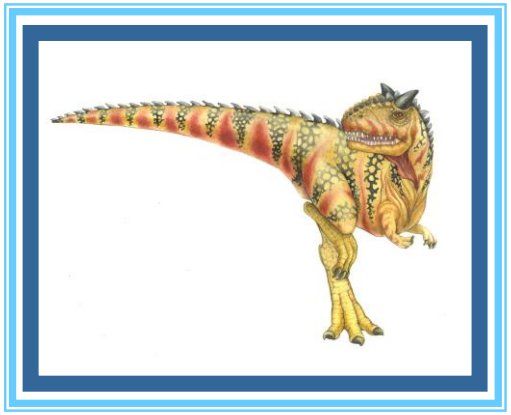
Chapter 2: Operating-System Services

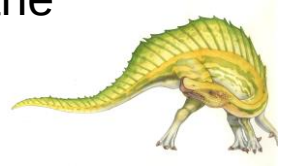


Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**), **touch-screen**, **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

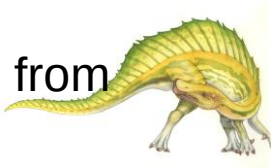
Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
    - **Shared memory:** Two or more processes read and write to a shared section of memory
    - **Message passing:** Packets of information in predefined format are moved between processes.
  - **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Sometimes it may need to halt the system, or terminate the error-causing process.

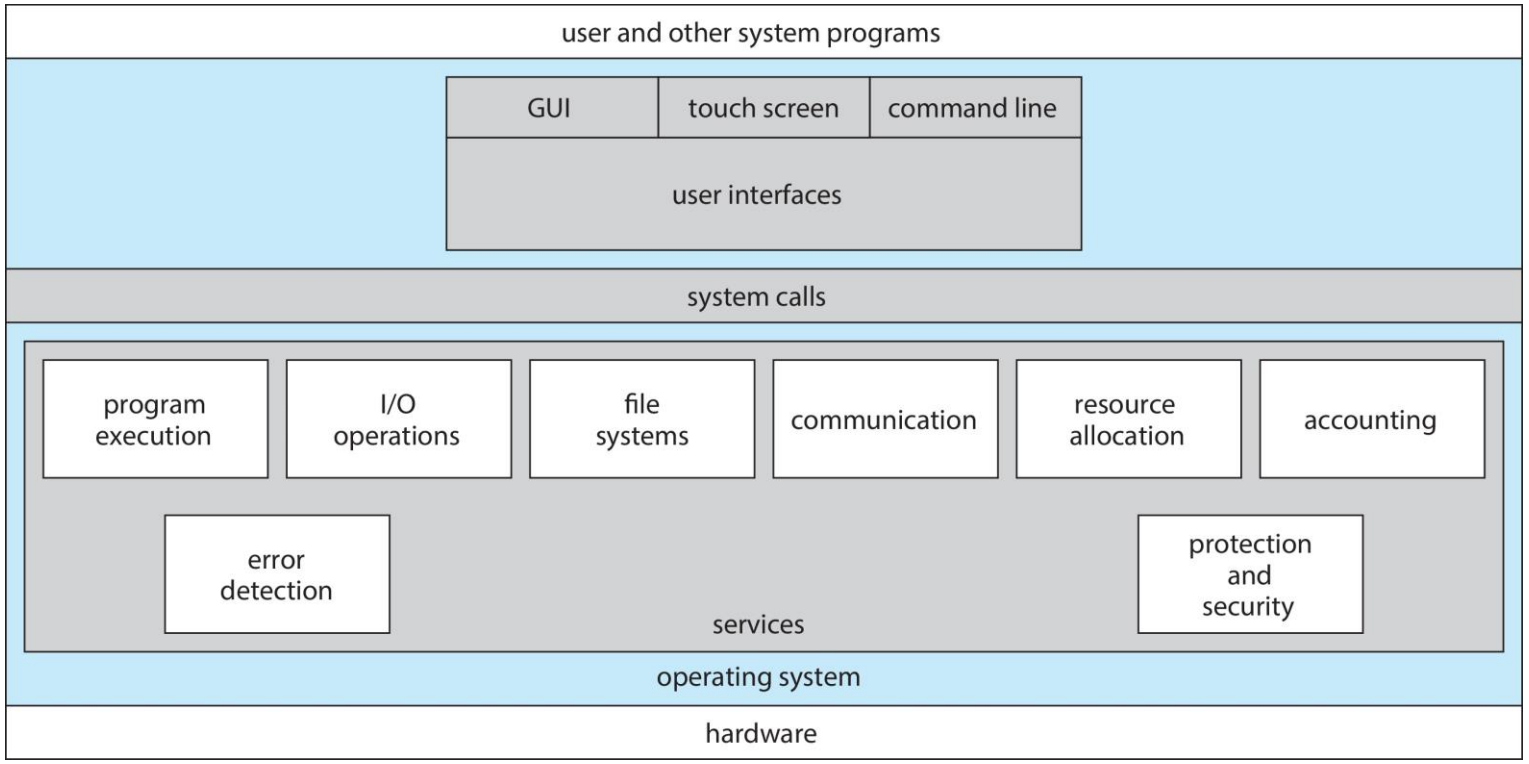


Operating System Services (Cont.)

