Bilkent University
Department of Computer Engineering
CS342 Operating Systems

Introduction - 2
Operating System Structures

Last Update: Sep 20, 2022

Outline

Outline

- Operating System Services
- User Operating System Interface
- System Calls
- System Programs
- Operating System Structure

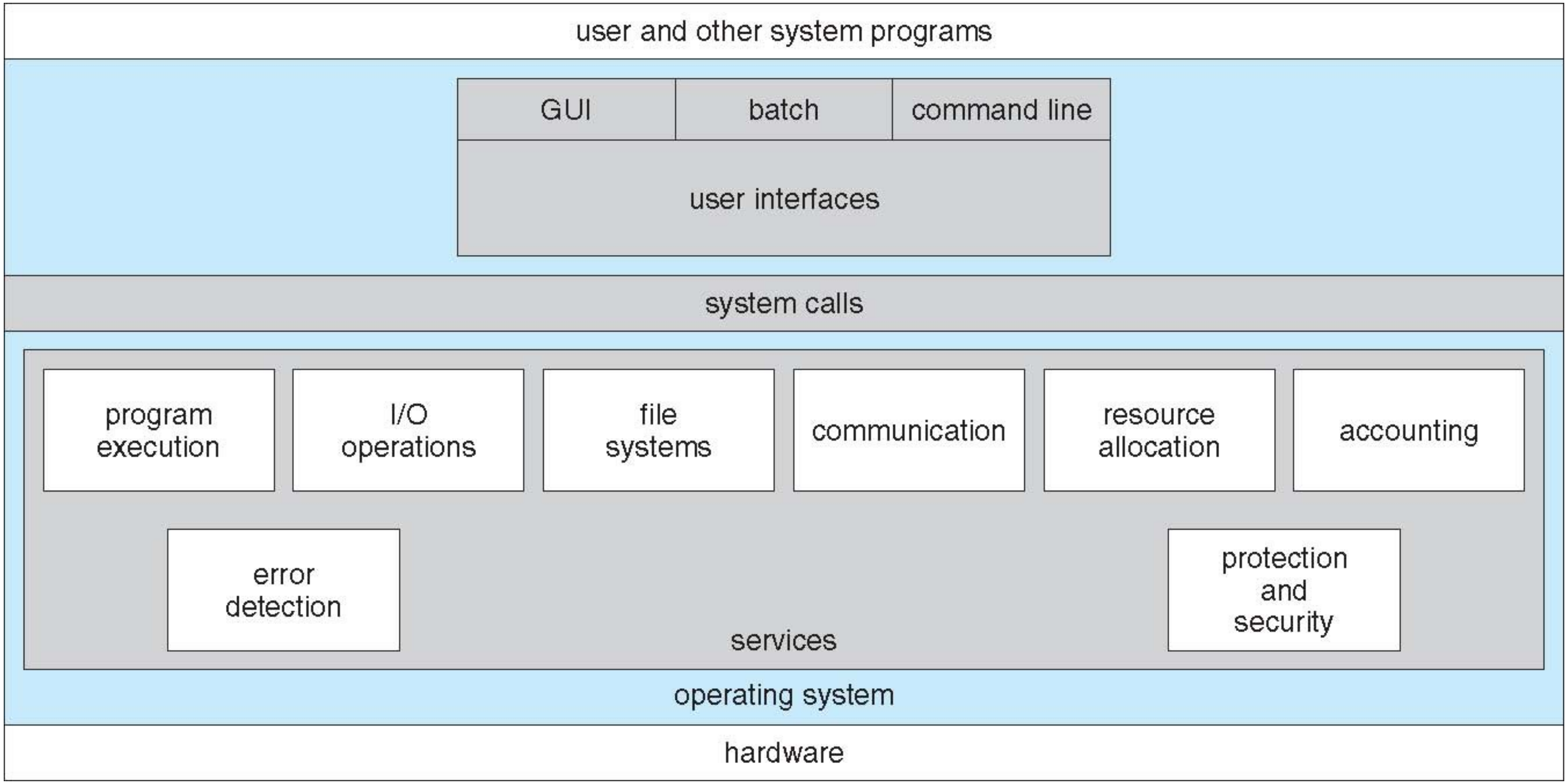
Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system

Operating System Services

- For user
  - User interface
  - Program execution
  - I/O operations
  - File-system manipulation
  - Process communication
  - Error detection and handling
- For System: efficiency and sharing
  - Resource allocation
  - Accounting
  - Protection and security

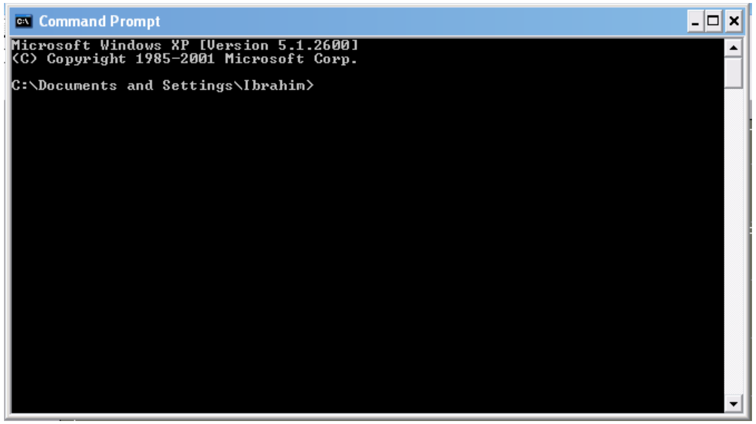
OS Services


user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

[User - Operating System] Interface - CLI

- **CLI**: Command Line Interface (CLI) or

command interpreter (shell)

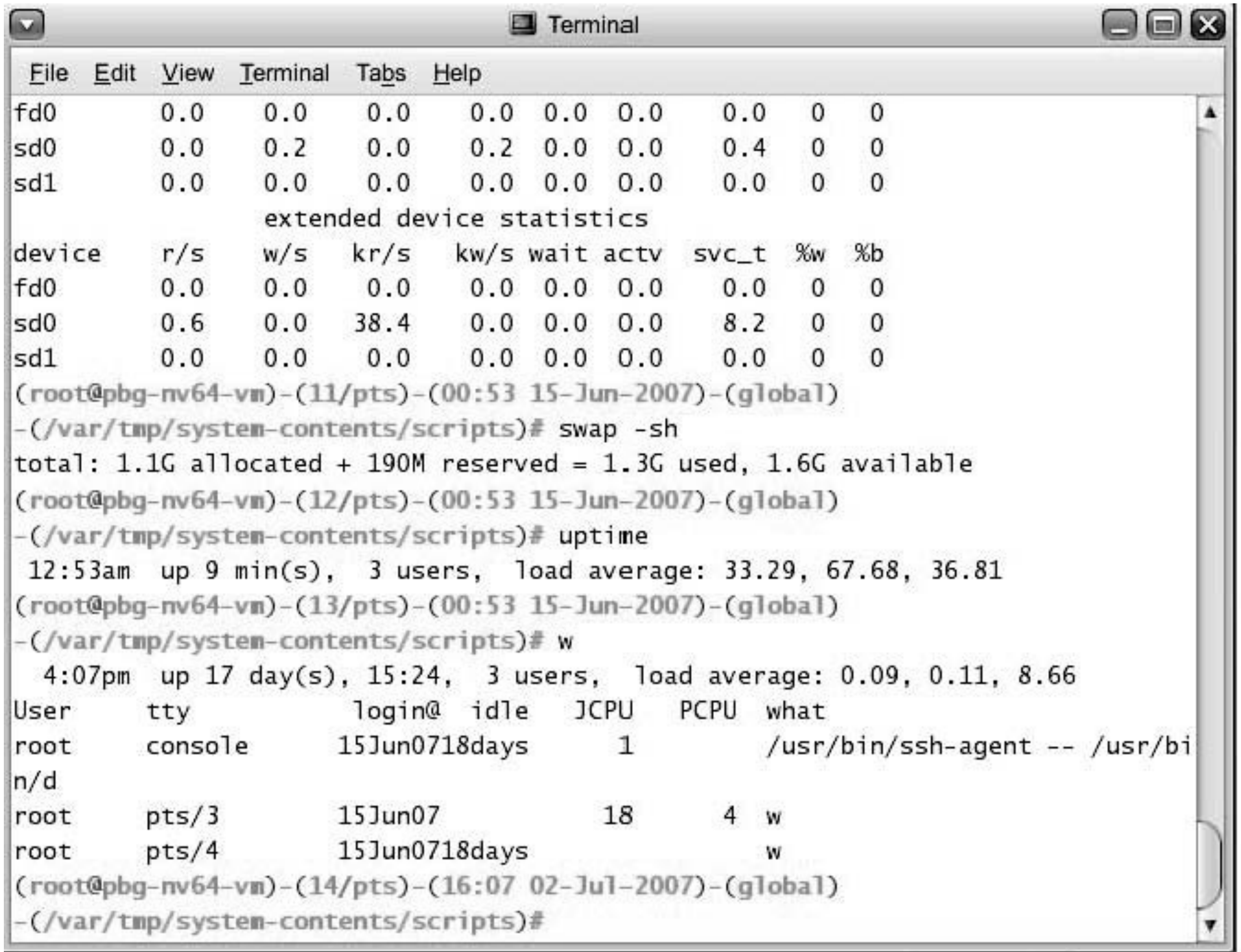


  - fetches a command from user and executes it
  - Interpreter is in kernel or is a system program,
  - many flavors
    - Some commands may be **built-in to** the shell,
    - Come commands may be **another program**
- **GUI**: User-friendly desktop interface
  - Icons represent files, programs, actions, etc.

