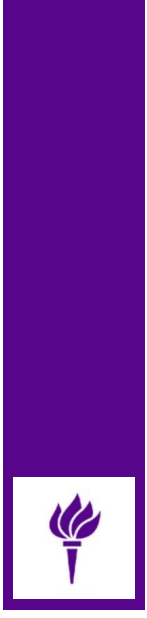


OS Structures & Services

Alex Delis
NYU-Abu Dhabi

January ’21

Operating Systems Services

■ An OS offers an environment for execution of programs and provides services to both programs and users.

⇒ Services that are of paramount importance to users:

- UI (user interface): all OSs offer some form of:
  - CLI: command line.
  - GUI: Graphics User Interface.
  - Batch: through the `tty`.
- Program Execution: An OS must be able to load a program into memory, run it, produce output, and end its execution either normally or abnormally (indicating error).
- I/O operations: an program in execution will inevitably require IOs to file/devices.

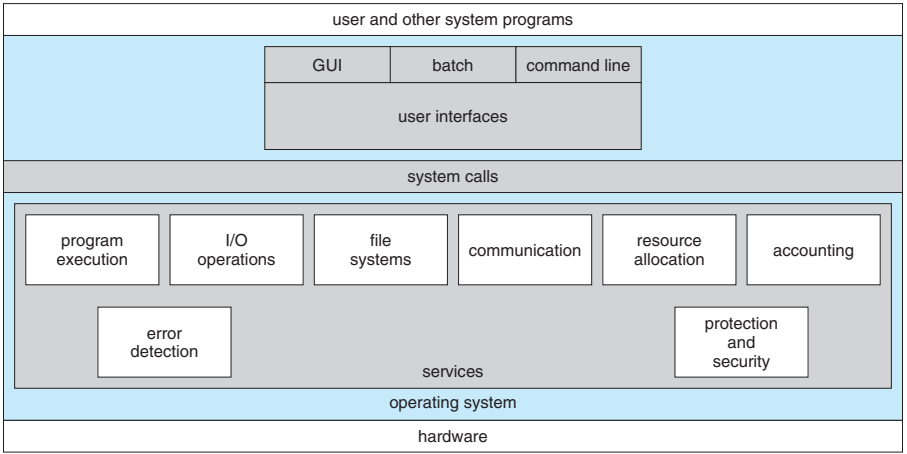
Services Available (and Visible) to all Users

- File-System Services:
  - Programs need to `create, delete, write, read, append, search, obtain-info, get&set-privileges` for both files and directories.
- Communication Services:
  - Help exchange messages within the same machine and among machines over a network.
  - Services are implemented through shared memory or messages.
- Error Detection: OS should be aware of emerging issues.
  - May occur in CPU, memory, I/O devices, and in user programs.
  - For every type of error, the OS should take a corrective action, if possible.
  - Debugging greatly enhances a programmer's abilities to effectively use the system.

OS Services Hidden to Users

- Hidden services ensure smooth operation of OS.
- Resource Allocation Services:
  - Numerous resources shared among users: CPU cycles, m.m., FileSys, I/O devices.
  - When multiple jobs run simultaneously, limited resources have to be split up equitably.
- Accounting Services:
  - keep track of which users use how much and what resource type.
- Protection & Security Services: multiple users do interfere with each other.
  - Protection is about offering controlled access to all resources.
  - Security from outsiders calls for authentication and mechanisms that help defend IO-devs from illegitimate attempts from outside access.

OS Services as parts of the OS



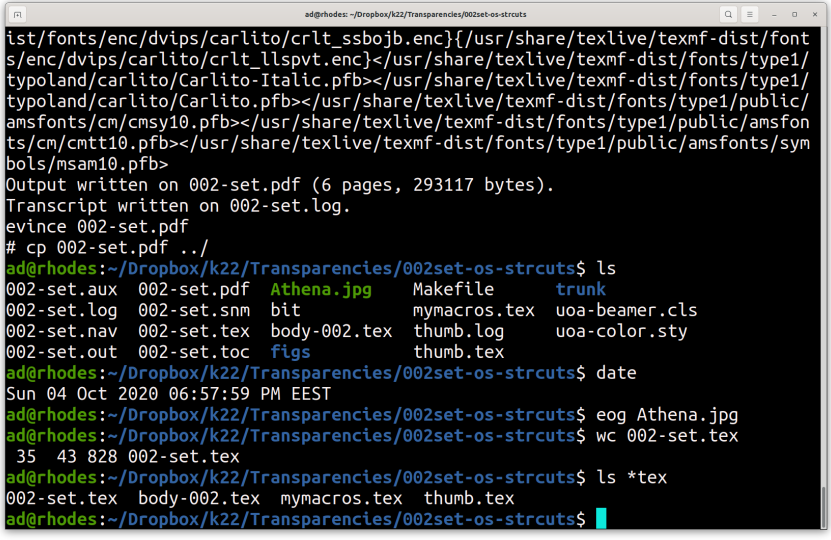
- All services are *exposed* beyond the OS through System Calls.

