|  |  |
| --- | --- |
| **How GDB works** | When a user inserts a breakpoint in a line of code, the debugger saves the opcode at that  given location and replaces it with **0xCC (int 3).** The INT 3 instruction generates a special  one byte opcode (CC) that is intended for calling the debug exception handle  Downside of SW breakpoints is that debugger needs to be able to modify running program,  which is not possible if program is running from read-only memory (quite common in  embedded world). |
| **Data center fabric** | A data center fabric is a system of switches and servers and the interconnections  between them that can be represented as a fabric. Cisco offering includes fabric  management via Application Policy Infrastructure Controller (APIC) or  Data Center Network Manager (DCNM) |
| **Switch words** | **Supervisor engine:** it is basically the control plane.  **Line card:** data plane/responsible for packet forwarding  **Fabric module:** It interconnects two different line-cards also supervisor card of the switch.  **Fabric extender**: a line card connected using fabric has no capability to store a forwarding  table or run any control plane protocols. They are fully managed as part of the parent  switch and do not require independent software upgrades, config backups, or maintenance. |
| **Virtual Memory Organization in 32-bit systems** | **Virtual Memory Organization in 32-bit systems**  High 1GB reserved for kernel-space  Next 3GB exclusive mapping available for each user space  4GB space is also configurable CONFIG\_VMSPLIT\_2G  Because of hardware limitations, the kernel cannot treat all pages as identical. Some pages,  because of their physical address in memory, cannot be used for certain tasks. Because of  this limitation, the kernel divides pages into different zones. The kernel uses the zones to  group pages of similar properties. Linux has to deal with two shortcomings of  hardware with respect to memory addressing:  Some hardware devices are capable of performing DMA (direct memory access)  to only certain memory addresses.  Some architectures are capable of physically addressing larger amounts of  memory than they can virtually address. Consequently, some memory is not  permanently mapped into the kernel address space.  Because of these constraints, there are three memory zones in Linux:  **ZONE\_DMA** This zone contains pages that are capable of undergoing DMA.  **ZONE\_NORMAL** This zone contains normal, regularly mapped, pages.  **ZONE\_HIGHMEM** This zone contains "[high memory](http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch11lev1sec9.html#ch11lev1sec9)," which are pages not  permanently mapped into the kernel's address space. |
| **copy\_from\_user** | In a virtual memory environment, there’s no guarantee that the whole block of memory  that you pass to write () is actually in RAM at the time. It (or part of it) could be a memory-mapped file, or it (or part of it)  could be paged out.  When any data is passed to the kernel space from userspace, it is the responsibility of the  kernel developer to make sure that everything is sanitized.  The **copy\_to\_user** function copies a block of data from the kernel into user space.  This function accepts a pointer to a user space buffer, a pointer to a kernel buffer, and a  length defined in bytes. After checking the ability to write to the user buffer (through access\_ok),  the page tables could change at any time, requiring the desired pages to be pinned  into memory so that they could not be swapped out while being addressed.  If a kernel can handle page faults, there is perhaps no use ( ignoring gory details like  security for simplicity).  One of the additional things that copy\_from\_user does is disable SMAP .  supervisor Mode Access Prevention) (if enabled) while accessing userspace memory.  If SMAP is enabled, accessing userspace address is not allowed. |
| **Definitions** | **The root filesystem** is the top-level directory of the filesystem. It must contain all the  files required to boot the Linux system before other filesystems are mounted.  The "**sysroot**" is the location the cross compiler will look for header files and libraries.  The sysroot directory acts as if it is the root of the system,  **clangd** understands your C++ code and adds smart features to your editor: code  completion compile errors, go-to-definition and more.  **Electric Make® (”eMake”),** is a new Make version .You can invoke  eMake interactively or through build scripts. Electric Accelerator is a software build  accelerator that dramatically reduces software build times by distributing the  build over a large cluster of inexpensive servers. |
