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*

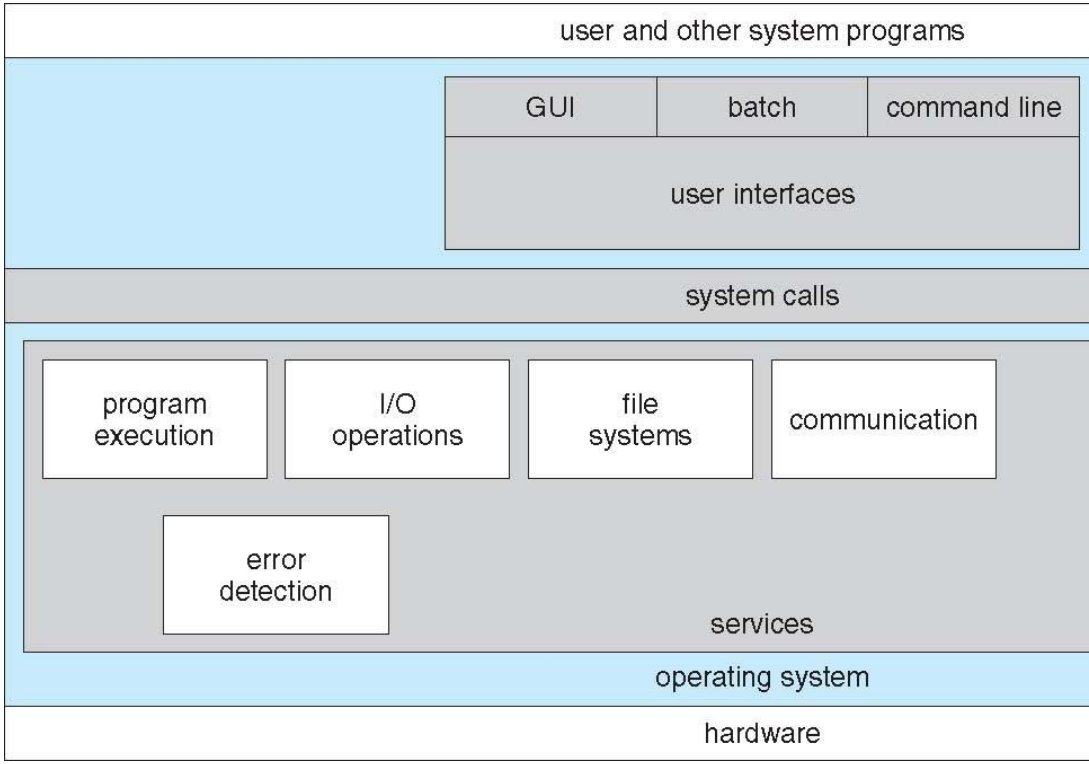
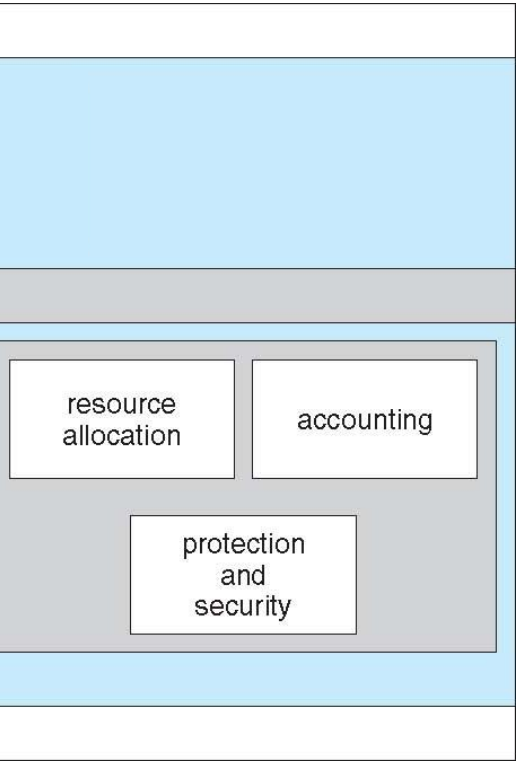
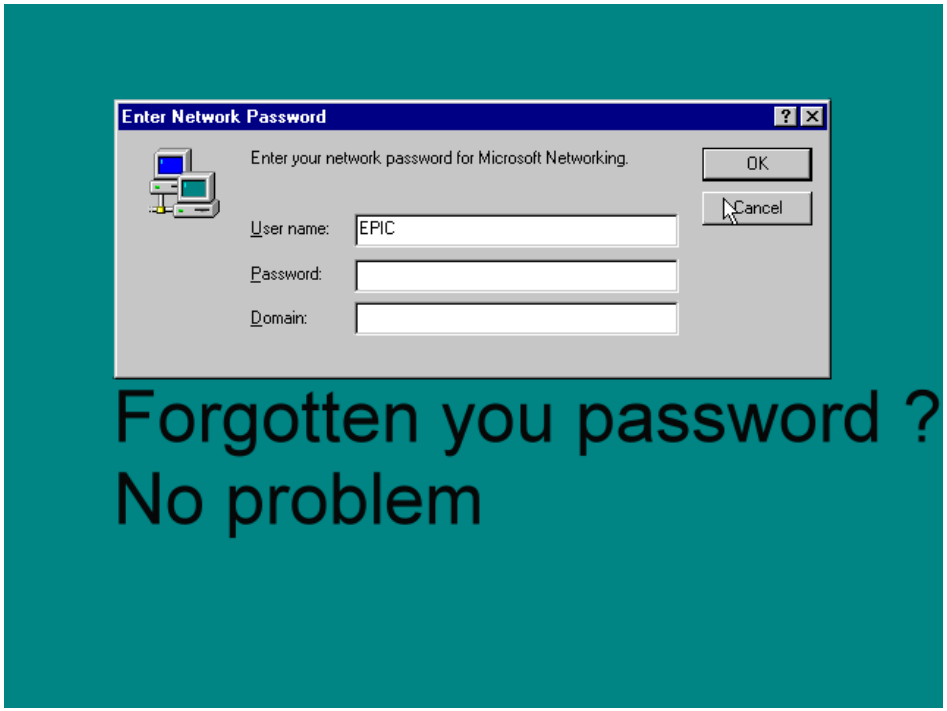


