**Q1) What is a process?**

A process is an instance of a program in execution. It includes the program code, current activity, and the associated resources such as open files and memory.

**Q2) What is a daemon? List some names of daemons.**

A daemon is a background process that runs continuously and performs specific tasks or waits for certain events to occur. Some examples include httpd (web server), sshd (secure shell), cron (scheduler), and syslogd (system log).

**Q3) What are the process states in Unix?**

Process States in Unix are different conditions a process can be in while it’s running in an operating system. The primary process states in Unix are:

* Running: The process is actively executing.
* Sleeping: The process is waiting for an event or resource.
* Stopped: The process has been stopped, usually by a signal.
* Zombie: The process has finished execution but still has an entry in the process table.

**Q4)** **What happens when you execute a program?**

When you execute a program, the operating system loads the program into memory, creates a process for it, and the CPU executes the program instructions.

**Q7) What is a zombie and orphan process?**

* A **zombie process** is a process that has completed execution but still has an entry in the process table.
* An **orphan process** is a process whose parent process has terminated, but the orphan process itself is still running. The init process typically adopts orphan processes.

**Q8) What is an advantage of executing a process in the background?**

Executing a process in the background allows the user to continue using the terminal or interface without waiting for the process to complete.

**Q9) What is the "ps" command for?**

The ps command displays information about active processes. It can show details like process ID, user, CPU usage, and memory usage.

**Q10) How would you kill a process?**

You can kill a process using the kill command followed by the process ID (PID). For example, kill 1234.

**Q11) What is a context switch? What all happens during a context switch? When do you need it?**

A context switch is the process of saving the state of a currently running process and restoring the state of a different process. It happens during multitasking to switch between processes. It is needed to ensure efficient CPU utilization.

**Q12)** **PID of init process?**

The PID of the init process is 1.

**Q13)** **Which functions are used to get parent process identification number and process identification number?**

* getpid() gets the process ID.
* getppid() gets the parent process ID.

**Q14)** **Parent process ID of a daemon process?**

Daemon processes typically have the init process (PID 1) as their parent.

**Q15)** **How do you execute one program from within another?**

we can execute one program from within another using functions like exec() family (execl, execp, etc.) after a fork() call.

**Q16)** **How can a parent and child process communicate?**

Parent and child processes can communicate using inter-process communication (IPC) methods such as pipes, message queues, shared memory, and signals

**Q17)** **Brief About The Initial Process Sequence While The System Boots Up.**

- BIOS/UEFI loads the bootloader from the boot device.

- Bootloader loads the kernel into memory.

- Kernel initializes the system, including hardware and memory.

- Kernel starts the ‘init’ process (‘PID 1’), which starts other system processes and user sessions.

**Q18)** **Explain Fork(), Exec(), Clone() System Calls and how they relate to process.**

- fork(): Creates a new process by duplicating the current process.

- exec(): Replaces the current process image with a new process image.

- clone(): Creates a new process or thread, with customizable options for shared resources.

**Q19)** **List The System Calls Used For Process Management**

- system(), fork(), wait(), waitpid(), exec(), clone(), kill()

**Q20)** **How Will You Run A Process In Background? How Will You Bring That Into Foreground And How Will You Kill That Process?**

- Run in background: ./process &

- Bring to foreground: fg %job number

- Kill the process: kill PID

**Q21)** **How Do You Find Which Process Is Taking How Much CPU?**

Using commands like htop, ps

**Q22)** **How will you distinguish if it is a process or a thread? Which is the first task what is spawned in linux kernel? What are the processes with PID 0 and PID 1?**

- Process has its own memory space, while threads share memory space within the process.

- First task spawned is the `swapper` process (`PID 0`).

- `init` process (`PID 1`) is the first user-space process.

**Q23)** **How to extract task\_struct of a particular process if the stack pointer is given?**

Use kernel functions/macros like ‘container\_of()’.

