



OPERATING SYSTEMS

Textbook : Operating Systems Concepts by Silberschatz



CS 206 Operating systems

Course Outcomes: After the completion of the course the student will be able to

- CO1: Explain the relevance, structure and functions of Operating Systems in computing devices.
- CO2: Illustrate the concepts of process management and process scheduling mechanisms employed in Operating Systems.
- CO3: Explain process synchronization in Operating Systems and illustrate process synchronization mechanisms using Mutex Locks, Semaphores and Monitors
- CO4: Explain any one method for detection, prevention, avoidance and recovery for managing deadlocks in Operating Systems.
- CO5: Explain the memory management algorithms in Operating Systems.
- CO6: Explain the security aspects and algorithms for file and storage management in Operating Systems.



INTRODUCTION



- Every computer is composed of two basic components:
 - Hardware
 - Software
- **Hardware:** Computer hardware includes the physical parts of a computer, such as central processing unit (CPU), monitor, mouse, keyboard, computer data storage, graphics card, sound card, speakers and motherboard.
- **Software:** is a set of programs which enables the user to solve problems using computer



SOFTWARE

- The Software is set of instructions or programs written to carry out certain task on digital computers.
- Generally, there are two main classifications of software
 - System software
 - Application software



SYSTEM SOFTWARE

- System software: is a collection of system programs which aids the effective execution of general users computational requirements on a computer system.
Eg: OS, linker , loader
- Application software: Application software or programs are used to solve some particular application problems.
E.g.: word processor ,spreadsheet, an accounting application etc.

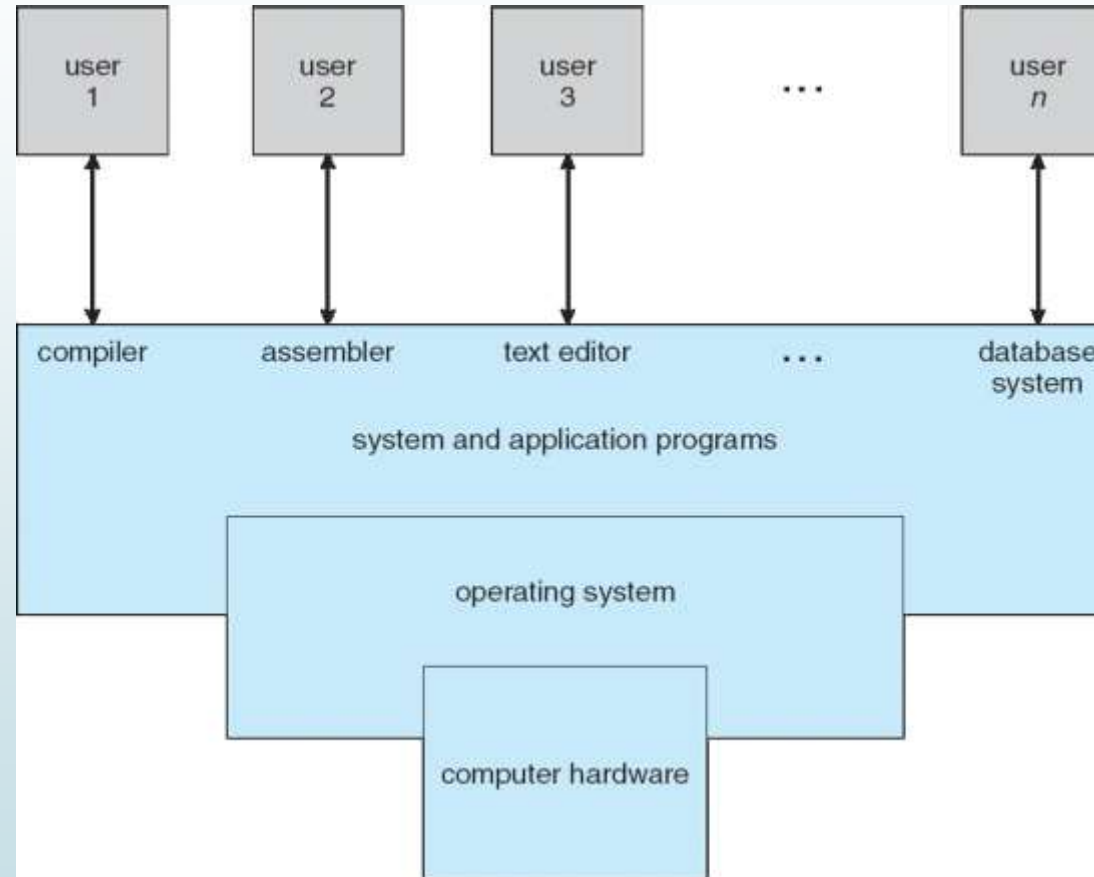


COMPONENTS OF COMPUTER SYSTEMS

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
- **System and Application programs** – define the ways in which the system resources are used to solve the computing problems of the users
 - Word processors, compilers, web browsers, database systems, video games
- **Users**
 - People, machines, other computers

Abstract view of the components of a computer system





OPERATING SYSTEM - A system software

OS takes care of effective and efficient utilization of hardware and software components of the computer system

- Act as an interface between the users and the system.
- Manages computer hardware and software resources
- Provides common services for computer programs.
- Provides an interface which is more user-friendly than the underlying hardware.
- Acts as an extended machine

The operating system masks or hides the details of the hardware from the programmers and general users and provides a convenient interface for using the system. OS is the program that hides the truth about the hardware from the user and presents a nice simple view of named files that can be read and written



OPERATING SYSTEM

- An operating system (OS) is basically a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a crucial component of the system software in a computer system.
- It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.
- The fundamental goal of computer systems is to execute user programs and to make solving user problems easier.
- The common functions of controlling and allocating resources for user programs are then brought together into one piece of software: the operating system.
- Most commonly used Operating systems for personal computers are
Microsoft Windows, macOS, Linux



OPERATING SYSTEM

Basic Functions of Operating System:

The various functions of operating system are as follows:

1. Process Management:

- A program does nothing unless their instructions are executed by a CPU. A process is a program in execution. A time shared user program such as a compiler is a process. A word processing program being run by an individual user on a pc is a process.
- A system task such as sending output to a printer is also a process. A process needs certain resources including CPU time, memory files & I/O devices to accomplish its task.
- These resources are either given to the process when it is created or allocated to it while it is running.