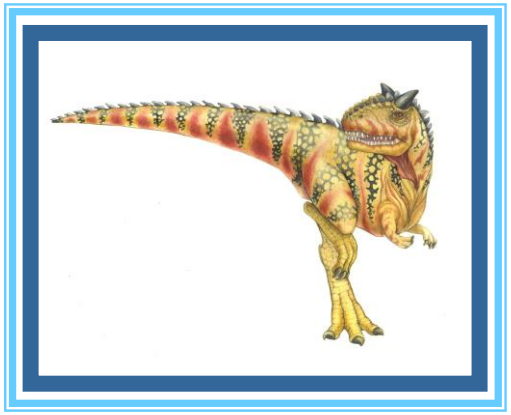
- Another set of OS functions exists for ensuring the efficient operation of the system itself via **resource sharing**
- These are not to help the user but to ensure efficient operation of the system.
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - *CPU cycles, main memory, file storage, I/O devices.*
      - i.e. OS have CPU-scheduling routines to make best use of CPU.
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, **concurrent processes should not interfere with each other**
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts



A View of Operating System Services



User and OS interfaces



Command Line interpreter

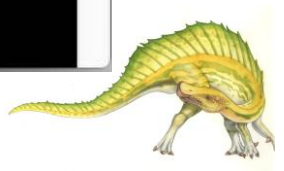
- CLI allows direct command entry
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Bash, C-Shell, Sh, Korn Shell, Zsh, etc.
  - Bash comes with Linux distros
  - Zsh comes with MacOS X

Bourne Shell Command Interpreter

1. root@r6181-d5-us01:~ (ssh)

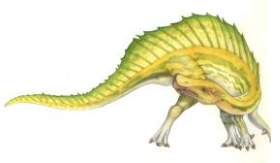
root@r6181-d5-u... ⌘1 | ssh ⌘2 | root@r6181-d5-us01... ⌘3

```
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                       50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1             477M   71M  381M  16% /boot
/dev/dssd0000         1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                       12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?   S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0      0     0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0      0     0 ?        S    Jul12 177:42 [vpthread-1-2]
root       3829  3.0  0.0      0     0 ?        S    Jun27 730:04 [rp_thread 7:0]
root       3826  3.0  0.0      0     0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x------ 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```



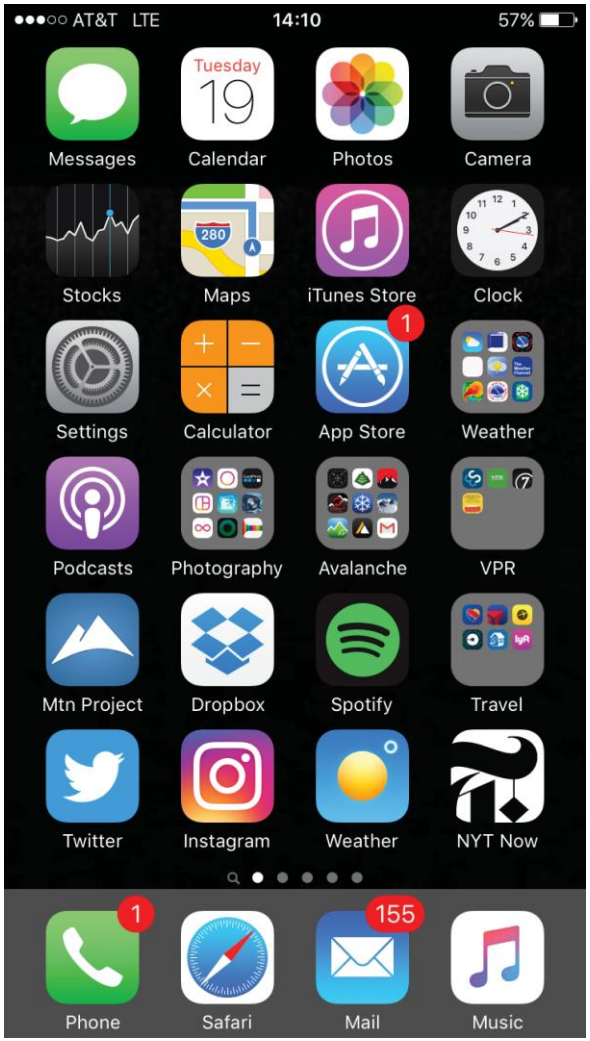
User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - 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 (CDE, KDE, GNOME)
    - Unix & Linux systems heavily use CLI.

