Linux Interview Essentials – Part I (OS concepts, System-calls, IPCs, Threads)

1 Basics:

1. **What is an operating system (OS)? List parts of an OS.**

   OS is a software used to interface between user applications and hardware. OS helps to manage resources efficiently (ex: divide memory and CPU among all applications) so that multi-tasking is achieved in system. Major parts of the OS are given below:
   - Process management
   - Memory management
   - IPC
   - Network management
   - VFS

2. **What is multi-tasking? How is it achieved in in any OS?**

   Multi-tasking is nothing but doing multiple jobs concurrently, even if you have only one CPU. Multi-tasking is done in OS by switching between processes frequently. The switching is based on scheduling policies of OS. Many scheduling policies are available, which will be configured in the OS depending on the application.

   - First Come First Serve
   - Round Robin
   - Pre-emptive
   - Earliest deadline first
   - Rate monotonic

3. **Differences between function calls and library calls. How do they work?**

Library calls	System calls
1. Switch to library which is present in user space. 2. Using stack to pass arguments to library. 3. Faster execution because no switching between spaces 4. Not portable, library dependency is always present. 5. Simple and Easy to use. 6. We can create and add dynamically any time.	1. Switch to kernel space, because it's defined there. 2. Using CPU registers to pass arguments. 3. Slower execution, delay for switching between User to Kernel space. 4. Always portable, kernel will be same for any Linux distribution. 5. Little complex to use and understand. 6. Adding new system call is difficult. We need to recompile the kernel after adding.

4. **What are the differences between static linking and dynamic linking?**

Static linking	Dynamic linking
1. Linking happens at compile time. 2. Binary size will be more, entire library added to it. 3. Loading time is less, only binary need to load. 4. Memory usage is high. Each time when you start application, library will be duplicated in memory.	1. Linking happens at run time. 2. Binary size will be less because library is separately loaded and linked. 3. Loading time will be more as we need to load library and link at run time. 4. Memory usage is less as the library is loaded dynamically

5. **What are the common errors in code segment? Explain with details**

   Code segment in Linux is a read only memory. So if we try to change any value inside this memory will leads to error called as Segmentation fault. All the instructions + constant values will be stored in this memory. Following are considered as constants.

- char *str = “Hello”; Here the hello is a string constant
- int x = 10; Here ‘10’ is integer constant
- float f = 2.5; Here ‘2.5’ is a double constant
- const int i = 1; Here ‘i’ is a constant variable (Constant variable will store in code segment for some compilers)

6. **What are the common errors in stack segment? Explain with details**

   Stack Overflow
   Whenever process stack limit is over try to access an outside stack memory leads to stack overflow. But this error also prints as segmentation fault in some Linux systems.
   E.g. Call a recursive function infinite times.

   Stack Smashing
   When you trying to access memory beyond limits.
   E.g. int arr[5]; arr[100];

7. **What is a memory leak? Explain with details.**

   When you allocate the memory dynamically and fail to de-allocate it leads to memory leak. Obviously the memory will be cleaned when process terminates. But think about an embedded system running for 24x7 and allocating memory without freeing. Eventually process heap memory will run-out which leads to crash your system. Such issue is called as Memory Leak. Hence it is very important to de-allocate your memory after its usage is completed.

2 System Calls and Kernel:

1. **What is "kernel?" Explain the difference between privilege mode and user mode.**

   An Operating System (OS) is a software package that communicates directly to the computer hardware and all your applications run on top of it while the kernel is the part of the operating system that communicates directly to the hardware. Though each operating system has a kernel, this is buried behind a lot of other software and most users don’t even know it exists. In summary it is the “core” part of the OS.

   - **Kernel Mode /privilege mode:**

   In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic; they will halt the entire PC.

   - **User Mode**

   In User mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system calls to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.