OPERATING SYSTEM



The OS is responsible for the following activities of process management.

- Creating & deleting both user & system processes.
- Suspending & resuming processes.
- Providing mechanism for process synchronization.
- Providing mechanism for process communication.
- Providing mechanism for deadlock handling.



OPERATING SYSTEM

2. Main Memory Management:

The OS is responsible for the following activities in connection with memory management.

- Keeping track of which parts of memory are currently being used & by whom.
- Deciding which processes are to be loaded into memory when memory space becomes available.
- Allocating & deallocating memory space as needed.



OPERATING SYSTEM

3. File Management:

For convenient use of computer system the OS provides a uniform logical view of information storage. The OS abstracts from the physical properties of its storage devices to define a logical storage unit the file. A file is collection of related information defined by its creator. The OS is responsible for the following activities of file management.

- Creating & deleting files.
- Creating & deleting directories.
- Supporting primitives for manipulating files & directories.
- Mapping files into secondary storage.
- Backing up files on non-volatile media.



Operating system

4. I/O System Management:

- OS keeps track of the devices,
- Decides who should get how much time and when the devices
- Allocate the device and initiate the I/O operations
- Reclaim the resource(device)

5. Secondary Storage Management:

- Most modern computer systems are disks as the storage medium to store data & program. The operating system is responsible for the following activities of disk management.
- Free space management.
- Storage allocation.
- Disk scheduling

Because secondary storage is used frequently it must be used efficiently.



OPERATING SYSTEM

Protection or security:

- If a computer system has multi users & allow the concurrent execution of multiple processes then the various processes must be protected from one another's activities.
- For that purpose, mechanisms ensure that files, memory segments, CPU & other resources can be operated on by only those processes that have gained proper authorization from the OS.