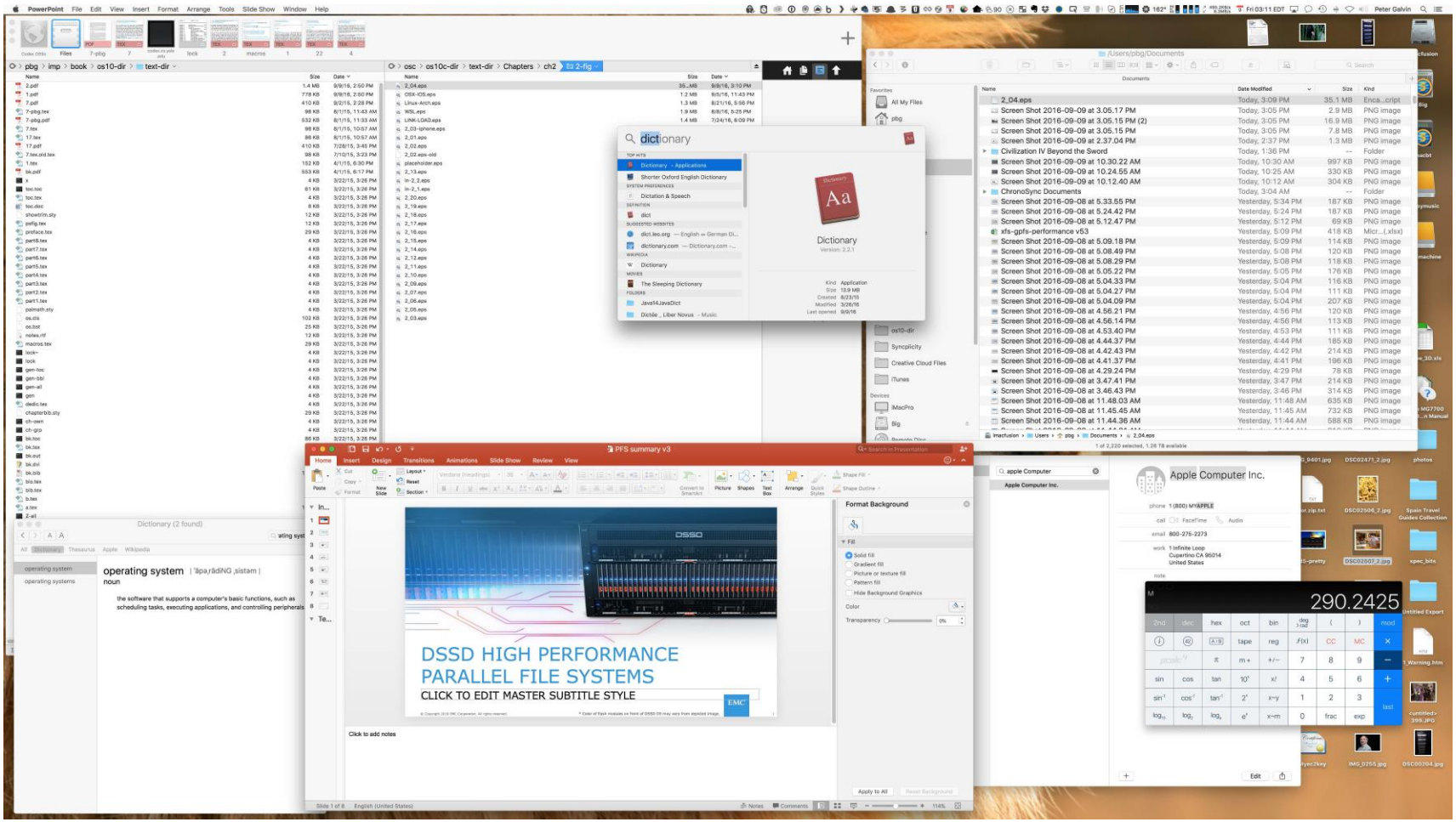


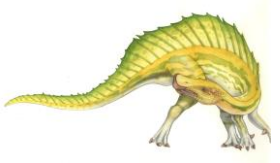
Touchscreen Interfaces

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



The Mac OS X GUI





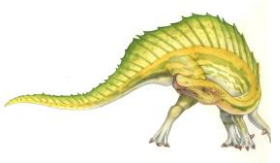
Choice of interface

- Personal preference
- Sysadmins and powerusers heavily use CLI
  - More efficient and faster Access
  - GUI can be limited
  - Can use Shell scripts to automate repetitive tasks
- Many users have no idea about shells and never use it
  - It is ok for them, but we need to know our way around it.

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
  - Why?
  - Program portability: Program using an API can be compiled and run on any system that supports the same API (in theory).
  - System calls can be more detailed and harder to work with.
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Note that the system-call names used throughout this text are generic

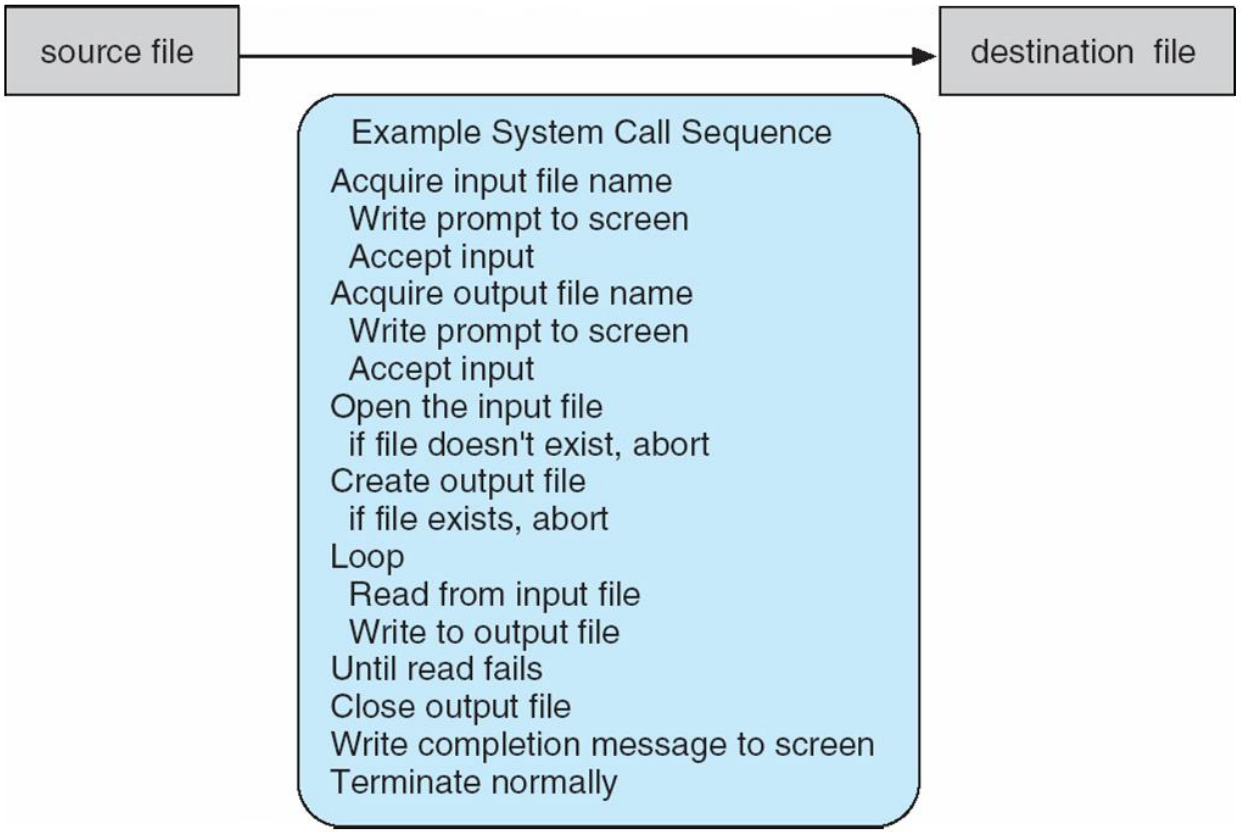


System calls

- For most programming languages, the **run-time support system** provides a **system-call interface**.