UI of OSs: CLI

■ Command line interpreter (CLI) allows for direct interaction with the services and system program of an OS.

- Very often a CLI is a system program and occasionally is implemented in the kernel.
- Often multiple flavors are implemented: `shells`.
- Main objective of a CLI: fetch an command from user, parse it and execute it.
- Individual commands can be either independent programs (flexible) or part of the CLI (less flexible, more traditional approach).

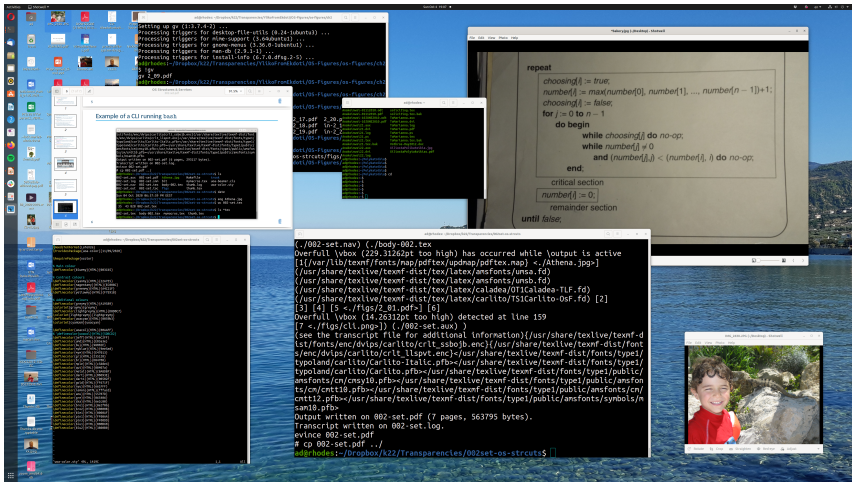
Example of a CLI running `bash`



Graphical User Interface for an OS

- User-friendly Desktop:
  - Involves keyboard, mouse, monitor.
  - *Icons* represent files, programs, and assorted actions.
  - Mouse buttons over objects in UI cause various actions such as provide information, options, execute function, open directory, etc.
  - Invented at Xerox PARC in the early 80s.
- Key OSs all offer CLIs and GUIs:
  - Microsoft Wins is a GUI with CLI command shell.
  - Linux/Unix have CLIs with optional (for workstations) GUIs such as *KDE*, *Gnome*. *xfc*, *cde*, *wayland*, etc.

Example of a Gnome Interface

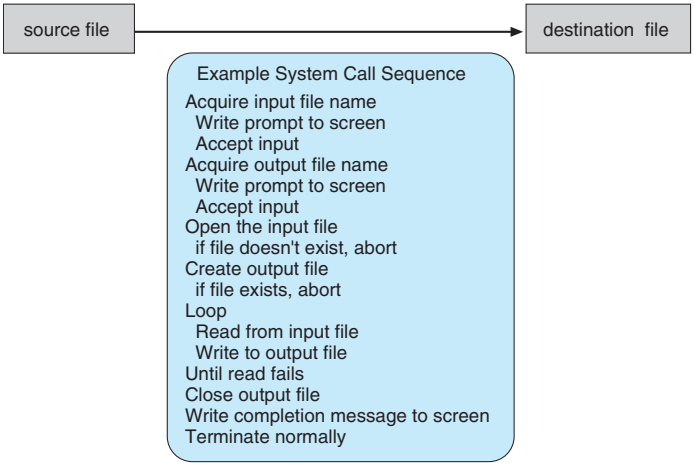


OS Enabler: System Calls

- System Calls provide an interface between a running program the OS.
  - Typically now written in a high level language (`C/C++`); in earlier times in `Assembly`, `Bliss` or `PL/360`.
  - System Calls are accessed/utilized by programs via a high-level Application Programming Interface (API) rather than direct system calls to the kernel.
  - 3 widely used APIs are:
    - *Win32 API* for Windows,
    - *POSIX API* for *POSIX*-based systems for all Unix-variants and Mac OS,
    - *Java API* for the Java virtual machine - *JVM*.

