

OPERATING-SYSTEM STRUCTURES

Ruben Acuña

Spring 2019

MODULE OVERVIEW

- The goal of this module is gain an understanding of the high-level user features which a operating-system provides, and the underlying system features and design which enable them.
- Learning Objectives:
 - Understand the high-level user features which a typical operating-system supports.
 - Analyze the use of system calls in a program.
 - Evaluate the appropriateness of a kernel structure for a development scenario.



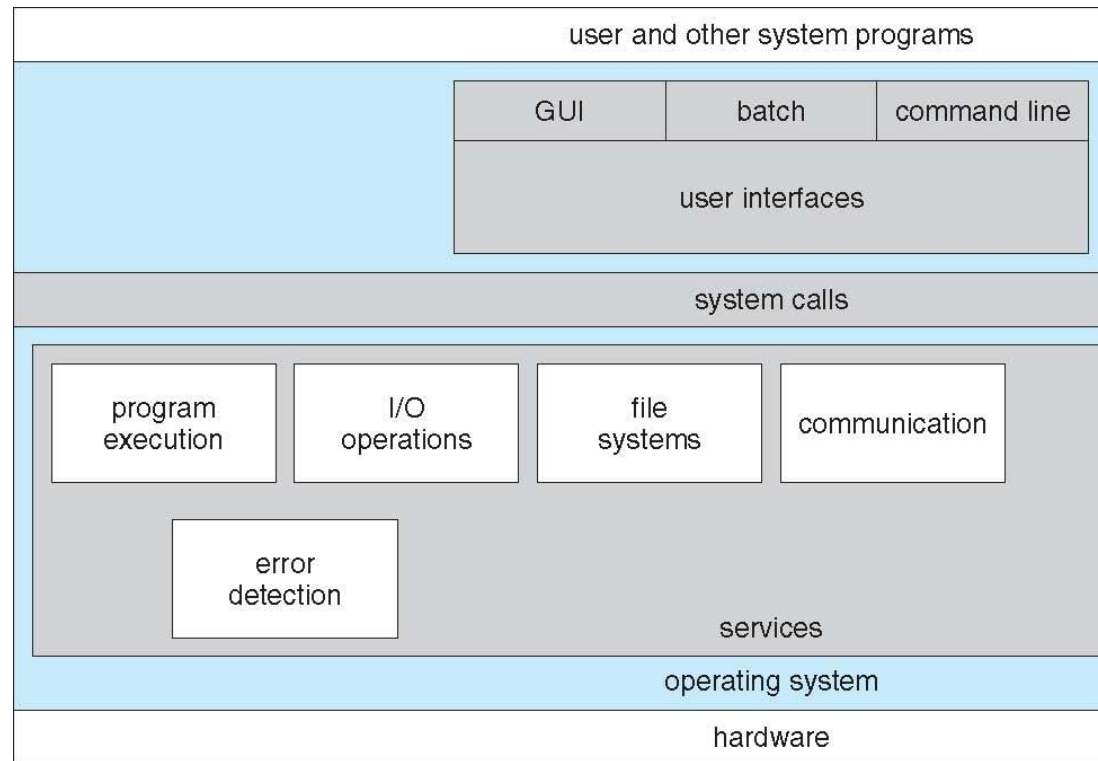
OPERATING-SYSTEM SERVICES

Silberschatz. Operating System Concepts (10e). Chapter 2.1.

SYSTEM SERVICES

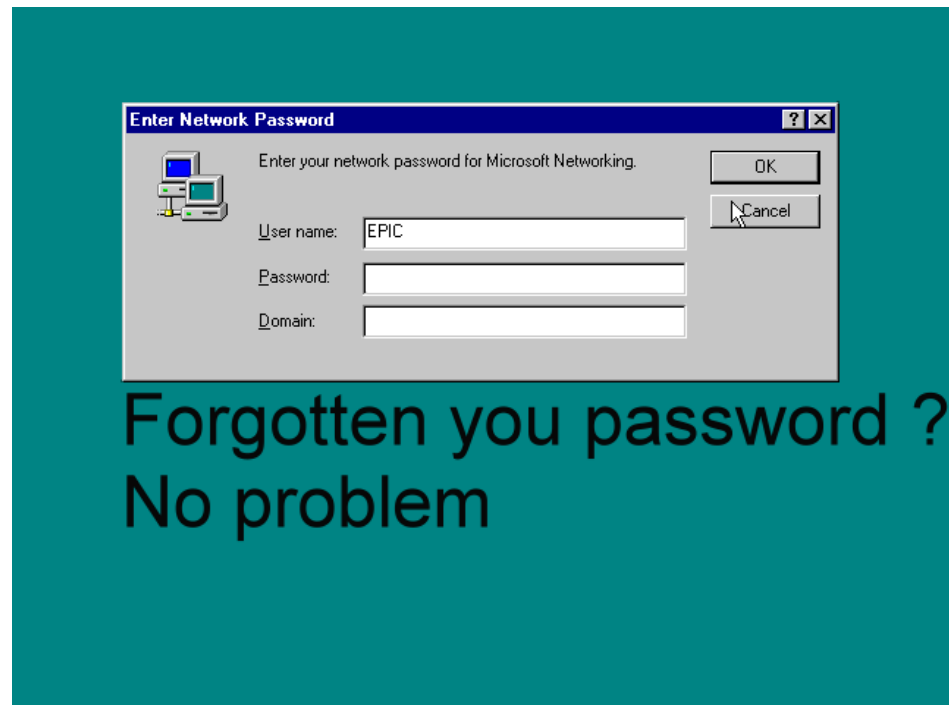
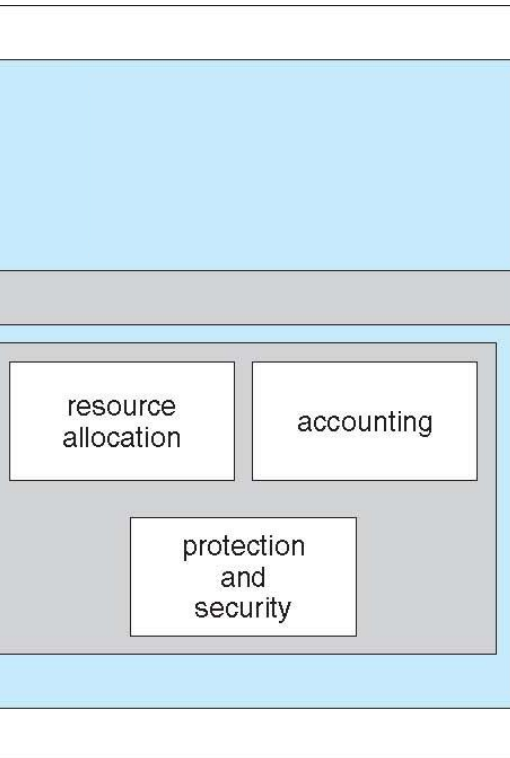
It's a little hard to classify, but generally an OS enables:

- *User interface*
- *Program execution*
- *I/O operations*
- *File-system manipulation*
- *Communications (local, net)*
- *Error detection*



OPERATION MANAGEMENT

- *Resource allocation* (exclusive vs non-exclusive)
- *Accounting*
- *Protection and security*



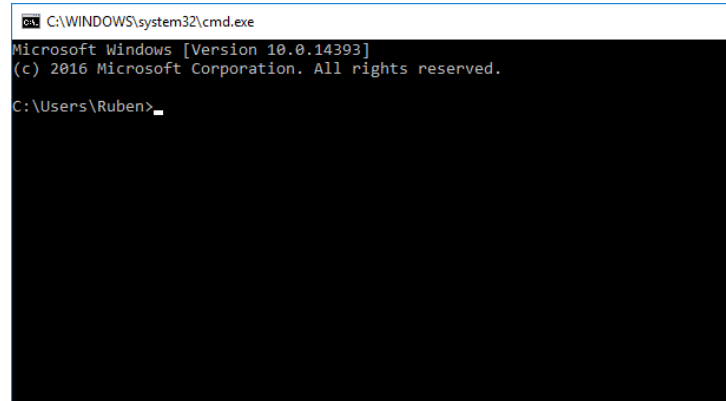


USER AND OPERATING-SYSTEM INTERFACE

Silberschatz. Operating System Concepts (10e). Chapter 2.2.

COMMAND LINE INTERFACES

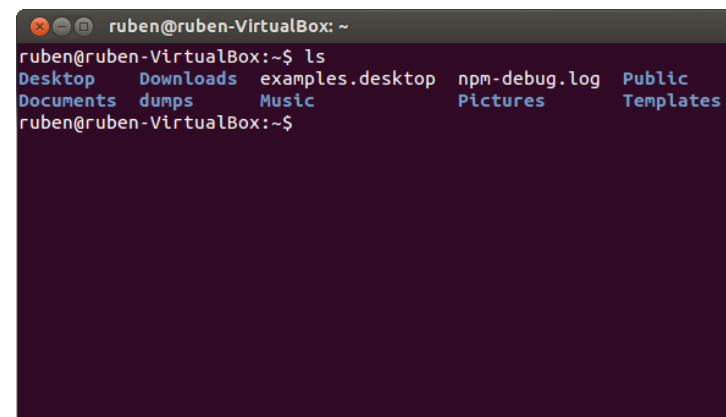
- The simplest way (?) for a user to interact with an OS, is via text.
- In a CLI, commands are processed by a *command interpreter*.
- You may also hear the term shell which indicates the command interpreter being used.
- For an OS, multiple shells may exist.
- There two general ways to implement a CLI's ability to execute commands may be implemented:
 - Self-contained
 - Modular system programs

A screenshot of a Windows 10 command prompt window. The title bar shows the file path "C:\WINDOWS\system32\cmd.exe". The window content displays the Microsoft Windows logo, the version "10.0.14393", and the copyright notice "(c) 2016 Microsoft Corporation. All rights reserved." Below this, the current directory is shown as "C:\Users\Ruben>".