Many operating systems now include both CLI and GUI interfaces

Linux: command shells available (CLI); KDE as GUI

Bourne Shell Command Interpreter



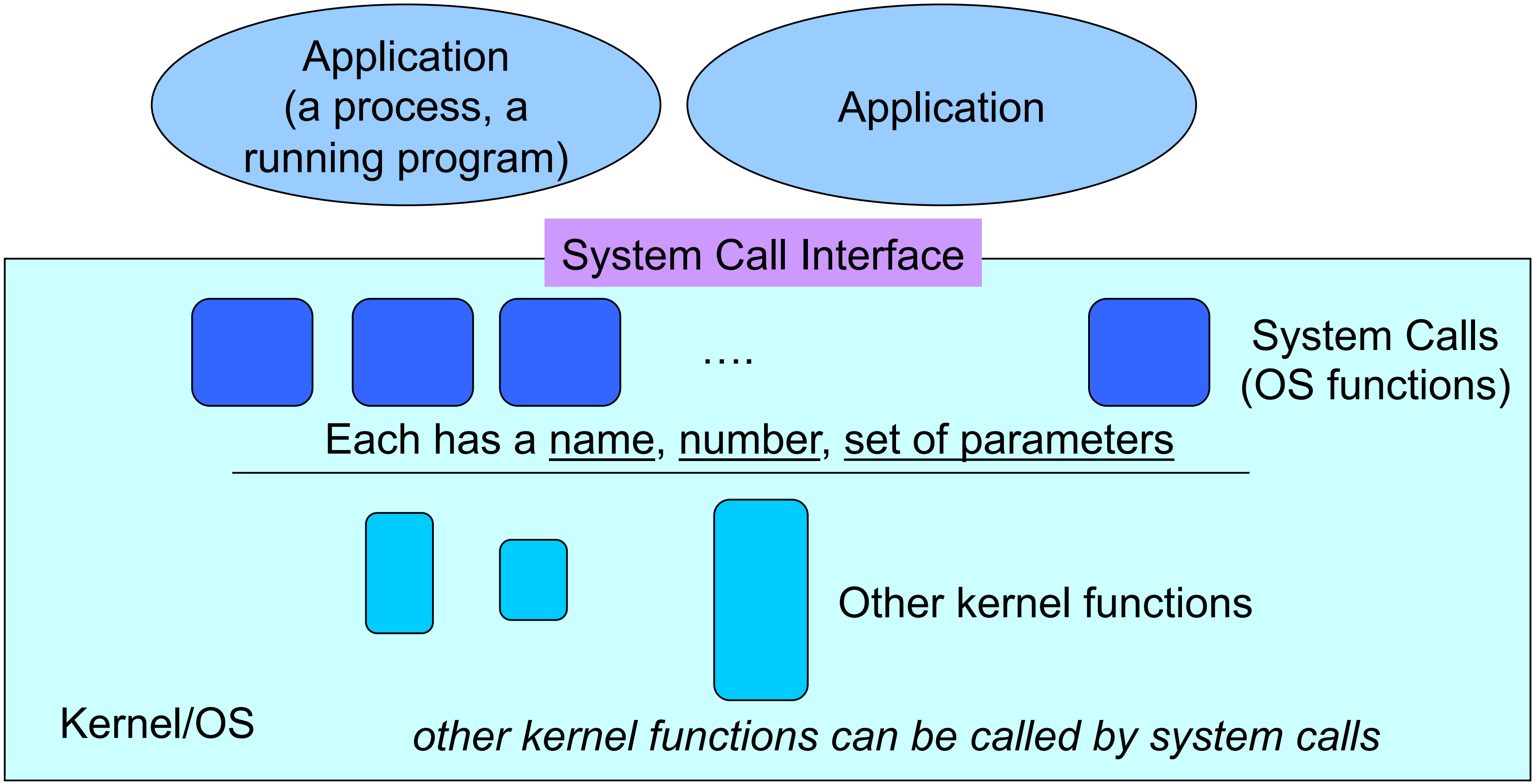

```
fd0          0.0    0.0    0.0    0.0  0.0  0.0    0.0   0    0
sd0          0.0    0.2    0.0    0.2  0.0  0.0    0.4   0    0
sd1          0.0    0.0    0.0    0.0  0.0  0.0    0.0   0    0
                 extended device statistics
device       r/s    w/s   kr/s   kw/s wait actv  svc_t  %w   %b
fd0          0.0    0.0    0.0    0.0  0.0  0.0    0.0   0    0
sd0          0.6    0.0   38.4    0.0  0.0  0.0    8.2   0    0
sd1          0.0    0.0    0.0    0.0  0.0  0.0    0.0   0    0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# uptime
 12:53am  up 9 min(s),  3 users,  load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# w
  4:07pm  up 17 day(s), 15:24,  3 users,  load average: 0.09, 0.11, 8.66
User     tty           login@  idle   JCPU   PCPU  what
root     console      15Jun0718days      1         /usr/bin/ssh-agent -- /usr/bi
n/d
root     pts/3        15Jun07           18      4  w
root     pts/4        15Jun0718days                w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-(/var/tmp/system-contents/scripts)#
```

The MacOS X GUI

System Calls

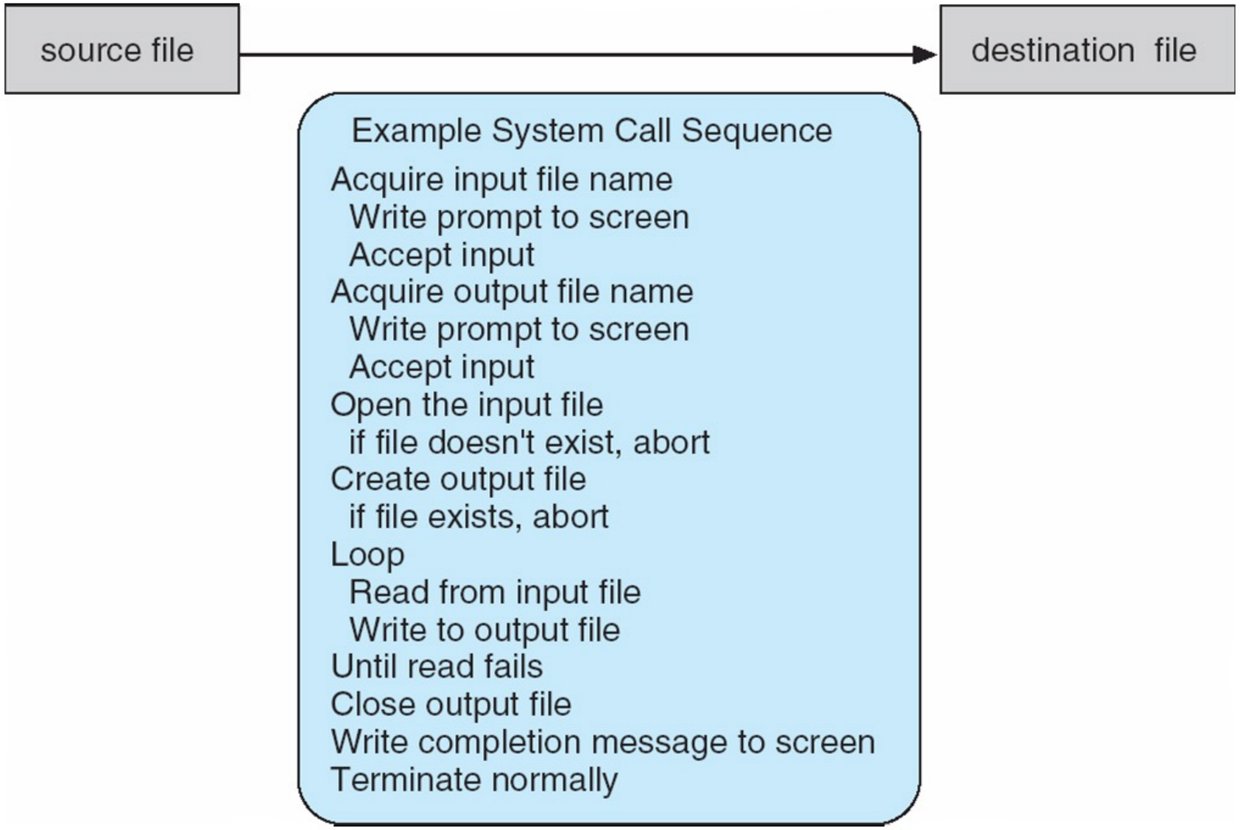
- Programming interface to the services provided by the OS
  - i.e., interface provided to applications (to commands and programs
- Typically written in a high-level language (C or C++)
- Are called by a running program to get service from kernel
  - We say "making a system call' or "calling a system call"
- Even a simple program may make a lot of system calls per second.

System Calls



Example of System Calls

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



Example copy program

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include <fcntl.h>


#define INSIZE 10


char infilename[128];
char outfilename[128];


char buf[1024];


int fd1;
int fd2;
```

```c
int main()
{
    printf ("enter input filename\n");
    scanf ("%s", infilename);
    printf ("enter output filename\n");
    scanf ("%s", outfilename);

    printf ("%s %s\n", infilename, outfilename);

    fd1 = open (infilename, O_RDONLY);
    if (fd1 < 0) {
        printf ("error in opening");
        exit (1);
    }
    fd2 = open (outfilename, O_WRONLY | O_CREAT, 0600);

    int n, m;
    while (1) {
        n  = read (fd1, buf, INSIZE);
        if (n > 1) {
            m  = write (fd2, buf, n);
            if (m < 0)
                printf ("write failed\n");

