Operating System Structures

Chapter #2

Course Instructor: Nausheen Shoaib

Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (UI).
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device

Operating System Services (Cont.)

- **File-system manipulation** – The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

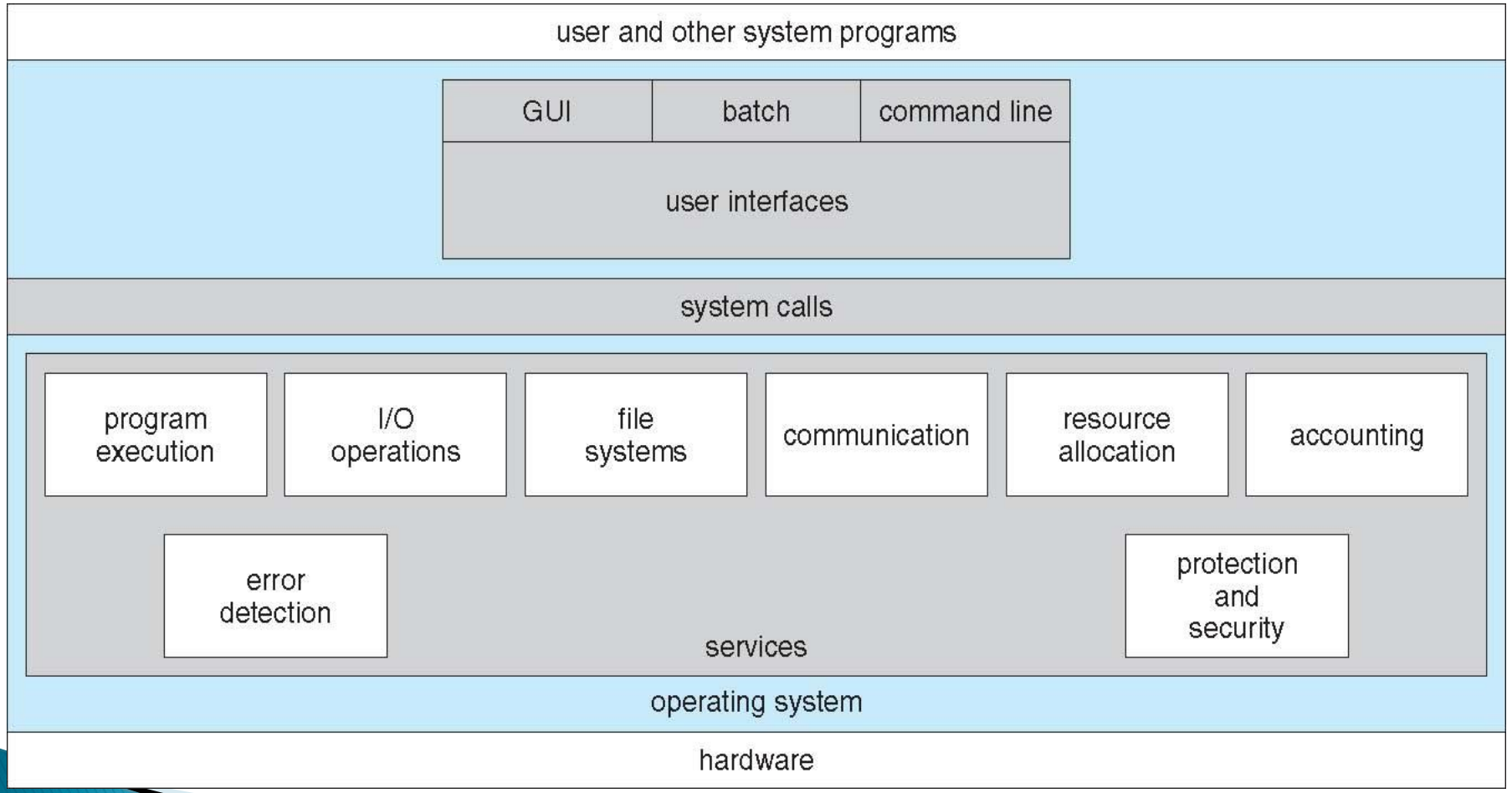
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)

- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action

Operating System Services (Cont.)

- **Resource allocation** – When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
  - Many types of resources – CPU cycles, main memory, file storage, I/O devices.
- **Accounting** – To keep track of which users use how much and what kinds of computer resources
- **Protection and security** – The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services



User Operating System Interface – CLI

CLI or **command interpreter** allows direct command entry

- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs

User Operating System Interface – GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
- Many systems now include both CLI and GUI interfaces
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