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Ruben>
```

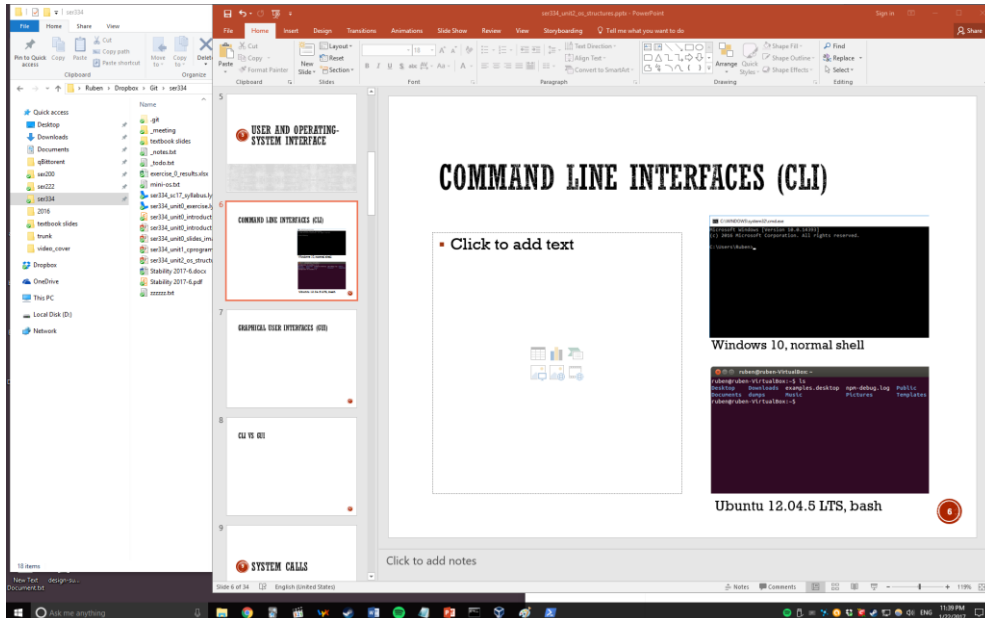
Windows 10, normal shell

A screenshot of a bash shell window running in a virtual machine. The title bar shows "ruben@ruben-VirtualBox: ~". The window content shows the prompt "ruben@ruben-VirtualBox:~\$" followed by the output of the "ls" command, which lists various directories and files in color. The prompt "ruben@ruben-VirtualBox:~\$" is repeated at the bottom.