Sequence of Systems Calls

- Calls that are to be issued while realizing the sys. program: `cp fileA fileB`



The `read()` Systems Call

■ In Unix/Linux, an omnipresent system call is that of `read()`.

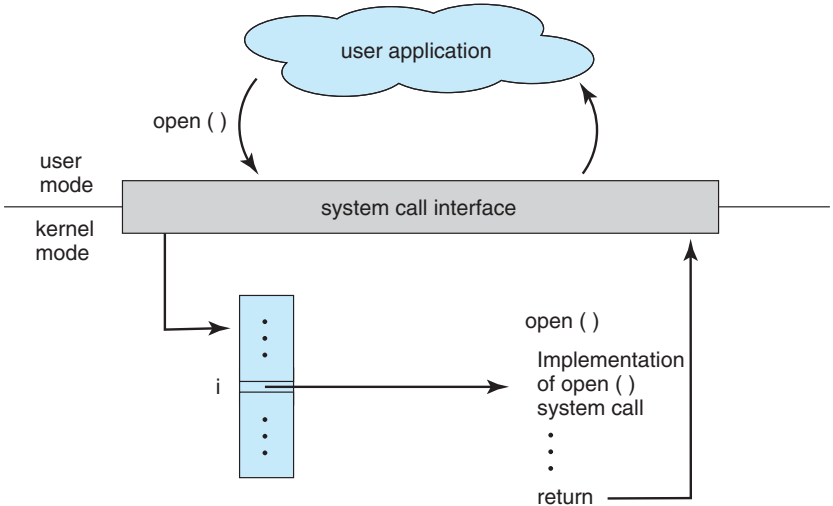
```
#include <unistd.h>

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

- The header `unistd.h` defines the `ssize_t` and `size_t`
- `fd` is a (low-level) file descriptor for a file.
- `buf` is the buffer allocated in the caller that will receive the data.
- `count` is the number of Bytes/character to be read into the `buff`.

■ On successful exit, the call returns the number of bytes read. A 0 points out an end of file, while -1 return value indicates that an error occurred.

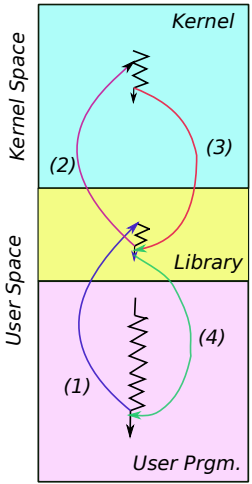
Handling of the System Call

There is a engagement between the user-space and the kernel-space



Library Call and Back

- The timeline for handling a library call is as follows:

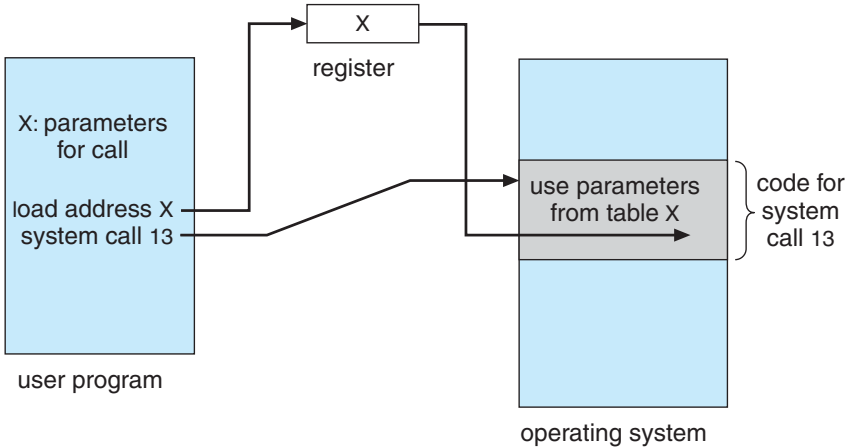


Issue: how do you Pass Parameters from a System Call to OS?

- There are 3 general ways to pass parameters from users-space to kernel.
  - Parameters are passed through registers (easiest).
  - Parameters are stored in a block, or table in memory and the address of block is passed as a parameter in a register (Linux).
  - Parameters are pushed on the the program stack and the OS pops them up to carry on the work.
- Why does a program always maintain a (run-time) stack?

Passing Parameters through a table/block

- A pointer to a block is passed through a board register.



Types of System Calls

- Broadly speaking, there are the following families of System Calls.
  - Process Control: to manage the execution of programs.
  - File System Management: to handle I/O requests.
  - Device Management: to help access devices.
  - Information Management: handle system data.
  - Communications: deal with data pipes.
  - Protection & Security: calls for authentication and thwarting outside attacks.

Process Control System Calls

- create process & terminate process
- gracefully end & abort
- load, execute
- get process attributes, set process attributes
- wait for a defined time period
- wait for an event or signal to occur
- allocate & free memory
- dump memory to a file if error
- offer locking primitives to coordinate access to shared data among multiple programs in execution.

