Operating System

Module - 1

Introduction to system software.

1.1

1) Definitions:

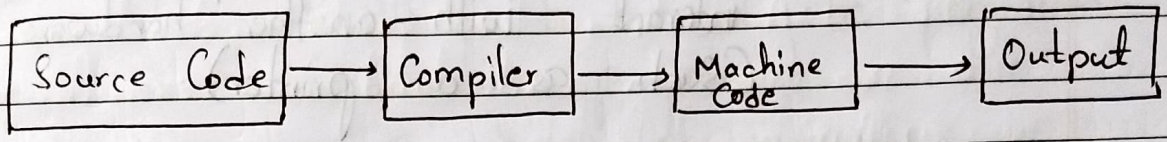
a) Assembler: An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations.
(it converts ~~code~~ ~~to~~ assembly language to bits)

b) Loader: A major component of an OS that ensures all necessary programs & libraries are loaded. It places the libraries & programs in the main memory in order to prepare them for execution.

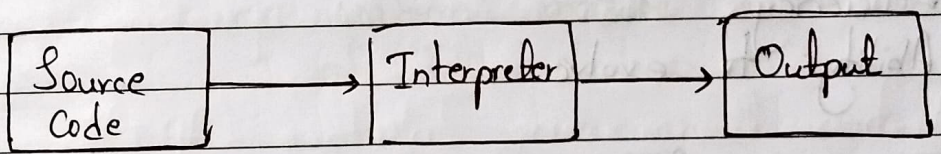
c) Linker: (or link editor) a computer system program that takes one or more object files (a computer file containing output of an assembler or compiler) and combines them into a single executable file.

d) Macro-Processors: A set of programs used to execute certain functions of the operating system. It is not integrated with a particular software.

e) Compilers: A special program that translates a programming language's source code into machine code.



f) Interpreter: A program that directly executes the instructions in a high-level language, without converting it into machine code.



The interpreter could parse the high-level source code & then perform the commands directly, which is done line-by-line. Or it could transform the high-level source code into an intermediate language that it then executes.

g) Operating Systems: ~~int~~ program that interface the machine (hardware) with the application programs. The operating system is designed in such a way that it can manage the overall

resources and operations of the computer. Operating system is a fully integrated set of specialized programs that handle all the operations of the computer.

h) Device Drivers (Software Drivers): A device driver is a particular form of software application that allows one hardware device (such as personal computer) to interact with another hardware device (such as a printer).

2) Operating system Objectives and functions.

An OS has 3 objectives:

a) Convenience
b) Efficiency
c) Ability to evolve

a) Convenience: An OS makes a computer more convenient to use.

If one were told to develop an application program as a set of machine instructions that are completely responsible for controlling the computer hardware, they would face overwhelming complexity.

Thus, to ~~easy~~ ease this chore, a set of system programs is provided. The most important system programs ~~are~~ make ~~comprize~~ the OS.

The OS masks the details of the hardware from the programmer and provides an interface.

→ The OS provides services in the following areas:

i) Program Development: The OS provides facilities to assist the programmer in creating programs.

ii) Program Execution: The OS handles scheduling duties (like resource loading) for the user when executing the program.

iii) Access to I/O devices: The OS provides a uniform interface for all different types of I/O devices.

iv) Controlled access to files: The OS may provide mechanisms to control access to files in a system with multiple users.

v) System access: For public systems, the OS controls access to the system as a whole.

vi) Error detection and response: The OS must provide a response that clears any error condition with the least impact on running programs. Error can be a memory error or a software error (like divide by 0).

vii) Accounting: A good OS will collect usage statistics for various resources and monitor performance parameters (eg: response time)

→ Three key interfaces in a typical computer system:

i) Instruction Set Architecture (ISA):
The ISA defines the specific instructions that a processor can understand and execute, eg: arithmetic calculations.

ii) Application Binary Interface (ABI):
The ABI is a set of rules and conventions that govern how binary objects, such as executables files and libraries, interact with each other & the operating system.

iii) ~~Application Programming Interface (API):~~
~~The API gives the program access to hardware resources~~

iii) Application Programming Interface (API):
The API abstracts away the hardware details and offers a standardized interface that can be accessed by various means, including high level language library calls.

CLibrary Call: A request to a programming library to use a certain function.

b) Efficiency : An OS allows the computer system resources to be used in an efficient manner.

A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The OS is responsible for managing these resources.

The OS, thus, has control over all resources, however it itself is a program executed by the processor.

- The OS functions in the same way as an ordinary computer software, that is, it is a program executed by the processor.
- The key difference is that the OS directs the processor in the use of system resources.
- In order to use any other system program, the processor has to cease executing the OS program and execute other programs.
- Thus, the OS stops running for the processor to do some useful work. Then OS program is run again by the processor for the next work.
- The OS must determine how much processor time is to be devoted for the execution of a program.

c) Ability to evolve: An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system functions.

A major OS will evolve over time for a number of reasons:

i) Hardware updates and new types of hardware.

ii) New Services: In response to user demand or in response to the needs of system managers, the OS expands to offer new services.

iii) Fixes: If any fault is found in the OS, fixes should be introduced.

1.2

The evolution of Operating Systems.

1) Serial Processing:

- In earlier computer models with serial processing, the programmer interacted directly with the computer hardware, there was no OS.
- These computers were run using display lights, toggle switches, input device and a printer.
- Programs were loaded in using the input device.

- If an error halted the program, the error message was indicated by the lights.
- If the program was successfully executed, the output was printed by the printer.

This system presented 2 main problems:

a) Scheduling: Hardcopy sign-up sheets were used to reserve computer time.
This could lead up to 2 problems:
i) User, after making a reservation for 1 hour, could be done with the job in 45 minutes. The 15 minutes of processor time would be wasted.

ii) If user's job isn't completed in the reserved time, the job will be forced to stop.

b) Setup time: Setting up the system for running took a lot of setting up that involved mounting or dismounting tapes.
If an error occured, the user would have to start again from the start.

This mode was called serial processing because users had to accept access the computer in series.

2) Simple Batch Systems

The idea behind the simple batch system was the use of a software called 'monitor'.

- The user no longer has direct access to the processor. Instead, the user submits the job on a tape to a computer ~~tape~~ operator.
- The computer operator batches the jobs that they receive ~~and~~ together sequentially & places the batch in an input device.
- The monitor accesses the input device. Each program returns back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.

There are 2 point of views in this scheme:

a) Monitor POV: The monitor controls the sequence of events. Most of the main memory is in the main memory, called resident monitor. The rest of the monitor contains often used functions which can be loaded in the resident memory depending on user's program's needs.

The monitor reads all the jobs from the input device one at a time and sends them

for execution by the processor. The control is passed to the processor.
After the job is completed, the control is passed to the monitor again, which sends the next job for execution.

The result for each job is printed.

b) Processor POV: Initially the processor executes the instruction from the resident monitor. It is instructed to read a job. Once the job has been read in, the monitor instructs the processor to execute the instruction in the program (job) that it has read.
After all instructions from the program are run or ~~if~~ an error occurs, the processor executes the next instruction given in the monitor.

Thus, 'control is passed to a job' simply means that the processor is instructed (by monitor) to execute the program's instructions.

'Control is passed to the monitor' means that the processor is running the instructions from the main memory after implementation or failure of a program.

- The monitor is a computer program. Certain hard-ware features are desirable for this scheme:

a) Memory Protection: When a job is executing, it should not alter the data in main memory.

b) Timer: A single job should not consume most of the processor time. A timer is set at the beginning of each job. if that timer is expires, the job is stopped.

c) Privileged Instructions: Certain instructions are considered privileged instructions and can only be executed by the monitor. Eg; I/O instructions. If a currently executing job requires I/O operation, then control is passed to the monitor to perform the operation for the job.

d) Interrupts: This features gives the OS more flexibility in giving & gaining control.

