

## UNIT-III PROCESS SYSTEM

### Q-1: Elaborate Process states and transitions in process system with example.

#### Process:

It is an instance of a program in execution. A set of processes combined together make a complete program. There are two categories of processes in Unix, namely

- **User processes** : They are operated in user mode.
- **Kernel processes** : They are operated in kernel mode.

#### Process States:

The lifetime of a process can be divided into a set of states, each with certain characteristics that describe the process. It is essential to understand the following states now:

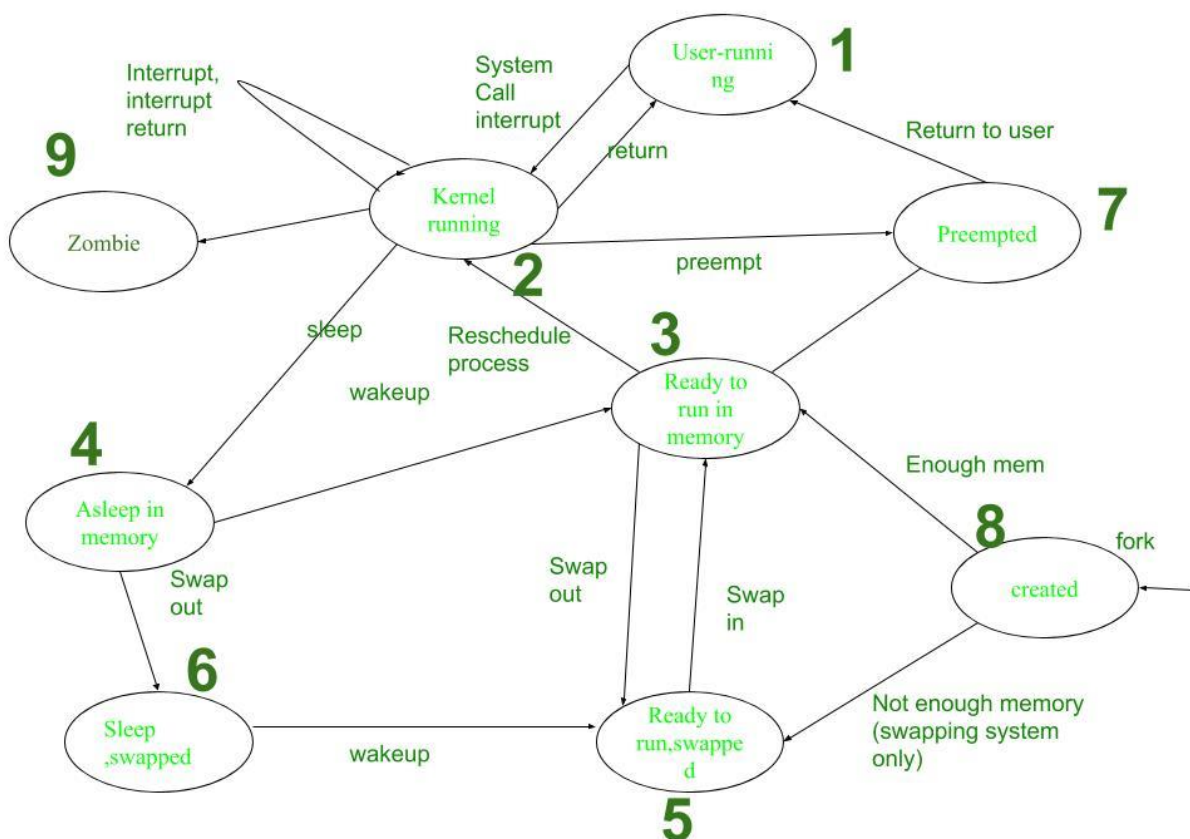
- The process is currently executing in user mode.
- The process is currently executing in kernel mode.
- The process is not executing, but it is ready to run as soon as the scheduler chooses it. Many processes may be in this state, and the scheduling algorithm determines which one will execute next.
- The process is sleeping. A process puts itself to sleep when it can no longer continue executing, such as when it is waiting for I/O to complete.
- The process is ready to run, but the swapper(process 0) must swap the process into main memory before the kernel can schedule it to execute.
- The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.
- The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process. The distinction between this state and state 3("ready to run") will be brought out shortly.
- The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. This state is the start state for all processes except process 0.
- The process executed the exit system call and is in the zombie state. The process no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state is the final state of a process.
- Because a processor can execute only one process at a time, at most one process may be in states 1 and 2. The two states correspond to the two modes of execution, user and kernel.