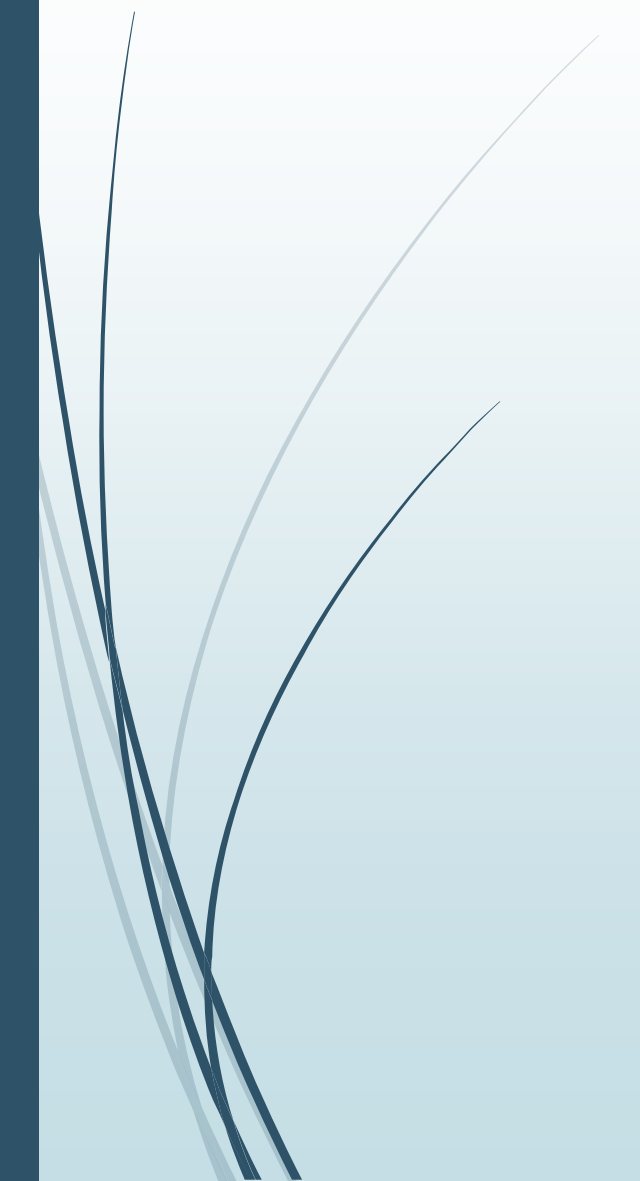


OPERATING SYSTEMS

Module1_Part2



Operating System Structure

- Simple structure
 - Layered
 - Microkernel
 - Modules
- 

Simple structure

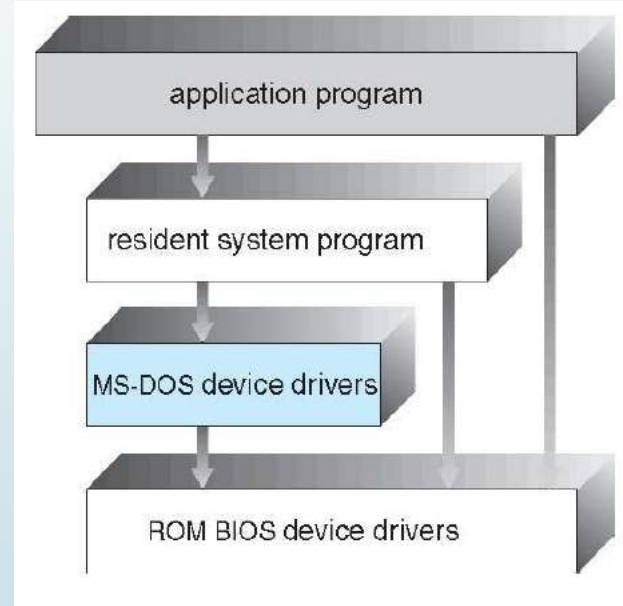
MS-DOS – written to provide the most functionality in the least space

- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

It leaves base hardware access for application programs (no protection)

Enormous amount of functionality to be combined into one level.

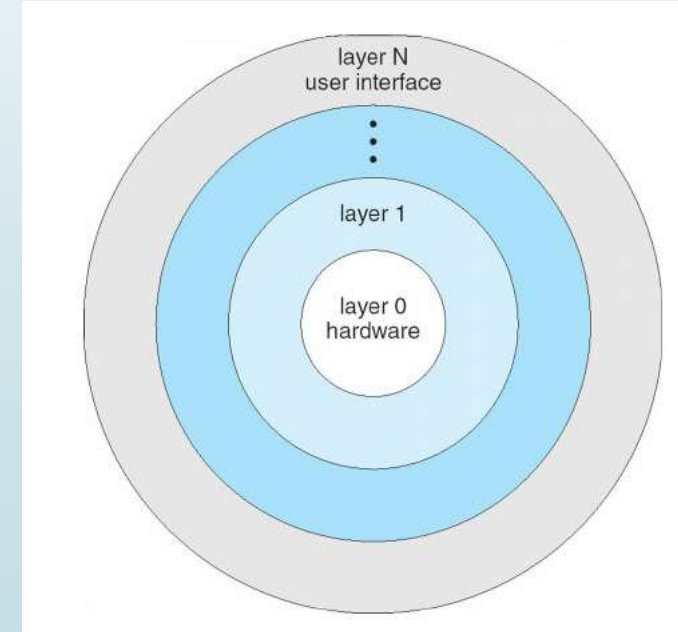
This monolithic structure was difficult to implement and maintain.



Layered Approach

- **Layered approach:** In the layered approach, the OS is broken into a number of layers (levels) each built on top of lower layers. The bottom layer (layer 0) is the hardware & top most layer (layer N) is the user interface.

The layers are selected such that each uses functions (or operations) & services of only lower layer.





Layered Approach

- This approach simplifies debugging & system verification, i.e. the first layer can be debugged without concerning the rest of the system. Once the first layer is debugged, its correct functioning is assumed while the 2nd layer is debugged & so on.
- If an error is found during the debugging of a particular layer, the error must be on that layer because the layers below it are already debugged. Thus the design & implementation of the system are simplified when the system is broken down into layers.
- Each layer is implemented using only operations provided by lower layers. A layer doesn't need to know how these operations are implemented; it only needs to know what these operations do.
- The layer approach was first used in THE operating system. It was defined in six layers.



Layered Approach

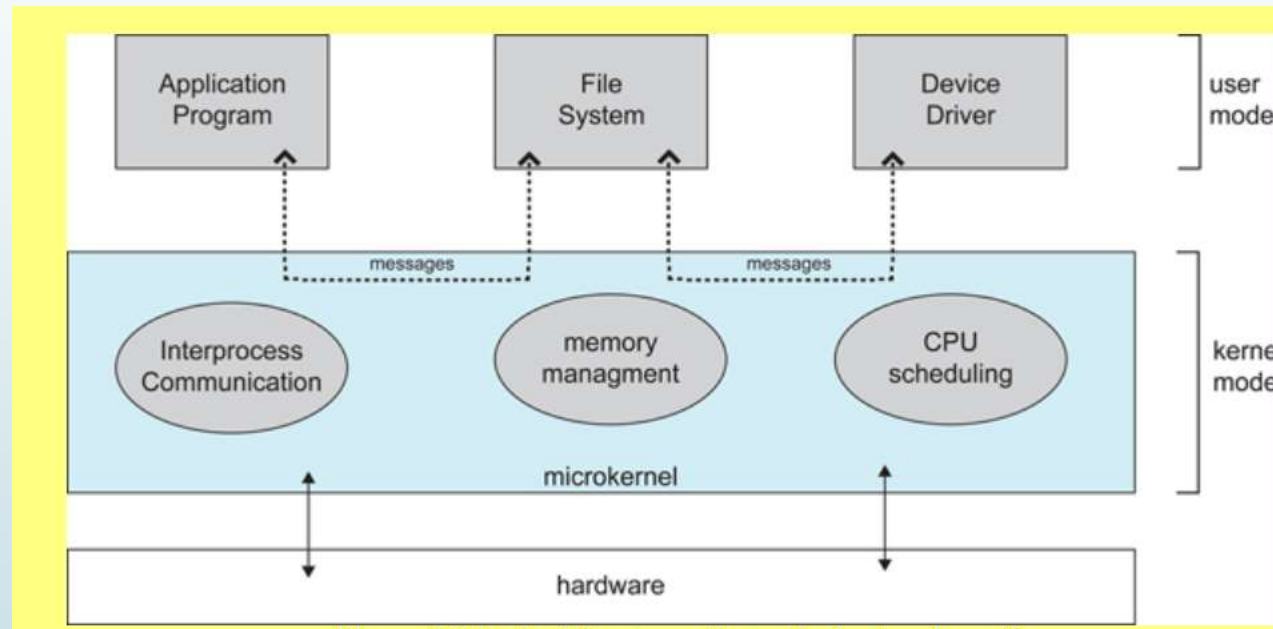
- The main disadvantage of the layered approach is:
- The main difficulty with this approach involves the careful definition of the layers, because a layer can use only those layers below it.
- It is less efficient than a non layered system (Each layer adds overhead to the system call & the net result is a system call that take longer time than on a non layered system).