Image from Operating System Concepts.

OPERATION MANAGEMENT



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



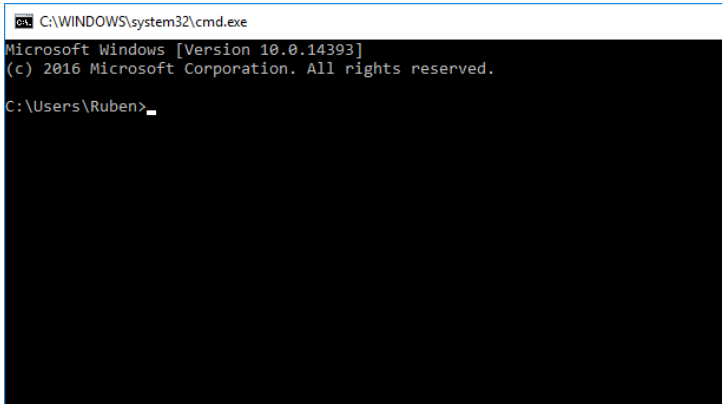
Left image from Operating System Concepts, right image from Unknown.

USER AND OPERATING-SYSTEM INTERFACE

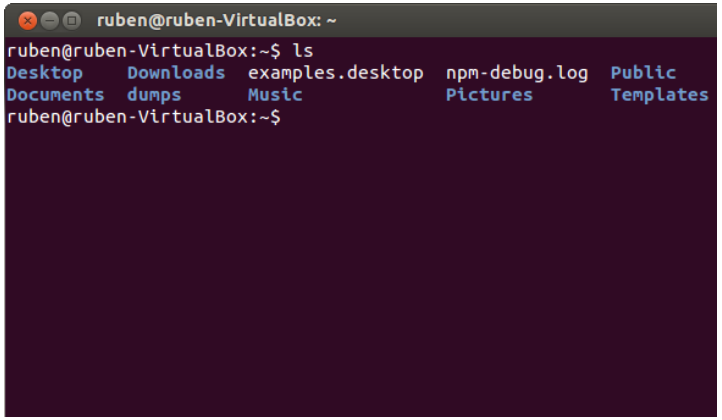
Silberschatz. Operating System Concepts (10e). Chapter 2.2.