3) Multiprogrammed Batch Systems

a) • The limitations of simple batch system is that a lot of processor time is wasted in waiting for the I/O device to read & write data.

• I/O devices are degrees slower that processor. Thus the processer ends up completing the operation quickly but has to wait for the

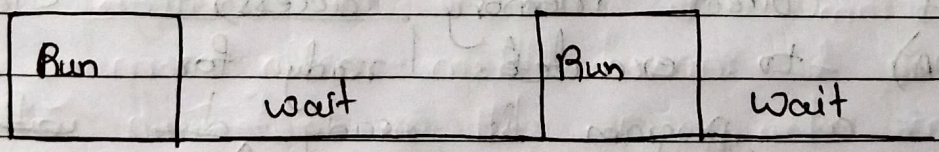
I/O device to bring in the next operation.

* To combat this, we ~~created~~ allotted more space in the main memory to user programs. Instead of keeping the OS and 1 user program only (in simple batch systems), more user programs can be stored in the main memory (along with the OS).

* Thus, after the processor is done with a task in user program1, it can directly start working on user program 2 because it is already loaded onto the main memory.
This is called multiprogramming or multitasking approach.

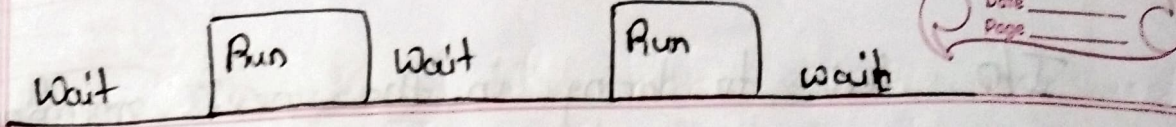
Example:

a) There are 3 jobs that the processor has to execute. JOB1 has heavy computation required & low I/O operations like disk access & printer access:

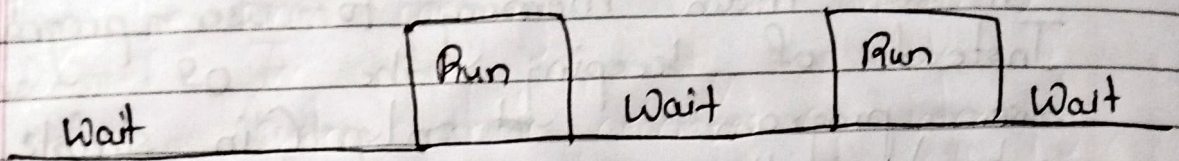


b) JOB2 and JOB3 require continuous disk & printer access & have lighter compution required (meaning lesser processor time required):

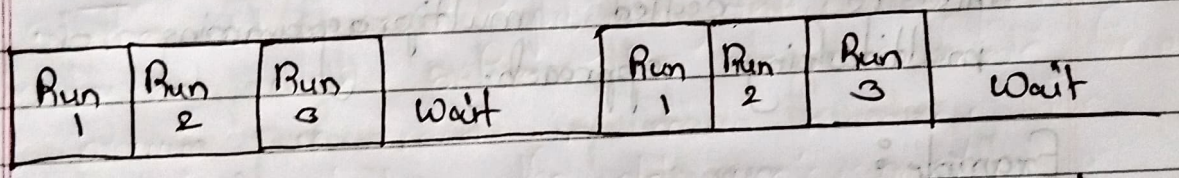
Job 2:



Job 3:



c) If the above JOBs were executed sequentially like in simple batch system, it would take a lot of time. Instead, in multiprogrammed systems, the JOBs will be done like this:



This method saves a lot of processor time.

* Hardware Requirement of Multiprogrammed systems:

This system requires I/O interrupts or DMA (Direct memory access) (Learned in COA) to execute. Cleared in

After a program is ~~ready~~ ready for I/O operation, it sends an ~~en~~ interrupt to the processor, which stops the ~~task~~ it was doing (saves the progress) ~~on the~~ program ~~does~~ the next operation required by ~~the program that interrupted it~~ and calls for the next program to execute. So basically, switching between different user

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
Total	31 μs

$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure 2.4 **System Utilization Example**

4) Time-Sharing Systems:

- All systems discussed so far don't have interactive models. A Job is given to the processor & it then exec executed executes the Job.
- For many jobs, an interactive mode is necessary, like a transaction.
- Multiprogramming systems can also be used to ~~had~~ handle interactive jobs by using a technique called time sharing.
- Time Sharing: The processor time is shared by multiple users (instead of simple job cards). These users access the system simultaneously through a terminal, where they give the next task to do.
- The OS interleaves the ~~exec~~ execution of each user program in short bursts of computation (meaning its executing instructions from many user programs in short bursts).
- This makes the system appear to be as fast as a personal computer, because the tasks are carried out sequentially for all programs one by one.
- The first model that used time-sharing system was CTSS. It had 32000 36-bit words

main memory, out of which 5000 were used by the OS. The job given to the processor then consumes the next bits available in the main memory (5000 onwards).

- As system clock generates an interrupt every 0.2 seconds to stop the processor from executing the current task and start the next job. This is called time slicing.

- It is able to stop the processor by giving control ~~by~~ to the OS, then OS, with the regained control, assigns the processor with some other user.

- The old user's program status is written to the disk before the new user's program is read in main memory for execution. When the old user's program execution is to be resumed, the data is read from the disk back into the main memory.

- It's not compulsory that the old user program data has to be written to the disk & removed from main memory (because time sharing is still an extension of multiprogramming). If there is enough space in the main memory, then the old user program data need not be written back to the disk.

Note:

Time sharing is referred to as 'multitasking' by some professors and 'multitasking' does not also mean 'multiprogramming' in their definition. So in exam, if multitasking is asked, know that they are asking for time sharing.

- One of the first time sharing systems developed was CTSS (Compatible time sharing system). As mentioned before, it had 32000 36-bit words main memory. If it had 4 jobs to do with
  Job 1 — 15000 36-bit words required
  Job 2 — 20000 "
  Job 3 — 5000 "
  Job 4 — 10000 "
  Then, the operation would look like:

{{ Image #3, CTSS operation }}

- Time sharing & multitasking introduced new problems because multiple jobs are now in main memory. Thus, they need to be protected from modifying data in each other.

Table 2.3 Batch Multiprogramming versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Preemptive? Non - Preemptive Preemptive

preempting means forestalling an action. multiprogramming is non-preemptive because it does not allow a user-program thats being executed to be preempted if another program is ready for operation. time-sharing allows that, thus it is preemptive. if a user program wants to be executed in multiprogramming system, then it needs to wait in queue.