| **Yocto build** | **Yocto Project**: A Linux Foundation project that acts as an umbrella for various efforts to  improve Embedded Linux.  **BitBake:** Bit Bake is a program written in the Python language. At the highest level, Bit Bake  interprets metadata, decides what tasks are required to run, and executes those tasks.  Similar to GNU Make, Bit Bake controls how software is built. GNU Make achieves its  control through "make files". Bit Bake uses "recipes".  **Metadata** is any data that describes other data. Document metadata gives information  About a document such as the author, when it was created when it was last modified,  and its size.  In Yocto project – Metadata is collection of below items   1. Configuration (\*.conf) :Drives the overall behavior of the build process 2. Recipes (\*.bb) : Usually describe build instructions for a single package 3. Append files (\*.bbappend) :Can add or override previously set values 4. Classes (\*.bbclass) :Inheritance mechanism for common functionality .   bbclass) are used to factorize recipe's code, to handle some general problems.  For instance, handling example inherit logging  Image for post  **bitbake-layers show-layers**  Yocto provides the environment for compiling all the packages that are required to  boot a board. It works on the meta layers. Under the meta layers, there are different  different recipes for each package. Under these recipes, there are .bb files for each  package. During compilation, these .bb files are used to get all the information about a  package. This. files contain information like the License, URL to download  the source code, what are the flags should pass at the time of configuration or compilation.  **Yocto terms**  1>**Poky** is the name of build system used by yocto  2> **Layer**: A collection of recipes. Typically, each layer is organized around a specific  theme, e.g. adding recipes for building web browser software. Open Embedded-Core is a  base layer of recipes, classes and associated files that is meant  3> All the artefacts generated are stored in the deploy folder.tmp/log/cooker will have all  logs  4>**Recipe script:** Bit Bake Recipes, which are denoted by the file extension .bb, are the most  basic metadata files. These recipe files provide Bit Bake with the following: version, existing  Dependencies  how to compile .  1>Locate and download source code ,  2>Unpack source into working directory  3>Apply any patches Perform any necessary pre-build configuration  4>Compile the source code  5>Installation of resulting build artifacts in WORKDIR  6>Copy artifacts to sysroot ,Create binary package(s)  We have a bbappend file that supplies a set of patches. It currently has the unintended side-effect of patching both the native version used during the Yocto build process, and the eventual target version. How do I modify the recipe such that it only acts upon the target version?  **SRC\_URI\_append\_class-target** = " file://..."  SRC\_URI Where to obtain the upstream sources and which patches to apply (this is called  “fetching”)  **bitbake-layers create-layer ->**Use a new custom layer for modularity and maintainability.  They all start with “meta-” by convention  **Class files(.bbclass)** extension, contain information that is useful to share between  metadata files. The BitBake source tree currently comes with one class metadata file called  base. bbclass. You can find this file in the class’s directory. The base.bbclass is special since  it is always included automatically for all recipes and classes. This class contains definitions  for standard basic tasks such as fetching, unpacking, configuring (empty by default),  compiling (runs any Makefile present), installing (empty by default) and  packaging (empty by default). These tasks are often overridden or extended by  other classes added during the project development process.  **Append files**, which are files that have the .bbappend file extension, add or  extend build information to an existing recipe file.  **busybox\_1.21.%. bbappend** That append file would match any busybox\_1.21.x.bb  version of the recipe. So, the append file would match the following recipe  names: busybox\_1.21.1.bb busybox\_1.21.2.bb busybox\_1.21.3.bb |