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

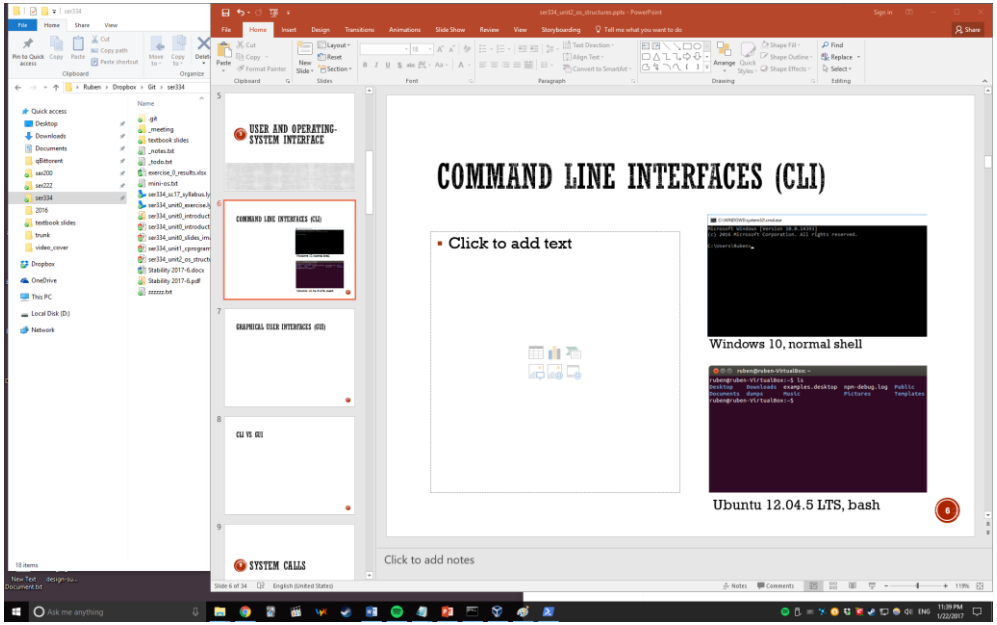


Windows 10, normal shell



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.)

iOS image from Operating System Concepts.

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

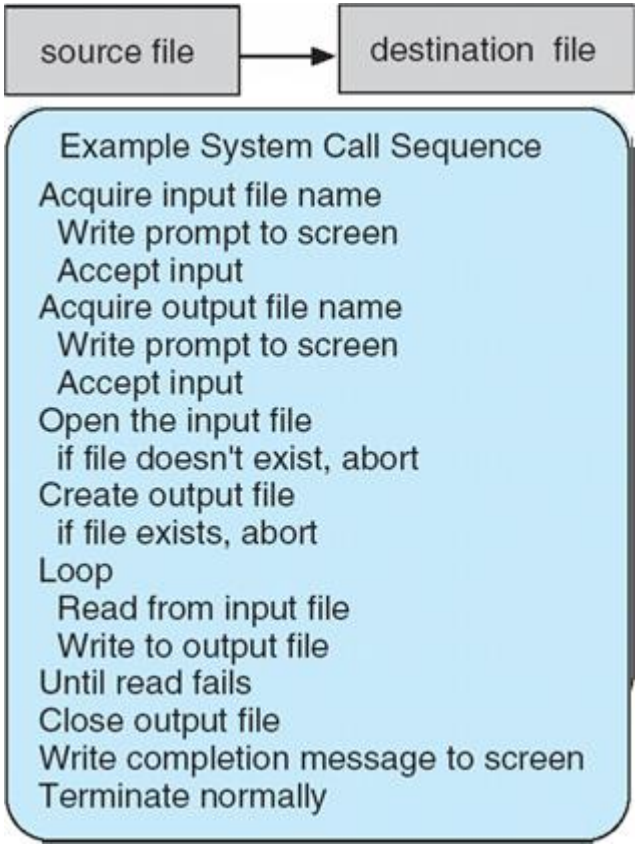


Image from Operating System Concepts.

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.

Image from Operating System Concepts.

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?

