**Linux Interview Questions**

**&**

**Its Answers!!!**

**Locking and Synchronization**

**1. On a uniprocessor machine with preemption disabled, what will happen internally when we say spin\_lock()?**

**2. Can I lock a spinlock in one CPU and unlock it another CPU?**

**3. What are the pros and cons of using per CPU variable as synchronization method?**

**4. What is the maximum amount of time CPU can be in critical section after acquiring**

**spinlock?**

**5. What is the difference between binary semaphore and mutex in Linux?**

**6. What is the difference between down\_interruptible vs down\_killable in semaphore?**

**7. Can i call kmalloc(GFP\_KERNEL) while holding a spinlock?**

**8. What are the lock free algorithms present in Linux kernel?**

**9. By disabling interrupts, will we get protection from concurrent access?**

**Memory Management**

**10. What are the use of L1, L2, and L3 Caches?**

**11. Difference between GFP\_KERNEL and GFP\_ATOMIC.**

**12. Where are page tables stored?**

**13. How do you test whether there are memory leaks in a Linux application?**

**14. Which file in Linux gives you information about memory zones?**

**15. Difference between kmalloc and vmalloc.**

**16. What does malloc(0) return?**

**17. What is the maximum memory that can be allocated using vmalloc?**

**18. What is the maximum memory that can be allocated using kmalloc?**

**19. What is the difference between VIRT, RES and SHR fields in top command?**

**20. What is the system call used by malloc and free?**

**21. What is the maximum memory that I can allocate using malloc?**

**22. What is cache coherence?**

**23. Linux intentionally leaves the first few kilobytes (or even megabytes) of each process's**

**virtual address space unmapped, so that attempts to dereference null pointers generate**

**an unhandled page fault resulting in an immediate SIGSEGV, killing the process. a) Yes b)**

**No**

**24. Virtual memory is divided into \_\_\_\_ a) pages b) bytes c) bits**

**25. What are the advantages of using virtual memory?**

**26. \_\_\_\_ is a method of data transfer between main memory to input/output device without**

**the need of processor.**

**27. Differences between vmalloc and kmalloc.**

**28. What is the maximum amount of RAM you can access with 32 bit processor?**

**29. When you call malloc from user space, from where it allocates memory: low mem/high mem or any other location?**

**30. Why use volatile memory? Why not always use nonvolatile memory?**

**31. What does kmalloc(0) return?**

**Process Management**

**32. What is the difference between context switch and preemption?**

>>

**Preemption** is the act of interrupting a process without its involvement. In this context, that probably means a timer interrupt will fire. The word comes from a legal concept of [preemption](http://dictionary.reference.com/browse/preemption): the act or right of claiming or purchasing before or in preference to others. For your purposes, that means that when the timer interrupt fires, that the interrupt service routine (ISR) has preference over the code which was previously running. This doesn't necessarily need to involve a kernel at all; you can have code running in any ISR which will run preemptively.

A **context switch** is what happens when the OS code (running preemptively) alters the state of the processor (the registers, mode, and stack) between one process or thread's context and another. The state of the processor may be at a certain line of code in a one thread. It will have temporary data in registers, a stack pointer at a certain region of memory, and other state information. A preemptive OS can store this state (either to static memory or onto the processes' stack) and load the state of a previous process. This is known as a context switch.

**33. On a multiprocessor system, how do you find out which process is running on which**

**processor?**

>>

Using “top” command on linux shell.

Press ‘f’ to enter file menu, then search for option “Last Used CPU”. Toggle it for display and exit.

Then CPU core affinity will be shown under coloumn ‘P’

**34. How to change the priority of a process in Linux?**

>>

In Linux we can set guidelines for the CPU to follow when it is looking at all the tasks it has to do. These guidelines are called **niceness** or **nice value**. The Linux niceness scale goes from -20 to 19. The lower the number the more priority that task gets. If the niceness value is high number like 19 the task will be set to the lowest priority and the CPU will process it whenever it gets a chance. The default nice value is zero. (RT priorities go from 0 to 139, where in last 40 priorities map to NICE values ie. 100🡪139 maps to (-20)🡪19).