Touchscreen Interfaces

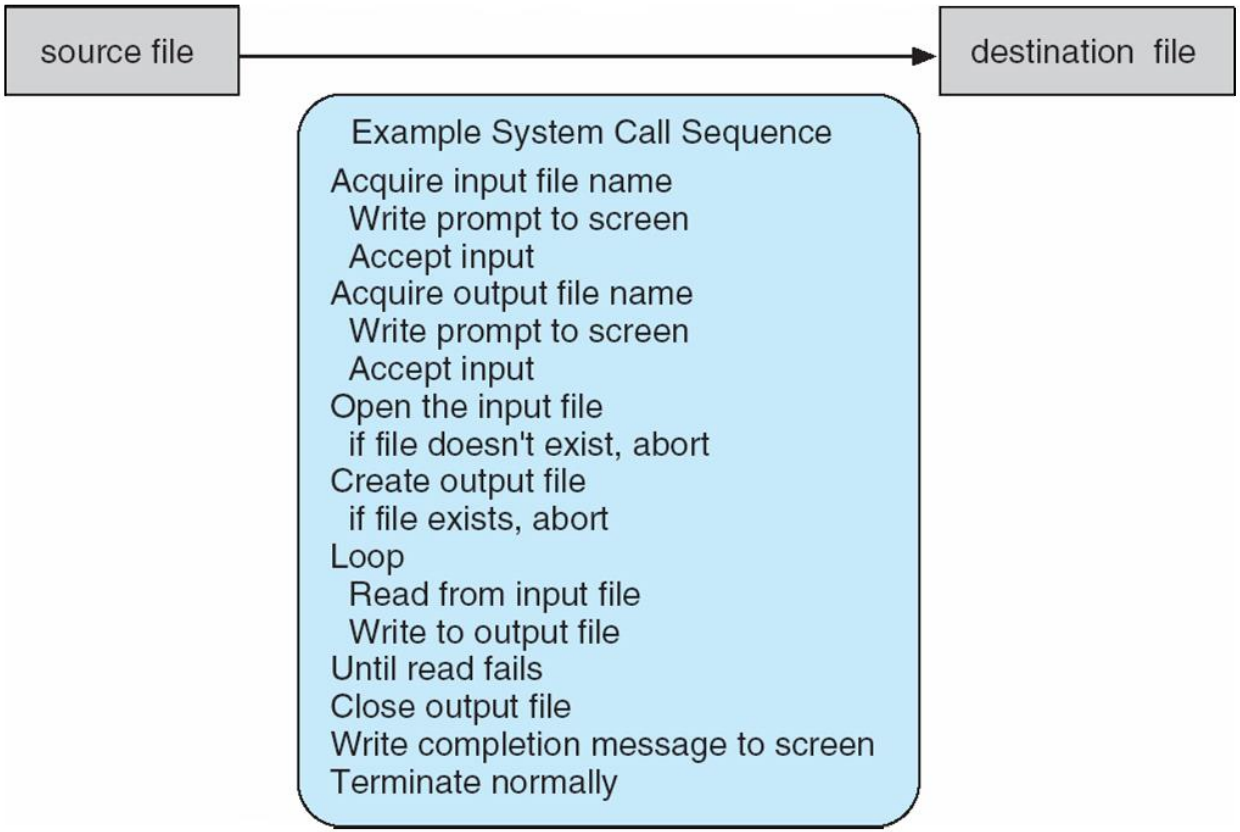
- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands.

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- 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)