2. **What is a "Monolithic" and "Micro" kernels?**

Monolithic kernel	Micro kernel
1. Kernel size is high because kernel + kernel subsystems compiled as single binary 2. Difficult to do extension or bug fixing 3. Need to compile entire source code at once 4. Faster, run as single binary communication between services is faster. 5. No crash recovery. 6. More secure Eg: Windows, Linux etc	1. Kernel size is small because kernel subsystems run as separate binaries 2. Easily extensible and bug fixing 3. Subsystems can be compiled independently 4. Slower due to complex message passing between services 5. Easy to recover from crash 6. Communication is slow 7. Less secure Eg: MacOS, WinNT

3. **That are the pros & cons of "monolithic" and "micro" kernels?**
   (Refer above answer)

4. **How real time systems (RTS) design** is **connected with kernel designs? Explain.**

5. **What is the connection between Real time OS (RTOS) & Embedded OS (EOS)? Explain.**

6. **What are the type of interrupts? Explain the differences.**

   Interrupts we can be divided into two categories. Hardware interrupts and software interrupts. Interrupts generated due to hardware changes (ex: USB plug-in to your PC) is called hardware interrupt. These are used to handle asynchronous hardware changes or data manipulation. An interrupt generated by software/instruction is called software interrupt. Eg: INT 0x80, sys_enter. These we are using to implement system calls. All the system calls you study as a part of Linux Internals course is also soft interrupts. In both case execution will jump to ISR.

   Another categorization of interrupts are mask-able and unmask-able interrupts.
   Assume when you executing inside an ISR and another interrupts comes. Default case it will stop the current ISR and start executing new ISR. This is a problem if you doing an important job (ex: critical real time task like air bag control) in ISR. So there is an option to change this by masking interrupts. But it's not possible for all interrupts available. So the interrupts which is possible to mask or block is called mask-able interrupts, and which is not possible to mask is non mask-able interrupts.

7. **What is an exception in a system? How the OS handles that? Explain**

   Exceptions belong to a special type of software interrupts. They are generated by the processor itself whenever some unexpected critical event occurs. For instance, a page fault exception (interrupt 14) is triggered when the processor attempts to access a page, which is marked as not-present. The exception handler can then reload the page from disk (virtual memory) and restart the instruction which generated the exception. Three types of exceptions can be generated by the processor: faults, traps and aborts.
   Eg: Fault – Divide by zero, segmentation fault, Bus error etc
   Traps – Debug, breakpoint and overflow.
   Abort- Double fault, machine check.

8. **What is system call? How is it implemented in Linux?**

   A system call is the programmatic way in which a computer program requests a service from the kernel of the operating it is executed on. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

Implementing system calls requires a control transfer from user space to kernel space, which involves some sort of architecture-specific feature. A typical way to implement this is to use a software interrupt or trap. Interrupts transfer control to the operating system kernel so software simply needs to set up some register with the system call number needed, and execute the software interrupt.

9. **Is it always necessary to implement system calls as soft interrupts?**
Yes. There is no other way to switch to kernel space.

10. **What are the drawbacks of system calls?**
    - Switching between spaces (user space to kernel space) makes a delay for execution.
    - Compare to library calls, usage is difficult.
    - System calls present in latest kernels may not be there in old kernels. So there is a difficulty to write a portable code.
    - If you want add a new system call we have to recompile the kernel after adding. Which need more time and entire source code.

11. **Explain Synchronous and Asynchronous methods of communication with details.**

Synchronous	Asynchronous
1. Sender and receiver must be in synch always. 2. Receiver need to wait for data transfer as data might come any time. 3. Received have to wait unnecessarily for a long time. 4. Will be faster 5. While waiting receiver cannot any other job.	1. Sender and receiver not in synch 2. Sender can send data any time at any speed 3. Receiver no need to wait for data transfer. 4. Whenever there is a data to read, receiver will get a notification. 5. Unnecessary waiting can be avoided

3 Virtual File System (VFS):

1. **What is file-system? Describe the Linux File System.**

A **filesystem** is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. The word is also used to refer to a partition or disk that is used to store the files or the type of the **filesystem**

2. **What is i-node number? Why is it important?**

An i-node is an entry in i-node table, containing information (the meta-data) about a regular file and directory. An i-node is a data structure on a traditional Unix-style file system such as ext3 or ext4. The name evolved from index number. Inode number will be unique for all files in system. It is important because it helps to locate any file in the system in a unique way.

3. **What is the importance of Virtual File System? Explain with an example**

VFS allows Linux to support many, often very different, file systems, each presenting a common software interface to the VFS. All of the details of the Linux file systems are translated by software so that all file systems appear identical to the rest of the Linux kernel and to programs running in the system. Linux's Virtual File system layer allows you to transparently mount the many different file systems at the same time.

