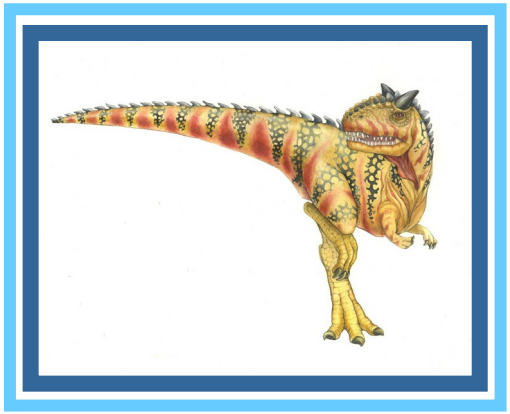
Theoretical Unit 1

(Book Chapters 1+2)

Introduction

(edited & enhanced by rufino@ipb.pt, 2025/2026)



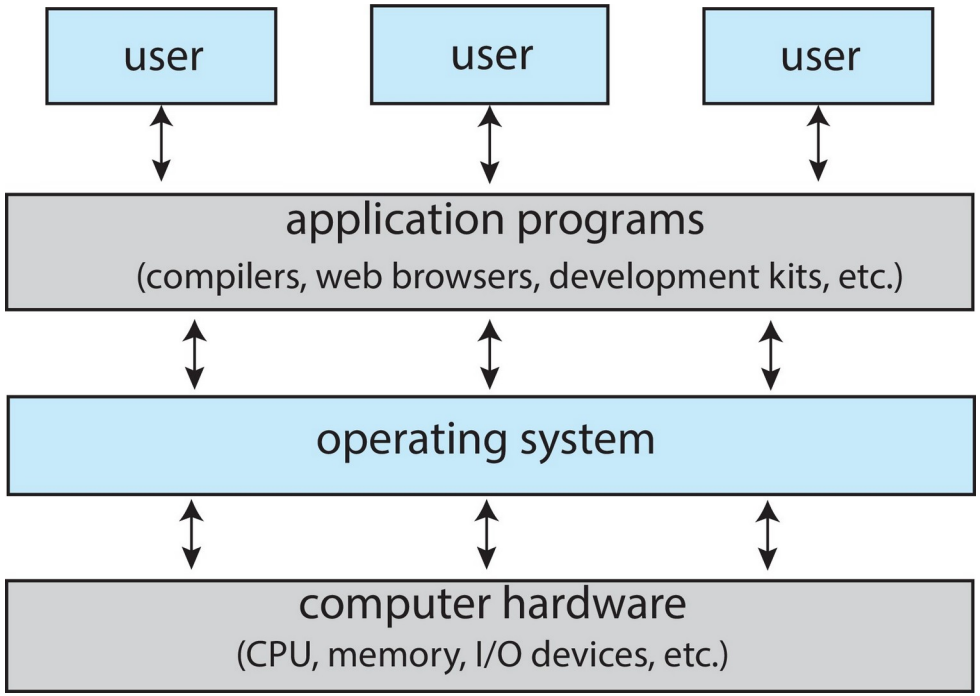
Theoretical Unit 1: Introduction

- Computer-System Structure
- Hardware Organization and Operation
- Operating System Definition
- Operating System Components
- System Programs and Applications
- Hardware Protection
- Operating System Structure
- Operating System Development
- Operating System Debugging and Monitoring
- Supplemental Slides

Theoretical Unit 1

1.1 Computer-System Structure

Computer-System Structure



Computer-System Structure

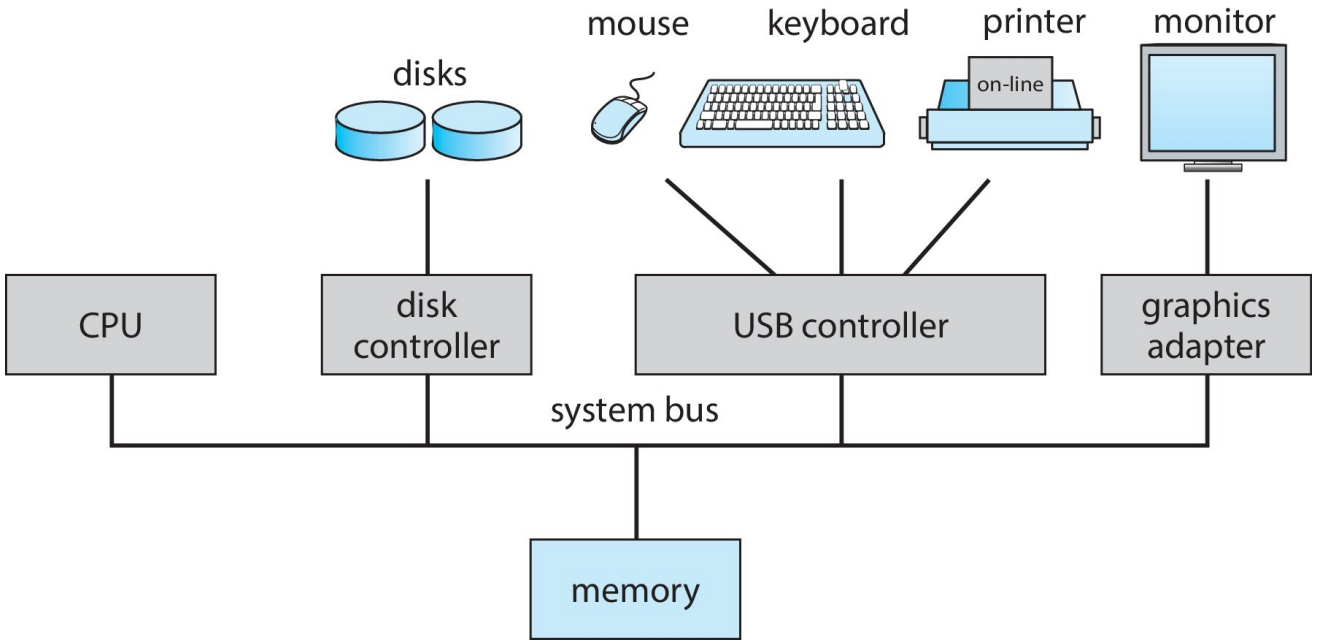
- Computer system can be divided into four components:
  - **Hardware** – provides basic computing resources
    - CPU, memory, I/O devices
  - **Operating system** – controls and coordinates use of hardware among various applications and users
  - **Application programs** – define ways in which the system resources are used to solve the users problems
    - Word processors, compilers, web browsers, database systems, video games, ...
  - **Users**
    - People, machines, other computers

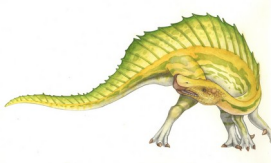
Theoretical Unit 1

1.2 Hardware Organization and Operation

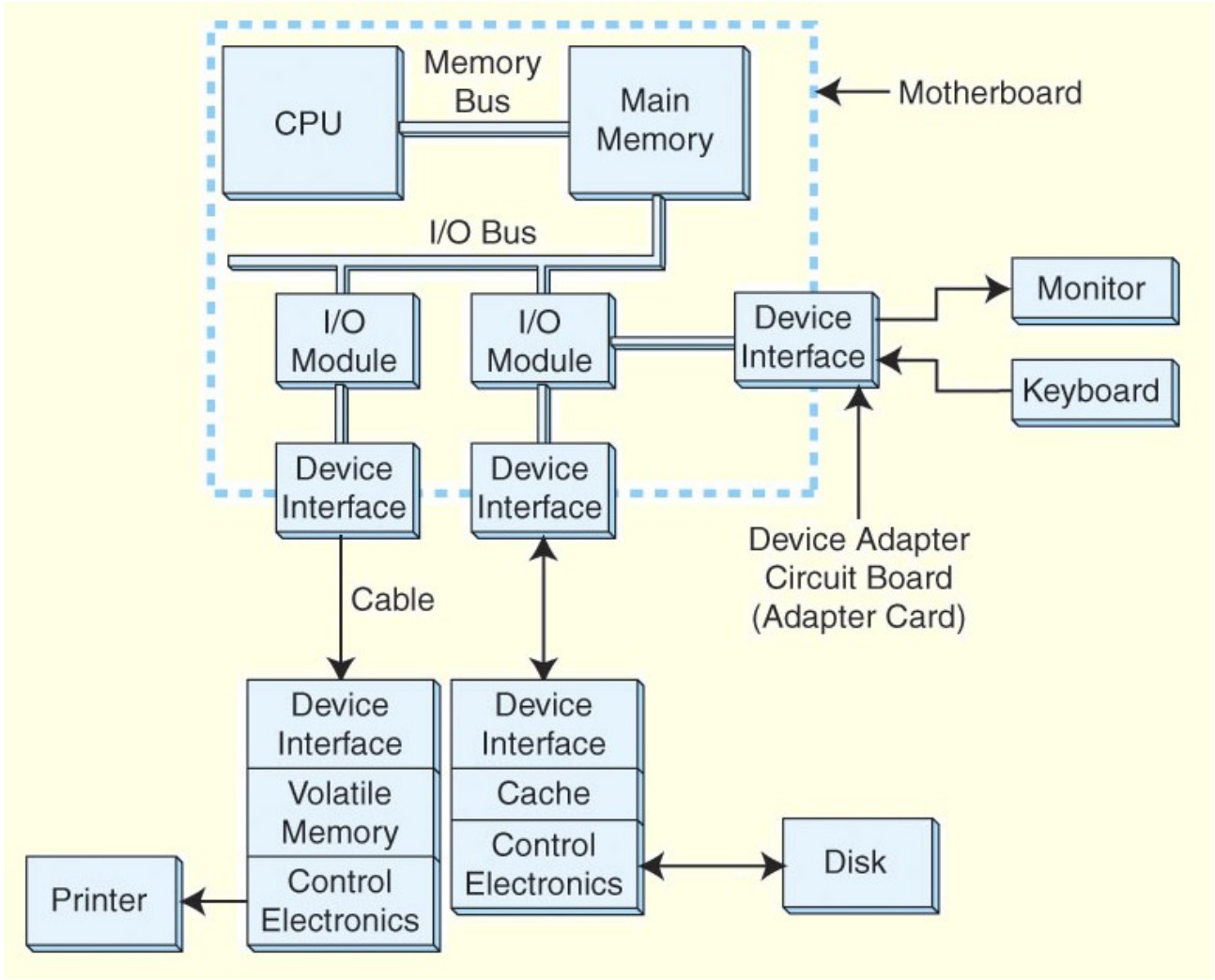
Hardware Organization and Operation

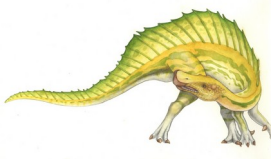
- CPU, memory, device controllers and devices, all interconnected (representation bellow, with 1 shared bus, is simplified; see next slide)
- Concurrent operation of CPU and devices competing for memory cycles





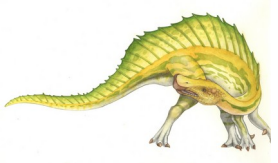
Hardware Organization and Operation





Hardware Organization and Operation

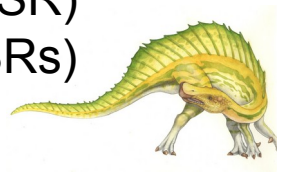
- I/O devices and the CPU can execute concurrently
- Each device controller
  - is in charge of a particular device type
  - has a local buffer
  - has an operating system **device driver** to manage it
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an **interrupt**
- Some devices are operated by a **DMA** controller
- Some devices require **polling** (querying)



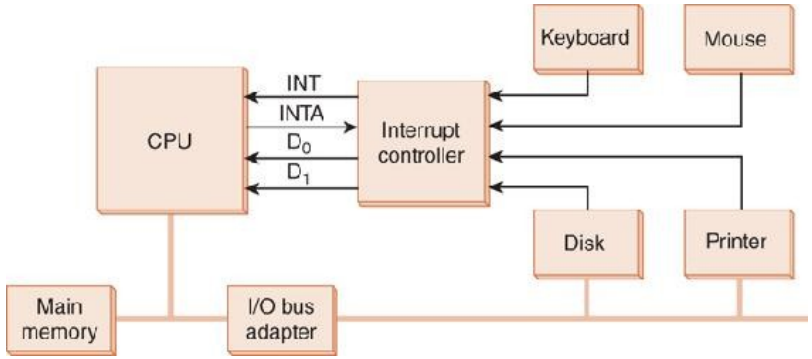
Interrupts

- Interrupts may be generated:
  - by hardware components (e.g., keyboard, mouse, NIC, …)
  - by the CPU itself (e.g., invalid memory access)
  - by the software; these interrupts are called **traps** or **exceptions**
    - this is the mechanism underlying the **system calls** !

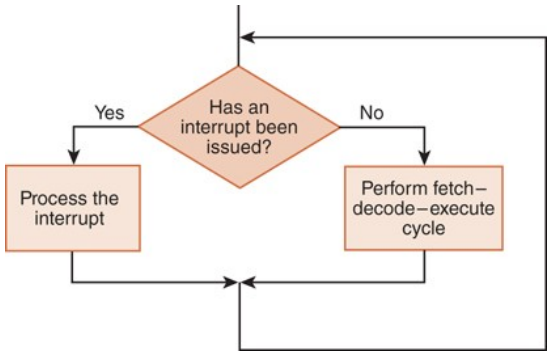
- Interrupts handling:
  - To later allow the interrupted program to continue, the OS preserves the state of the CPU by storing registers (program counter, etc.)
  - Determines which type of interrupt has occurred:
    - **polling**
    - **vectored** interrupt system
  - Interrupt transfers control to the **interrupt service routine** (ISR) via the **interrupt vector** (contains the addresses of all the ISRs)



