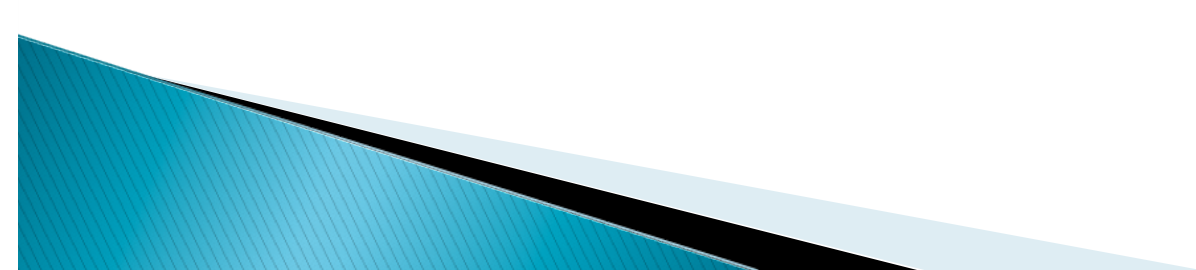


Chapter 2:

Operating-System Structures

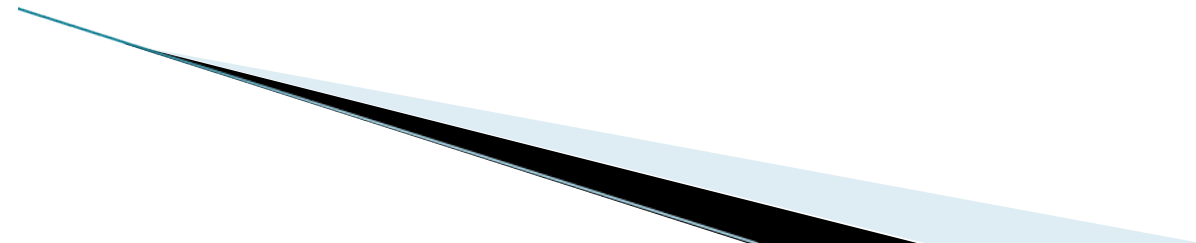
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Let's use an **example** to illustrate how system calls are used:
- **writing a simple program to read data from one file and copy them to another file.**
- The first input that the program will need is the names of the two files. One approach is for the program to ask the user for the names of the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.
- Now that both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions.