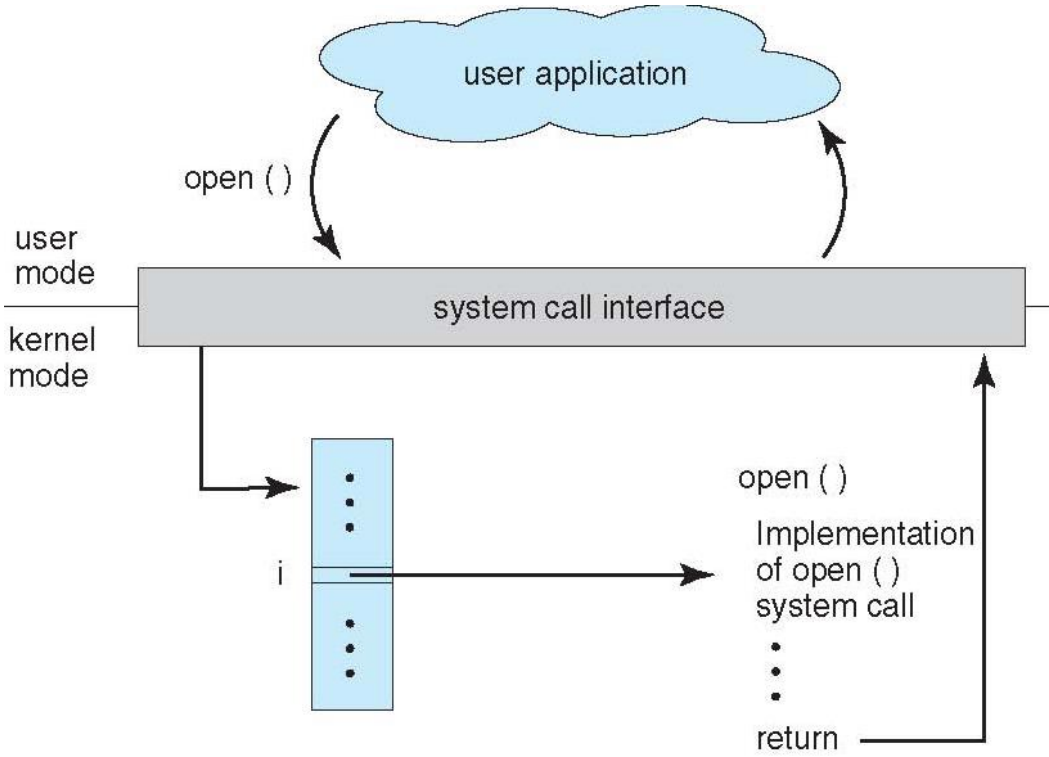


Image from Operating System Concepts.

TYPES OF SYSTEMS CALLS

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

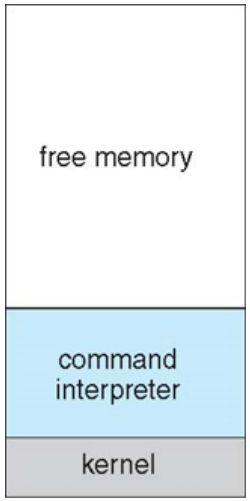
OVERVIEW

- 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).

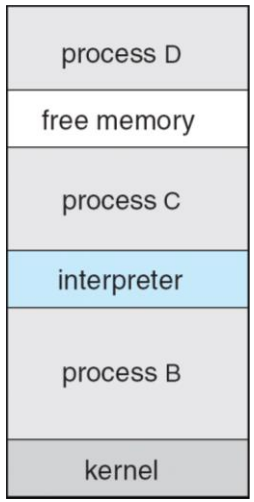
PROCESS AND JOB CONTROL



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

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.



How abstract is this image?

Images from Operating System Concepts.

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

- 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().

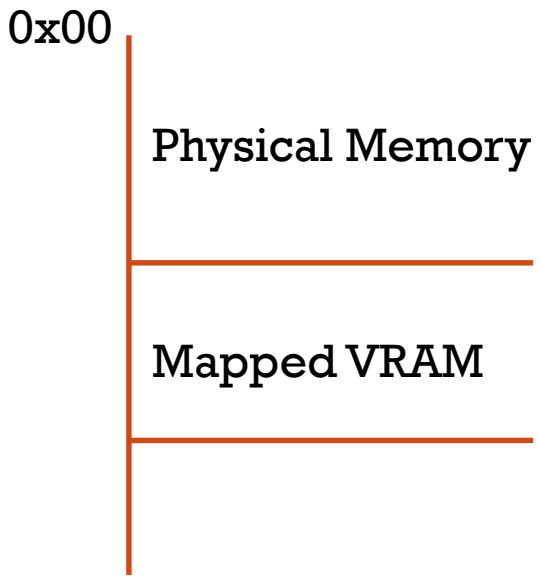
Such as?

Is this enough to do work?

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

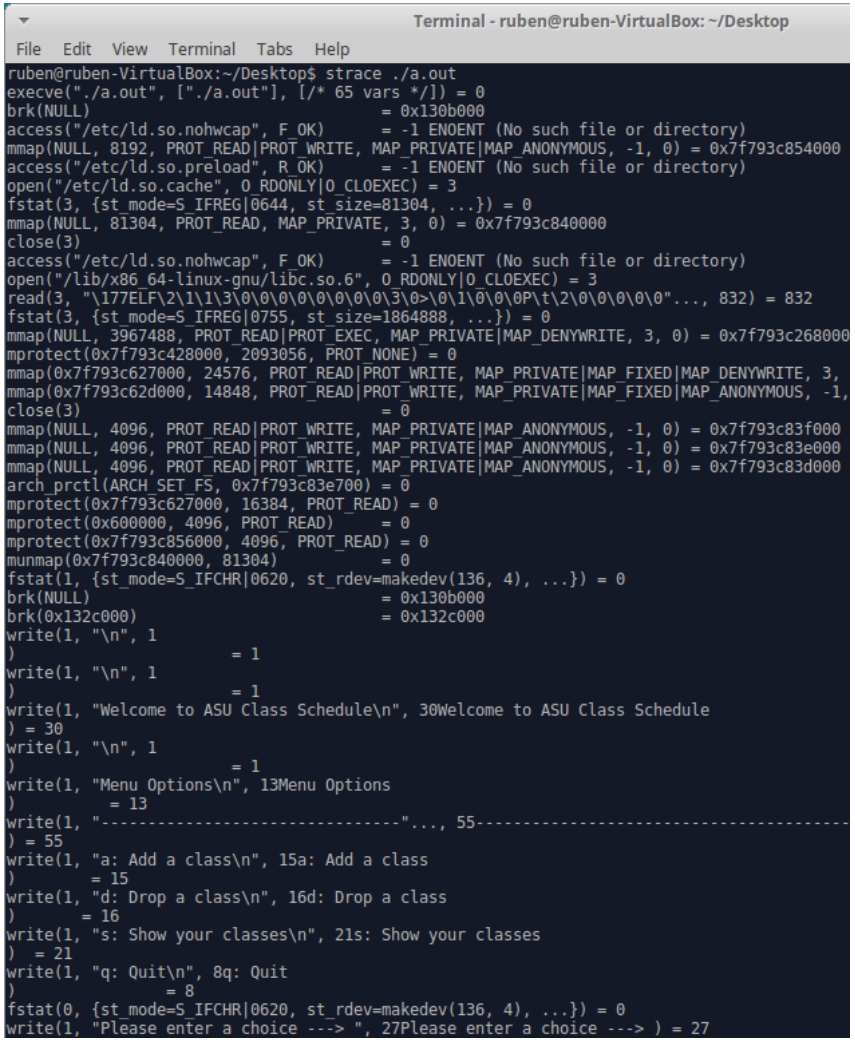
- At a glance, it might seems like these three functions can't really do a lot. Not so!