  - It provides the necessary infrastructure to execute a program in the target environment.
  - Takes care of many lower-level concerns that the program might encounter during its execution.
    - Hence, it's called runtime.
- JRE: Java Runtime Environment
  - Includes JVM, core libraries and other stuff to run Java apps.
- CLR: Common Language Runtime
  - Runtime for .NET apps, provides memory management, security, etc.

Example of System Calls

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



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

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

return value — function name — parameters

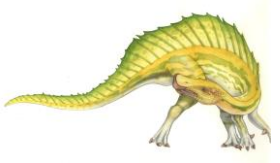
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

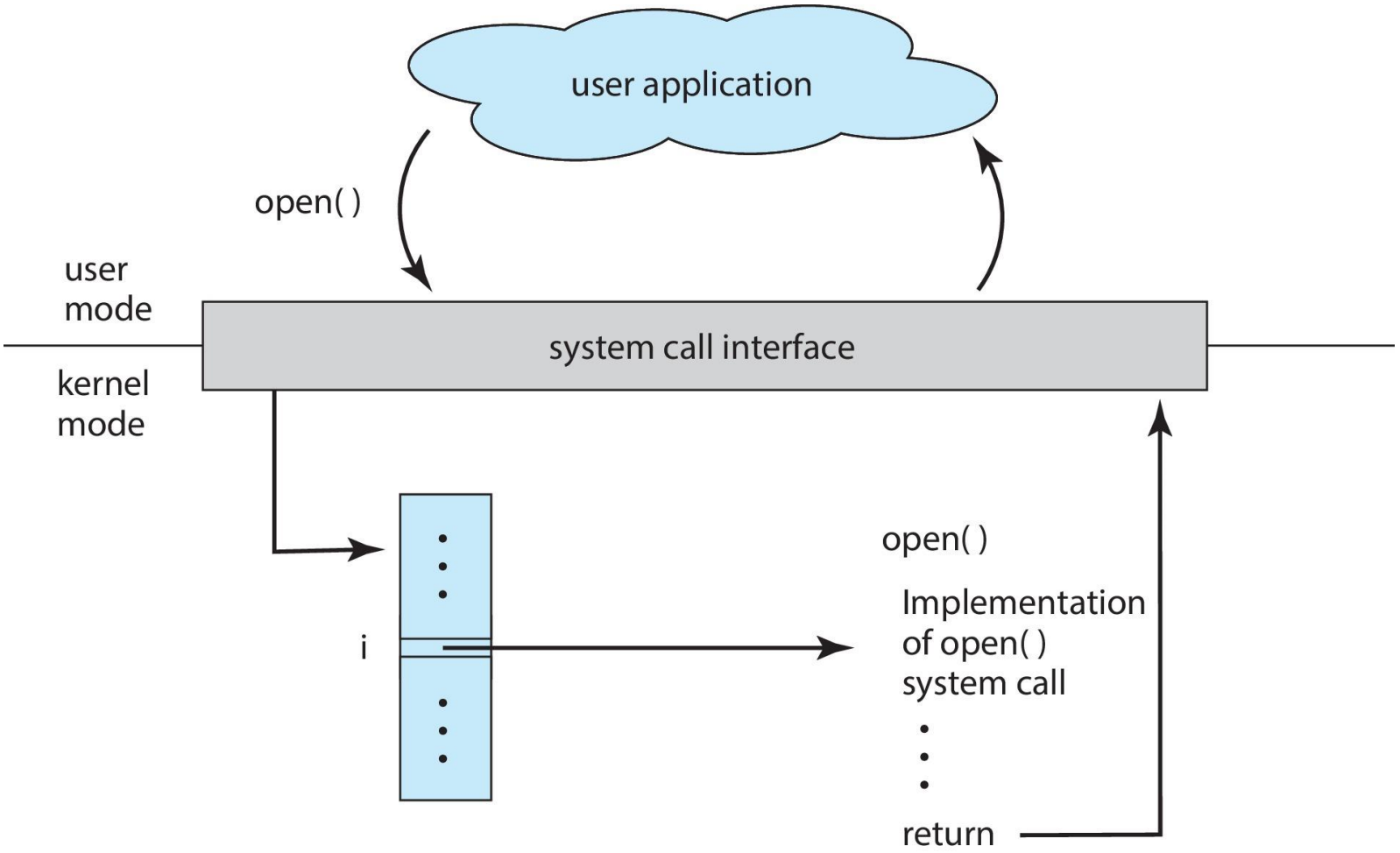
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Call Implementation

- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)



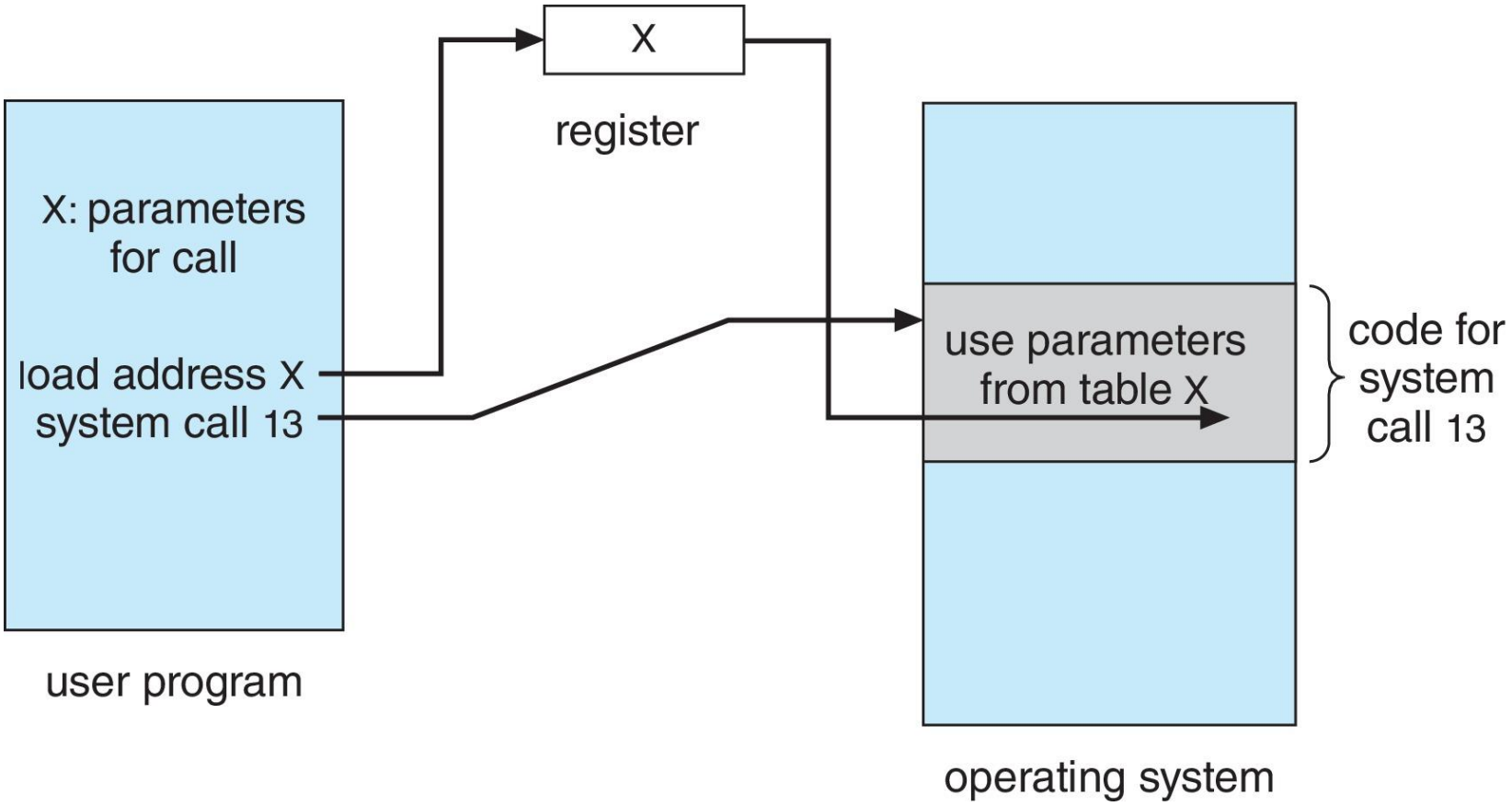
API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - **Simplest**: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Some OS prefer block and stack methods.
    - Because they do not limit the number or length of parameters being passed.

Parameter Passing via Table



Types of System Calls

- Process control
  - create process, terminate process
  - **end, abort**
  - **Dump memory if error**
  - **Debugger** for determining **bugs, single step** execution
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - **Locks** for managing access to shared data between processes

- If a running program is halted normally (end) or abnormally (abort); the system takes a dump of memory and show an error message.
- This dump is written to disk and may be examined by debugger.
  - A system program designed to help the programmer to find and correct errors, bugs.

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **load, execute**
  - **get process attributes, set process attributes**
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - **Locks** for managing access to shared data between processes

- A process or job executing one program may want to load or execute another program.
  - What happens when the existing program?
  - What will happen when the new program halts? Where will it return?