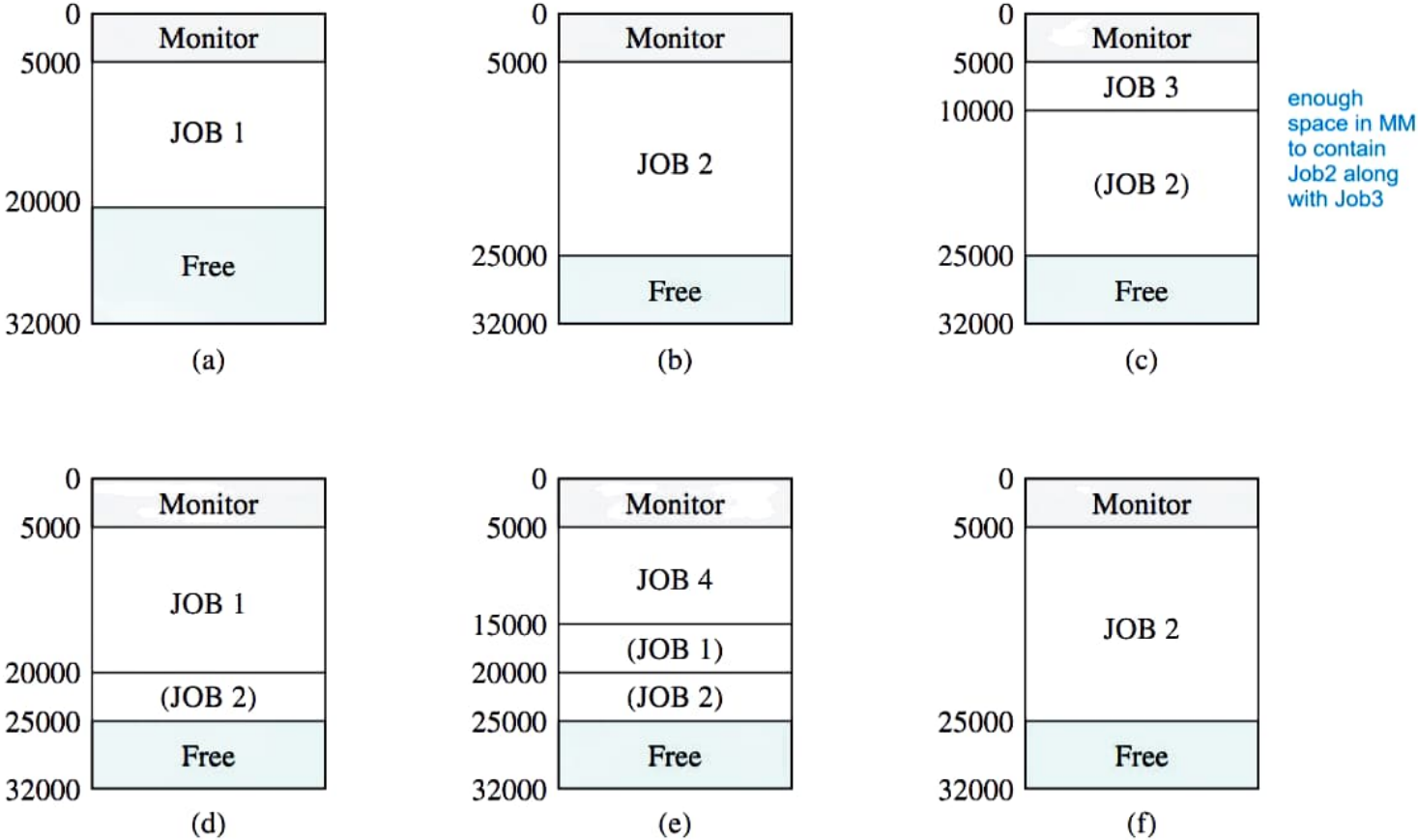


Figure 2.7 CTSS Operation

Definitions:

- Kernel: The central component of an OS, responsible for managing the system's resources and providing essential services to enable communication between hardware and software. It acts as a bridge between applications & the underlying hardware, making it possible for applications to interact with hardware without having to understand the low-level details of the hardware's implementation.

- Process: When a program is under execution, it is called a process. A program is a physical entity & a process is its executable form.

- Threads: A segment of a process is called a thread.
  Eg: Process - Opening the browser
  Thread - opening a tab.
  A thread may or may not be created under a process.

What is a multiprocessor?

In a multiprocessor system, more than 1 processors are used in the system. These processors are used for parallel processing. Multiprocessor system employs a distributed approach, more than 1 processor do the subtasks of a given task & a single processor does not perform

the tasks the complete task.

1) Symmetric Multiprocessors (SMP):

An SMP is a single computer system with the following characteristics:

a) there are 2 or more similar processors of comparable capability.

b) Each processor is interconnected with the system & each other such that the memory access time is similar for each processor.

c) Shared access to I/O devices for all processors.

d) All processors can perform the same functions (hence, are symmetric).

e) The system is controlled by an integrated operating system that provides interaction between processors & their programs.

* What is a cluster?
Ans) A computer cluster is a set of computers that work together so that they can be viewed as a single system.

* What is the difference between an SMP and a cluster?
Ans) In an SMP, the degree of interaction is higher than in a cluster. In a cluster, 1 processor can send one complete file to another,

While a processor in SMP can communicate by sending individual data elements as well.

2) # Advantages of SMP:

a) Performance: If a work can be divided into smaller work, an SMP will be more efficient than uniprocessor.

b) Availability: Since all processors can perform the same function in an SMP, if one processor dies, the program is still not halted as the job can be taken up by another processor.

c) Incremental growth: Performance can be improved by adding a processor.

d) Scaling: Depending on the number of systems, the price can be scaled down or up easily.

3) Organization: • Each processor has shared access to I/O devices and main memory.

• Each processor has a dedicated cache memory. Precaution has to be taken to avoid cache coherence problem. (local cache has updated the value of X, but it hasn't changed globally).

{{ SMP organization }}

4) Multicore Computers

A multicore computer (or chip multiprocessor) combines 2 or more processors (or cores) on a single chip.
These processors have all the components in each processor (registers, ALU, etc.).
They also have caches of multiple level.

* Advantage of multicore?
Ans) Over the years it was observed that ~~increasing~~ the decreasing the size of the components, thereby decreasing the ~~dis~~ physical distance between the components helped improve the performance of the system.

This decrease in size is the result of increased complexity of the processor.
The ideal performance was observed when multiple processors & cache are put in a single chip.

5) OS design Considerations for Multiprocessors

In an SMP system the kernel gives the program execution to any processor.
The kernel can be designed to support multiple processes or multiple threads to execute in parallel, thus portions of the kernel can be executing in parallel.
This complicates the OS.

The OS designer must design the OS to deal with the problems caused by parallel processing (sharing resources, accessing devices, etc.)

To the user the SMP system should appear to be same as a multiprogramming uniprocessor system in regards to computer resources used & program output received.

The very dumb user may give an application to run when all the processors are already busy.

Thus, the OS has to have all the features of multiprogramming as well as some additional features.

a) Simultaneous concurrent processes or threads:

Kernel routines should re-entrant, meaning a kernel code can be implemented parallely by multiple processors by without corrupting the data.

kernel tables (tables) & management structures must be managed properly.

b) Scheduling: (Point will be understood after module 2). Processors have their own scheduling which may disturb the

scheduling protocol policy used by the kernel to assure data or structure corruption is avoided. Thus, the OS must enforce stricter rules to avoid corruption.

c) Synchronization: When multiple processes are running with shared resources, care has to be taken to ensure synchronization, meaning the events should be mutually exclusive (giving the same output that it would've given if the events were serial) or and should have proper event ordering (to avoid dirty read or write, like in transactions using precedence graph).

d) Memory Management: When multiple processors are accessing the same pages, the page replacement must be coordinated so that no page can be accessed with its old content.

e) Reliability and fault tolerance: If a processor fails the scheduler & OS must recognise that and reconstruct the management tables accordingly.

Many All OS considerations of uniprocessor multiprogramming systems are included in multiprocessor

systems as well.

6) OS considerations of Multicore systems.

- All considerations discussed so far for SMP are included in the consideration for multicore system, but there are some additional concerns.
- Many-Core systems: The no. of cores (processors) in a chip keep increasing to the point that a 'multi' core system is called 'many' core system.
- Main concern is to use the on chip resources ~~carefully~~ efficiently while maintaining good parallelism.
- There are 3 levels of parallelism that ~~can be~~ is implemented in multicore system:

a) Hardware Parallelism – instruction level parallelism, meaning instructions in a processor can be executed parallely.

b) Multiprogramming & multithreaded ~~execut~~ execution – Each processor implements multiprogramming or multithreading to run multiple threads or processes within the processor simultaneously.

This is more complicated than instruction level multi parallelism because in ILP,

instructions of a single program are run in the processor, while multiple threads or processes need to also access the resources parallely & coordinate amongst themselves to do the task.

c) Concurrent processes or threads across multiple cores: A single application can be run concurrently as multiple processes or threads in the cores.

- There are ~~2~~ many ~~major~~ approaches to improve parallelism, 2 of them are:

a) Parallelism within Applications: Most applications can be divided into categories that can be independently executed parallely.