#### Process Transitions:

The working of Process is explained in following steps:

- **User-running:** Process is in user-running.
- **Kernel-running:** Process is allocated to kernel and hence, is in kernel mode.

- **Ready to run in memory:** Further, after processing in main memory process is rescheduled to the Kernel. i.e. The process is not executing but is ready to run as soon as the kernel schedules it.
- **Asleep in memory:** Process is sleeping but resides in main memory. It is waiting for the task to begin.
- **Ready to run, swapped:** Process is ready to run and be swapped by the processor into main memory, thereby allowing kernel to schedule it for execution.
- **Sleep, Swapped:** Process is in sleep state in secondary memory, making space for execution of other processes in main memory. It may resume once the task is fulfilled.
- **Pre-empted:** Kernel preempts an on-going process for allocation of another process, while the first process is moving from kernel to user mode.
- **Created:** Process is newly created but not running. This is the start state for all processes.
- **Zombie:** Process has been executed thoroughly and exit call has been enabled. The process, thereby, no longer exists. But, it stores a statistical record for the process. This is the final state of all processes.



## Example:

### 1. New (Created):

- A process is being created. Memory and resources are allocated.
- **Background = unattended jobs** (logging, backups, monitoring, long downloads).
- **Execute:** `./samplefile.sh > download.log 2>&1 &`

(2>&1 means: send errors (stderr) to the same place as stdout)

```
/usr/bin/bash --login -i
dell@Angel MINGW64 ~$ nano samplefile.sh
dell@Angel MINGW64 ~$ chmod +x samplefile.sh
dell@Angel MINGW64 ~$ ./samplefile.sh > download.log 2>&1 &
[3] 944
dell@Angel MINGW64 ~$ cat download.log
Download started at Sun Aug 24 21:42:57 IST 2025
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   Dload  Upload   Total             Spent    Left     Speed
  0   0   0    0    0    0     0      0  --:--:--  0:00:02 --:--:--    0
curl: (35) schannel: next InitializeSecurityContext failed: SEC_E_CERT_EXPIRED (
0x80090328) - The received certificate has expired.
Download finished at Sun Aug 24 21:43:02 IST 2025
[3]-  Done                  ./samplefile.sh > download.log 2>&1
dell@Angel MINGW64 ~$ |
```

- **Foreground = interactive jobs** (taking input, asking user decisions).

```
dell@Angel MINGW64 ~$ nano script2.sh
dell@Angel MINGW64 ~$ chmod +x script2.sh
dell@Angel MINGW64 ~$ ./script2.sh
Enter a number:
18
18 is Even
dell@Angel MINGW64 ~$ |
```