- We want it to return to the same place, so we need to save the memory image of the previous (parent) program.

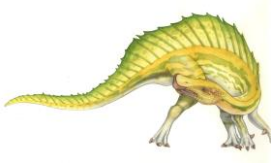
- When we create a new job or process, we should be able to control its execution.
  - This control requires the ability to determine and reset the attributes of the:
  - Job's priority,
  - Maximum allowable execution time, etc.

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - load, execute
  - get process attributes, set process attributes
  - **wait for time**
  - **wait event, signal event**
  - allocate and free memory
  - Locks for managing access to shared data between processes

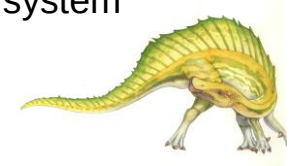
- Having created new jobs or processes, we may need to wait for them to finish their execution.
- We may want to wait for a certain amount of time:
  - Wait_event()
- Or we may want to wait for an event:
  - Signal_event()

- Often, two or more processes share data. To ensure the integrity of the shared data, OS often provide system calls allowing a process to **lock** shared data.
  - Acquite_lock(), release_lock()



Types of System Calls (Cont.)

- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes *(name, type, etc.)*
  - Some OS may have system calls like move(), copy(); or use an API that performs these operations using code or other system calls.
- **Device management**
  - request device, release device *(main memory, disk device, Access to files, etc)*
    - *After finished, we release()*
      - *Similar to open() and close()*
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- File and device management are similar, that's why some OS (e.g. Unix) merges them: **file-device structure.**
- It is also possible that the UI make them seem similar, even though the underlying system calls are different.

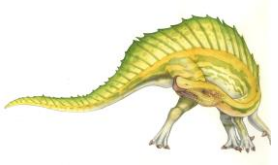


Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Many system calls exist for the purpose of transferring information between the user program and operating system.
  - Time(), date()
  - Return info about the system, how many users, version number, amount of free memory, etc.
- OS keeps information about the system and uses system calls to access these information.

Types of System Calls (Cont.)

- Communications
  - Interprocess communication
    - Message-passing model
    - Shared-memory model



Types of Communications

- **Message-passing**
  - Communicating processes message each other to transfer information.
  - They can be exchanged either **directly** or **indirectly** (through a common mailbox).
  - Each process has a *process_name*, and this is translated into an identifier so that OS can refer to the process.
    - *Get_hostid(), get_processid()*
    - Later, the identifiers are passed to *open()* and *close()*
      - Or *open_conn()* and *close_conn()* depending on OS.
  - Recipient must give permission for the communication
    - *Accept_communication()*
    - Usually, processes receiving connections are system programs designed for that purpose and they execute a *wait_for_connection()* call and awakened when a connection is made.
  - Then, client and server exchange messages by using *read_message()* and *write_message()* system calls.

Types of Communications

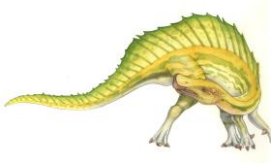
- **Shared-memory model**
  - Normally, OS tries to prevent one process from accessing another process' memory.
  - Shared memory requires two or more processes agree to remove this restriction.
    - Shared_memory_create()
    - Shared_memory_attach() to gain
      - To create and gain access to regions of memory owned by other processes
  - They can Exchange information by reading and writing data in the shared areas.
  - The form of data is determined by the processes, not OS.
    - Processes are also responsible for ensuring that they are not writing to the same location simultaneously. (We'll see them later)

Examples of Windows and Unix System Calls

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()`

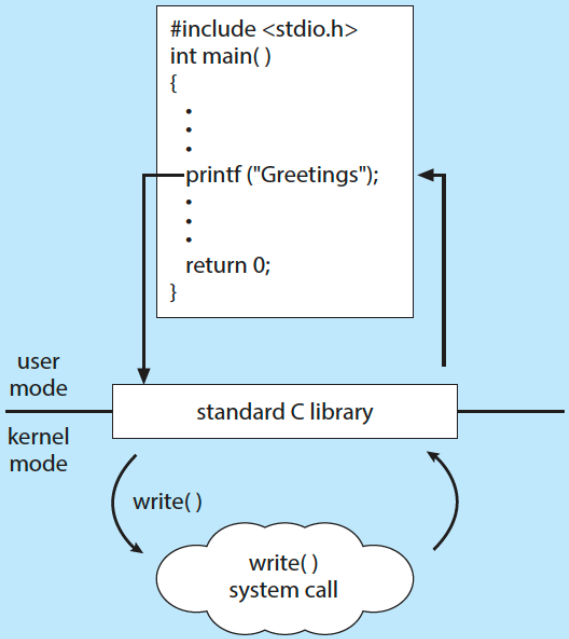


Standard C Library Example

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

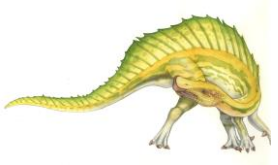
THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



System Services

- System programs (system utilities) provide a convenient environment for program development and execution. They can be divided into:
  - **File manipulation**
    - Create, delete, copy, rename, print, dump, list and generally manipulate files and directories.
  - **Status information**
    - Some ask the system for info - date, time, amount of available memory, disk space, number of users
    - Others provide detailed performance, logging, and debugging information
    - Typically, these programs format and print the output to the terminal or other output devices
    - Some systems implement a **registry** - used to store and retrieve configuration information
  - **Programming language support**
    - Compilers, assemblers, debuggers, interpreters are often provided.