Execute TOP command on shell, and search for column named “PR (Priority)” or “NI (Nice)”.

To change the priority when issuing a new command, you do **nice -n [nice value] [command]**:

**$ nice -n 10 apt-get upgrade**

This will increment the default nice value by a positive 10 for the command, ‘**apt-get upgrade**’ This is often useful for times when you want to upgrade apps but don’t want the extra process burden at the given time.

To change the priority of an existing process just do renice [nice value] -p [process id]:

**$ renice 10 -p 21827**

This will increment the priority of the process with an id of 21827 to 10.

**Note: Only root can apply negative nice values.**

**35. How can I find out the Count of number of times a process has been preempted in Linux?**

>>

**$ cat /proc/<tid>/status**

contains many statistics, including **voluntary\_ctxt\_switches** and **nonvoluntary\_ctxt\_switches**.

<tid> -- thread id of linux task

**36. What happens internally during context switch in Linux kernel?**

>>

A process runs on the CPU until it is [context switched](https://osr507doc.xinuos.com/en/PERFORM/PERFORM_Glossary.html#glP_context_switch).

This happens when one of the following occurs:

* The process exits.
* The process uses up its time slice.
* The process requires another resource that is not currently available or needs to wait for I/O to complete.
* A resource has become available for a sleeping process. If there is a higher priority process ready to run, the kernel will run this instead (the current process is [preempted](https://osr507doc.xinuos.com/en/PERFORM/PERFORM_Glossary.html" \l "glP_preemption)).
* The process relinquishes the CPU using a semaphore or similar system call.

A [context switch](https://osr507doc.xinuos.com/en/PERFORM/PERFORM_Glossary.html#glP_context_switch) occurs when the kernel transfers control of the CPU from an executing process to another that is ready to run. The kernel first saves the [context](https://osr507doc.xinuos.com/en/PERFORM/PERFORM_Glossary.html#glP_context) of the process. The context is the set of CPU register values and other data that describes the process' state. The kernel then loads the context of the new process which then starts to execute.

When the process that was taken off the CPU next runs, it resumes from the point at which it was taken off the CPU. This is possible because the saved context includes the instruction pointer. This indicates the point in the executable code that the CPU had reached when the context switch occurred.

**37. What is buffer/cache?**

>>

YouTube Link: <https://www.youtube.com/watch?v=x2vegjeJICk>

Reading from a disk is very slow compared to accessing (real) memory. In addition, it is common to read the same part of a disk several times during relatively short periods of time. For example, one might first read an e-mail message, then read the letter into an editor when replying to it, then make the mail program read it again when copying it to a folder. Or, consider how often the command ls might be run on a system with many users. By reading the information from disk only once and then keeping it in memory until no longer needed, one can speed up all but the first read. This is called **disk buffering**, and the memory used for the purpose is called the **buffer cache**.

Most operating systems have buffer caches (although they might be called something else), but not all of them work according to the above principles. Some are *write-through*: the data is written to disk at once (it is kept in the cache as well, of course). The cache is called *write-back* if the writes are done at a later time. Write-back is more efficient than write-through, but also a bit more prone to errors: if the machine crashes, or the power is cut at a bad moment, or the floppy is removed from the disk drive before the data in the cache waiting to be written gets written, the changes in the cache are usually lost. This might even mean that the filesystem (if there is one) is not in full working order, perhaps because the unwritten data held important changes to the bookkeeping information.

Because of this, you should never turn off the power without using a proper shutdown procedure or remove a floppy from the disk drive until it has been unmounted (if it was mounted) or after whatever program is using it has signaled that it is finished and the floppy drive light doesn't shine anymore. The **sync** command *flushes* the buffer, i.e., forces all unwritten data to be written to disk, and can be used when one wants to be sure that everything is safely written. In traditional UNIX systems, there is a program called **update** running in the background which does a **sync** every 30 seconds, so it is usually not necessary to use **sync**. Linux has an additional daemon, **bdflush**, which does a more imperfect sync more frequently to avoid the sudden freeze due to heavy disk I/O that **sync** sometimes causes.