### 2. Ready (Waiting in Queue):

- Process has all the resources except the **CPU**.
- It is waiting for CPU allocation by the scheduler.
- Example: Multiple programs waiting in the CPU ready queue.

```
dell@Angel MINGW64 ~$ nano download.sh
dell@Angel MINGW64 ~$ chmod +x download.sh
dell@Angel MINGW64 ~$ ./download.sh > download.log 2>&1 &
[3] 976
dell@Angel MINGW64 ~$ jobs
[2]+  Stopped                  ./script2.sh
[3]-  Running                  ./download.sh > download.log 2>&1 &
dell@Angel MINGW64 ~$ ps
  PID   PPID   PGID   WINPID   TTY      UID     STIME  COMMAND
  976     881    976    14796   pty0     197609  22:06:36 /usr/bin/bash
  880      1    880    13248   ?        197609  20:56:50 /usr/bin/mintty
  977     976    976    14264   pty0     197609  22:06:36 /usr/bin/sleep
  978     881    978    9932    pty0     197609  22:06:45 /usr/bin/ps
  881     880    881    14112   pty0     197609  20:56:50 /usr/bin/bash
  S    926     881    926    10784   pty0     197609  21:22:50 /usr/bin/bash
dell@Angel MINGW64 ~$ nano sample25.sh
dell@Angel MINGW64 ~$ chmod +x sample25.sh
dell@Angel MINGW64 ~$ ./sample25.sh > download.log 2>&1 &
[4] 981
dell@Angel MINGW64 ~$ jobs
[2]+  Stopped                  ./script2.sh
[3]-  Running                  ./download.sh > download.log 2>&1 &
[4]-  Running                  ./sample25.sh > download.log 2>&1 &
dell@Angel MINGW64 ~$ ps
  PID   PPID   PGID   WINPID   TTY      UID     STIME  COMMAND
  976     881    976    14796   pty0     197609  22:06:36 /usr/bin/bash
  880      1    880    13248   ?        197609  20:56:50 /usr/bin/mintty
  981     881    981    7248    pty0     197609  22:08:03 /usr/bin/bash
  977     976    976    14264   pty0     197609  22:06:36 /usr/bin/sleep
  984     881    984    12556   pty0     197609  22:08:17 /usr/bin/ps
  881     880    881    14112   pty0     197609  20:56:50 /usr/bin/bash
  S    926     881    926    10784   pty0     197609  21:22:50 /usr/bin/bash
  S    983     881    981    7084    pty0     197609  22:08:03 /usr/bin/sleep
dell@Angel MINGW64 ~$ |
```

The **kernel scheduler** will:

1. Give download.sh some CPU time.
2. Switch to sample25.sh after a short time slice.
3. Switch back to download.sh.
4. Repeat until both are done.
5. The nice command lets you start a process with a given priority.
6. Range: -20 (highest priority) to +19 (lowest priority).
7. Default = 0.
8. \$ nice -n 10 ./download.sh & # Lower priority
9. \$ nice -n -5 ./sample25.sh & # High priority
10. Higher-priority processes run before lower-priority ones.
11. \$ renice -n 5 -p 12345 #reschedule

### 3. Running:

- o Process is being executed on the CPU.
- o Only **one process per CPU core** can be in this state at a time.
- o Example: When your ./script.sh program is actually executing its code.

### 4. Blocked (Waiting / Waiting for I/O):

- o Process is waiting for some **event** (like I/O completion, resource availability).
- o Example: If your program tries to read a file and waits until the disk read is complete.

### 5. Terminated (Exit):

- o Process has finished execution. Resources are released.
- o Example: When your program reaches the return 0; Normal exit or killed. (e.g. kill 9 1234)

**Q-2: Describe the context of a process and saving the context of a process with detailed scenario.**

### **Context of a Process:**

The **context** of a process is all the information that the operating system (OS) needs to **suspend** a running process and later **resume** it correctly.

When the CPU switches from one process to another (**context switch**), the OS saves the old process's context and loads the new one.

#### **1. CPU Context**

Information directly related to the CPU execution state.

- **Program Counter (PC):**
  - Address of the **next instruction** to execute.
  - Essential for resuming execution at the correct point.
- **CPU Registers:**
  - General-purpose registers (e.g., accumulator, index registers).
  - Stack Pointer (SP): points to the process's current stack frame.
  - Status/Flag registers (carry, zero, sign, interrupt flags).

Example: If a process was performing an addition, the intermediate values in registers must be saved.

#### **2. Memory Context**

Information about how the process's memory is organized.

- **Base and Limit Registers:** define the memory boundaries of the process.
- **Page Table / Segment Table:** mapping of virtual addresses to physical memory.
- **Stack and Heap Pointers:** keep track of function calls and dynamic memory.