  - **Program loading and execution**
    - **Linkers and loaders**: System tools essential for executing programs.
      - Linker combines multiple pieces of compiled code into a single executable.
      - Loader places this file into system memory, preparing it for execution by the OS.



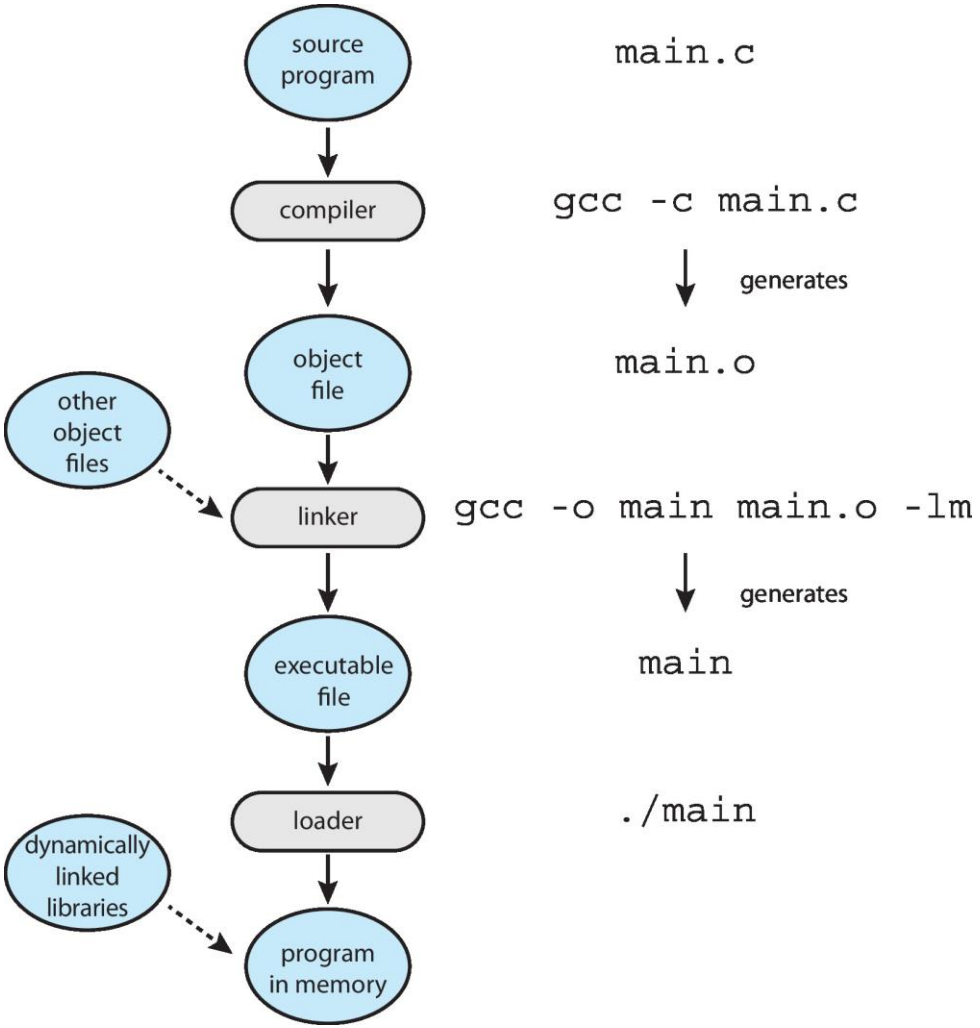
System Services (Cont.)

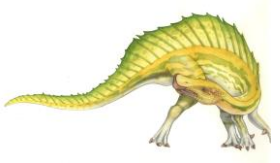
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another
- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context (by using daemons)
  - Known as **services**, **subsystems**, **daemons**
- **Application programs**
  - Web browsers, text processors, etc.
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke
- Most users' view of the operating system is defined by system programs, not the actual system calls

Linkers and Loaders

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

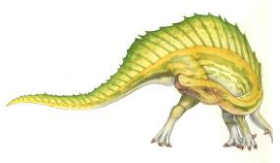
The Role of the Linker and Loader





Why Applications are Operating System Specific

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

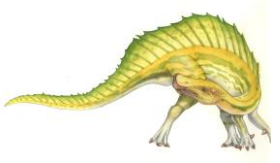


Operating System Structure

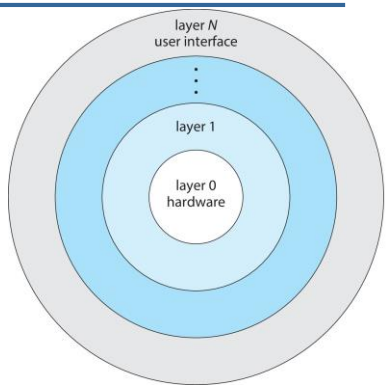
- General-purpose OS is very large program
- One approach is to partition into small components or modules
- The alternative is a **monolithic** system.