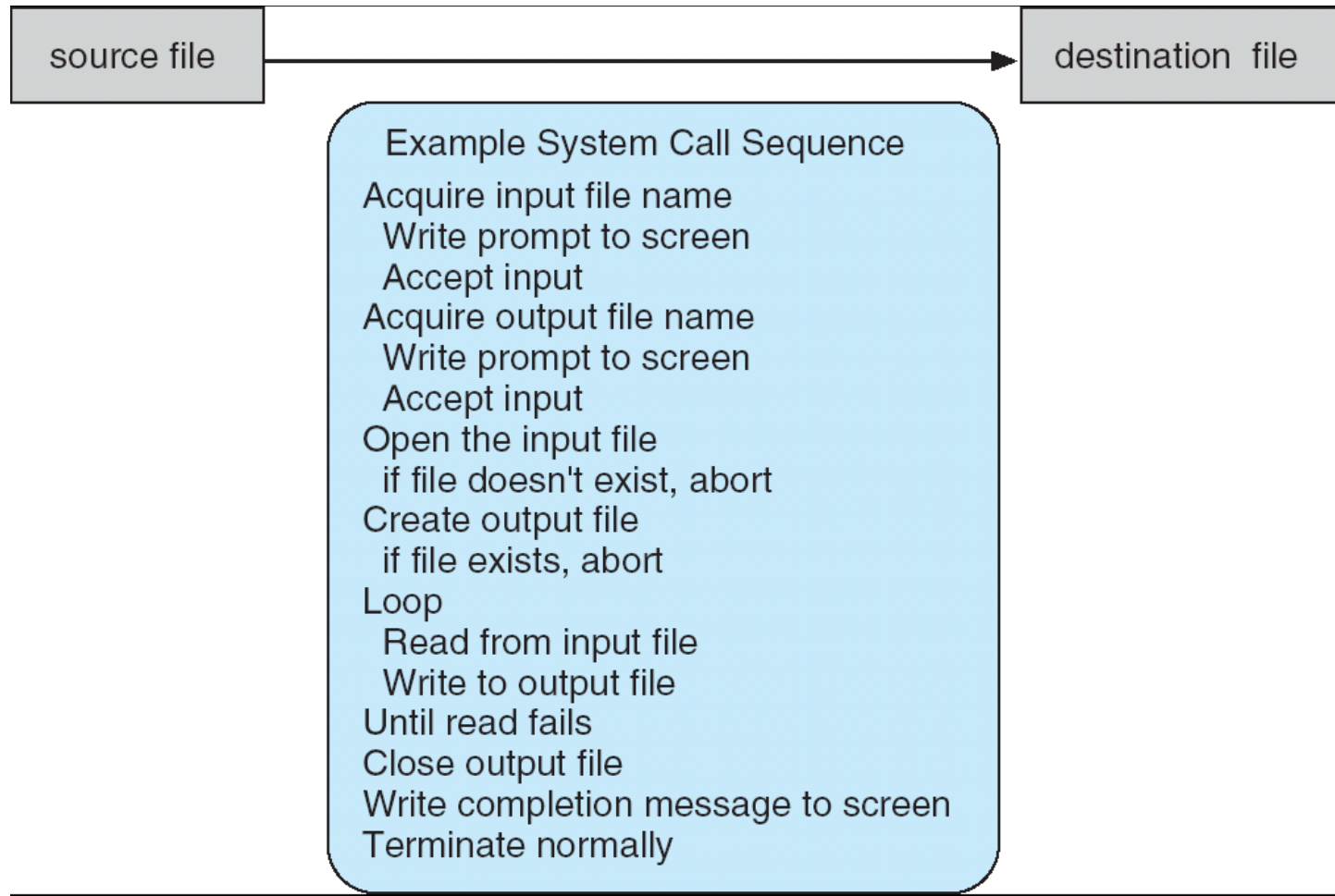
System Calls

- Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). As we can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second.



Example of System Calls

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



System Calls

- Most programmers never see this level of detail. Typically, application developers design programs according to an application programming interface (API). The API specifies a set of functions that are available to an application.
- 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)**
 - **Java API for the Java virtual machine (JVM)**
- Why use APIs rather than system calls?