        }  else
            break;
    }

    close (fd1);
    close (fd2);
    printf ("finished copying\n");
}
```

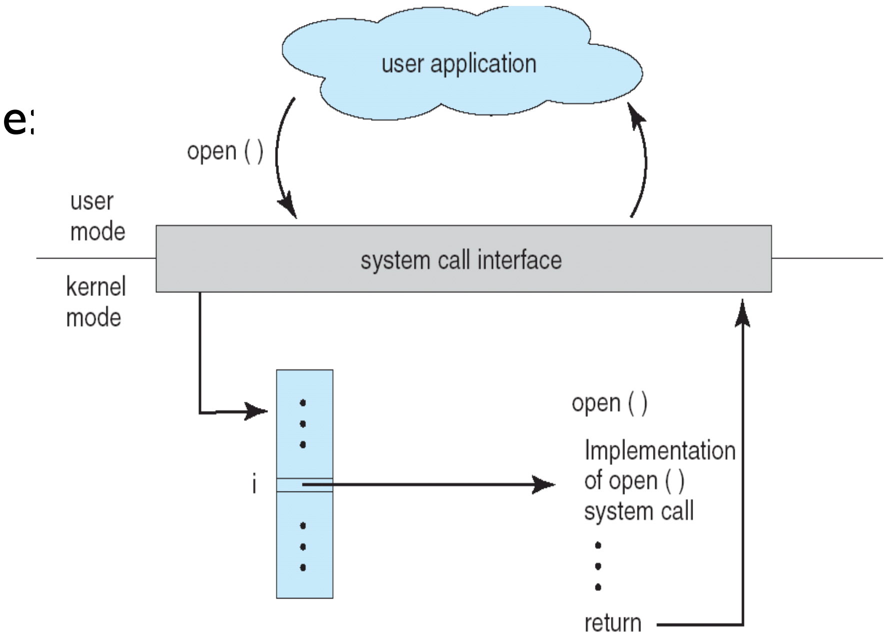
strace output

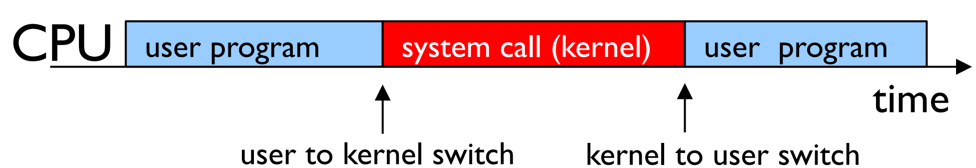
```
….
read(0, Makefile
"Makefile\n", 1024)               = 9
write(1, "enter output file\n", 18enter output file
)      = 18
read(0, out
"out\n", 1024)                    = 4
write(1, "Makefile out\n", 13Makefile out
)           = 13
openat(AT_FDCWD, "Makefile", O_RDONLY)  = 3
openat(AT_FDCWD, "out", O_WRONLY|O_CREAT, 0600) = 4
read(3, "\n\nall: myc", 10)                = 10
write(4, "\n\nall: myc", 10)               = 10
read(3, "opy\n\nmycop", 10)                = 10
write(4, "opy\n\nmycop", 10)               = 10
read(3, "y:\tmycopy.", 10)                 = 10
write(4, "y:\tmycopy.", 10)                = 10
read(3, "c\n\tgcc -Wa", 10)                = 10
write(4, "c\n\tgcc -Wa", 10)               = 10
read(3, "ll -o myco", 10)                  = 10
write(4, "ll -o myco", 10)                 = 10
read(3, "py mycopy.", 10)                  = 10
write(4, "py mycopy.", 10)                 = 10
read(3, "c\n\n\nclean:", 10)               = 10
write(4, "c\n\n\nclean:", 10)              = 10
….
```

System Call Implementation and Calling

- Typically,
  - a number associated with each system call
  - Number is used as index to a table: System Call Table
  - Table keeps addresses of system calls (system routines)
- When called, system call:
  - runs
  - may block (e.g., read system call)
  - returns
- Caller does not know system call implementation
  - Just knows the interface

user program calling open() system call





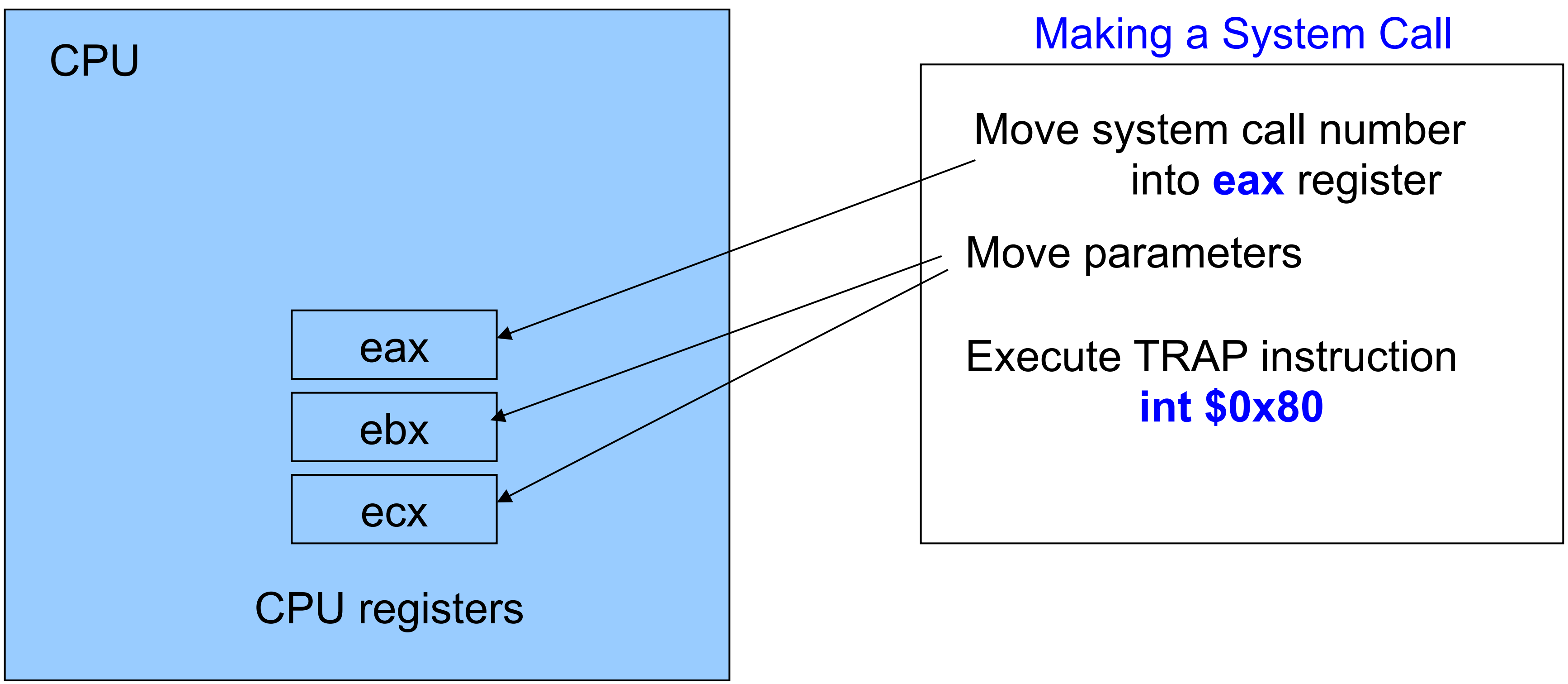
Some Linux x86-64 system calls (64 bit)

0 read // read from a file
1 write // write into a file
2 open // open a file
3 close // close a file
4 stat // get file attributes
9 mmap // map a file
12 brk // extend heap segment
32 dup2 // dup file desc

33 pause
37 alarm // set an alarm
39 getpid // get process id
57 fork // create a new process
59 execve // run a new program
60 _exit // terminate the program
61 wait4 // wait for child
62 kill // send signal

Around 400 system calls in total in Linux Kernel.

Invoking a system:
Linux and x86 32 bit architecture



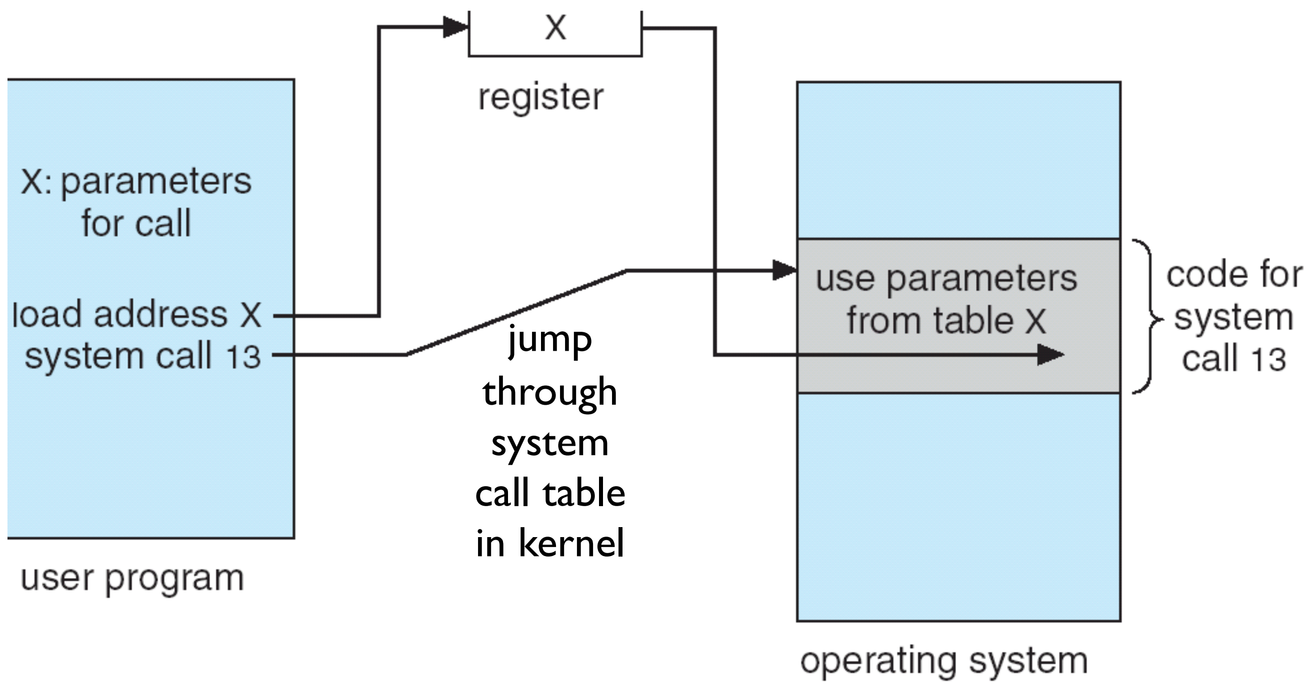
syscall or int instruction is used to make system call

System Call Parameter Passing

- Often, more information is required than the identity (number) of the desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the system call
  - 1) Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - 2) Parameters stored in a *block*, or table, in memory, and address of the block passed as a parameter in a register
  - 3) Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system

Last two methods do not limit the number or length of parameters being passed

Parameter Passing via Table

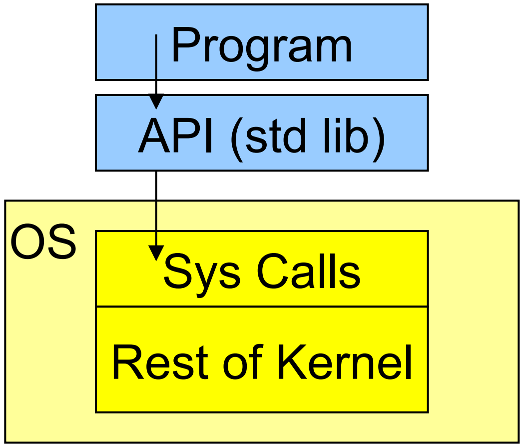


kernel can access user memory

Accessing and executing System Calls

- System calls typically not accessed directly by programs
- Mostly accessed by programs via a high-level Application Program Interface (API) (i.e., a library) rather than direct system call use

- Three most common APIs are :
  - Win32 API for Windows 32 bit system,
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
  - Java API for the Java virtual machine (JVM)



Example of Standard API - Windows

- Consider the ReadFile() function in the Win32 API — a function for reading from a file

return value

```
BOOL   ReadFile c  (HANDLE         file,
                    LPVOID         buffer,
                    DWORD          bytes To Read,
                    LPDWORD        bytes Read,
                    LPOVERLAPPED   ovl);
```

function name

parameters

- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

Example of Standard API - Linux

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)
```

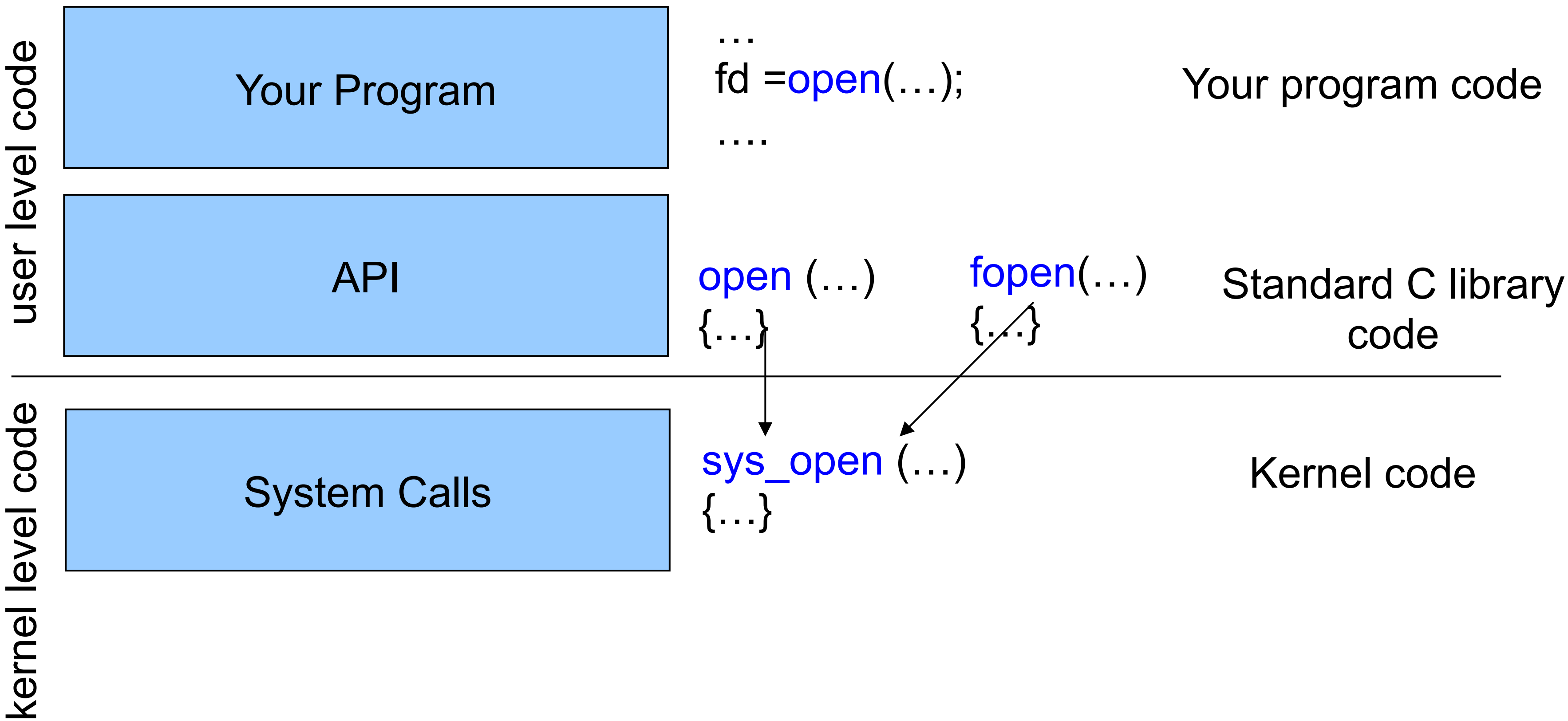
return value | function name | parameters

