

Assignment 2

Concepts of Operating System

30 August 2024 13:40

Part A

What will the following commands do?

echo "Hello, World!"

It will print the Hello, World!

```
cdac@Abhay:~$ echo "Hello, World!"  
Hello, World!  
cdac@Abhay:~$ _
```

name="Productive"

It will store the String Productive

```
cdac@Abhay:~$ name="Productive"  
cdac@Abhay:~$ _
```

touch file.txt

It will create file named file.txt

```
cdac@Abhay:~$ touch file.txt  
cdac@Abhay:~$
```

ls -a

It will list out all the files and directories.

```
cdac@Abhay:~$ ls -a  
. .bash_history .bashrc .local .profile file.txt  
.. .bash_logout .cache .motd_shown .sudo_as_admin_successful  
cdac@Abhay:~$ _
```

rm file.txt

It will delete the file.txt

```
cdac@Abhay:~$ rm file.txt
cdac@Abhay:~$ ls
cdac@Abhay:~$
```

cp file1.txt file2.txt

It will copy file1.txt and rename it as file2.txt

```
cdac@Abhay:~$ cp file1.txt file2.txt
cdac@Abhay:~$ ls
file1.txt  file2.txt
cdac@Abhay:~$
```

mv file.txt /path/to/directory/

It will copy file.txt to directory

```
cdac@Abhay:~$ mv file.txt directory
cdac@Abhay:~$ cd directory
cdac@Abhay:~/directory$ ls
file.txt
cdac@Abhay:~/directory$
```

chmod 755 script.sh

It will create executable script.sh file.

```
cdac@Abhay:~$ chmod 755 script.sh
cdac@Abhay:~$ ls
directory  file1.txt  file2.txt  script.sh
```

grep "pattern" file.txt

It will display the word "pattern" which is inside the file.txt

```
cdac@Abhay:~$ nano file.txt
cdac@Abhay:~$ grep "pattern" file.txt
This is a pattern.
cdac@Abhay:~$
```

kill PID

It is used to kill process. PID is process ID.

```
cdac@Abhay:~$ ps
  PID TTY          TIME CMD
   358 pts/0        00:00:00 bash
  11535 pts/0        00:00:00 ps
cdac@Abhay:~$ kill 11535
-bash: kill: (11535) - No such process
cdac@Abhay:~$
```

mkdir mydir && cd mydir && touch file.txt && echo "Hello, World!" > file.txt && cat file.txt

mkdir will create directory mydir then cd will change directory to mydir then touch will create file.txt in mydir and then echo will print "Hello World!" in file.txt then > will take output of left side commands as input to right side commands of > then cat will display "Hello World!" as an output.

```
cdac@Abhay:~$ mkdir mydir && cd mydir && touch file.txt && echo "Hello, World!" > file.txt && cat file.txt
Hello, World!
cdac@Abhay:~/mydir$
```

ls -l | grep ".txt"

ls -l will list all files and directories in long format.

Pipe | is used to take output of ls -l as an input for grep ".txt"

grep ".txt" will search for the files which contains .txt in their names.

```
cdac@Abhay:~$ cd abc
cdac@Abhay:~/abc$ ls
file.txt file1.txt file2.txt file3.sh file4
cdac@Abhay:~/abc$ ls -l | grep ".txt"
-rw-r--r-- 1 cdac cdac 0 Aug 30 14:54 file.txt
-rw-r--r-- 1 cdac cdac 0 Aug 30 14:54 file1.txt
-rw-r--r-- 1 cdac cdac 0 Aug 30 14:54 file2.txt
cdac@Abhay:~/abc$
```

cat file1.txt file2.txt | sort | uniq

cat will display the content of file1.txt and file2.txt. Then sort will sort it down alphabetically. Then uniq will display the duplicate content of files at one time only.

```
cdac@Abhay:~$ cat file1.txt file2.txt | sort | uniq
.
I
am
are
fine
hello
hi
how
thank
you
cdac@Abhay:~$
```

ls -l | grep "^d"

grep "^d" will search files or directories that start with d.

```
dir dir1 file1 file2
cdac@Abhay:~$ ls -l | grep "^d"
drwxr-xr-x 2 cdac cdac 4096 Aug 30 15:27 dir
drwxr-xr-x 2 cdac cdac 4096 Aug 30 15:27 dir1
cdac@Abhay:~$
```

grep -r "pattern" /path/to/directory/

it will search "pattern" present in the file.

```
cdac@Abhay:~$ grep -r "pattern" file1
pattern123
cdac@Abhay:~$
```

cat file1.txt file2.txt | sort | uniq -d

uniq -d is used to display content of file only once when there is duplicate content present. If file contains hi, 2 times then it will display hi 1 time. If hi is present only one time then it will not display it.

```
cdac@Abhay:~$ cat file1.txt file2.txt | sort | uniq -d
good
hello
hi
morning
```

chmod 644 file.txt

chmod will change permissions. 6 will change owner permission to read and write. 4 will change group permission to read. Next 4 will change other permission to read.

```
cdac@Abhay:~$ ls -l
total 0
----- 1 cdac cdac 0 Aug 30 15:43 file.txt
cdac@Abhay:~$ chmod 644 file.txt
cdac@Abhay:~$ ls -l
total 0
-rw-r--r-- 1 cdac cdac 0 Aug 30 15:43 file.txt
cdac@Abhay:~$ _
```

cp -r source_directory destination_directory

cp will copy the source_directory to destination_directory. -r will include all the files and directories present in source_directory.

```
cdac@Abhay:~$ cp -r dir dir2
cdac@Abhay:~$ ls
dir  dir2
cdac@Abhay:~$ cd dir2
cdac@Abhay:~/dir2$ ls
file1
cdac@Abhay:~/dir2$ _
```

find /path/to/search -name "*.txt"

It will find files in given directory which name contains ".txt"

```
cdac@Abhay:~$ find dir -name "*.txt"
dir/file.txt
dir/file2.txt
dir/file1.txt
cdac@Abhay:~$ _
```

chmod u+x file.txt

chmod will change the permission of user of file.txt as executable.

```
cdac@Abhay:~$ ls -l
total 0
-rw-r--r-- 1 cdac cdac 0 Aug 30 16:05 file.txt
cdac@Abhay:~$ chmod u+x file.txt
cdac@Abhay:~$ ls -l
total 0
-rwxr--r-- 1 cdac cdac 0 Aug 30 16:05 file.txt
cdac@Abhay:~$ _
```

echo \$PATH

it will give path of different directories.

```
cdac@Abhay:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:
```

Part B

Identify True or False:

1. ls is used to list files and directories in a directory.

=> True

2. mv is used to move files and directories.

=> True

3. cd is used to copy files and directories.

=> False. cd is used to change directory.

4. pwd stands for "print working directory" and displays the current directory.

=> True

5. grep is used to search for patterns in files.

=> True

6. chmod 755 file.txt gives read, write, and execute permissions to the owner, and read and execute permissions to group and others.

=> True

7. mkdir -p directory1/directory2 creates nested directories, creating directory2 inside directory1 if directory1 does not exist.

=> True

8. rm -rf file.txt deletes a file forcefully without confirmation.

=> True

Identify the Incorrect Commands:

1. chmodx is used to change file permissions.

=> chmod is used.

2. cpy is used to copy files and directories.

=> cp is used to copy files and directories.

3. mkfile is used to create a new file.

=> touch and nano is used to create new files.

4. catx is used to concatenate files.

=> cat is used to display file content.

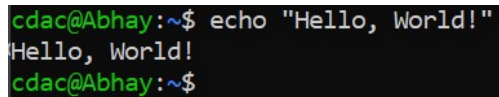
5. rn is used to rename files.

=> mv is used to rename and move.