| **NX-Linux (NXL)** | NXL is a Linux Distribution for NXOS based on XE linux distro (Yocto Thud) with  NXOS customizations including GCC 5.2 and Clang 7.0. We're based off of XELinux thud  release. as of right now, we have a snapshot of their layers in our gitlab and are using that.  Some of the layers XELinux uses come from yocto, XELinux may have made changes  on top of those layers. No changes have been directly made to any of the layers coming  from them. If any changes need to be made to the recipes, then there's a bbappend file  in the meta-nx-linux layer, which we created and control. Layer information is on the build  wiki page, and are locally cloned into our gitlab group.  Following git repository mirrors are setup from XE sources - meta-open embedded,  meta-virtualization, meta-security, scripts, meta-nx-Linux (Specific to NXOS for FOSS  customizations  Open-source packages should be sourced from the NX-Linux Distro |
| **initramfs** | initramfs, short for initial RAM filesystem, is a cpio archive of the initial  filesystem loaded into memory after the kernel finishes initializing the system  and before user-space begins the init process. The Linux kernel mounts the  contents of initramfs as the initial root filesystem, before the real root (e.g. on  your hard drive) is mounted. This initial root contains files needed to mount the  real root filesystem and initialize your system—the most important bits being  kernel modules.This feature is made up from a cpio archive of files that enables an initial  root filesystem and init program to reside in kernel memory cache, rather than  on a ramdisk, as with initrd filesystem initrd is for Linux kernels 2.4 and lower. |
| **SPI or I2C? serial port**  **Communication** | 1. I2C is mainly half duplex, that is it uses only one line for sending and receiving data 2. I2C is mainly for master to many slaves’ communication, you can connect up   to 127 slave and one master to control them all   1. I2C can be used to read Temperature. 2. SPI is full duplex, that makes it faster at the same clock speed. 3. while SPI is designed to be a One Master to One Slave communication, adding   another slave will cost you another hardware pin for chip select.   1. SPI can used Refresh a screen. |
| **Kernel interview** | Linux Memory Management  **Dirty bit** means data in cache is modified and the memory has stale copy.  kernel code has to supply its own library implementations (memcpy, crypto,  tar No memory protection, oops. Never use floating point numbers in kernel code.  Your code may need to run on low-end processor without a floating-point unit.  Fixed stack (8 or 4 KB) size Unlike user space, no Swapping, don’t used recursion  User mode driver- written using user space language Perl, propriety, cannot  crash the kernel. can use floating-point computation. Potentially higher  performance. Especially for memory-mapped Devices due to avoidance of  system calls UMD, drawback is Increased interrupt latency Less straightforward to handle  interrupts  Many embedded architectures (x86,ppc) have lot of non-discoverable hardware  (serial, Ethernet, I2C, Nand flash UART controllers, Ethernet controllers, SPI or  I2C controllers, graphic or audio devices. Depending on the architecture, such hardware is  either described in BIOS ACPI tables (x86), using C code directly within the kernel, or using a  special hardware description language in a *Device Tree.*Each node can have a number of  properties describing various properties of the devices: addresses, interrupts, clocks,  power, pin muxing, consumptions etc. At boot time, the kernel is given a compiled version,  the Device Tree Blob, which is parsed to instantiate all devices described in the DT.  -      The Linux USB Subsystem  The USB core now knows the association between the vendor/product IDs    When a USB device is detected with id xxx USB Device controller try to find  matching device driver and Called Probe function . The -probe()  function is responsible for initializing the device , mapping I/O memory,  registering the interrupt and registering it in the appropriate kernel  **Slab allocation** is a [memory management](https://en.wikipedia.org/wiki/Memory_management) mechanism intended for the efficient  memory allocation of kernel objects. It eliminates [fragmentation](https://en.wikipedia.org/wiki/Fragmentation_(computer)) caused by  allocations and deallocations. The technique is used to retain allocated memory  that contains a data object of a certain type for reuse upon subsequent allocations  of objects of the same type. Slab is the original, available since Linux kernel version 2.2.  Slub is the next-generation replacement default since Linux kernel since 2.6.23.  