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

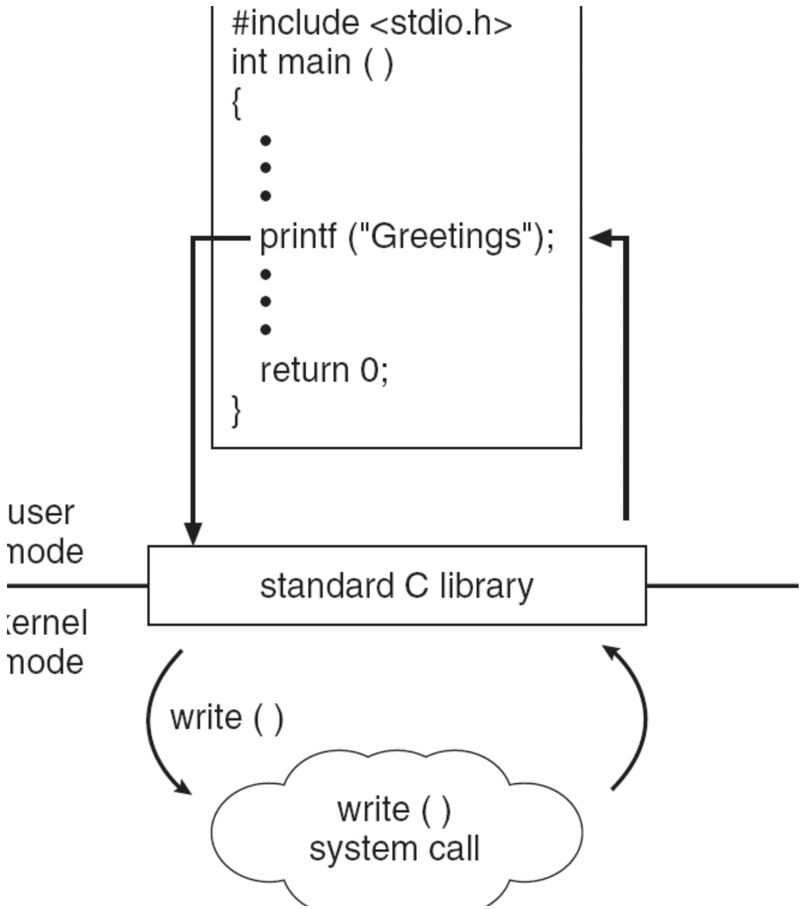
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Why use APIs rather than system calls directly?



Standard C Library Example

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



Types of System Calls

- We can categorize system calls:
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications
  - Protection

Examples of Windows and Unix 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()

Hello World with Linux System Call

hello.c

```
int main()
{
    write(1,
          "hello, world\n",
          13);
    _exit(0)
}
```

In 64 bit Linux
for x86_64 arch

hello.S

```
.section .data
string:
        .ascii "hello, world\n"
string_end:
        .equ len, string_end - string.section
.text.globl
main:
        movq $1, %rax    # write system call number
        movq $1, %rdi     # put file handle for output
        movq $string, %rsi  #put the string
        movq $len, %rdx     #put length
        syscall    #make the call
        movq $60, %rax    #exit system call number
        movq $0, %rdi
        syscall.    #make the call
```

Hello World with direct Linux System Call

Compile and run the assembly version as below:

```
$ gcc -c  hello.S  -o hello.o
$ gcc -no-pie hello.o -o hello
$ ./hello
```

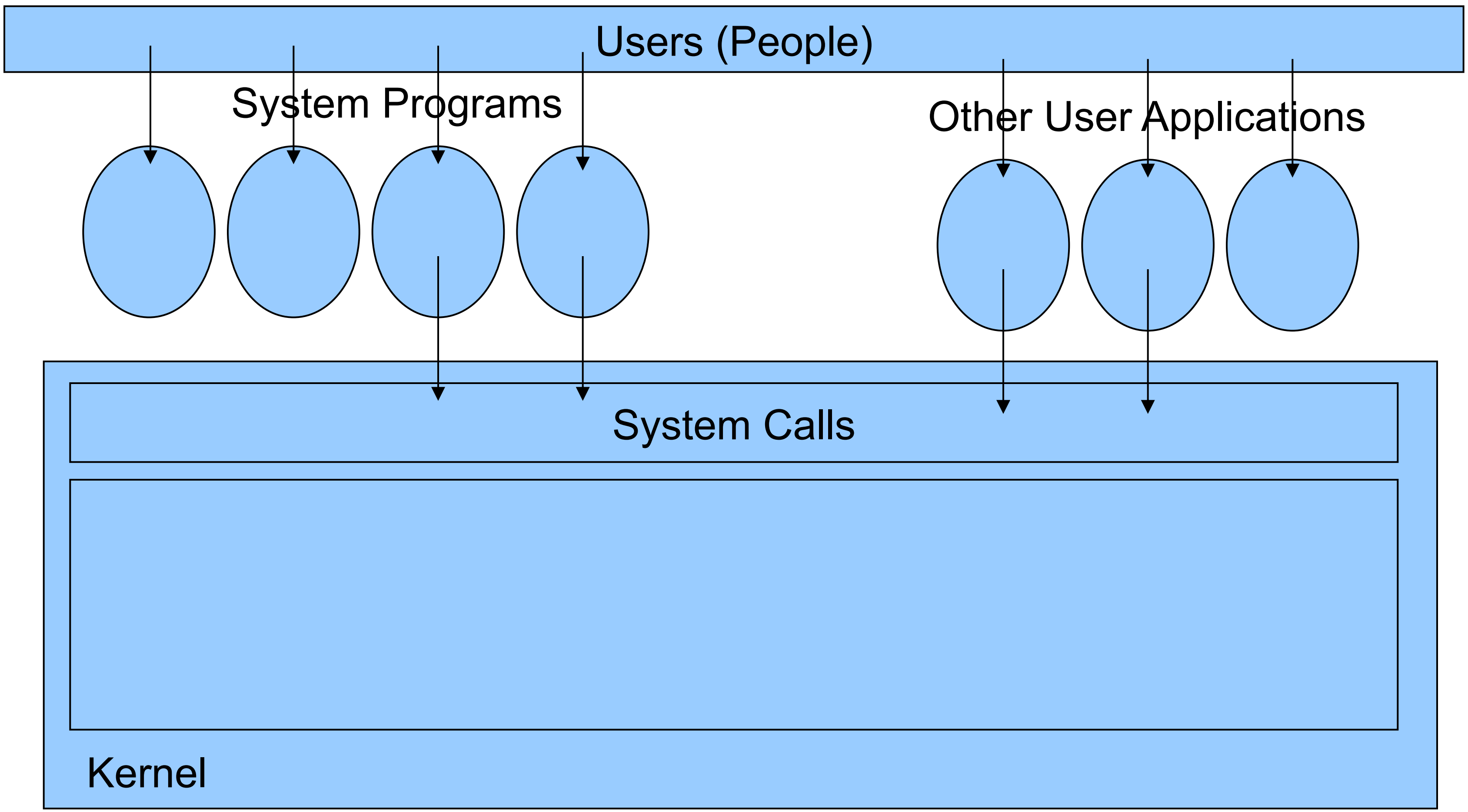
System Services/Programs

- System programs provide a convenient environment for program development and execution. They help you to manage the system. They can be divided into categories:
  - File manipulation (create, delete, copy, rename, print, list, …)
  - Status information (date, time, amount of available memory, disk space, who is logged on, …)
  - File modification (text editors, grep, …)
  - Programming language support (compiler, debuggers, …)
  - Program loading and execution (loaders, linkers)
  - Communications (ftp, browsers, ssh, …)
- Other System Utilities/Applications may come with OS CD (games, math solvers, plotting tools, database systems, spreadsheets, word processors, …).

System Services/Programs

- Most users' view of the operation system is defined by system programs, not the actual system calls.
- Some of the system programs are simply user interfaces to system calls; others are considerably more complex.
  - **create file**: simple system program that can just call "create" system call or something similar (open system call with O_CREAT flag).
  - compiler: complex system program.