Example of System Calls

- System call sequence to copy the contents of one



Example of Standard API

EXAMPLE OF STANDARD API

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)
```

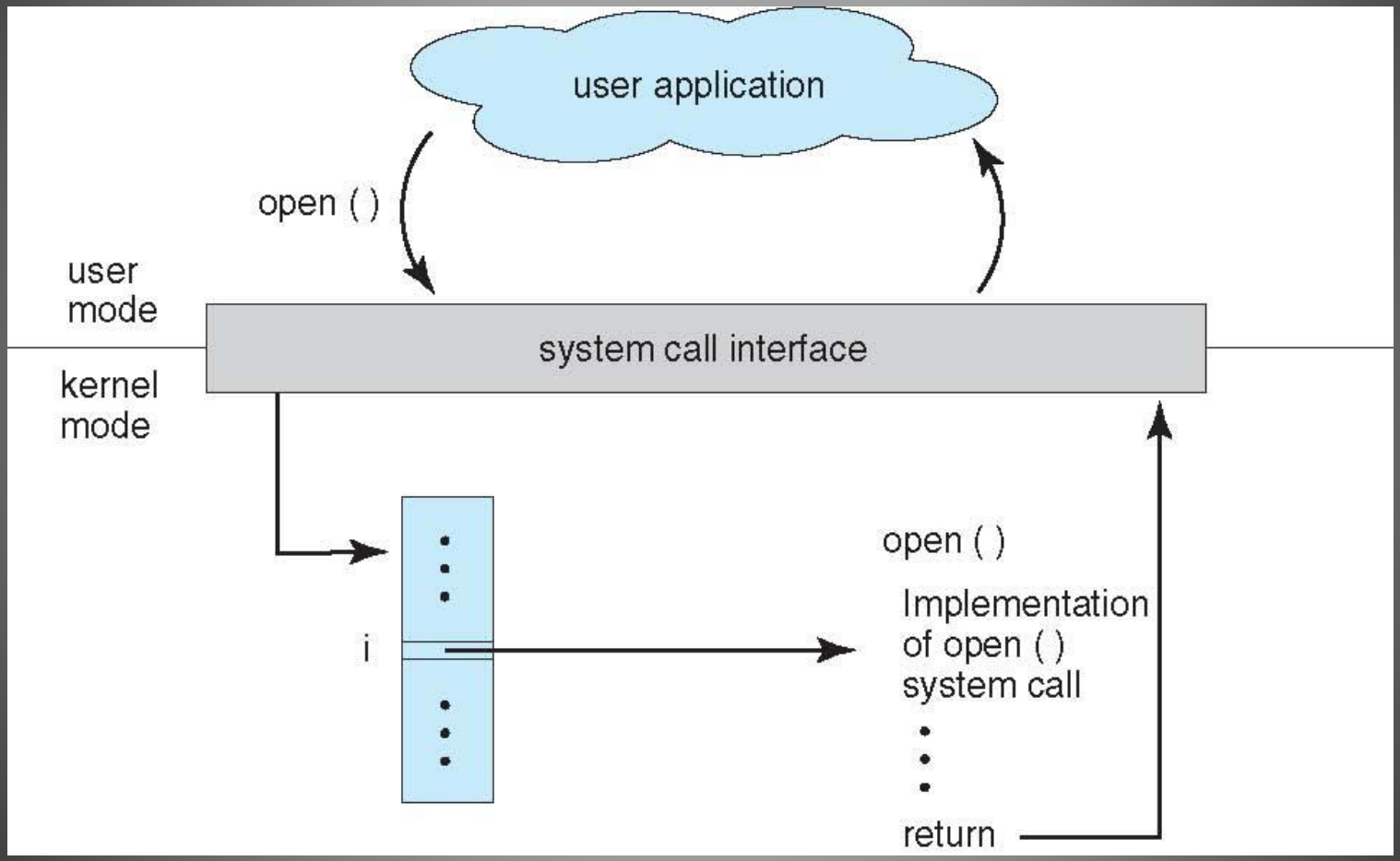
return value — function name — parameters

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.

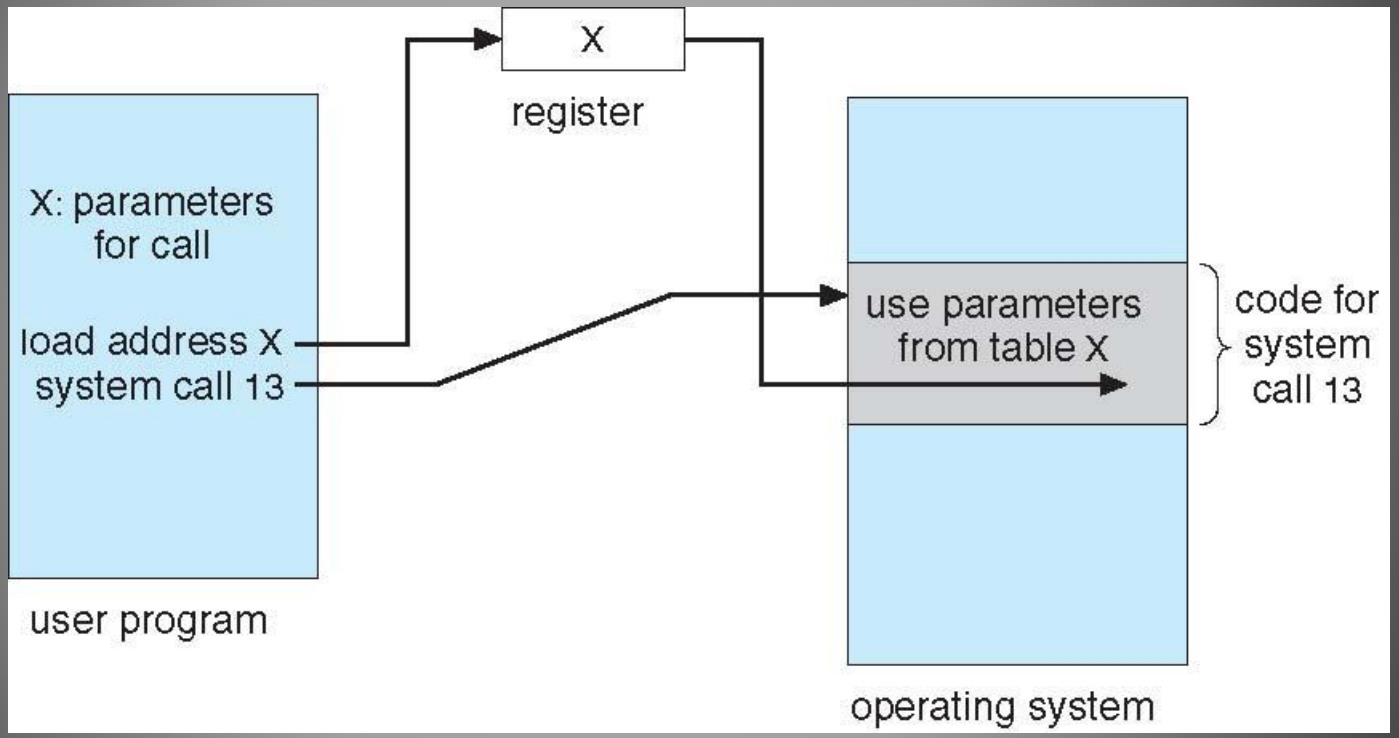
API – System Call – OS Relationship



System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system

System Call Parameter Passing via Table



Types of System Calls

System calls can be roughly grouped into five major categories:

1. Process Control
2. File management
3. Device Management
4. Information Maintenance
5. Communication

Types of System Calls

1.Process Control

- load
- execute
- create process (for example, fork on Unix-like systems or NtCreateProcess in the Windows NT Native API)
- terminate process
- get/set process attributes
- wait for time, wait event, signal event
- allocate free memory

Types of System Calls

2.File management

- create file, delete file
- open, close
- read, write, reposition
- get/set file attributes

3.Device Management

- request device, release device
- read, write, reposition
- get/set device attributes
