1. **What happens when you type ls -l in the shell**?
2. Check shell key word, command interpreter checks whether there such special words in it's own language: shell keywords or shell built-ins.
3. ls isn't among shell keywords, then it checks aliases and replace alias with it's value, most likely there should be something like that: ls='ls --color=auto'
4. current process (we call parent process, or parent) find ls in $PATH variable, eg /usr/bin/ls
5. It forks (fork()) new process and execs it's code (exec()). Process env is inherited from parent process to new "ls" process.
6. New process becomes session leader and starts working on foreground (bash is moved to background)
7. ls process loads shared libraries from LD\_PATHs (ldd /bin/ls)
8. it executes lots of systemcalls, you can check by strace, and the main part I believe are openat() and getdents() first opens directory and second reads entries inside there.
9. prints output and exits, sends wait() signal and parent process bash terminates it completely.
10. **What happend when I press the power-on button of Linux operating system, i.e., what happened during the bootstrap process of Linux?**

<https://leetcode.com/discuss/interview-question/124638/what-happens-in-the-background-from-the-time-you-press-the-Power-button-until-the-Linux-login-prompt-appears>

1. the Basic Input/Output System (BIOS) initializes the hardware, including the screen and keyboard, and tests the main memory. This process is also called POST (Power On Self Test).

2. Master Boot Records (MBR) and Boot Loader

The BIOS software is stored on a ROM chip on the motherboard. After this, the remainder of the boot process is completely controlled by the operating system.

the most common ones are **GRUB** (for **GRand Unified Boot loader**) and **ISOLINUX** (for booting from removable media).

the boot loader is responsible for loading the kernel image and the initial RAM disk (which contains some critical files and device drivers needed to start the system) into memory.

3. Kernel,

The boot loader loads both the kernel and an initial RAM–based filesystem (**initramfs**) into memory so it can be used directly by the kernel.  
When the kernel is loaded in RAM, it immediately initializes and configures the computer’s memory and also configures all the hardware attached to the system. This includes all processors, I/O subsystems, storage devices, etc. The kernel also loads some necessary user space applications.

4. /sbin/init and Services

Once the kernel has set up all its hardware and mounted the root filesystem, the kernel runs the **/sbin/init** program. This then becomes the initial process, which then starts other processes to get the system running. Most other processes on the system trace their origin ultimately to init; the exceptions are kernel processes, started by the kernel directly for managing internal operating system details.

5. Text-Mode Login

Near the end of the boot process, **init** starts a number of text-mode login prompts (done by a program called **getty**). These enable you to type your username, followed by your password, and to eventually get a command shell.

**2. what happened after ctr-c ?**

Ctrl+C is the interrupt signal. When you type this in a terminal, bash sends SIGINT to the job in the foreground. If there is no job (which is the case when you've just opened a terminal), nothing happens. The terminal emulator program is not a job running in the shell, so, it doesn't get the signal and doesn't close.

If you want to close the terminal with a control key, use Ctrl+D (EOF) which causes bash to exit (and closes the terminal too).

**3. what is pipe**? ex: ls | grep , what happens when you type this.

redirection. Allows one’s output become one’s input

4. **what is "file description"**?

a number represents an opened file,

descriptor 0 with Standard Input of a process,

file descriptor 1 with Standard Output,

file descriptor 2 with Standard Error

find /proc/$pid/fd

**5. Does the user thread run it on a kernel thread?**

**or**

**Can work user thread without being associated with a kernel thread?**

User thread will not have access to some of these critical features. e.g. a text editor can never shoot a thread which has the ability to change the physical address of the process. But if needed, a user thread can map to kernel thread and issue some of the system calls which it couldn't do as an independent entity. The kernel thread would then map this system call to the kernel and would execute actions if deemed fit.

**5 . What is system call?**

<https://www.geeksforgeeks.org/introduction-of-system-call/>

A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system’s kernel. System call provides the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Services Provided by System Calls :

* Process creation and management
* Main memory management
* File Access, Directory and File system management
* Device handling(I/O)
* Protection
* Networking, etc.

**6. ELF file**

<https://www.geeksforgeeks.org/executing-main-in-c-behind-the-scene/>

The executable file created after compiling a C source code is a [Executable and Linkable Format (ELF) file](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format).