```
ruben@ruben-VirtualBox: ~
ruben@ruben-VirtualBox:~$ ls
Desktop  Downloads  examples.desktop  npm-debug.log  Public
Documents  dumps      Music             Pictures        Templates
ruben@ruben-VirtualBox:~$
```

Ubuntu 12.04.5 LTS, bash

GRAPHICAL USER INTERFACES



- The commands of a system may be represented by a graphical metaphor: a *desktop*.
 - Uses *icons* as proxies for objects.
 - Uses *folders* as containers. (FS: *directory*)
- Alternatively, a mobile device might use touch gestures.
- How do pointer and touch differ in usage? (skipping multi-touch.)



CLI VERSUS GUI

- Why use a CLI instead of a GUI?
- Conversely, why use a GUI instead of a CLI?

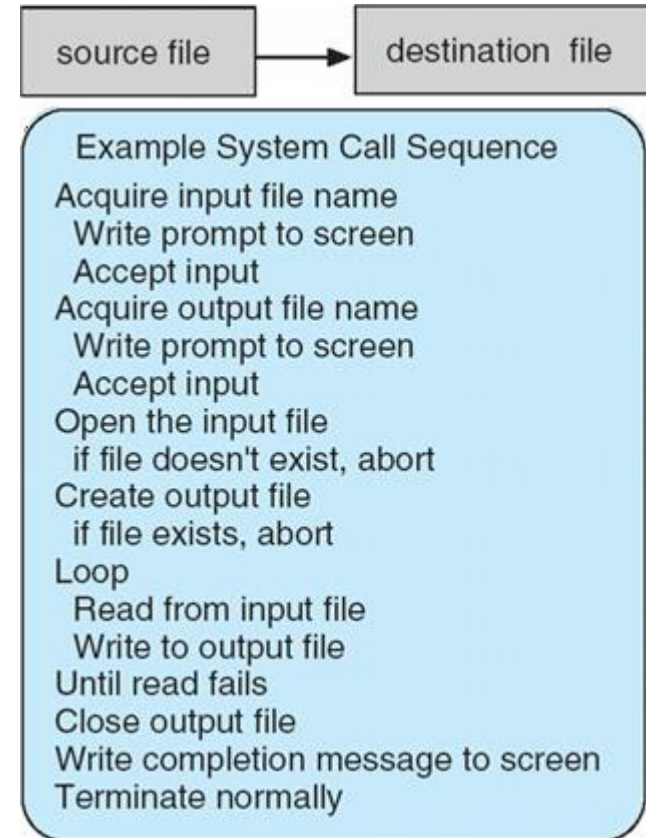


SYSTEM CALLS

Silberschatz. Operating System Concepts (10e). Chapter 2.3.

THE CONCEPT

- A *system call* is a call that invokes some low-level subroutine that likely requires a high level of privilege such that the OS alone offers its service.
- Consider an example program that makes a copy of a file. How many system level calls are needed?
 - Fetch Parameters: read filename, display message
 - Open: open files, checks errors (display, terminate)
 - Copy: read from src, write to dest
 - Close: close files, display message, terminate



APIs: ABSTRACTION

- In general, we don't make system calls directly.
- Instead, an API is used: Windows API, POSIX API, Java VM API.
- APIs form a layer of **abstraction** (for some reason the book doesn't use this term...) over system calls.
- Sometimes APIs expose system calls with almost no change, just a different signature – so why bother?

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

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

return value	function name	parameters
--------------	---------------	------------

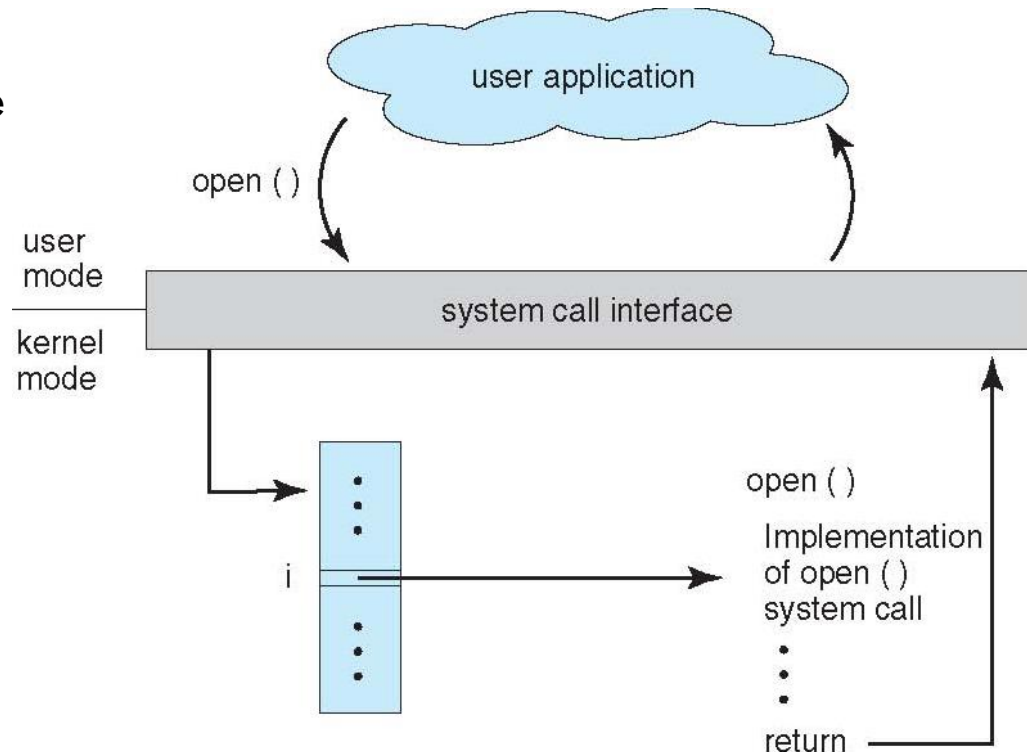
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

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

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

RUN-TIME EXECUTION

- Say you make an API call that triggers a system call from the *system-call interface*.
- Then that system call is invoked from the current *run-time support library*.
 - Why not bake the code for system calls right into the executable?





TYPES OF SYSTEMS CALLS

Silberschatz. Operating System Concepts (10e). Chapter 2.3.

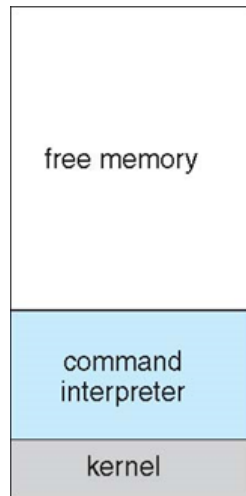
OVERVIEW

- Recall that a system call that requires some high level of privilege that only the operating-system's kernel will have.
- Often times, this looks like a function that requests a *resource allocation* (e.g., needs to print on a printer), which requires interacting with *low level hardware* (e.g., getting system time), or which are *potentially dangerous* (e.g., stopping another program).
- We will review over several typical classes of system calls:
 - Process control
 - File management
 - Device management
 - Information maintenance
 - Communications
 - Protection
 - (Note that this list is not comprehensive!)

PROCESS CONTROL

- In general, we say that a program's execution is a *process*.
- Processes encapsulate a program's state: data, program code, current instruction, resource allocation, etc.
- Processes also have some attributes: owner, priority, etc.
- Processes are permitted to create other processes.
 - Child processes may either run independently, or the parent may wait for them to terminate.
- Provided we have multiple processes, then we can imagine them competing over access to a limited resource: this necessitates the idea of a resource *lock* (acquire and release).

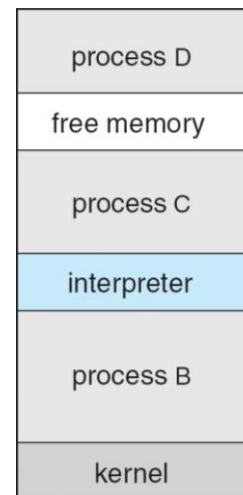
PROCESS AND JOB CONTROL



Why bother making the interpreter smaller? BSD doesn't.

- MS-DOS: single task.
- Normally, interpreter is running. When user wants to execute program, then a smaller “bootstrap” interpreter is loaded and program is moved into memory.

- BSD: supports multi-tasking.
- Whenever a program is run, then it is loaded into memory.
- Requires notation of *background* task.



How abstract is this image?

FILE MANAGEMENT

- *File Management*: systems typically have some kind of persistent file store. The store contains files and directories, with some attributes.
- Typical operations: `create()`, `delete()`, `read()`, `write()`, `reposition()`, `get_file_attributes()`, `set_file_attributes()`.
- Higher level functionality may also be provided: `move()`, `copy()`.

DEVICE MANAGEMENT

Such
as?

- Device Management: any system has some set of devices external to main memory which make resources available to executing programs.
- The book draws the distinction between physical and abstract/virtual devices. Hmm.
- Typical operations: `read()`, `write()`, `reposition()`.

Is this enough
to do work?

THE POWER OF READ(), WRITE(), AND REPOSITION()

- At a glance, it might seem like these three functions can't really do a lot. Not so!
- Consider two simplifications of these concepts:
 - POKE address, value (set a value in mem)
 - PEEK address (read a value in mem)
- These functions are enough for us to do something like implement drawing on a screen - provided VRAM is “memory-mapped”.

0x00

Physical Memory

Mapped VRAM

INFORMATION MAINTENANCE