Part C

Question 1: Write a shell script that prints "Hello, World!" to the terminal.

```
echo "hello, World!"
```

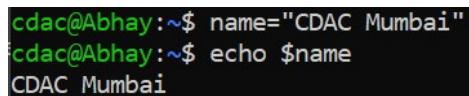


```
cdac@Abhay:~$ echo "Hello, World!"  
Hello, World!  
cdac@Abhay:~$
```

Question 2: Declare a variable named "name" and assign the value "CDAC Mumbai" to it. Print the value of the variable.

```
name="CDAC Mumbai"
```

```
Echo $name
```



```
cdac@Abhay:~$ name="CDAC Mumbai"  
cdac@Abhay:~$ echo $name  
CDAC Mumbai
```

Question 3: Write a shell script that takes a number as input from the user and prints it.

```
nano num
```

```
[
```

```
read num
```

```
echo num is, $num
```

```
]
```

```
bash num
```

```
cdac@Abhay:~$ nano num
cdac@Abhay:~$ bash num
12
num is, 12
```

```
read num
echo num is, $num
```

Question 4: Write a shell script that performs addition of two numbers (e.g., 5 and 3) and prints the result.

```
nano sum
[
echo take two numbers
read num1
read num2
echo sum is, $((num1 + num2))
]
bash sum
```

```
echo take two numbers
read num1
read num2
echo result is, $((num1 + num2))
```

```
cdac@Abhay:~$ nano sum
cdac@Abhay:~$ bash sum
take two numbers
5
3
result is, 8
```

Question 5: Write a shell script that takes a number as input and prints "Even" if it is even, otherwise prints "Odd".

```
nano evenodd
[
echo take a number
read num
if ((num%2==0))
then
    echo $num is even
else
    echo $num is odd
fi
]
bash evenodd
```



```
echo take a number
read num
if((num%2==0))
then
    echo $num is even
else
    echo $num is odd
fi
```

```
cdac@Abhay:~$ nano evenodd
cdac@Abhay:~$ bash evenodd
take a number
12
12 is even
cdac@Abhay:~$ bash evenodd
take a number
15
15 is odd
cdac@Abhay:~$ _
```

Question 6: Write a shell script that uses a for loop to print numbers from 1 to 5.

```
nano loop
[
#!/bin/bash
num=0
for num in 1 2 3 4 5
do
    echo $num
done
]
bash loop
```

```
#!/bin/bash
num=0
for num in 1 2 3 4 5
do
    echo $num
done
```

```
cdac@Abhay:~$ nano loop
cdac@Abhay:~$ bash loop
1
2
3
4
5
```

Question 7: Write a shell script that uses a while loop to print numbers from 1 to 5.

```
nano loop
[
#!/bin/bash
num=1
while [ $num -lt 6 ]
do
    echo $num
    num=`expr $num + 1`
done
]
bash loop
```

```
#!/bin/bash
num=1
while [ $num -lt 6 ]
do
    echo $num
    num=`expr $num + 1`
done
```

```
cdac@Abhay:~$ nano loop
cdac@Abhay:~$ bash loop
1
2
3
4
5
```

Question 8: Write a shell script that checks if a file named "file.txt" exists in the current directory. If it does, print "File exists", otherwise, print "File does not exist".

```
nano find
[
```

```
#!/bin/bash
if [ -f "find.txt" ]
then
    echo "Exist"
else
    echo "Not Exist"
]
Bash find
```

```
#!/bin/bash
if [ -f "file.txt" ]
then
    echo "Exist"
else
    echo "Not Exist"
fi
```

```
cdac@Abhay:~$ nano find
cdac@Abhay:~$ bash find
Not Exist
cdac@Abhay:~$ touch file.txt
cdac@Abhay:~$ bash find
Exist
```

Question 9: Write a shell script that uses the if statement to check if a number is greater than 10 and prints a message accordingly.

```
nano gt
[
#!/bin/bash
echo take a number
read num
If [ $num -gt 10 ]
then
    echo $num is greater than 10
else
    echo $num is less than 10
fi
]
bash gt
```

```
#!/bin/bash
echo take a number
read num
if [ $num -gt 10 ]
then
    echo $num is greater than 10
else
    echo $num is less than 10
fi
```

```
cdac@Abhay:~$ nano gt
cdac@Abhay:~$ bash gt
take a number
5
5 is less than 10
cdac@Abhay:~$ bash gt
take a number
65
65 is greater than 10
cdac@Abhay:~$
```

Question 10: Write a shell script that uses nested for loops to print a multiplication table for numbers from 1 to 5. The output should be formatted nicely, with each row representing a number and each column representing the multiplication result for that number.

```
nano table
[
#!/bin/bash
for num in 1 2 3 4 5
do
for num1 in 1 2 3 4 5 6 7 8 9 10
do
    Printf "%d " $(( $num*$num1 ))
done
echo
Done
]
bash table
```

```
#!/bin/bash

for num in 1 2 3 4 5
do
for num1 in 1 2 3 4 5 6 7 8 9 10
do
    printf "%d " $(( $num*$num1 ))
done
echo
done
```

```
cdac@Abhay:~$ nano table
cdac@Abhay:~$ bash table
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
cdac@Abhay:~$
```

Question 11: Write a shell script that uses a while loop to read numbers from the user until the user enters a negative number. For each positive number entered, print its square. Use the break statement to exit the loop when a negative number is entered.

```
nano loop
[
#!/bin/bash
num=0
while true
do
    read -p "enter number: " num
    If [ $num -lt 0 ]
    then
        Break
    else
        echo "square: $(( $num * $num ))"
    fi
done
]
Bash loop
```

```
#!/bin/bash
num=0
while true
do
    read -p "enter number: " num
```

```
#!/bin/bash
num=0
while true
do
    read -p "enter number: " num
    if [ $num -lt 0 ]
    then
        break
    else
        echo "square: $(( $num * $num ))"
    fi
done
```

```
cdac@Abhay:~$ nano loop
cdac@Abhay:~$ bash loop
enter number: 5
square: 25
enter number: -2
cdac@Abhay:~$
```

1. Consider the following processes with arrival times and burst times:

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	6

Calculate the average waiting time using First-Come, First-Served (FCFS) scheduling.

Process	Wait Time [Allocation Time-Arrival Time]	Turn Around Time (TAT) [Completion Time - Arrival Time]
P1	0 - 0 = 0	5 - 0 = 5
P2	5 - 1 = 4	8 - 1 = 7
P3	8 - 2 = 6	14 - 2 = 12

Gantt Chart

P1	P2	P3	
0	5	8	14

Avg. Wait Time : $9/3 = 3$

Avg. TAT Time : $24/3 = 8$

2. Consider the following processes with arrival times and burst times:

Process	Arrival Time	Burst Time
P1	0	3
P2	1	5
P3	2	1
P4	3	4

Calculate the average turnaround time using Shortest Job First (SJF) scheduling.

process	Wait Time [Allocation Time-Arrival Time]	Turn Around Time (TAT) [Completion Time - Arrival Time]
1	0 - 0 = 0	3 - 0 = 3
2	8 - 1 = 7	13 - 1 = 12
3	3 - 2 = 1	4 - 2 = 2
4	4 - 3 = 1	8 - 3 = 5

Gantt Chart:

P1	P3	P4	P2	
0	3	4	8	13

Avg. Wait Time : $9/4 = 2.25$

Avg. TAT Time : $22/4 = 5.5$

3. Consider the following processes with arrival times, burst times, and priorities (lower number indicates higher priority):

Process	Arrival Time	Burst Time	Priority
P1	0	6	3
P2	1	4	1
P3	2	7	4
P4	3	2	2

Calculate the average waiting time using Priority Scheduling.