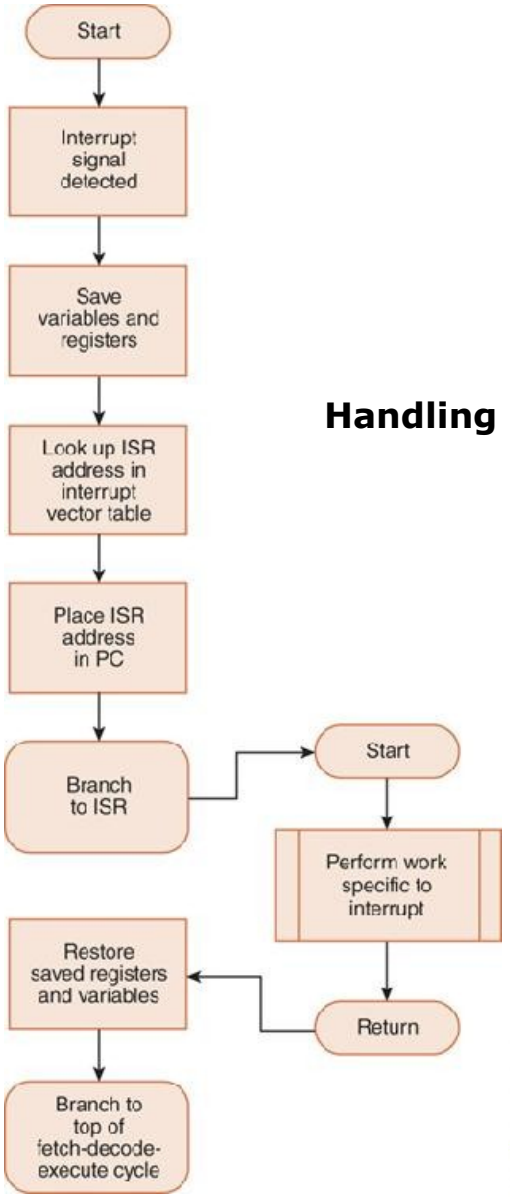
Interrupts



Generation and Identification



Detection

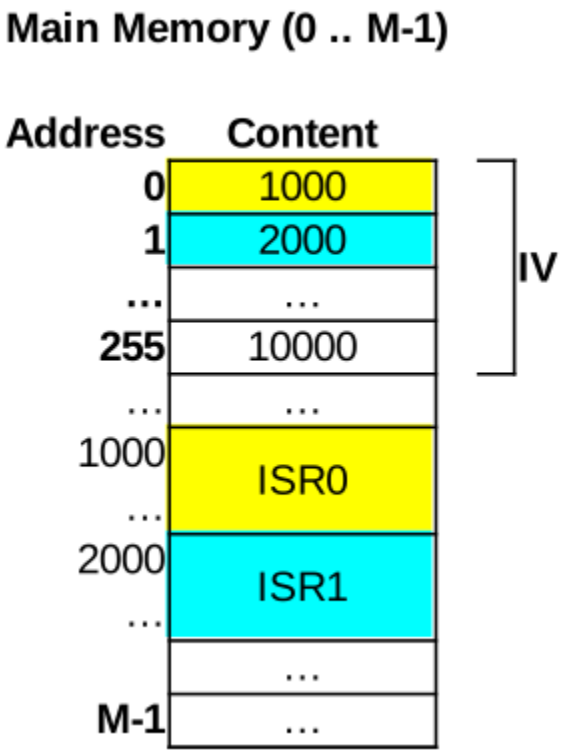


Handling

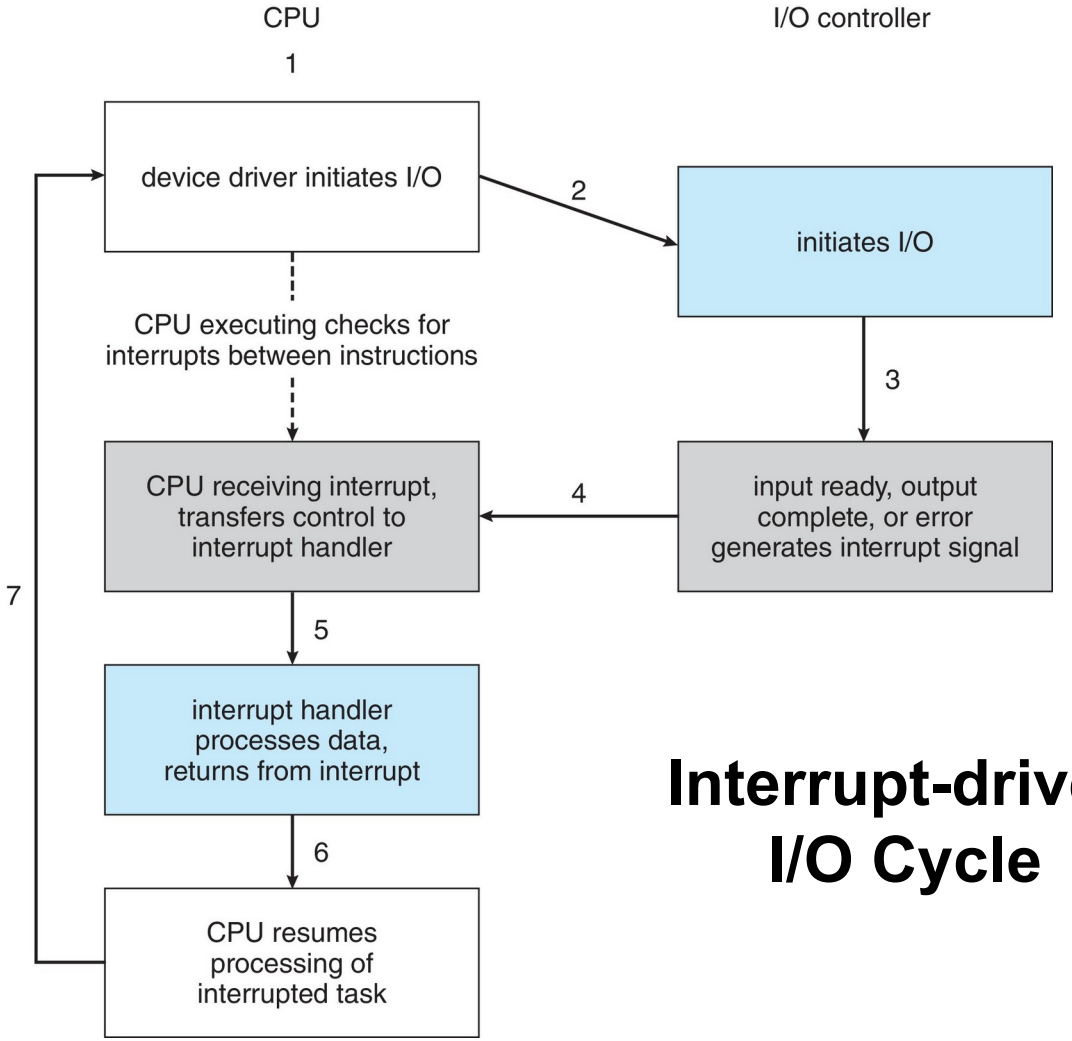
Interrupts

Intel x86 Interrupt Vector

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts



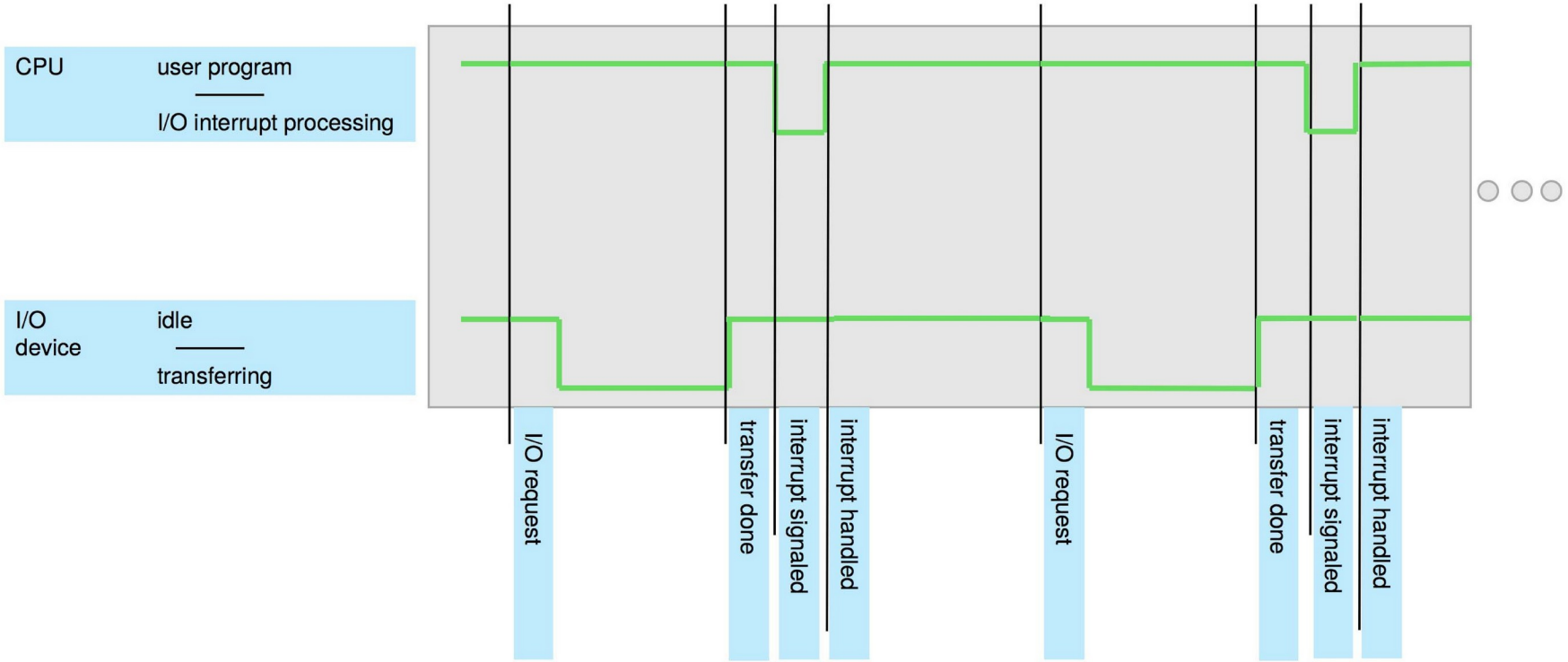
Interrupts



Interrupt-driven I/O Cycle

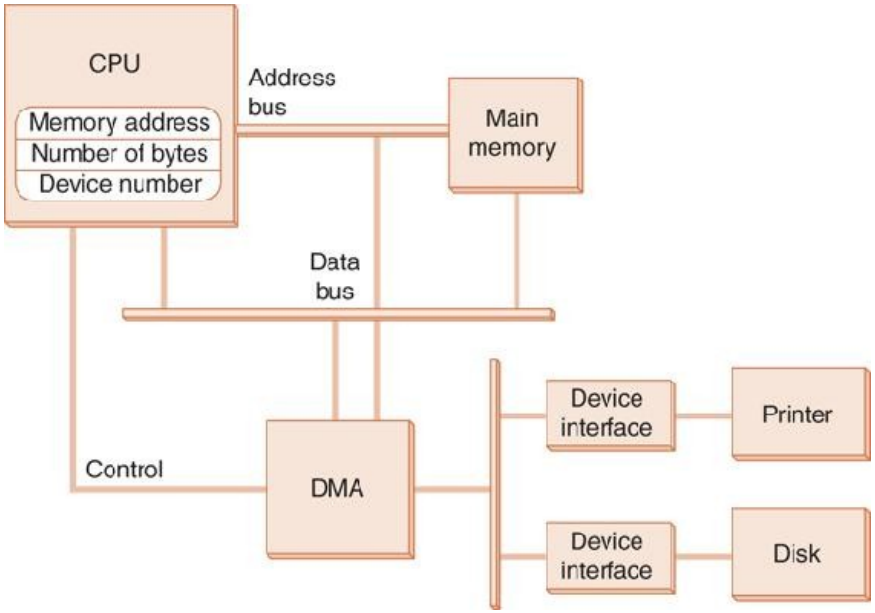
Interrupts

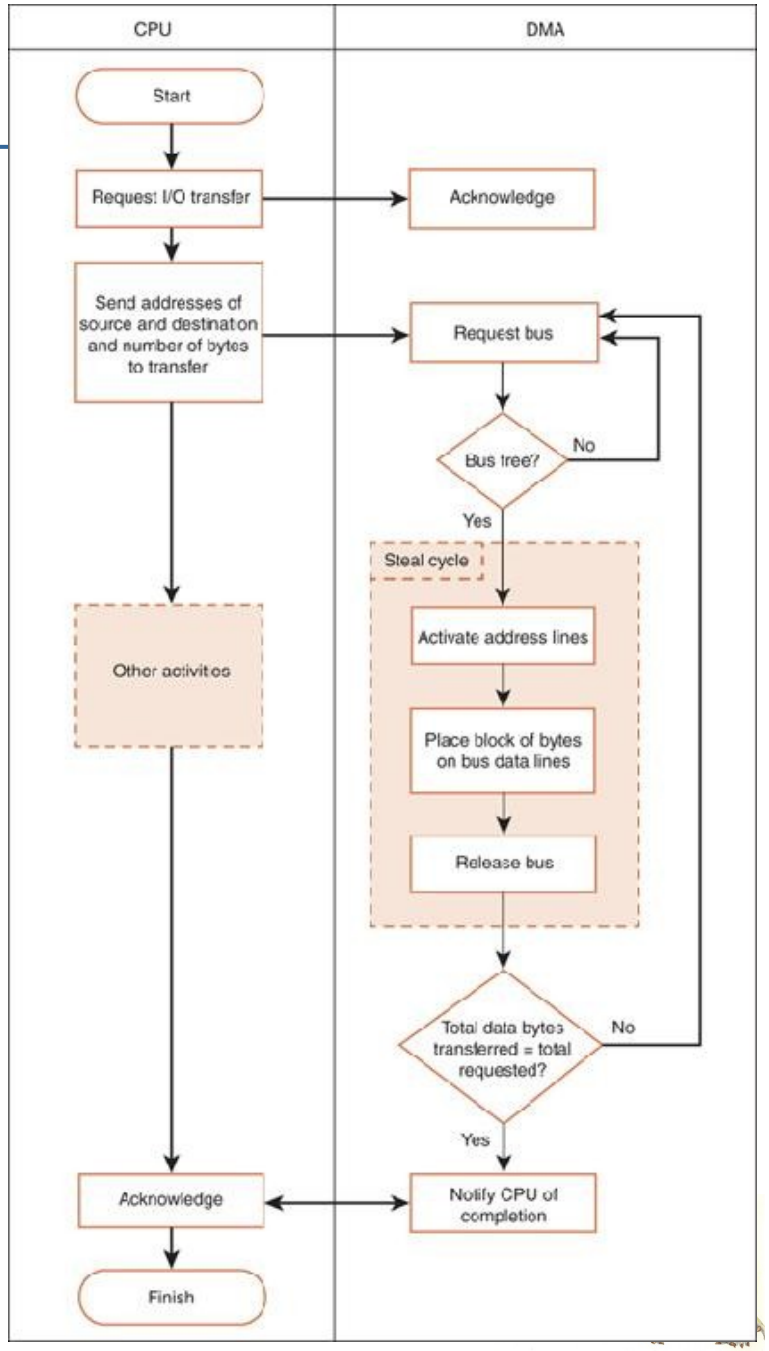
Interrupt-driven I/O Cycle (Timeline)



Direct Memory Access

- Used for high-speed I/O devices able to transmit data at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

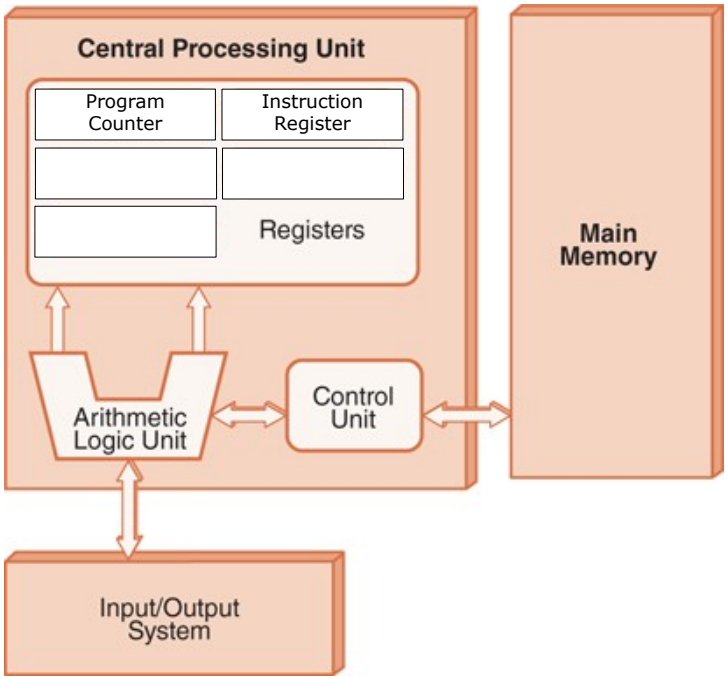




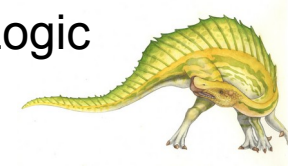
Instruction Execution

- **von Neumman execution model**

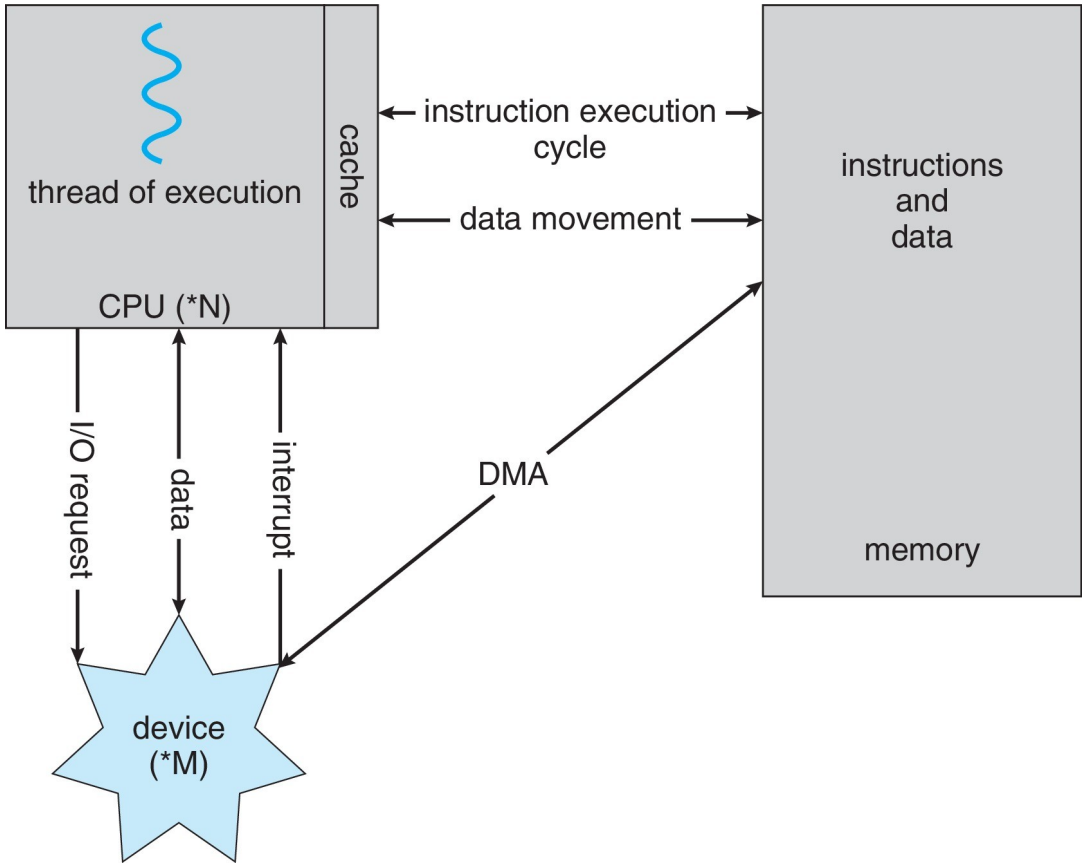
  (stored-program architecture)