Monolithic Structure – Original UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level



Layered Approach

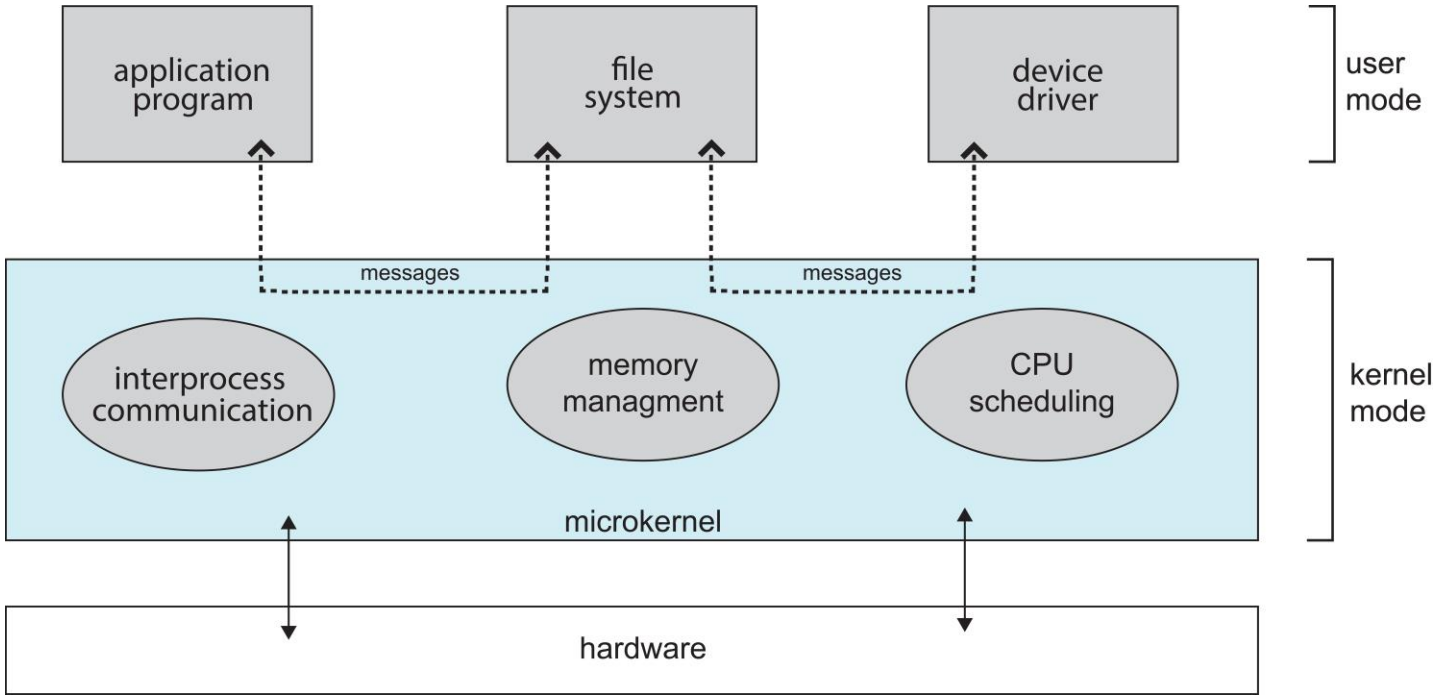


- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
  - This helps OS to retain much greater control over the computer and applications.
- Main advantage is simplicity of construction and debugging.
  - Layers can only use functions and services of only lower-level layers.
  - This helps for debugging because now the first layer can be debugged without any concern regarding the rest of the system.
    - Then, we can go to 2nd layer easily because we know layer 1 is ok.
- A layer does not need to know how operations of lower levels are implemented. They only know what they do.
  - Each layer hides the existence of certain data structures, operations and hardware from higher-level layers (information hiding)
- ***Difficulty:*** Since a layer can only use lower-layers, designing it is hard. i.e. The device driver for backing store (disk space for virtual memory algo) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

Microkernels

- As the OS expands, kernel can be difficult to manage.
  - CMU developed Mach.
    - It uses **microkernel** approach.
    - Structures the OS by removing all non-essential components from the kernel and implements them as system and user-level programs.
    - Therefore, we have a smaller kernel.
- Main function is to provide communication between the client and various services which are also running in the user space.
- Communication is provided through **message passing**
  - When the client wants to Access a file, it must interact with file server. Cannot interact with service directly.
- Benefits:
  - Extending the OS is easier
  - All new services are added to user space, don't need to modify the kernel.
  - Easier to port the operating system to new architectures
  - More secure and reliable (less code is running in kernel mode)
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure

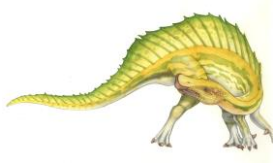


Modules

- Many modern operating systems implement **loadable kernel** mo**dules** (**LKMs**)
  - Each core component is separate
  - Each talks to the others over known interfaces
    - No need to invoke message passing
  - Each is loadable as needed within the kernel
    - Instead of modifying the kernel new modules are added
- Overall, similar to layered approach but with more flexible
  - Because any module can call other modules
- Similar to microkernel
  - Primary modüle has only core functions and knowledge of how to load and communicate with others.
- The idea is for the kernel to provide core services while other services are implemented dynamically, as the kernel is running.

Summary

- At the lowest level, system calls allow a running program to make requests from the OS directly.
- At higher levels, the command interpreter or Shell provides a mechanism for the user to issue a request without writing a program.
  - Commands can come from terminal or GUI.
  - System programs are provided to satisfy many common user requests.
- System call level must provide the basic functions:
  - Process control, file and device manipulation, etc.
- Higher level requests are translated into a sequence of system calls.
- Designing an OS is hard. Modularity is important.
  - Using a layered approach or microkernel is considered good technique.



End of Chapter 2