Example of Standard API

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

return value



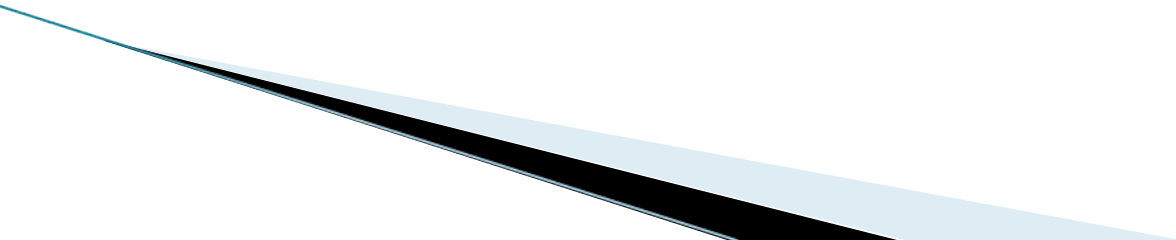
```

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


parameters

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

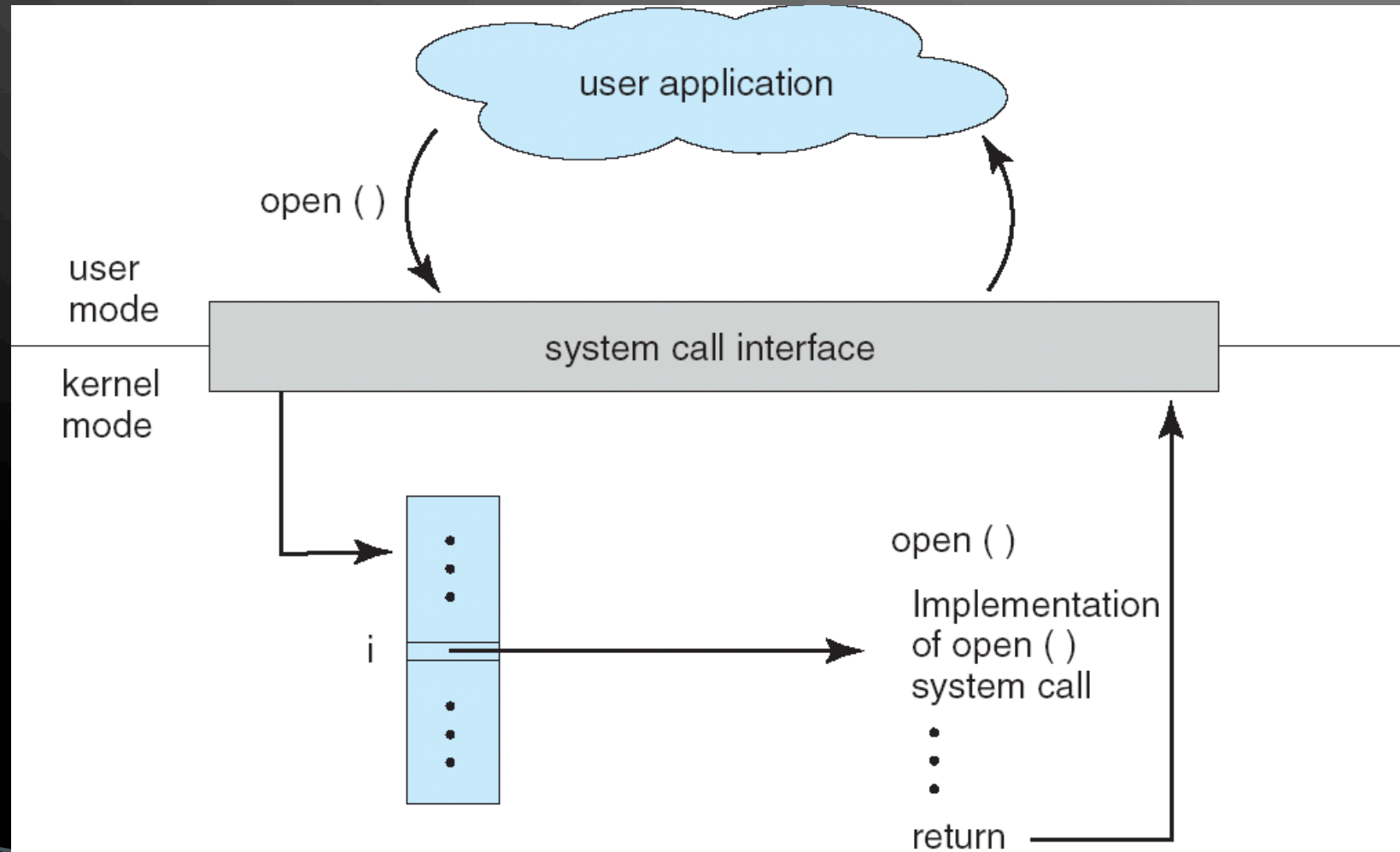
System Calls

- **Why use APIs rather than system calls?**
 - One benefit of programming according to an API concerns program portability: An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API
 - Secondly, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer.
- 

System Call Interface

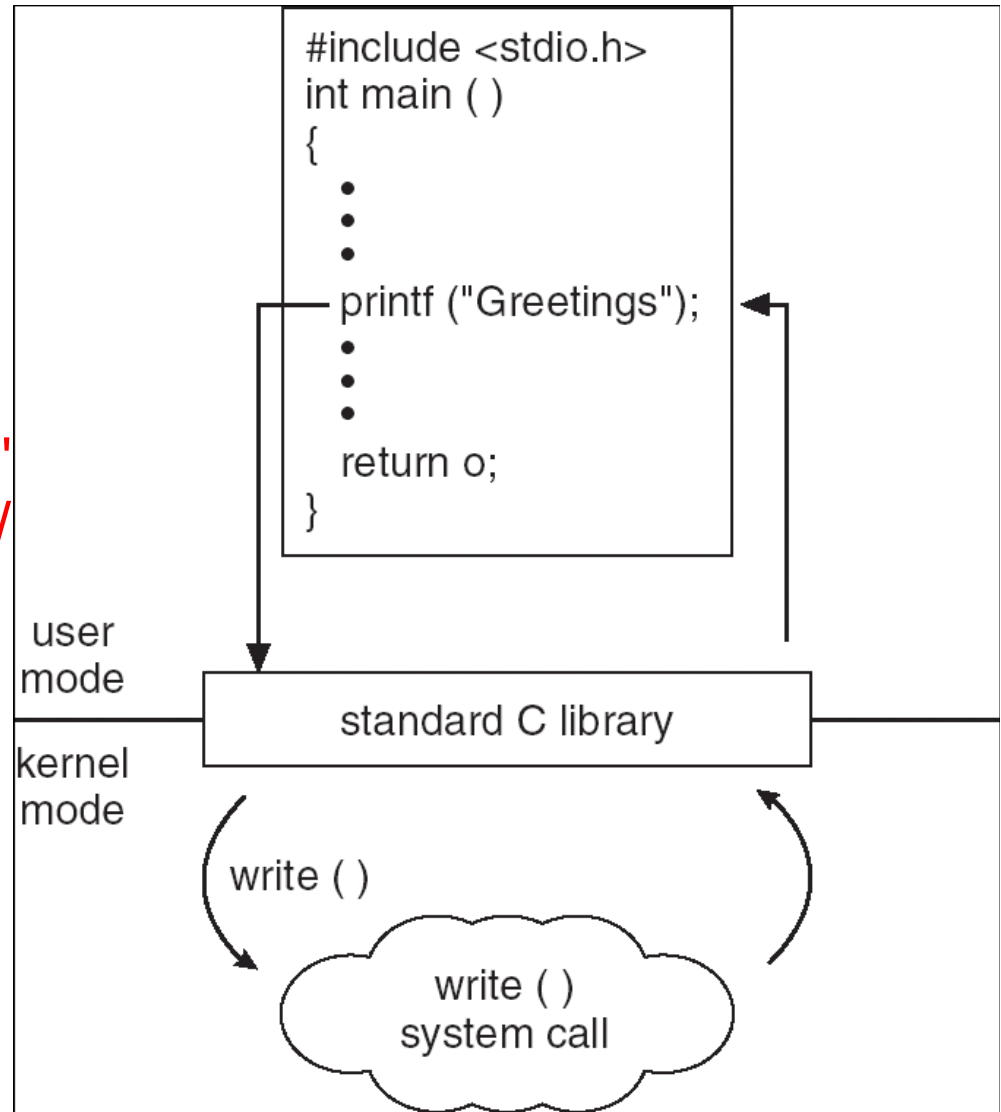
- The run-time support system (a set of functions built into libraries included with a compiler) for most programming languages provides a system-call interface that serves as the link to system calls made available by the operating system.
 - The system-call interface intercepts function calls in the API and invokes the necessary system call within the operating system.
 - The caller needs to know nothing about how the system call is implemented or what it does during execution. Rather, it just needs to obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.
- 

API – System Call – OS Relationship



Standard C Library Example

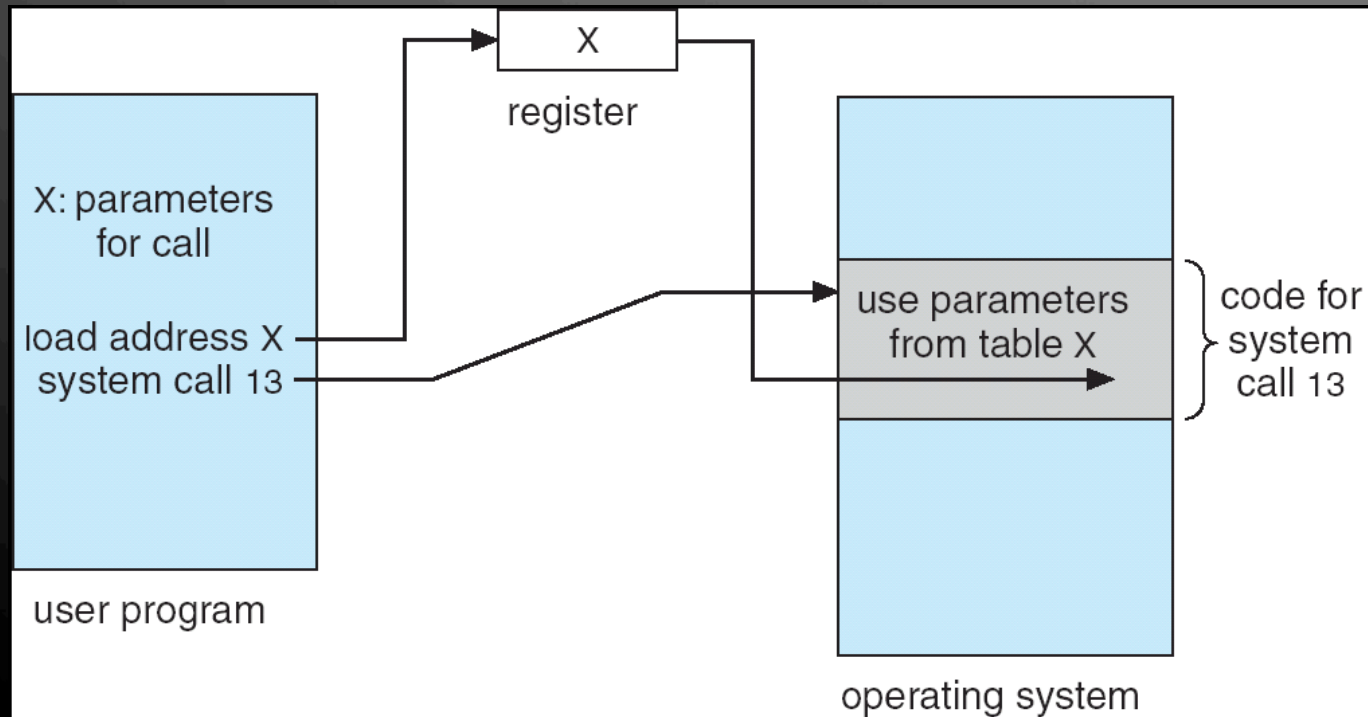
- C program invoking printf() library call, which calls write() system call