Every ELF file have a ELF header where there is a **e\_entry** field which contains the program memory address from which the execution of executable will start. This memory address point to the **\_start()** function.  
After loading the program, loader looks for the **e\_entry** field from the ELF file header. [Executable and Linkable Format (ELF)](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) is a common standard file format used in UNIX system for executable files, object code, shared libraries, and core dumps.

The linker combines multiple object files and library modules into a single executable file

* Resolving symbols defined within these files
* Listing symbols needing to be resolved by loader

The Loader reads the executable file

* Allocates memory
* Maps addresses within file to physical memory
* addresses
* Resolves names of dynamic library items

**7. Linking**

* The program we write might make use of other programs (which is usually the case), or libraries of programs. These other programs or libraries must be brought together with the program we write in order to execute it. Linking is performed as the last step in compiling a program.
* (write -> compile -> link -> load -> execute).
* Static linking is the result of the linker copying all library routines used in the program into the executable image. This may require more disk space and memory than dynamic linking, but is both faster and more portable, since it does not require the presence of the library on the system where it is run.
* Dynamic linking is accomplished by placing the name of a sharable library in the executable image. Actual linking with the library routines does not occur until the image is run, when both the executable and the library are placed in memory. An advantage of dynamic linking is that multiple programs can share a single copy of the library.

**8. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.**

* **Process**  
  Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.
* **Thread**  
  A thread is an entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

4. when you execute "telnet google.com 80", what happened?

* telnet: find problem with TCP connectivity

**Fails:**

* **The remote server might not be listening on that port:**  in this instance it will not respond and the connection will fail as it’s not expecting connections on the port. If you have access to the remote server you could run a command such as ‘netstat’ to get an idea of what ports the server is listening on and accepting connections for.
* **The remote server may not be responding on that port:**  however it may be running a firewall which is configured to block the connection resulting in a failure of the telnet client. If you have access to the remote server confirm with ‘netstat’ that a service is listening on the port specified, then investigate any firewalls in use such as Windows firewall or iptables. The telnet client is a good tool for testing server security quickly if you want to see if a server is responding on a port that it shouldn’t be.
* **The connection may be blocked in your own network, or otherwise somewhere else along the route:** Perhaps you are running a firewall in your own network which is blocking various outbound ports, the connection attempt may not even leave your local network depending on your configuration. You could simply try disabling your firewall and performing the test again to rul 9h0p[, e this out as being part of the problem.

**D**omain **N**ame **S**ystem (**DNS**) services. Translate human readable to IP address. When you need to access a website, you type the domain name, such as www.google.com, into the web browser instead of typing an IP address. www.google.com to 172.217.12.46,  This conversion is a DNS query

Swap memory

<http://manpages.ubuntu.com/manpages/cosmic/man8/swapspace.8.html>

**7. What is swap memory**

<https://www.linux.com/news/all-about-linux-swap-space/>

* Swapping is the process a page of memory is copied to the preconfigured space on the hard disk, called swap space, to free up that page of memory. The combined sizes of the physical memory and the swap space is the amount of virtual memory available.

total memory = RAM + swapped = virtual memory

* Swapping is necessary for two important reasons. First, when the system requires more memory than is physically available, the kernel swaps out less used pages and gives memory to the current application (process) that needs the memory immediately. Second, a significant number of the pages used by an application during its startup phase may only be used for initialization and then never used again. The system can swap out those pages and free the memory for other applications or even for the disk cache.
* However, swapping does have a downside. Compared to memory, disks are very slow.

Linux has two forms of swap space: the swap partition and the swap file. The swap partition is an independent section of the hard disk used solely for swapping; no other files can reside there. The swap file is a special file in the filesystem that resides amongst your system and data files.

**A swap file** (or swap space or, in Windows NT, a pagefile) is a space on a hard disk used as the virtual memory extension of a computer's real memory (RAM). Having a swap file allows your computer's operating system to pretend that you have more RAM than you actually do. The least recently used files in RAM can be "swapped out" to your hard disk until they are needed later so that new files can be "swapped in" to RAM

7. system是否应该在swap memory的时候传signal通知对应的process。如果system这样设计，会有怎样的坏处。

SIGUSR1

8. container 限制process可以使用的cpu和memory。

docker constrain CPU Share

So stop and prune all the containers you do not need running.