process	Wait Time [Allocation Time-Arrival Time]	Turn Around Time (TAT) [Completion Time - Arrival Time]
1	$0 - 0 + 6 = 6$	$12 - 0 = 12$
2	$1 - 1 = 0$	$5 - 1 = 4$
3	$12 - 2 = 10$	$19 - 2 = 17$
4	$5 - 3 = 2$	$7 - 3 = 4$

Gantt Chart:

P1	P2	P4	P1	P3	
0	1	5	7	12	19

Avg. Wait Time : $18/4 = 4.5$

Avg. TAT Time : $37/4 = 9.25$

4. Consider the following processes with arrival times and burst times, and the time quantum for

Round Robin scheduling is 2 units:

Process	Arrival Time	Burst Time
P1	0	4
P2	1	5
P3	2	2
P4	3	3

Calculate the average turnaround time using Round Robin scheduling.

process	Wait Time [Allocation Time-Arrival Time]	Turn Around Time (TAT) [Completion Time - Arrival Time]
1	$0 - 0 + 6 = 6$	$10 - 0 = 10$
2	$2 - 1 + 6 + 2 = 9$	$15 - 1 = 14$
3	$4 - 2 = 2$	$6 - 2 = 4$
4	$6 - 3 + 4 = 7$	$13 - 3 = 10$

Gantt Chart:

P1	P2	P3	P4	P1	P2	P4	P2
----	----	----	----	----	----	----	----

0	2	4	6	8	10	12	14
---	---	---	---	---	----	----	----

Avg. Wait Time : $24/4 = 6$

Avg. TAT Time : $38/4 = 9.5$

5. Consider a program that uses the `fork()` system call to create a child process. Initially, the parent process has a variable `x` with a value of 5. After forking, both the parent and child processes increment the value of `x` by 1. What will be the final values of `x` in the parent and child processes after the `fork()` call?

Child process will contain value similar to parent process. So initially parent process and child process have value 5. But after incrementing value by 1 then both parent and child processes will have value 6.

Part D

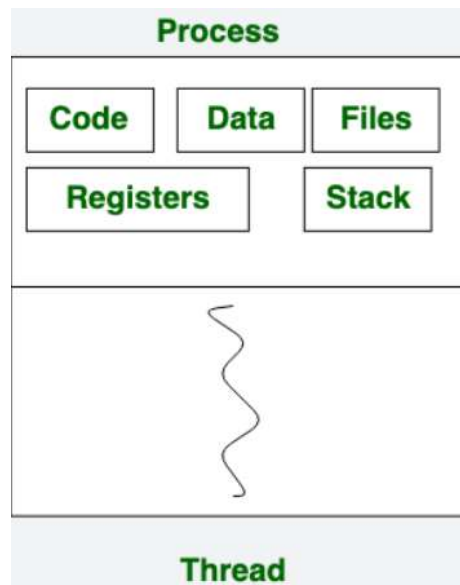
Common Interview Questions (Must know)

1. What is an operating system, and what are its primary functions?

An operating system (OS) is an interface between the computer hardware and the user, managing software resources and computer hardware. The primary functions of an operating system are process management, memory management, file systems management, device management, and security and privacy.

2. Explain the difference between process and thread.

Process and threads are the basic components in OS. Process is the program under execution whereas the thread is part of process. Threads of a process can be used when same process is required multiple times. A process can consists of multiple threads.



Process	Thread
Process means any program is in execution.	Thread means a segment of a process.
The process takes more time to terminate.	The thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
It also takes more time for context switching.	It takes less time for context switching.
The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
The process is isolated.	Threads share memory.
The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
If one process is blocked, then it will not affect the execution of other processes.	If a user-level thread is blocked, then all other user-level threads are blocked.
The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.

Changes to the parent process do not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.
A system call is involved in it.	No system call is involved, it is created using APIs.
The process does not share data with each other.	Threads share data with each other.

3. What is virtual memory, and how does it work?

Virtual memory is a method that computers use to manage storage space to keep systems running quickly and efficiently. Using the technique, operating systems can transfer data between different types of storage, such as random access memory (RAM), also known as main memory, and hard drive or solid-state disk storage. At any particular time, the computer only needs enough active memory to support active processes. The system can move those that are dormant into virtual memory until needed.

how does it work:

Virtual memory uses both the computer's software and hardware to work. It transfers processes between the computer's RAM and hard disk by copying any files from the computer's RAM that aren't currently in use and moving them to the hard disk. By moving unused files to the hard disk, a computer frees up space in its RAM to perform current tasks, such as opening a new application. If the computer later needs to use its RAM for a more urgent task, it can again swap files to make the most of the available RAM.

RAM is a limited resource stored on chips in the computer's CPU. Installing more RAM chips can be expensive, so virtual memory allows the computer to move files between systems as needed to optimize its use of the available RAM.

4. Describe the difference between multiprogramming, multitasking, and multiprocessing.

Multiprogramming – Multiprogramming is known as keeping multiple programs in the main memory at the same time ready for execution.

Multiprocessing – A computer using more than one CPU at a time.

Multitasking – Multitasking is nothing but multiprogramming with a Round-robin scheduling algorithm.

Multithreading is an extension of multitasking.

Feature	Multiprogramming	Multitasking	Multithreading	Multiprocessing
Definition	Running multiple programs on a	Running multiple tasks (applications)	Running multiple threads within a	Running multiple processes on

	single CPU	on a single CPU	single task (application)	multiple CPUs (or cores)
Resource Sharing	Resources (CPU, memory) are shared among programs	Resources (CPU, memory) are shared among tasks	Resources (CPU, memory) are shared among threads	Each process has its own set of resources (CPU, memory)
Scheduling	Uses round-robin or priority-based scheduling to allocate CPU time to programs	Uses priority-based or time-slicing scheduling to allocate CPU time to tasks	Uses priority-based or time-slicing scheduling to allocate CPU time to threads	Each process can have its own scheduling algorithm
Memory Management	Each program has its own memory space	Each task has its own memory space	Threads share memory space within a task	Each process has its own memory space
Context Switching	Requires a context switch to switch between programs	Requires a context switch to switch between tasks	Requires a context switch to switch between threads	Requires a context switch to switch between processes
Inter-Process Communication (IPC)	Uses message passing or shared memory for IPC	Uses message passing or shared memory for IPC	Uses thread synchronization mechanisms (e.g., locks, semaphores) for IPC	Uses inter-process communication mechanisms (e.g., pipes, sockets) for IPC

5. What is a file system, and what are its components?

A file system is a method an operating system uses to store, organize, and manage files and directories on a storage device. Some common types of file systems include:

- FAT (File Allocation Table): An older file system used by older versions of Windows and other operating systems.
- NTFS (New Technology File System): A modern file system used by Windows. It supports features such as file and folder permissions, compression, and encryption.
- ext (Extended File System): A file system commonly used on Linux and Unix-based operating systems.
- HFS (Hierarchical File System): A file system used by macOS.
- APFS (Apple File System): A new file system introduced by Apple for their Macs and iOS devices.

6. What is a deadlock, and how can it be prevented?

A deadlock can occur in almost any situation where processes share resources. It can happen

in any computing environment, but it is widespread in distributed systems, where multiple processes operate on different resources.

In the deadlock prevention process, the OS will prevent the deadlock from occurring by avoiding any one of the four conditions that caused the deadlock. If the OS can avoid any of the necessary conditions, a deadlock will not occur. That conditions are mutual exclusion, hold and wait, no preemption, and circular wait.