- In C#
// Read the file as one string.
string text =
System.IO.File.ReadAllText(@"
C:\Users\Public\TestFolder\W
riteText.txt");



System Call Parameter Passing

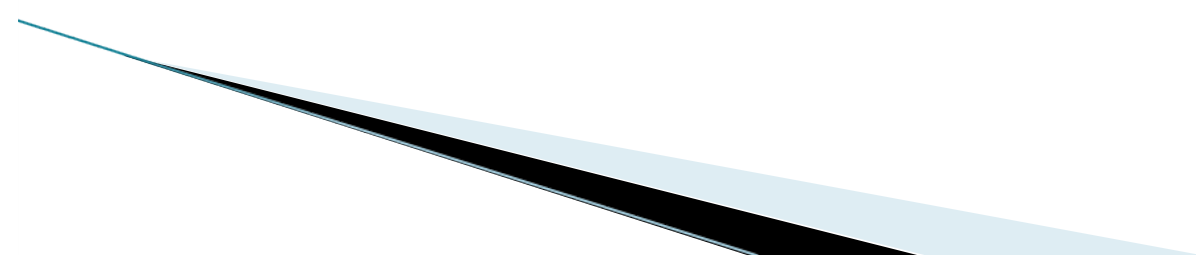
- Often, more information is required than simply identity of desired system call
 - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
 - Simplest: pass the parameters in **registers**
 - In some cases, may be more parameters 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
 - *Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

System calls can be grouped roughly into five major categories

- ❑ Process control
 - ❑ File management
 - ❑ Device management
 - ❑ Information maintenance
 - ❑ Communications
- 

Process Control

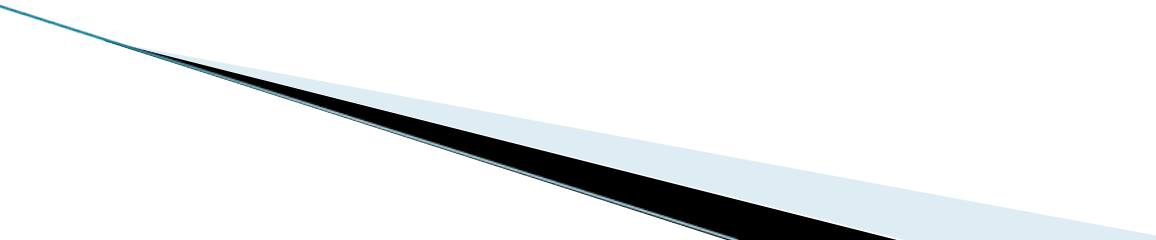
It contains following type of system calls normally.

- 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

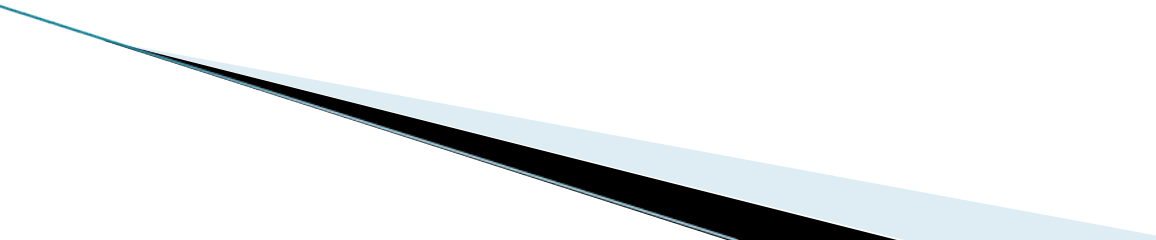
Process Control (System calls)

- A running program needs to be able to halt its execution either normally (**end**) or abnormally (**abort**). In case of trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a debugger.
- A process or job executing one program may want to **load and execute** another program. This feature allows the command interpreter to execute a
- program as directed by, for example, a user command, the click of a mouse.