Microkernels

- The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible.
- Most microkernels provide basic process and memory management, and message passing between other services, and not much more.
- Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.
- System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.
- Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic.

Microkernels



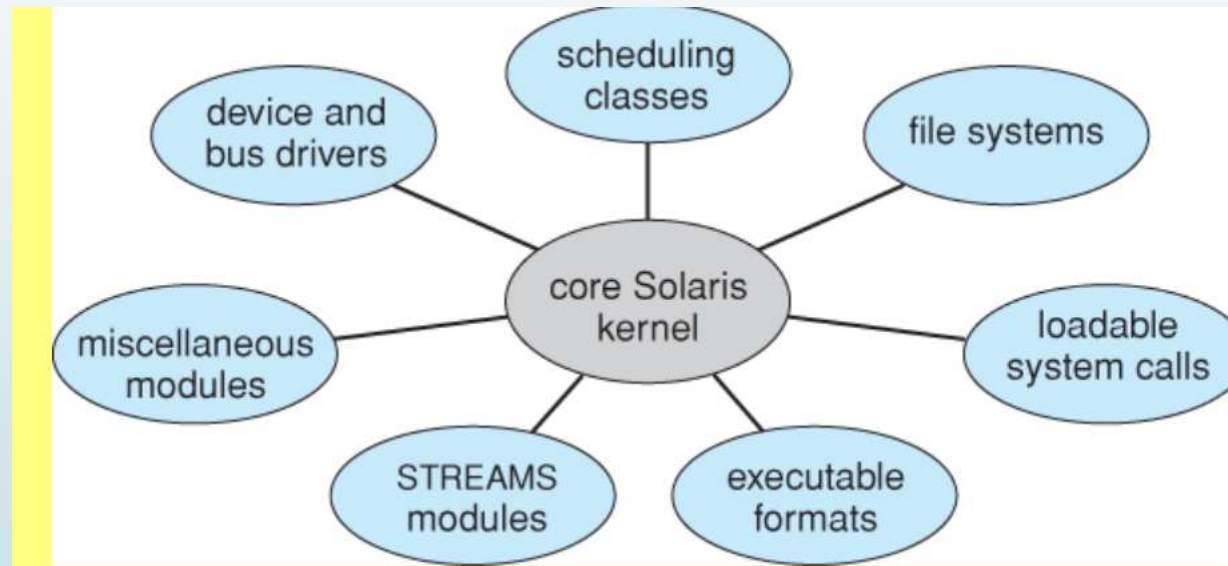
Architecture of a typical microkernel



MODULES

- Modern OS development is object-oriented, with a relatively small core kernel and a set of *modules* which can be linked in dynamically.
- Modules are similar to layers, in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers.
- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.

MODULES



Solaris loadable modules



OPERATING SYSTEMS

Module1_Part3



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**).

Eg: Command-Line (CLI), Graphics User Interface (GUI), touch-screen

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

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.

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.

Error detection – OS needs to be constantly aware of possible errors

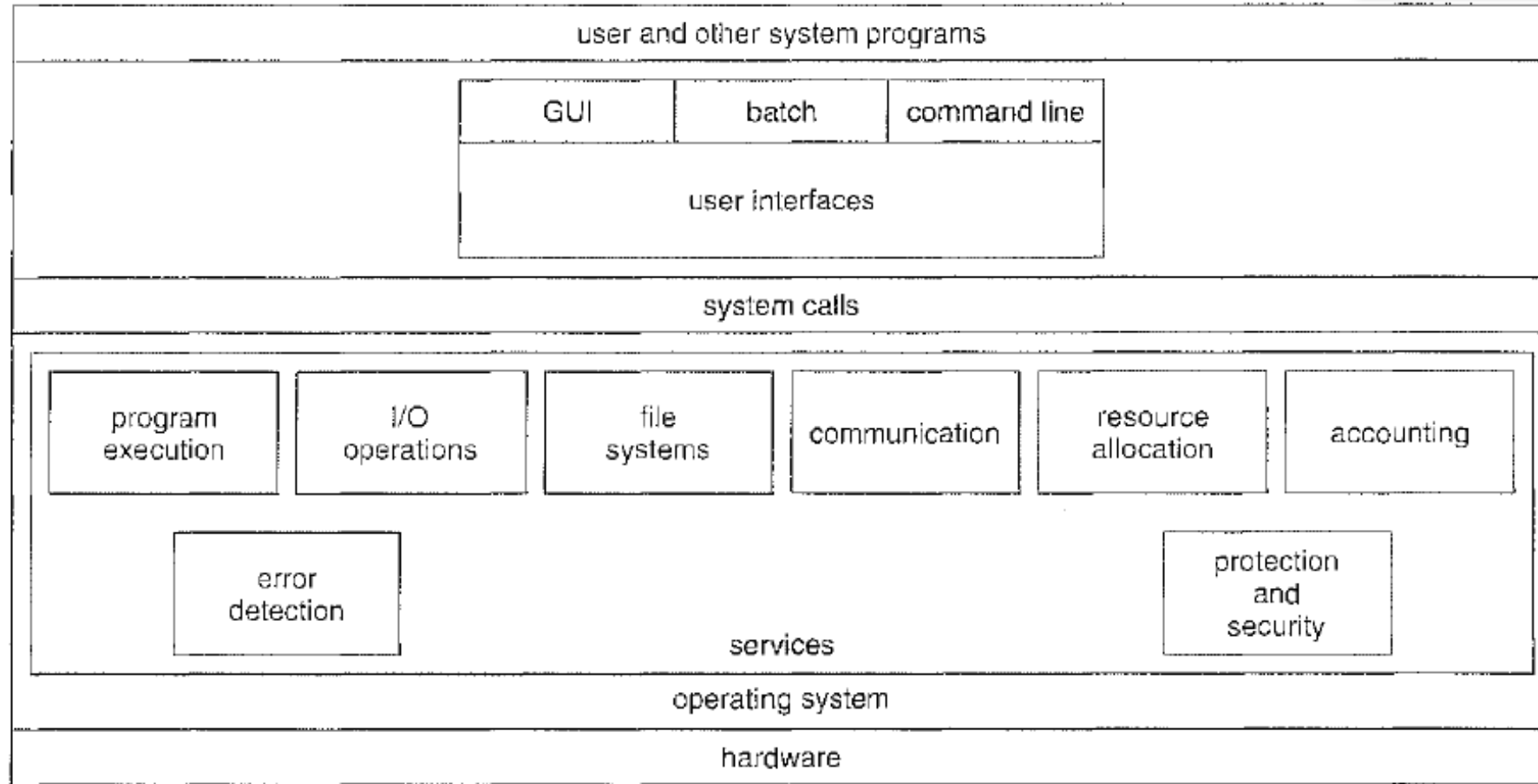
Resource allocation - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them