```
Terminal - ruben@ruben-VirtualBox: ~/Desktop
```

```
File Edit View Terminal Tabs Help

ruben@ruben-VirtualBox:~/Desktop$ strace ./a.out
execve("./a.out", ["/a.out"], [/ 65 vars */]) = 0
brk(NULL)                                = 0x130b000
access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f793c854000
access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|[O_CLOEXEC] = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=81304, ...}) = 0
mmap(NULL, 81304, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f793c840000
close(3)                                 = 0
access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|[O_CLOEXEC] = 3
read(3, "\177ELFv2\1\13\0\0\0\0\0\0\0\0\0\0\3\0-\0\1\0\0\0P\t2\0\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1864888, ...}) = 0
mmap(NULL, 3967488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f793c268000
mprotect(0x7f793c268000, 2093056, PROT_NONE) = 0
mmap(0x7f793c270000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0x7f793c270000
mmap(0x7f793c2d0000, 14848, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f793c2d0000
close(3)                                 = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f793c83f000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f793c83e000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f793c83d000
arch_prctl(ARCH_SET_FS, 0x7f793c83e700) = 0
mprotect(0x7f793c270000, 16384, PROT_READ) = 0
mprotect(0x6000000, 4096, PROT_READ)     = 0
mprotect(0x7f793c856000, 4096, PROT_READ) = 0
munmap(0x7f793c840000, 81304)            = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
brk(NULL)                                = 0x130b000
brk(0x132c000)                           = 0x132c000
write(1, "\n", 1)                         = 1
write(1, "\n", 1)                        = 1
write(1, "Welcome to ASU Class Schedule\n", 30>Welcome to ASU Class Schedule
) = 30
write(1, "\n", 1)                       = 1
write(1, "Menu Options\n", 13>Menu Options
) = 13
write(1, "-----"..., 55>-----
) = 55
write(1, "a: Add a class\n", 15>a: Add a class
) = 15
write(1, "d: Drop a class\n", 16>d: Drop a class
) = 16
write(1, "s: Show your classes\n", 21>s: Show your classes
) = 21
write(1, "q: Quit\n", 8>q: Quit
) = 8
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
write(1, "Please enter a choice --> ", 27>Please enter a choice ---> ) = 27
```

```

top - 07:50:08 up 176 days, 21:07, 1 user, load average: 1.03, 1.26, 1.65
Tasks: 138 total, 2 running, 136 sleeping, 0 stopped, 0 zombie
%Cpu(s): 12.5 us, 0.0 sy, 0.0 ni, 87.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 2064676 total, 1483988 used, 580688 free, 195476 buffers
KiB Swap: 2101244 total, 10112 used, 2091132 free, 919364 cached