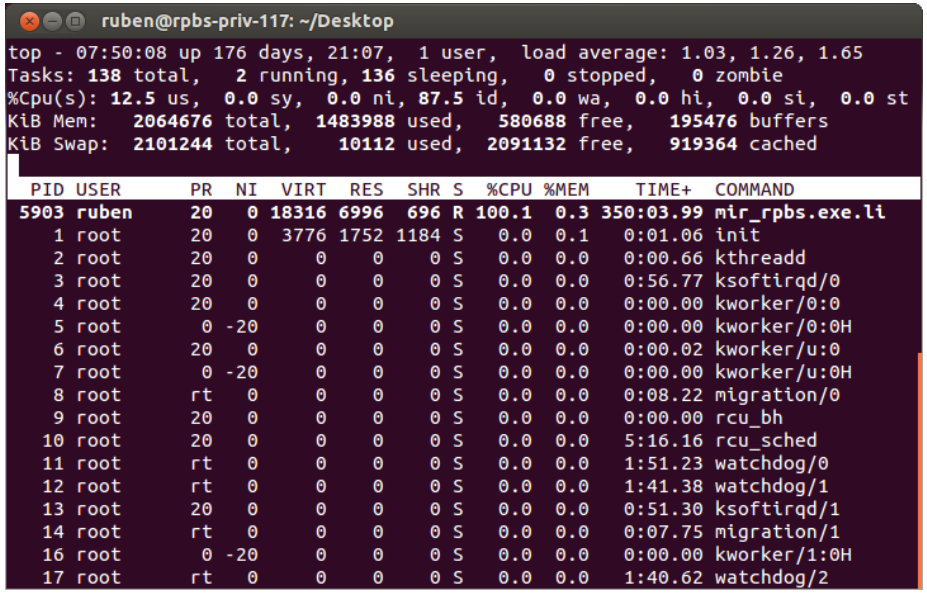
- Consider two simplifications of these concepts:
  - POKE address, value (set a value in mem)
  - PEEK address (read a value in mem



- These functions are enough for us to do something like implement drawing on a screen - provided VRAM is "memory-mapped".

