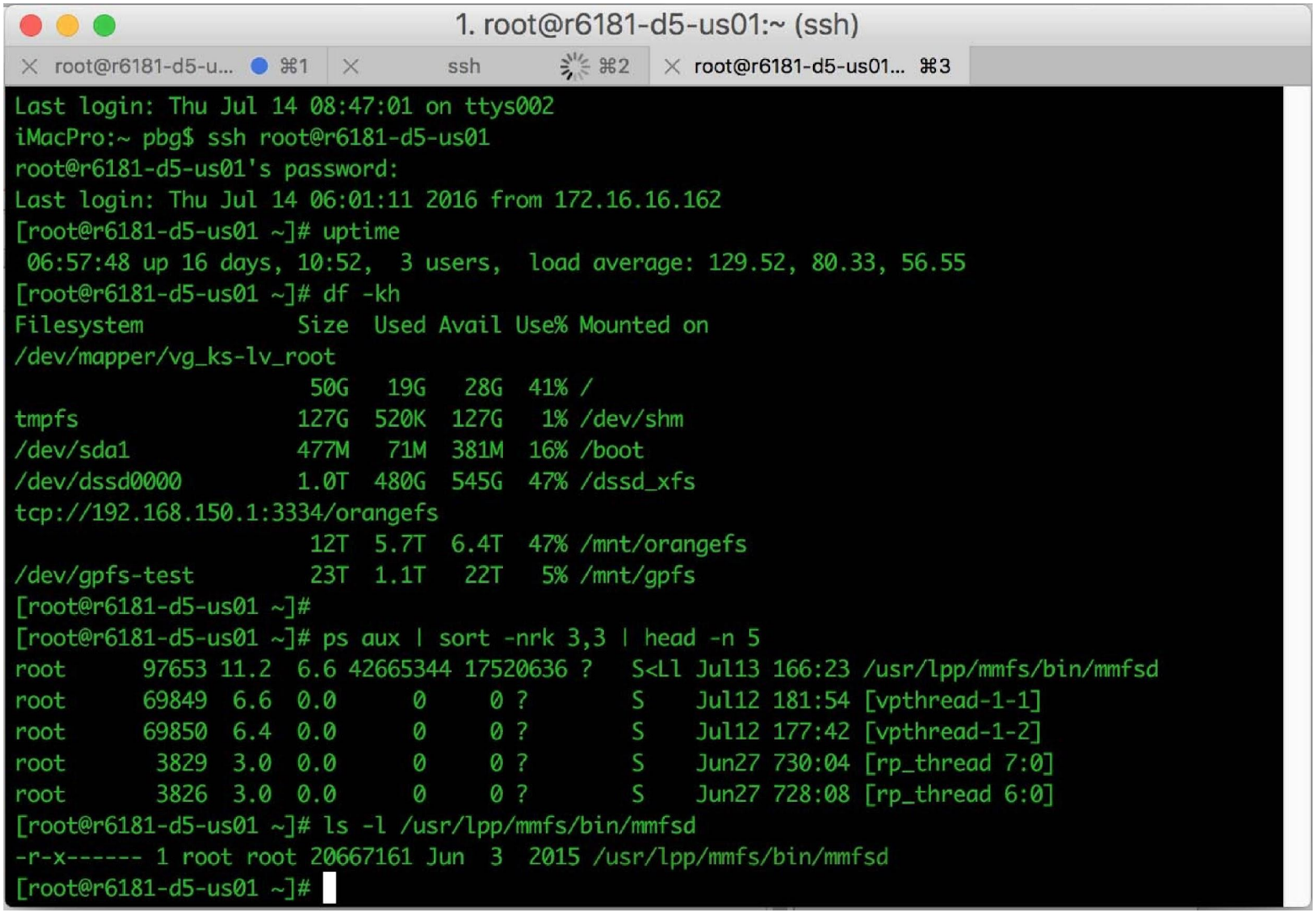
User Interfaces

- GUI
- Touch Screen
- Command Line Interface (CLI)
  - Uses text command
  - Allows users to directly enter commands executed by the OS
- Shells: Command Line Interpreters
  - C-Shell
  - Bourne-Again Shell (bash)
  - Korn Shell
- Example: Shell commands to
  - Create, delete, list, print, copy, execute - files

Interfaces

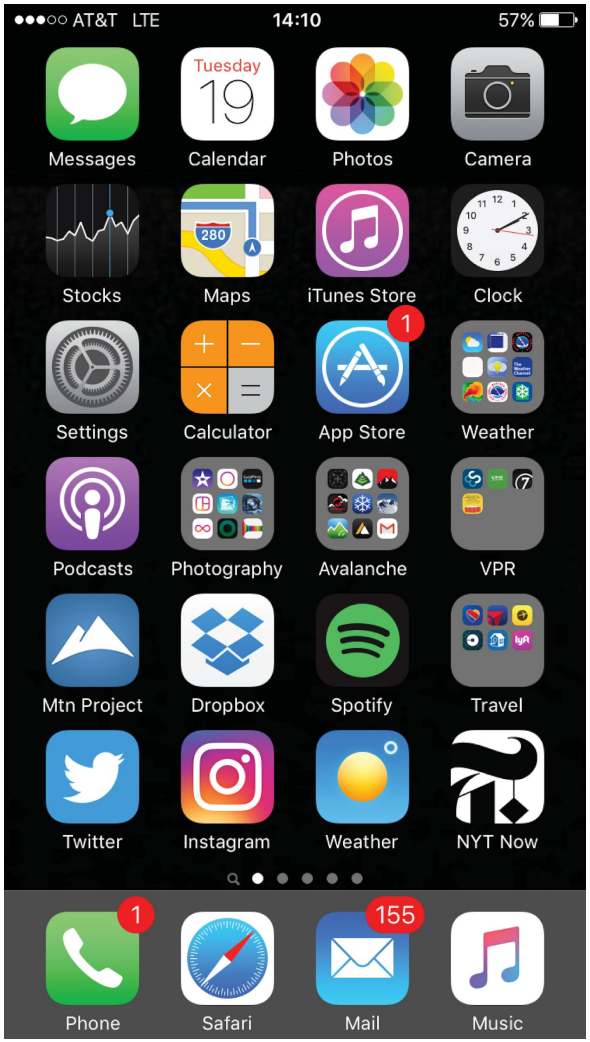
- GUIs are convenient for common users
  - Not all functionalities are available with GUIs
- CLIs : More powerful .. Used by advanced users and system administrators
  - More efficient, faster access, makes repetitive tasks easier
  - If a frequent task requires a set of command- line steps, those steps can be recorded into a file, and that file can be run just like a program
    - This program is interpreted by CLI
      - Shell Scripts
- Windows OS provides GUI
- MacOS provides both GUI and CLI

Bourne Shell Command Interpreter



Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands



System Calls - types