Under Linux, **bdflush** is started by **update**. There is usually no reason to worry about it, but if **bdflush** happens to die for some reason, the kernel will warn about this, and you should start it by hand (**/sbin/update**).

The cache does not actually buffer files, but blocks, which are the smallest units of disk I/O (under Linux, they are usually 1 KB). This way, also directories, super blocks, other filesystem bookkeeping data, and non-filesystem disks are cached.

The effectiveness of a cache is primarily decided by its size. A small cache is next to useless: it will hold so little data that all cached data is flushed from the cache before it is reused. The critical size depends on how much data is read and written, and how often the same data is accessed. The only way to know is to experiment.

If the cache is of a fixed size, it is not very good to have it too big, either, because that might make the free memory too small and cause swapping (which is also slow). To make the most efficient use of real memory, Linux automatically uses all free RAM for buffer cache, but also automatically makes the cache smaller when programs need more memory.

**38. Difference between orphan and zombie process.**

>>

**Zombie Process:**

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

// A C program to demonstrate Zombie Process.

// Child becomes Zombie as parent is sleeping

// when child process exits.

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

    // Fork returns process id

    // in parent process

    pid\_t child\_pid = fork();

    // Parent process

    if (child\_pid > 0)

        sleep(50);

    // Child process

    else

        exit(0);

    return 0;

}

**Orphan Process:**

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

However, the orphan process is soon adopted by init process, once its parent process dies.

// A C program to demonstrate Orphan Process.

// Parent process finishes execution while the

// child process is running. The child process

// becomes orphan.

#include<stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

    // Create a child process

    int pid = fork();

    if (pid > 0)

        printf("in parent process");

    // Note that pid is 0 in child process

    // and negative if fork() fails

    else if (pid == 0)

    {

        sleep(30);

        printf("in child process");

    }

    return 0;

}

**39. What is the use of swapper process in Linux?**

>>

Unices stopped using swapping a long time ago.

The swapper process, as was, used to perform process swap operations. It used to swap entire processes — including all of the kernel-space data structures for the process — out to disc and swap them back in again.

g all of the kernel-space data structures for the process — out to disc and swap them back in again. It would be woken up, by the kernel, on a regular basis, and would scan the process table to determine what swapped-out-and-ready-to-run processes could be swapped in and what swapped-in-but-asleep processes could be swapped out.

**40. How to kill the process which is in TASK\_UNINTERRUPTIBLE state?**

>>

top shows the process as D which is documented as "uninterruptible sleep".

How can I get rid of this? It prevents me from unloading the VirtualBox kernel driver to load a newer one.

**Simple answer:** you cannot.

**Longer answer:** the uninterruptable sleep means the process will not be woken up by signals. It can be only woken up by what it's waiting for. When I get such situations eg. with CD-ROM, I usually reset the computer by using suspend-to-disk and resuming.

**41. What is load average in Linux?**

>>

Linux load averages are "system load averages" that show the running thread (task) demand on the system as an average number of running plus waiting threads. This measures demand, which can be greater than what the system is currently processing. Most tools show three averages, for 1, 5, and 15 minutes:

$ uptime

5:37 up 14 days, 13:30, 3 users, load averages: **1.85 2.02 2.15**

Some interpretations:

* If the averages are 0.0, then your system is idle.
* If the 1 minute average is higher than the 5 or 15 minute averages, then load is increasing.
* If the 1 minute average is lower than the 5 or 15 minute averages, then load is decreasing.
* If they are higher than your CPU count, then you might have a performance problem (it depends).

As a set of three, you can tell if load is increasing or decreasing, which is useful. They can be also useful when a single value of demand is desired, such as for a cloud auto scaling rule. But to understand them in more detail is difficult without the aid of other metrics. A single value of 23 - 25, by itself, doesn't mean anything, but might mean something if the CPU count is known, and if it's known to be a CPU-bound workload.