7. Explain the difference between a kernel and a shell.

S.No.	Shell	Kernel
1.	Shell allows the users to communicate with the kernel.	Kernel controls all the tasks of the system.
2.	It is the interface between kernel and user.	It is the core of the operating system.
3.	It is a command line interpreter (CLI).	Its a low level program interfacing with the hardware (CPU, RAM, disks) on top of which applications are running.
4.	Its types are – Bourne Shell, C shell, Korn Shell, etc.	Its types are – Monolithic Kernel, Micro kernel, Hybrid kernel, etc.
5.	It carries out commands on a group of files by specifying a pattern to match	It performs memory management.
6.	Shell commands like ls, mkdir and many more can be used to request to complete the specific operation to the OS.	It performs process management.
7.	It is the outer layer of OS.	It is the inner layer of OS.
8.	It interacts with user and interprets to machine understandable language.	Kernel directly interacts with the hardware by accepting machine understandable language from the shell.
9.	Command-line interface that allows user interaction	Core component of the operating system that manages system resources
10.	Interprets and translates user commands	Provides services to other programs running on the system
11.	Acts as an intermediary between the user and the kernel	Operates at a lower level than the shell and interacts with hardware
12.	Provides various features like command history, tab completion, and scripting	Responsible for tasks such as memory management, process scheduling, and

	capabilities	device drivers
13.	Executes commands and programs	Enables user and applications to interact with hardware resources

8. What is CPU scheduling, and why is it important?

Scheduling of processes/work is done to finish the work on time. CPU Scheduling is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I / O etc, thus making full use of the CPU. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer. CPU scheduling is a key part of how an operating system works. It decides which task (or process) the CPU should work on at any given time. This is important because a CPU can only handle one task at a time, but there are usually many tasks that need to be processed.

why it is imp:

Scheduling is important in many different computer environments. One of the most important areas is scheduling which programs will work on the CPU. This task is handled by the Operating System (OS) of the computer and there are many different ways in which we can choose to configure programs.

Process Scheduling allows the OS to allocate CPU time for each process. Another important reason to use a process scheduling system is that it keeps the CPU busy at all times. This allows you to get less response time for programs.

Considering that there may be hundreds of programs that need to work, the OS must launch the program, stop it, switch to another program, etc. The way the OS configures the system to run another in the CPU is called “context switching”. If the OS keeps context-switching programs in and out of the provided CPUs, it can give the user a tricky idea that he or she can run any programs he or she wants to run, all at once.

So now that we know we can run 1 program at a given CPU, and we know we can change the operating system and remove another one using the context switch, how do we choose which programs we need. run, and with what program?

That’s where scheduling comes in! First, you determine the metrics, saying something like “the amount of time until the end”. We will define this metric as “the time interval between which a function enters the system until it is completed”. Second, you decide on a metrics that reduces metrics. We want our tasks to end as soon as possible.

9. How does a system call work?

A system call is a programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to interact with the operating system. A computer program makes a system call when it requests the

operating system's kernel.

Users need special resources: Sometimes programs need to do some special things that can't be done without the permission of the OS like reading from a file, writing to a file, getting any information from the hardware, or requesting a space in memory.

The program makes a system call request: There are special predefined instructions to make a request to the operating system. These instructions are nothing but just a "system call". The program uses these system calls in its code when needed.

Operating system sees the system call: When the OS sees the system call then it recognizes that the program needs help at this time so it temporarily stops the program execution and gives all the control to a special part of itself called 'Kernel'. Now 'Kernel' solves the need of the program.

The operating system performs the operations: Now the operating system performs the operation that is requested by the program. Example: reading content from a file etc.

Operating system give control back to the program : After performing the special operation, OS give control back to the program for further execution of program.

10. What is the purpose of device drivers in an operating system?

Device Driver in computing refers to a special kind of software program or a specific type of software application that controls a specific hardware device that enables different hardware devices to communicate with the computer's Operating System. A device driver communicates with the computer hardware by computer subsystem or computer bus connected to the hardware.

Device Drivers are essential for a computer system to work properly because without a device driver the particular hardware fails to work accordingly, which means it fails in doing the function/action it was created to do. Most use the term Driver, but some may say Hardware Driver, which also refers to the Device Driver.

11. Explain the role of the page table in virtual memory management.

A Page Table is a data structure used by the operating system to keep track of the mapping between virtual addresses used by a process and the corresponding physical addresses in the system's memory.

A Page Table Entry (PTE) is an entry in the Page Table that stores information about a particular page of memory. Each PTE contains information such as the physical address of the page in memory, whether the page is present in memory or not, whether it is writable or not, and other access permissions.

The size and format of a PTE can vary depending on the architecture of the system and the

operating system used. In general, a PTE contains enough information to allow the operating system to manage memory efficiently and protect the system from malicious or accidental access to memory.

12. What is thrashing, and how can it be avoided?

Thrashing is a condition or a situation when the system is spending a major portion of its time servicing the page faults, but the actual processing done is very negligible.

Causes of thrashing:

- High degree of multiprogramming.
- Lack of frames.
- Page replacement policy.

Techniques to handle:

1. Working Set Model –

This model is based on the above-stated concept of the Locality Model.

The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash.

According to this model, based on parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

The accuracy of the working set is dependent on the value of parameter A. If A is too large, then working sets may overlap. On the other hand, for smaller values of A, the locality might not be covered entirely.

2. Page Fault Frequency –

A more direct approach to handling thrashing is the one that uses the Page-Fault Frequency concept.

The problem associated with Thrashing is the high page fault rate and thus, the concept here is to control the page fault rate.

If the page fault rate is too high, it indicates that the process has too few frames allocated to it. On the contrary, a low page fault rate indicates that the process has too many frames.

Upper and lower limits can be established on the desired page fault rate as shown in the diagram.

If the page fault rate falls below the lower limit, frames can be removed from the process.

Similarly, if the page fault rate exceeds the upper limit, more frames can be allocated to the

process.

In other words, the graphical state of the system should be kept limited to the rectangular region formed in the given diagram.

Here too, if the page fault rate is high with no free frames, then some of the processes can be suspended and frames allocated to them can be reallocated to other processes. The suspended processes can then be restarted later.

13. Describe the concept of a semaphore and its use in synchronization.

Semaphores are just normal variables used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes.

The process of using Semaphores provides two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is >0 . If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. The initial value of a semaphore determines how many processes can get past the wait operation.

14. How does an operating system handle process synchronization?

An operating system handles process synchronization by making sure that when multiple processes (or programs) are running at the same time, they don't interfere with each other, especially when they need to access shared resources like memory or files. Imagine several people trying to use the same computer at once; the operating system ensures that they take turns in a way that prevents them from accidentally overwriting each other's work. It does this by "locking" resources when one process is using them so that other processes have to wait until the resource is free. This prevents errors and ensures that everything runs smoothly, even with many processes happening simultaneously.

15. What is the purpose of an interrupt in operating systems?

An interrupt in an operating system is like a signal that tells the computer to stop what it's doing and pay attention to something more important. Imagine you're watching a movie, and suddenly, your phone rings; you pause the movie to answer the call because it's more urgent.

Similarly, when a computer gets an interrupt, it pauses whatever task it's working on to quickly deal with something that needs immediate attention, like a mouse click or a new data coming in. Once the interrupt is handled, the computer goes back to what it was doing before. This helps the computer respond quickly to important events without missing a beat.

16. Explain the concept of a file descriptor.

A file descriptor is a unique identifier or reference that the operating system assigns to a file when it is opened. It allows programs to interact with files, sockets, or other input/output (I/O) resources. The file descriptor is used by the operating system to keep track of the file and perform operations on it. File descriptors are typically represented as non-negative integers. The operating system assigns the lowest available file descriptor to a newly opened file. The file descriptor is used by the program to refer to the file when performing read, write, or other operations. To open a file and obtain its file descriptor, you can use functions provided by your programming language or operating system. For example, in C, you can use the `open()` function, which returns the file descriptor associated with the opened file. The file descriptor can then be used for subsequent operations on the file.