Process Control (System calls)

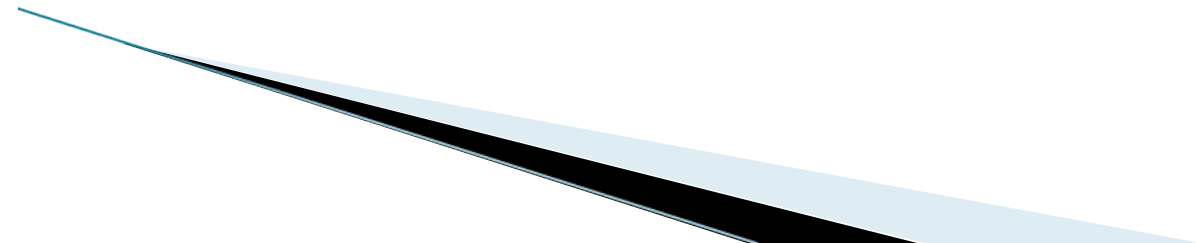
- If a program loads and executes another program and both programs continue concurrently, we have created a new job or process to be multiprogrammed. Often, there is a system call specifically for this purpose (**create process or submit job**).
 - We may also want to terminate a job or process that we created (**terminate process**) if we find that it is incorrect or is no longer needed.
- 

Process Control (System calls)

- If 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 a job or process, including the job's priority, its maximum allowable execution time, and so on (**get process attributes and set process attributes**).
 - 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 to pass (**wait time**); more probably, we will want to wait for a specific event to occur (**wait event**). The jobs or processes should then signal when that event has occurred (**signal event**).
- 

Process Control (System calls)

- Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump memory. This provision is useful for debugging.



File Management (System calls)

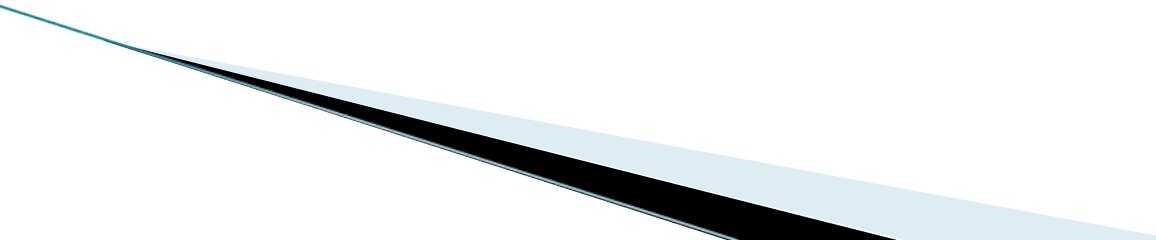
It contains following type of system calls normally.

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

File Management (System calls)