Four different commands to check the load average in linux

* Command 1: Run the command, “cat /proc/loadavg” .
* Command 2 : Run the command, “w” .
* Command 3 : Run the command, “uptime” .
* Command 4: Run the command, “top” . See the first line of top command's output.

**42. What is resident memory in process?**

>>

The actual size of physical memory used by a process (known as ***resident* memory** of the process) is really a hard game… and the sole component of the system that actually knows about it is the kernel (it’s even one of its jobs). Fortunately, the kernel exposes some interfaces that will let you retrieve some statistics about the system or a specific process.

On Linux, those data are exposed through the /proc file-system and more specifically by the content of /proc/[pid]/. These directories (one per process) contain some pseudo-files that are API entry points to retrieve information directly from the kernel.

These (**ps, top, pmap**) tools display the data retrieved from the kernel with little to no modification. As a consequence they are good entry points to understand how the kernel classifies the memory. In this article we will analyze the memory-related outputs of top and pmap.

**top: process statistics**

top is a widely known (and used) tool that allows monitoring the system. It displays one line per process with various columns that may contain CPU related, memory related, or more general information.

When running top, you can switch to the memory view by pressing G3. In that view, you will find, among others, the following columns: %MEM, VIRT, SWAP, RES, CODE, DATA, SHR. With the exception of SWAP, all these data are extracted from the file /proc/[pid]/statm that exposes some memory related statistics. That file contains 7 numerical fields: size (mapped to VIRT), **resident** (mapped to **RES**), share (mapped to SHR), text (mapped to CODE), lib (always 0 on Linux 2.6+), data (mapped to DATA) and dt (always 0 on Linux 2.6+, mapped to nDrt).

The first one contains the number of anonymous resident pages (MM\_ANONPAGES), the second one is the number of file-backed resident pages (MM\_FILEPAGES). Some pages may be considered as resident for more than one process at once, so the sum of the RES may be larger than the amount of RAM effectively used, or even larger than the amount of RAM available on the system.

**43. What is the use of Procedure-linking table (PLT) while application is starting up?**

>>

PLT stands for Procedure Linkage Table which is, put simply, used to call external procedures/functions whose address isn't known in the time of linking, and is left to be resolved by the dynamic linker at run time.

GOT stands for Global Offsets Table and is similarly used to resolve addresses. Both PLT and GOT and other relocation information is explained in greater length in [this article](http://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html).

**44. Difference between fork and vfork.**

>>

Both**fork()** and **vfork()** are the **system calls** that creates a new process that is identical to the process that invoked fork() or vfork(). Using **fork()** allows the execution of parent and child process simultaneously. The other way, **vfork()** suspends the execution of parent process until child process completes its execution.

The primary difference between the fork() and vfork() system call is that the child process created using fork has separate address space as that of the parent process. On the other hand, child process created using vfork has to share the address space of its parent process.

|  |  |  |
| --- | --- | --- |
| **BASIS  FOR  COMPARISON** | **FORK()** | **VFORK()** |
| Basic | Child process and parent process has separate address spaces. | Child process and parent process shares the same address space. |
| Execution | Parent and child process execute simultaneously. | Parent process remains suspended till child process completes its execution. |
| Modification | If the child process alters any page in the address space, it is invisible to the parent process as the address space are separate. | If child process alters any page in the address space, it is visible to the parent process as they share the same address space. |
| Copy-on-write | fork() uses copy-on-write as an alternative where the parent and child shares same pages until any one of them modifies the shared page. | vfork() does not use copy-on-write. |

### **Definition of fork()**

The **fork()** is a system call use to create a **new process**. The new process created by the fork() call is the child process, of the process that invoked the fork() system call. The code of child process is identical to the code of its parent process. After the creation of child process, both process i.e. parent and child process start their execution from the next statement after fork() and both the processes get executed **simultaneously**.

The parent process and child process do have **separate address space**. Hence, when any of the processes modifies any statement or variable in the code. It would not be reflected in other process codes. Let’s suppose if child process modifies the code it would not affect the parent process.