- It is the application's developer's responsibility to decide how the application can be split up.

- The OS can execute the tasks parallely and efficiently by allocating resources among the parallel tasks defined by the developer.

- The Grand Central Dispatch (GCD) that comes as multicore support system in MAC OS makes the split up tasks run parallely ~~in~~ easy.

- GCD uses thread pool mechanism, which is used

to efficiently manage & ~~rec~~ rewc a group or pool of ~~thre~~ threads for executing tasks.

- GCD also supports anonymous functions to define & specify tasks for concurrent execution.
  This makes the developer's task easier by improving the simplicity & readability of the code.

- Apple's slogan for GCD "islands of serialization in a sea of concurrency" explains how the application developer is saved by the island from the sea of concurrency.

  ~~Apple~~ The developer can split up the ~~task~~ application into tasks even if they are interdependent, reducing the problems brought on by concurrency, like deadlocks.

b) Virtual Machine approach:

- The OS's processor's model discussed so far had two distinct types, the kernel mode & the user mode.

- The user programs that did not require hardware hardware resources and the programs running in the kernel model needed those resources.

- This abstracted the processor into two parts processors. The OS had to switch control frequently between these processors depending on the program.

- The overhead of switching between the 2 processors would be very high for a many-core system, they the virtual machine approach is used.

- In the virtual machine approach, the distinction between user mode & kernel

mode is discarded and the responsibility of resource management is given to the programs themselves.

- Thus, the programs that do required hardware resources ~~will that~~ should themselves manage those resources.

- Now, the OS only has to assign a program to a processor. The remaining tasks of using resources required will be done by the program itself!
  Thus, the OS acts like a hypervisor.

- The developer should make the application so that it can access the resources it needs.

1.5 System Calls

1) What are system calls and how are they made?

- System call is an interface provided by an OS that allows user level processes to access services from the kernel & hardware and software resources.
- System calls are written in C or C++ and sometimes assembly language, for tasks requiring direct hardware access.
- The system calls are used like in the example given below:

Task: Copy content from 1 file & paste it on another.

System calls will be made to:

a) Write prompt to the user to give the source file name.

b) Write prompt to get the distination file name.

c) Read from the input the source & destination file names.

d) Opening the source file. and the destination file.

e) Printing errors in step d, eg: if source or destination files do not exist.

f) Terminate abnormally if errors were found.

g) If no errors are found, reading from the input file & writing to the output file requires system calls.

h) If ~~sy~~ destination file name already exists, a sending prompt to the user to either abort or replace existing file.

i) If the disk space is over before the content can be written, ~~showing~~ sending an error to user, then terminating abnormally.

j) In case of no errors, closing both files, writing success message to console and terminating normally.

Thus, even simplest operations require lots of system calls.
Programmers don't have to see this level of detail because of APIs (Application Programming Interface).

The API specifies a set of functions available to a ~~us~~ programmer. The API invokes the system calls required by the application

Programmer.

Using an API has following advantages for the application programmer:

a) The program written using a certain API can be expected to run on any device that supports that API.

b) Working with actual system calls is difficult, ~~to work with~~ the API interfaces the details.

2) System Call interface:

- The system call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.

- Each type of system call is ~~num~~ numbered and stored in a table.

- The system call interface ~~calls~~ invokes the required system call and returns the status of the call along with any return value.

- The application programmer need only know the functions provided by the API, not the details of each system call.

3) Parameters in a system call:

Sometimes, extra information may be needed to ~~send~~ be sent along with the system call.

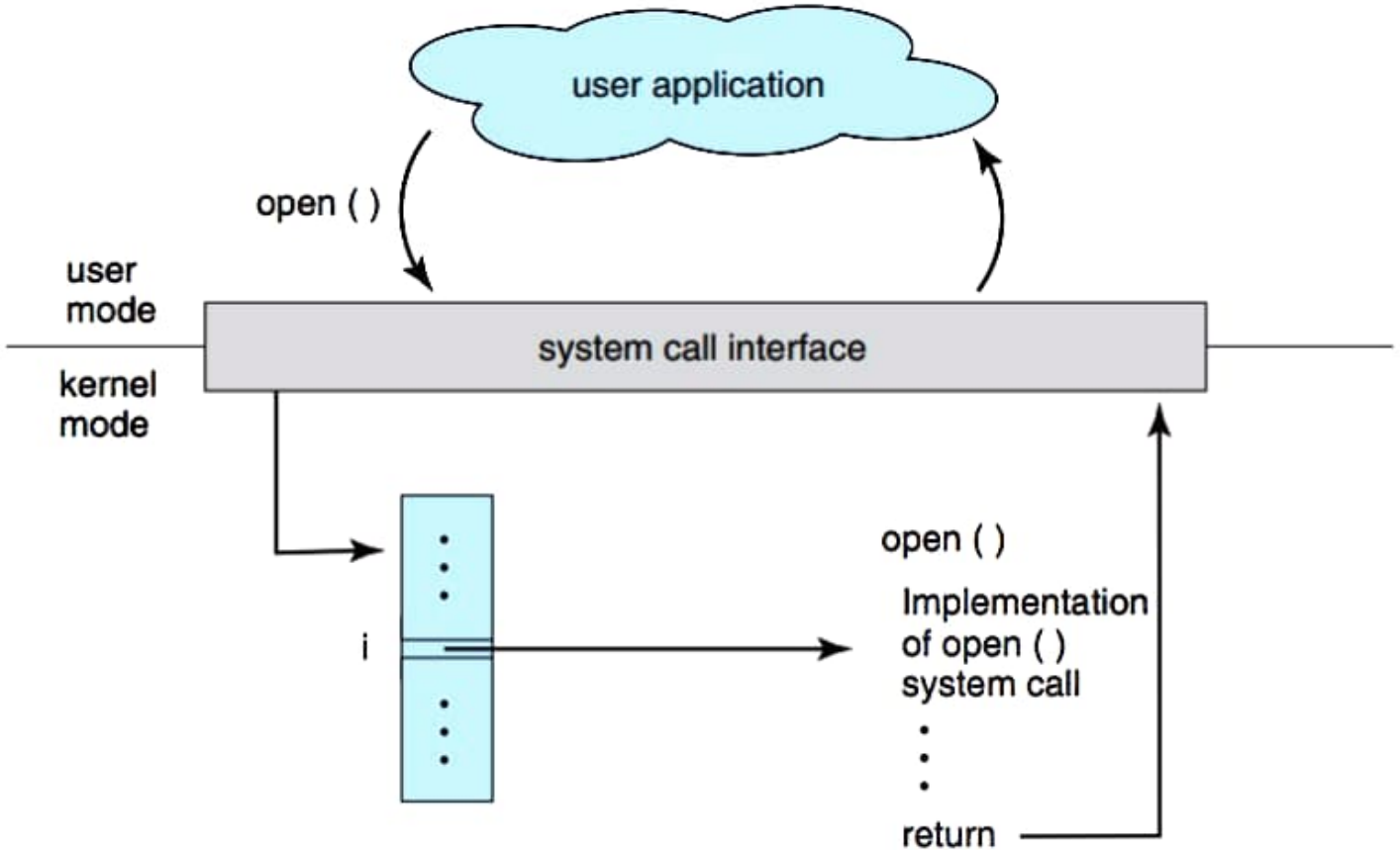


Figure 2.6 The handling of a user application invoking the `open()` system call.

Eg: Copy the ~~f~~ a particular portion of the file onto the other file. The extra parameters needed to be known are the starting & the ending part of the portion.

These parameters are passed along with the system call as the context.