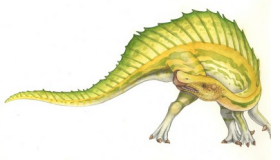
- **fetch**: Control Unit (CU) brings to the Instruction Register the content (instruction) in the Main Memory address defined in the Program Counter
- **decode**: CU decodes the instruction, identifying the opcode and the operands; this may imply further Main Memory accesses
- **execute**: the instruction is executed, e.g. using the Arithmetic Logic Unit; the results may need to be stored back in Main Memory



How a Modern Computer Works



A von Neumann architecture



Theoretical Unit 1

1.3 Operating System Definition

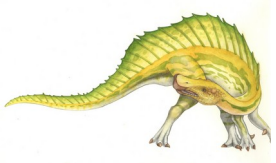
Defining Operating Systems

- Term “Operating System” covers many roles
  - Because of myriad designs and uses of OSes
  - Present in toasters through ships, spacecraft, game machines, TVs and industrial control systems
  - Born when fixed use computers for military became more general purpose and needed resource management and program control

- An Operating System
  - is an **intermediary** between a user and the computer hardware
  - provides an adequate **environment** for the execution of programs
  - executes user programs and makes solving user problems easier
  - makes the computer system **convenient** to use
  - uses the computer hardware in an **efficient** manner

What Operating Systems Do

- Depends on the point of view
- Users want **convenience** (**ease of use**) and **good performance**
  - Don't care about **resource utilization** (**efficiency**)
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy
  - OS is a **resource allocator** and **control program** making **efficient** use of HW and managing execution of user programs
- Users of dedicate systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- Mobile devices like smartphones and tablets are resource poor, optimized for usability and battery life
  - Mobile user interfaces such as touch screens, voice recognition
- Some computers have little or no user interface, such as embedded computers in devices and automobiles
  - Run primarily without user intervention



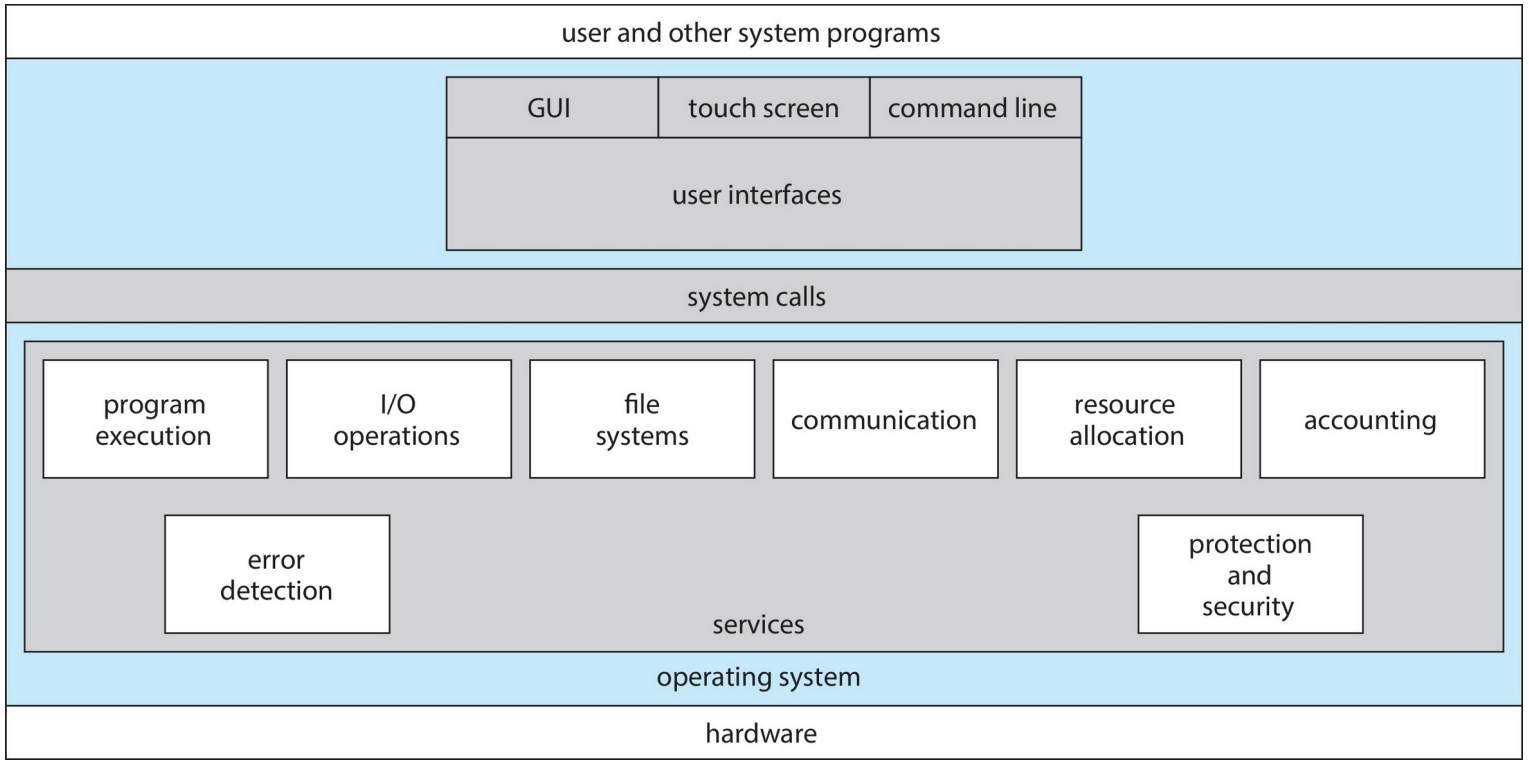
What makes up the Operating System

- No universally accepted definition
- "Everything a vendor ships when you order an operating system" is a good approximation
  - But varies wildly
- "The one program running at all times on the computer" is the **kernel,** part of the operating system
- Everything else is either
  - a **system program** (ships with the operating system, but not part of the kernel), or
  - an **application program**, all programs not associated with the operating system
- Today's OSes for general purpose and mobile computing also include **middleware** – a set of software frameworks that provide addition services to application developers such as databases, multimedia, graphics

Theoretical Unit 1

1.4 Operating System Components

Operating System Components



Operating System Services

- Operating systems provide several services to programs and users.
- Some OS services provide functions that are **convenient** to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between i) **Command-Line (CLI)**, including support for **Batch** files (shell scripts), ii) **Graphics User Interface (GUI)**, **iii) touch-screen, voice recognition**
  - **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.)

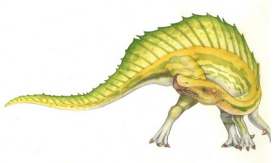
- Some OS services provide functions that are **convenient** to the user:
  - **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
  - **Communications** – Processes may exchange information, 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, memory, I/O devices, user program, ...
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont.)

- Other OS functions exist for ensuring the **efficient** operation of the system:
  - **Resource allocation -** With 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.
  - **Logging / 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** ensures that all access to system resources is controlled
    - **Security** requires user authentication, extends to defending external I/O devices from invalid access attempts

User Operating System Interface - CLI

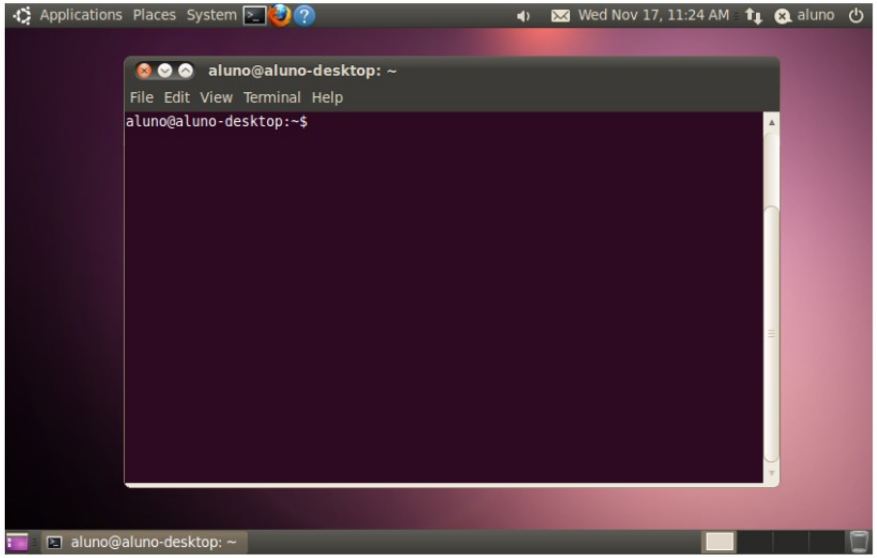
- CLI or **command interpreter** allows direct command entry
  - Implemented in kernel, or by system program
  - Simple algorithm: 1) read command(s); 2) parse command(s); 3) execute command(s); 4) go back to 1)
  - Sometimes multiple flavors implemented – **shells**
    - Unix / Linux: Bourne shell, Bourne-Again Shell (bash), C shell, K shell; cat /etc/shells (valid login shells)
    - MS-DOS: command.com; Windows: cmd.exe, powershell
  - Sometimes commands built-in (**internal commands**), sometimes just names of programs (**external commands**)
  - **shell scripts**: executable text files with commands interpreted by the shell; allow task automation; usually non-interactive; syntax and constructs like those of a programming language



BASH Command Interpreter

```
fso@fso-desktop:~$ date
Sat Oct  8 23:07:03 WEST 2011
fso@fso-desktop:~$
fso@fso-desktop:~$ ls -la *.c
-rw-r--r-- 1 fso fso 76 2011-10-08 23:05 ola_mundo.c
fso@fso-desktop:~$
fso@fso-desktop:~$ gcc ola_mundo.c -o ola_mundo.exe
fso@fso-desktop:~$
fso@fso-desktop:~$ ola_mundo.exe
hello world
fso@fso-desktop:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:.
fso@fso-desktop:~$
fso@fso-desktop:~$ cp ola_mundo.c  ola_mundo.c.bak
fso@fso-desktop:~$
fso@fso-desktop:~$ rm ola_mundo.exe
fso@fso-desktop:~$
fso@fso-desktop:~$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
fso@fso-desktop:~$
```

BASH session in a text-mode terminal



BASH GUI terminal

```
#!/bin/bash
for jpg in "$@" ; do                          # use $jpg in place of each filename given, in turn
    png="${jpg%.jpg}.png"                     # find the PNG version of the filename by replacing .jpg with .png
    echo converting "$jpg" ...                # output status info to the user running the script
    if convert "$jpg" jpg.to.png ; then       # use the convert program (common in Linux) to create the PNG in a temp file
        mv jpg.to.png "$png"                  # if it worked, rename the temporary PNG image to the correct name
    else                                      # ...otherwise complain and exit from the script
        echo 'error: failed output saved in "jpg.to.png".' 1>&2
        exit 1
    fi                                        # the end of the "if" test construct
done                                          # the end of the "for" loop
echo all conversions successful               # tell the user the good news
```

BASH script

MS-DOS Command Interpreter

```
Starting MS-DOS...

HIMEM is testing extended memory...done.

C:\>C:\DOS\SMARTDRV.EXE /X
C:\>command

Microsoft(R) MS-DOS(R) Version 6.22
            (C)Copyright Microsoft Corp 1981-1994.

C:\>_
```

MS-DOS text-mode session

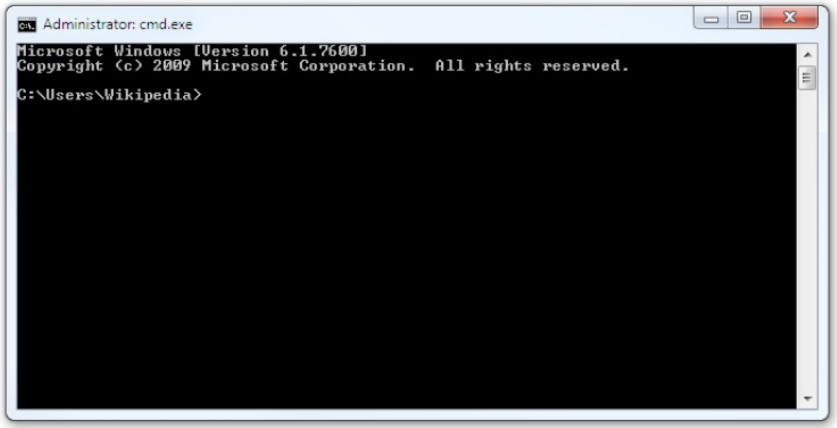
```
@echo off
prompt $P$G
PATH=C:\DOS;C:\WINDOWS
set TEMP=C:\TEMP
set BLASTER=A220 I7 D1 T2
goto %CONFIG%
:WIN
 lh smartdrv.exe
 lh mouse.com /Y
 win
goto END
:XMS
 lh smartdrv.exe
 lh doskey
 goto END
:END
```

autoexec.bat

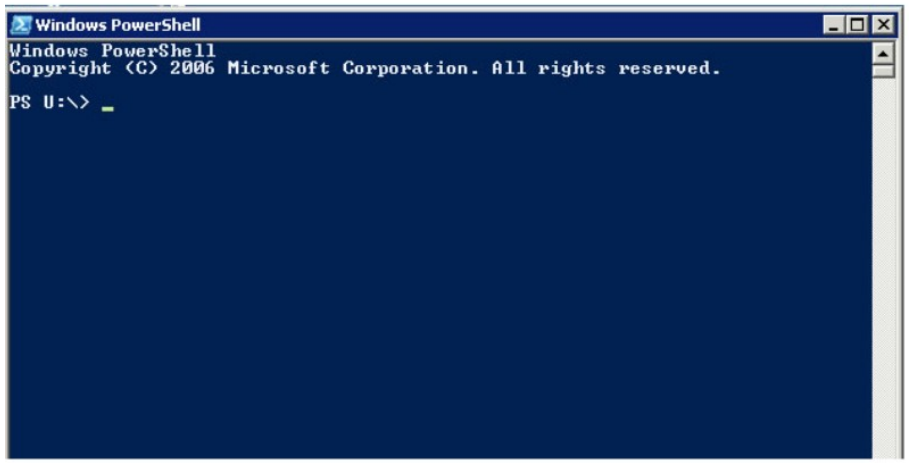
```
[menu]
 menuitem=WIN, Windows
 menuitem=XMS, DOS with only Extended Memory
 menudefault=WIN, 10
[common]
 device=c:\dos\himem.sys
 dos=high,umb
 shell=c:\dos\command.com c:\dos /e:512 /p
 country=44,437,c:\dos\country.sys
[WIN]
 device=c:\dos\emm386.exe ram
 devicehigh=c:\windows\mouse.sys
 devicehigh=c:\dos\setver.exe
[XMS]
 device=c:\dos\emm386.exe noems
```

config.sys

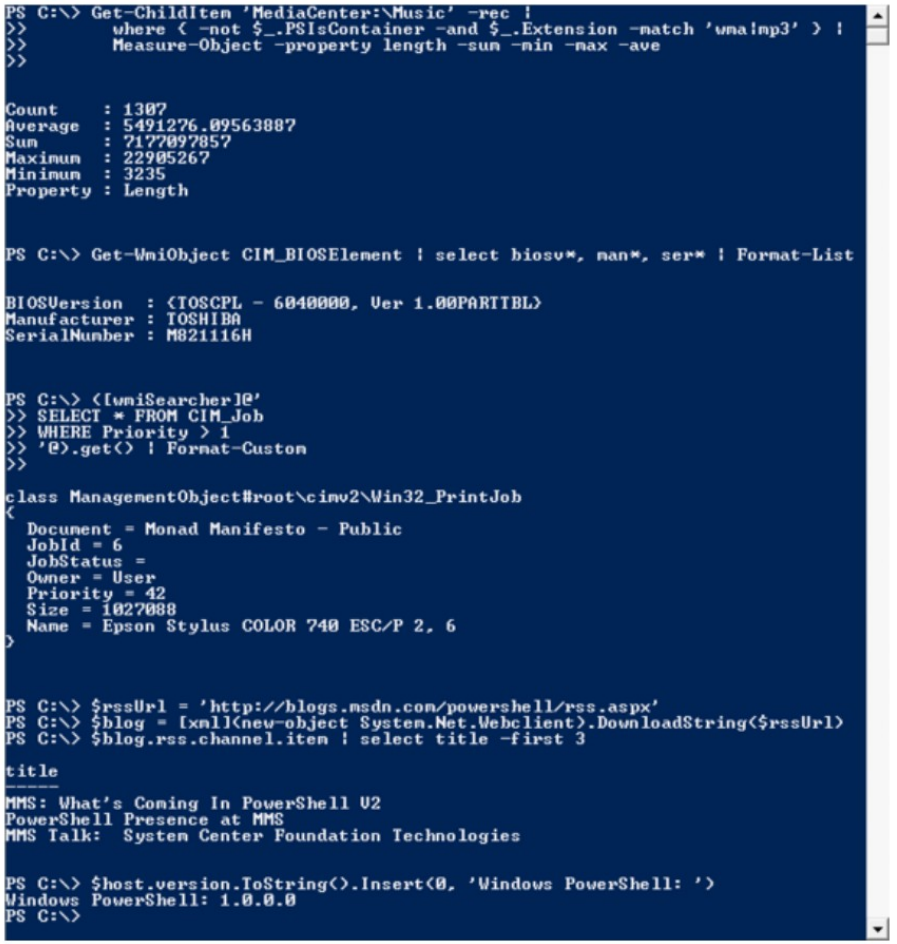
MS Windows Command Interpreters



cmd.exe



powershell



Powershell session example

User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**))
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (KDE, GNOME)
- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
  - Voice command