System Programs



From OS's view: system+user programs are all applications

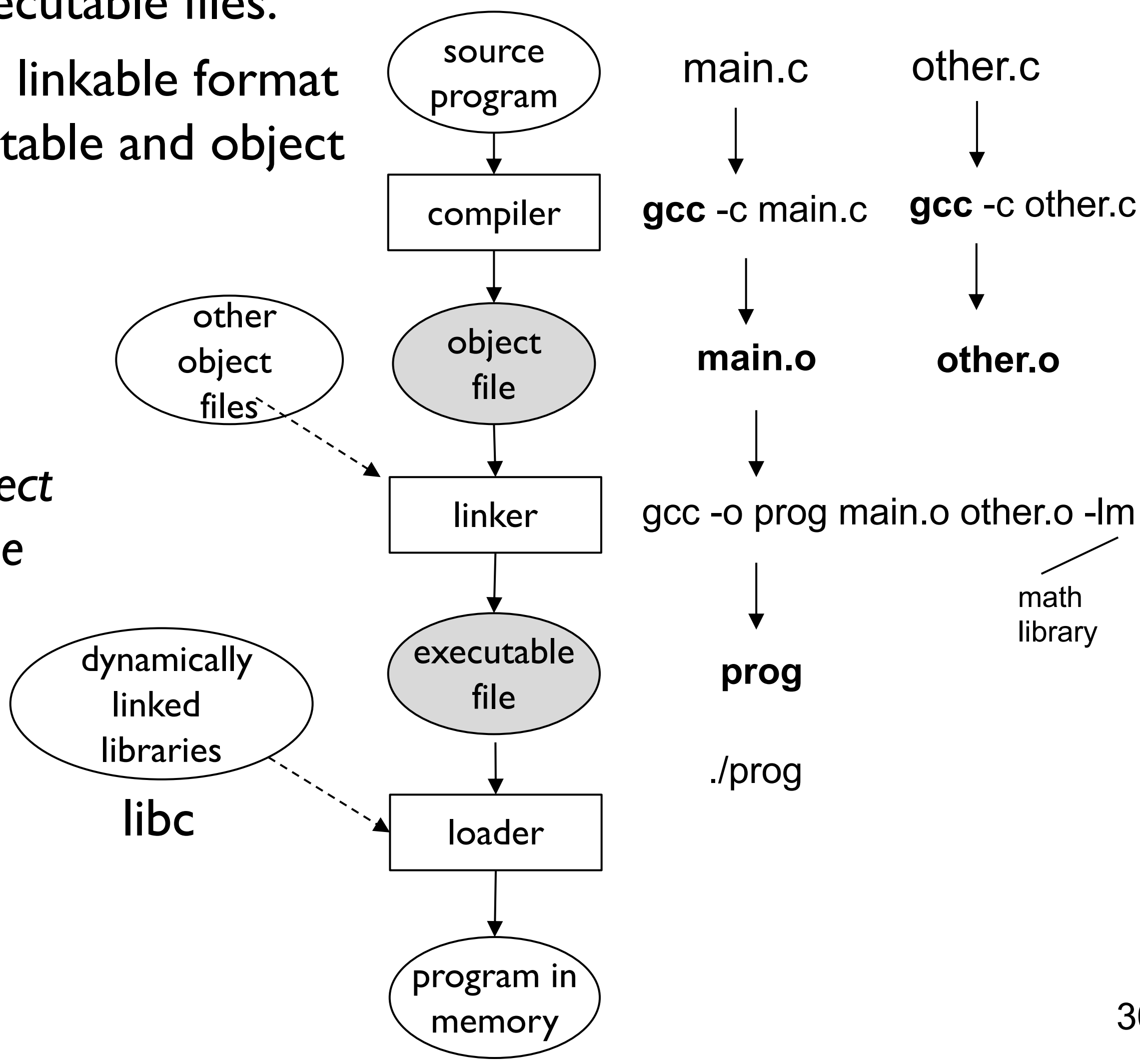
Linkers and loaders

- They generate and load executable files.
- ELF format: executable and linkable format (this is the format of executable and object files in Linux)

To get information about an object or executable file, we can use the following tools:

readelf

objdump (more general)



OS Design and Implementation Issues

- We need to set design goals
  - User goals: some goals are user oriented.
  - System goals: some goals are system related, not directly visible to users.
- Mechanisms and policies
  - Principle: Separate mechanisms from policies
  - Mechanism: how to do
  - Policy: what to do
- Implementation
  - C, C++ (high level language)
  - Porting to different architectures

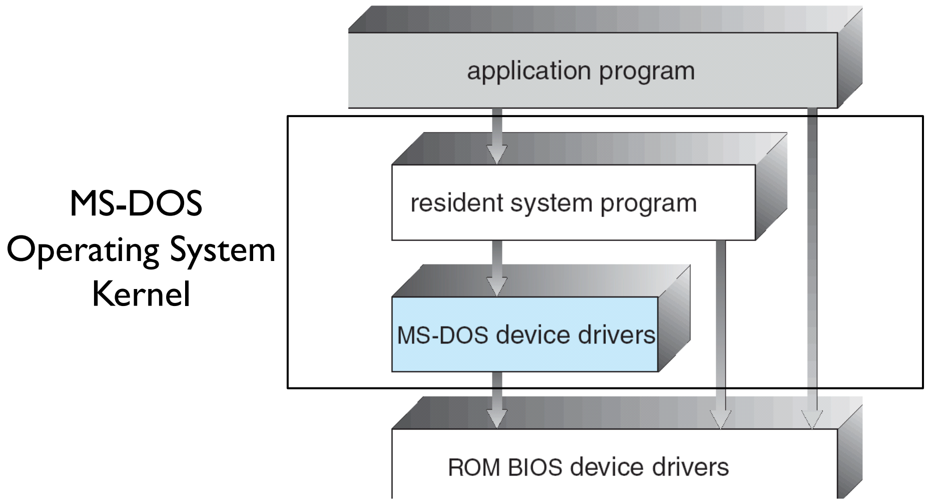
Structuring Operating System

OS Structure

- Simple Structure (MSDOS)
- Layered Approach
- Microkernel Approach
- Modules Approach

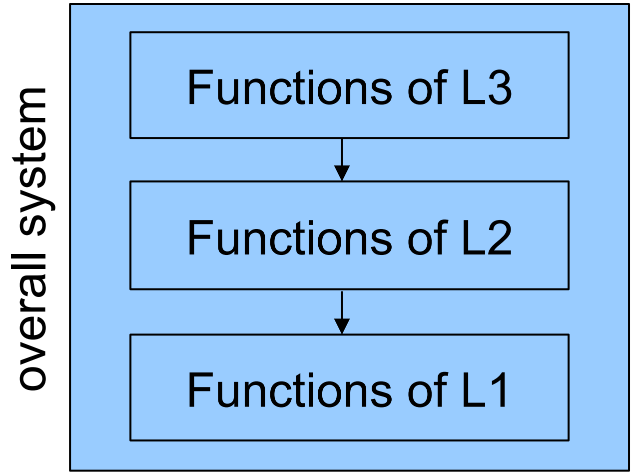
Simple Structure

- MS-DOS: written to provide the most functionality in the least space
  - Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.

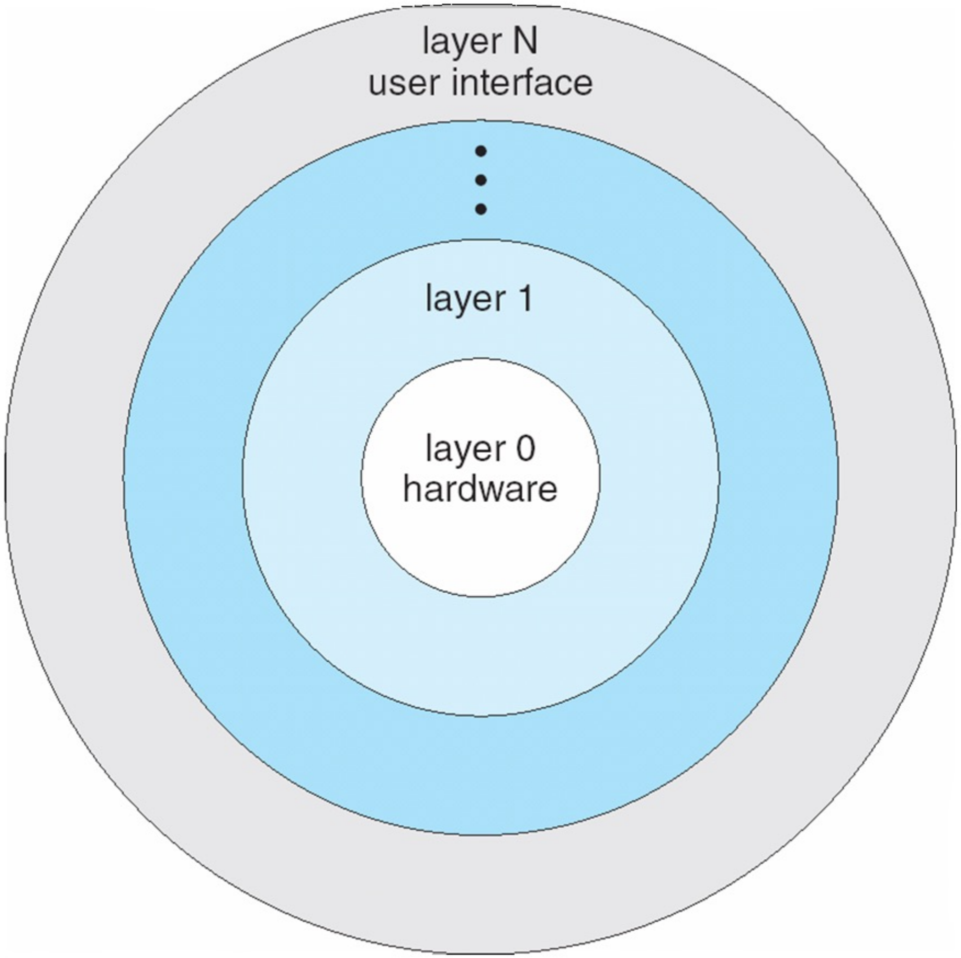


Layered Approach

- The operating system is divided into a number of layers (levels), each built on each other. The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- Layers are selected such that each layer uses functions (operations) and services of only the layer below it.