Systems Calls for FS and Device Management

- File-System 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 (i.e., mount)

System Calls for Information Management and Communications

- Information Management:
  - 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 a communication connection (channel)
  - send, receive messages if message passing model to a host-name or a process-name
  - send, receive messages between a sever and a client.
  - If shared-model is adopted, create and gain access to memory regions.
  - transfer status information
  - attach and detach remote devices (i.e., work of `nfs`, `AndrewFS`, `Ceph`).

System Calls for Protection

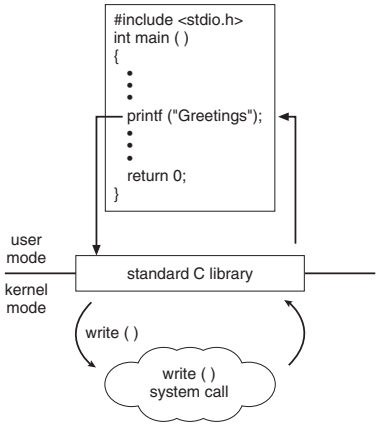
- Control access to resources
- Get and set permissions
- Allow and deny user access

Examples of of Selected 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()

Handling a `printf()` call

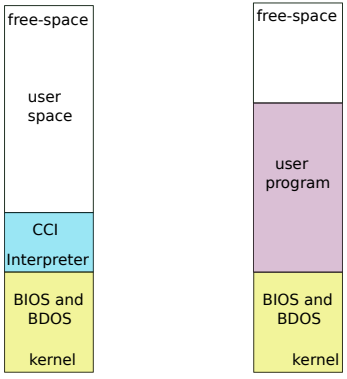
- A `C` program invoking a library call `printf()`



- The `printf()` is transformed to a `write()` sys-call with a set of params.

MS-DOS as an Example

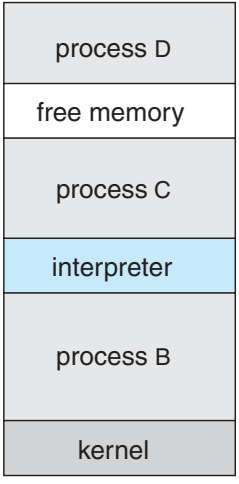
- MS-DOS is a single job at a time
- Shell is invoked when system booted
- No process created is created.
- Single memory space
- Loads program into memory, overwriting all but the kernel
- When program exited, shell reloaded



(a) Start-up and (b) Running a Program

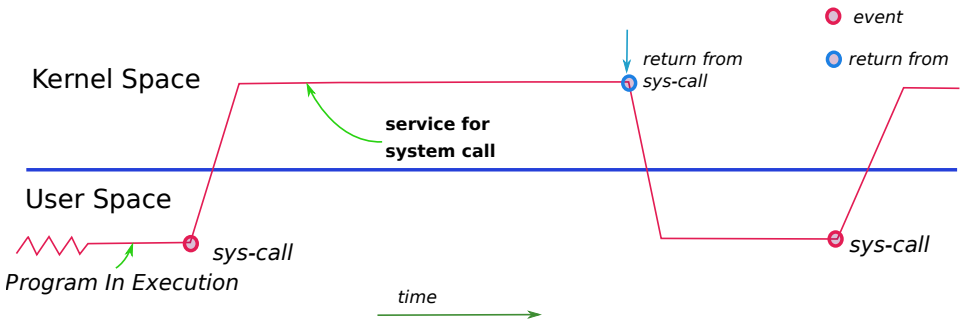
FreeBSD as an Example

- `FreeBSD` is a *Unix* variant.
- Supports multiple tasks.
- When User logins, the shell of choice is invoked.
- Shells execute `fork()` system call to create process
- Shell execute `exec()` to load (new) program images
- Shell waits for process to terminate and/or continues with user commands
- Process exits with code 0 for no-error and code $>$ 0 for some error.



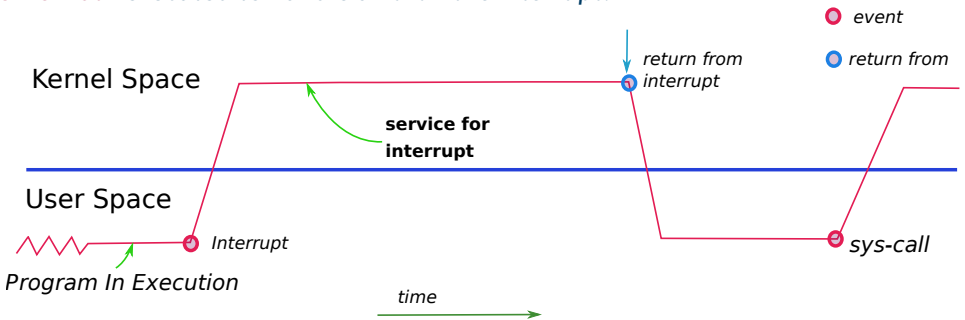
Kernel Execution Path when a System Call is Involved

- Kernel Path executed to handle a system call.



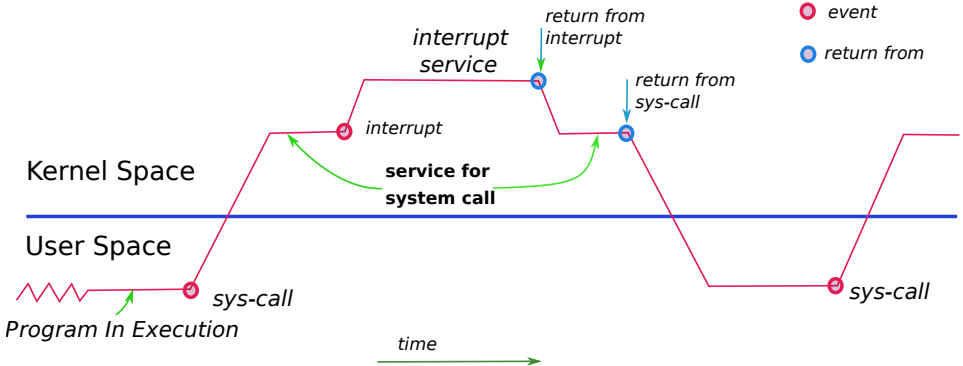
Kernel Execution Path when a Hardware Interrupt is Involved

- Kernel Path executed to handle a *hardware interrupt*.



Interleaved 2 Kernel Execution Paths

■ An Interleaved 2 Kernel Execution Path occurs when the kernel works for service a *system call* and an *hardware interrupt* occurs.



System Programs