SLOB (Simple List Of Blocks) is a memory allocator optimized for embedded systems with  Low memory footprint  **Modules** are dynamic plugin, stored as a separate file in the filesystem so no  possible With Module reduce boot time and image size, signed modules  Amongst the non-discoverable devices, a huge family are the devices that are  directly part of a system-on-chip: UART controllers, Ethernet controllers, SPI or  I2C controllers, graphic or audio devices, etc. In the Linux kernel, a special bus, called the  **platform bus** has been created to handle such devices.it works like any other bus (USB, PCI), except that devices are  enumerated statically instead of being discovered dynamically  **udev (userspace /dev)** Udev is the device manager for the Linux 2.6 kernel that  creates/removes device nodes in the /dev directory dynamically. It is the  successor of devfs and hotplug. It runs in userspace, and the user can change  device names using Udev rules.    A very important UNIX design decision was to represent most system objects as  Files. It allows applications to manipulate all system objects with the normal file API  (open, read, write, close, etc.)So, devices had to be represented as files to the applications  ls -l /dev/ttyS0 /dev/tty1 /dev/sda /dev/sda1 /dev/sda2 /dev/sdc1 /dev/zero  Example C code that uses the usual file API to write data to a serial port  *int fd;*  *fd = open("/dev/ttyS0", O\_RDWR);*  *write(fd, "Hello", 5);*  *close(fd);*  Within the kernel, all block and character devices are identified using a *major* and  a *minor* number. The *major number* typically indicates the family of the device.  The *minor number* allows drivers to distinguish the various devices they manage.  Most major and minor numbers are statically allocated   1. Dmesg - kernel keeps its messages in a circular buffer 2. CONFIG\_COMPAT is a config flag. 64 bit kernel supports for 32 bit emulation 3. Modern SoCs (System on Chip) include more and more hardware blocks pins are   Multiplexed   1. three types of devices: - network, block(usb,harddisk) ,serial and others (graphics)   all block and character devices are identified using a major and a minor number. –  represent as file   1. "Zero-copy" describes computer operations in which the CPU does not perform   the task of copying data from one memory area to another.   1. Kmalloc calls slab page is usually 4K, but can be 8k 16k,PIO - IN and OUT instructions 2. Kernel pre-emption, if enabled, causes the kernel to switch from the execution 3. Mutex - The kernel’s main locking primitive. It’s a binary lock , mutex\_trylock   Use mutexes in code that is allowed to sleep -not in spinlock   1. Spinlocks cause kernel pre-emption to be disabled on the CPU executing them,   No sleeping, several variants like Doesn’t disable interrupts, Disables software  interrupts, but not hardware ones   1. lock-free algorithms -rcu lock, atomic instructions 2. Some device controllers embedded their own DMA controller, DMA deals with   physical addresses But the DMA does not access the CPU cache, so one needs to  take care of cache coherency (cache content vs. memory content). |
| **interrupt handler** | Each device must register its interrupt handler. whenever an interrupt occurs the  OS does the most important part of handler "upper half" to respond to interrupt,  create a data structure containing device specific data called "**lower half**” for  later processing when CPU becomes available. This way interrupt handlers can be  used in Bottom halves are required because as we know when ISR executes,  it disables all other interrupts on running processor and same interrupt on all  processors. To increase the response time and throughput of the system, we need  to finish ISR as soon as possible.  Acknowledge the interrupt to the device (otherwise no more interrupts will be  generated, or the interrupt will keep firing repeatedly  滴水穿石- OSDI - Viva La Vida  The softirqs handlers are executed with all interrupts enabled. They are executed once all  interrupt handlers have completed, before the kernel resumes scheduling processes,  The number of softirqs is fixed in the system, so softirqs are not directly used by  drivers, but by complete kernel subsystems (network, etc.)  HI\_SOFTIRQ and TASKLET\_SOFTIRQ are used to execute tasklets,Example usage of softirqs – NAPI  Work queues typically be used for background work with can be scheduled. |