17. How does a system recover from a system crash?

When a system crashes, it's like a car suddenly breaking down on the road. To recover, the computer goes through a process similar to restarting the car. It first shuts down any ongoing processes to prevent further damage, then restarts itself, often running checks to make sure everything is in order, like checking the car's engine before driving again. The system might also use backups or save points it created earlier to restore important data, just like how you might rely on a GPS history if your car's navigation resets. This recovery process helps the computer get back to normal operation as quickly as possible, while trying to avoid losing any important work you were doing.

18. Describe the difference between a monolithic kernel and a microkernel.

S. No.	Parameters	Microkernel	Monolithic kernel
1.	Address Space	In microkernel, user services and kernel services are kept in separate address space.	In monolithic kernel, both user services and kernel services are kept in the same address space.
2.	Design and Implementation	OS is complex to design.	OS is easy to design and implement.
3.	Size	Microkernel are smaller in size.	Monolithic kernel is larger than microkernel.
4.	Functionality	Easier to add new	Difficult to add new functionalities.

		functionalities.	
5.	Coding	To design a microkernel, more code is required.	Less code when compared to microkernel
6.	Failure	Failure of one component does not effect the working of micro kernel.	Failure of one component in a monolithic kernel leads to the failure of the entire system.
7.	Processing Speed	Execution speed is low.	Execution speed is high.
8.	Extend	It is easy to extend Microkernel.	It is not easy to extend monolithic kernel.
9.	Communication	To implement IPC messaging queues are used by the communication microkernels.	Signals and Sockets are utilized to implement IPC in monolithic kernels.
10.	Debugging	Debugging is simple.	Debugging is difficult.
11.	Maintain	It is simple to maintain.	Extra time and resources are needed for maintenance.
12.	Message passing and Context switching	Message forwarding and context switching are required by the microkernel.	Message passing and context switching are not required while the kernel is working.
13.	Services	The kernel only offers IPC and low-level device management services.	The Kernel contains all of the operating system's services.
14.	Example	Example : Mac OS.	Example : Microsoft Windows 95.

19. What is the difference between internal and external fragmentation?

Internal fragmentation	External fragmentation
In internal fragmentation fixed-sized memory, blocks square measure appointed to process.	In external fragmentation, variable-sized memory blocks square measure appointed to the method.
Internal fragmentation happens when the method or process is smaller than the memory.	External fragmentation happens when the method or process is removed.
The solution of internal fragmentation is the best-fit block .	The solution to external fragmentation is compaction and paging .
Internal fragmentation occurs when memory is divided into fixed-sized partitions .	External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.

The difference between memory allocated and required space or memory is called Internal fragmentation.	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, which is called External fragmentation.
Internal fragmentation occurs with paging and fixed partitioning.	External fragmentation occurs with segmentation and dynamic partitioning .
It occurs on the allocation of a process to a partition greater than the process's requirement. The leftover space causes degradation system performance.	It occurs on the allocation of a process to a partition greater which is exactly the same memory space as it is required.
It occurs in worst fit memory allocation method .	It occurs in best fit and first fit memory allocation method.

20. How does an operating system manage I/O operations?

An operating system manages I/O operations by controlling and coordinating the use of the hardware among the various system and application programs.

In more detail, the operating system (OS) plays a crucial role in managing input/output (I/O) operations. It acts as an intermediary between users and the computer hardware, ensuring that all I/O operations are carried out efficiently and effectively. The OS is responsible for managing all the I/O devices and the I/O operations in a computer system. For a deeper understanding of these responsibilities, see the section on the functions of operating systems.

The OS manages I/O operations through a component known as the I/O subsystem. The I/O subsystem is designed to provide a uniform interface for any I/O device irrespective of its underlying specifics. It is responsible for monitoring the status of operations, managing buffers and caches, and providing error handling mechanisms.

The OS uses device drivers to communicate with the hardware devices. A device driver is a specific type of software that controls a specific type of hardware device. It acts as a translator between the device and the applications or operating system that use it. Each device has its own set of specialised commands that only its driver knows. In other words, the device driver knows the language of the device and how to control it.

The OS also uses interrupt techniques to manage I/O operations. An interrupt is a signal sent to the processor that an event has occurred that needs immediate attention. When an I/O operation is completed, the device sends an interrupt to the processor. The OS then stops its current operations and starts addressing the interrupt. This allows the OS to manage multiple I/O operations simultaneously, improving the overall efficiency of the system.

Buffering is another technique used by the OS to manage I/O operations. A buffer is a

temporary storage area in memory where data is held before it is sent to the device. Buffering helps to cope with the speed mismatch between the processor and the I/O devices. It allows the processor to move on to other tasks while the slower I/O operation is still in progress. To explore more about how the OS optimises the management of these resources, you might want to read about the role of the operating system in resource management.

Understanding the overarching purpose of operating systems can also provide additional context to the myriad ways in which these systems manage not only I/O operations but also other crucial system functionalities.

21. Explain the difference between preemptive and non-preemptive scheduling.

- In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.
- The executing process in preemptive scheduling is interrupted in the middle of execution when a higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.
- In Preemptive Scheduling, there is the overhead of switching the process from the ready state to the running state, vise-verse, and maintaining the ready queue. Whereas in the case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.
- In preemptive scheduling, if a high-priorThe process The process non-preemptive low-priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. , in non-preemptive scheduling, if CPU is allocated to the process having a larger burst time then the processes with a small burst time may have to starve.
- Preemptive scheduling attains flexibility by allowing the critical processes to access the CPU as they arrive in the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
- Preemptive Scheduling has to maintain the integrity of shared data that's why it is cost associative which is not the case with Non-preemptive Scheduling.

Parameter	PREEMPTIVE SCHEDULING	NON-PREEMPTIVE SCHEDULING
Basic	In this resources(CPU Cycle) are allocated to a process for a limited time.	Once resources(CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting state
Interrupt	Process can be interrupted in between.	Process can not be interrupted until it terminates itself or its time is up

Starvation	If a process having high priority frequently arrives in the ready queue, a low priority process may starve	If a process with a long burst time is running CPU, then later coming process with less CPU burst time may starve
Overhead	It has overheads of scheduling the processes	It does not have overheads
Flexibility	flexible	Rigid
Cost	Cost associated	No cost associated
CPU Utilization	In preemptive scheduling, CPU utilization is high	It is low in non preemptive scheduling
Waiting Time	Preemptive scheduling waiting time is less	Non-preemptive scheduling waiting time is high
Response Time	Preemptive scheduling response time is less	Non-preemptive scheduling response time is high
Decision making	Decisions are made by the scheduler and are based on priority and time slice allocation	Decisions are made by the process itself and the OS just follows the process's instructions
Process control	The OS has greater control over the scheduling of processes	The OS has less control over the scheduling of processes
Overhead	Higher overhead due to frequent context switching	Lower overhead since context switching is less frequent
Examples	Examples of preemptive scheduling are Round Robin and Shortest Remaining Time First	Examples of non-preemptive scheduling are First Come First Serve and Shortest Job First

22. What is round-robin scheduling, and how does it work?

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of the First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique.

The period of time for which a process or job is allowed to run in a pre-emptive method is called time quantum.

Each process or job present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will end else the process will go back to the waiting table and wait for its next turn to complete the execution.

How does the Round Robin Algorithm Work?

- All the processes are added to the ready queue.
- At first, The burst time of every process is compared to the time quantum of the CPU.
- If the burst time of the process is less than or equal to the time quantum in the round-robin scheduling algorithm, the process is executed to its burst time.
- If the burst time of the process is greater than the time quantum, the process is executed up to the time quantum (TQ).
- When the time quantum expires, it checks if the process is executed completely or not.
- On completion, the process terminates. Otherwise, it goes back again to the ready state.

