Operating-System Structures

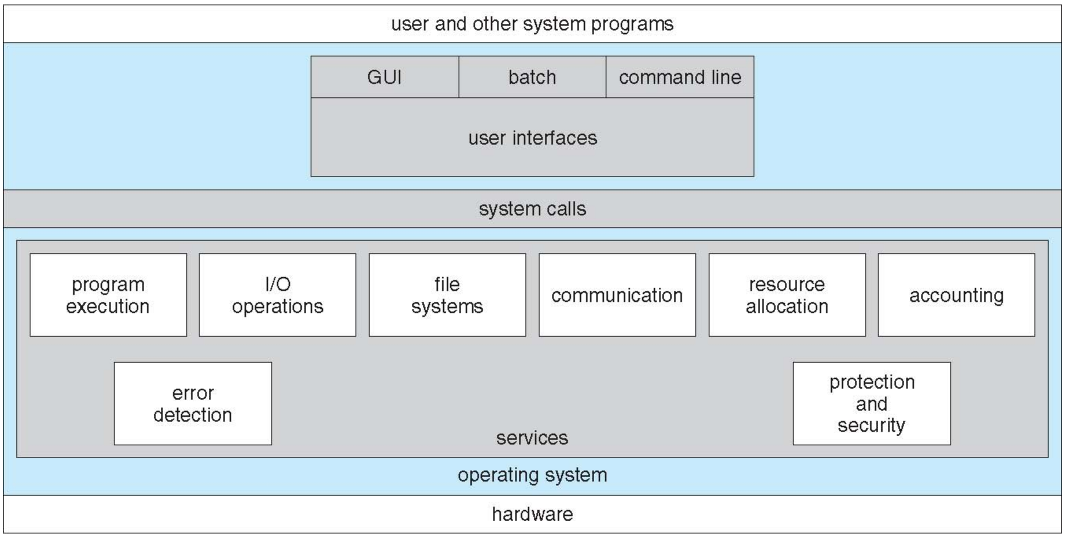
Operating-System Structures

- An environment for the execution of programs and services to programs and users
- Providing functions that are helpful to the user:
  - User interface
  - Program execution
  - I/O operations
  - File-system manipulation
  - Communications
  - Error detection

Operating System Services (Cont.)

- Another set of OS functions exists to ensure the efficient operation of the system itself via resource-sharing
  - Resource allocation
  - Protection and security
  - Protection involves ensuring that all access to system resources is controlled
  - Security of the system from outsiders requires user authentication extends to defending external I/O devices from invalid access attempts

A View of Operating System Services

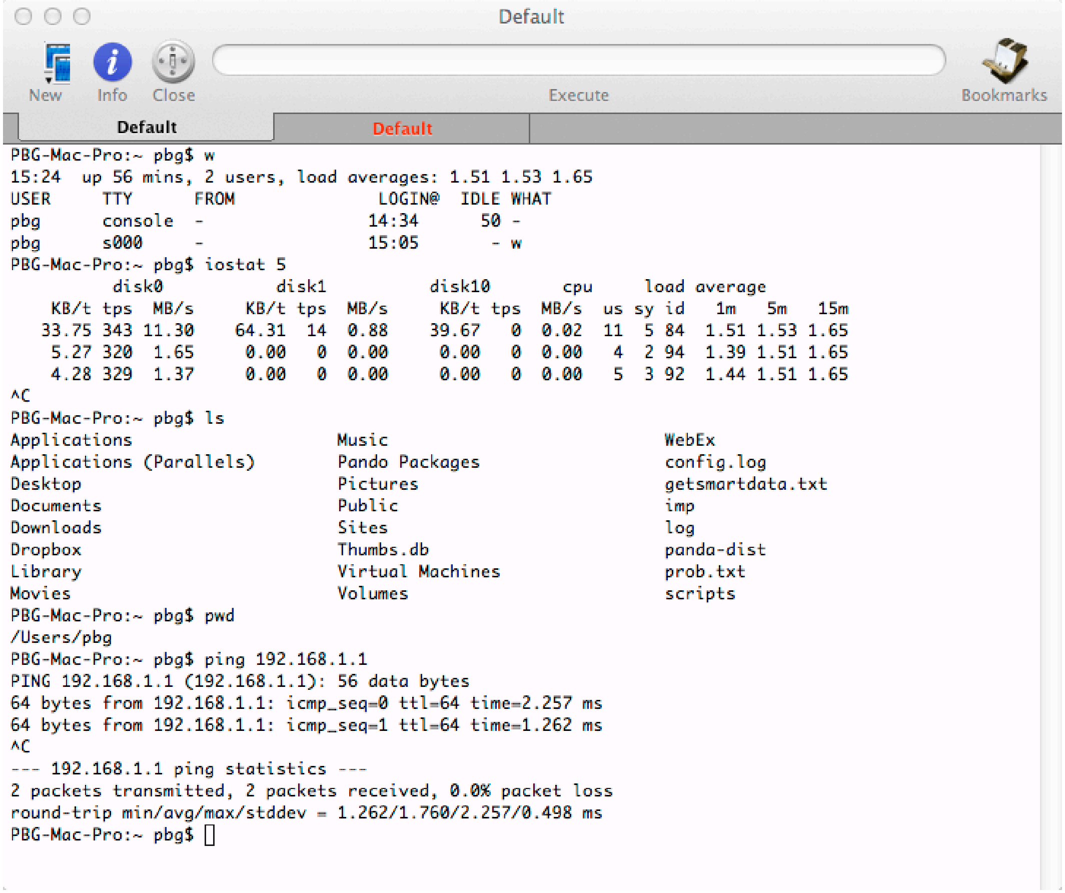


User Operating System Interface - CLI

- CLI or command interpreter allows direct command entry
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – shells

- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
- If the latter, adding new features doesn’ t require shell modification