Example: If a process has allocated memory on the heap (malloc in C), the OS must remember where it is.

#### **3. Kernel / System Context**

Information stored in the **Process Control Block (PCB)** by the OS.

- **Process State:** (Ready, Running, Waiting, Terminated).
- **Process ID (PID).**
- **Scheduling Information:** priority, time quantum, nice value.
- **Accounting Information:** CPU time used, user/system time.
- **I/O Information:** open file descriptors, pending I/O requests.
- **Signal Information:** pending signals, handlers.

Example: If a process opened a file (cat file.txt), the OS must remember the file descriptor so it can continue I/O later.

**Example:**

\$ sleep 100 &

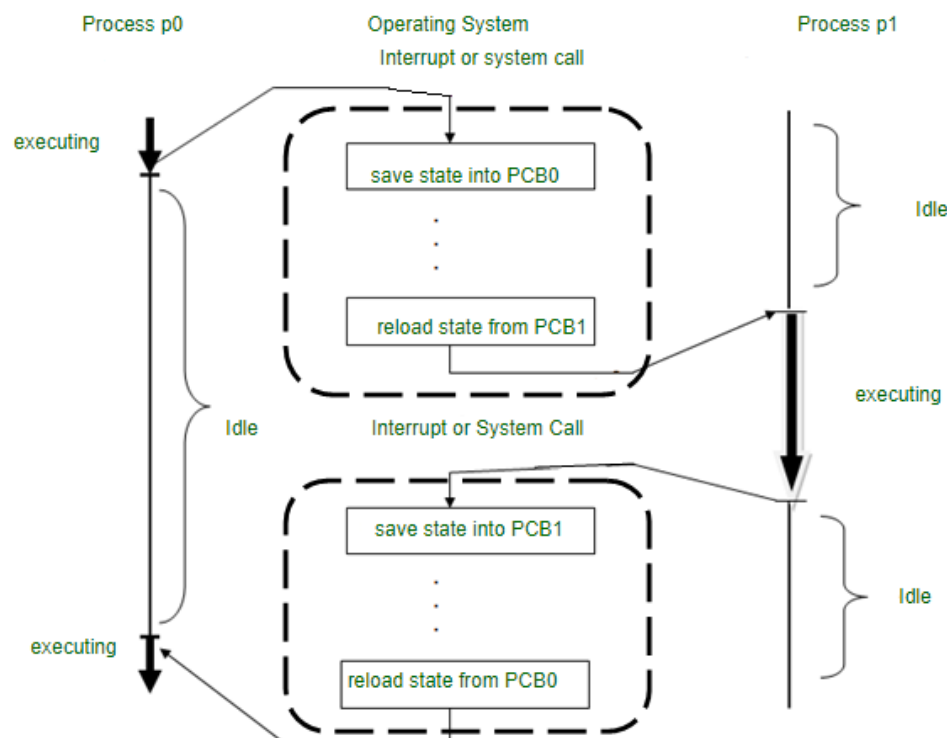
- The OS creates a process.
- Its **context** includes:
  - PID: e.g., 2345
  - Program counter: instruction in sleep
  - CPU registers: current values
  - State: Running/Ready
  - Open files: stdin, stdout, stderr

If another process is scheduled, the OS saves this info in the **Process Control Block (PCB)**, and later restores it when this process runs again.

**Saving the Context of a Process:**

When the CPU switches from one process to another (**context switch**), the **state of the currently running process** must be preserved so that it can resume later **from exactly the same point**.

This preservation is called **saving the context of a process**. The saved information is stored in the **Process Control Block (PCB)**.



The CPU alternates execution between two processes — p0 and p1 — through context switching.

**1. Execution Phase (Process p0)**

- Process p0 is running while p1 remains idle.
- At some point, an interrupt or system call occurs (e.g., a timer interrupt or I/O completion).

**2. Saving Current State (p0 → PCB0)**

- The operating system pauses p0.
- The current state of p0 (register values, program counter, etc.) is saved into its Process Control Block (PCB0).
- This ensures that p0 can later resume exactly where it left off.