■ System programs provide a convenient environment for program development and execution.

The are divided in groups or families:

- File manipulation
- Status information (occasionally stored in a File modification)
- Programming language support
- Program loading and execution
- Communications
- Background services
- Application programs

■ Most users view the OS as a collection of systems programs and they never get to see system calls!

System Programs

- File Management:
  - create, delete, copy, rename, print, catalogue, dump, list, and in general, manipulate files & directories
  - directories are sets of files and other directories.
- Status Information: programs that provide
  - date, time, amount of available memory, disk space, number of users
  - detailed performance, logging, and debugging information
  - format and print the output to the terminal or other output devices
  - offer a registry that is used to store and retrieve configuration information

(More) System programs

- File Modification
  - Text editors to create and modify files and cmds to search content of files
  - Commands to perform transformations of the text
- Programming-language Support
  - Compilers, Interpreters, Assemblers, and Debuggers.
- Program Loading and Execution
  - Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for machine language.
- Communications
  - Provide the mechanisms for creating virtual connections among processes, users, and networked computer systems.
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another, etc.

(More) System Programs

- Background Services:
  - Launched at boot time.
  - Some last during the system startup and then terminate, while others remain active until shutdown.
  - facilities including disk checking, process scheduling, error logging, printing (a.k.a., spooling).
  - Run in user context not kernel context and are known as *services*, *subsystems*, and *daemons*.

- Applications Programs
  - They are run by the users and do not pertain to the system.
  - Typically are not deemed part of OS
  - Launched by command line, mouse click, or finger poke.

OS Design Goals and Implementation

- User and Systems goals co-exist:
  - *User goals*: OS should be convenient to use, easy to learn, reliable, safe, and fast.
  - *System goals*: OS should be easy to design, implement, and maintain as well as flexible, reliable, error-free, and efficient!