Bourne Shell Command Interpreter



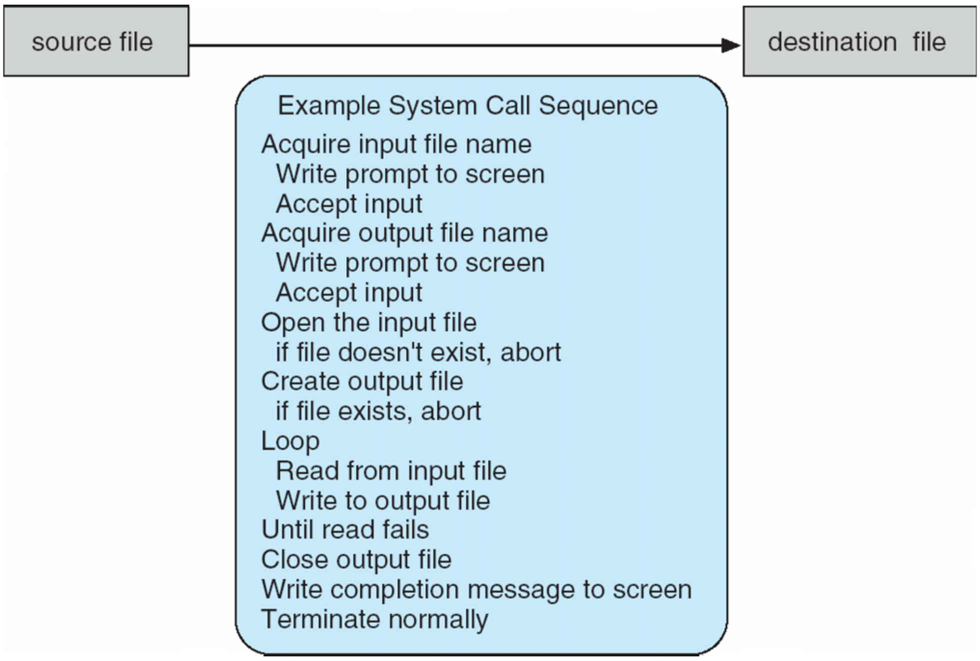
```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                14:34      50 -
pbg      s000     -                15:05       - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1            disk10       cpu     load average
    KB/t tps  MB/s     KB/t tps  MB/s     KB/t tps  MB/s  us sy id   1m   5m   15m
   33.75 343 11.30    64.31  14  0.88    39.67   0  0.02  11  5 84  1.51 1.53 1.65
    5.27 320  1.65     0.00   0  0.00     0.00   0  0.00   4  2 94  1.39 1.51 1.65
    4.28 329  1.37     0.00   0  0.00     0.00   0  0.00   5  3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications                    Music                           WebEx
Applications (Parallels)        Pando Packages                  config.log
Desktop                         Pictures                        getsmartdata.txt
Documents                       Public                          imp
Downloads                       Sites                           log
Dropbox                         Thumbs.db                       panda-dist
Library                         Virtual Machines                prob.txt
Movies                          Volumes                         scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$
```

System Calls

- Programming interface to the services provided by the OS Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

System Calls

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



Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Application Programming Interface

- Typically, application developers design programs according to an application programming interface (API).
- The API specifies a set of functions available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.
- A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called libc.

Application Programming Interface

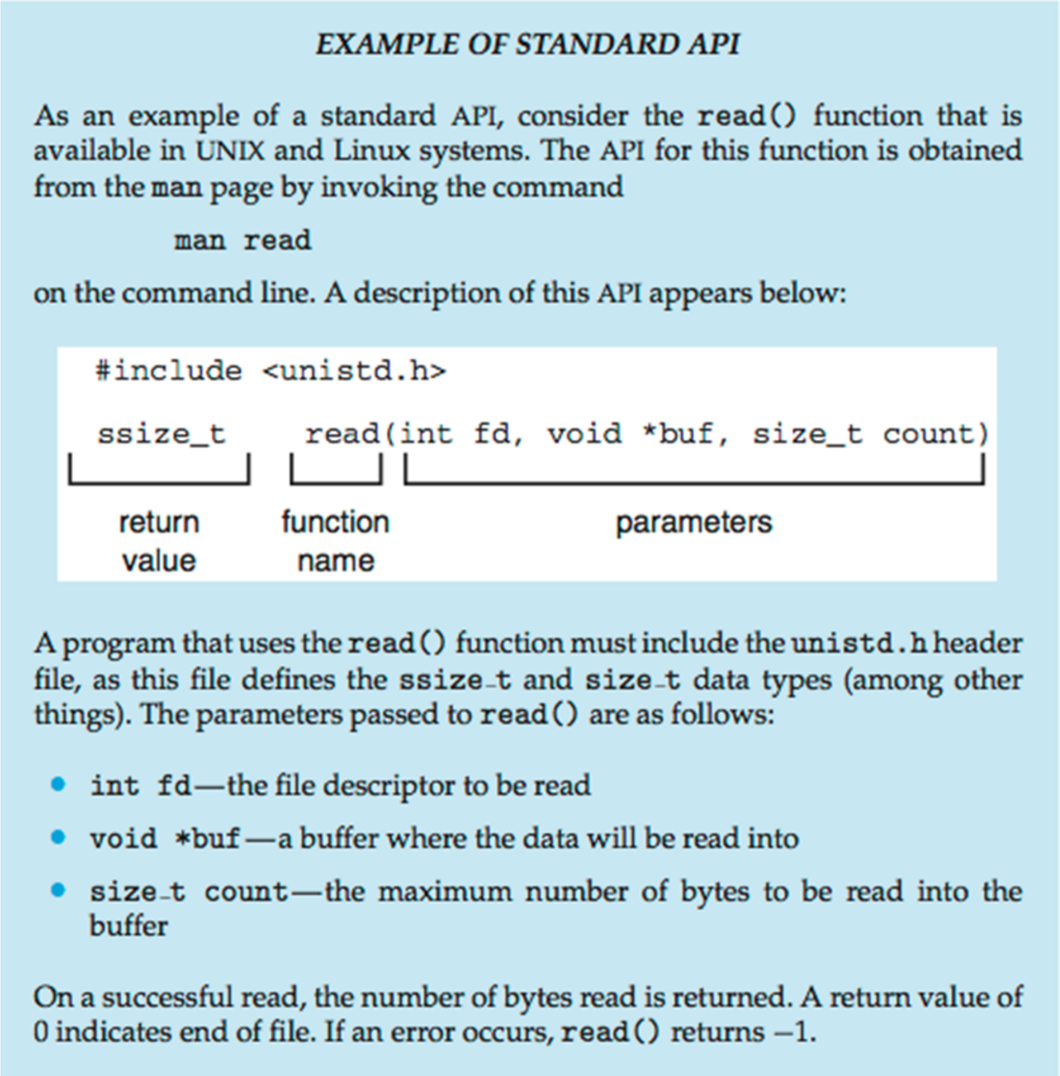
- Why would an application programmer prefer programming according to an API rather than invoking actual system calls?
  - Program portability
  - Actual system calls can often be more detailed and difficult to work with than the API.
  - A strong correlation between a function in the API and its associated system call within the kernel.