Eg:

Write(1, buf, len);	Writing to stdout -> Terminal
Write(fd, buf, len);	Writing to normal file -> Hard disk
Write(socket, buf, len);	Writing to socket -> Network

Write(device, buf, len);   Writing to device file -> peripheral interface

In all these case we are using same system call but different types of files. In this case it's VFS who will identify where to pass the data. In other words with the help of VFS only "Everything is file" is possible in Linux.

4 Processes:

1. **What are the differences between a program & process?**

    - A program is a passive entity, such as file containing a list of instructions stored on a disk
    - Process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
    - A program becomes a process when an executable file is loaded into main memory

2. **What are the various states of the process? Explain**

    As a process executes it changes state according to its circumstances. Linux processes have the following states:
    - **Running -** The process is either running (it is the current process in the system) or it is ready to run (it is waiting to be assigned to one of the system's CPUs).
    - **Waiting -**The process is waiting for an event or for a resource. Linux differentiates between two types of waiting process; *interruptible* and *uninterruptible*. Interruptible waiting processes can be interrupted by signals whereas uninterruptible waiting processes are waiting directly on hardware conditions and cannot be interrupted under any circumstances.
    - **Stopped -** The process has been stopped, usually by receiving a signal. A process that is being debugged can be in a stopped state.
    - **Zombie -**This is a halted process which, for some reason, still has a task_struct data structure in the task vector. It is what it sounds like, a dead process.

3. **Can I say if I execute "./a.out" in my command prompt that the process is running? If not what state it is in? When exactly I can say it is running?**

    When you do ./a.out in terminal the program will be loaded to main memory. But not yet started running. It will be in a state called ready state where scheduler keep all process which is ready to run. According to priority and scheduling policy this process will move from ready state to running stage where the CPU picks up instructions from its code segment to execute. In that situation you can say it's actually in running state.

4. **List the various ids related to a process:**

    - PID   process id
    - PPID parent process id
    - GID group id
    - UID user id
    - SID session id

5. **What is a parent process and a child process? How will you create child process?**

    The process which creates a new process called parent process. To create a child process we can use fork() system call. Fork will create a child process which is a duplicate of parent process with a new process id.

6. **How the fork() system call work? Explain them with differences.**

    A fork () system call creates a new process. After creating a process fork returns two different values for parent and child to differentiate both process. Fork returns new child id to parent and returns 0 to child. So by checking the return value of fork we can separate parent and child.

7. **What is orphan process? How can you create an orphan process?**

    - An orphan process is a process whose parent process has finished or terminated, though it remains running itself.
    - Orphaned children are immediately "adopted" by init process in Linux
    - Init automatically cleans its children.

8. **What is zombie process? How can you create a zombie?**

    - Zombie process is a process that has terminated but has not been cleaned up yet
    - It is the responsibility of the parent process to clean up its zombie children
    - If the parent does not clean up its children, they stay around in the system, as zombie

9. **What is daemon process?**

    A daemon is a type of program on Linux operating systems that runs without interacting in the background, rather than under the direct control of a user.

10. **What is importance of PCB/TCB? How the Linux manages processes using that?**

    Process Control Block (PCB, also called Task Controlling Block / Process table / Task Structure, or Switch frame) is a data structure in the operating system kernel containing the information needed to manage a particular process. This is like a data base that Kernel maintains to keep track of all the processes that are currently available in the system. Once the process completes executing, corresponding PCB is deleted. It mains many information about a process, some of them is given below.

    - Process state: State may enter into new, ready, running, waiting, dead depending on CPU scheduling.
    - Process ID: a unique identification number for each process in the operating system.
    - Program counter: a pointer to the address of the next instruction to be executed for this process.
    - CPU registers: indicates various register set of CPU where process need to be stored for execution for running state.
    - CPU scheduling information: indicates the information of a process with which it uses the CPU time through scheduling.

11. **What is OS scheduler? What is context switching?**

    In a multiprogramming OS the process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

    Context switch is the process of storing and restoring the state (more specifically, the execution context) of a process or thread so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU.

12. **What is a blocking call?**

    A function/system call which will move your process / thread to waiting/sleeping state.
    Eg: wait(), pthread_lock(), sem_wait() etc

13. **What is the difference between fork() and exec()?**

    Fork will create a new process with new PID. Exec will replace current process with new and preserves the PID.