**Q24)** **What is process kernel stack and process user stack? What is the size of each and how are they allocated?**

- Kernel stack: Used by the kernel during process execution, size is typically 8KB.

- User stack: Used by user-space applications, size can vary and is allocated dynamically.

**Q25)** **Why do we need separate kernel stack for each process?**

To maintain process isolation and ensure that the kernel has a private stack for handling system calls and interrupts.

**Q26)** **What is thread\_info? Why is it stored at the end of kernel stack?**

- thread\_info: Structure containing information about the thread.

- Stored at the end of the kernel stack for quick access and efficient context switching.

**Q27)** **What is the use of preempt\_count variable?**

To keep track of the preemption state of a process, ensuring correct and efficient process scheduling.

**Q28)** **What is the difference between interruptible and uninterruptible task states?**

- Interruptible: Task can be interrupted by signals.

- Uninterruptible: Task cannot be interrupted until a specific condition is met.

**Q29)** **How processes and threads are created? (from user level till kernel level)**

- User-level libraries call system calls like fork(), pthread\_create(), etc.

- Kernel-level functions handle process/thread creation and resource allocation.

**Q30)** **How to proceed if system is sluggish**

- Identify resource-hogging processes using top or htop.

- Check disk I/O with iostat.

- Ensure enough free memory and minimal swapping.

**Q31)** **How system call causes change from user to kernel space**

Via a context switch involving CPU instructions like int, syscall, or svc.

**Q32)** **How to determine if some high-priority task is hogging CPU. Priority inversion, priority inheritance, priority ceiling? How priority inheritance will work?**

**Priority Inversion**

Priority inversion occurs when a lower-priority task holds a resource needed by a higher-priority task, causing the higher-priority task to wait. In this situation, the lower-priority task effectively "inverts" the priorities, leading to suboptimal system performance.

**Priority Inheritance**

Priority inheritance is a mechanism used to resolve priority inversion. When a higher-priority task is blocked by a lower-priority task holding a resource, the lower-priority task temporarily inherits the higher priority. This allows the lower-priority task to complete its work and release the resource more quickly, reducing the waiting time for the higher-priority task.

**How Priority Inheritance Works:**

1. **Higher-Priority Task Blocked**: When a high-priority task (Task A) needs a resource currently held by a lower-priority task (Task B), Task A is blocked.
2. **Inheritance Begins**: Task B temporarily inherits the priority of Task A, raising its execution priority.
3. **Resource Release**: Task B completes its critical section and releases the resource.
4. **Restoration**: Task B's priority is restored to its original value.
5. **Higher-Priority Task Resumes**: Task A acquires the resource and continues its execution.

**Priority Ceiling**

Priority ceiling is a technique used to prevent priority inversion by assigning a fixed priority to resources. When a task acquires a resource, its priority is elevated to the resource's priority ceiling, which is the highest priority of any task that may use the resource. This ensures that no task with a lower priority can block the resource, effectively preventing priority inversion.

These mechanisms help maintain system performance and ensure that high-priority tasks are not unduly delayed by lower-priority ones. Let me know if you need more details or if there's anything specific you'd like to dive into!

**Q33)** **What is Process Preemption**

Interrupting a currently running process to schedule another process.

**Q34)** **What is Priority Inversion & Priority Inheritance**

- Priority inversion: Lower-priority task blocks a higher-priority task.

- Priority inheritance: Boosts the blocking task's priority to that of the blocked task.

**Q35)** **Explain about process address space with data seg, code seg.**

- Code segment: Contains executable code.

- Data segment: Contains initialized global and static variables.

- Stack segment: Contains function call stack.

- Heap segment: Contains dynamically allocated memory.

**Process Address Space**

A process address space is a range of memory addresses that a process can use to execute instructions and store data. It is a virtual space provided by the operating system, and it isolates each process, ensuring that processes do not interfere with each other's memory.