System Call Implementation

- Typically, a number 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 the status of the system call and any return values The caller needs to know nothing about how the system call is implemented

- 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)

Example of Standard API



EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

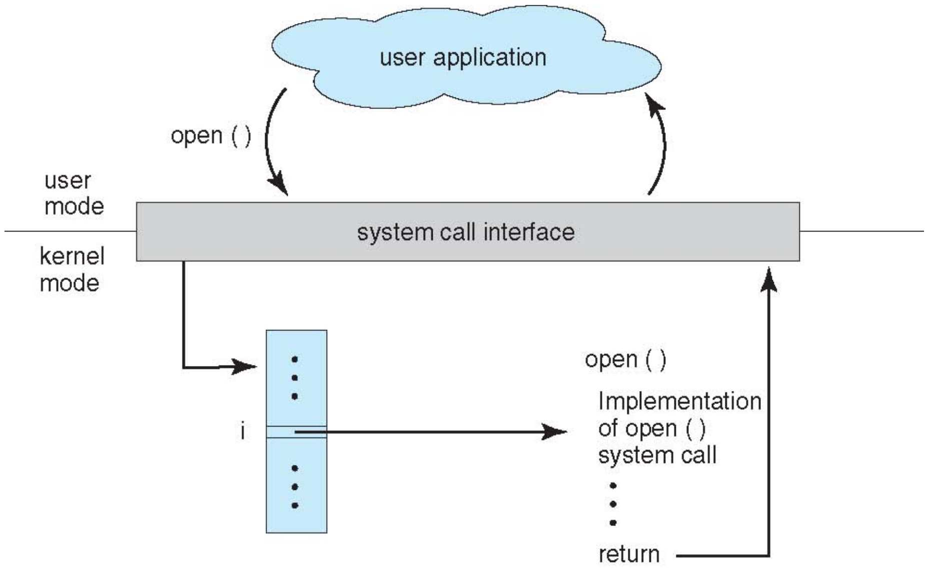
return value
function name
parameters

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

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

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

API – System Call – OS Relationship



Types of System Calls

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

Types of System Calls (Cont.)

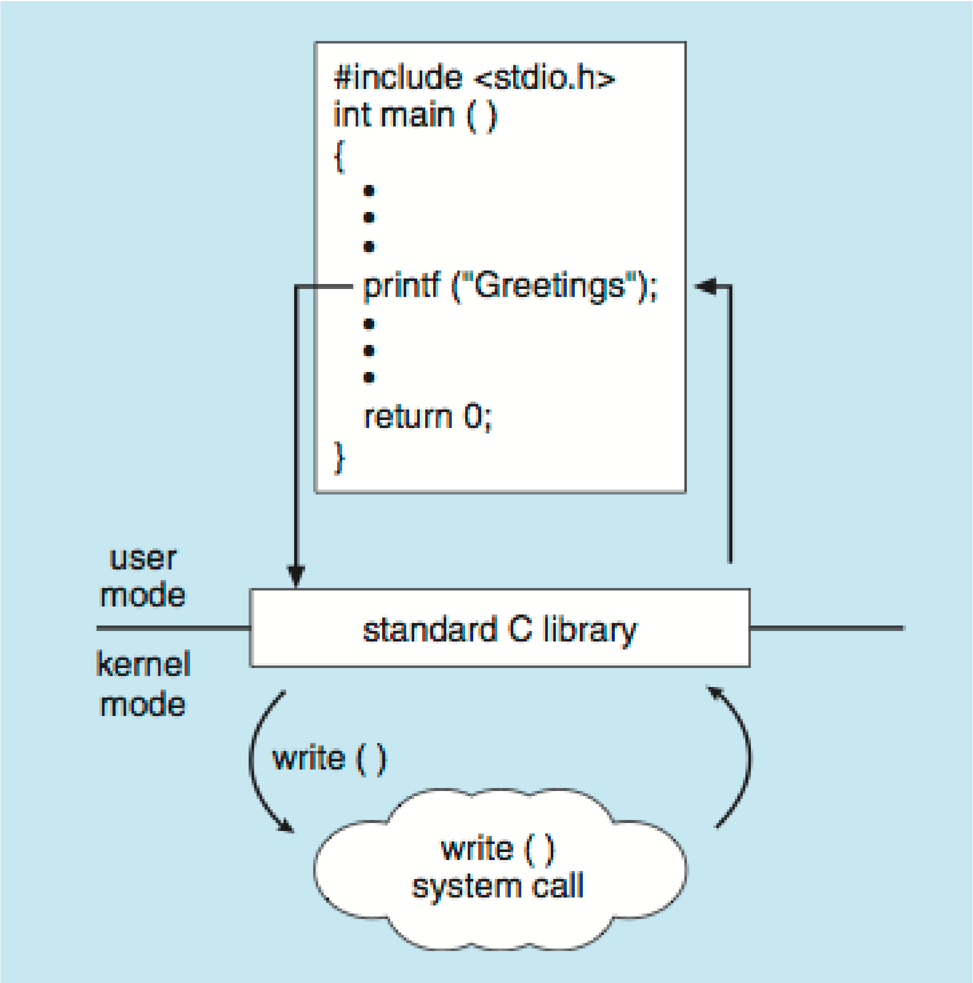
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

Types of System Calls (Cont.)