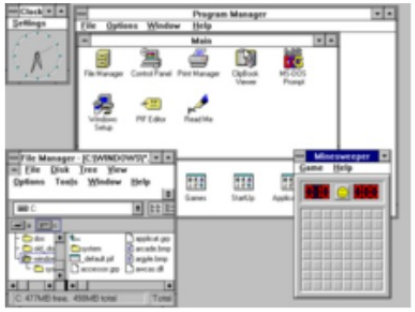
GUI Evolution – Apple and Microsoft

Xerox Alto (1973)

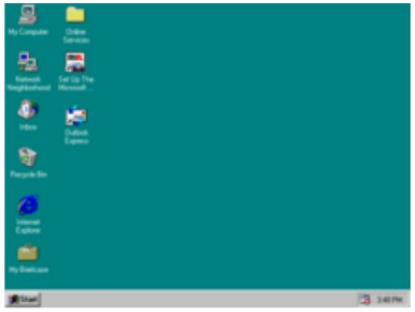
System 1.0 (1984)

Mac OS 9 (1999)

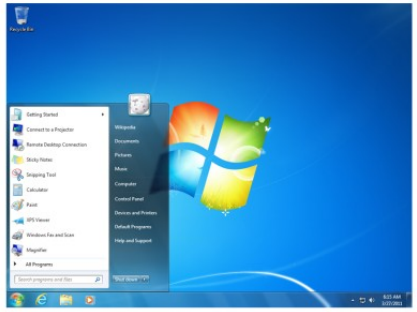
Mac OS X 10.5 (2007)



Windows 3.11 (1993)



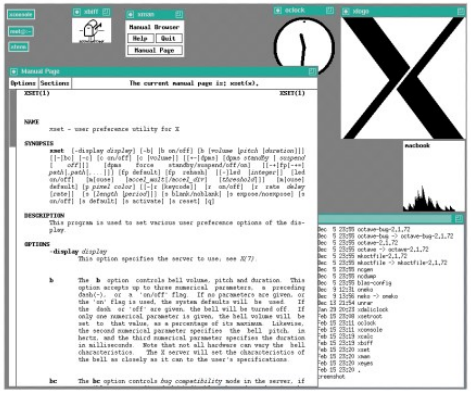
Windows 95 (1995)



Windows 7 (2009)

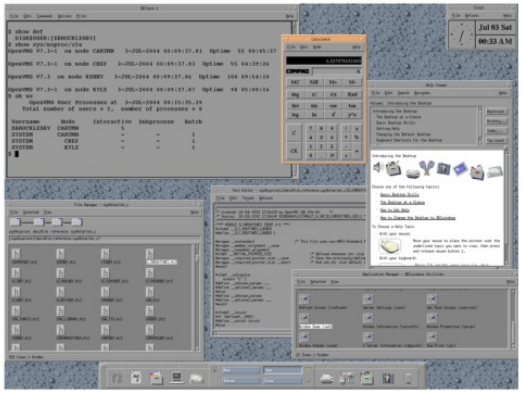
Windows 10 (2015)

GUI Evolution – Unix and Linux

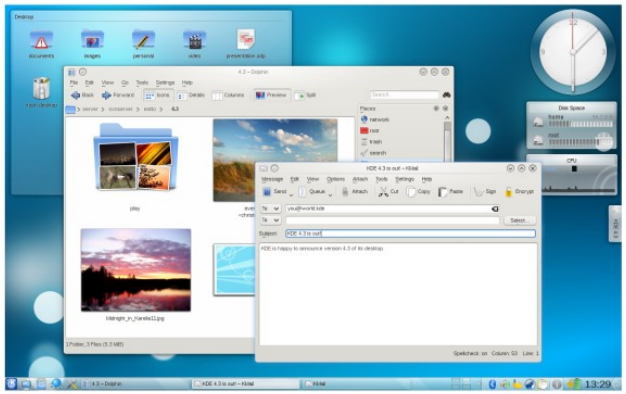


X Windows (>1984)

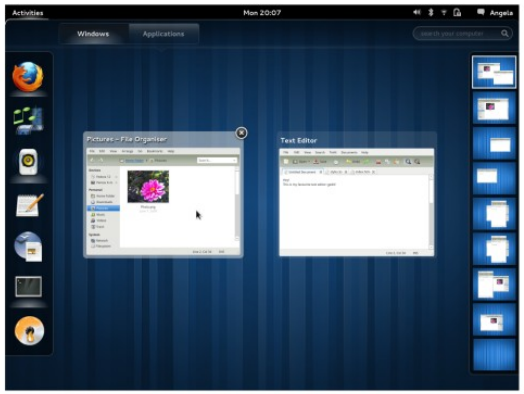
NeXTSTEP (1990)



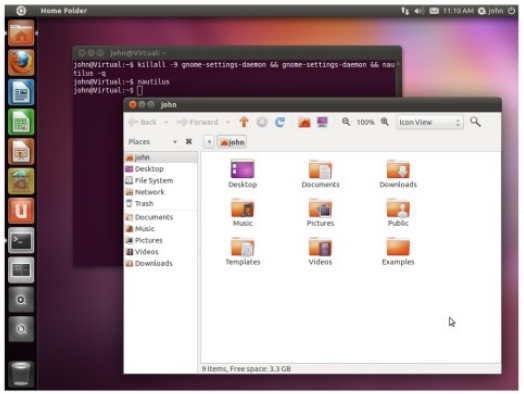
CDE (1995)



KDE 4.3 (2010)



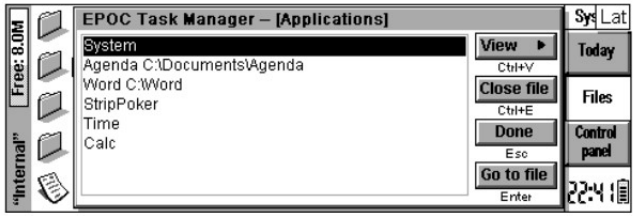
Gnome 3.2 (2011)



Unity (2011)

GUI Evolution – Handheld Systems

Apple Newton OS (1993)



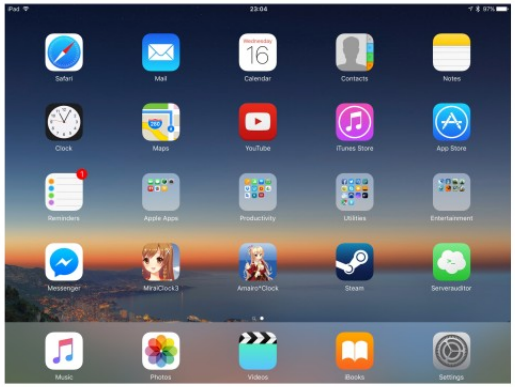
Psion EPOC32 (1997)

Palm OS 5 (2002)

Nokia Symbian (2005)

Windows Mobile 6.1 (2008) + 10 (2015)

Android 5 "Lollipop"(2014)

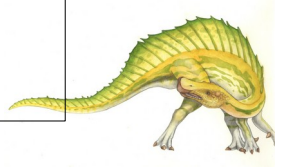


Apple iOS 9 (2015)

System Calls

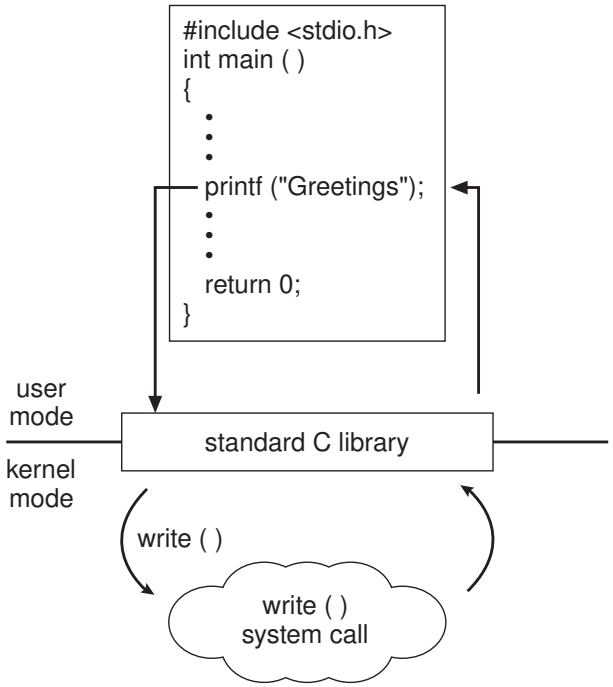
- System Calls (or simply *syscalls*) are functions offered by the OS, providing applications with access to the OS services
- Example: **strace cp x.c y.c** : shows the syscalls invoked by cp

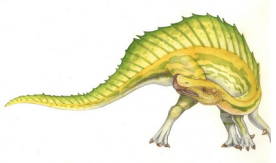
```
execve("/bin/cp", ["cp", "x.c", "y.c"], [/* 38 vars */]) = 0
...
stat64("y.c", {st_mode=S_IFREG|0644, st_size=41, ...}) = 0
stat64("x.c", {st_mode=S_IFREG|0644, st_size=53, ...}) = 0
stat64("y.c", {st_mode=S_IFREG|0644, st_size=41, ...}) = 0
open("x.c", O_RDONLY|O_LARGEFILE)       = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=53, ...}) = 0
open("y.c", O_WRONLY|O_TRUNC|O_LARGEFILE) = 4
fstat64(4, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
read(3, "#include <stdio.h> main() { prin"..., 32768) = 53
write(4, "#include <stdio.h> main() { prin"..., 53) = 53
read(3, "", 32768)                          = 0
close(4)                                    = 0
close(3)                                    = 0
close(0)                                    = 0
close(1)                                    = 0
close(2)                                    = 0
```



System Calls

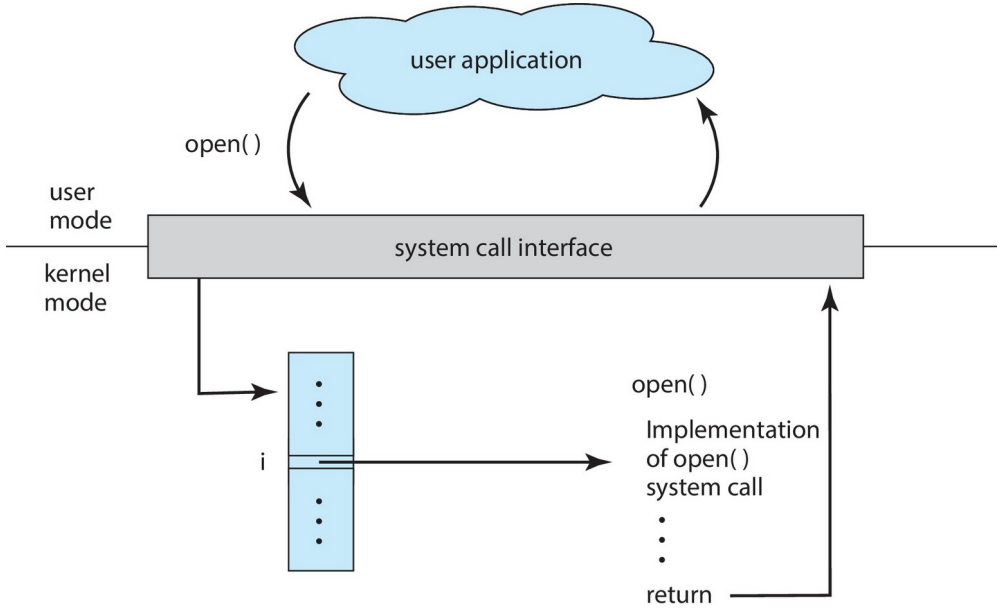
- Syscalls are mostly accessed by programs via a High-Level (HL) Application Programming Interface (API), rather than direct called
- Common HL APIs: Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Example: C program invoking printf() library call, which calls write() syscall





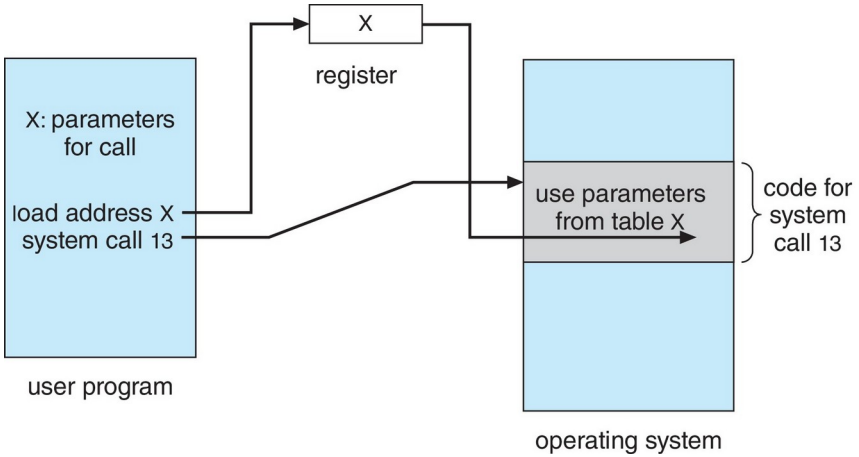
System Calls: Interface

- Each system call has a unique number (integer) associated
- The **system-call interface**
  - keeps a table indexed by these numbers
  - invokes the intended syscall in the OS kernel
  - returns status of the syscall and any return values
- The caller knows nothing about how the syscall is implemented; just obeys to the API and understands what OS will do as a result



System Calls: Parameter Passing

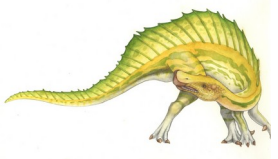
- (A) Pass the parameters in **registers**:
  - adequate for a small number of parameters
  - used in Linux for at most 5 parameters
- (B) Parameters stored in a **block**, or **table**, in **memory**, and **address** of block passed as a parameter in a register (Linux, for >5 parameters)



- (C) Parameters **pushed** onto the user program **stack**
- (B) and (C) do not limit the number or length of parameters

System Calls: Types

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File 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
- Information maintenance
  - 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 communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices



System Calls: Windows and Unix Examples

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Theoretical Unit 1

1.5 User-Level System Programs, System Services, and User Applications

User-Level System Programs, System Services, and User Applications

- Provide a convenient and productive user environment
- Some are simply user interfaces to system calls …
- … others are considerably more complex
- They can be divided into:
  - File manipulation
  - Status information (sometimes stored in a file)
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

User-Level System Programs

- **File management –** commands to create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

- **Status information**
  - Some commands ask the system for info - date, time, amount of available memory, disk space, number of users
  - Other commands provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems (like Windows) implement a **registry** - used to store and retrieve configuration information

User-Level System Programs

- **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
- **Program loading and execution** -
  - Absolute loaders, relocatable loaders, linkage editors
  - Debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and 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

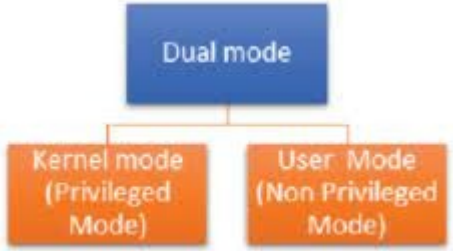
User-Level Services and Applications

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like file system health checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as user-level **services**, **subsystems**, **daemons**
- **Application programs**
  - Don't pertain to system; Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke
- Most users' view of the operation system is defined by system and application programs, not the actual system calls

Theoretical Unit 1

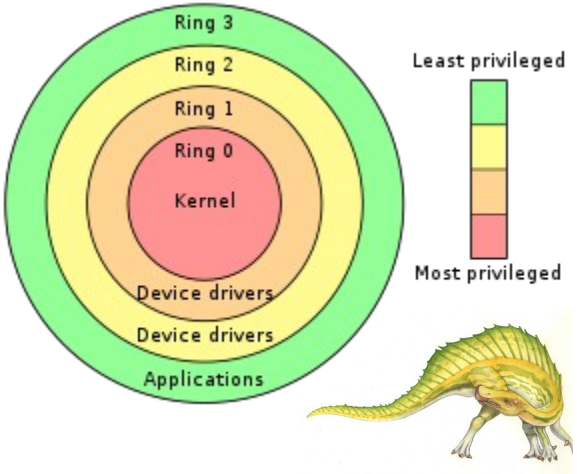
1.6 Hardware Protection

Dual-mode and Multimode Operation



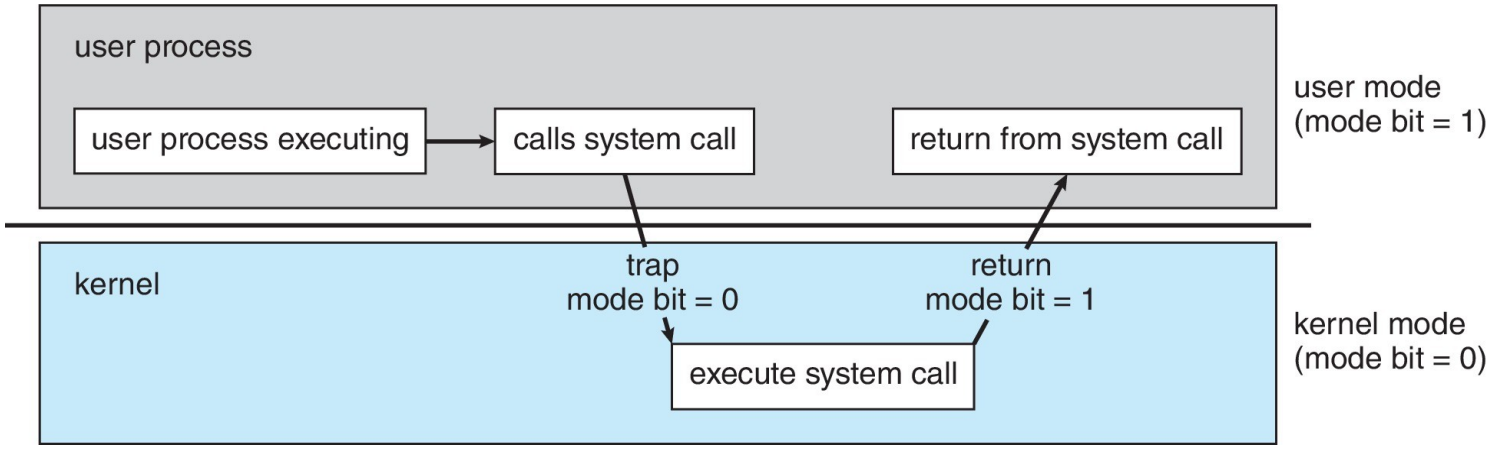
- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

- Increasingly CPUs support multi-mode operations
  - i.e. **virtual machine manager** (**VMM**) mode for guest **VMs**

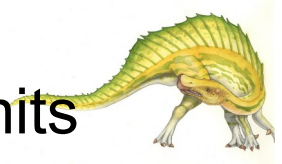


Requesting Privileged Services

- User-level processes ask the kernel for privileged services by means of system calls; these trigger a software interrupt and the user code is suspended while the kernel carries on the privileged service

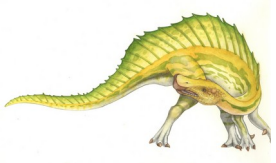


- Privileged operations: IO operations, changing the CPU operation mode bit, changing the interrupt vector, changing the timer, changing memory limits



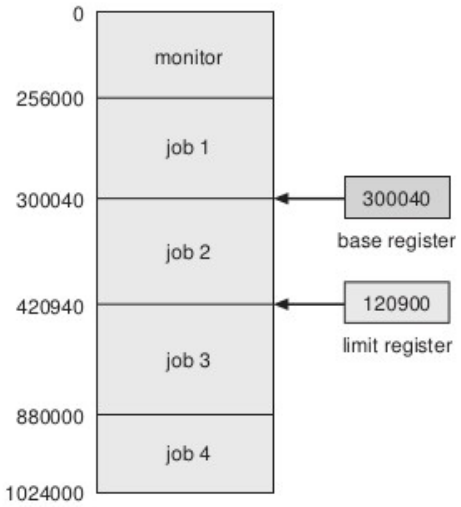
CPU Protection

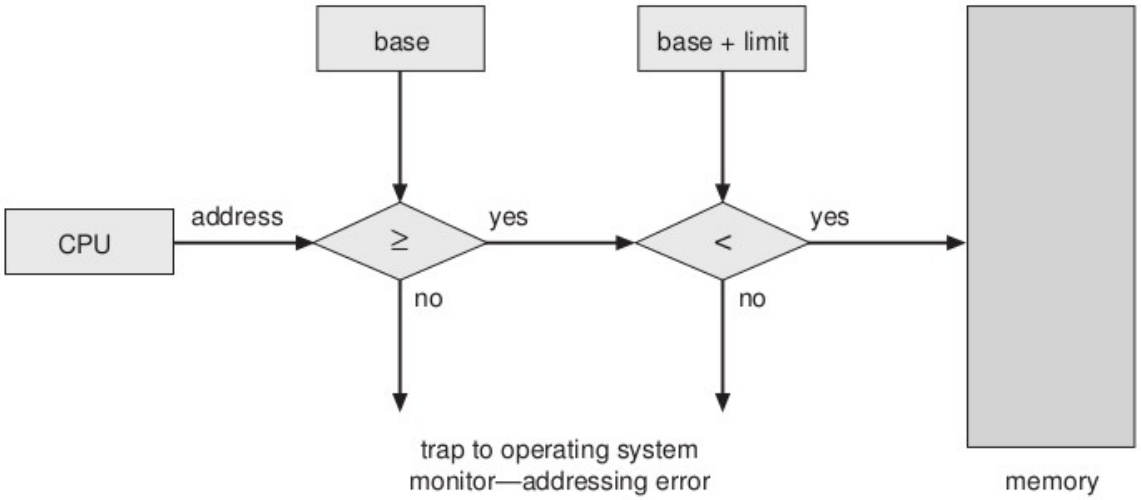
- Timer to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock
  - Operating system set the counter (privileged instruction)
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time



Memory Protection

- Goal: to prevent a process from accessing memory zones outside the ones it was assigned
- Mechanism: the memory limits of each process belong to its context (attributes) and are checked by the CPU Memory Management Unit for every access
- Privileged code accesses the entire memory freely





Theoretical Unit 1

1.7 Operating System Structure

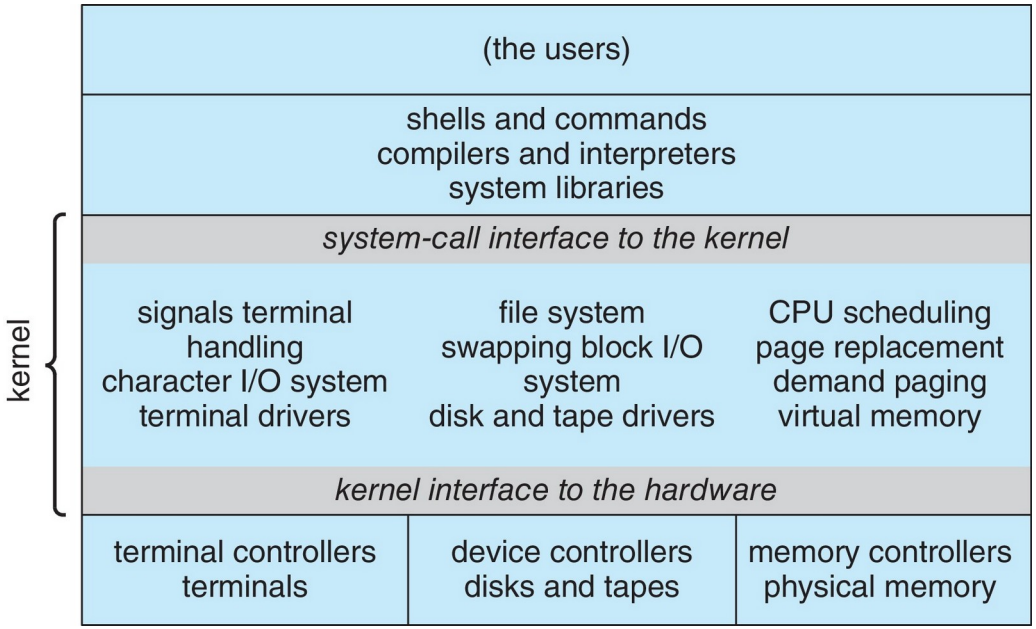
Operating System Structure

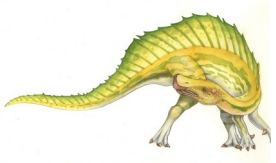
- General-purpose OS is a very large and complex software (SOs for specific purposes are smaller and simpler)

- There are various ways to structure the OS
  - Monolithic
  - Layered
  - Microkernel
  - Modular
  - Hybrid

Reference Monolithic Structure: Original UNIX

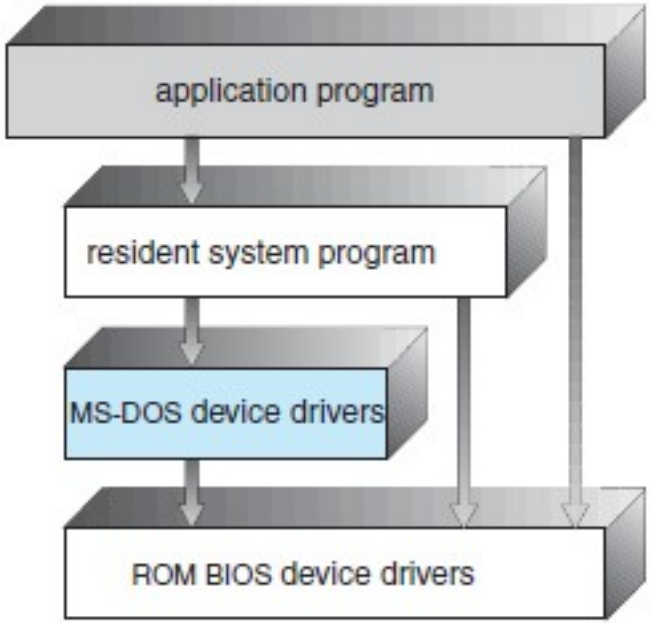
- Limited by hardware functionality, the original **UNIX** had limited structuring.
- The UNIX OS consists of two separable parts:
  - Systems programs + **The kernel**
- **The kernel**: everything below the syscall interface and above the physical HW; provides the file system, CPU scheduling, memory management, and other OS functions; a large number of functions for one single level

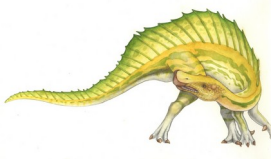




Simple Monolithic Structure: MS-DOS

- also limited by hardware functionality
  - no dual-mode on the 8088 CPU, RAM limited to 640KB
- tries to provide most functionality under very limited resources
- interfaces and functional layers not isolated; no clear separation

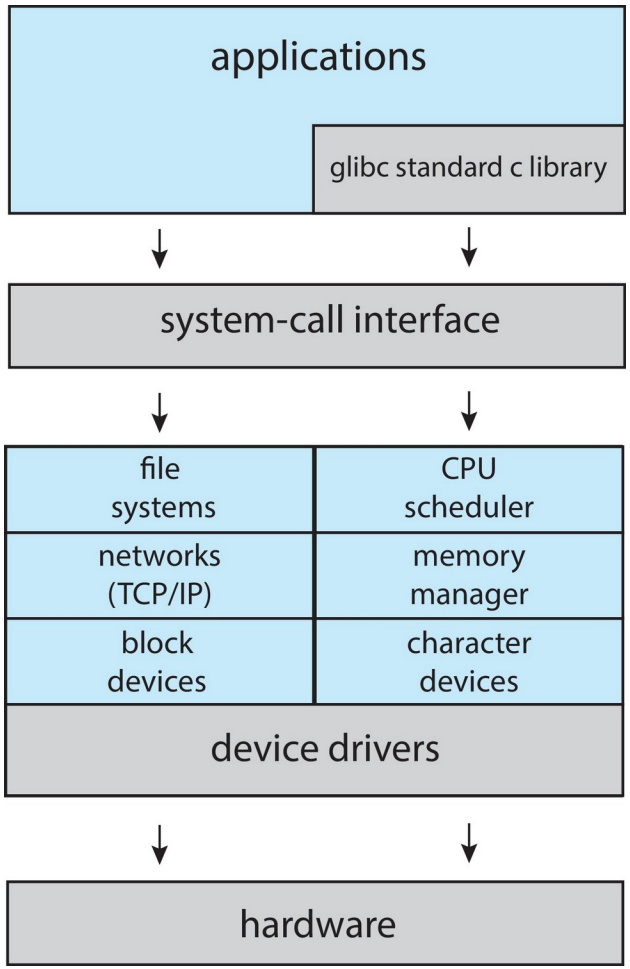


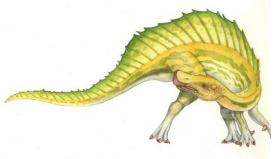


Modern Monolithic Structure: Linux

- Linux:
  - **monolithic** once it is a UNIX-like OS (kernel = 1 file = /boot/vmlinuz)
  - also **modular**, once allows kernel to be modified in run-time
  - apps may call *syscalls* directly, or via the glibc standard C library

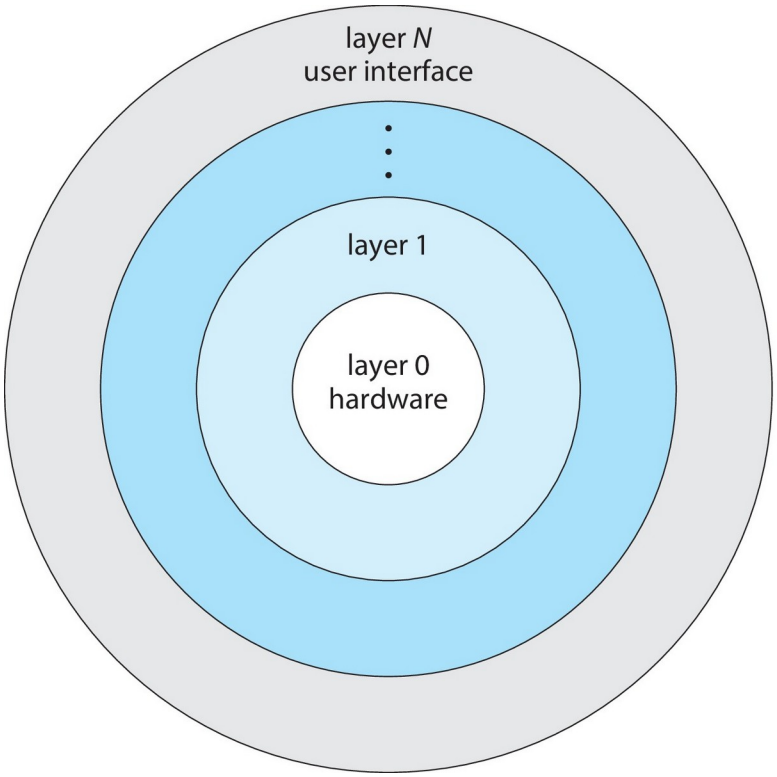
- Advantages / Disadvantages of the Monolithic Approach
  - apparent simplicity …
  - … but hard to implement and extend
  - good performance:
    - small overhead on the syscall interface and intra-kernel communications
  - still used on most modern OSes