Some child process after their creation immediately calls **exec()**. The exec() system call **replaces the process** with the program specified in its parameter. Then the separate address space of child process is of no use. The one alternative here is copy-on-write.

The **copy-on-write** let the parent and child process to share same address space. If the any of the processes writes on the pages in address space the copy of address space is created to let both the process work independently.

### **Definition of vfork()**

The modified version of fork() is vfork(). The **vfork()** system call is also used to create a new process. Similar to the fork(), here also the new process created is the child process, of the process that invoked vfork().  The child process code is also identical to the parent process code. Here,the child process **suspends the execution** of parent process till it completes its execution as both the process share the same address space to use.

As the child and parent process shares the **same address space**. If any of the processes modifies the code, it is visible to the other process sharing the same pages. Let us suppose if the parent process alters the code; it will reflect in the code of child process.

As using vfork() does not create separate address spaces for child and parent processes. Hence, it must be **implemented** where the child process calls **exec()** immediately after its creation. So, there will be no wastage of address space, and it is the **efficient** way to create a process.  vfork does not use **copy-on-write**.

**45. What is kernel preemption?**

>>

A preemptive kernel is one that can be interrupted in the middle of executing code - for instance in response for a system call - to do other things and run other threads, possibly those that are not in the kernel.

The main advantage of a preemptive kernel is that sys-calls do not block the entire system. If a sys-call takes a long time to finish then it doesn't mean the kernel can't do anything else in this time. The main disadvantage is that this introduces more complexity to the kernel code, having to handle more end-cases, perform more fine grained locking or use lock-less structures and algorithms.

Under Linux, user-space programs have always been preemptible : the kernel interrupts user-space programs to switch to other threads, using the regular clock tick. So, the kernel doesn't wait for user-space programs to explicitly release the processor (which is the case in cooperative multitasking). This means that an infinite loop in an user-space program cannot block the system.

However, until 2.6 kernels, the kernel itself was not preemtible : as soon as one thread has entered the kernel, it could not be preempted to execute an other thread. The processor could be used to execute another thread when a syscall was terminated, or when the current thread explictly asked the scheduler to run another thread using the schedule() function. This means that an infinite loop in the kernel code blocked the entire system, but this is not really a problem : the kernel code is designed so that there are no infinite loops.

Kernel preemption has been introduced in 2.6 kernels, and one can enable or disable it using the CONFIG\_PREEMPT option. If CONFIG\_PREEMPT is enabled, then kernel code can be preempted everywhere, except when the code has disabled local interrupts. An infinite loop in the code can no longer block the entire system. If CONFIG\_PREEMPT is disabled, then the 2.4 behaviour is restored.

Pros:The preemption kernel can improve latency and scalability, and it can make high priority task run and respond timely.

Cons: It make writing code difficult in preemption kernel, especially in SMP, and you must consider many factors.

**46. What is process kernel stack and process user stack?**

>>

In a Linux system, every user process has 2 stacks, a user stack and a dedicated kernel stack for the process. The user stack resides in the user address space (first 3GB in 32-bit x86 arch.) and the kernel stack resides in the kernel address space (3GB-4GB in 32bit-x86) of the process.

When a user process needs to execute some privileged instruction (a system call) it traps to kernel mode and the kernel executes it on behalf of the user process. **This execution takes place on the process' kernel stack.**

**47. What is difference between background process and daemon?**

>>

Service is something the operating system provides for all users. xinetd/inetd is one such example. xinetd takes care of popular services like ftpd.  
  
A process is a running instance of an executable. It is represented by a process id and has an address space assigned to it.  
  
Daemon is an application that has no terminal associations.  One such example is init.