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

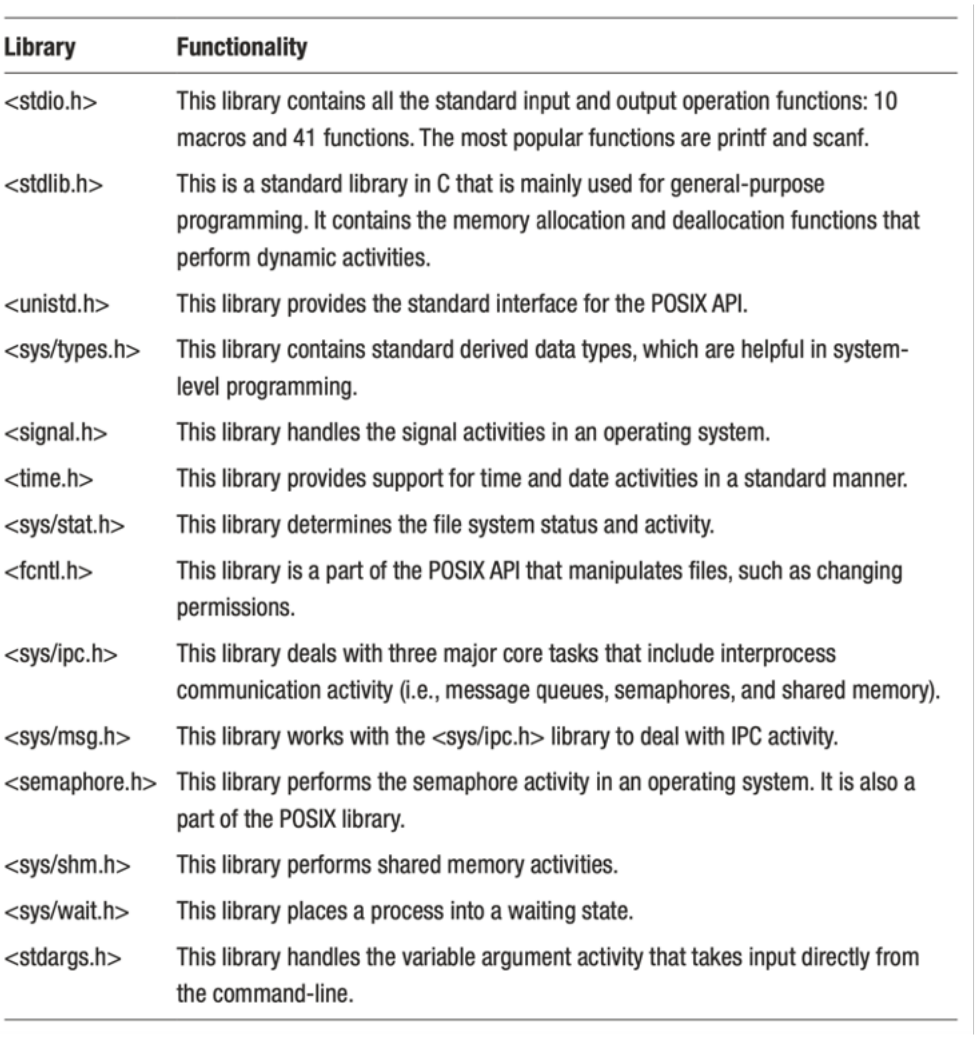
Standard C Library Example

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



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

C Libraries



Library	Functionality
<stdio.h>	This library contains all the standard input and output operation functions: 10 macros and 41 functions. The most popular functions are printf and scanf.
<stdlib.h>	This is a standard library in C that is mainly used for general-purpose programming. It contains the memory allocation and deallocation functions that perform dynamic activities.
<unistd.h>	This library provides the standard interface for the POSIX API.
<sys/types.h>	This library contains standard derived data types, which are helpful in system-level programming.
<signal.h>	This library handles the signal activities in an operating system.
<time.h>	This library provides support for time and date activities in a standard manner.
<sys/stat.h>	This library determines the file system status and activity.
<fcntl.h>	This library is a part of the POSIX API that manipulates files, such as changing permissions.
<sys/ipc.h>	This library deals with three major core tasks that include interprocess communication activity (i.e., message queues, semaphores, and shared memory).
<sys/msg.h>	This library works with the <sys/ipc.h> library to deal with IPC activity.
<semaphore.h>	This library performs the semaphore activity in an operating system. It is also a part of the POSIX library.
<sys/shm.h>	This library performs shared memory activities.
<sys/wait.h>	This library places a process into a waiting state.
<stdargs.h>	This library handles the variable argument activity that takes input directly from the command-line.

System Calls and I/O Operations for Files

creat

- This system call creates a new empty file with a system call. It is available in the fcntl.h library, which is a file handling library for Unix and Linux.
- The return type for this function is an integer. If file creation is successful, it returns a non-negative integer. If the creation of the file fails, it returns -1.
- The following shows the syntax. `int creat(char *filename, mode_t mode);`
- The first parameter in the creat function is the name of a file.

creat contd...

- The second parameter, mode, deals with the permissions of the file. The permission modes are different from normal Linux file system permissions. There are various modes available for this flag, but the following are the most common modes.

- O_RDONLY: If you set this flag mode to the creat function, the file has read-only permission.
- O_WRONLY: This mode gives write permissions.
- O_RDWR: This mode gives both read and write permissions.
- O_EXCL: This flag mode prevents the creation of a file if it already exists.
- O_APPEND: This mode appends the content to existing file data without overriding it.
- O_CREAT: This flag mode creates a file if it does not exist.
- If you want to use multiple modes at the same time, you can use the bitwise OR operator.

Example: create.c

open

- The open system call function opens a file and can perform read and write operations based on the mode set to the function. An open system call can also create a file.
- If the specified filename is not available, then it automatically creates a new file with the given name. The return type of this function is an integer.
- If the file opens successfully, it returns a positive integer value; otherwise, it returns -1.
- The following shows the syntax.

```
int open(const char *filepath, int flags, ...);
```

- The first parameter deals with the absolute path of a file that you want to open.
- The flags that are passed as a second argument are O_RDONLY, O_WRONLY, O_RDWR, and so forth.
- Example: open.c

close

- This system call closes the file descriptor that was created to open, create, or read the contents in a file. The return type of this function is an integer.
- If the file descriptor is closed successfully, it returns 0; otherwise, it returns -1.
- The following shows the syntax.

  ```
  int close(int file_descriptor);
  ```

- file_descriptor is an integer value that identifies the open file in a process.
- Example: close.c

read

This function system call reads the content of a file that was indicated by a file descriptor. The return type of this function is an integer. It returns -1 if an error occurs or when any signal interrupt occurs during a read operation. A successful read of a file returns the number of bytes read during the operation.

The following shows the syntax.

```
size_t read (int file_descriptor, void* buffer, size_t size);
```

- file_descriptor is a unique integer value that identifies the open file in a process.
- The `buffer` argument reads the file data.
- `size` is the third argument indicates the size of the buffer that you want to read from the file.

Example:read.c

write

- This function writes content to a given file descriptor.
- The return type of this file is an integer. It returns -1 for an error or if any signal interrupt is raised; otherwise, it returns the number of bytes that are returned to a file.
- The following shows the syntax.

```
size_t write (int file_descriptor, void* buffer, size_t size);
```

- This function syntax is the same as the read function. However, the key difference is that it writes the content in a file using the buffer. The read function reads the content from a file using a buffer.
- Example: write.c

System Calls for Directories: Creating a Directory

- The creation of a directory is done with the mkdir function, which is available in the sys/ stat.h library. The return type of this function is an integer.
- It returns 0 on the successful creation of a directory; it returns -1 for a failure.

```
int mkdir(const char *path, mode_t mode);
```

- path is the first argument that describes the path and the new directory name to create in the system.
- mode represents the permissions to give to a new directory.

Deleting a Directory

- The deletion of a directory is done with the rmdir function, which is available in the sys/stat.h library. The return type of this function is an integer.
- It returns 0 on the successful deletion of a directory; it returns -1 if a failure.
- The following shows the syntax.

```
int rmdir(const char *pathname);
```

- pathname determines the directory name with the absolute path to remove from the system.

Getting the Current Working Directory

- The getcwd function gets the current working directory.
- It is available in the unistd.h library. The return of this function is a character data type. It returns the program's current working directory.

```
char getcwd(char *buffer, size_t buffersize);
```

- buffer is the first argument; it describes the char array that stores the buffer content.
- buffersize is the second argument; it is the length of the buffer.

Changing Directory

- There is a chdir system call that changes directory in your operating system. It is available in the unistd.h library.
- The return type of this function is an integer. It returns 0 on the successful change in a directory; it returns -1 for a failure.
- The following shows the syntax.

```
int chdir(const char *path);
```

- path describes the path to change.