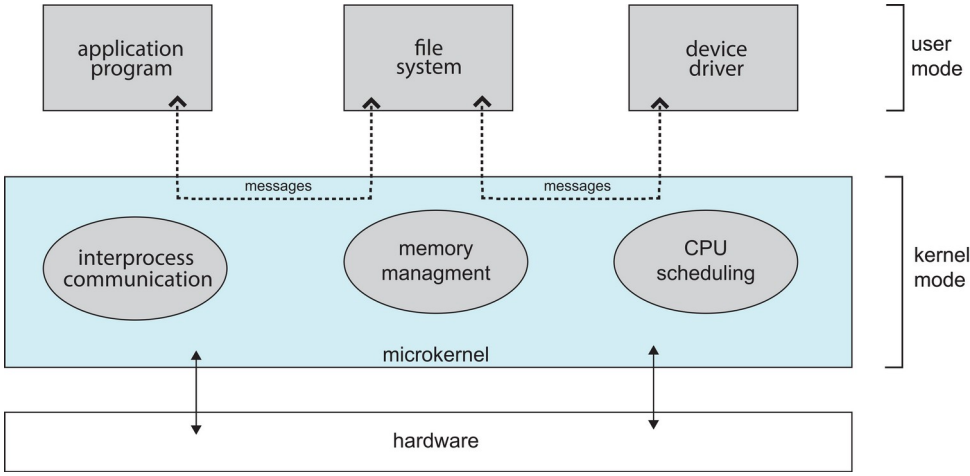
Layered Approach

- The operating system is divided into a number of **layers** (levels), each **built on top** of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- Layers are selected such that each uses functions (operations) and services of only lower-level layers.
- **Simple design**, simple to implement and maintain.
- May be difficult to define completely separated layers (dependencies)
- **More layers ==> lower performance** (more layers to traverse per request)

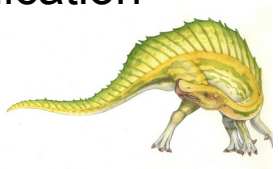


Microkernels

- Moves as much services as it can from kernel into user land
- Kernel preserves memory and CPU management, and IPC
- User modules communicate via kernel message passing

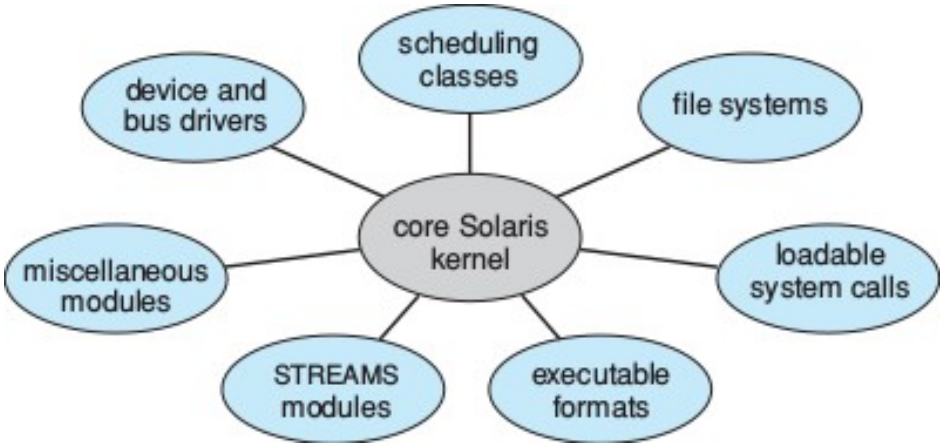


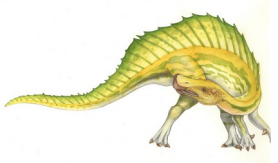
- Benefits:
  - Easier to extend a microkernel; new modules created in user space
  - Easier to port the operating system to new architectures
  - More reliable, more secure (less code is running in kernel mode => bugs are less probable – and their side-effects –; attack surface is narrower);
- Detriments:
  - Performance overhead of user space to kernel space communication
- **examples:** Mach, Tru64 UNIX, Windows NT 3.x, QNX (RTOS)



Modules

- Many modern OSs implement **loadable kernel modules** (**LKMs**)
  - Uses object-oriented approach
  - Each core component is separate
  - Each module talks to the others over known interfaces
  - Each module is (un)loadable as needed into (from) the kernel
- Similar to Layered Approach and to Microkernels but more flexible (modules may communicate directly between each other)





Hybrid Systems

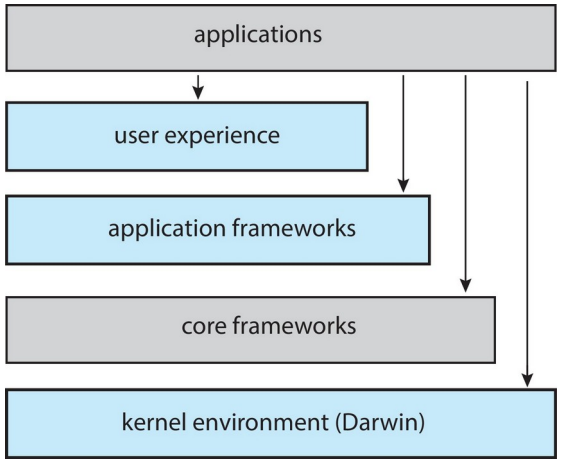
- Most modern operating systems are actually not one pure model
  - **Hybrid** combines multiple approaches to address performance, security, usability needs
  - **Linux** and Solaris are **monolithic**, plus **modular** for dynamic loading of functionality; Linux: see **lsmod**, **modinfo**, **insmod**, **rmmod** commands and /etc/modules, /etc/modprobe.d/* files
  - **Windows** mostly **monolithic**, plus **microkernel** for different subsystem *personalities*
  - **Apple macOS** has a layered architecture, where **Aqua** UI plus **Cocoa** programming environment sit on top of a kernel consisting of **Mach microkernel** and **BSD Unix parts**, plus I/O kit and dynamically **loadable** modules (**kernel extensions**)
  - **Apple iOS** follows a similar approach to macOS
  - **Google Android** has a stack similar to Apple iOS

Hybrid Systems – macOS and iOS

General Architecture

macOS: 

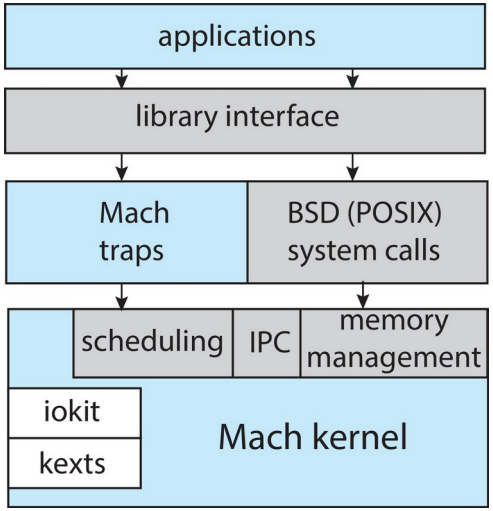
iOS: 



- **user experience**: mac OS – *Aqua* user interface (for mouse and keyboard); iOS – *Springboard* interface for touch sensitive screens
- **application frameworks**: includes Cocoa (macOS) and Cocoa Touch (iOS), with an API to develop GUI applications in Objective-C or Swift
- **core frameworks**: for graphics (OpenGL), multimedia (Quicktime), ...
- **Darwin kernel**: hybrid – Mach microkernel, plus a BSD UNIX kernel
  - Darwin source-code is open, contrarily to the upper layers code
- user **applications** may access directly any of these layers

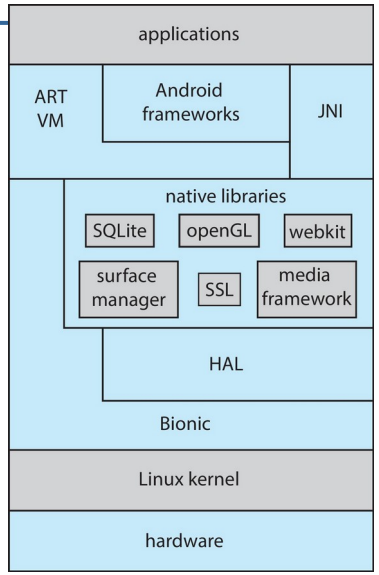
Hybrid Systems – macOS and iOS

- **Darwin kernel:**

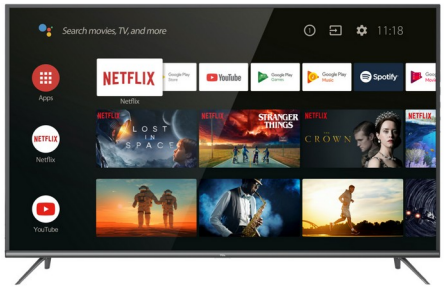


  - **system call interface:** Mach primitives (**traps**) and BSD primitives
  - **Mach microkernel**
    - memory and process management, inter-process communication
    - supports **kernel abstractions**: for instance, a **BSD process** created by **`fork`** will be represented as a **Mach kernel task**
    - an **iokit** supports the development of *device drivers;* **supports** dynamically loadable modules (**kexts**: kernels extensions)
    - it is not a pure microkernel: the various subsystems share the same address space, to improve message passing performance

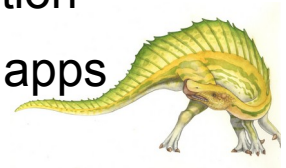
Hybrid Systems – Android



- **Android Architecture:**

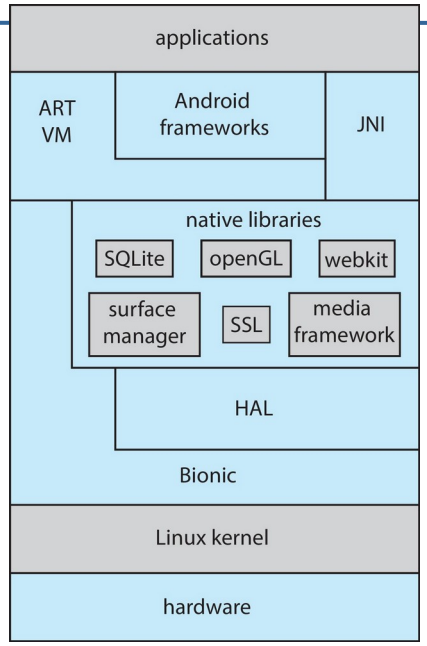


- **open-source**, developed by Google, targets **many different architectures**
- **layered** architecture; several frameworks and libraries above the kernel
- **hybrid**: similarly to iOS, frameworks/libraries seen as part of the OS
- applications developed in **Java**
  - Android API for Java: apps compiled to run in an **ART VM** (Android RunTime virtual machine), optimized for a small RAM size and modest CPUs; apps compiled to native machine code during installation
  - Java Native Interface (**JNI**): direct HW access; non-portable apps



Hybrid Systems – Android

Android Architecture:



- **native-libraries:** DBs (SQLite), browsing (webkit), multimedia, …
- **Hardware Abstraction Layer (HAL)**: makes easier to run Android on different HW; offers a consistent view of the various HW devices (sensors, GPS, cameras, ...); allows developing highly-portable applications
- **Bionic**: like the **`glibc`** (GNU C library), but lighter (less RAM and CPU)
- **Linux kernel**: modified for mobile devices (power management, etc.)

Theoretical Unit 1

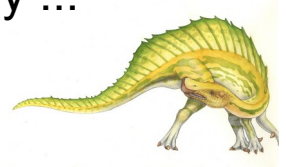
1.8 Operating System Development

Operating System Implementation

- early OSes fully programmed in assembly language;
- then system programming languages used, like Algol, PL/1
- modern OSes mainly coded in C

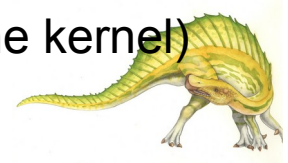
- OSes are usually coded in a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts

- high-level languages pluses and minuses
  - faster development, increased portability; code easier to understand, debug and tune; benefits from compiler evolution
  - generated code is typically slower than hand-written assembly ...



Operating System Generation

- Commonly, operating system already installed on purchased computer
  - Kernel, system programs and applications already compiled, with a default optimization level and broad peripheral support
- Some times, it is necessary to recompile the OS kernel
  - test experimental drivers, turn on/off features (tuning)
  - only possible if the full source code is available (same for the apps)
  - compiling the Linux kernel is a stress test and may take a long time
- Also possible to assemble a set of system utilities and user apps which, together with the kernel, make a personalized computing environment
  - in the Linux world, this is known as a “Linux Distribution”
  - example: Linux From Scratch to build a personalized distribution (this involves compiling all utilities and applications, not just the kernel)



Free and Open-Source Operating Systems

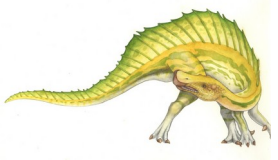
- Operating systems made available in source-code format rather than just binary **closed-source** and **proprietary**
- Started by **Free Software Foundation (FSF)**, which has "copyleft" **GNU Public License (GPL)**
- But **free software** and **open-source** are two different ideas
  - https://moqod-software.medium.com/understanding-open-source-and-free-software-licensing-c0fa600106c9
- Examples include operating systems like **GNU/Linux** and **BSD UNIX** (including core of **Mac OS X**), and many more
- Besides OSs, these ideas also apply to normal applications
- If software is free and/or source code is open, how do software companies earn money ? Support contracts, extra docs (KB), earlier or exclusive updates, paid courses, certification exams ...
- Some companies abuse / do not respect the GPL license terms

Theoretical Unit 1

1.9 Operating System Debugging and Monitoring

Operating System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
  - It is also **performance tuning** (performance problems may be viewed as errors, meaning tuning is a form a debugging)
- When a process fails
  - event may be registered in a **log file** containing error information (usual for system processes, not usual for user-level processes)
  - failure generate **core dump** file capturing memory of the process (file may be later used for a postmortem analysis by a debugger)
- Operating system failure generates **crash dump** file with kernel memory
  - risky to save this file in a normal file system (may be corrupted)
  - file saved in a separate raw partition and recovered during boot



Operating System Debugging

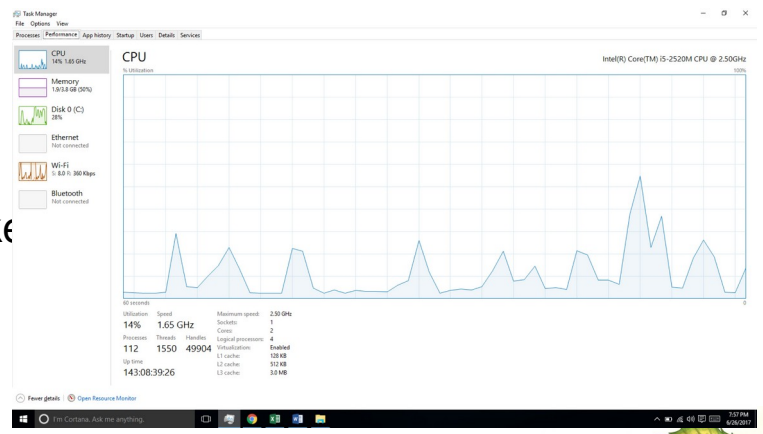
- Beyond crashes, performance tuning can optimize system performance
  - This is done by removing **bottlenecks**
  - Bottlenecks may be identified by **monitoring** and **profiling**
- Monitoring/Profiling tools
  - may be per **per-process** or **system-wide**
  - gather information through kernel-level **counters** or **tracing** (collects data for a specific event, such as steps involved in a system call invocation)
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