Layered Operating System

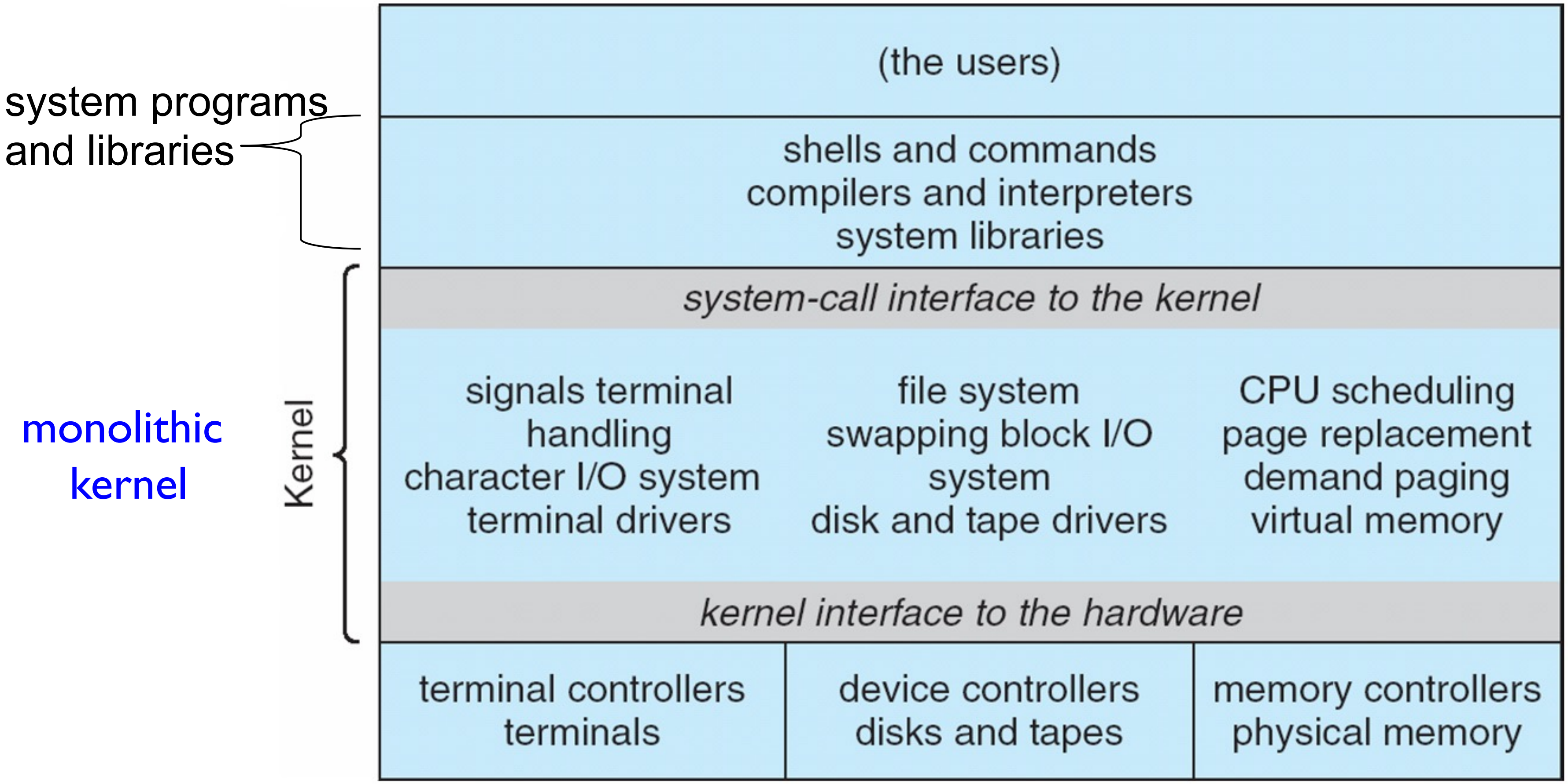


Unix

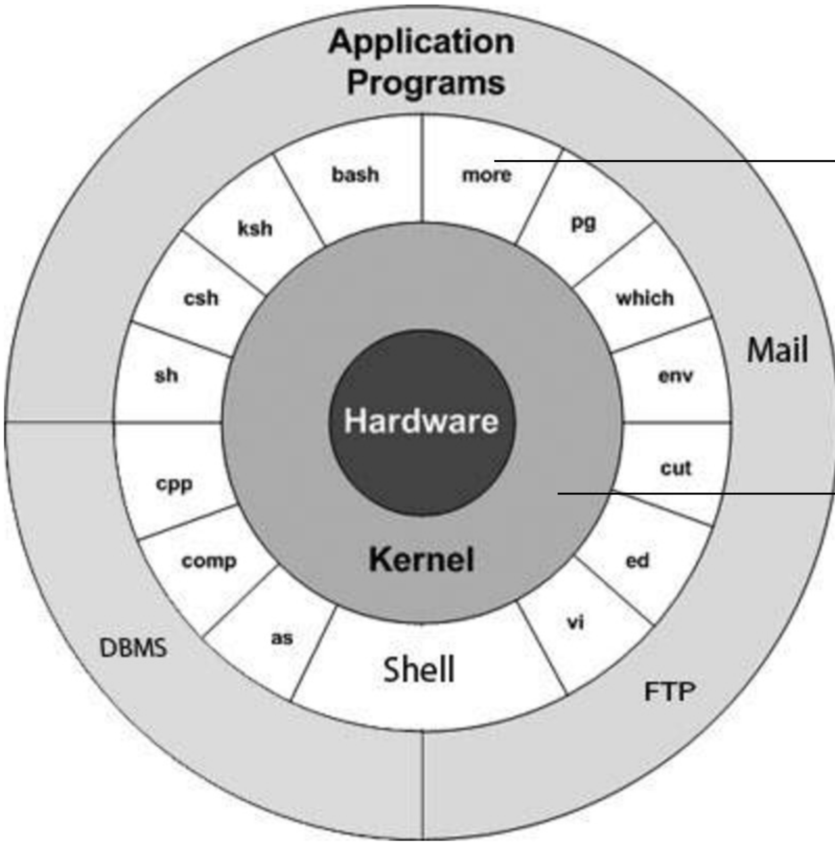
- UNIX: limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts:
  - Systems programs and libraries (run in user mode)
  - The kernel (runs in kernel mode)
- The kernel consists of everything below the system-call interface and above the physical hardware. Provides:
  - file system,
  - CPU scheduling,
  - memory management,
  - and other operating-system functions.

  a large number of functions for one level (monolithic kernel)

Traditional UNIX System Structure



Unix OS architecture



System programs and System libraries (one layer)

Kernel (another layer)

From https://www.tutorialspoint.com

Linux OS architecture

Various layers within Linux, also showing separation between the userland and kernel space

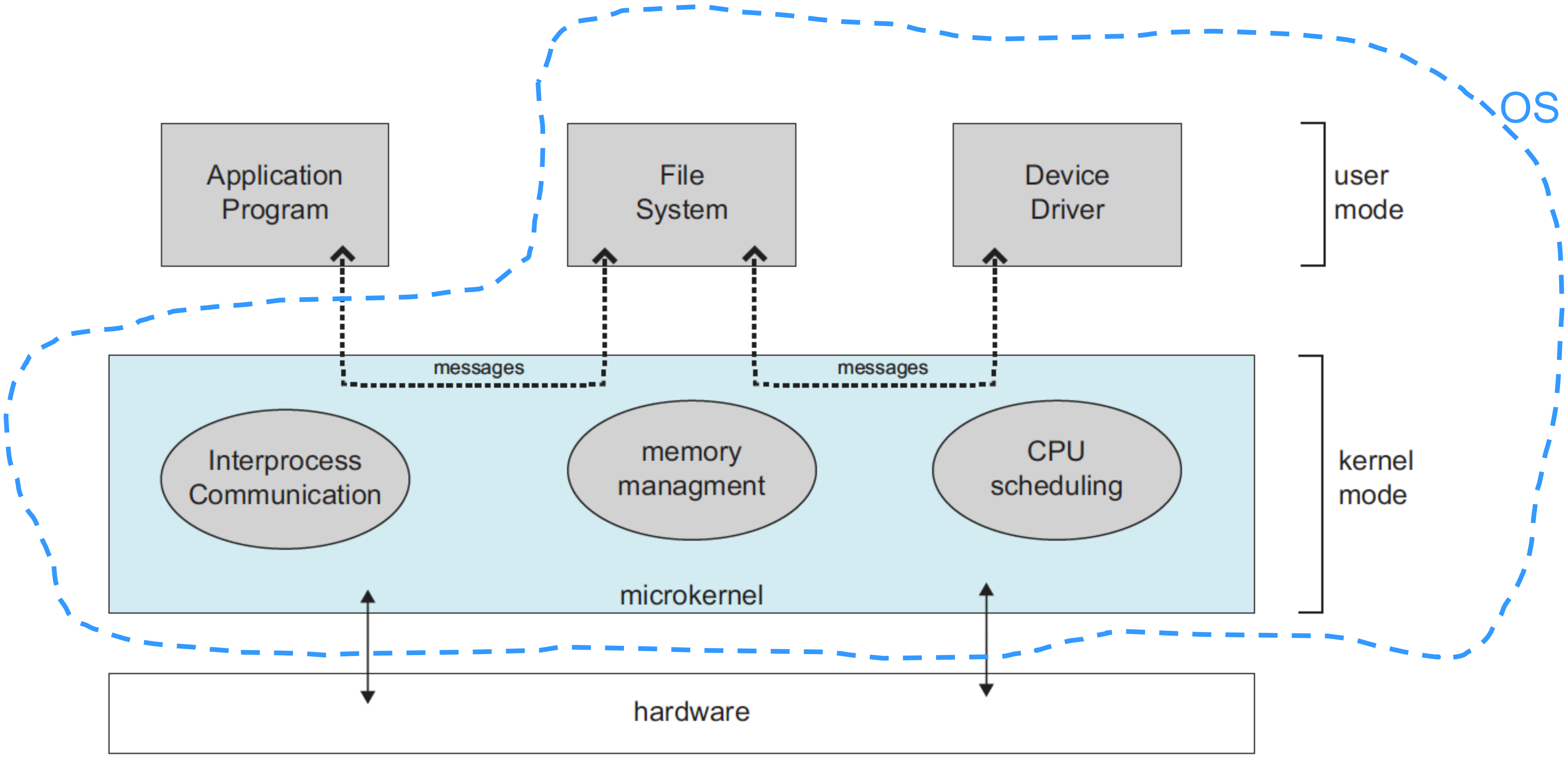
<table>
<tr><td rowspan="3">User mode</td><td>User applications</td><td colspan="5">For example, bash, LibreOffice, GIMP, Blender, 0 A.D., Mozilla Firefox, etc.</td></tr>
<tr><td>Low-level system components:</td><td>System daemons:
systemd, runit, logind, networkd, PulseAudio, ...</td><td>Windowing system:
X11, Wayland, Mir, SurfaceFlinger (Android)</td><td colspan="2">Other libraries:
GTK+, Qt, EFL, SDL, SFML, FLTK, GNUstep, etc.</td><td>Graphics:
Mesa, AMD Catalyst, ...</td></tr>
<tr><td>C standard library</td><td colspan="5"><code>open()</code>, <code>exec()</code>, <code>sbrk()</code>, <code>socket()</code>, <code>fopen()</code>, <code>calloc()</code>, ... (up to 2000 subroutines)
glibc aims to be POSIX/SUS-compatible, uClibc targets embedded systems, bionic written for Android, etc.</td></tr>
<tr><td rowspan="3">Kernel mode</td><td rowspan="3">Linux kernel</td><td colspan="5"><code>stat</code>, <code>splice</code>, <code>dup</code>, <code>read</code>, <code>open</code>, <code>ioctl</code>, <code>write</code>, <code>mmap</code>, <code>close</code>, <code>exit</code>, etc. (about 380 system calls)
The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)</td></tr>
<tr><td>Process scheduling subsystem</td><td>IPC subsystem</td><td>Memory management subsystem</td><td>Virtual files subsystem</td><td>Network subsystem</td></tr>
<tr><td colspan="5">Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter
Linux Security Modules: SELinux, TOMOYO, AppArmor, Smack</td></tr>
<tr><td colspan="7">Hardware (CPU, main memory, data storage devices, etc.)</td></tr>
</table>

from wikipedia

Microkernel System Structure

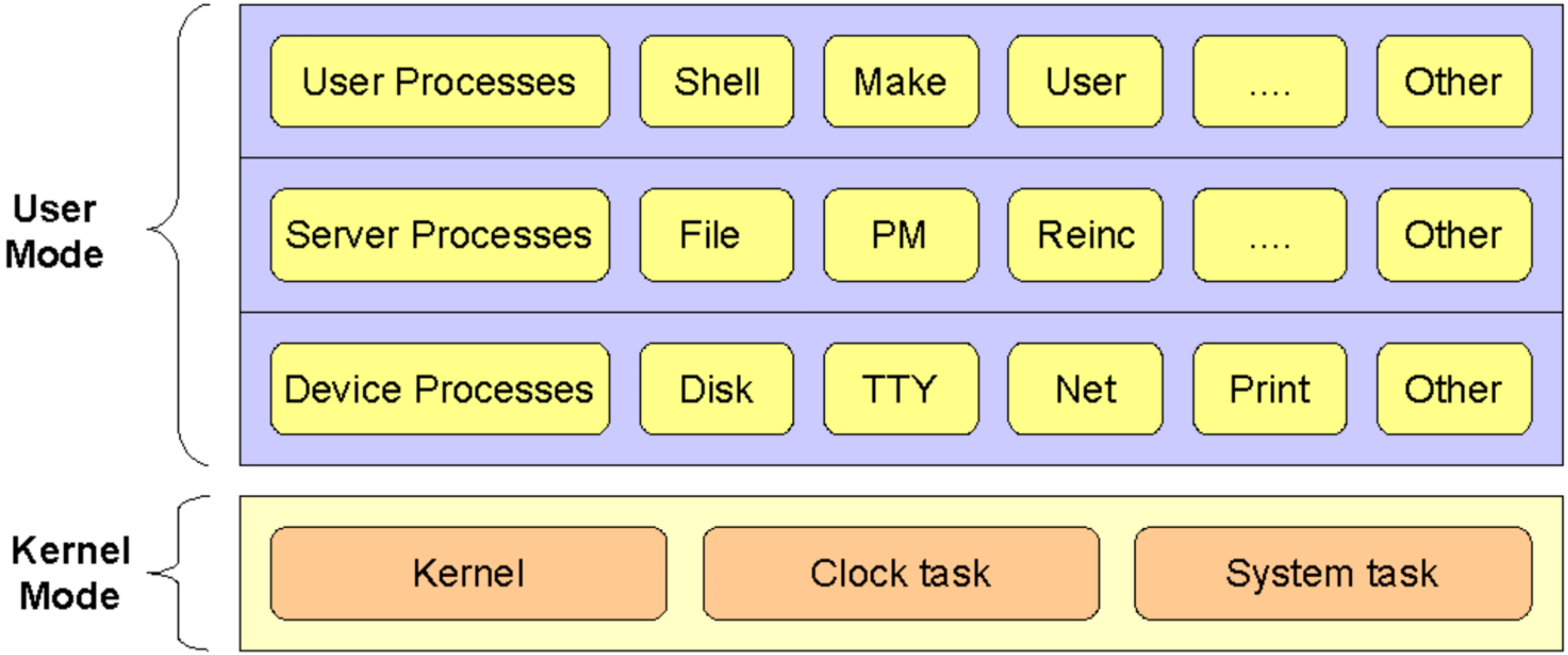
- Moves as much from the *kernel* into *user* space
- Communication takes place between user-mode OS processes using message passing.
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication.