Reading a Directory

- There are two types of functions that read the content in directories: opendir and readdir.
- They are available in the dirent.h library. The return type of the opendir function is the directory stream.
- A directory stream is an ordered sequence of all directory entries in a directory. A directory entry represents the files. This directory stream points to the start position.
- The return type of **readdir** is a **dirent** structure, which returns NULL if the directory reaches its end. Dirent is a built-in structure that is implemented in the dirent.h library.
- The following shows the syntax.

```
DIR *opendir(const char *path);
```

Reading a Directory Contd..

- The path argument indicates the value that you want to open.

```
struct dirent *readdir(DIR* directorypointer);
```

- The `directorypointer` argument should contain the directory stream pointer, which is a return value of the opendir function.
- The internal structure of dirent is as follows.

```
struct dirent{

ino_t d_ino; // inode number

off_t d_off; // offset to the next dirent unsigned short d_reclen; //
length of this record unsigned char d_type; // type of file;

char d_name[256]; // filename

};
```

Closing a Directory

- The closedir function closes the directory stream that is running in a process. The return type of this function is an integer.
- It returns 0 on the successful closing of a directory; it returns -1 for a failure.
- The following shows the syntax.

```
int closedir(DIR *directorypointer);
```

- **directorypointer** is an argument that contains the directory stream pointer, which is simply a return value of the opendir function.

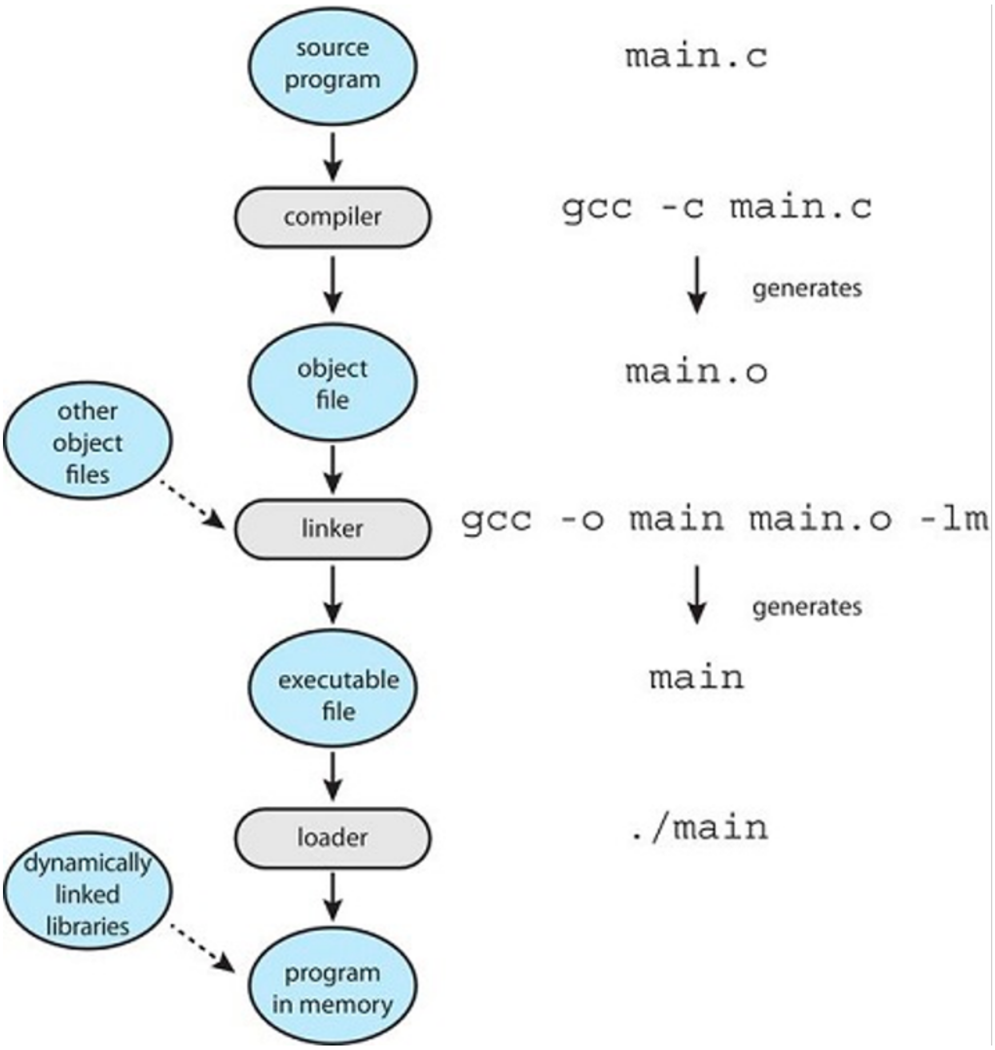
System Programs

- Provide a convenient environment for program development and execution
  - File management
  - Programming-language support
  - Program loading and execution
  - Communications

Linker loader

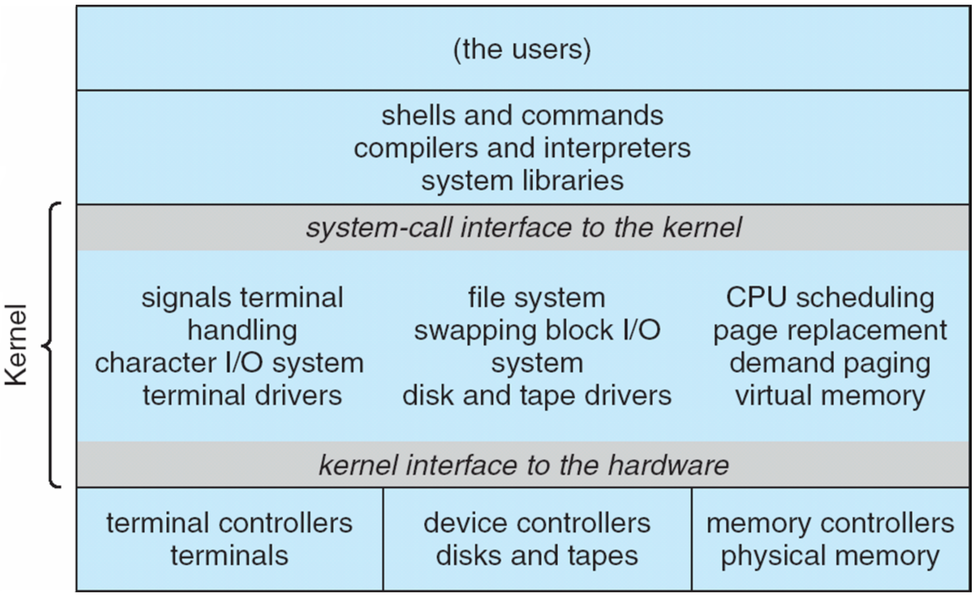
- A loader is used to load the binary executable file into memory, where it is eligible to run on a CPU core.
  - An activity associated with linking and loading is relocation , which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses.
  - Most systems allow a program to dynamically linked libraries (DLLs) as the program is loaded.
  - Object files and executable files typically have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program.
  - For UNIX and Linux systems, this standard format is known as ELF (for Executable and Linkable Format).