Accounting - To keep track of which users use how much and what kinds of computer resources

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

Operating system services





CLI -- command line interpreter

Sometimes implemented in kernel, sometimes by systems program

Sometimes multiple flavors implemented – **shells**

Primarily fetches a command from user and executes it

GUI—Graphical user interface

users employ a mouse-based window and-menu system

Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory.

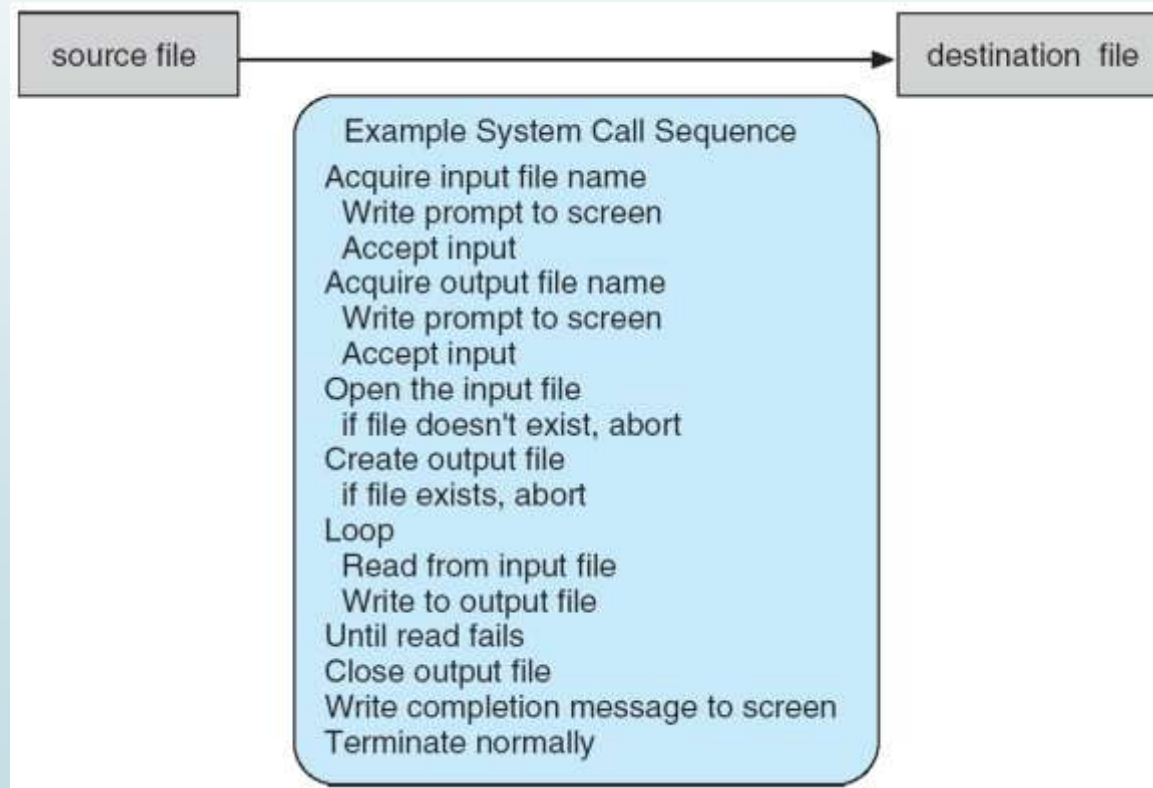


System calls

- A system call is a mechanism that provides the interface between a process and the operating system. A system call is a programmatic way a program requests a service from the kernel.
- To understand how an operating system works, you first need to understand how system calls work.
- System calls are very similar to function calls, which means they accept and work on arguments and return values. The only difference is that system calls enter a kernel, while function calls do not.
- System call offers the services of the operating system to the user programs via API (Application Programming Interface).