Microkernels



Architecture of a typical microkernel OS

MINIX3 OS architecture
a microkernel operating system

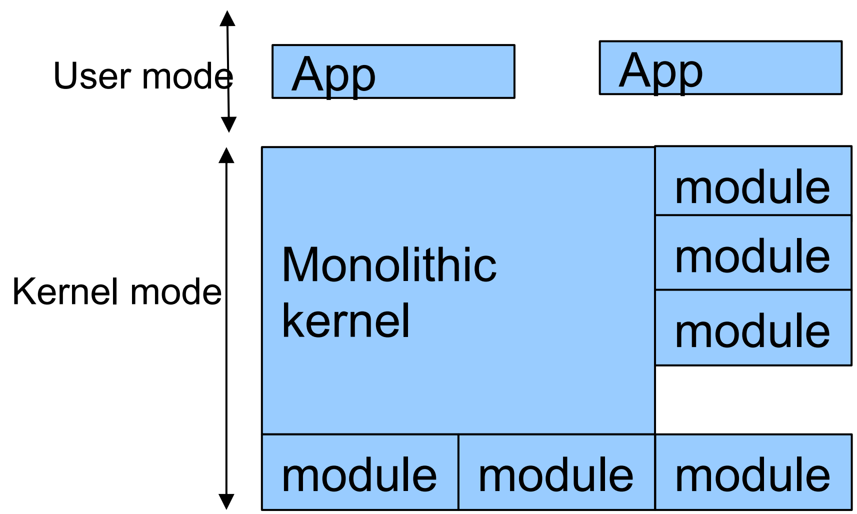


The MINIX 3 Microkernel Architecture

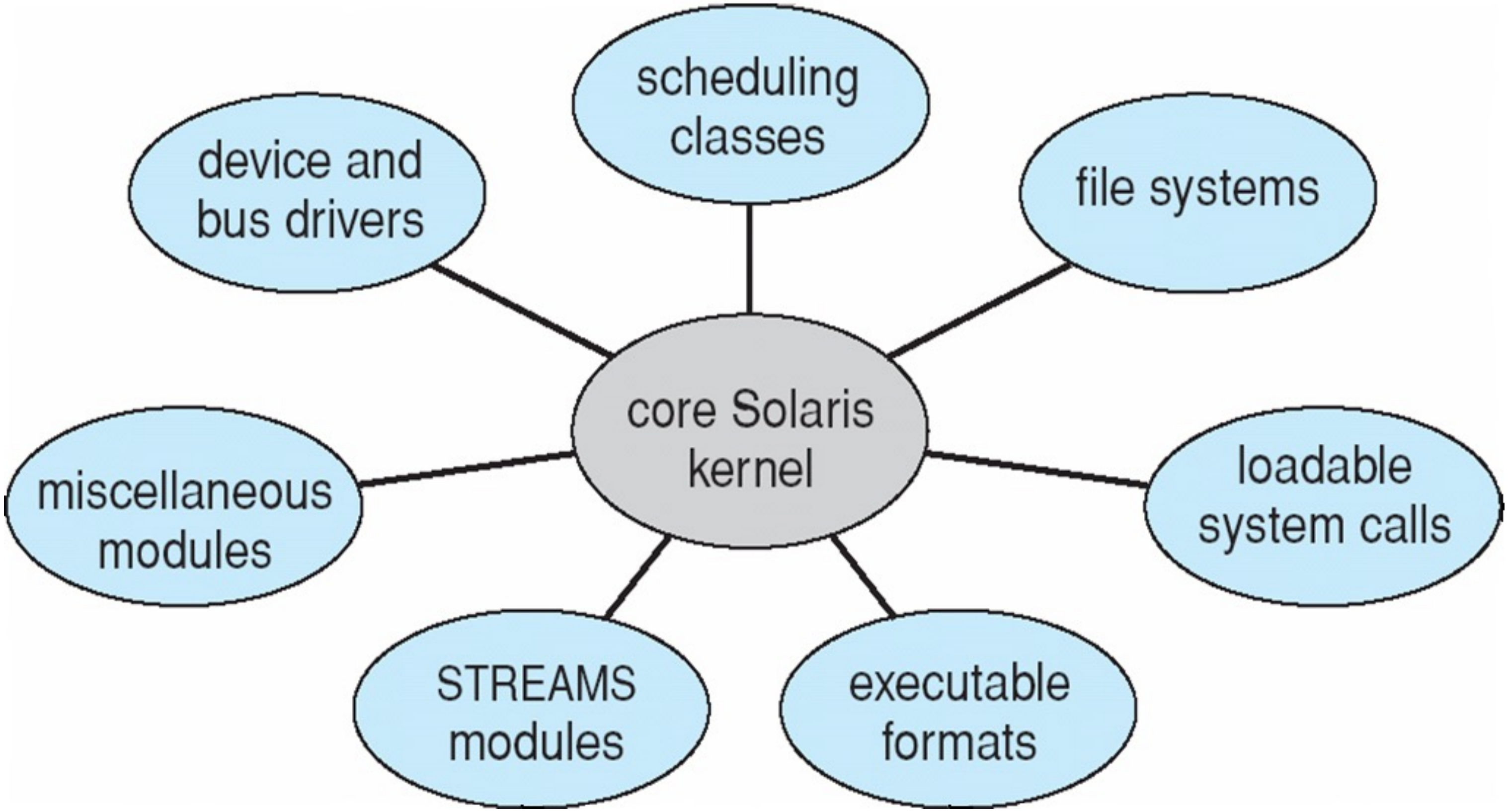
from wikipedia

Modules

- Most modern operating systems implement kernel modules. In this way they improve their monolithic nature.
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces (function calls in the same address space)
  - Each is loadable as needed within the kernel
- Overall, similar to layers but more flexible.
- Linux supports modules.
- You can insert modules.
- You can remove modules.

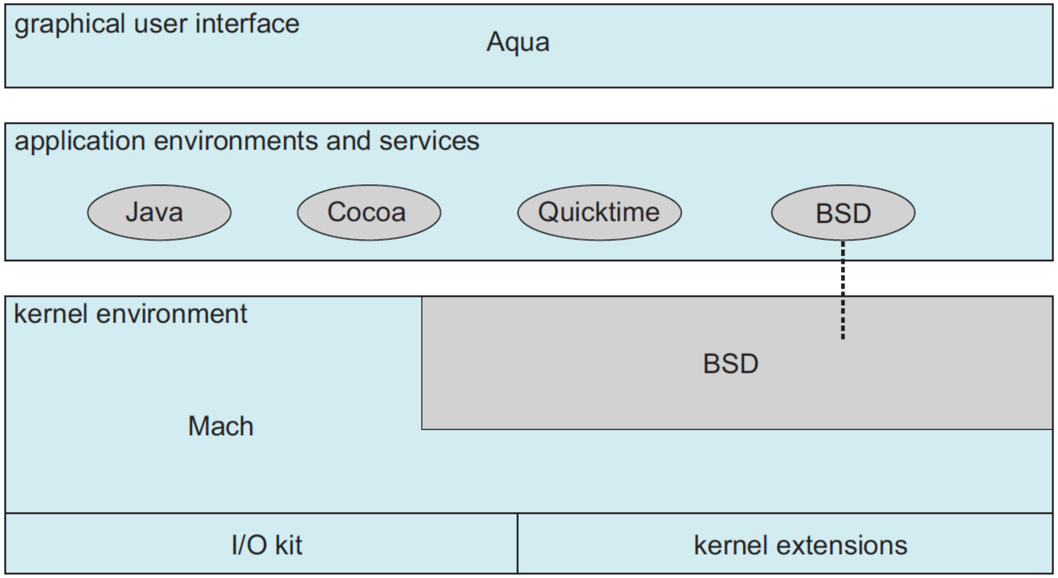


Solaris Modular Approach



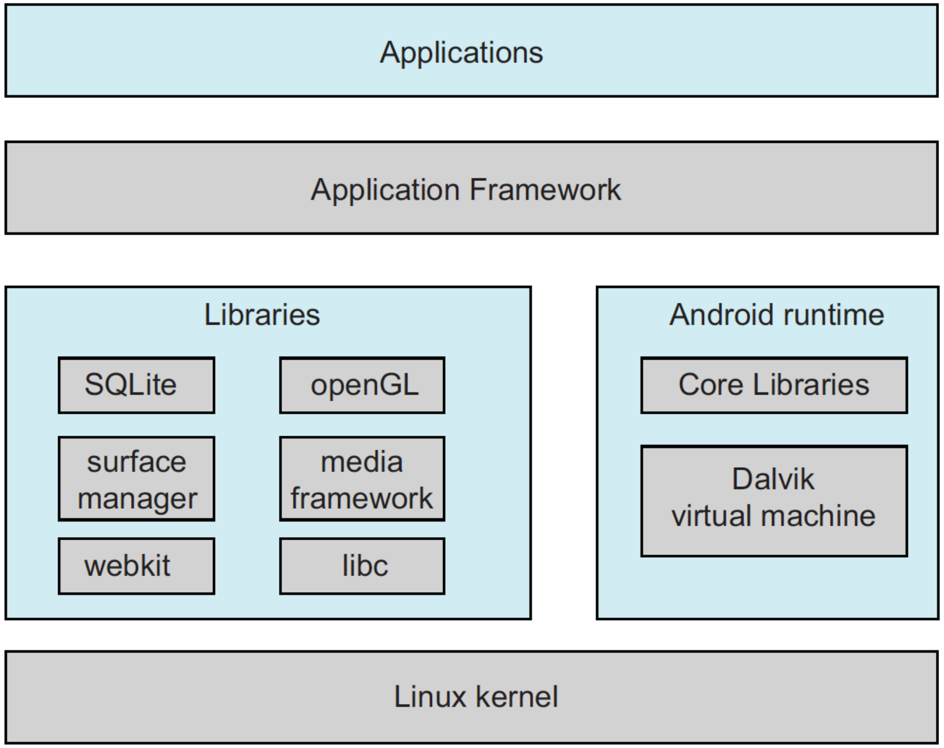
Hybrid Systems

- Mac OS X
  - Uses Mach and BSD kernels
  - Mach: microkernel
  - BSD: monolithi



Hybrid Systems

- Android
  - Uses Linux kernel
  - Layered stack of software on Linux kernel.
- A rich set of frameworks to develop and run Android applications.