There are 3 ways of sending these parameters to the OS:

a) Each parameter is stored in a register and sent.

b) ~~If the no. of registers are not enough, then the parameters are arranged in a table that is kept inside a register.~~

b) If the no. of registers is not enough, the parameters are stored in a block or table in memory & the address of the block is stored in the register. Approach used by Linux.

c) Parameters can be pushed in a stack by the program & the OS pops the parameters out. This approach does not limit the number or size of the parameters, because no registers are used.
~~The parameters are stored in a block~~

{{ Passing of parameters as a table }}

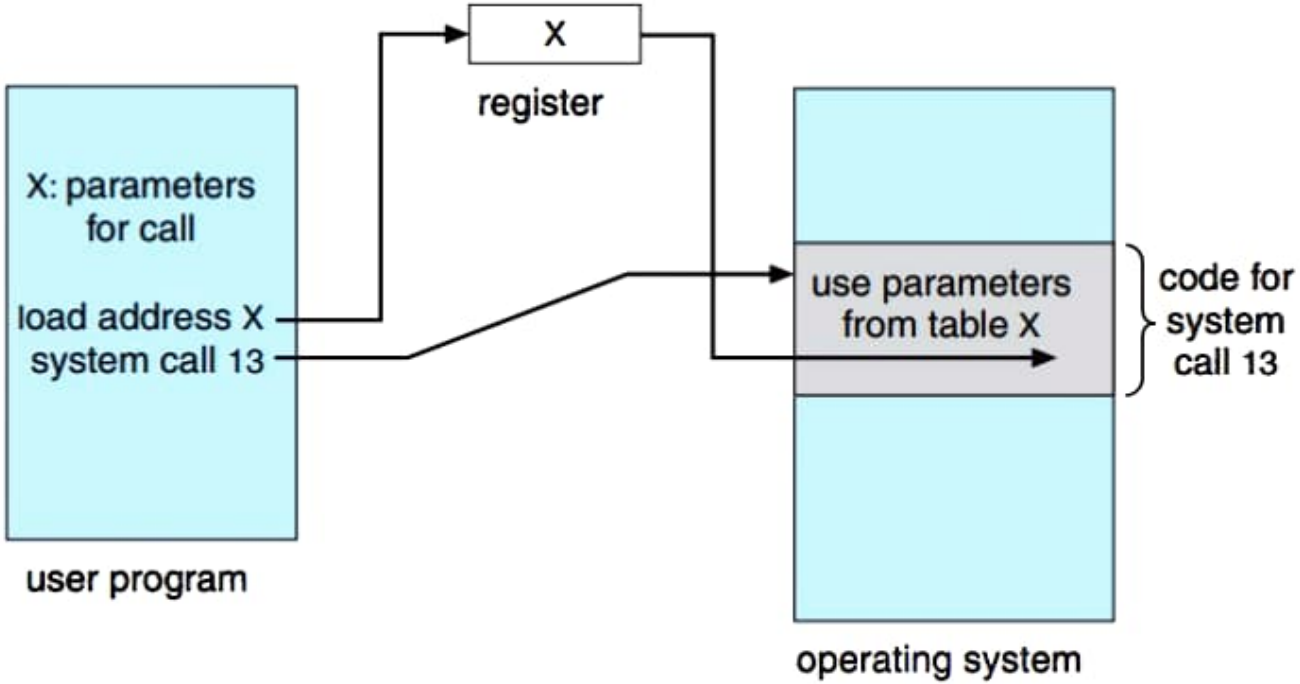


Figure 2.7 Passing of parameters as a table.

4) Types of System calls

System call can be divided into 6 major categories:

a) Process Control
b) File manipulation
c) Device manipulation
d) Information Maintanence
e) Communications
f) Protection

a) Process Control:

- end(): When a process ends normally, this system call is made.
- abort(): When the process ends abnormally, this system call is made.
- load(): When a ~~program~~ process wants to load another program, this system call is made.
- Execute(): When a ~~program~~ process wants to execute another program, this system call is made.
- create_process(): When a process wants to create an entirely new process, this system call is made.

Φ Note: load() and execute() calls are used when an already existing program has to be converted into a process.

While create_process() is used to create entirely new, non existing processes.

The distinction between a program and a process is important here.

* A program is set of instructions that defines a specific task. It is not in an active state in memory.
* A process is active instance of a program that is loaded onto the main memory & being executed in CPU.

- submit_job(): When a job or task has to be submitted, this call is made.

- get_process_attributes(): When the attributes of a created job has to be determined, this call is made.

- set_process_attributes(): When the attributes of a created job has to be changed, this call is made.

- terminate_process() : When we want to terminate a job we created, this system call is made.

- wait_time() : When certain amount of time has to be waited by the parent process, this system call is made.

- wait_event() : When process needs to wait for a specific event to occur, this call is made.

- signal_event() : When jobs signal that the event has occured, this call is made.

- acquire_lock() : When data items need to be locked for a particular process so that no other process can accces it, this system call is made. (in concurrent systems)

- release_lock() : When the process that had a data item locked is done executing & wants to release the lock, this call is made.

{{ MS DOS execution }}

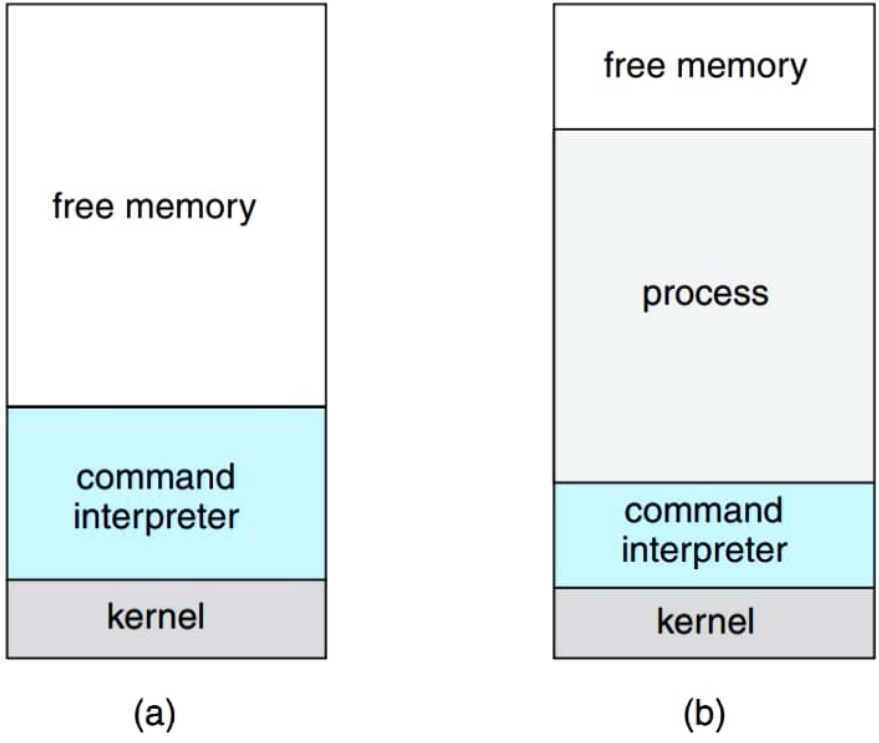


Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program.

→ MS-DOS OS is a single-tasking system, thus only 1 task can be running at a time in the main memory.

When a task is to be loaded, it is written over the free memory available. After it is done, the next process is overwritten.

If there was an error in the previous process, the error is reported to the user or the next program.

The command interpreter is responsible to find out & load the next process to be executed.

{{FreeBSD running multiple programs}}

→ FreeBSD OS is a multitasking system. The command interpreter will continue running while another program is executing.

- fork(): To start a new process in FreeBSD OS, this call is made.

- exec(): To ~~execute~~ load onto memory & execute the selected program, this call is made.