- logically attach or detach devices

Types of System Calls

4.Information Maintenance

- get/set time or date
- get/set system data
- get/set process, file, or device attributes

5.Communication

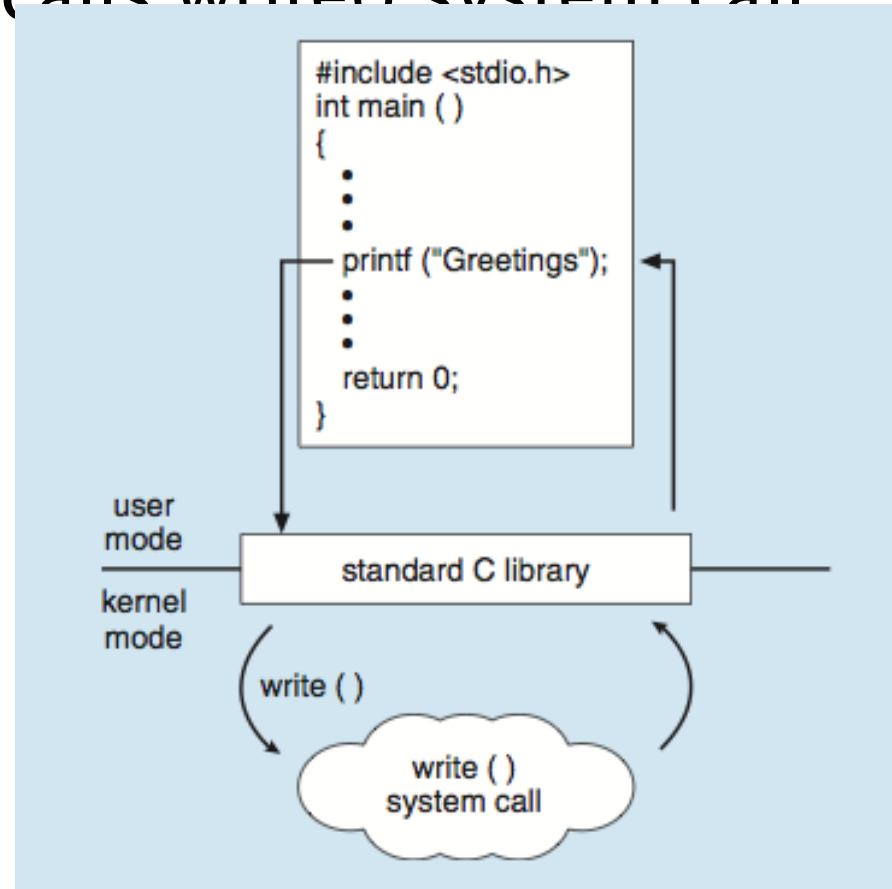
- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote device

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	WriteConsole()	write()
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shmget()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

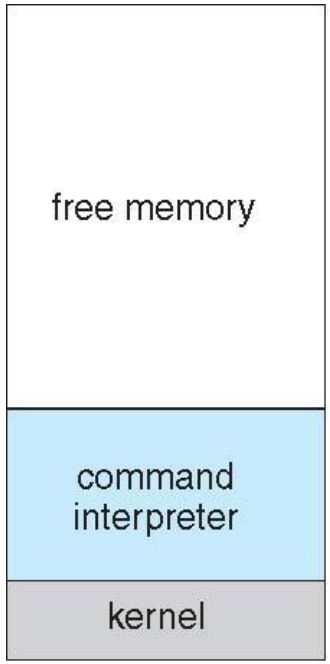
Standard C Library Example

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



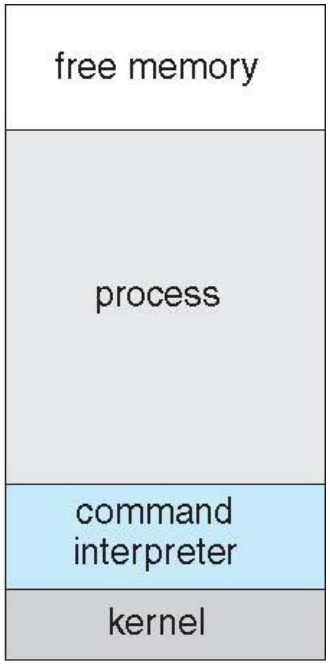
Example: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a)

At system startup

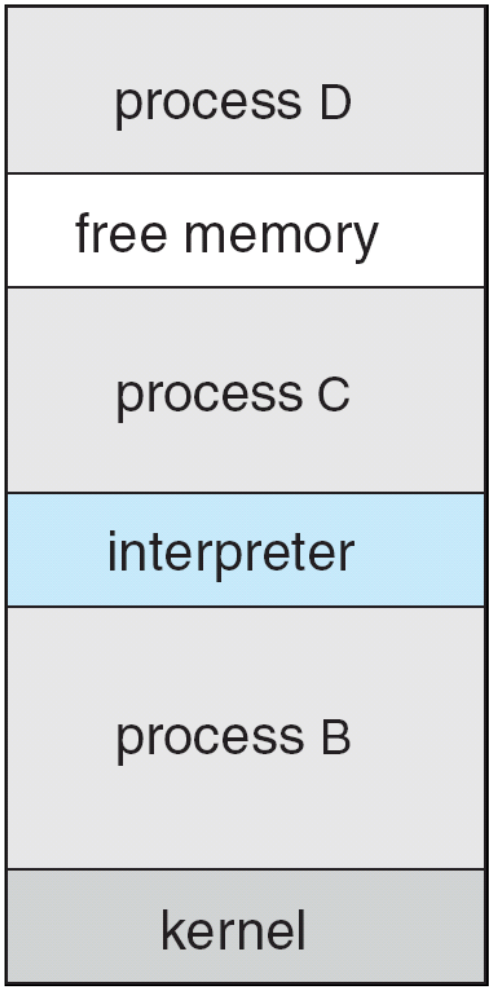


(b)

running a program

Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

System Programs

- **File management** – Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info – date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information

System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** – Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**– debugging systems for higher-level and machine language