Operating System Debugging

- **Linux** tools that explore **counters**
  - per-process / selection of processes: **ps**, **top**, **htop**
  - system-wide: **vmstat** (memory), **netstat** (network), **iostat** (HD/SSD)
  - many of these tools query the /**proc** pseudo file-system
    - kernel exposes per-process and system metrics in /proc
    - cat /**proc**/**uptime**: 1st value - total number of seconds the system has been up; 2nd value - sum of how much time each core has spent idle, in seconds;
    - cat /**proc**/**12345**/**statm**: the status of the 12345 process memory (total program size (KB), … , number of dirty pages)

- **Windows** tool that explores **counters**:
  Task Manager (taskmgr.exe, Ctrl+SHIFT+Esc)

- **Linux** tools that exploit **tracing**
  - per-process: **strace** - trace system calls invoke by a process; **gdb** - source-level debugger
  - system-wide: **perf** - set of Linux performance tools; **tcpdump** - collects network packets



Operating System Debugging

- Debugging interactions between user-level and kernel code nearly impossible without tool-set that understands both
  - performance impact should also be minimal
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux that fulfills these conditions
  - BCC is a front-end to eBPF (extended Berkeley Packet Filter)
  - BPF initially (1990s) developed as a network filter
  - eBPF extended BPF to capture kernel events of many types
  - eBPF probes written in a subset of C, compiled to “eBPF instructions” that may be dynamically inserted into a running kernel
  - pure C-based eBPF probes are complex; BCC allows development in Python
    - a BCC tool written in Python includes embedded C code that interfaces with the eBPF instrumentation that interacts with the kernel
    - C code originates eBPF instructions that are inserted in the kernel using **probes** or **tracing points**

Operating System Debugging

- **`disksnoop.py`**: traces system-wide 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
```

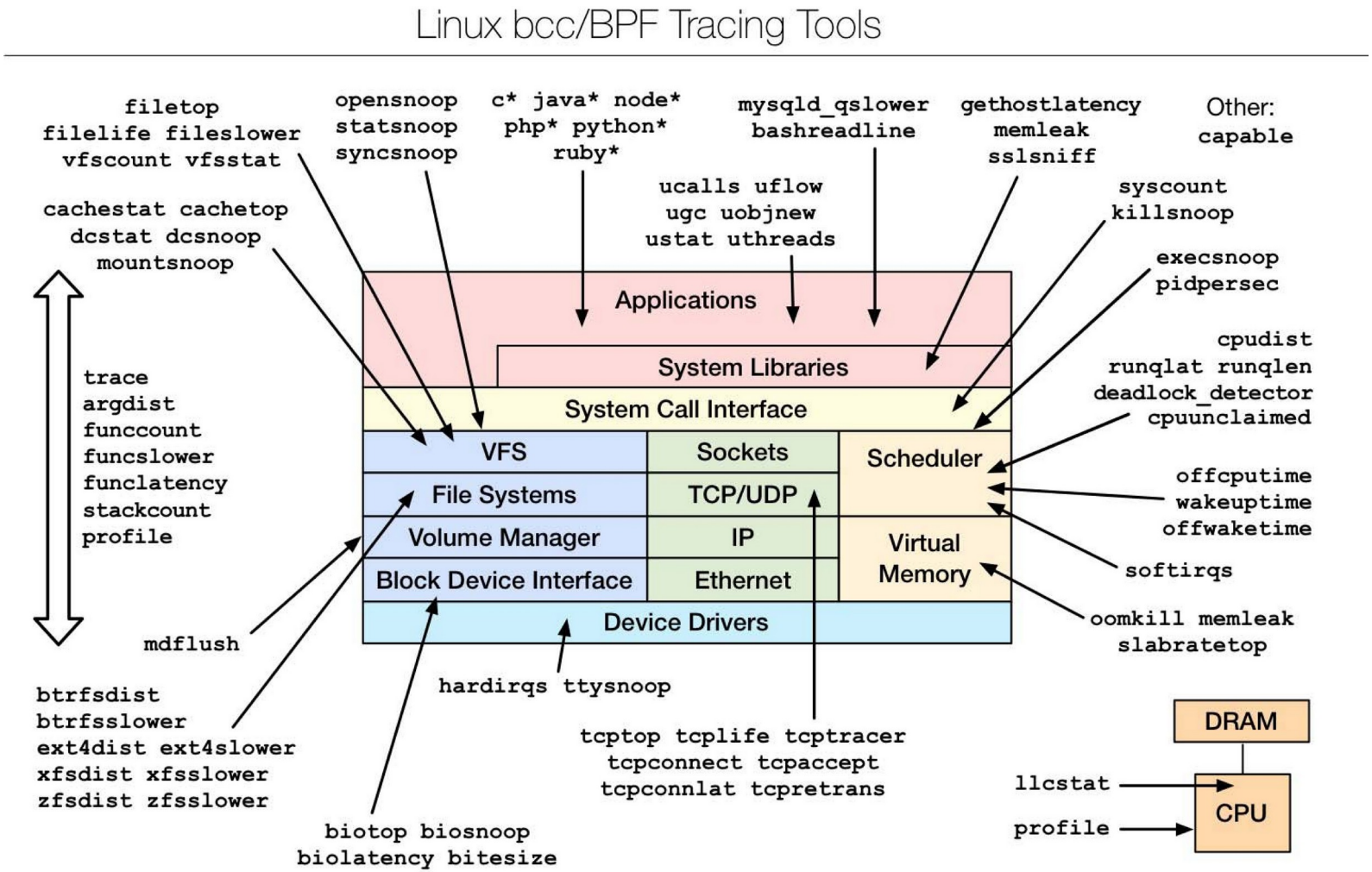
  - TIME(s): operation timestamp; T: operation type (R = Read, W = Write); BYTES: number of bytes of the operation; LAT(ms): operation latency (duration)

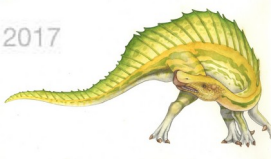
- **`opensnoop.py -Tp 1956`**: monitors calls to open system call

```
TIME(s)       PID    COMM               FD ERR PATH
0.000000000   1956   supervise           9   0 supervise/status.new
0.000289999   1956   supervise           9   0 supervise/status.new
1.023068000   1956   supervise           9   0 supervise/status.new
1.023381997   1956   supervise           9   0 supervise/status.new
```

  - TIME(s): operation timestamp; PID: process pid; COMM: process command; PATH: file open (in this example, the same file is open twice per second)

Operating System Debugging



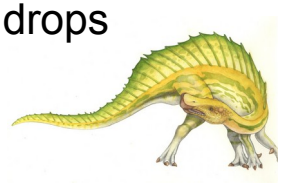


Theoretical Unit 1

1.10 Supplemental Slides
(note: check the portuguese version for more slides)

Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - **Random access**
  - Typically **volatile**
  - Typically **random-access memory** in the form of **Dynamic Random-access Memory (DRAM)**
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
- **Hard Disk Drives** (**HDD**) – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer
- **Non-volatile memory (NVM)** devices– faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular as capacity and performance increases, price drops



Storage Definitions and Notation Review

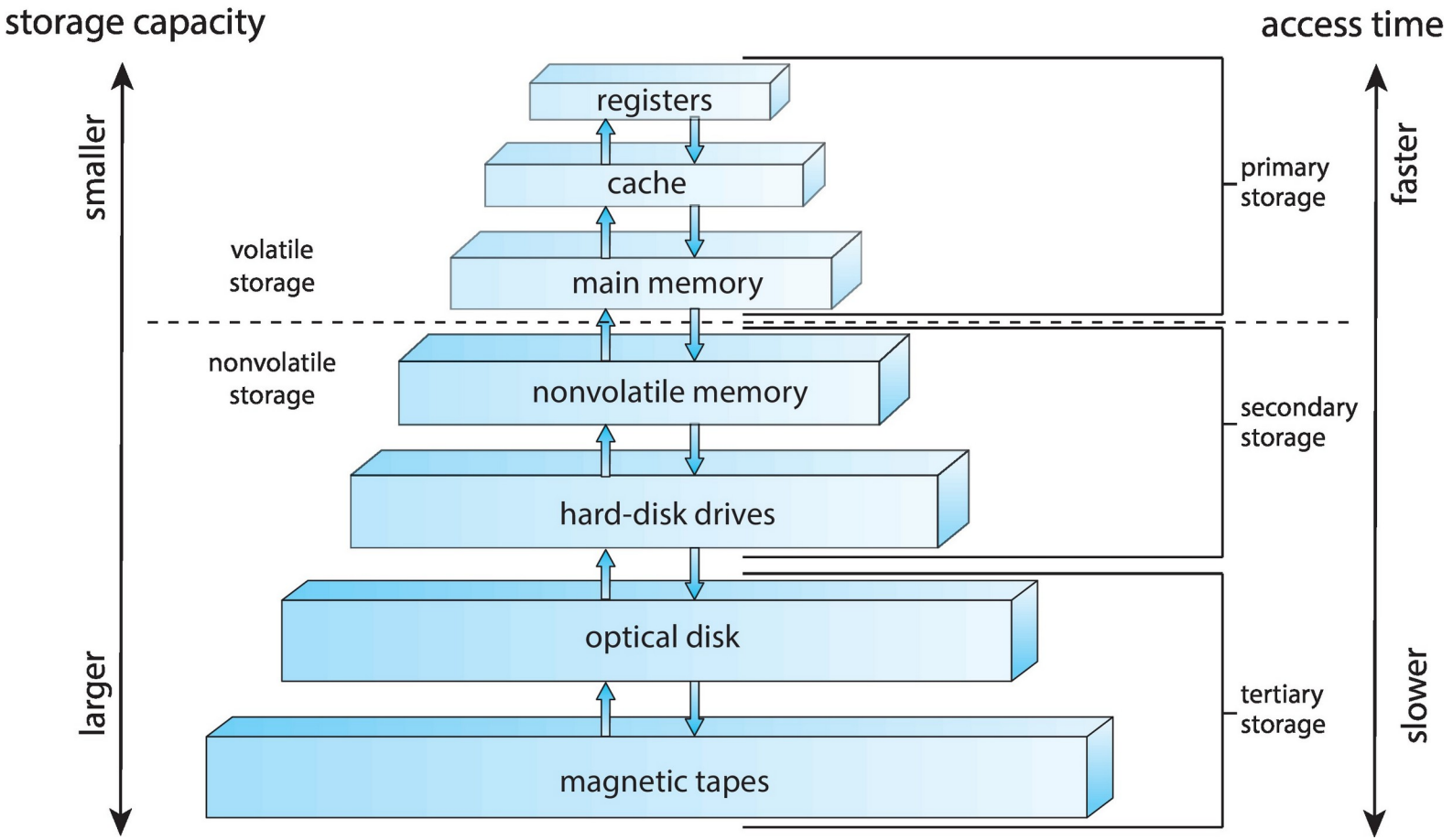
The basic unit of computer storage is the **bit** . A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

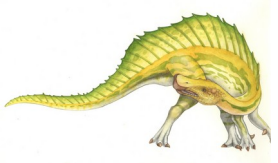
Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte** , or KB , is 1,024 bytes; a **megabyte** , or **MB** , is $1,024^2$ bytes; a **gigabyte** , or GB , is $1,024^3$ bytes; a **terabyte** , or **TB** , is $1,024^4$ bytes; and a **petabyte**, or **PB** , is $1,024^5$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

Storage-Device Hierarchy





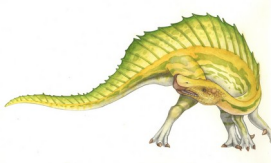
Characteristics of Various Types of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Movement between levels of storage hierarchy can be explicit or implicit

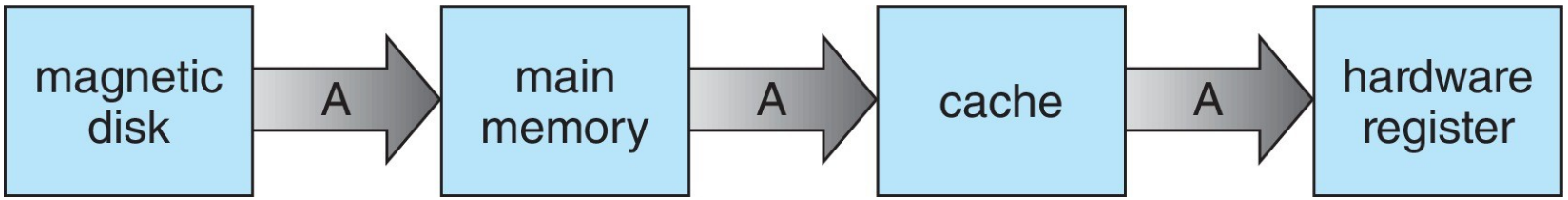
Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy



Migration of data “A” from Disk to Register

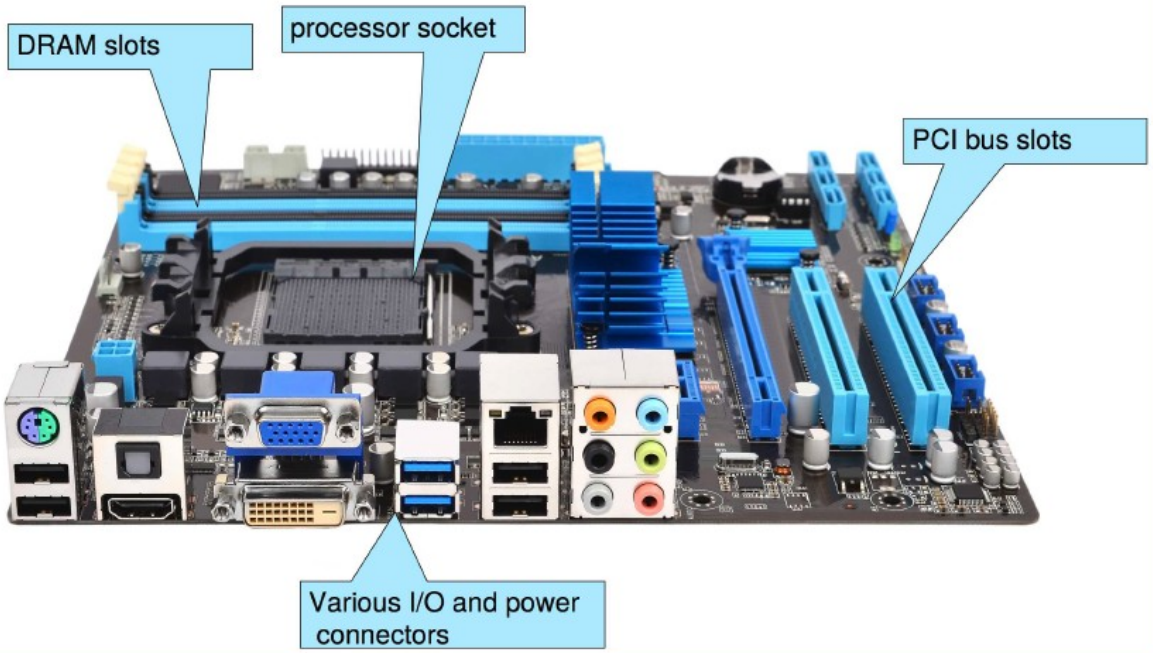
- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist
  - Various solutions covered in Chapter 19

Computer-System Architecture

- Most systems use one or more general-purpose processor, and several special-purpose (micro-)processors as well

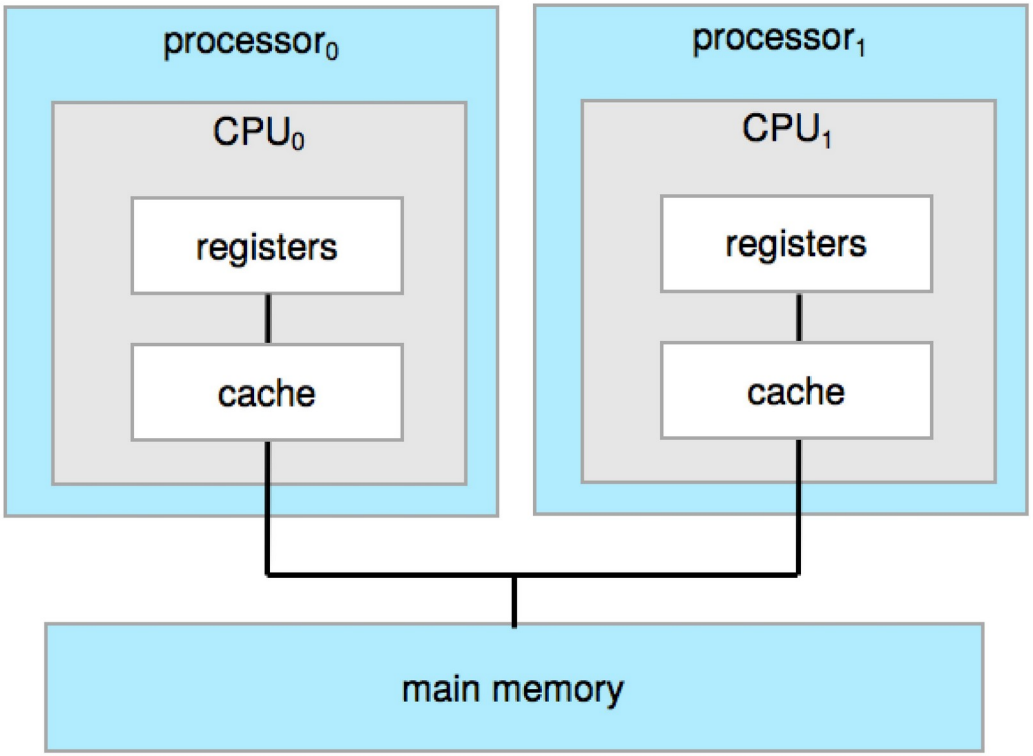


This board is a fully-functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

Computer-System Architecture

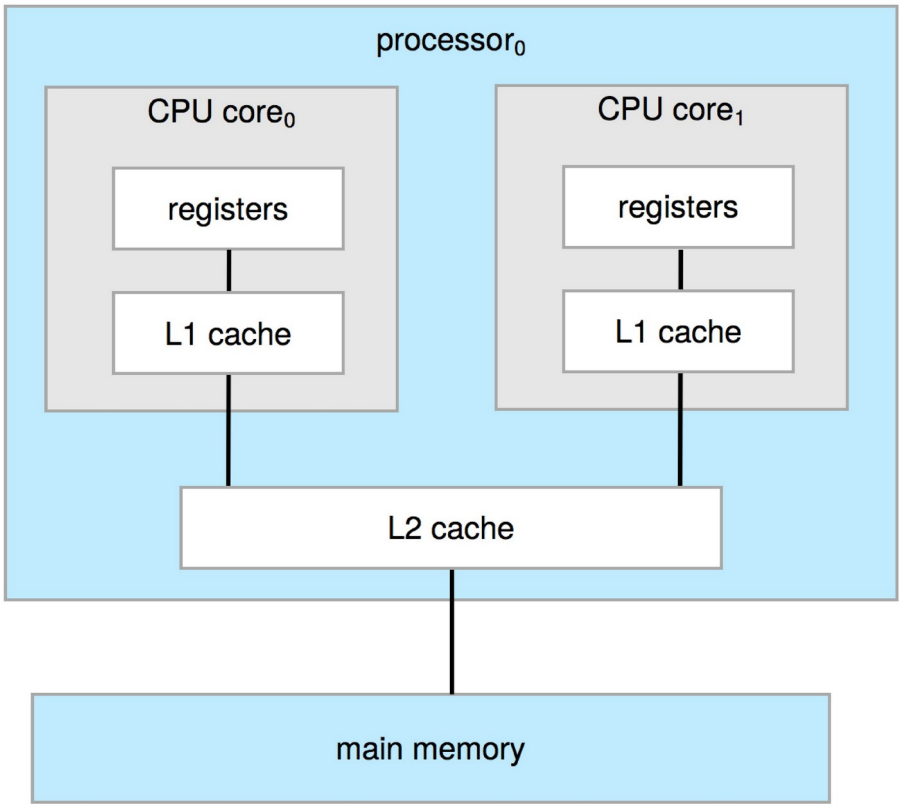
- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    1. **Increased throughput**
    2. **Economy of scale**
    3. **Increased reliability**
       - graceful degradation or fault tolerance
  - Two types:
    1. **Asymmetric Multiprocessing**
       - each processor is assigned a specie task
    2. **Symmetric Multiprocessing**
       - each processor performs all tasks

Symmetric Multiprocessing Architecture



A Dual-Core Design

- Multi-chip and **multicore**
- Systems containing all chips
  - Chassis containing multiple separate systems



Implementation of a System Call in Linux

USER-LAND code:

- test program (compiled with "gcc test.c -I<path to absolute.h folder> -o test.exe")