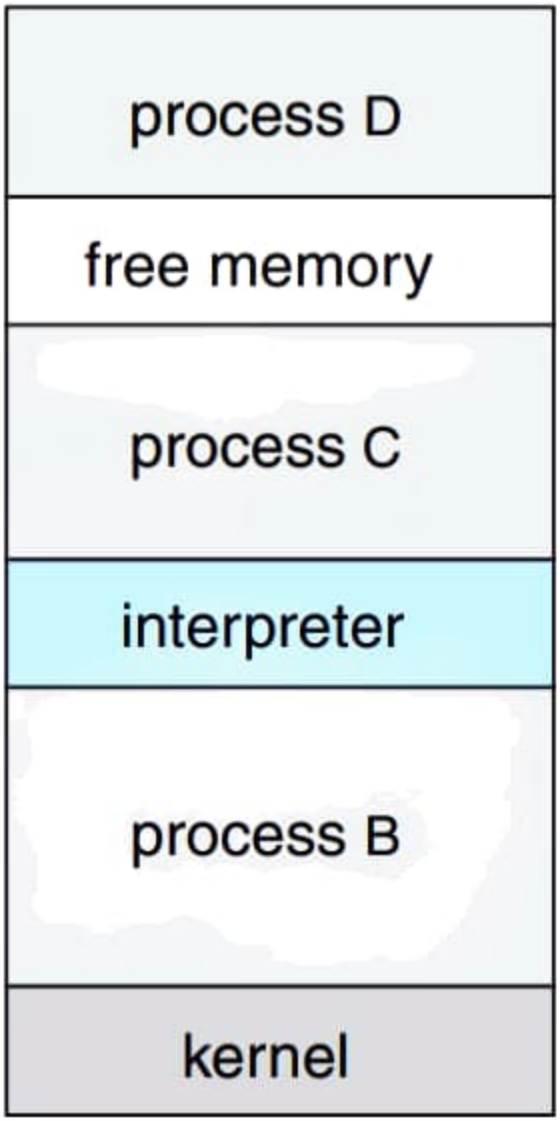


Figure 2.10 FreeBSD running multiple programs.

The process can either wait for the other process to finish execution or it can run in the background.

The process that is running in the background cannot access any I/O resources (like input from the keyboard).

exit()

- end(): When a process ends and wants to terminate, this system call is made.

---

* When an error occurs, a dump of memory is taken to write the error message. This is examined by the debugger which aids the programmer in finding the error.

* Some systems have an error level defined for each error which ranges from 0 to 10, with 0 being a small error and 10 being a severe error. According to the error level the system takes next action.

{{ Process Control Calls }}

- 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

b) File Management:

- create(): To create a file, this call is made.
- delete(): To delete a file, this call is made.
- open(): To open a file and use it.
- read(): Reading a file.
- write(): Writing to a file.
- reposition(): To reposition the file pointer to a different location, this call is made.
- close(): To close the file.
- get_file_attributes(): To find the values of different attributes of a file, eg: file name, size, type, etc.
- set_file_attributes(): To reset the file attributes, this call is made.
- move(): ~~move~~ To move the file from one place to another, this call is made.
- copy(): To create a duplicate of a file at a new location.

c) Device Management:

* What is a device?

Ans) The various resources managed by the operating system are devices. Eg: disk drives, main memory, ~~etc~~ I/O, etc.

Even files are a device, they are just abstract in nature.

- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

{{File & Device Management}}

PAGE NO.:

- request(): To request for a device to get exclusive use of it, this call is made.

- release(): To release the device for other operations, this call is made.

- ~~open~~ read(): Same as file management
- write(): "
- reposition(): "

$ File & device management are similar in great degree.

- get_file_attributes(): Same as file management.
- set_file_attributes(): "

d) Information Maintenance:
Used for transferring information between user program and operating system.

- time(): Returns current time.
- date(): Returns current date.
- dump(): When a dump memory is needed for debugging, this system call is made.

- trace: Lists all the system calls as they are made.

- set_file_attribute(): Same as file management.
- get_file_attributes(): "

{{Information Maintenance}}

- 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

e) Communications: For communication between processes. There are 2 models:

i) Message Passing Model

ii) Shared - Memory Model.

i) Message Passing Model: Communicating processes send messages to one another to share information.

- get_hostid(): The ID of the sender device.
- get_processid(): The ID of the sender process.
- open(): Same as File Management
- close(): "
- accept_connection(): When the recipient needs to give permission to establish communication.
- wait_for_connection(): Used to awaken a process when the connection is made.

* Daemons - The processes that will be receiving connection request are daemons. They execute the wait_for_connection() call for themselves.

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

- read_message(): To read a message.
- write_message(): To write a message.
- close_connection(): To close a connection and terminate communication.

ii) Shared - Memory Model:

- shared_memory_create(): Used when processes need to create access to regions of memory owned by other processes.

- shared_memory_attach(): Used when processes need to gain access to regions of memory owned by other processes.

The restriction to not allow multiple processors to use same resources has to be removed for shared memory model.

f) Protection:

Needed for mechanism for controlling access to the resources provided by the computer system.

- set_permission(): To change the permission of some resource.

- get_permission(): To get the permissions on a resource.

- allow_user(): Specifies whether a specific deny_user() user can be allowed access to certain resources.

1.4 Operating System Structures

1) Simple Structure:

- The MS-DOS implemented this simple structure. It did not have clear distinction between the different components of the operating system.
- The hardware could not be protected from access from any program.

{{MS-DOS structure}}

2)

- Another example is the original UNIX operating system. It had 2 separable parts, the kernel and the system programs.
- The kernel was further divided into smaller components.
- A lot of functionalities are combined to be in the kernel.
- There was more layering in UNIX in comparison to MS-DOS.