| **Linux memory** | **swap: The** primary function of swap space is to substitute disk space for RAM memory  when real RAM fills up and more space is needed.    Using **ramfs or tmpfs** you can allocate part of the physical memory to be used as a partition. You can mount this  partition and start writing and reading files like a hard disk partition. Since you’ll be  reading and writing to vivthe RAM, it will be faster.  Non-Uniform Memory Access is a computer memory design used in  multiprocessing, where the memory access  time depends on the memory location relative to the processor.  Allocating kernel memory (buddy system and slab system)  The main drawback in buddy system is internal fragmentation as  larger block of memory is acquired then required.  cat/proc/buddyinfo displays as follows:  Node 0, Zone DMA 0 4 5 4 4 3 ...  Node 0, Zone Normal 1 0 0 1 101 8 ...  Node 0, Zone highmem 2 0 0 1 1 0 ...  static memory allocation linux kernel : boottime by driver reserve contagious memory  A second strategy for allocating kernel memory is known as slab allocation. It eliminates  fragmentation caused by allocations and deallocations. A slab is made up of one or more  physically contiguous pages. cat /proc/slabinfo |
| **Linux Scheduling Policy** | Linux supports real-time scheduling out of the box. There is a misconception that Linux has  to be patched to provide support for real-time scheduling. The only issue is that the  scheduling latencies may not satisfy the hard real-time requirements of critical applications.  There are patches that try to address this, like the CONFIG\_PREMPT\_RT patch  We have two categories of scheduling policies. Normal and real-time. Real-time scheduling  policy has two sub-types, round-robin and first-in first-out, identified by SCHED\_RR and  SCHED\_FIFO.The **sched\_setscheduler()** system call set scheduling policy of thread to real  Time SCHED\_FIFO ,SCHED\_RR .  SCHED\_FIFO and SCHED\_RR priorities allow priorities from 1 to 99. SCHED\_OTHER, which is  the default supports only the value of 0. In case of SCHED\_FIFO, for tasks of the same  priority, the currently running task has to yield before the next one can run .  Tasks of the same priority when running with RR\_SCHEDULING will get an equal interval run.  Default Priority is 20 with nice value 0  Linux Kernel implements two separate priority ranges –   * **The nice value range** is -20 to +19 where -20 is highest, 0 is default/   Nice value: minus 20 to plus 19; larger (+19) nice correspond to lower priority.   * Real-time priority: 0 to 99; higher real-time priority values correspond to   a greater priority. **PR = 20 plus nice ,**  0 is default nice priority of process |
| **Packet journey** | The high-level path a packet takes from arrival to socket receive buffer is as follows:  Driver is loaded and initialized.  **Receive Side**   1. Packet arrives at the NIC from the network. 2. Packet is copied (via DMA) to a ring buffer in kernel memory. 3. Hardware interrupt is generated to let the system know a packet is in memory. 4. Data that was DMA’d into memory is passed up the networking layer as an ‘skb’ for   more processing.   1. Tapping in eth layer (tcp dump), net filter in ip layer, state machine in tcp layer later   queue to socket and copy to application and invoke poll.   1. Tcp maintain state machine, fragmentation and assembly 2. Skb and packet memory is free after socket read and copy to user memory   **send side**  1. skb is allocated in TCP layer and enqueue in write queue, empty space in  beginning of skb. Mac or neighbor discover if not cached, ipl layer net filter on send side  2. Add Checksum before sending it. Nic does frame checksum on both recv and send side  ,nic checksum offload  3. Modern nic has more hardware queue (faster) , in recv side packet belong to same  Stream Goes to same queue useful in SMP  NAPI = New API, Principe: when the network traffic exceeds a given threshold (”budget”),  Disable network interrupts and consume incoming packets through a polling function,  instead of processing each new packet with an interrupt.  incoming network data frames are distributed among multiple CPUs if packet  steering is enabled or if the NIC has multiple receive queues. |
| **UBOOT** | U-Boot is both a first stage and second-stage bootloader. If there are size  constraints, U-Boot may be split into stages:  U-Boot performs both first-stage  (e.g., configuring memory controllers and SDRAM) and second-stage booting. ,  U-Boot Linux booting requires its boot commands to explicitly specify the physical memory  addresses as destinations for copying data (kernel, ramdisk, device tree, etc.)  and for jumping to the kernel and as arguments for the kernel |
| **Linux booting facts** | Hot plug is the addition of a component to a running computer system without  significant interruption to the operation of the system. Hot plugging a device  does not require a restart of the system. |