You can quickly do that (in your development environment) using:

docker stop $(docker ps -a -q) #stop ALL containers

To now remove all containers, run

docker rm -f $(docker ps -a -q) # remove ALL containers

Limiting Memory

To limit memory we use the memory flag when starting a container. For example, we used the following to limit our NGINX server to only 256 MB of RAM.

docker run -d -p 8081:80 --memory="256m" nginx

This sets a hard limit. That means that under no circumstances will the container be allowed to use more than 256 MB of RAM. Alternatively, we could set a soft limit. Soft limits ensure our container can still request additional memory after hitting its limit, preventing service outages.

docker run -d -p 8081:80 --memory-reservation="256m" nginx

The flag to set a soft limit is memory-reservation. To set a soft limit of 256 MB or RAM we would run the following command.

Limiting CPU

Allowing one container to monopolize(垄断) the processors in your Docker host could cause service outages, by starving your other services and containers. Limit how much CPU a container can use.

Limit Number of Cores

We can limit the number of cores available to container by using the cpus flag.

Lock Container to Specific Cores

Just limiting the number of cores means your process will use any available core available. For most purposes this is fine. Sometimes, however, you may want to lock your containers to specific cores.

Limit CPU Time

Limiting CPU time ensure how often a process is able to interrupt the processor or a set of cores.

Shares and Weights

Rather than breaking out the calculator and being very specific about how many cores or CPU time a process can have, apply shares to your process instead. This allows more critical containers to have priority over the CPU when needed.

9. gdb工作原理，attach to the process

The GNU Debugger, usually called just GDB， GDB allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

following commands are among the more useful gdb commands:

• b main - Put a breakpoint at the beginning of the program

• b - Put a breakpoint at the current line

• b N - Put a breakpoint at line N

• b +N - Put a breakpoint N lines down from the current line

• b fn - Put a breakpoint at the beginning of function "fn"

• d N - delete breakpoint number N

• info break - list breakpoints

• r - Run the program until a breakpoint or error

• c - continue running the program until the next breakpoint or error

• f - Run until the current function is finished

• s - run the next line of the program

• s N - run the next N lines of the program

• n - like s, but don't step into functions

• p var - print the current value of the variable "var"

• q - Quit gdb

12. MMU memory management unit

<https://stackoverflow.com/questions/3828358/how-mmumemory-management-unit-unit-in-a-processor-protects-the-memory-segments>

* The MMU is the hardware component that implements the translation from the logical addresses that a process uses to the physical addresses that the hardware uses. It also provides security features such as marking only some parts of memory as executable. It provide the data structures that the kernel needs to implement process swapping and virtual memory, so that the memory pages belonging to process A cannot even be seen by process B, but both A and B can be seen by the trusted kernel.

the physical memory is divided into pages, typically 4KB in size. Each logical address is broken into a page number and an offset. The page numbers index a table in the MMU that translates each logical page to some physical address. This translation happens during every memory access cycle. A single physical page can be mapped into no processes (its probably also in a pool of free pages in that case), exactly one, or several. ???

* The stack, data, and heap of a process is generally made up of pages mapped into exactly that one process. That helps prevent a bug in one process from affecting others, because each process can only write to its own stack, data and heap pages.
* The MMU has a mechanism that throws an exception when a page is accessed that hasn't been mapped to a process. Handling that exception makes it possible to implement virtual memory, and to grow the amount of memory allocated to a process as its needs change.

13. why kernel space user space separated

The *really* simplified answer is that the kernel runs in kernel space, and normal programs run in user space. User space is basically a form of sand-boxing -- it restricts user programs so they can't mess with memory (and other resources) owned by other programs or by the OS kernel. This limits (but usually doesn't entirely eliminate) their ability to do bad things like crashing the machine.

The kernel is the core of the operating system. It normally has full access to all memory and machine hardware (and everything else on the machine). To keep the machine as stable as possible, you normally want only the most trusted, well-tested code to run in kernel mode/kernel space.

14. Difference between load and utilization(usage)

Load average is a measurement of how many tasks are waiting in a kernel run queue (not just CPU time but also disk activity) over a period of time.