INFORMATION MAINTENANCE

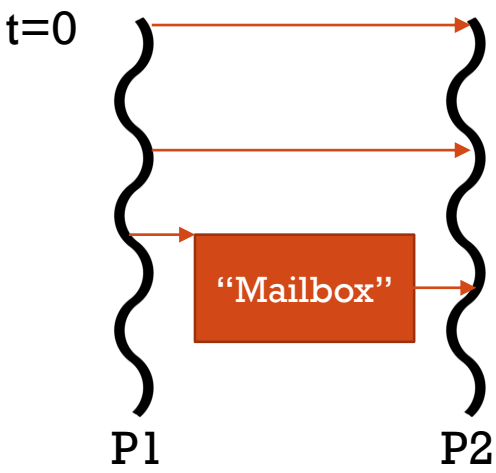




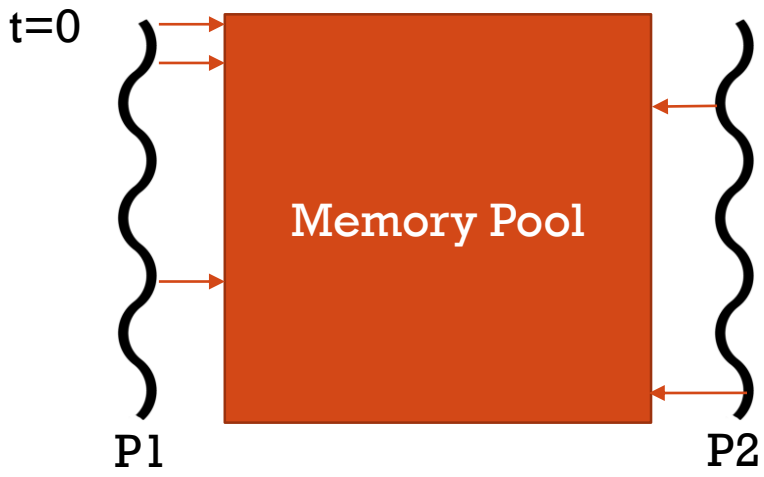
- Sometimes, we just need to get system information. Applications:
  - *top* – show currently running processes and resource allocation.
  - *strace* – show system calls for invocation of program.
  - Profilers – show function level performance of a program.

COMMUNICATION AND PROTECTION

- Message-Passing Model



- Shared Memory Model

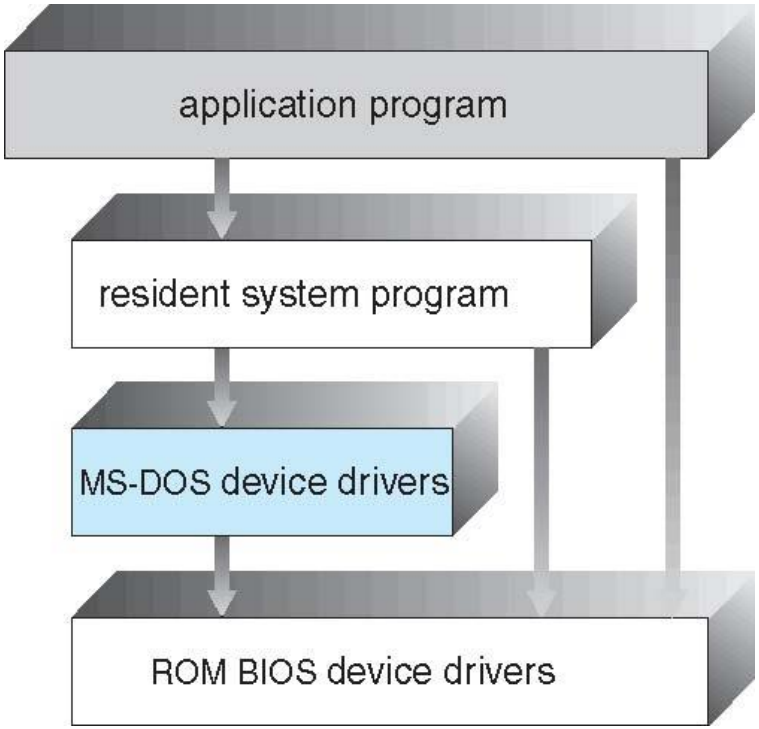


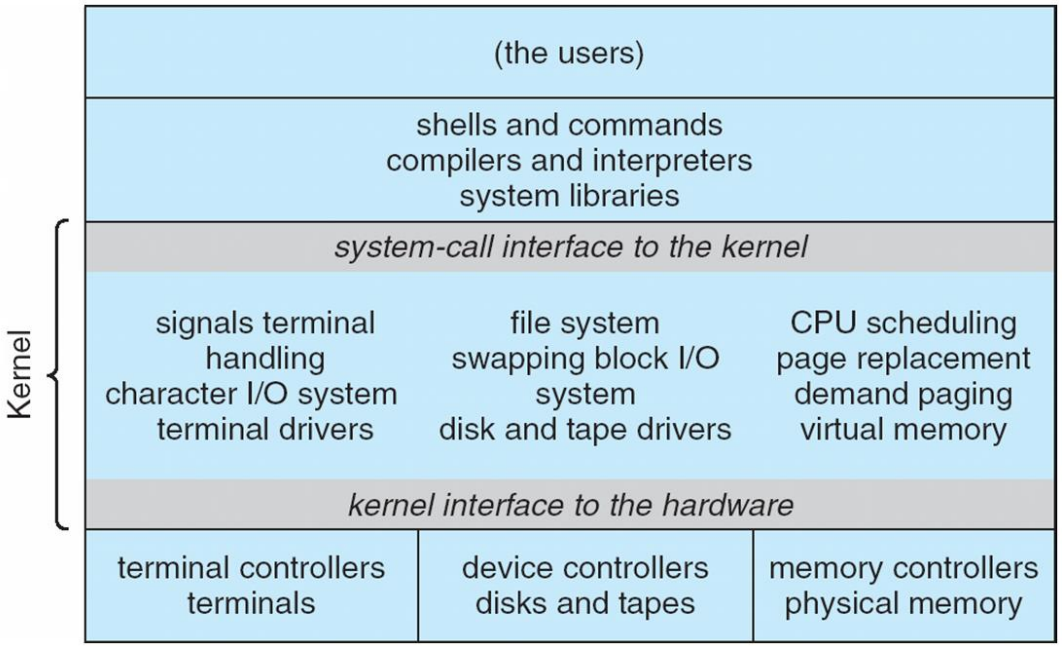
- The last area is protection – thoughts on what type of operations would be system calls?