| **Synchronization mechanisms inside Linux kernel** | **Critical Region:** A critical section is a piece of code which should be executed  under mutual exclusion. Suppose that two threads are updating the same  variable which is in parent process’s address space  **Atomic operation:** This is the very simple approach to avoid race condition or  deadlock. Atomic operators are operations, like add and subtract,  which perform in one clock cycle (uninterruptible operation). and another  that operates on individual bits. All atomic functions are inline functions.  **Semaphore**: This is another kind of synchronization mechanism which will be  provided by the Linux kernel. When some process is trying to access  semaphore, which is not available, semaphore puts process on wait queue (FIFO)  and puts task on sleep. That’s why semaphore is known as a sleeping lock.  After this processor is free to jump to another task which is not requiring this semaphore. As soon as semaphore get available, one of task from wait queue in  invoked. There two flavors of semaphore is present.Basic semaphore **Reader-Writer**  **Semaphore**  **Semaphore** puts a task on sleep. So, the semaphore can be only used in process  context. Interrupt context cannot sleep. Operation to put task on sleep is time consuming(overhead) for CPU. So  semaphore is suitable for lock which is holding for long term. A code holding a semaphore  can be preempted. It does not disable kernel preemption.  After disabling interrupts from some tasks, semaphore should not acquire.  Because task would sleep if it failed to acquire the semaphore, at this time the  interrupt has been disabled and current task cannot be scheduled out.  **Semaphore wait list is FIFO in nature**. So, the task which tried to acquire semaphore  first will be waken up from wait list first.. Semaphore can be acquired/release from any process/thread.  **Spin-lock**: This is special type of synchronization mechanism which is preferable to use in  multi-processor (SMP) system. Basically, its a busy-wait locking mechanism until the  lock is available. In case of unavailability of lock, it keeps thread in light loop and keep  checking the availability of lock. Spin-lock is not recommended to use in single processor  system. If some procesq\_1 has acquired a lock and other process\_2 is trying to acquire  lock, in this case process 2 will spins around and keep processor core busy until it acquires  lock. process\_2 will create a deadlock, it dosent allow any other process to execute because CPU core is busy in light  loop by semaphore.  Couple of observations about nature of spinlocks:   1. Spinlocks are very much suitable to use in interrupt(atomic) context because it   doesn’t put process/thread in sleep.   1. In the uni processor environment, if the kernel acquires a spin lock, it would   **disable preemption first** ; if the kernel releases the spin lock, it would enable  preemption. This is to avoid dead lock on uni processor system   1. Spin-locks is not recursive. A thread may call lock on a recursive mutex   repeatedly. Ownership will only be released after the thread makes a matching  number of calls to unlock  4. Special care must be taken in case where spinlock is shared b/w interrupt handler  and thread. Local interrupts must be disabled on the same CPU(core) before acquiring  spin-lock. In the case where interrupt occurs on a different processor, and it spins on the  same lock, does not cause deadlock because the processor who acquire lock will be able to  release the lock using the other core.   1. When data is shared between two tasklet, there is not need to disable interrupts   because tasklet dose not allow another running tasklet on the same processor.  There two flavors of spin-lock is present.  Basic spin-lock  Reader-Writter Spin-lock  With increasing the level of concurrency in Linux kernel read-write variant of spin-lock is  introduces. This lock is used in the scenario where many readers and few writers are present.  Any reader will not get lock until writer finishes it.  **Sequence Lock**: This is very useful synchronization mechanism to provide a lightweight  and scalable lock for the scenario where many readers and a few writers are present.  Sequence lock maintains a counter for sequence. When the shared data is written, a lock is  obtained and a sequence counter is incremented by 1. Write operation makes the sequence  counter value to odd and releasing it makes even.   1. atomic variables and cpu local variables 2. spin locks (and reader/writer locks) 3. sequential locks 4. RCU locks 5. mutex 6. semaphores 7. completions 8. waitqueues 9. • memory barriers |

|  |  |
| --- | --- |