CPU utilization is a measure of how busy the CPU is right now.

High load is also not necessarily a bad thing, most of the time it just means the system is being utilized to it's fullest capacity or maybe is beyond it's capability to keep up

15. <http://nonfunctionaltestingtools.blogspot.com/2013/03/vmstat-output-explained.html>

free: The amount of Idle Memory  
buff: Memory used as buffers, like before/after I/O operations  
cache: Memory used as cache by the Operating System

Under Swap we have:

si: Amount of memory swapped in from disk (/s). This shows page-ins

so: Amount of memory swapped to disk (/s). This shows page-outs. The so column is zero consistently, indicating there are no page-outs.

Under IO we have:

bi: Blocks received from block device - Read (like a hard disk)  
bo: Blocks sent to a block device – Write

what kind of system alarm can't be ignored

how to download resource from internet -> wget

how to acquire file detail status -> man

$ free -m

total used free shared buffers cached

Mem: 7976 6459 1517 0 865 2248

-/+ buffers/cache: 3344 4631

Swap: 1951 0 1951

16. **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.

**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.

**Trouble Shooting**

**1. The memory is crush and the system reboots due to a highly memory usage, what would you do**?

**How can you find out which process is causing this problem.(use "top" try to find the highest I/O requests)**

* top**command is used to show the Linux processes. It provides a dynamic real-time view of the running system. Usually, this command shows the summary information of the system and the list of processes or threads which are currently managed by the Linux Kernel.**
* **How to find which file is being currently most reading or requesting?**

**You may use "procfile" command.**

* **How can you avoid this happened? How to avoid the system rebooting due to the memo crush? (write a monitoring script, if the memory load is high, kill the process which highly use the memory resource)**
* **do you know how "memory swap" work? (swap the sleeping process out, and keep the active ones inside the memo).­­­­­­**

**2. whether or not I/O is causing system slowness**

**1. To identify whether I/O is causing system slowness you can use several commands but the easiest is the unix command top.**

**# top**

top - 14:31:20 up 35 min, 4 users, load average: 2.25, 1.74, 1.68

Tasks: 71 total, 1 running, 70 sleeping, 0 stopped, 0 zombie

Cpu(s): 2.3%us, 1.7%sy, 0.0%ni, 0.0%id, 96.0%wa, 0.0%hi, 0.0%si, 0.0%st

Mem: 245440k total, 241004k used, 4436k free, 496k buffers

Swap: 409596k total, 5436k used, 404160k free, 182812k cached

**From the CPU(s) line you can see the current percentage of CPU in I/O Wait; The higher the number the more cpu resources are waiting for I/O access.**

**wa -- iowait**

**Amount of time the CPU has been waiting for I/O to complete.**

**3. Finding which disk is being written to**

**The above top command shows I/O Wait from the system as a whole but it does not tell you what disk is being affected; for this we will use the iostat command.**

**4. Finding the processes that are causing high I/O**

**Iotop**

**5. managing CPU overload, and these alternatives are:**

11.**1. Add more processors (CPUs) to the server.**

**2. Load balance the system tasks by rescheduling large batch tasks to execute during off-peak hours.**

Top

shows the summary information of the system and the list of processes or threads which are currently managed by the Linux Kernel.

statistics of processes and resource usage.

* **PID:** Shows task’s unique process id.
* **PR:**Stands for priority of the task.
* **SHR:** Represents the amount of shared memory used by a task.
* **VIRT:** Total virtual memory used by the task.
* **USER:** User name of owner of task.
* **%CPU:**Represents the CPU usage.
* **TIME+:** CPU Time, the same as ‘TIME’, but reflecting more granularity through hundredths of a second.
* **SHR:** Represents the Shared Memory size (kb) used by a task.
* **NI:** Represents a Nice Value of task. A Negative nice value implies higher priority, and positive Nice value means lower priority.
* **%MEM:**Shows the Memory usage of task.

ps

command is used to list the currently running processes and their PIDs along with some other information

**PID –** the unique process ID  
**TTY –** terminal type that the user is logged into  
**TIME –** amount of CPU in minutes and seconds that the process has been running  
**CMD –** name of the command that launched the process.