**3. Loading New Process State (PCB1 → p1)**

- The OS retrieves the saved state of p1 from PCB1.
- This step restores p1's CPU context so it can continue execution from its last saved point.

**4. Execution Phase (Process p1)**

- Now, p1 is running while p0 is idle.
- Another interrupt or system call occurs, triggering another switch.

**5. Saving State of p1 (p1 → PCB1):** The OS saves the current state of p1 into PCB1.

**6. Reloading State of p0 (PCB0 → p0)**

- The saved state of p0 from PCB0 is loaded back into the CPU registers.
- p0 resumes execution right where it left off.

**Q-3: Explain the scenario of manipulating process address space with example.**

**Manipulating process address space** in Unix refers to the act of altering the memory layout of a process during its execution. It means **changing how a process uses its allocated memory regions** while it is running.

**Region Loading**

- A **region** is a contiguous area in a process's virtual address space (e.g., code, data, stack).
- The **Per-Process Region Table** keeps track of regions for each process.
- Each region entry contains:
  - **Page Table Address** → where the page table for this region is located.
  - **Process Virtual Address** → starting address in virtual memory.
  - **Size and Protection** → size of the region + access rights (read/write/execute).

When a process grows its memory (e.g., expanding stack or heap), the OS updates these region entries using **Growreg** (grow region).



## Shell Program:

```
#!/bin/bash
```

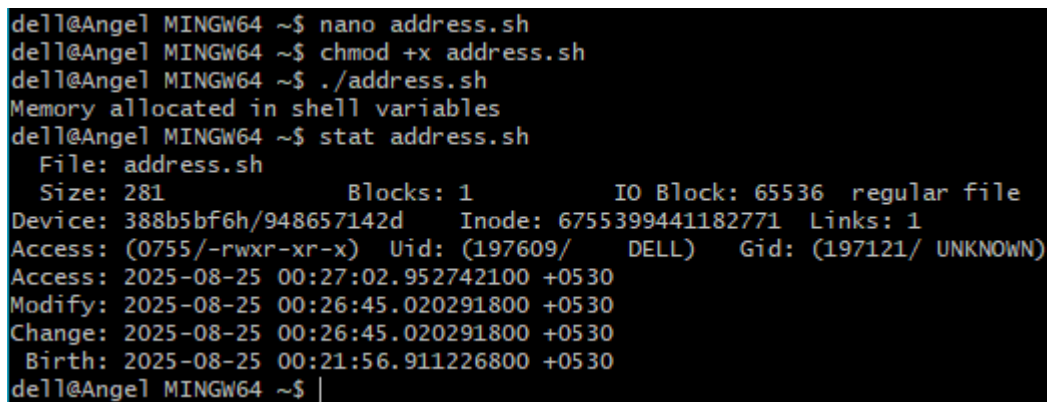
```
# Simulate memory allocation in shell
```

```
arr=$(head -c 1000 </dev/zero | tr '\0' 'a')           # Allocate ~1000 bytes
```

```
arr2=$(head -c 5000 </dev/zero | tr '\0' 'b')          # Allocate ~5000 bytes
```

```
echo "Memory allocated in shell variables"
```

```
sleep 5                                                  # Optional: prevent immediate release
```



```
dell@Angel MINGW64 ~$ nano address.sh
dell@Angel MINGW64 ~$ chmod +x address.sh
dell@Angel MINGW64 ~$ ./address.sh
Memory allocated in shell variables
dell@Angel MINGW64 ~$ stat address.sh
  File: address.sh
  Size: 281          Blocks: 1          IO Block: 65536   regular file
Device: 388b5bf6h/948657142d  Inode: 6755399441182771  Links: 1
Access: (0755/-rwxr-xr-x)  Uid: (197609/  DELL)   Gid: (197121/ UNKNOWN)
Access: 2025-08-25 00:27:02.952742100 +0530
Modify: 2025-08-25 00:26:45.020291800 +0530
Change: 2025-08-25 00:26:45.020291800 +0530
 Birth: 2025-08-25 00:21:56.911226800 +0530
dell@Angel MINGW64 ~$ |
```

```
# Check memory usage of this shell process using $stat address.sh
```