Linker and Loaders

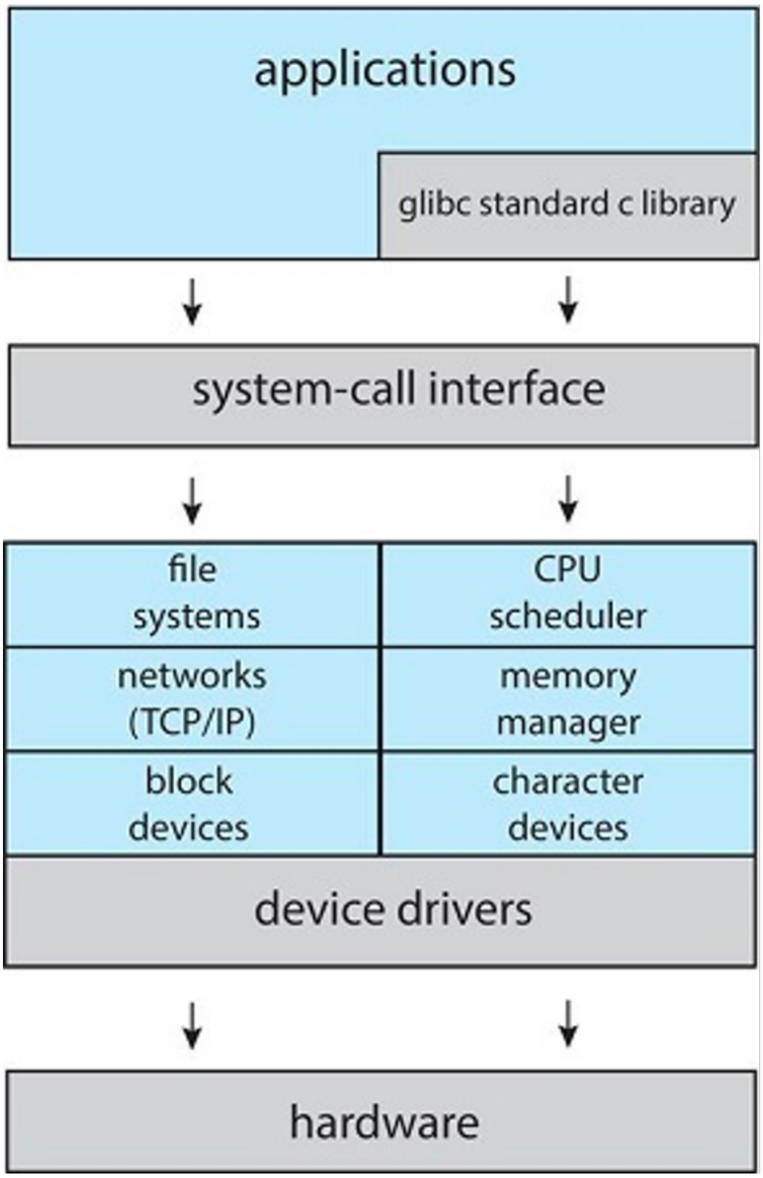


Operating System Design: Monolithic UNIX System Structure

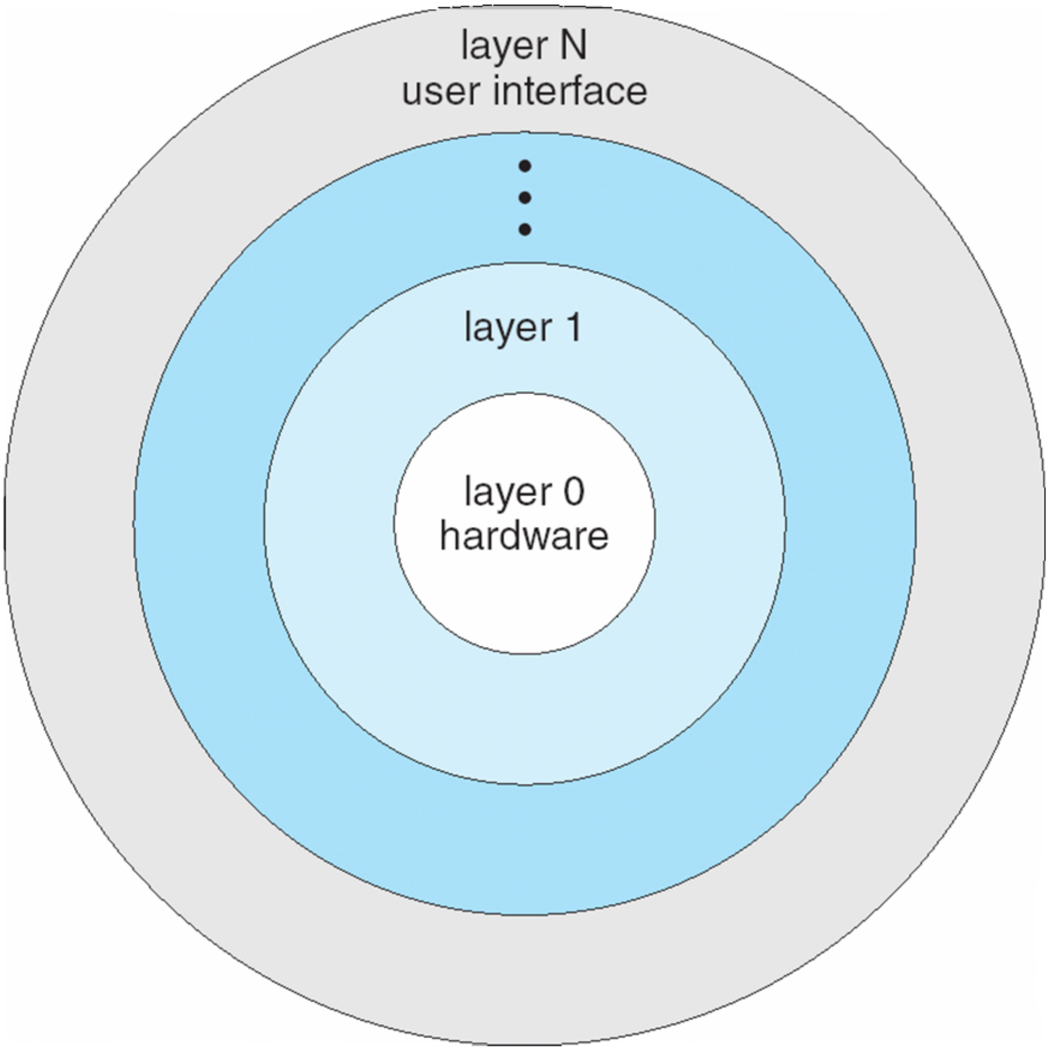
- Beyond simple but not fully layered



Linux System Structure

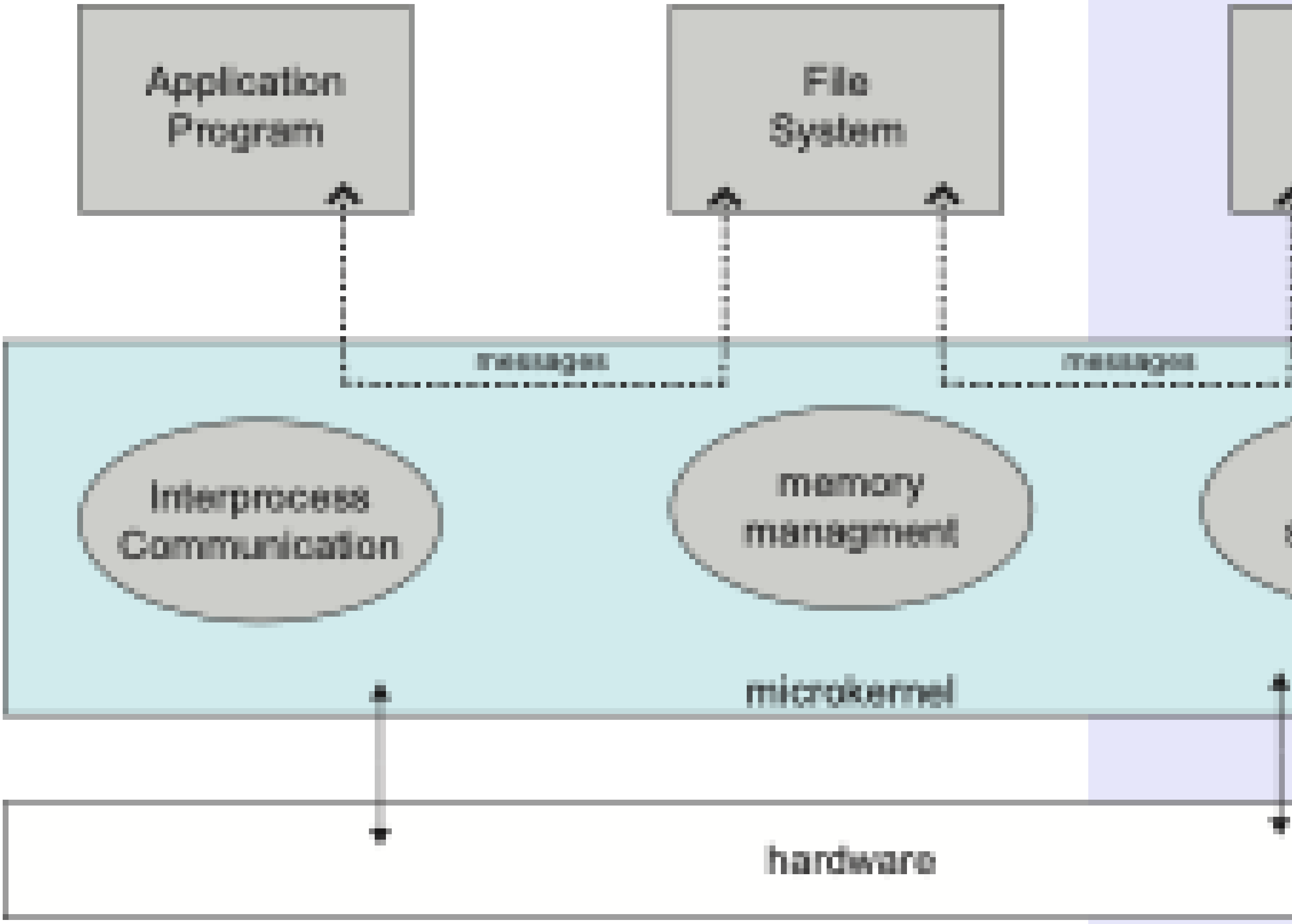


Layered Approach



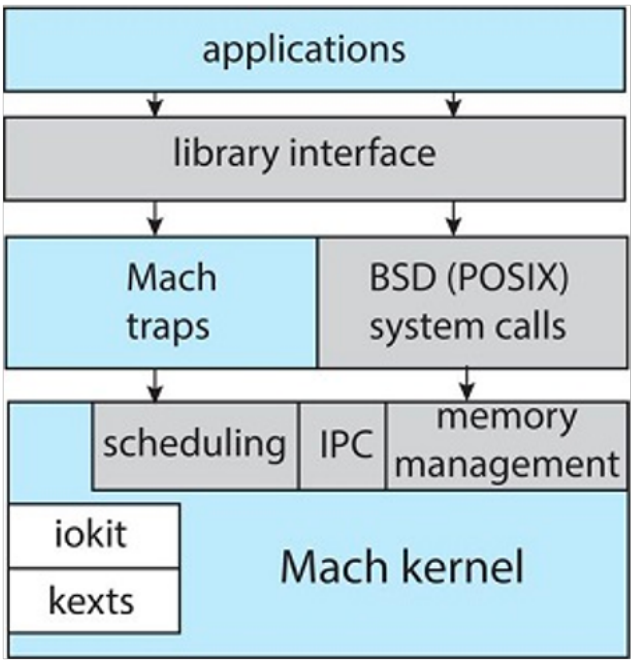
Microkernel System Structure

- Darwin macOS is example of microkernel OS

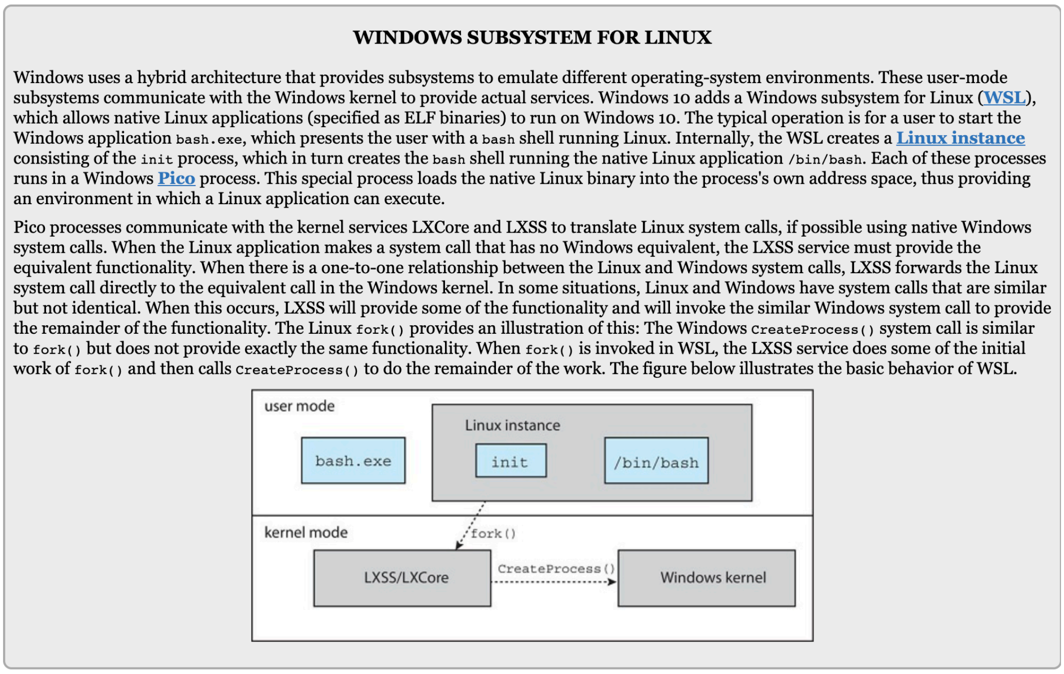


Hybrid Systems

- Hybrid combines multiple approaches to address performance, security, usability needs
- Apple Mac OS X hybrid, layered, Aqua UI plus Cocoa programming environment Two system call interfaces: Mach traps and BSD



Window Subsystem for Linux



WINDOWS SUBSYSTEM FOR LINUX

Windows uses a hybrid architecture that provides subsystems to emulate different operating-system environments. These user-mode subsystems communicate with the Windows kernel to provide actual services. Windows 10 adds a Windows subsystem for Linux (**WSL**), which allows native Linux applications (specified as ELF binaries) to run on Windows 10. The typical operation is for a user to start the Windows application `bash.exe`, which presents the user with a `bash` shell running Linux. Internally, the WSL creates a **Linux instance** consisting of the `init` process, which in turn creates the `bash` shell running the native Linux application `/bin/bash`. Each of these processes runs in a Windows **Pico** process. This special process loads the native Linux binary into the process's own address space, thus providing an environment in which a Linux application can execute.