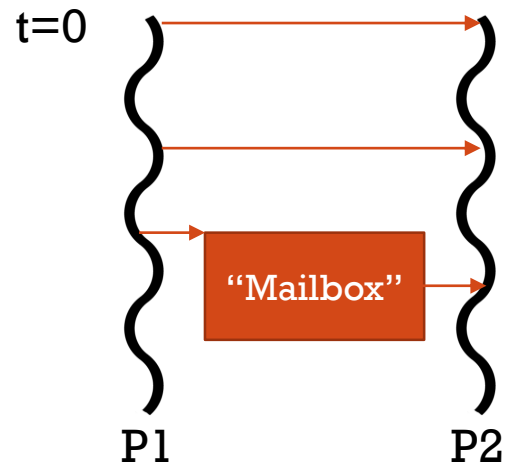
  PID USER      PR  NI  VIRT  RES  SHR S  %CPU  %MEM    TIME+  COMMAND
 5903 ruben    20   0 18316 6996  696 R 100.1   0.3   350:03.99 mir_rpbs.exe.li
    1 root      20   0 3776 1752 1184 S   0.0   0.1    0:01.06 init
    2 root      20   0   0   0   0 S   0.0   0.0    0:00.66 kthreadd
    3 root      20   0   0   0   0 S   0.0   0.0    0:56.77 ksoftirqd/0
    4 root      20   0   0   0   0 S   0.0   0.0    0:00.00 kworker/0:0
    5 root      0 -20   0   0   0 S   0.0   0.0    0:00.00 kworker/0:0H
    6 root      20   0   0   0   0 S   0.0   0.0    0:00.02 kworker/u:0
    7 root      0 -20   0   0   0 S   0.0   0.0    0:00.00 kworker/u:0H
    8 root      rt   0   0   0   0 S   0.0   0.0    0:08.22 migration/0
    9 root      20   0   0   0   0 S   0.0   0.0    0:00.00 rcu_bh
   10 root      20   0   0   0   0 S   0.0   0.0    5:16.16 rcu_sched
   11 root      rt   0   0   0   0 S   0.0   0.0    1:51.23 watchdog/0
   12 root      rt   0   0   0   0 S   0.0   0.0    1:41.38 watchdog/1