23. Describe the priority scheduling algorithm. How is priority assigned to processes?

In Priority scheduling, there is a priority number assigned to each process. In some systems, the lower the number, the higher the priority. While, in the others, the higher the number, the higher will be the priority. The Process with the higher priority among the available processes is given the CPU. There are two types of priority scheduling algorithm exists. One is Preemptive priority scheduling while the other is Non Preemptive Priority scheduling.

The priority number assigned to each of the process may or may not vary. If the priority number doesn't change itself throughout the process, it is called static priority, while if it keeps changing itself at the regular intervals, it is called dynamic priority.

24. What is the shortest job next (SJN) scheduling algorithm, and when is it used?

In Shortest Job Next(SJN), when choosing the next job to run, look at all the processes in the ready state and dispatch the one with the smallest service time. In this case, we need to know the service times at any given point in time. It is a non-preemptive algorithm. A new job will not be given a chance at the CPU until the current job finishes even if the new job is shorter.

The Shortest Job Next (SJN) scheduling algorithm is like a line at a grocery store where the cashier decides to serve the person with the fewest items first, so they get out of the way quickly. In a computer, when multiple tasks or processes are waiting to be done, SJN looks at all of them and picks the one that will take the least time to complete. This way, the system can quickly finish and move on to the next task, making everything run more efficiently. SJN is often used in situations where it's important to reduce the time tasks spend waiting in line, like in systems where many small, quick tasks need to be processed as fast as possible.

25. Explain the concept of multilevel queue scheduling.

It may happen that processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a foreground (interactive) process and a background (batch) process. These two classes have different

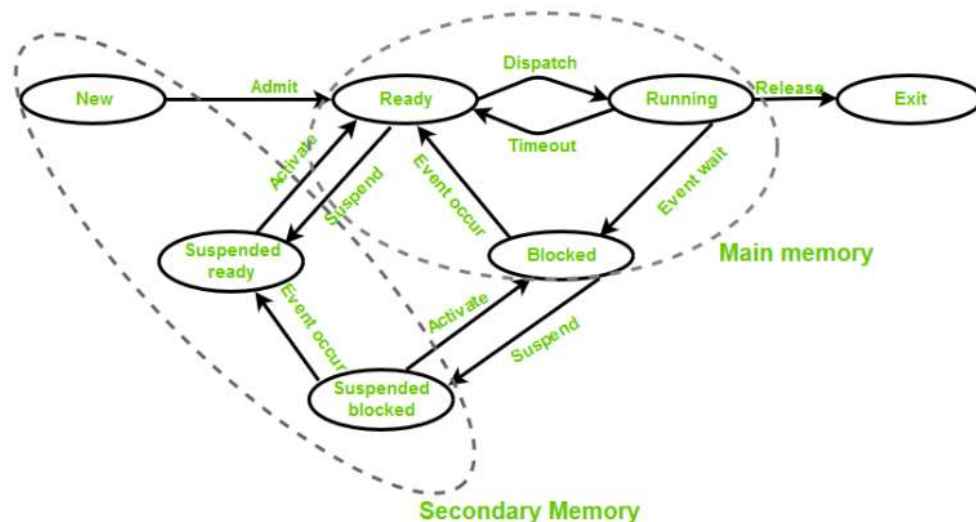
scheduling needs. For this kind of situation, Multilevel Queue Scheduling is used.

26. What is a process control block (PCB), and what information does it contain?

While creating a process, the operating system performs several operations. To identify the processes, it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc.

All this information is required and must be saved when the process is switched from one state to another. When the process makes a transition from one state to another, the operating system must update information in the process's PCB. A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCBs, which logically contains a PCB for all of the current processes in the system.

27. Describe the process state diagram and the transitions between different process states.



A process can move between different states in an operating system based on its execution status and resource availability. Here are some examples of how a process can move between different states:

New to Ready: When a process is created, it is in a new state. It moves to the ready state when the operating system has allocated resources to it and it is ready to be executed.

Ready to Running: When the CPU becomes available, the operating system selects a process

from the ready queue depending on various scheduling algorithms and moves it to the running state.

Running to Blocked: When a process needs to wait for an event to occur (I/O operation or system call), it moves to the blocked state. For example, if a process needs to wait for user input, it moves to the blocked state until the user provides the input.

Running to Ready: When a running process is preempted by the operating system, it moves to the ready state. For example, if a higher-priority process becomes ready, the operating system may preempt the running process and move it to the ready state.

Blocked to Ready: When the event a blocked process was waiting for occurs, the process moves to the ready state. For example, if a process was waiting for user input and the input is provided, it moves to the ready state.

Running to Terminated: When a process completes its execution or is terminated by the operating system, it moves to the terminated state.

28. How does a process communicate with another process in an operating system?

Process communication in an operating system is the exchange of data among multiple processes within a computing system.

In a computer system, multiple processes often run concurrently, and these processes may need to share data or information. This is where process communication comes into play. It is a mechanism that allows processes to communicate and synchronise their actions. The communication between these processes can be performed using shared memory or message passing.

In shared memory, a common region of memory is shared by multiple processes. Each process can read from or write to this shared memory area. This method is fast and efficient as it allows direct access to the data. However, it requires careful management to avoid conflicts and inconsistencies, especially when multiple processes are trying to write to the same area of memory at the same time.

On the other hand, message passing involves processes communicating by sending and receiving messages. These messages can be sent directly from one process to another, or they can be sent via a communication channel, such as a queue. Message passing is more controlled and less prone to conflicts than shared memory, but it can be slower due to the overhead of sending and receiving messages.

Process communication is crucial in operating systems as it allows for the coordination of activities among different processes. For example, in a printing task, one process might be responsible for receiving the print command, another for formatting the document, and another for sending the document to the printer. These processes need to communicate to ensure the task is completed correctly. It is essential to understand the broader functions of operating systems to appreciate how they facilitate this coordination.

Furthermore, process communication is also essential in distributed systems, where multiple computers work together to perform a task. In such systems, processes running on different machines need to exchange data and coordinate their actions, which is achieved through process communication. This interaction between different systems highlights the importance of resource management by the operating system. Moreover, the principles underlying these interactions are critical in both centralised and distributed systems.

29. What is process synchronization, and why is it important?

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.

The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems. Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.

Advantages of Process Synchronization

- Ensures data consistency and integrity
- Avoids race conditions
- Prevents inconsistent data due to concurrent access
- Supports efficient and effective use of shared resources

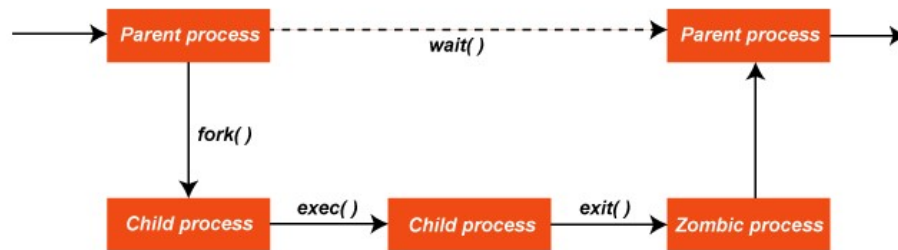
30. Explain the concept of a zombie process and how it is created.

A zombie process or defunct process is a process that has completed execution (via the exit system call) but still has an entry in the process table. This occurs for the child processes, where the entry is still needed to allow the parent process to read its child's exit status. Once the exit status is read via the wait system call, the zombie's entry is removed from the process table and said to be "reaped". A child process always first becomes a zombie before being removed from the resource table.