- We first need to be able to **create and delete** files. Either system call requires the name of the file and perhaps some of the file's attributes.
- Once the file is created, we need to **open** it and to use it. We may also **read, write, or reposition** (rewinding or skipping to the end of the file, for example). Finally, we need to **close** the file, indicating that we are no longer using it.

In addition, we need to be able to get/set file **attributes** like file name, type, protection codes, accounting information, and so on.

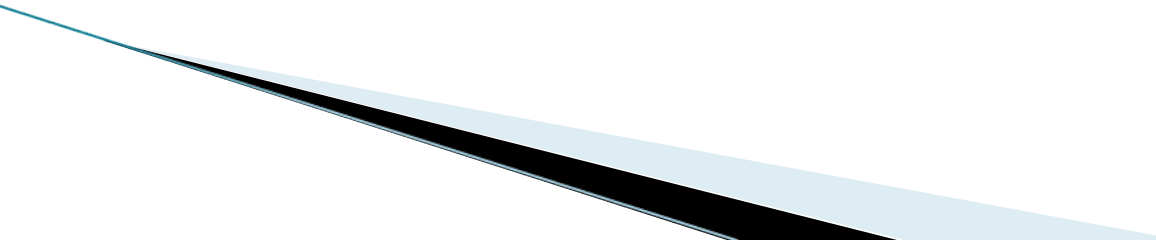


Device Management (System calls)

It contains following type of system calls normally.

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

Device Management (System calls)

- A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.
 - The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, tapes), while others can be thought of as abstract or virtual devices (for example, files).
- 

Device Management (System calls)

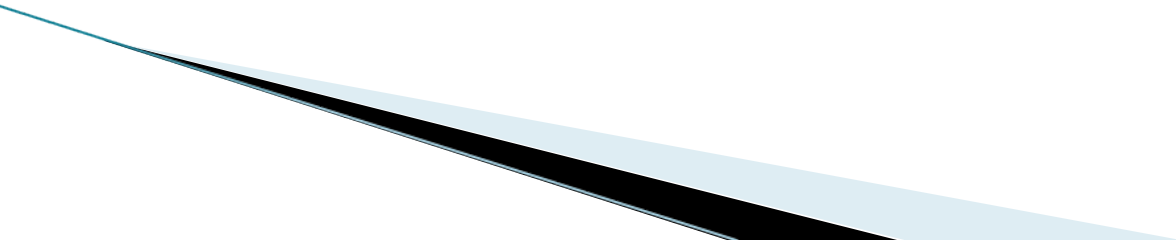
- In case of multiple users of the system, we need to first **request** the device and after we are finished we **release** it.(deadlock?)
- Once the device has been requested (and allocated to us), we can **read, write**.

Information Maintenance (System calls)

It contains following type of system calls normally.

- **Information maintenance**
 - get time or date, set time or date