| Memory Layout of C Programs | <https://cdncontribute.geeksforgeeks.org/wp-content/uploads/memoryLayoutC.jpg>  **From low to high memory**  **1 Text Segment:** contains executable instructions.  **2. Data Segment:** contains global & static variables  3. heap goes up  4. stack goes down |

|  |  |
| --- | --- |
| **Mutex V/s Semaphore** | 1> In mutexes has an ownership property, only the thread that took the  lock has the key. Only that thread alone can release the lock. Binary semaphores  doesn’t have an ownership property, as any thread can take the key to open the  lock.  2)A semaphore is a generalized mutex. In lieu of single buffer, we can split  the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore  can be associated with these four buffers. The consumer and producer  can work on different buffers at the same time. |
| **Dockers** | A software container is a way to bundle and isolate processes (software) running  on a server. Virtual machines and containers differ in several ways, but the primary  difference is that containers provide a way to virtualize an OS so that multiple  workloads can run on a single OS instance. With VMs, the hardware is being  virtualized to run multiple OS instances.  Docker is an open source project that makes it easier to create, deploy and run  Applications in containers .The applications are packaged in a docker container  which contains all the dependencies (libraries, packages) that are needed to  deploy the application. By using docker, an application can be easily  moved around from the developer’s laptop, into the testing  environment and finally into production.  Podman, new in Red Hat Enterprise Linux 7.6 Beta, can replace the docker CLI,  allowing you to run standalone (non-orchestrated) containers ... |
| **Docker/ Kubernetes** | |  | | --- | | Docker is what enables us to run, create and manage containers on a single host  Kubernetes can then allow you to automate container provisioning, networking,  load-balancing, security and scaling across all these nodes from a single command line  or dashboard.  A collection of nodes that is managed by a single Kubernetes instance is referred to  as a **Kubernetes cluster**.  Now, why would you need to have multiple nodes in the first place? The two main  motivations behind it are:  1. To make the infrastructure more robust: Your application will be online, even if  2. some of the nodes go offline, i.e, High availability.  3. To make your application more scalable: If workload increases, simply spawn more  4. containers and/or add more nodes to your Kubernetes cluster.  Kubernetes automates the process of scaling, managing, updating and removing  containers. In other words, it is a container orchestration platform. While Docker is  at the heart of the containerization, it enables us to have containers in the first place.  Differences Between Kubernetes and Docker In principle, Kubernetes can work with  any containerization technology. | | **Kubernetes pods** :A Kubernetes pod is a group of containers that are  deployed together on the same host. If you frequently deploy single container  s, you can generally replace the word "pod" with "container" and accurately  understand the concept.  **Docker** is a run time engine running on your computer. It’s a daemon that is  in charge of containers start, stop on that single computer. So Docker is  about managing works within a single machine.  **Kubernetes is kind of a cluster management software**. It is a group of  daemons that is in charge of a cluster of machines. Though there is a single  daemon (kubelet) running on an individual machine, the kubelet by itself  does not have much value on the table; it is these group of kubelets (  along with kubernetes controllers that control them) make decisions about  the whole cluster. So k8s is about managing works for a cluster of machines. | |
| **ASLR** | Address space layout randomization (ASLR) is a memory-protection process for  operating systems (OSes) that guards against buffer-overflow attacks by randomizing  the location where system executables are loaded into memory.  The success of many cyberattacks, particularly zero-day exploits, relies on the  hacker's ability to know or guess the position of processes and functions in memory.  ASLR is able to put address space targets in unpredictable locations. If an  attacker attempts to exploit an incorrect address space location, the target application  will crash, stopping the attack and alerting the system.  ASLR works alongside virtual memory management to randomize the locations of  different parts of the program in memory. Every time the program is run, components  (including the stack, heap, and libraries) are moved to a different address in virtual |