Pico processes communicate with the kernel services LXCore and LXSS to translate Linux system calls, if possible using native Windows system calls. When the Linux application makes a system call that has no Windows equivalent, the LXSS service must provide the equivalent functionality. When there is a one-to-one relationship between the Linux and Windows system calls, LXSS forwards the Linux system call directly to the equivalent call in the Windows kernel. In some situations, Linux and Windows have system calls that are similar but not identical. When this occurs, LXSS will provide some of the functionality and will invoke the similar Windows system call to provide the remainder of the functionality. The Linux `fork()` provides an illustration of this: The Windows `CreateProcess()` system call is similar to `fork()` but does not provide exactly the same functionality. When `fork()` is invoked in WSL, the LXSS service does some of the initial work of `fork()` and then calls `CreateProcess()` to do the remainder of the work. The figure below illustrates the basic behavior of WSL.

System Boot

- Bootloader is the first program that executes after power, and it never relies on the kernel. Like Uboot, Vivi, BIOS
- Bootstrap, the second stage boot loader, belongs to kernel code that act as a link between bootloader and kernel mirroring.
- Bootstrap typically verifies kernel mirroring, compresses kernel mirroring, deploys kernel mirroring to memory, and provides the appropriate context for kernel execution Execution Process –>bootloader–>bootstrap (HEAD.O)–> kernel vmlinux (HEAD.O)–> kernel Start_kernel (MAIN.O)

Performance monitoring tools

- Counters
  - Operating systems keep track of system activity through a series of counters.
  - Per-Process