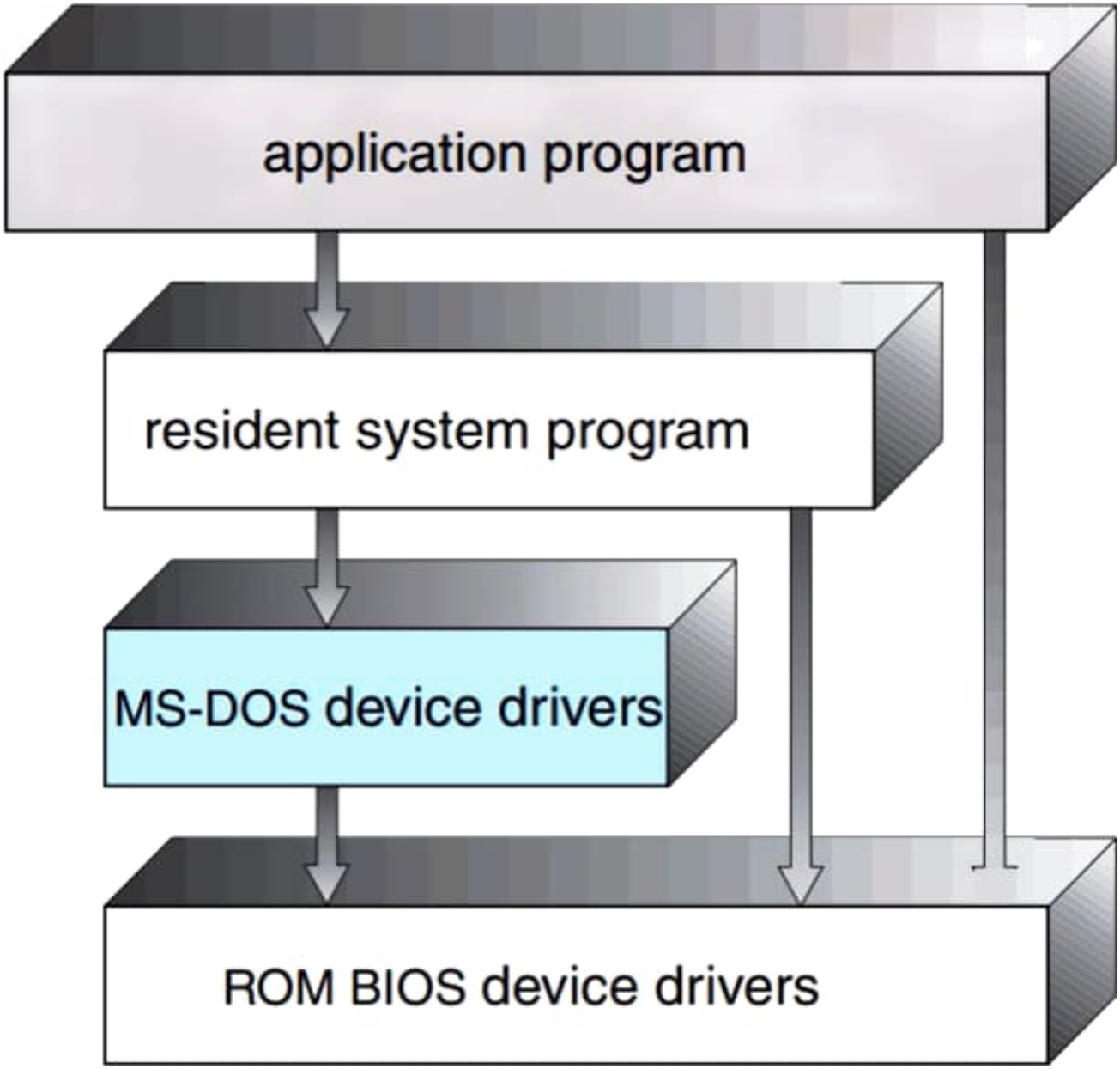


Figure 2.11 MS-DOS layer structure.

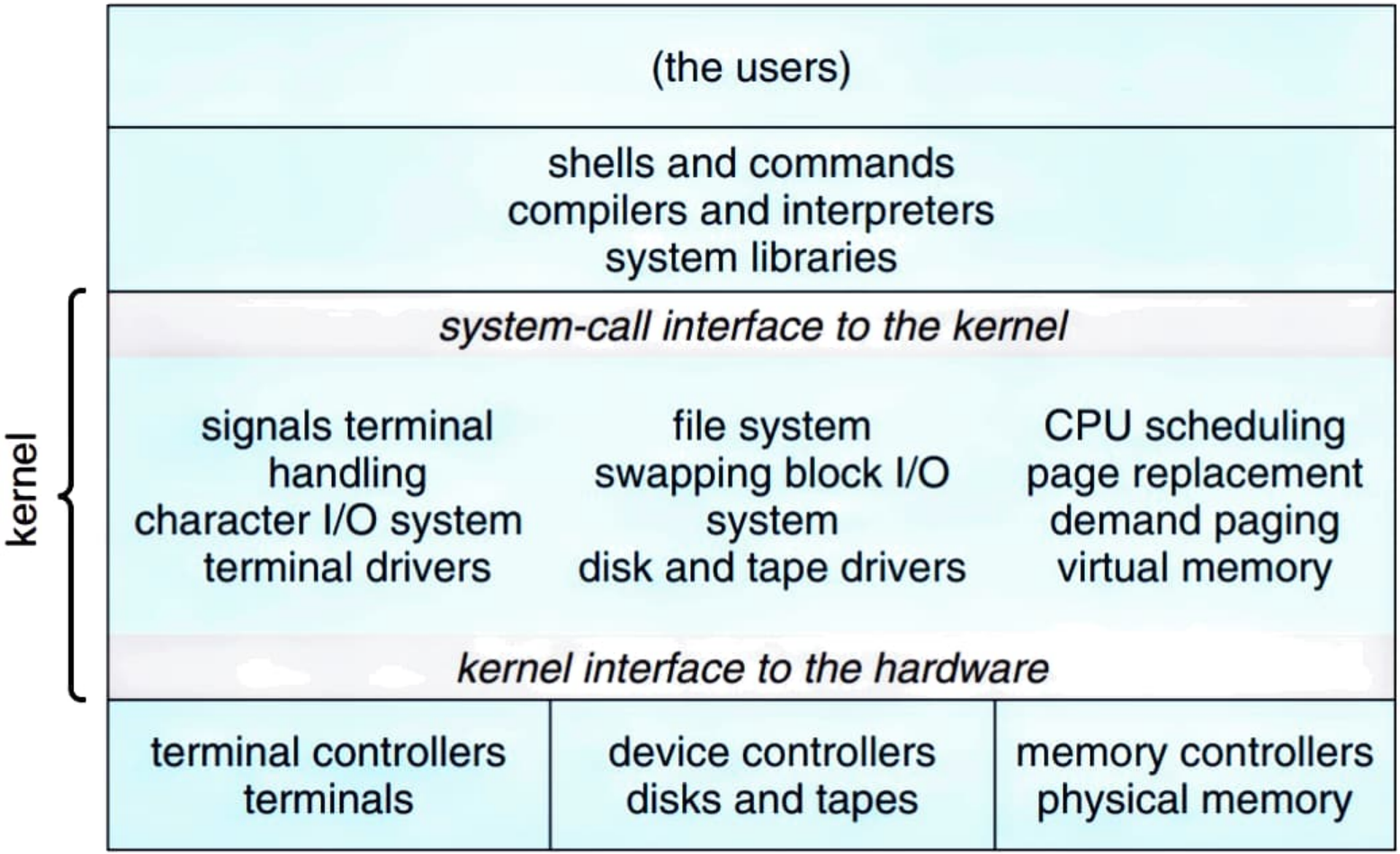


Figure 2.12 Traditional UNIX system structure.

{{ UNIX OS ~~ker~~ Structure }}

PAGE NO.:

- The performance in the simple structure is fast because system calls could directly access the required hardware resource.

2) Layered Approach:

- Greater division between the components is supported in the layered approach.
- Layered approach makes it easier for programmers to work with the system.
- The system is divided into many layers, with level 0 being hardware and layer N (highest layer) being the user interface.
- A layer can invoke data structures and routines of any layer that comes below it.
- If an error is found, the bottom up approach of finding the bug is used. Starting from level 0. Once a bug is found in one of the layers, we can ~~assume~~ know that that layer was the ~~re~~ reason for the error, as the layers beneath it are already checked.
- Each layer hides the details & complexity