14. **Explain about Copy-On-Write (COW) optimization strategy. What are its benefits?**
    - Copy-on-write (called COW) is an optimization strategy
    - When multiple separate process use same copy of the same information it is not necessary to re-create it
    - Instead they can all be given pointers to the same resource, thereby effectively using the resources
    - However, when a local copy has been modified (i.e. write) ,the COW has to replicate the copy, has no other option

15. **It's always better to use wait() system call over sleep() in order for the parent to wait till the child dies. Why is it so? Explain it with respect to sync / Async communication and process states.**

    With sleep we can only provide a fixed time for waiting. Where using wait it waits until child change states. Sleep only wait for child, but wait will clean the resources to avoid zombie. Sleep is a synchronous wait, where is consumed CPU cycles unnecessarily.

5 IPC:

1. **What is IPC (inter process communication)? List different types of IPC possible in Linux.**

   IPC is a mechanism provided by kernel in order to communicate between two or more processes in system.
   - Pipes
   - FIFO
   - Signal
   - Shared memory
   - Message queue
   - Sockets

2. **What are the advantages & disadvantages of Pipes? Explain**

Pros	Cons
• Naturally synchronized • Simple to use and create • No extra system calls required to Communicate (read/write)	• Less memory size (4K) • Only related process can communicate. • Only two process can communicate • One directional communication • Kernel is involved

3. **What are the advantages & disadvantages of FIFO? Explain**

Pros	Cons
• Naturally synchronized • Simple to use and create • Unrelated process can communicate. • No extra system calls required to communicate (read/write) • Work like normal file	• Less memory size (4K) • Only two process can communicate • One directional communication • Kernel is involved

4. **What are the advantages & disadvantages of shared memory? Explain**

Pros	Cons
• Any number process can communicate same time • Kernel is not involved • Fastest form of IPC • Memory customizable	• Manual synchronization is necessary, failing which will result in race condition • Separate system calls are required to handle shared memory • Complex implementation

5. **Shared memory is the fastest IPC. Why is it so?**

   After attaching shared memory to a process, we can access memory using a pointer. So no need to use any system calls to read/write from the memory. Means no kernel involved for read/write, hence its fastest IPC.

6. **What are the difference between normal pointers & function pointers in C?**

Function pointer	Normal pointer
• Holds address of function • Pointing to an address from code segment. • Dereference to execute the Function • Pointer arithmetic not valid	• Holds address of an object (data) • Pointing to an address from stack/heap/data segment • Dereference to get value from address • Pointer arithmetic is valid

7. **What is a call back function? How it is implemented using function pointers in C?**

   In computer programming, a callback is a reference to executable code, or a piece of executable code that is passed as an argument (**Function pointer**) to other code. This allows a lower level software layer to call a subroutine (or function) defined in a higher-level layer.

8. **Where and how callback functions are used? Explain with an example**

   In Linux callback functions are used in signals (one of the examples). If we want to change behavior of a signal, we need to write handler function and register to kernel. While registering we use function pointer for our handler which will act as callback function. In this case kernel will use the function pointer to call our handler when signal receives.

9. **List all default propositions that a process can do with respect to signal.**

   - Terminate
   - Core dump
   - Ignore
   - Stop/pause
   - Continue

10. **Explain “kill” command with respect to signal handling**

    Kill command is used to send a particular signal to a process
    Kill <signal number> <pid>

11. **Explain process semaphores in Linux.**

    Process semaphore are used to synchronize between multiple processes in system.
    In Linux we two standards are available to implement process semaphores. POSIX and System V

12. **What is the difference between communication and synchronization problems?**
    These are two different classifications of IPC. In communication IPC data is transferring across processes, but in synchronization IPC no data is transferring. Its using to synchronize processes while accessing shared data.

6 Synchronization

1. **Explain the problem of synchronization in a multi-tasking environment.**
   - Dead lock (Refer to question number 3 below)
   - Starvation
     Starvation occurs when a scheduler process (i.e. the operating system) refuses to give a

particular thread any quantity of a particular resource (generally CPU). If there are too many high-priority threads, a lower priority thread may be starved. This can have negative impacts, though, particularly when the lower-priority thread has a lock on some resource. This will lead to priority inversion.
- Priority inversion (Refer to question number 3)

2. **What is the difference between synchronization and scheduling?**
In synchronization we make processes to wait for a resources using semaphores. Here the shared memory will be the common resource for process. In scheduling also scheduler will synchronize between process, but here CPU will act as a resource here.