System calls

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

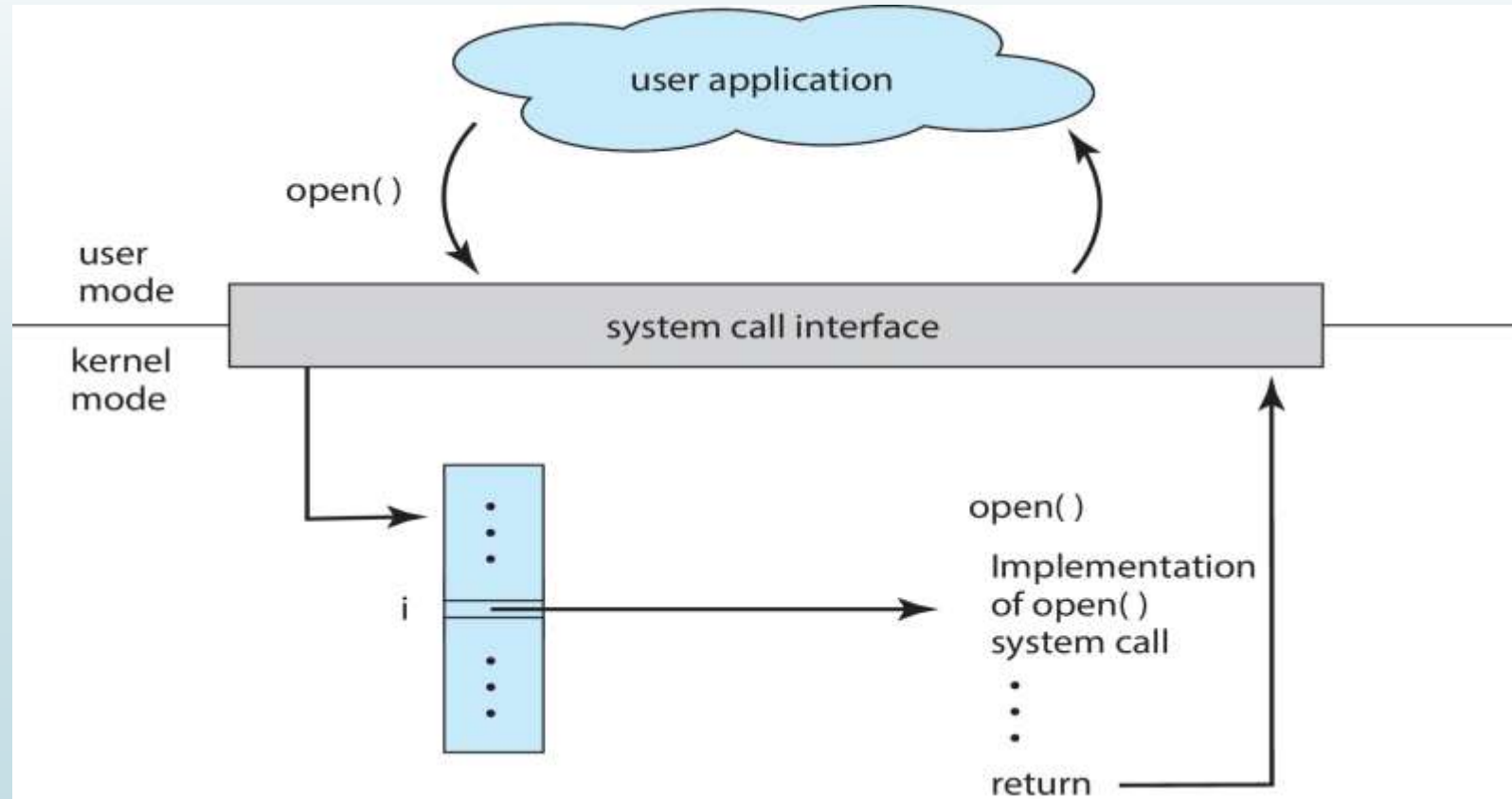




System call implementation

- 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 not know anything 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

API: System Call to Open a File





OPERATING SYSTEMS

Module1_Part4

Textbook : Operating Systems Concepts by Silberschatz



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
 - locks for managing access to shared data between processes



Types of system calls

- 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



Types of System calls

➤ **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
 - From **client** to **server**
- shared-memory model create and gain access to memory regions
- attach and detach remote devices



Types of System calls

- Protection
 - control access to resources
 - get and set permissions
 - allow and deny user access