  - OS is mostly affected by the choice of the underlying hardware.
- Key Principle: Policy vs. Mechanism
  - Mechanism: **How** to do it?
  - Policy: **What** has to to be done?
  - The separation of 2 is a key pillar in design for it allows later to change *Policy* aspects that do not quite work for a CS.
  - Examples: timer, buffering, program ageing, etc.
- OS implemented in Assembly Languages, `C`, `C++`, and scripting languages.
  - High level languages help as they are easier to port.
  - Emulation can also help for an OS to run on non-native hardware.

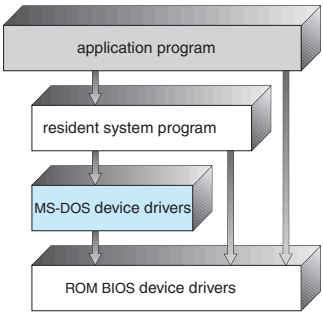
OSes of Various Flavors

- Various ways to structure OSes in the past have been followed:
  - `MS-DOS` has been generally straightforward.
  - *Unix has been more complex and flexible.*
  - *Layered-abstraction approach.*
  - *Microkernel architecture of* `Mach`.

Structure of MS-DOS

■ Key objective of MS-DOS was to offer the most functionality in the least space.

- Not divided in modules.
- Its levels and interfaces are not well separated.

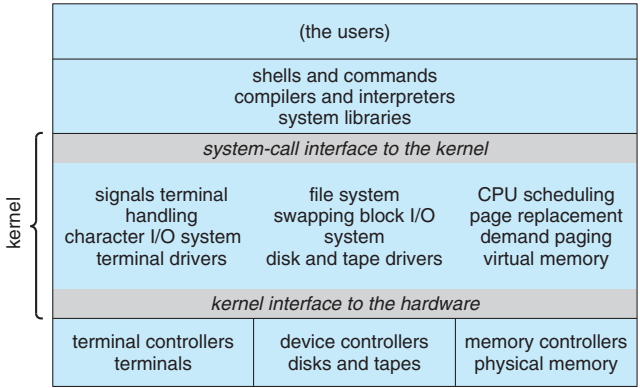


UNIX Structure

- The original version(s) was limited in terms of structure.

⊗ *UNIX OS consisted of 2 parts:*

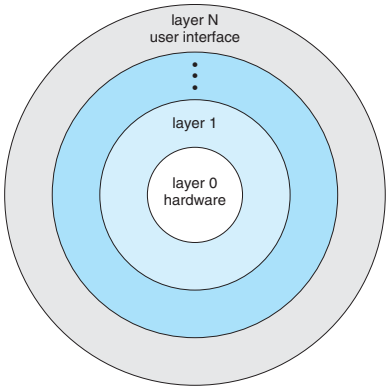
- *System Programs*
- *The UNIX kernel:*
  - *Included everything below the system call interface.*
  - *Offers file system, CPU scheduling, memory management, and other functions in one level.*



Layered Approach in Structuring OSes

■ The OS is built on a number of layers.

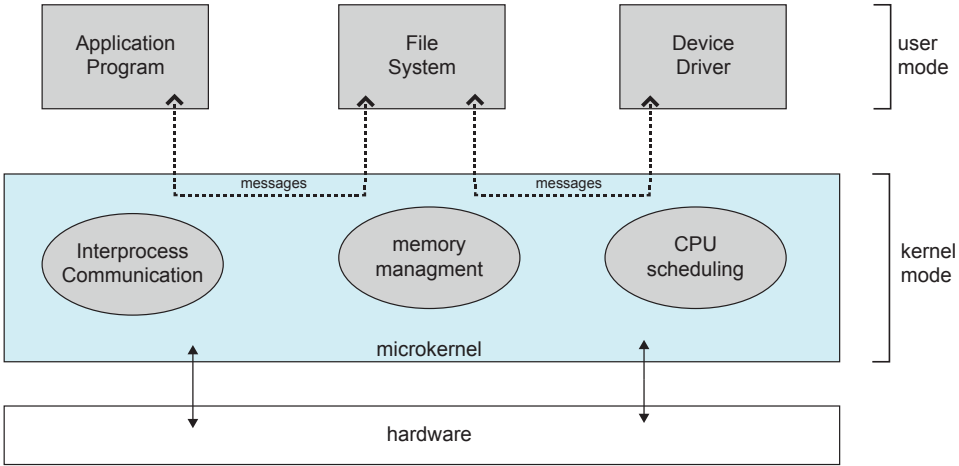
- Each layer is built of top of another in *ring* fashion.
- The bottom layer 0 is the hardware and the highest layer N is the user interface.
- The introduced modularity allows every layer to use functions/operations and services of only lower-level layers.



The Microkernel Structure

- Major objective: move as much space from kernel to user!
  - CMU's `Mach` was an implementation of a *microkernel* structure.
  - Communication takes place between user modules using message passing.
  - Pros: Easier to extend a microkernel, easier to port the OS to new architectures, more reliable (less code in kernel!) and more secure.
  - Cons: Performance overhead of user-space to kernel-space communication.

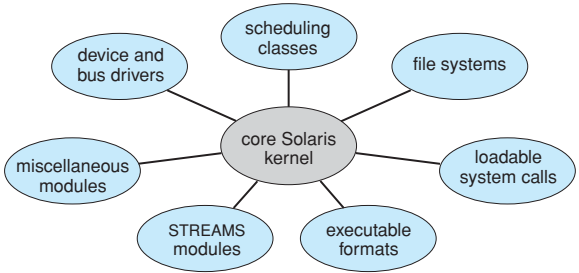
The Mach Microkernel Structure