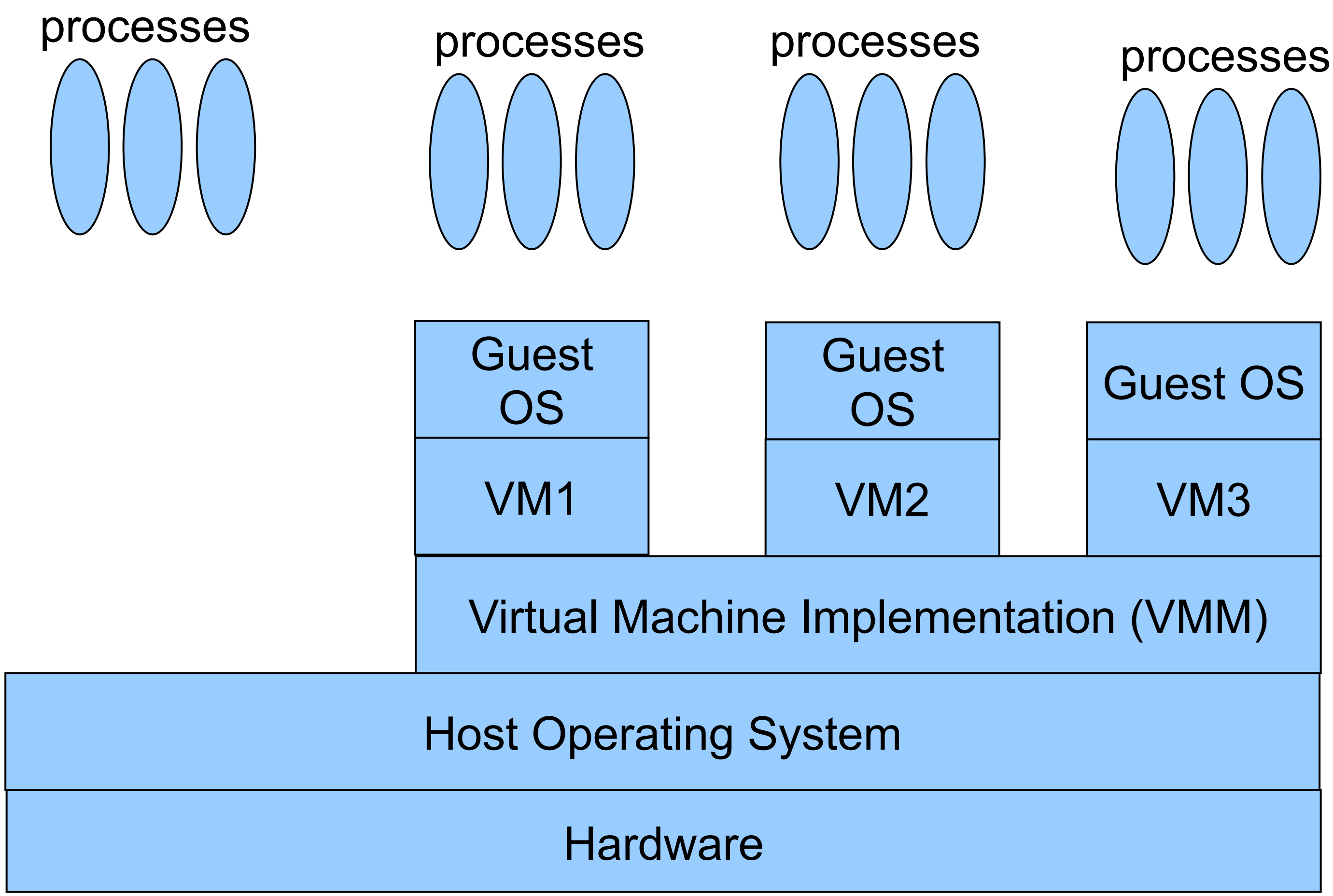
Virtualization

- Virtualization abstracts/virtualizes the hardware so that OSs and programs can run on a virtualized machine or a virtual environment
- Many different types of virtualizations:
  - Virtual machines: run OS/Apps on a VM that is identical to the bare hardware
    - For example: run Linux OS compiled for Intel x86 on a virtual machine duplicating x86 hardware.
  - Emulation: run OS or App developed for a machine that is different than the bare hardware
    - For example: run app developed for MIPS on Intel x86 machine using an emulator (MIPS emulation on Intel x86).
  - Abstract machines: run applications developed/compiled for an abstract machine running on bare hardware (interpretation)
    - For example JVM. Run a Java app on JVM running on x86

Virtual Machines

- Hardware is abstracted into several different execution environments.
  - Virtual machines.
- Each virtual machine provides an interface that is identical to the bare hardware.
- A *guest* process/kernel can run on top of a virtual machine.
  - We can run several operating systems on the same *host*.
  - Each virtual machine will run another operating system (guest).

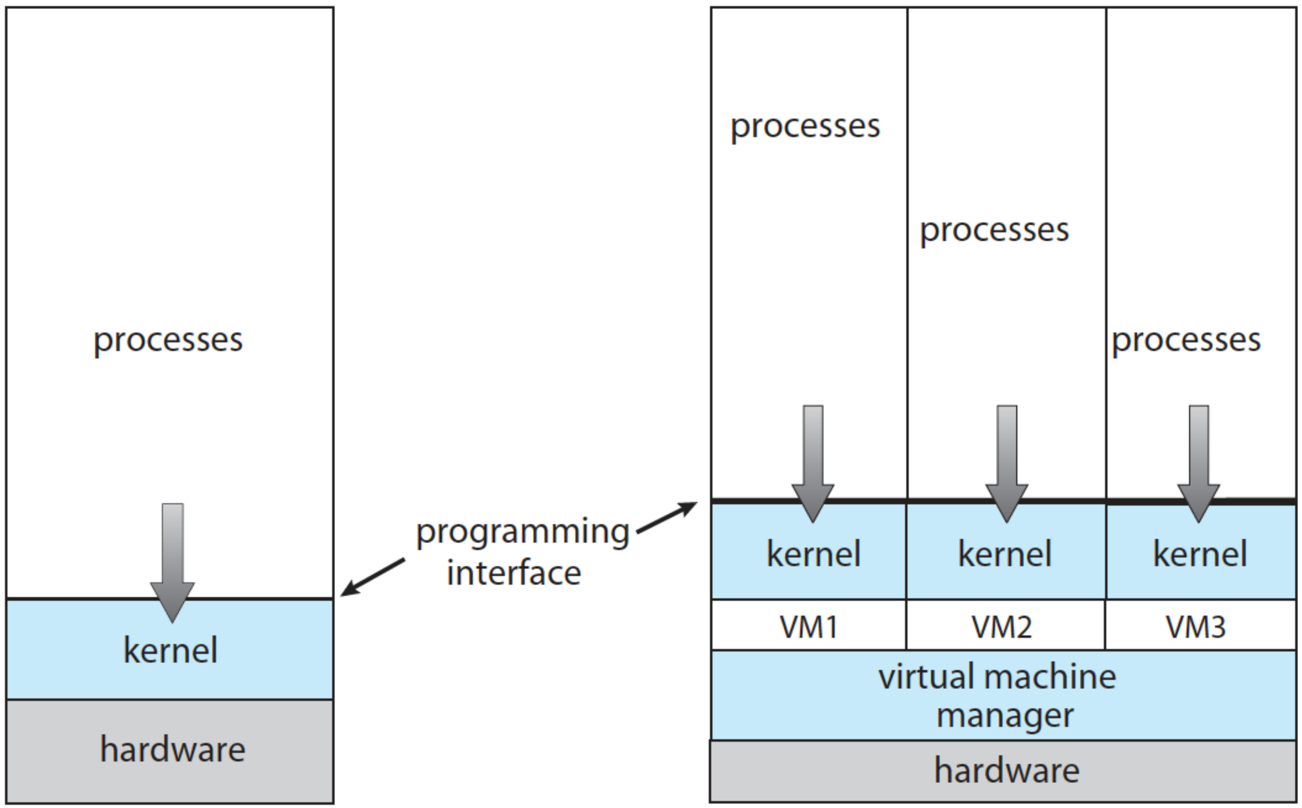
Virtual Machines



1) VMM is a *hosted virtual machine* (running on a host OS) – Type-2 Hypervisor

Example: WMware Workstation

Virtual Machines



(a) (b)

2) VMM is *running directly on hardware* like an OS: Type-1 Hypervisor

Example: Wmware ESX Hypervisor

Virtualization

- Vmware (Virtual Machine Manager– VMM)
  - Abstracts Intel x86 hardware
- Java virtual machine (JVM)
  - Specification of an abstract computer
- .NET Framework
  - Specification of an abstract computer again

Operating System Debugging

- Failure analysis
  - Log files
  - Core dump
  - Crash dump
- Performance tuning
  - Monitor system performance
    - Add code to kernel
    - Use system tools like “top”
- DTrace
  - Facility for dynamically adding probes to a running system (both to processes and to the kernel)
  - Probes can be queried using D programming language to obtain info

Operating System Generation

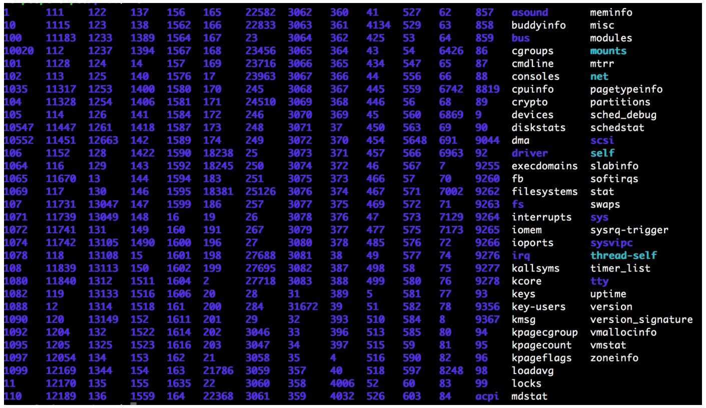
- Configure the kernel
- Compile the kernel

System Boot

- Bootstrap program (loader) locates the kernel, loads it and starts the kernel.
- This can be a two-step procedure.
    - Bootstrap program loads another more complex boot program
    - That boot program loads the kernel
- Then control is given to the kernel.
- Kernel starts the environment and makes the computer ready to interact with the user (via a GUI or command shell).
- Details depend on the system.

Performance Monitoring and Tuning

- We may need to monitor system performance.
- There are tools for that
- In Linux, for exmaple,
  - ps
  - top
  - vmstat (memory usage statistics)
  - netstat (network usage statistics)
  - iostat (I/O device usage statistics)



- In Linux, /proc file system gives kernel information (statistics)
  - cd /proc
    - we see lots of directories
    - one directory per process (named with process id – pid)

Summary

- System calls are the programming interface to kernel (used by programs)
- System programs are useful for users (a nice environment for program development and running)
- Different approaches to structure kernel
  - monolithic
  - layered
  - microkernel
- Virtualization is important
  - Has various forms.

References

- Operating System Concepts, Silberschatz et al.
- Modern Operating Systems, Andrew S. Tanenbaum et al.
- Computer Systems, A Programmer's Perspective, 3rd edition, R. E. Bryant and D. R. O'Hallaron, Pearson, 2016.

Additional Study Material