Examples of Windows and Unix System Calls

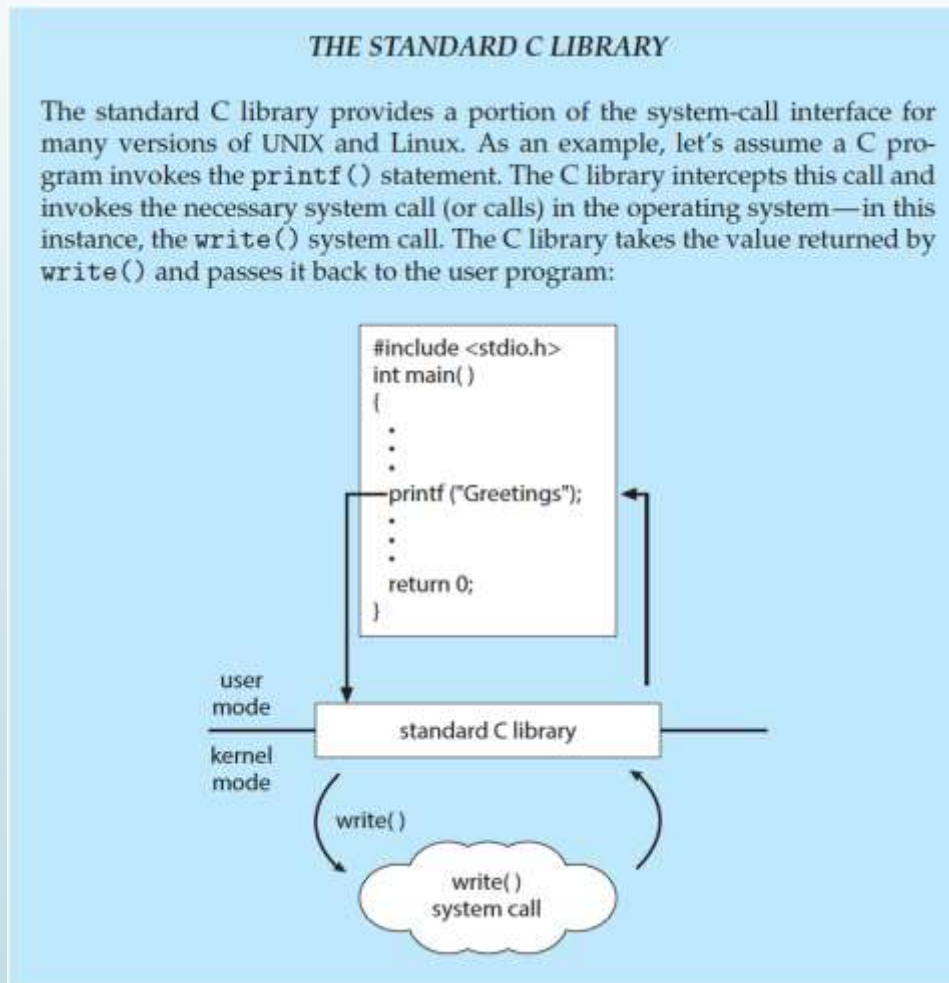
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

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

Standard C Library Example

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



Important System Calls Used in OS

wait()

In some systems, a process needs to wait for another process to complete its execution. This type of situation occurs when a parent process creates a child process, and the execution of the parent process remains suspended until its child process executes.

The suspension of the parent process automatically occurs with a wait() system call. When the child process ends execution, the control moves back to the parent process

fork()

Processes use this system call to create processes that are a copy of themselves. With the help of this system call, parent process creates a child process, and the execution of the parent process will be suspended till the child process executes.

exec()

This system call runs when an executable file in the context of an already running process that replaces the older executable file. However, the original process identifier remains same as a new process is not built, but stack, data etc. are replaced by the new process.



Important System Calls Used in OS

kill():

The `kill()` system call is used by OS to send a termination signal to a process that urges the process to exit.

exit():

The `exit()` system call is used to terminate program execution. Specially in the multi-threaded environment, this call defines that the thread execution is complete. The OS reclaims resources that were used by the process after the use of `exit()` system call.

OPERATING SYSTEMS

Module1_Part5

Textbook : Operating Systems Concepts by Silberschatz



System boot process

- The procedure of starting a computer by loading the kernel is known as *booting* the system.
- On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution.
- When a CPU receives a reset event-for instance, when it is powered up or rebooted -the instruction register is loaded with a predefined memory location, and execution starts there
- . At that location is the initial bootstrap program.
 - Small piece of code – bootstrap loader, BIOS, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
 - Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk

Fork() system call

fork() system call is used to create child processes in a C program.

It takes no arguments and returns a process ID.

After a new child process is created, *both* processes will execute the next instruction following the *fork()* system call.

Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**.

If **fork()** returns a negative value, the creation of a child process was unsuccessful.

fork() returns a zero to the newly created child process.

fork() returns a positive value, the *process ID* of the child process, to the parent.

A process can use function **getpid()** to retrieve the process ID assigned to this process

The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer.

make two identical copies of address spaces, one for the parent and the other for the child.

both processes have identical but separate address spaces



exit() system call

When a process terminates it executes an exit() system call.

The prototype for the exit() call is: #include <stdlib.h> void **exit**(int status);

Macro: *int* EXIT_SUCCESS

This macro can be used with the exit function to indicate successful program completion

opendir() system call

DIR * opendir (const char * dirname)

dirname

The path of the directory to be opened. It can be relative to the current working directory, or an absolute path.

Returns a pointer to DIR structure

readdir() system call

struct dirent *readdir(DIR **dirp*);

The *readdir()* function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*.

A **dirent structure** contains the character pointer d_name, which points to a string that gives the name of a file in the directory.



Exec system call

The `exec()` system call is used to execute a file which is residing in an active process. When `exec()` is called the previous executable file is replaced and new file is executed.

process id will be the same.

Stat system call

Stat system call is a system call in Linux to check the status of a file .The `stat()` system call actually returns file attributes.

the type of the file, the size of the file, when the file was accessed (modified, deleted) that is time stamps, and the path of the file, the user ID and the group ID, etc



Wait() system call

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.

Close() system call

A close system call is a system call used to close a file descriptor by the kernel. For most file systems, a program terminates access to a file in a filesystem using the close system call.

```
int close (int filedes)
```



fork() system call

fork() system call is used to create child processes in a C program.

It takes no arguments and returns a process ID.

After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.

Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**.

If **fork()** returns a negative value, the creation of a child process was unsuccessful.

fork() returns a zero to the newly created child process.

fork() returns a positive value, the ***process ID*** of the child process, to the parent.