Loadable Modules

■ Contemporary OSes realize loadable kernel modules

- Borrows for O-O approach.
- Each core component is separate.
- Each component talks to others over known interfaces.
- Each module is loadable as needed within the kernel.
- Linux and Solaris have followed this approach.



Hybrid Operating Systems

- *Hybrid OSes* combine multiple approaches to address performance, security, usability needs.
- Linux and Solaris have monolithic kernels and feature dynamic loading of functionality.
- Windows is also monolithic along with microkernel for different subsystems (i.e., workstation/server services, security, POSIX, Win32, OS2 modules).
- Mac OS X is a combination of `Mach` microkernel, `BSD Unix` parts and dynamically loadable modules (a.k.a., as kernel extensions).

Mobile Operating Systems

- iOS is based on Mac OS X and added functionalities; it does not run OS X apps natively.
  - Objective-C API (Cocoa Touch)
  - Media Services (graphics, audio, video)
  - Core Services (cloud comp, databases)

- Android is based on Linux kernel - also offers:
  - Power management
  - Runtime environment include core set of libraries and the `Davlik VM`.
  - Apps written in Java and Android API.
  - Java class files compiled to Java-bytecode then translated to executables that run on `Davlik`.
  - Libraries include frameworks for `webkit`, `SQLite`, multimedia, and a reduced `libc`.

Debugging the OS

- Debugging is about revealing and fixing errors or bugs.
  - OS create log files with errors produced.
  - Core Dumps are files that capture the state of a crashing program.
  - Crash dump files contain kernel memory once OS fails.
  - Trace listings of activities: to get a glimpse of what has happened – `dtrace`.
  - Profiling: seeking to understand patterns of use in the progression of PC.

System Boot

- Once the CS is powered on, execution commences at a *fixed memory location*.
  - Firmware ROM is used to store initial boot code.
  - Often the process is has 2-steps.
    - Boot block is at fixed location is loaded by ROM code.
    - This block loads the *bootstrap loader* from disk.
  - A widely used bootstrap loader such as `GRUB2` allows for the selection of kernel from multiple disks, versions, kernel options.
  - Kernel is loaded in main-memory and the OS is running.

Alex Delis, alex.delis -AT+ nyu.edu

NYU Abu Dhabi