   13 root      20   0   0   0   0 S   0.0   0.0    0:51.30 ksoftirqd/1
   14 root      rt   0   0   0   0 S   0.0   0.0    0:07.75 migration/1
   16 root      0 -20   0   0   0 S   0.0   0.0    0:00.00 kworker/1:0H
   17 root      rt   0   0   0   0 S   0.0   0.0    1:40.62 watchdog/2

```

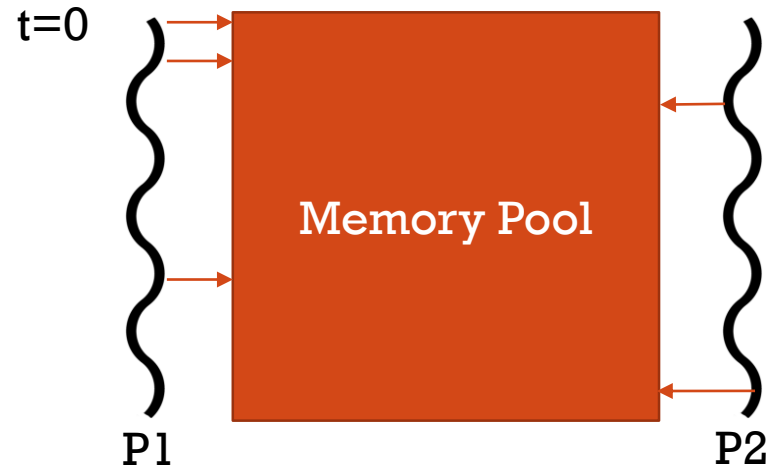
- Sometimes, we just need to get system information. Applications:
 - *top* – show currently running processes and resource allocation.
 - *strace* – show system calls for invocation of program.
 - Profilers – show function level performance of a program.

COMMUNICATION AND PROTECTION

- Message-Passing Model



- Shared Memory Model



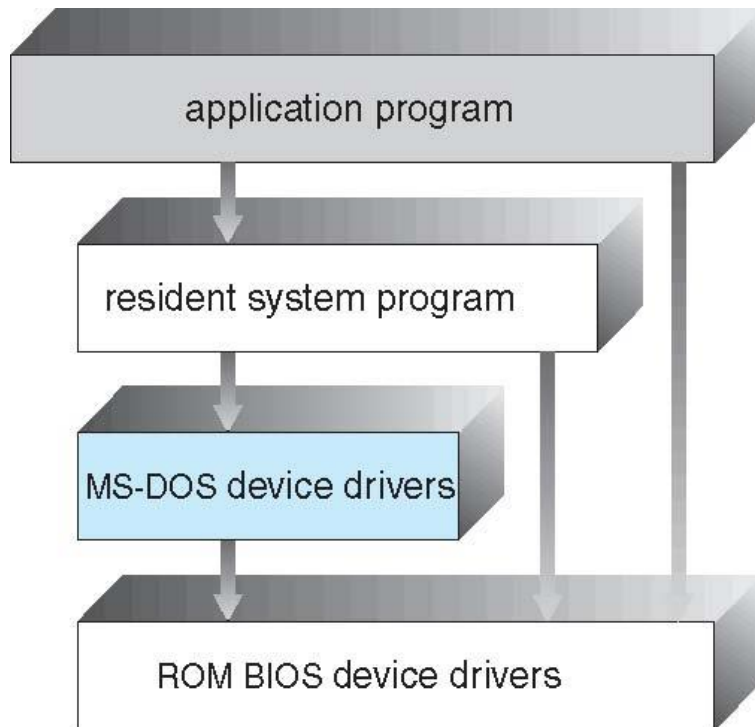
- The last area is protection – thoughts on what type of operations would be system calls?



OPERATING-SYSTEM STRUCTURE

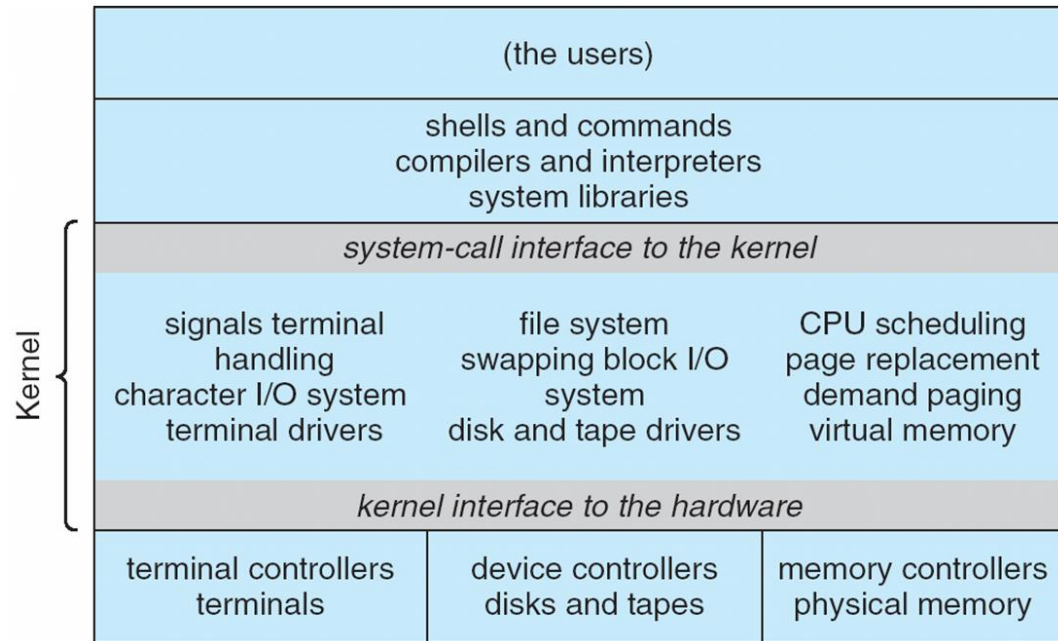
Silberschatz. Operating System Concepts (10e). Chapter 2.8.

ORGANIC (SIMPLE STRUCTURE)



■ MS-DOS