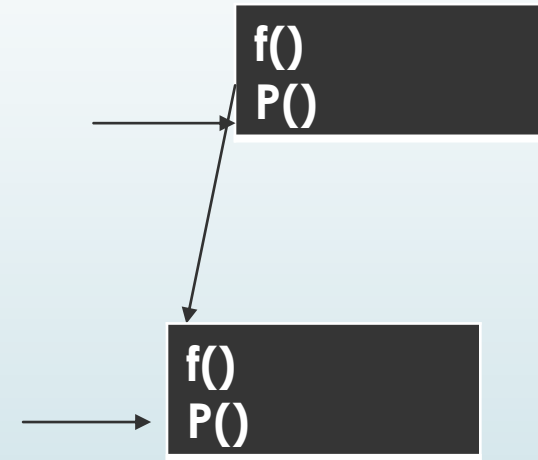
fork() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

output

```
Hello world!
Hello world!
```

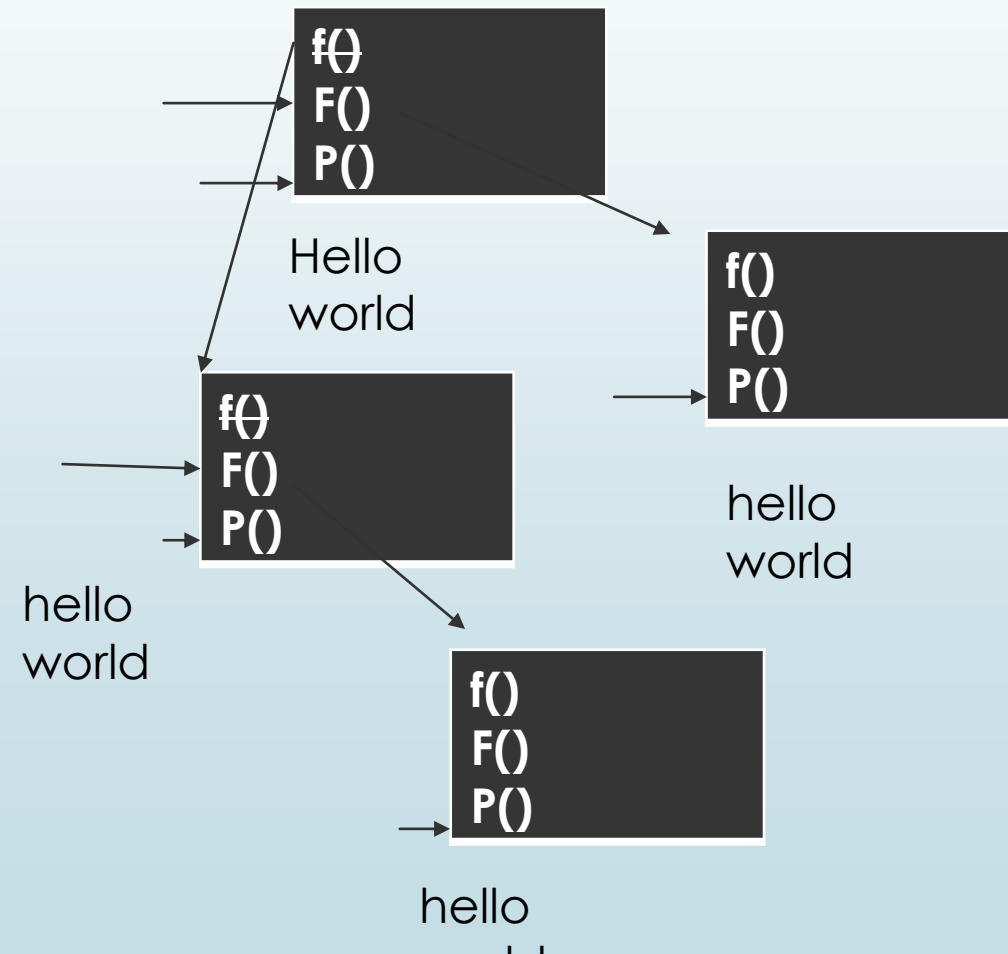


fork() system call

► Consider this code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
main()
{
    fork();
    fork();
    printf("hello world");
}
```

Four times hello world will be printed



getpid() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
Int  main(void)
{
//variable to store calling function's process id
pid_t process_id; // pid_t unsigned integer type
//variable to store parent function's process id
pid_t p_process_id;

//getpid() - will return process id of calling function
process_id = getpid();
//getppid() - will return process id of parent function
p_process_id = getppid();

//printing the process ids
printf("The process id: %d\n",process_id);
printf("The process id of parent function: %d\n",p_process_id);
return 0;
}
```

Output

The process id: 31120

The process id of parent function: 31119



Exec system call

The `exec()` system call is used to execute a file which is residing in an active process. When `exec()` is called the previous executable file is replaced and new file is executed.

process id will be the same.

Exec() system call

- There are two programs ex1.c ex2.c

Ex1.c

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
Int main(int argc,char *argv[])
```

```
{
```

```
printf("Pid of ex1.c=%d\n",getpid());
```

```
Char *args[] ={hello",NULL};
```

```
execv("./ex2",args);
```

```
printf("Back to Ex1.c");
```


```
Return 0;
```

```
}
```



► Ex2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
Int main(int argc,char *argv[])
{
printf("We are in ex2.c\n");
printf("Pid of ex2.c=%d\n",getpid());
Return 0;
}
```


- 
- Compile these two programs
 - `gcc ex1.c -o ex1`
 - `gcc ex2.c -o ex2`
 - Run the first program `./ex1`

 - Pid of ex1.c=5962
 - We are in ex2.c
 - Pid of ex2.c=5962