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?

Images from Operating System Concepts.

LAYERED

- Obviously organically designed systems are hard to maintain – we can do better.

- In an explicitly layered system, each kth layer (in 0 to N) may only call functions defined by lower layers.
  - Pros?
  - Cons?

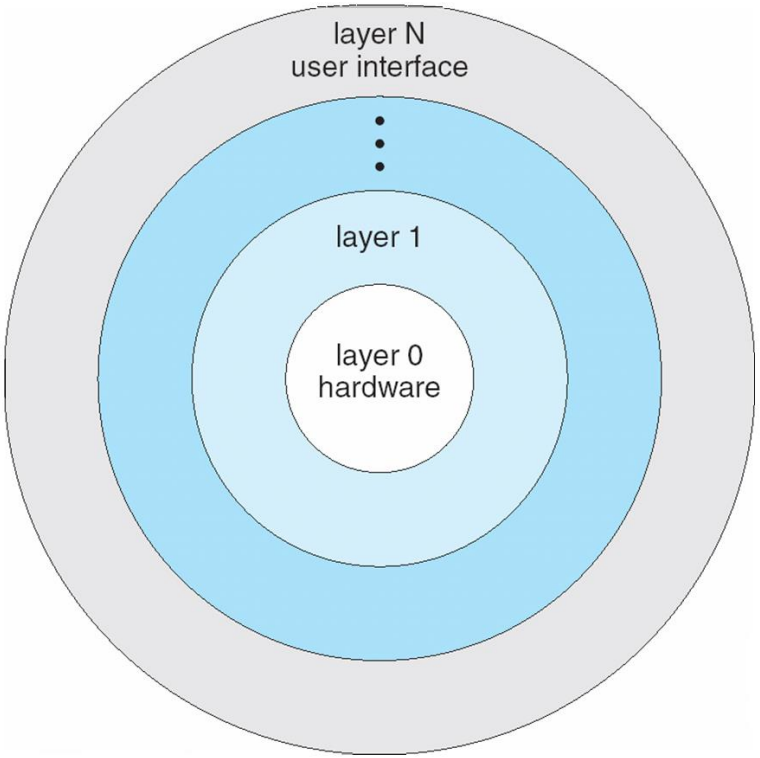


Image from Operating System Concepts.