## Code Explanation:

- `head -c N </dev/zero` → generates N null bytes.
- `tr '\0' 'a'` → replaces null bytes with 'a', so the string is stored in the shell variable.
- The variables `arr` and `arr2` are now occupying memory in the shell process.
- `ps -o pid,rss,comm -p $$` → shows memory used by the current shell process (`$$` = PID of current shell).
- `sleep 5` → gives you time to check memory usage with another terminal if you want.

## Expanding Memory (Perform Arithmetic with Variables)

```
#!/bin/bash

a=15

b=25

echo "a = $a, b = $b"

echo "Sum: $((a + b))"

echo "Product: $((a * b))"
```



```
dell@Angel MINGW64 ~$ nano address.sh
dell@Angel MINGW64 ~$ ./address.sh
Memory allocated in shell variables
a = 15, b = 25
Sum: 40
Product: 375
dell@Angel MINGW64 ~$ stat address.sh
  File: address.sh
  Size: 374          Blocks: 1          IO Block: 65536   regular file
Device: 388b5bf6h/948657142d  Inode: 6755399441182771  Links: 1
Access: (0755/-rwxr-xr-x)  Uid: (197609/  DELL)   Gid: (197121/ UNKNOWN)
Access: 2025-08-25 00:33:18.390019500 +0530
Modify: 2025-08-25 00:33:05.163865800 +0530
Change: 2025-08-25 00:33:05.163865800 +0530
Birth: 2025-08-25 00:21:56.911226800 +0530
dell@Angel MINGW64 ~$ |
```

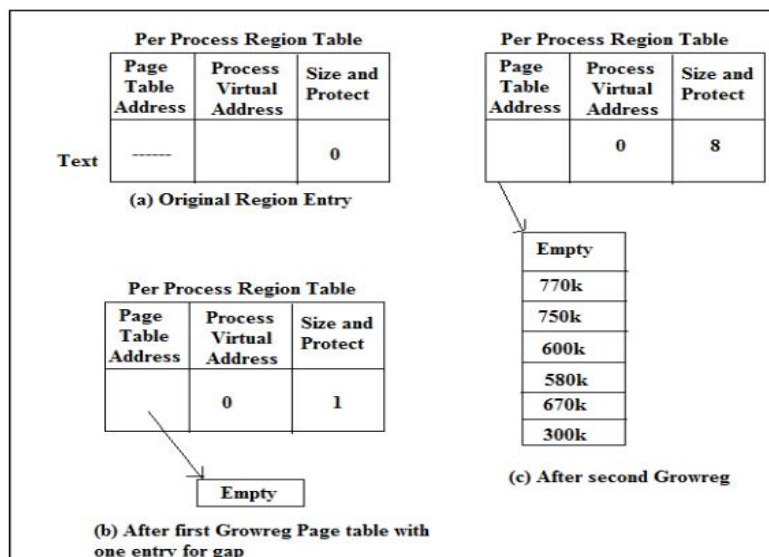


Fig. Loading a Region

### (a) Original Region Entry

- At the start, the process has **no pages allocated** for the region.
- The **Page Table Address** is empty (dashed).
- **Process Virtual Address** = 0 (base address).
- **Size** = 0 (region is not yet populated).

This means the process has a defined region but hasn't used any memory there yet.

### (b) After First Growreg

- The process requests more memory (for example, heap expansion).
- The OS uses the **Growreg operation** to allocate a page table for this region.
- Now:

- **Page Table Address = 0** → new page table assigned at base address 0.
  - **Process Virtual Address = 0** → still starts at address 0.
  - **Size = 1** → region now has 1 page.
- The page table contains **one entry** for the allocated page, and the rest remains empty.

This ensures the process has an allocated “gap” in memory for growth.