  - ps —reports information for a single process or selection of processes
  - top—reports real-time statistics for current processes

Performance monitoring tools

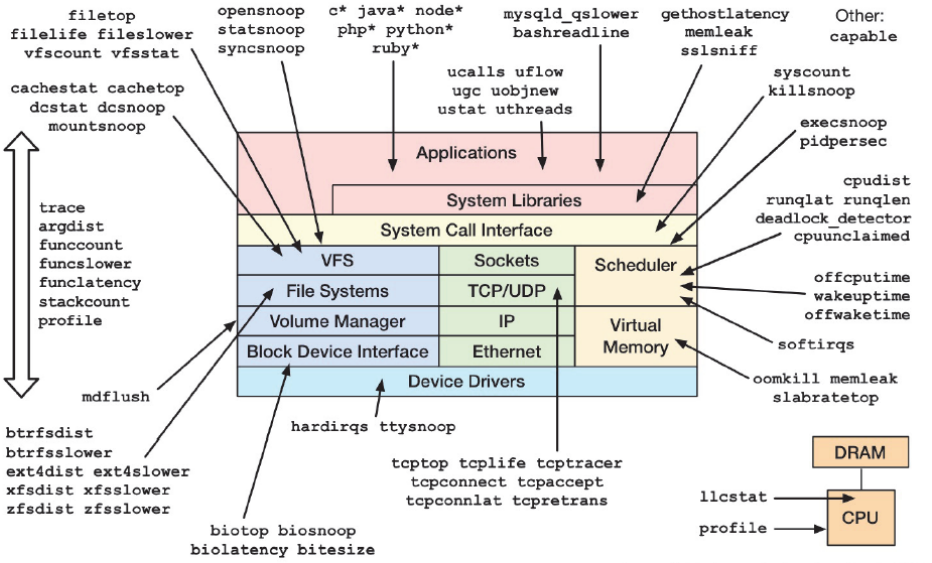
- System-Wide
  - vmstat —reports memory-usage statistics
  - netstat—reports statistics for network interfaces
  - iostat —reports I/O usage for disks
  - Most of the counter-based tools on Linux systems read statistics from the /proc file system. /proc is a “pseudo” file system

Performance monitoring tools

- Tracing
  - Whereas counter-based tools simply inquire on the current value of certain statistics that are maintained by the kernel, tracing tools collect data for a specific event—such as the steps involved in a system-call invocation.
  - Per-Process
  - strace —traces system calls invoked by a process
  - gdb —a source-level debugger
  - System-Wide
  - perf—a collection of Linux performance tools
  - tcpdump —collects network packets

BCC (BPF Compiler Collection)

- On virtualBox Ubuntu look at /usr/share/bcc/tools to see tools
- The details for the tools are given at https://github.com/iovisor/bcc



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

References

- A. Silberschatz, P.B. Galvin, and G. Gagne. Operating System Concepts , 10th Edition; 2018; John Wiley and Sons.
- W. Stallings. Operating Systems: Internals and Design Principles. Prentice Hall, Upper Saddle River, NJ, Eigth edition, 2014.
- K.A. Robbins and S. Robbins. UNIX Systems Programming: Concurrency, Communication, and Threads. Prentice Hall, Upper Saddle River, NJ, Second edition, 2003

Related Sections

- All sections included except subsections: 2.2.4, 2.7.3, 2.8.5, 2.10.4