In most cases, zombies are immediately waited on by their parents and then reaped by the system under normal system operation. Processes that stay zombies for a long time are

generally an error and cause a resource leak, but they only occupy the process table entry.

In the term's metaphor, the child process has died but has not yet been reaped. Also, unlike normal processes, the kill command does not affect a zombie process.



What is Zombie Process

Zombie processes should not be confused with orphan processes. An orphan process is a process that is still executing, but whose parent has died. When the parent dies, the orphaned child process is adopted by init (process ID 1). When orphan processes die, they do not remain as zombie processes; instead, they are waited on by init. The result is that a process that is both a zombie and an orphan will be reaped automatically.

31. Describe the difference between internal fragmentation and external fragmentation.

=> Already answered

32. What is demand paging, and how does it improve memory management efficiency?

Demand paging is a way for a computer to manage its memory more efficiently by only loading parts of a program into memory when they're actually needed, rather than loading the entire program at once. Imagine you have a huge book, but instead of carrying the whole book around, you only take out the pages you're going to read at that moment. Similarly, demand paging keeps the computer's memory less crowded by only bringing in the parts of a program that are actively being used. This allows the system to run larger programs or multiple programs at once without running out of memory, improving overall efficiency. If a program tries to access a part that's not yet in memory, the system quickly retrieves it from storage, so it seems like everything is there when you need it.

33. Explain the role of the page table in virtual memory management.

The page table in virtual memory management acts like a map or a directory that helps the computer keep track of where pieces of data are stored. Imagine your computer's memory as a big, organized storage area with many sections. When a program runs, it might not fit entirely in the available memory, so the computer breaks it into smaller chunks called "pages" and stores them wherever there's space, even on a different storage device like a hard drive. The

page table is used to keep track of where each of these pages is stored, translating the program's request for data into the correct memory location. This way, even if parts of the program are scattered across different areas, the page table ensures the program can access everything it needs seamlessly, making it seem like all the data is right there in the computer's main memory.

34. How does a memory management unit (MMU) work?

A Memory Management Unit (MMU) is a hardware component in a computer that helps manage and organize memory. Its main job is to translate virtual addresses used by programs into physical addresses in the computer's memory.

Here's how it works: When a program runs, it uses virtual addresses, which are like imaginary locations in memory. The MMU takes these virtual addresses and converts them into real, physical addresses where the data is actually stored in the computer's RAM. This translation allows programs to run without worrying about the actual physical location of their data. The MMU also handles tasks like checking permissions (to make sure a program can access certain memory) and supporting features like demand paging by keeping track of which parts of memory are in use. This helps the computer efficiently manage memory, protect data, and allow multiple programs to run simultaneously without interfering with each other.

35. What is thrashing, and how can it be avoided in virtual memory systems?

=> Already answered

36. What is a system call, and how does it facilitate communication between user programs and the operating system?

A system call is like a special request a user program makes to the operating system when it needs to do something that requires the OS's help, such as accessing hardware, managing files, or communicating with other programs.

Imagine you're in an office and you need to use the printer, but only the office manager (the operating system) has access to it. You'd have to ask the manager (through a system call) to print your document for you. Similarly, when a program needs to perform tasks like reading from a file, sending data over the network, or creating new processes, it makes a system call to ask the operating system to handle these tasks. The OS then executes the request in a controlled and safe way, ensuring that the program can get what it needs without causing any problems for other programs or the system itself. System calls are essential because they allow user programs to interact with the hardware and system resources while keeping everything secure and well-managed.

37. Describe the difference between a monolithic kernel and a microkernel.

=> Already answered

38. How does an operating system handle I/O operations?

=> Already answered

39. Explain the concept of a race condition and how it can be prevented.

A race condition occurs when two or more processes or threads try to access and modify shared data or resources at the same time, leading to unexpected or incorrect outcomes. Imagine two people trying to edit the same document at the same time without knowing what the other is doing. If they both make changes simultaneously, the final version of the document might be a messy mix of their edits, or one person's changes might overwrite the other's, causing errors.

To prevent race conditions, synchronization techniques are used to ensure that only one process or thread can access the shared resource at a time. This is done using mechanisms like locks, semaphores, or monitors, which essentially put up a "do not disturb" sign when a process is working with the shared resource. By carefully controlling access, the system makes sure that processes take turns modifying shared data, preventing conflicts and ensuring that everything works correctly.

40. Describe the role of device drivers in an operating system.

=> Already answered

41. What is a zombie process, and how does it occur? How can a zombie process be prevented?

A zombie process is a process that has completed its execution but still has an entry in the process table, which means it's still occupying system resources. This happens because the process's parent hasn't read its exit status yet, so the operating system keeps the process in a "zombie" state.

To prevent zombie processes, the parent process should properly "reap" the terminated child processes. This can be done by the parent process using the `wait()` or `waitpid()` system calls, which allow the parent to retrieve the exit status of the child process and free up its resources. If a parent process is not interested in the exit status of its children, it can handle the `SIGCHLD` signal, which is sent when a child process terminates, to automatically reap the child processes without needing to explicitly call `wait()` or `waitpid()`.

Another approach is to design the parent process to regularly check for and clean up zombie processes, ensuring they don't linger and consume resources unnecessarily.

42. Explain the concept of an orphan process. How does an operating system handle orphan processes?

An orphan process is a process whose parent has terminated before it did, leaving it without a parent. In such cases, the orphan process is still running but lacks the parent process that originally created it.

The operating system handles orphan processes by assigning them a new parent, typically the ``init`` process (or its modern equivalent, such as ``systemd`` in many systems). The ``init`` process is the root process of the system, and it's responsible for reaping orphan processes. When an orphan process is adopted by ``init``, it ensures that the process can continue running and eventually complete its execution properly.

By adopting orphan processes, ``init`` or the `systemd` ensures that these processes are properly managed and cleaned up, preventing them from becoming zombies and occupying system resources indefinitely. This process of adopting orphans helps maintain system stability and resource management.

43. What is the relationship between a parent process and a child process in the context of process management?

In process management, a parent process is a process that creates one or more child processes. The relationship between them is somewhat hierarchical: the parent process starts the child processes, and the child processes are essentially sub-tasks or extensions of the parent.

Here's how they interact: When a parent process creates a child process, it often sets up the initial conditions or context for the child, and then the child process runs independently, performing its specific tasks. The parent process can control or communicate with its child processes, such as waiting for them to finish, sending signals, or retrieving their exit statuses. This relationship helps in organizing tasks, managing resources, and maintaining control over multiple processes in a structured way.

The operating system keeps track of this relationship using process identifiers (PIDs) and maintains a process hierarchy. If a parent process terminates, its child processes are reassigned to the ``init`` process (or its modern equivalent) to ensure they continue running or are properly cleaned up.

44. How does the `fork()` system call work in creating a new process in Unix-like operating systems?

In Unix-like operating systems, the `fork()` system call is used to create a new process. Here's how it works in simple terms:

When a process calls `fork()`, the operating system creates a new process, known as the child process. This child process is a duplicate of the calling process, known as the parent process. The new process inherits most of the parent's attributes, such as file descriptors and memory layout, but it gets its own unique process identifier (PID).

The `fork()` call returns twice: once in the parent process and once in the child process. In the parent process, `fork()` returns the PID of the child process, while in the child process, it returns `0`. This allows both processes to determine which one they are and to execute different code based on this information.

The child process starts execution from the point where `fork()` was called, and from then on, both processes run independently. The parent and child processes can then perform different tasks or coordinate their actions, depending on their specific needs.