**48. Which state the process is in when executing the below line and waiting for input?**

**scanf("%d", &val);**

**a) TASK\_INTERRUPTIBLE b) TASK\_UNINTERRUPTIBLE**

>>

**TASK\_INTERRUPTIBLE**

**Interrupts**

**49. Which hardware is responsible for generating timer interrupts in Linux kernel?**

**50. How to direct interrupt to a particular cpu in Linux kernel**

**51. Interrupt handler in Linux kernel run with current interrupt line disabled on all processors.**

**a)True b) False**

**52. What is the difference between request\_irq and request\_threaded\_irq?**

**53. What are the advantages of disabling interrupts?**

**P a g e | 3**

**54. What happens when two interrupts arrive at the same time in Linux?**

**55. Does Linux kernel supports interrupt nesting?**

**56. APIC vs PIC in x86.**

**57. Can we use printk inside interrupt handler?**

**58. What happens when we are in interrupt handler and another interrupt arrives?**

**59. Can we disable interrupts on all processors in Linux kernel?**

**60. How Message Signalled Interrupt works?**

**61. What happens if we call schedule() in interrupt handler?**

>>

In kernel code you can be either in **interrupt context** or in **process context**. When you are in interrupt context, you cannot call any blocking function (e.g., schedule()) or access the current pointer. That's related to how the kernel is designed and there is no way for having such functionalities in interrupt context. (See also [this answer](https://stackoverflow.com/questions/1053572/why-kernel-code-thread-executing-in-interrupt-context-cannot-sleep)).

Depending on what is your purpose, you can find some strategy that allows you to reach your goal. To me, it sounds strange that you have to call schedule() explicitly instead of relying on the natural kernel flow.

One possible approach follows (but, again, it depends on your specific goal). From the IRQ you can schedule the work on a [**work queue**](http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/workqueue.txt) through schedule\_work(). The work queue, in fact, by design, executes kernel code in process context. From there, you are allowed to call blocking functions and access the current process data.