```
// test.c
#include <stdio.h>
#include <absolute.h>

main() {
int a;
scanf("%d", &a);
printf("%d\n", absolute(a));
}
```

- header file absolute.h

```
// src/linux-4.4.0/absolute/absolute.h
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

unsigned int absolute(int i) {
return syscall(XXX, i);
// XXX is the unique syscall integer id
}
```

Implementation of a System Call in Linux

KERNEL-LAND code:

- definition of the unique id of the primitive

```
// src/linux-4.4.0/arch/x86/entry/syscalls/syscall_64.tbl
...
326     common      absolute                    sys_absolute
...
```

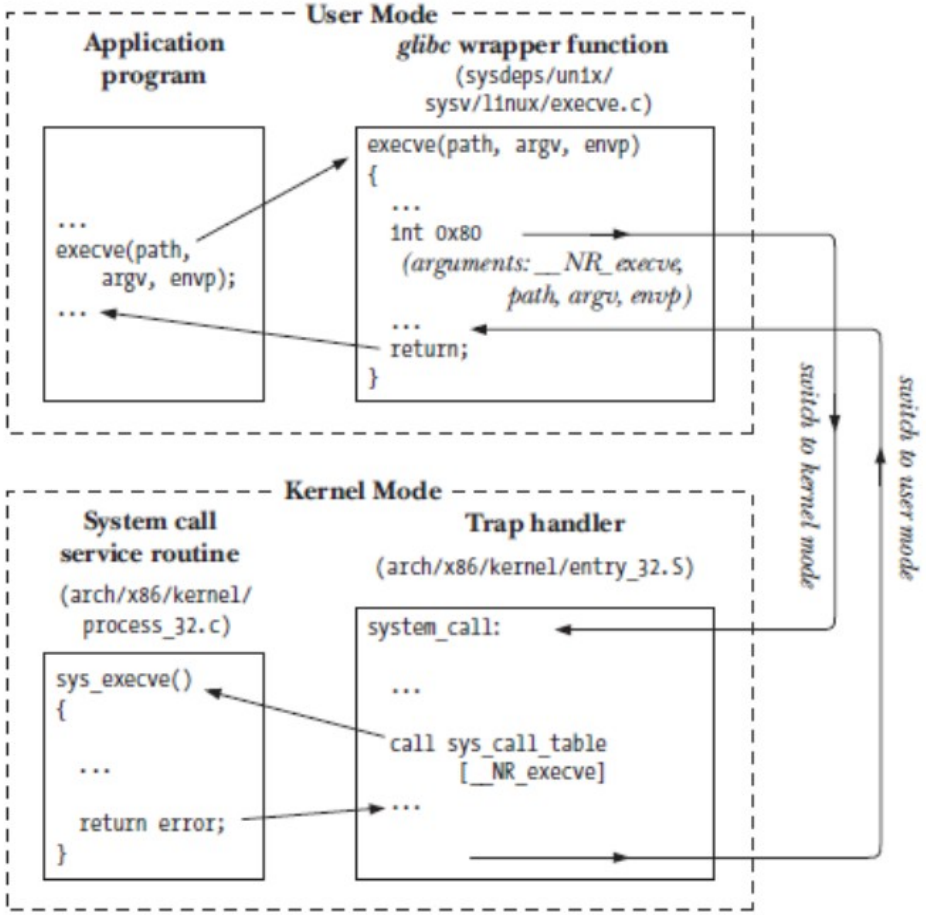
- declaration of the primitive

```
// src/linux-4.4.0/include/linux/syscalls.h
...
asmlinkage unsigned int sys_absolute(int);
...
```

- code of the primitive

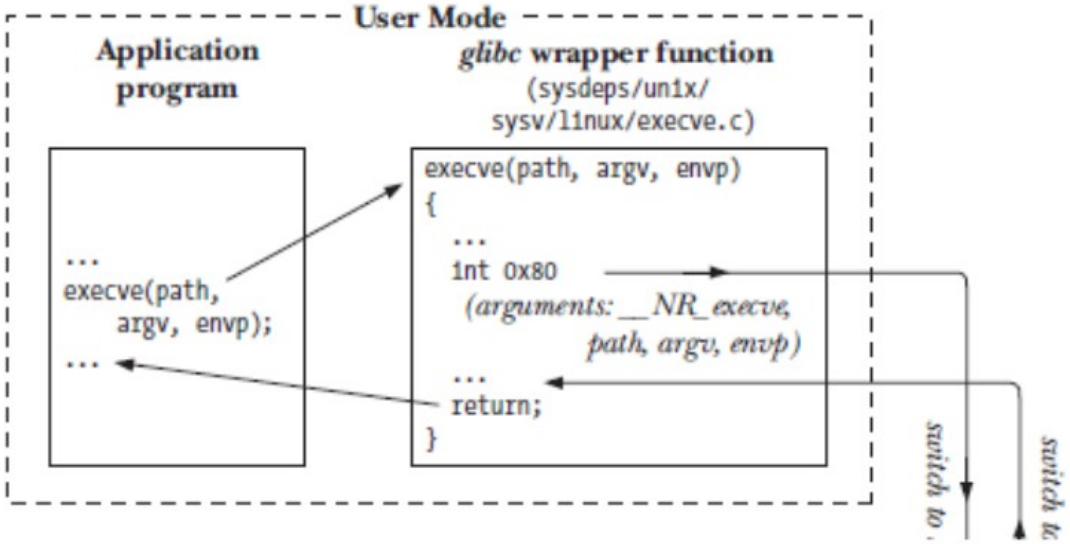
```
// src/linux-4.4.0/absolute/absolute.c
#include <linux/kernel.h>

asmlinkage unsigned int sys_absolute(int i) {
//printk(KERN_DEBUG, "sys_absolute: received %d\n", i);
return(i>=0 ? i:(-i));
}
```

Anatomy of a System Call in Linux



[Fonte: "The Linux Programming Interface - Section 3.1", Michael Kerrisk, No Starch, 2010]

Anatomy of a System Call in Linux



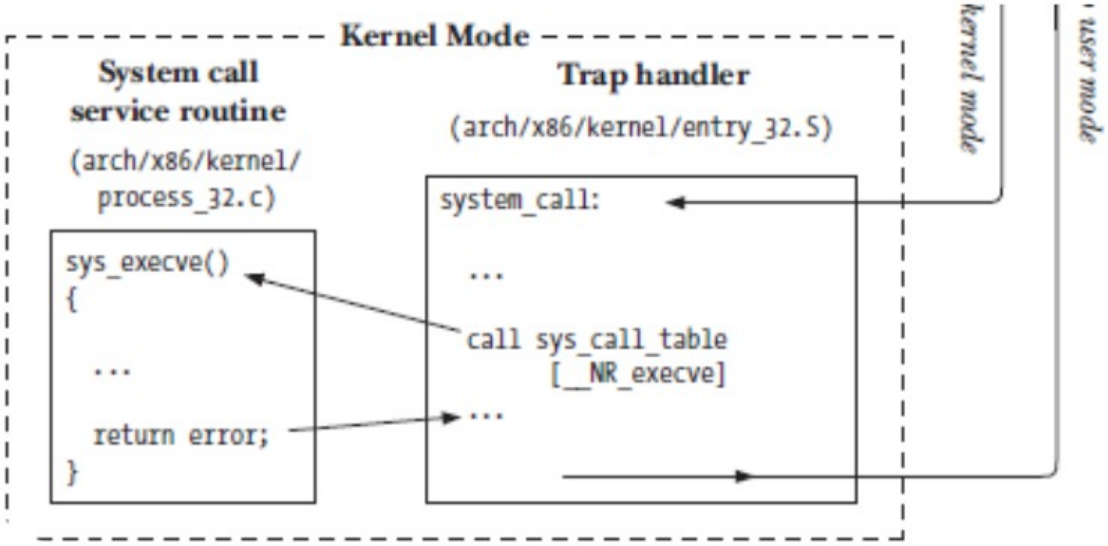
1- The application program makes a system call by invoking a wrapper function in the C library (e.g., execve in *glibc*).

2- The wrapper must make all of the system call arguments available to the system call trap-handling routine (which belongs to to the kernel – see bellow). The arguments are passed to the wrapper via the stack, but the kernel expects them in specific registers. The wrapper copies the arguments to these registers.

3- Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. To permit this, the wrapper function copies the system call number (*__NR_execve* = 11) into a specific CPU register (%eax).

4- The wrapper function executes a *trap* machine instruction (`int 0x80`), which causes the processor to switch from user mode to kernel mode and execute code pointed to by location 0x80 (128 decimal) of the system's trap vector.

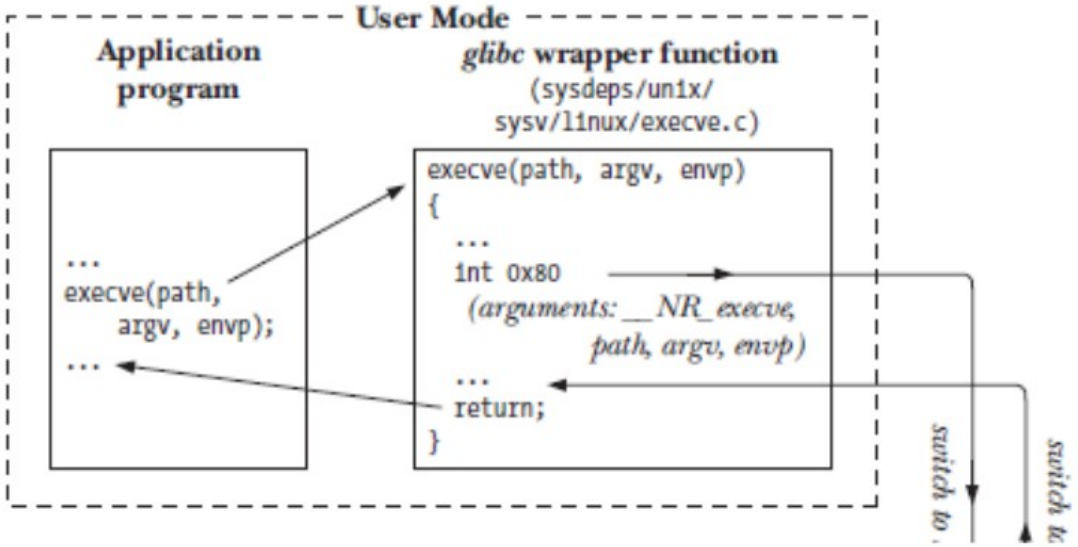
Anatomy of a System Call in Linux



5- In response to the trap to location 0x80, the kernel invokes its *system_call()* routine (located in the assembler file `arch/i386/entry.S`) to handle the trap. This handler:

a) Saves register values onto the kernel stack;
b) Checks the validity of the system call number (*__NR_execve*);
c) Invokes the appropriate system call service routine, which is found by using the system call number (*__NR_execve*) to index a table of all system call service routines (the kernel variable *sys_call_table*). If the system call service routine has any arguments, it first checks their validity. Then the service routine performs the required task. Finally, the service routine returns a result status to the *system_call()* routine.
d) Restores register values from the kernel stack and places the system call return value on the stack.
e) Returns to the wrapper function, simultaneously returning the processor to user mode.

Anatomy of a System Call in Linux



6- If the return value of the system call service routine is an error, the wrapper function sets the global variable *errno* using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

Theoretical Unit 1

References:

- "Operating System Concepts, 10th Ed.", Silberschatz \& Galvin, Addison-Wesley, 2018: Chapters 1 and 2

- Interrupt Vector: https://www.sciencedirect.com/topics/engineering/interrupt-vector

- Cpu Rings, Privilege and Protection: https://web.archive.org/web/20180713151101/https://manybutfinite.com/post/cpu-rings-privilege-and-protection/

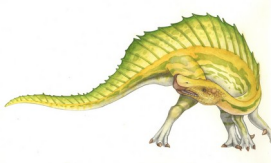
- System Calls Make the World Go Round: https://manybutfinite.com/post/system-calls/

- "The Linux Programming Interface", Michael Kerrisk, No Starch, 2010: Section 3.1

- How Computers Boot Up: https://manybutfinite.com/post/how-computers-boot-up/

- The Kernel Boot Process: https://web.archive.org/web/20180501083929/https://manybutfinite.com/post/kernel-boot-process/

- BCC Tracing Tools: https://github.com/iovisor/bcc



End of Theoretical Unit 1