### (c) After Second Growreg

- The process again requests more memory (e.g., allocating arrays, stack expansion).
- The OS expands the region further using Growreg.
- Now:
  - **Page Table Address = 0** (same base, region still anchored at virtual address 0).
  - **Size = 8** → region now has 8 pages allocated.
  - The page table contains multiple valid entries, mapping process virtual addresses to **physical addresses**:
  - 770k
  - 750k
  - 600k
  - 580k
  - 670k
  - 300k

Each corresponds to a physical frame in memory.

- The **empty slots** represent unused space in the page table (still available for growth).

### Q-4: Extend process creation and termination with example.

#### Process:

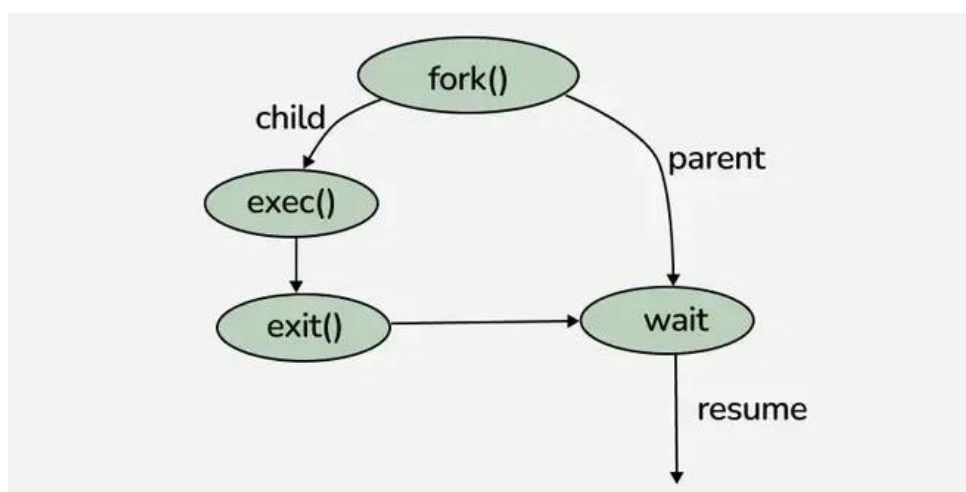
A process is an instance of a program running, and its lifecycle includes various stages such as creation, execution, and deletion.

- The operating system handles process creation by allocating necessary resources and assigning each process a unique identifier.
- Process deletion involves releasing resources once a process completes its execution.
- Processes are often organized in a hierarchy, where parent processes create child processes, forming a tree-like structure.
- A new process is always created by a parent process. The process that creates the new one is called the parent process, and the newly created process is called the

child process. A process can create multiple new processes while it's running by using system calls to create them.

### Process creation:

1. When a new process is created, the operating system assigns a unique Process Identifier (PID) to it and inserts a new entry in the primary process table.
2. Then required memory space for all the elements of the process such as program, data, and stack is allocated including space for its Process Control Block (PCB).
3. The process identification part is filled with PID assigned to it in step (1) and also its parent's PID.
4. The processor register values are mostly filled with zeroes, except for the stack pointer and program counter. The stack pointer is filled with the address of the stack-allocated to it in step (2) and the program counter is filled with the address of its program entry point.
5. The process state information would be set to 'New'.
6. Priority would be lowest by default, but the user can specify any priority during creation. Then the operating system will link this process to the scheduling queue and the process state would be changed from 'New' to 'Ready'. Now the process is competing for the CPU.
7. Additionally, the operating system will create some other data structures such as log files or accounting files to keep track of process activity.
8. The `fork()` system call creates a copy of the current process, including all its resources, but with just one thread.
9. The `exec()` system call replaces the current process's memory with the code and data from a specified executable file. It doesn't return; instead, it "transfers" the process to the new program.
10. The `waitpid()` function makes the parent process wait until a specific child process finishes executing.



## Process Deletion:

A parent may terminate a process due to one of the following reasons:

1. When task given to the child is not required now.
2. When the child has taken more resources than its limit.
3. The parent of the process is exiting, as a result, all its children are deleted. This is called cascaded termination.