 - get system data, set system data
 - get process, file, or device attributes
 - set process, file, or device attributes

Information Maintenance (System calls)

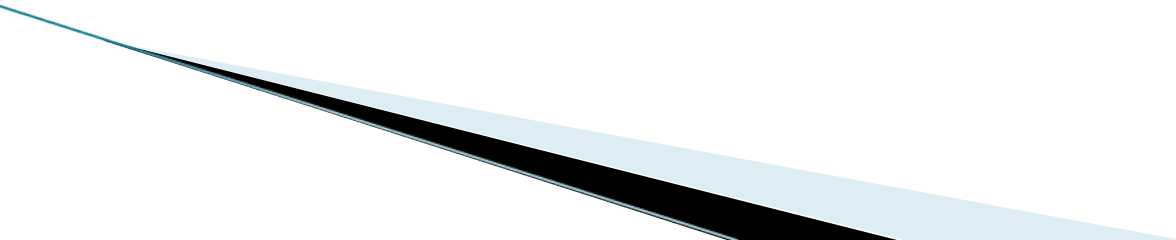
- Many system calls exist simply for the purpose of transferring information between the user program and the operating system.
 - For example, most systems have a system call to return the **current time and date** .
 - Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.
- 

Communication (System calls)

It contains following type of system calls normally.

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

Communication (System calls)

- There are two common models of inter process communication: the **message-passing model** and the **shared-memory model**.
 - In the message-passing model, the communicating processes exchange messages with one another to transfer information.
 - In the shared-memory model, processes use shared memory and use system calls to create and gain access to regions of memory owned by other processes.
- 

Communication (System calls)

- Normally, the OS prevents one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
 - Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for inter computer communication.
 - Shared memory allows maximum speed and convenience of communication, since it can be done at memory speeds when it takes place within a computer
- 