3. **Explain the following definitions:**
    - Race condition
    The situation where multiple threads/process trying to access same resource at the same time is called race condition. Because of these race condition there is chance of corrupting data inside that memory. This leads to getting unexpected output. All this happening because they are running concurrently at same time and switching between thread/process will happen at any pint of time.
    - Critical section
    A section or piece of code where we are doing some important job and that code should be executed by only one thread/process at a time is called critical section.
    We can create a critical section using mutex or semaphores.
    - Atomicity
    One reason of race condition is frequent switching between thread/process.
    To avoid this, one solution is to use atomic variable. Where ever we use an atomic variable scheduler will avoid the switching of process. Scheduler will let the process to finish instructions where atomic variables are coming.
    - Mutual exclusion
    This a synchronization mechanism to avoid race condition. Idea of this mechanism to create a critical section by locking and unlocking a piece of code. So make sure that only one thread is executing the portion of code, and others are waiting for lock
    - Priority inversion
    This a situation where a high priority task is waiting for a mutex/semaphore which acquired by a low priority process and low priority process is pre-empted by a medium priority process.
    - Deadlock
    Deadlock is a situation where process or thread waiting for a mutex/semaphore and will never happen at all. Means thread/process wait forever.

4. **Explain working details of Mutex (Refer to Mutex point in above question)**

5. **Explain working details of Semaphore**

    We need use semaphores when we have multiple resources and multiple threads. Semaphores acts like a counter for resources available for threads. Before accessing resources thread will use a wait() function call which will decrement the counter by one. When counter is already 0 then thread will wait until it’s a positive value. When a thread done with resource, it will release the resource and do post operation to increment the counter.

6. **What are the differences between Mutex & Semaphore?**

    Mutexes and semaphores have some similarities in their implementation, they should always be used differently.
    - A mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).
    - Semaphore is signaling mechanism (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wake up.

7. **What are the differences between Binary semaphore and Mutex?**

Both Binary semaphore and mutex achieve the same purpose. But we can use them separately where it can be used as a signaling vs locking mechanisms (as mentioned above). This means mutex can only be unlocked by the same thread, whereas semaphore can be signaled by any other thread.

7 Threads:

1. **What is a thread (pthread)? Why use threads instead of processes.**

   Threads, like processes, are a mechanism to allow a program to do more than one thing at a time. As with processes, threads appear to run concurrently. Each thread may be executing a different part of the program at any given time. Threads can be used to achieve concurrency, similar to processes, which consumes lesser resources than a process.

2. **What are the advantages/disadvantages of writing a server using multiple threads instead of multiple processes?**

Process	Threads
1. Easy to create. 2. If clients want to share data among them, IPC is required. 3. Memory overhead is more. 4. No dependency of thread library.	1. Complex to create. 2. Sharing data among clients are easy. 3. Less memory overhead. 4. Proper synchronization is required. 5. Chances for dead lock.

3. **How concurrently is achieved in threads?**

   The Linux kernel and the pThreads libraries work together to administer the threads. The kernel does the context switching, scheduling, memory management, cache memory management, etc. There is other administration done at the user level also. The kernel treats each process-thread as one entity. It has its own rules about time slicing that take processes (and process priorities) into consideration but each sub-process thread is a schedulable entity.

4. **How the Kernel does schedules threads?**

   Threads under Linux are implemented as processes that share resources. The scheduler does not differentiate between a thread and a process. Threads on Linux are kernel threads (in the sense of being managed by the kernel). Means to scheduler threads and process are same.
   Eg try command ps -eLf (shows thread info also).

5. **What is a joinable and detached threads? Explain with an example**

   A joinable thread, like a process, is not automatically cleaned up by GNU/Linux when it terminates. Thread's exit state hangs around in the system (kind of like a zombie process) until another thread calls pthread_join to obtain its return value. Only then are its resources released. A detached thread, in contrast, is cleaned up automatically when it terminates

6. **How to pass data to a thread and get it back?**

   To pass arguments to a thread use fourth argument of pthread_create. If multiple arguments need to pass, create a structure and pass structure.
   To return value from thread use second argument of pthread_join. Pass address of type variable you are returning.