A process can be terminated/deleted in many ways. Some of the ways are:

1. **Normal termination:** The process completes its task and calls an `exit()` system call. The operating system cleans up the resources used by the process and removes it from the process table.
2. **Abnormal termination/Error exit:** A process may terminate abnormally if it encounters an error or needs to stop immediately. This can happen through the `abort()` system call.
3. **Termination by parent process:** A parent process may terminate a child process when the child finishes its task. This is done by the using `kill()` system call.
4. **Termination by signal:** The parent process can also send specific signals like `SIGSTOP` to pause the child or `SIGKILL` to immediately terminate it.

## Q-5: Demonstrate various properties of signals with example.

- A **signal** is a software interrupt delivered to a process.
- It notifies the process that some event has occurred (like pressing `Ctrl+C`, illegal memory access, timer expiration, etc.).
- Signals are identified by numbers (e.g., 2 for `SIGINT`) and names (e.g., `SIGKILL`).

\$ `kill -l`

```
dell@Angel MINGW64 ~$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGEMT     8) SIGFPE      9) SIGKILL     10) SIGBUS
11) SIGSEGV    12) SIGSYS    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGURG     17) SIGSTOP   18) SIGTSTP    19) SIGCONT    20) SIGCHLD
21) SIGTTIN    22) SIGTTOU   23) SIGIO      24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH   29) SIGPWR     30) SIGUSR1
31) SIGUSR2    32) SIGRTMIN  33) SIGRTMIN+1 34) SIGRTMIN+2 35) SIGRTMIN+3
36) SIGRTMIN+4 37) SIGRTMIN+5 38) SIGRTMIN+6 39) SIGRTMIN+7 40) SIGRTMIN+8
41) SIGRTMIN+9 42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMIN+16 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4
61) SIGRTMAX-3 62) SIGRTMAX-2 63) SIGRTMAX-1 64) SIGRTMAX
dell@Angel MINGW64 ~$
```

Signal	Number	Meaning
SIGHUP	1	Hangup (terminal closed)
SIGINT	2	Interrupt (usually Ctrl+C)
SIGQUIT	3	Quit (usually Ctrl+\)
SIGKILL	9	Kill (force termination, cannot be caught/ignored)
SIGTERM	15	Terminate (default signal for kill)
SIGSTOP	19	Stop process (cannot be caught/ignored)
SIGCONT	18	Continue stopped process
SIGALRM	14	Alarm clock (timer expired)
SIGSEGV	11	Segmentation fault (invalid memory access)
SIGCHLD	17	Sent to parent when child terminates

## Sending Signals

You can send a signal to a process using the kill command:

```
kill -SIGTERM 1234      # send SIGTERM to process
kill -9 1234            # force kill (SIGKILL)
```

Example:

```
sleep 100 &
kill -2 <pid> # same as pressing Ctrl+C
```

## Handling Signals in Programs

Example: Shell Script Handling SIGINT (Ctrl+C)

```
#!/bin/bash
trap 'echo "Ctrl+C pressed, but ignored!"' SIGINT
echo "Running... press Ctrl+C"
while true; do
    sleep 1
done
```

- trap → defines how to handle a signal.
- Here, SIGINT is caught, and instead of terminating, it prints a message.

## Trapping Signals:

When you press the Ctrl+C or Break key at your terminal during execution of a shell program, normally that program is immediately terminated, and your command prompt returns. This may not always be desirable. For instance, you may end up leaving a bunch of temporary files that won't get cleaned up.

### Cleaning up temporary files (Trap SIGINT and SIGTERM):

```
#!/bin/bash

trap 'echo "Cleaning up..."; rm -f temp.txt; exit' SIGINT SIGTERM

echo "Creating temp file..."

touch temp.txt

echo "Running... press Ctrl+C to stop"

while true; do

    sleep 1

done
```

(If you press Ctrl+C, the script removes temp.txt before exiting)