**62. What all actions cannot be performed in interrupt handler?**

**PCI**

**63. How do you check how many lanes are being used by pcie card in Linux?**

**64. Maximum number of PCI devices that can be connected to a host?**

**65. What are lanes in PCI?**

**66. How auto detection of PCI devices happen in PCI?**

**67. What is a PCI bridge?**

**68. Is PCI serial protocol or parallel protocol. What is the maximum data rate achieved with PCI?**

**69. What is the use of Base Address Register in PCI?**

**70. What is the use of PCI protocol?**

**Processor**

**71. Difference between processor and core?**

**72. What is NUMA?**

**73. Does microprocessor have internal memory? a) Yes b) No**

**74. System bus = address bus + data bus + control bus. Yes/No**

**75. Consider a square matrix of x size, which technique will be faster to perform sum of**

**elements of the matrix?**

**A) Traversing row by row and adding elements**

**B) Traversing column by column and adding elements**

**76. What is a cache line?**

**77. What is SMP?**

**78. Comment the registers you know about any processor and its purpose.**

**79. What are the factors you consider when selecting a processor for your project?**

**80. Why do we need PCI, USB and other kind of buses why can't we directly connect**

**peripherals to the processor?**

**81. Explain differences between SRAM and DRAM in terms of cost, access time and size.**

**82. What is difference between motherboard, chipset and processor?**

**83. Difference between pads and pins on a SoC.**

**84. What is out of order execution in processor?**

**Libraries (Static and Shared) + Compilation**

**85. Where do executables look for shared objects at runtime?**

**86. What is the purpose of fPIC flag while generating shared library?**

**87. What are the advantages of static library over shared library?**

**88. What is present inside a static library?**

**89. What are relocations in an object file?**

**90. What is the benefit of declaring a symbol as weak?**

**91. What are the operations performed by linker during compilation process?**

**92. When two userspace executables uses same shared library .so, how it organised in**

**process address space?**

**93. Will a code compiled in one Linux distribution work on all other Linux distribution?**

**94. Why c code is converted to assembly and then to machine code. What is the benefit of**

**converting it to assembly? Why not directly to machine code?**

**95. Will the assembly of the below code generated by compiler same on Windows/Linux?**

**int add(int a, int b)**

**{**

**return a+b;**

**}**

**96. What are drawbacks of using inline?**

**97. What will be the output of echo $?, when you return 256 from your c main function**

**98. C code compiled on one Linux distribution (eg ubuntu) will it execute on other Linux**

**distribution (redhat)?**

**99. Whether GCC searches for both static and shared library. Which is searched initially by**

**gcc compiler?**

**100. What is a toolchain?**

**101. How do you find out whether a particular binary has debugging symbols or not?**

**Memory Mapped IO and Port Mapped IO**

**102. Difference between IO Mapped IO and Memory Mapped IO.**

**103. cat /proc/ioports does it list all the I/O Ports used by the processor?**

**a) Yes b) No**

**104. Will there be any data in /proc/ioports for ARM?**

**105. Does x86 uses memory mapped IO? A) Yes B) No**

**106. Difference between microcontroller and System on Chip.**

**107. Which processors uses I/O Mapped I/O for accessing peripheral devices**

**memories/registers?**

**Linux Kernel Module Programming**

**108. Will a module be loaded if it has while(1) loop in module\_init function?**

**109. Where do you find the information of builtin modules from running Linux image?**

**110. What is the use of likely and unlikely macros in Linux kernel?**

**111. The Makefile macro that one sets to identify what file for the kernel Makefile to make**

**into a module is \_\_\_\_\_. a. obj-m b. obj-y c. target d. list**

**112. Why we can't unload the module when bug() is call?**

**113. Is Linux source sufficient to compile a Linux kernel module or do I need to install the**

**kernel for building modules?**

**114. How do you find the Linux version of a compiled kernel module?**

**115. What happens if we don't specify MODULE\_LICENSE macro in C Code?**

**Character Drivers**

**116.What is the maximum major number in case of Character and block device driver?**

**117. Which one will you use to empty a file?**

**a) /dev/null b) /dev/zero**

**118. What are the examples of character devices in Linux?**

**119. What is the use of file->private\_data in a device driver?**

**Commands**

**120. How to create 100 files in a single command?**

**121. Why running ps command without options on shell shows only two entries?**

**122. \_\_\_\_ command will you use to find list of C files present in a directory.**

**123. Which Linux commands you run when you get a new embedded hardware?**

**124. Difference between 'echo 0>file' and 'echo 0 > file'.**

**125. What is the first tool you use if your c/cpp application is not behaving properly in Linux?**

**Time Management**

**126. Will the call to date command read from rtc?**

**127. gettimeofday and clock\_gettime both returns the wall time. Which one will you use in your application and how do you decide?**