MICROKERNELS

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

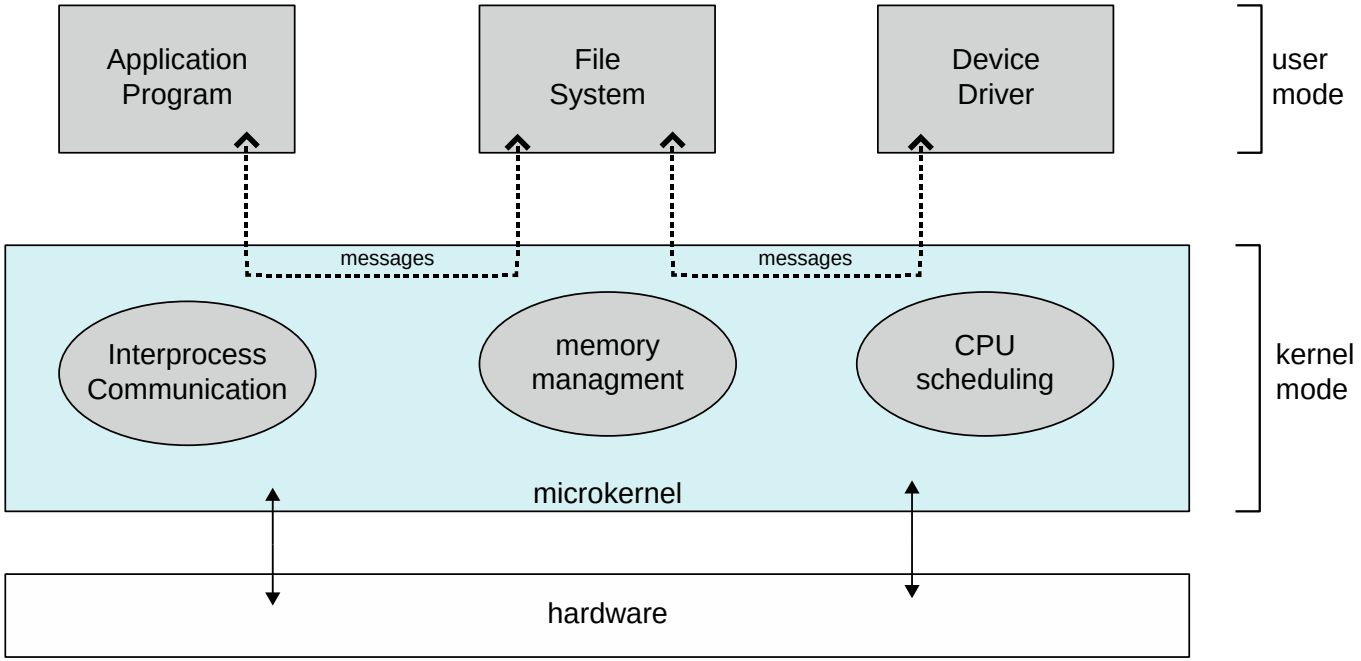


Image from Operating System Concepts.

MODULES

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

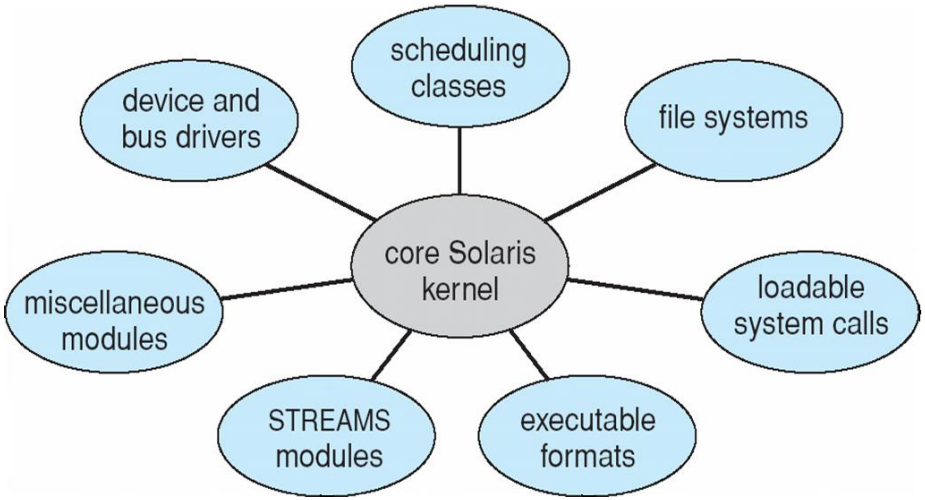
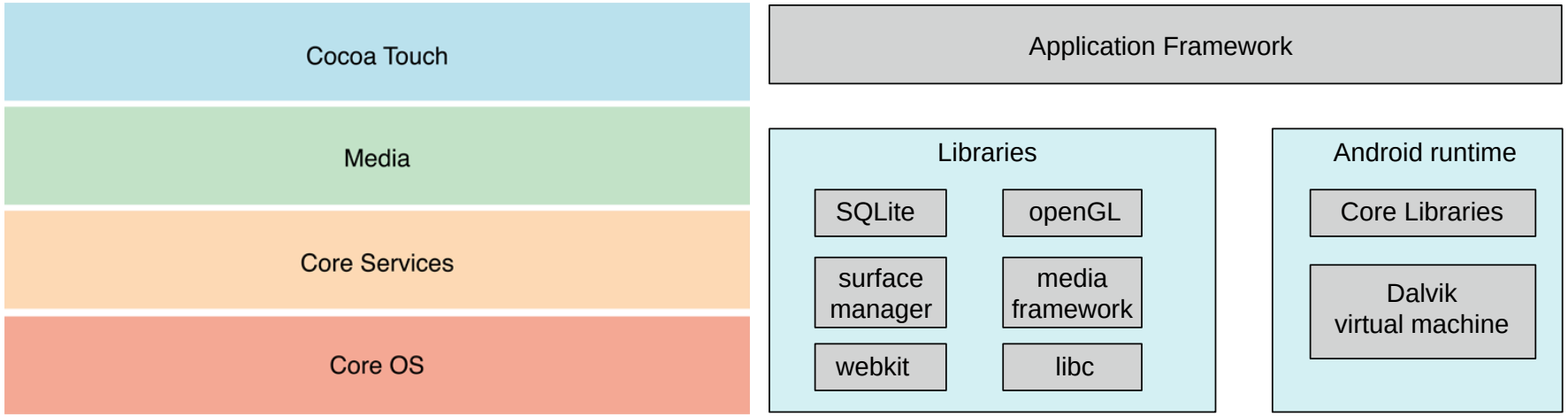


Image from Operating System Concepts.

HYBRID



- iOS
  - Layered System, different focuses though.
- Android
  - Hybrid

Images from Apple and Operating System Concepts.