45. Describe how a parent process can wait for a child process to finish execution.

A parent process can wait for a child process to finish execution using system calls like `wait()` or `waitpid()`. Here's how it works:

When a parent process creates a child process using `fork()`, it can call `wait()` to pause its own execution until one of its child processes completes. The `wait()` system call makes the parent process wait until a child process terminates, and it collects the exit status of the child process. This status can provide information about how the child process ended, such as whether it finished successfully or encountered an error.

If the parent process wants to wait for a specific child process or handle multiple child processes, it can use `waitpid()`. This system call allows more control, such as waiting for a particular child process by specifying its PID or using various options to handle multiple children or non-blocking waits.

By using these system calls, the parent process ensures that it properly cleans up resources associated with the child process and can handle any results or errors reported by the child.

46. What is the significance of the exit status of a child process in the `wait()` system call?

The exit status of a child process is significant in the `wait()` system call because it provides important information about how the child process ended. When a child process terminates, it returns an exit status code to the operating system. This status code can indicate whether the process completed successfully or encountered an error.

the exit status matters because:

Error Handling: The exit status helps the parent process determine if the child process completed its task successfully or if it failed. For instance, a status of 0 usually means success, while non-zero values typically indicate different types of errors or failure conditions.

Resource Management: By retrieving the exit status, the parent process can properly handle any cleanup or resource management required after the child process terminates.

Process Coordination: The exit status can be used by the parent process to make decisions about subsequent actions. For example, if a child process is responsible for a specific task and it fails, the parent might decide to retry the task, log an error, or take alternative actions.

47. How can a parent process terminate a child process in Unix-like operating systems?

A parent process can terminate a child process in Unix-like operating systems using the `kill()` system call.

Sending Signals: The `kill()` system call is used to send signals to processes. The parent process can send a termination signal, typically `SIGTERM` (signal 15), which requests the child process to terminate gracefully. If the child process doesn't respond to `SIGTERM`, the parent can send a stronger signal, `SIGKILL` (signal 9), which forces the child process to terminate immediately without cleaning up.

Identifying the Child: The parent process needs to know the Process ID (PID) of the child process to send the signal. This PID is obtained when the child process is created, usually through the `fork()` system call.

Using `kill()`: The parent process calls `kill(pid, signal)`, where `pid` is the Process ID of the child, and `signal` is the signal to be sent. For example, `kill(child_pid, SIGTERM)` requests a graceful termination, while `kill(child_pid, SIGKILL)` forces immediate termination.

48. Explain the difference between a process group and a session in Unix-like operating systems.

Aspect	Process Group	Session
Definition	A collection of related processes that are managed together.	A collection of one or more process groups that share common attributes and control.
Purpose	Used to manage and control related processes, especially for job control and signal handling.	Used to manage a set of processes with a common control terminal and to handle process group leadership.

Creation	Created by a process using the <code>setpgid()</code> system call or implicitly when a new process is created with <code>fork()</code> .	Created by a process using the <code>setsid()</code> system call.
Control	Allows sending signals to all processes in the group using the <code>killpg()</code> system call.	Provides a mechanism to manage process groups as a unit, especially for terminal handling.
Leader	The process that creates a process group is the group leader.	The process that creates a session is the session leader.
Session Leader	Not necessarily related to process groups. A process group leader does not need to be a session leader.	The session leader is the initial process of a session and is usually the first process in that session.
Terminal	Process groups within a session can be associated with a controlling terminal.	The session has a controlling terminal that is shared among its process groups.
Terminating	Killing a process group affects all processes within that group.	Killing the session leader does not terminate the entire session but may affect the controlling terminal and its process groups.

49. Describe how the `exec()` family of functions is used to replace the current process image with a new one.

The `exec()` family of functions in Unix-like operating systems is used to completely replace the current running process with a new one. Imagine you're reading a book, and suddenly you decide to switch to a completely different book—everything about the old book, including its pages and storyline, is discarded, and you start fresh with the new book. Similarly, when a process calls one of the `exec()` functions, it abandons its current program and replaces it with a new program. This means the process gets a fresh start with the new program's code, data, and execution context, while keeping its original process ID. The old program is entirely gone, and the new program takes over as if it had been running all along.

50. What is the purpose of the `waitpid()` system call in process management? How does it differ from `wait()`?

The `waitpid()` system call in process management is used by a parent process to wait for a specific child process to finish, or to handle various child processes based on certain conditions. Imagine a parent who wants to check on a particular child at school to see if they have completed their homework, rather than just waiting for any child to come home. The `waitpid()` function lets the parent specify which child process they are interested in, allowing them to get precise information about that child's completion status. In contrast, the `wait()` function simply waits for any child process to finish without distinguishing between them. So,

``waitpid()`) offers more control and flexibility, letting the parent process manage specific children more efficiently.`

51. How does process termination occur in Unix-like operating systems?

In Unix-like operating systems, process termination is like wrapping up a job and leaving the workplace. When a process finishes its tasks, it can either end voluntarily by calling an exit function or be forced to stop by receiving a termination signal. Once a process terminates, the operating system cleans up its resources, such as memory and file handles. If the process has children, the operating system ensures they are taken care of—either by waiting for them to finish or reassigning them to a new "parent" process like ``init``. The process's status is then recorded so that its parent process can check how it ended, and finally, the system removes any remaining traces of the process, making sure everything is tidy and ready for the next task.

52. What is the role of the long-term scheduler in the process scheduling hierarchy? How does it influence the degree of multiprogramming in an operating system?

The long-term scheduler, also known as the admission scheduler, is like a gatekeeper that decides which processes get to enter the computer's main memory and start running. It manages the queue of processes waiting to be executed, selecting which ones should be loaded into memory from a pool of processes that are waiting in storage. By controlling how many processes are brought into memory at any given time, the long-term scheduler influences the degree of multiprogramming, which is the number of processes that can be kept in memory and run concurrently. If the long-term scheduler allows more processes to enter memory, it increases multiprogramming, making better use of system resources. Conversely, if it limits the number of processes, it decreases multiprogramming, potentially leaving some resources underutilized.

53. How does the short-term scheduler differ from the long-term and medium-term schedulers in terms of frequency of execution and the scope of its decisions?

Short-Term Scheduler: This scheduler operates very frequently, often multiple times per second. Its main job is to make quick decisions about which process in the ready queue (the list of processes waiting to use the CPU) should run next. It handles short-term decisions about CPU time allocation, ensuring that processes get a fair chance to use the CPU and that system responsiveness is maintained.

Long-Term Scheduler: This scheduler runs less frequently, usually less often than the short-term scheduler. Its role is to manage which processes are admitted into memory from the pool of processes waiting on disk or other storage. It controls the degree of multiprogramming by deciding which processes should be loaded into main memory based on factors like system load and resource availability.

Medium-Term Scheduler: This scheduler operates between the long-term and short-term

schedulers and manages processes that are in memory but may not be actively running. It handles the swapping of processes between main memory and disk, deciding which processes should be temporarily moved out of memory to free up space for other processes. Its frequency of execution is more moderate, adjusting memory allocation based on current needs and system performance.

54. Describe a scenario where the medium-term scheduler would be invoked and explain how it helps manage system resources more efficiently.

Imagine you're working on a computer with several programs open, like a web browser, a word processor, and a video editor. The computer has limited memory, and as you start using the video editor more intensively, it needs more memory than what's currently available. The medium-term scheduler steps in to manage this situation by temporarily moving some of the less active programs (like the web browser or word processor) from memory to a slower storage area like the hard drive. This frees up memory for the video editor to run smoothly. When you switch back to those less active programs, the medium-term scheduler will bring them back into memory and might swap out the video editor if it's not needed at that moment. By managing which programs are in memory and which are stored elsewhere, the medium-term scheduler helps ensure that the system runs efficiently, making the best use of available resources without overloading the memory.