- Almost looks well organized – what's the issue?

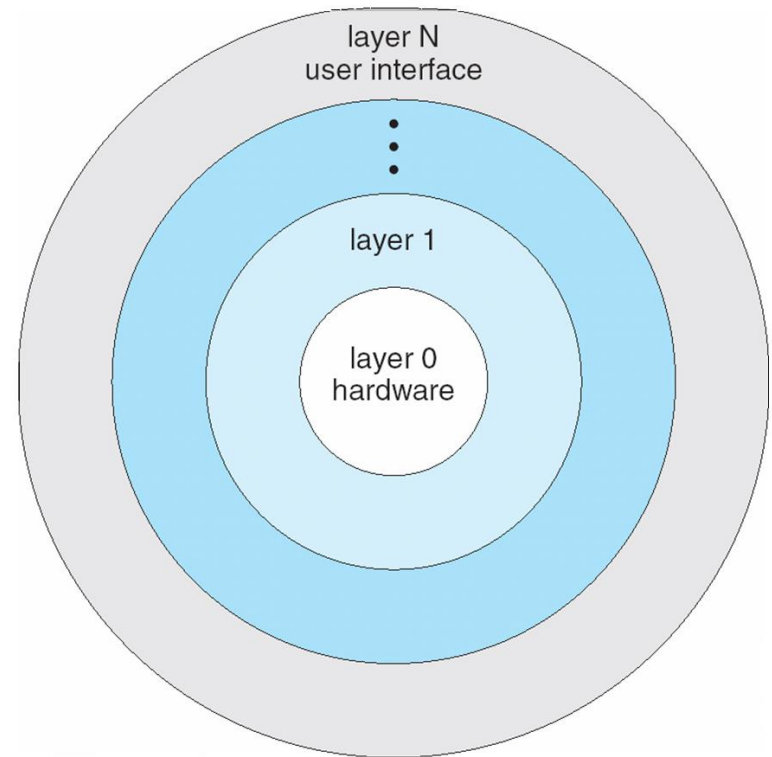


■ UNIX

- Same order, but monolithic (see middle).
- Pros? Cons?

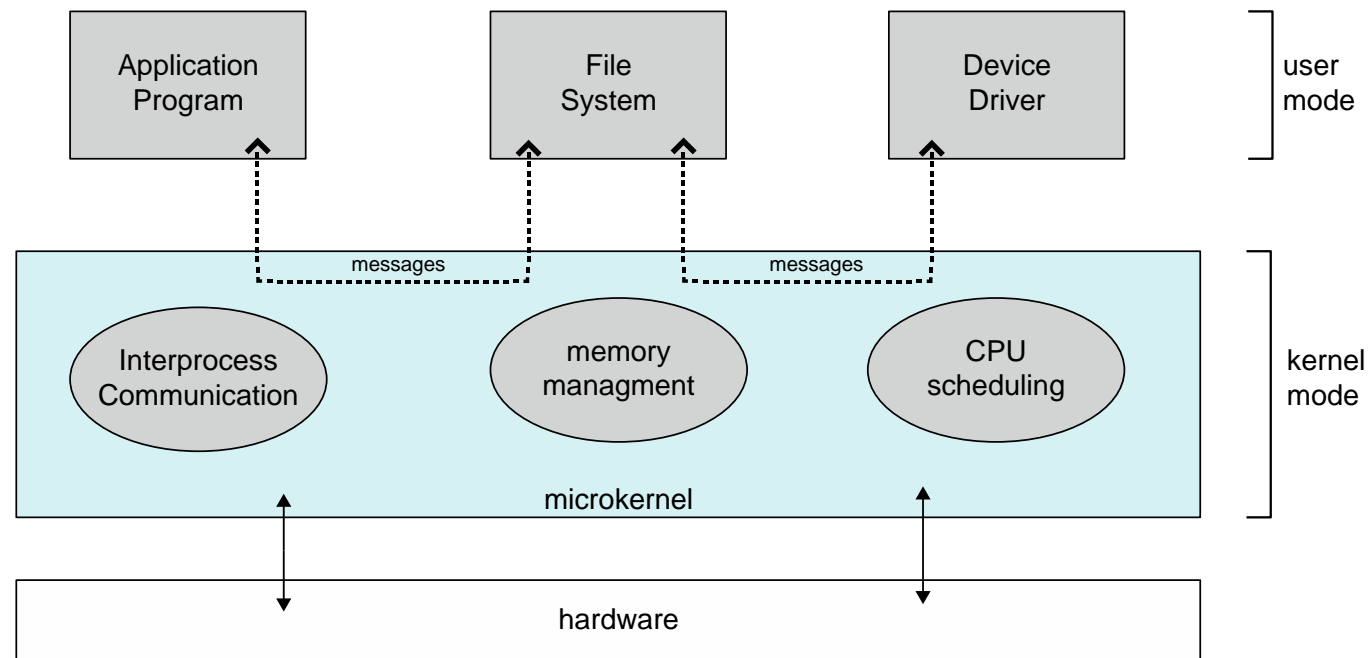
LAYERED

- Obviously organically designed systems are hard to maintain – we can do better.
- In an explicitly layered system, each k th layer (in 0 to N) may only call functions defined by lower layers.
 - Pros?
 - Cons?



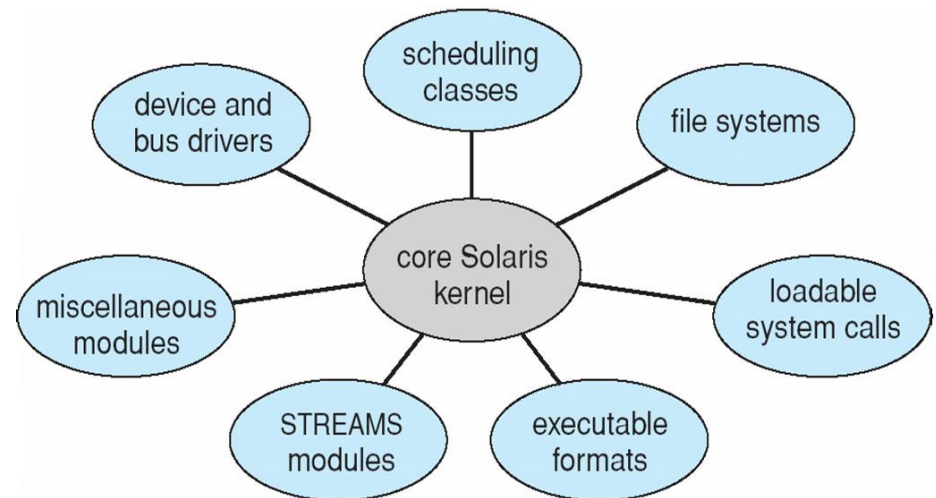
MICROKERNELS

- Explicit separation of base kernel functionality from system programs.
- Communication is via message passing.
- Typically manages processes, memory, and communication (IPC).
 - Pros?
 - Cons?

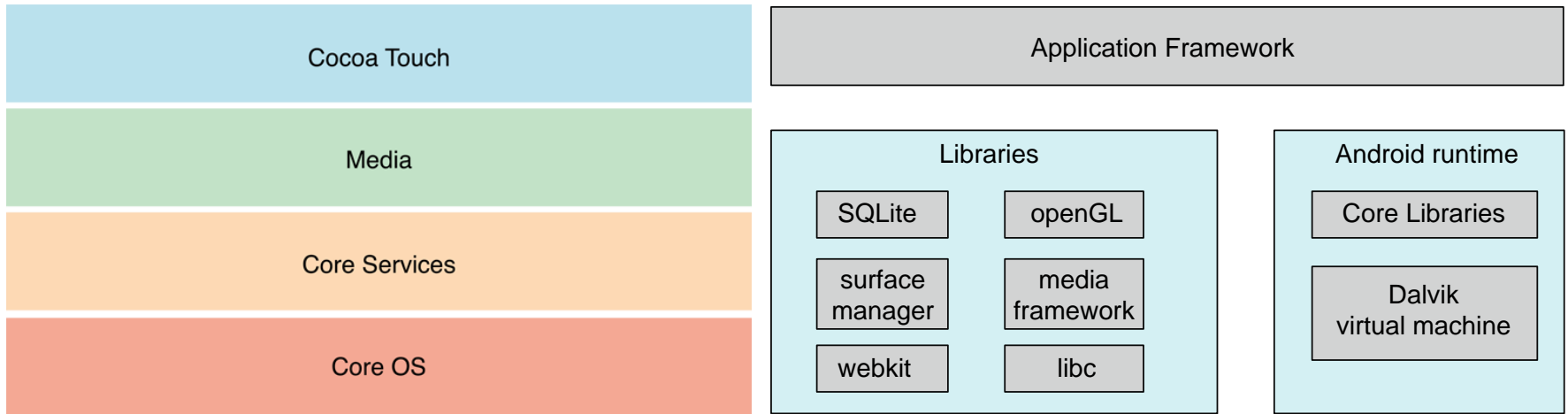


MODULES

- Design aims to support dynamically loading modules during start up or run-time to add functionality to a base kernel.
- Key difference from microkernel is that modules can communicate directly, instead of via the kernel.



HYBRID



- iOS
 - Layered System, different focuses though.

- Android
 - Hybrid