The process address space is typically divided into several segments, each serving a specific purpose. The primary segments include:

1. Code Segment (Text Segment)

The code segment, also known as the text segment, contains the executable instructions of a program. This segment is typically marked as read-only to prevent accidental modification of the program's instructions. It is loaded into memory when the program is executed and remains unchanged throughout the program's execution.

1. Data Segment

The data segment is divided into two main parts: initialized data and uninitialized data (BSS).

- Initialized Data: This part of the data segment contains global and static variables that are explicitly initialized by the programmer. For example: int initializedVar = 10;

- Uninitialized Data (BSS): This part of the data segment contains global and static variables that are declared but not explicitly initialized. These variables are initialized to zero by the operating system. For example: int uninitializedVar;

3. Heap Segment

The heap segment is used for dynamic memory allocation. Memory allocated on the heap is managed through functions like `malloc` in C or `new` in C++. Unlike stack memory, which is automatically managed, heap memory must be explicitly allocated and deallocated by the programmer. This segment grows dynamically as more memory is allocated during program execution.

4. Stack Segment

The stack segment is used to store function call information, local variables, and control flow data such as return addresses. The stack operates on a Last-In-First-Out (LIFO) principle, meaning that the last item pushed onto the stack is the first item popped off. Each function call creates a stack frame containing the function's local variables and return address. The stack grows and shrinks as functions are called and return.

### Summary

Here’s a brief summary of the segments:

Segment Description

Code Segment Contains executable instructions; typically read-only |

| \*\*Data Segment\*\*| Contains global and static variables; split into initialized and uninitialized (BSS) data |

| \*\*Heap Segment\*\*| Used for dynamic memory allocation; grows as needed |

| \*\*Stack Segment\*\*| Stores function call information and local variables; operates on LIFO principle |

**Q36)** **How much memory is occupied by process address space?**

Depends on the process, but can be found using commands like pmap or top.

**Q37**) **When a same executable is executed in two terminals like terminal 1 execute ./a.out and terminal 2 executed ./a.out what will the program address space look like on RAM? If a global variable defined as int V1=100 in program\_1 and modified this V1=200 in program\_2, what will be printed in Program\_1 and Program\_2?**

- Each execution has its own address space.

- `Program\_1`: V1=100

- `Program\_2`: V1=200

When the same executable is executed in two different terminals, each execution creates a separate process in the operating system. Each of these processes will have its own independent address space, meaning they do not share the same memory. Here's how it works:

### Separate Address Spaces

- \*\*Process 1 (executed in Terminal 1)\*\*: Has its own code segment, data segment, heap, and stack.

- \*\*Process 2 (executed in Terminal 2)\*\*: Has its own code segment, data segment, heap, and stack.

### Global Variable Scenario

If a global variable `int V1 = 100;` is defined in the program, each process will have its own instance of this global variable in its data segment. Modifying the variable in one process will not affect the other process.

1. \*\*Process 1\*\* (Terminal 1 execution)

- Initial value of `V1`: 100

- Prints `V1`: 100

2. \*\*Process 2\*\* (Terminal 2 execution)

- Initial value of `V1`: 100

- Modifies `V1`: 200

- Prints `V1`: 200

### Summary

- \*\*Program\_1 (Process 1)\*\* will print `V1 = 100`.

- \*\*Program\_2 (Process 2)\*\* will print `V1 = 200`.

Each process operates in its own separate address space, so changes made to the global variable in one process do not affect the global variable in the other process.

**Q38)** **What is the difference between process and threads?**

- Process: Independent execution unit with its own memory space.

- Threads: Light-weight execution units sharing the same memory space within a process.

**Q39)** **Will threads have their own stack space?**

Yes, each thread has its own stack.

**Q40)** **Can one thread access the address space of another thread?**

Yes, threads within the same process share the address space.

**Q41)** **What is task\_struct and how are task states maintained?**

- task\_struct: Data structure in the kernel that represents a process.

- Task states are maintained using state variables within task\_struct.