- **Communications** – Provide the mechanism for creating virtual connections among processes, users, and computer systems

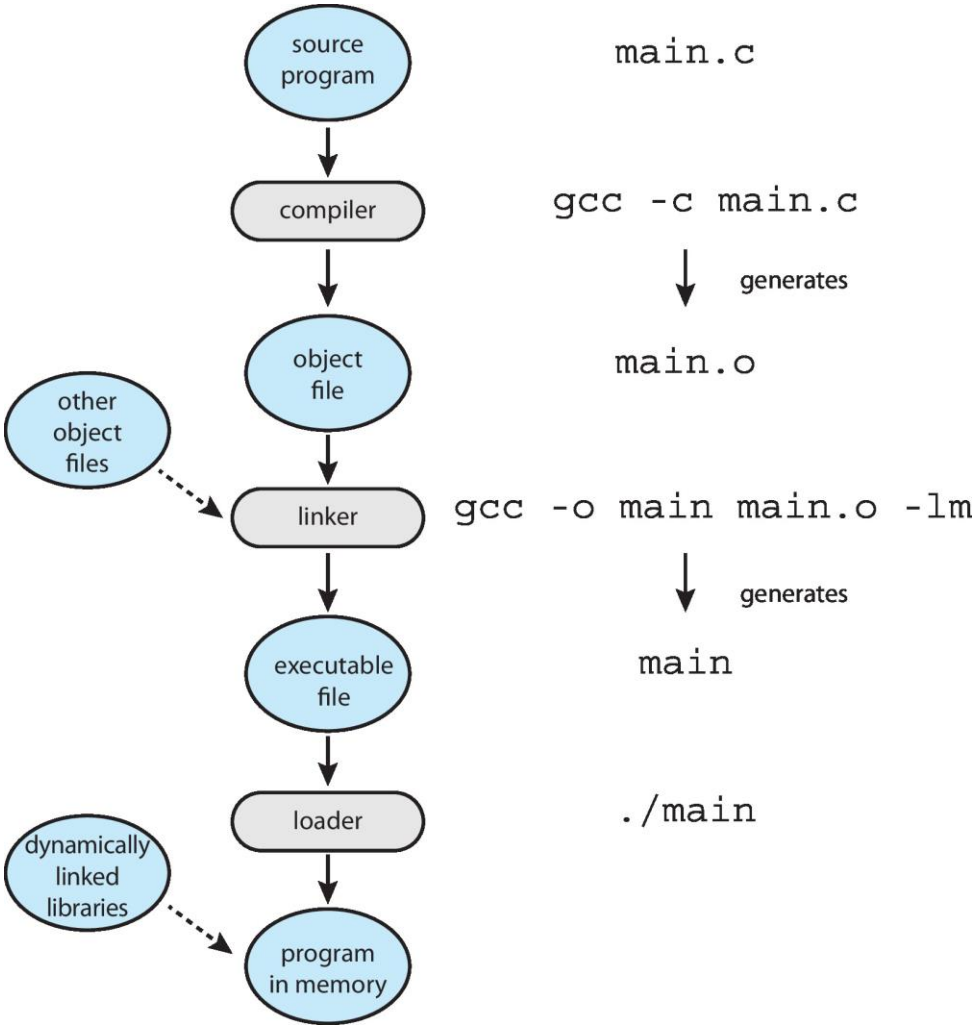
System Programs (Cont.)

- **Background Services**
  - Launch at boot time
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as services, subsystems, daemons
- **Application programs**
  - Run by users

Linkers and Loaders

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Modern general purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object, executable files have standard formats, so operating system knows how to load and start them

The Role of the Linker and Loader



Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
  - Own file formats, etc
- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- Application Binary Interface (ABI) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc

Operating System Design and Implementation

- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- User goals and System goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

Operating System Design and Implementation (Cont.)

- Important principle to separate
  Policy: *What* will be done?
  Mechanism: *How* to do it?

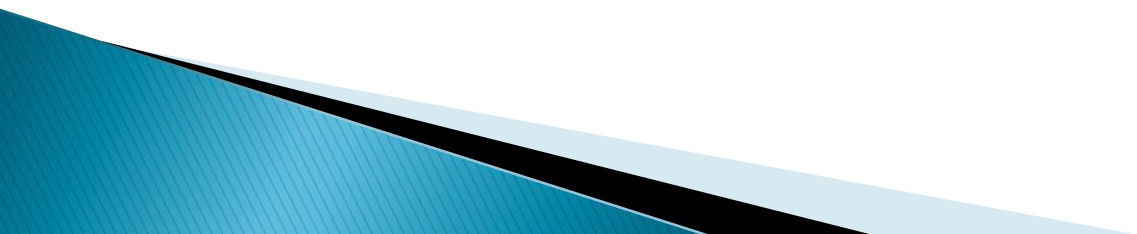
- Mechanisms determine how to do something, policies decide what will be done

- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

Operating System Structure

There are six different structures:

1. Simple Structure
2. Layered Systems
3. Microkernels
4. Modules
5. Hybrid Machines



Operating System Structure

1.Monolithic Structure:

- DOS has no modern software engineering techniques,
- Does not break the system into subsystems, and has no distinction between user and kernel modes.
- Allow all programs direct access to hardware.

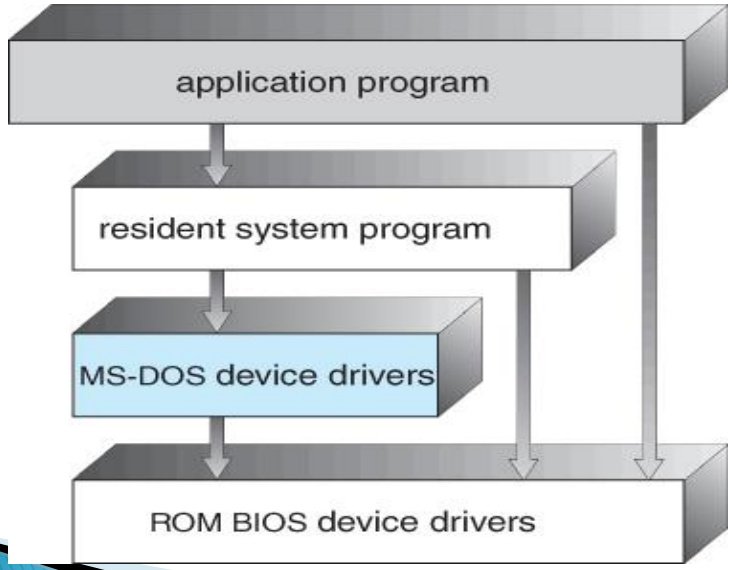


Fig: MS-DOS layer structure

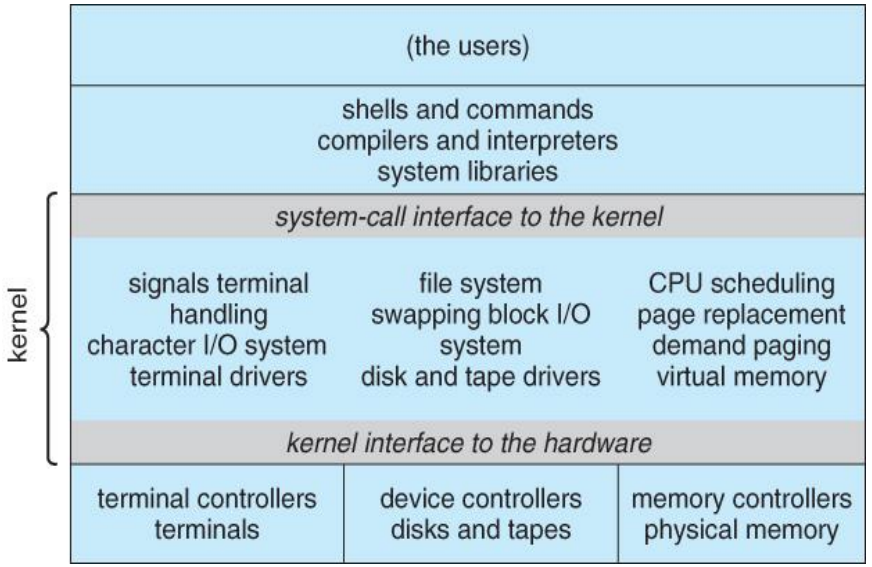
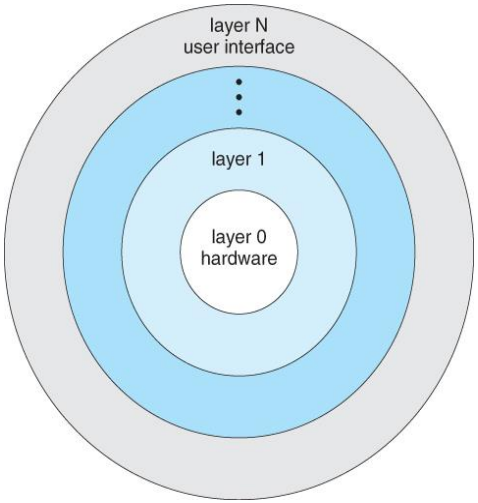


Fig: Traditional UNIX system structure

Operating System Structure

2.Layered Approach:

- Another approach is to break the OS into number of smaller layers and relies on the services provided by next lower layer.



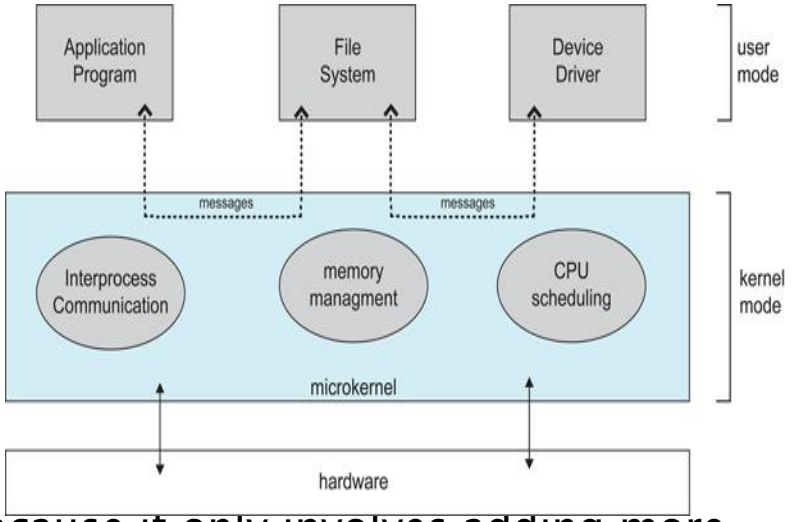
- allows each layer to be developed and debugged independently, with assumption that all lower layers is already debugged.
- The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer.
- Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.

Operating System Structure

3.Microkernels:

- micro kernels is to remove all non-essential services from the kernel, and implement them as system applications

- provide basic process and memory management, and message passing between other services



- Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.

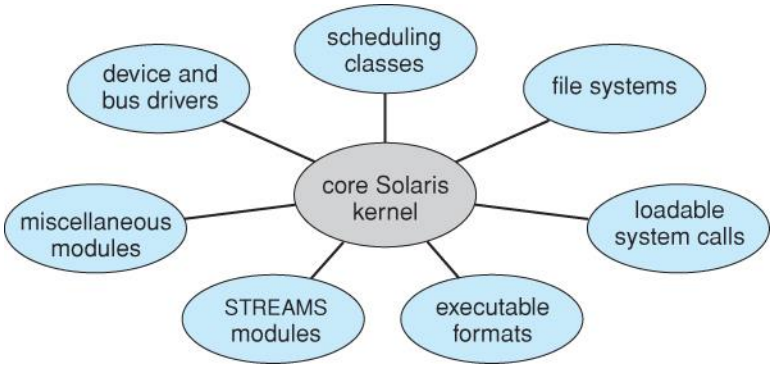
- System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.

- Mach, Windows NT are examples of kernel

Operating System Structure

4.Modules

- Modern OS development is object-oriented, with a relatively small core kernel and a set of ***modules*** which can be linked in dynamically.



- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces.

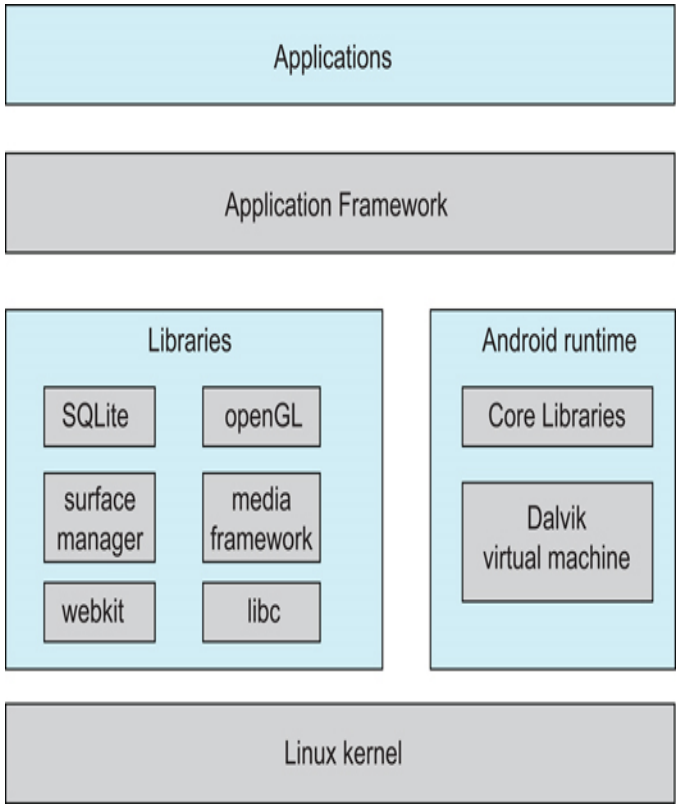
- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.

Operating System Structure

5.Hybrid Systems: Most OS today do not strictly adhere to one architecture, but are hybrids of several.

Android:

- open-source OS
- includes versions of Linux and a JVM
- apps are developed using Java



Operating-System Debugging

- Debugging is finding and fixing errors, or bugs
- OS generate log files containing error information
- Failure of an application can generate core dump file capturing memory of the process
- Operating system failure can generate crash dump file containing kernel memory