**Files**

**128. What is the use of O\_SYNC flag while opening the file?**

**129. What is the difference between file and inode?**

**130. I have closed stdout close(1) Can we reopen it?**

**Scheduling**

**131. How is scheduler invoked in Linux kernel?**

**Signals**

**132. What are the advantages of sigaction over signal?**

**Bottom Halves**

**133. Which context softirq/tasklets/workqueues runs?**

**134. What are the various bottom halves techniques available in Linux Kernel?**

**135. Why do we need to divide interrupt handler into top half and bottom half?**

**136. Differences between softirqs and tasklets.**

**Linux Boot Flow**

**137. Which is loaded first Initrd or kernel image during boot?**

**Kernel Threads**

**138. irq/ - Threaded interrupt handlers**

**ksoftirqd/ - Softirq Threads**

**kworker/ - Workqueue Threads**

**What more kernel threads you see in ps?**

**139. How do you identify kernel thread from ps output?**

**Linux Kernel Internals**

**140. What is difference between operating system and kernel?**

**141. Do we need device tree if we don't have any SPI or I2C devices connected?**

**142. Is Linux Kernel Monolithic? a) Yes b) No**

**143. A Linux distribution is a combination of Linux kernel + \_\_\_\_\_**

**144. What are the different ways to find out the kernel command line parameters?**

**145. What do you mean by vanilla kernel?**

**146. Is it possible to boot Linux kernel without initrd/initramfs?**

**147. Which one will you use for custom driver operations:sysfs or ioctl?**

**148. Difference between printk and trace\_printk.**

**149. Which kernel parameters you added to kernel command line?**

**150. When an unallocated pointer is derefrenced, who will check that this is invalid**

**instruction?**

**151. What is the difference between initrd and initramfs?**

**152. What is the use of asmlinkage in Linux kernel?**

**153. /proc/kallsyms contains both global and exported symbols. How do you identify**

**exported symbols from output of /proc/kallsyms?**

**154. We say Linux kernel is monolithic. What does it mean?**

**155. What is the use of compatible property in nodes of device tree?**

**156. Will /proc/kallsyms contains the global variables defined in Linux kernel modules?**

**157. What exactly is kernel headers? Why should we install them for generating kernel**

**modules?**

**158. Which time is used in dmesg logs?**

**159. Explain the role of the variable preempt\_count in the Linux kernel.**

**160. What is the initial value of jiffies on boot?**

**161. What is jiffies in Linux kernel?**

**162. Difference between mdelay and msleep in Linux kernel.**

**163. Can I disable drivers which are configured as builtin from loading?**

**164. How Linux kernel modules are automatically loaded in any Linux distribution?**

**System Calls**

**165. How do you allow only root to run your user application. If normal user is running your**

**user application, you need to print a message and exit?**

**166. What are the maximum number of arguments we can have for a system call?**

**167. Who updates the errno in Linux? a) glibc b) Linux Kernel c) None of these d) Both**

**168. Can I update the system call table from Linux Kernel module?**

**169. Does calling gettimeofday() from your user space code perform a mode switch from user to kernel space?**

**Sysfs**

**170. Attributes in sysfs are mapped to a) directories b)files**

**171. kobjects in sysfs are mapped to a) directory b)files**

**172. What is the use of /etc/fstab file in Linux?**

**Linux Kernel Compilation**

**172. What should be the number of jobs when compiling Linux kernel make –j?**

**173. What are the different commands you can run on a generated kernel image**

**(bzImage/vmLinux/vmlinuz) before installing?**

**174. Difference between vmLinux and bzImage?**

**Misc**

**175. What are the minimum requirements of Linux to work on a hardware?**

**176. What is the asm-generic folder in Linux source code? What it contains?**

**177. What is the use of initrd image while booting?**

**178. How do you find out which init manager is running on your Linux machine:**

**sysvinit/systemd?**

**179. What is the first function that will be called when Linux kernel is loaded?**

**180. What are the data structure implemented in Linux kernel?**

**181. As an embedded developer, how do you test a temperature sensor without any special hardware?**

**182. We have systemd-analyze plots for boot up time measurement,Do we have any such**

**similar tools to measure the different sequence of shut down time?**

**183. Can we run Linux on microcontroller?**

**184. Given a device datasheet, what all you look for?**

**185. Kernel code on Intel processor is running on ring a) 0 b) 1 c) 2 d) 3**

**186. What is difference between debug build and release build?**

**187. What is segmentation fault?**

**188. What are the various mechanisms available for asynchronously notifying user space**

**from kernel space about a particular event?**

**189. When should we add Suffixes like 'L', 'U', 'LL', 'UL', to integers in C?**

**190. What is the difference between if(1 < x < 2) and if( (1 < x) && (x<2))?**

**191. Which one do you prefer and why? char buf[256] = {0}; or char buf[256]; memset(buf,**

**0, sizeof(buf));?**

**192. Why a c code compiled in Linux, will not work in windows even they are of same**

**architecture?**

**193. What is a loopback device?**

**194. What is the significance of LD\_LIBRARY\_PATH environmental variable?**

**195. What is the significance of /proc/devices file?**

**196. What is the use of environmental variable PATH?**

**197. How do you detect segmentation fault?**

**198. What is monotonic clock?**

**199. What is the simplest technique you can do to make someone Linux system unusable?**

**200. What is vdso?**

**201. Which will execute faster / or >> ?**

**Linux Device Model**

**202. Explain the relation between ksets, kobjects and ktypes?**