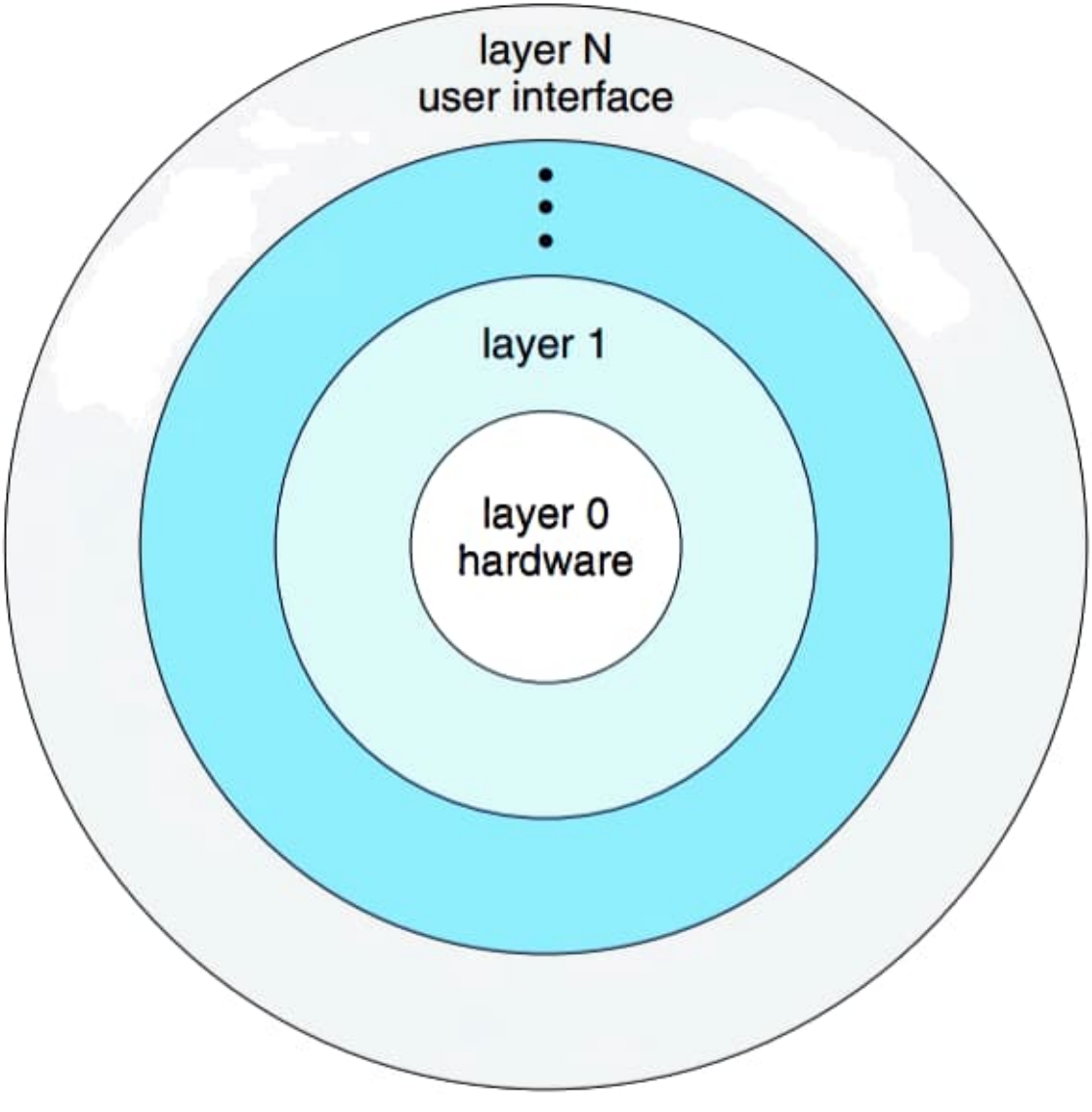


Figure 2.13 A layered operating system.

like data structures from the layer above it.

- One concern of the layered approach is to decide the layers & their functions appropriately.
- Limitation - The layered approach is less efficient in comparison to simple structure. If a system call is made, it will be sent to the next lower level with some parameters, then that layer will send it lower with parameters & structures of its own, till it reaches the hardware level.

3) Microkernels:

- An operating system called Mach was developed, which divided the kernel into smaller parts using the microkernel approach.
- The non-essential components of the kernel were removed and placed in the user space. Thus, the kernel space and the user space were created.
- The main function of the microkernel is to provide communication between user and kernel space.

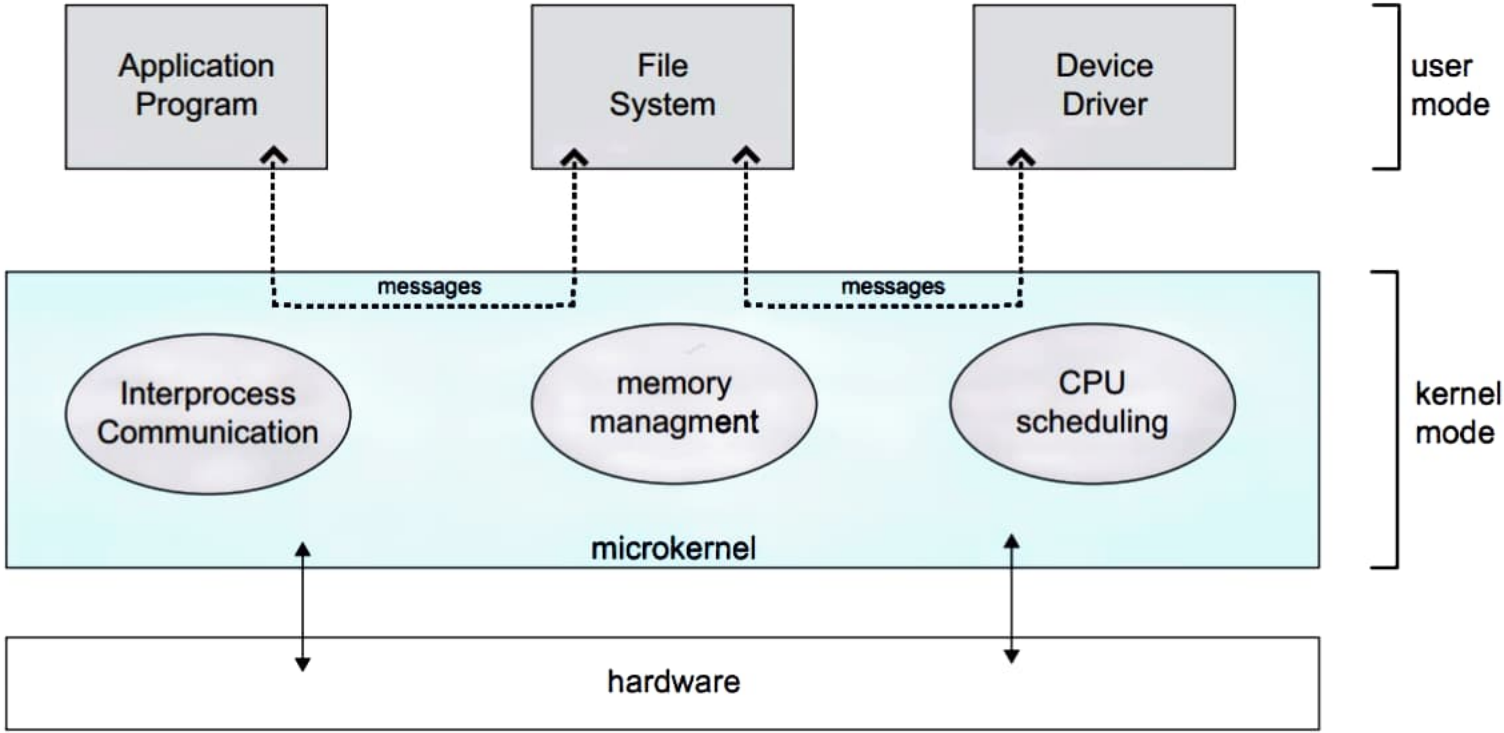


Figure 2.14 Architecture of a typical microkernel.

This communication is done using message passing.

- The main advantage of the microkernel approach is that extending the OS becomes easier.

- Most new updates are for the user space and the micro kernel need not be updated often.

- If update has to be made, the update on micro kernel are usually small because the microkernel is small.

- This approach provides more security because most services are user programs. If a service fails, the microkernel is untouched. Thus, the other processes can continue normally.

1.6 System Boot

- Booting: The procedure of starting a computer by loading the kernel is known as booting the system.

- Bootstrap Program or Bootstrap Loader: This program locates the kernel and loads it onto main memory while booting.

- When a CPU is powered up, the instruction register is loaded with predefined memory location, and execution starts there.
- This location is where the bootstrap program is.
- The program is in the form of ROM, because ROM doesn't need initialization & robust.
- The bootstrap program:
  - → Runs diagnostics to determine state of machine, if the diagnostics pass, booting can continue.
  - → Initializing all aspects of the program (registers, etc).
  - → Starting the operating system.
- Firmware: All forms of ROM are known as firmware, because they lie somewhere in between hardware and software.

  Disadvantage: Slower execution than in RAM.

  Some systems copy the code in the RAM & then execute.

  It's expensive, so small amounts are used.

- In large OS (like Windows, MAC OS) the bootstrap loader is in firmware and the OS is on the disk.

- Boot Block: Bootstrap loader has some code that can read a single block at a fixed location, that is called the Boot Block.

  The Boot Block contains the OS.

- GRUB: Open Source Bootstrap program for Linux systems.

- running state: When the Bootstrap loader loads the OS in the main memory & the OS has started execution, the system is in the running state.