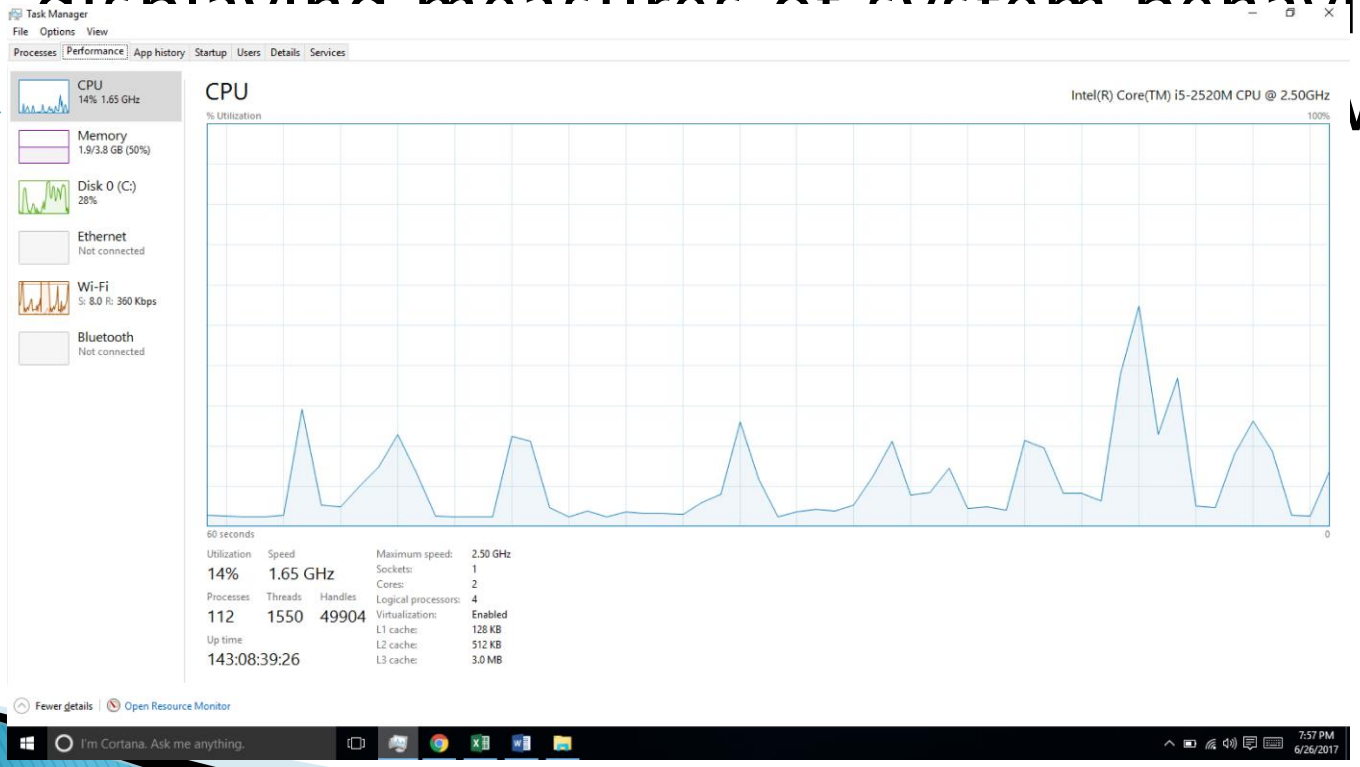
Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Operating System Generation

- SYSGEN program obtains information concerning the specific configuration of the hardware system
  - Used to build system-specific compiled kernel or system-tuned

Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- ws



Tracing

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
  - strace – trace system calls invoked by a process
  - gdb – source-level debugger
  - perf – collection of Linux performance tools
  - tcpdump – collects network packets

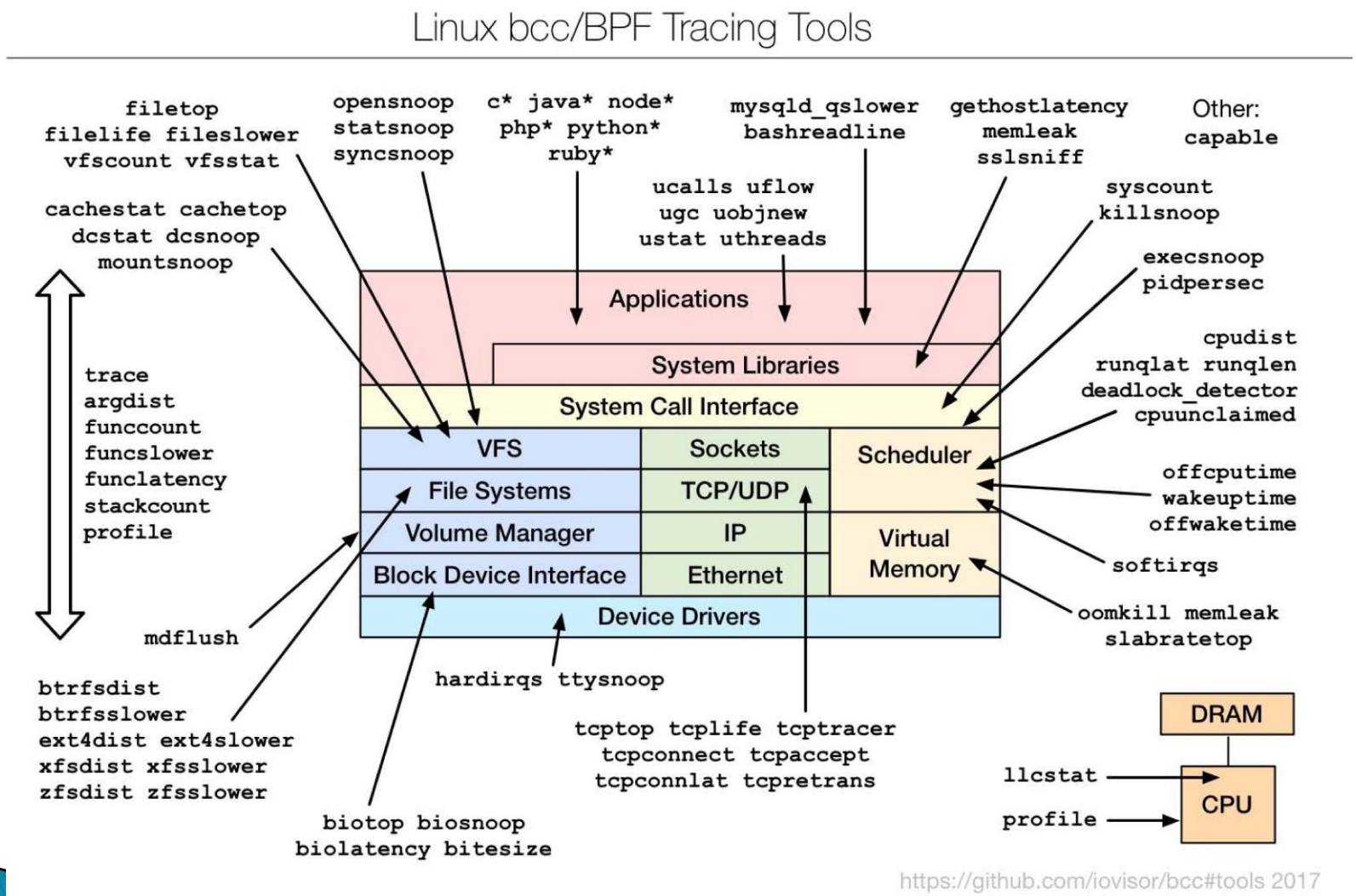
BCC

- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both and an instrument their actions
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
  - See also the original DTrace
- For example, disksnoop.py traces disk I/O activity

```
TIME(s)            T      BYTES      LAT(ms)
1946.29186700      R      8              0.27
1946.33965000      R      8              0.26
1948.34585000      W      8192           0.96
1950.43251000      R      4096           0.56
1951.74121000      R      4096           0.35
```

- Many other tools (next slide)

Linux bcc/BPF Tracing Tools



https://github.com/iovisor/bcc#tools 2017