| **Object Size Checking (OSC)** | Object Size Checking (OSC) leverages a builtin compiler technique to determine  buffer overflows in C/C++ code. various optimization passes enabled with -O2 |
| **xspace** | Making the stack (and heap) non-executable provides a high degree of protection  against many types of buffer overflow attacks for existing programs.  is that execution occurs in the code section, which is neither stack nor heap. |
| **Reader-writer lock** | When a writer is writing the data, all other writers or readers will be blocked  until the writer is finished writing. Readers–writer locks are usually constructed  on top of [mutexes](https://en.wikipedia.org/wiki/Mutex) and  [condition variables](https://en.wikipedia.org/wiki/Condition_variable), or on top of [semaphores](https://en.wikipedia.org/wiki/Semaphore_(programming)). |
| **STACK** | **Key Differences Between Stack and Heap Allocations**   1. In a stack, the allocation and deallocation is automatically done by   whereas, in heap, it needs to be done by the programmer manually.   1. Memory shortage problem is more likely to happen in stack whereas   the main issue in heap memory is fragmentation.   1. Stack is not flexible, the memory size allotted cannot be change |
| **OS concept** | Page Fault: A page is a fixed-length block of memory that is used as a unit of  transfer be­tween physical memory and external storage like a disk, and a  page fault is an interrupt (or exception) to the software raised by the hardware  when a program accesses a page that is mapped in address space, but not  loaded in physical memory. |
| **DMA** | Direct Memory is a feature which provides direct access (read/write) to system  memory with­out interaction from the CPU. using “DMA Controller” |
| **STACK protection** | in a multi-threaded environment, there can be multiple stacks in a process.  One threat to the stack is malicious program input, which can overflow a buffer  and over­write stack pointers, simple method GCC, you use -fstack-protector-all. |
| **vmalloc** | vmalloc allocates virtually contiguous memory space (not necessarily physically  contiguous), while kmalloc allocates physically contiguous memory (also virtually  contiguous). Most of the memory allocations in Linux kernel are done using kmalloc,  due to the following reasons:  On many architectures, hardware devices don’t understand virtual address. Therefore,  their device drivers can only allocate memory using kmalloc.  kmalloc has better  performance in most cases because physically contiguous memory region is more  efficient than virtually contiguous memory. interval of time. |
| **Kernel mode** | Kernel mode  -----------  Enter using interupt/Trap  1. Access to privileged instructions  --> CPU control instructions (CLI, STI, HLT, WAIT, LOCK, ...)  --> IN, OUT (direct hardware access)  2. Full access to physical memory (RAM)  User mode  ----------  1. Restricted instruction set  2. No direct hardware access  3. No access to entire physical memory (RAM)  4. Memory access only by virtual addresses (Virtual memory)  5. Memory access can happen via demand-paging |
| How system call works | 1. Application program makes a system call by invoking wrapper function in C library 2. This wrapper functions makes sure that all the system call arguments are available   to trap-handling routine   1. Generally, a stack is used to pass these arguments to wrapper function. But the   Kernel looks into specific registers for these arguments. Hence the wrapper  function also takes care of copying these arguments to specific registers   1. Each system call has a unique call number which is used by kernel to identify   which system call is invoked? The wrapper function again copies the system call  number into specific CPU registers   1. Now the wrapper function executes trap instruction (int 0x80). This instruction   causes the processor to switch from 'User Mode' to 'Kernel Mode'   1. The code pointed out by location 0x80 is executed (Most modern machines use   sysenter rather than 0x80 trap instruction)   1. In response to trap to location 0x80, kernel invokes system\_call() routine which is   located in assembler file arch/i386/entry.S (also called handler)   1. This **handler saves register values** onto kernel stack and does some validations like   verifying system call number etc.   1. A map of system call number as key and the appropriate system call as value exists. This is called system\_call\_table. The handler uses this table to invoke appropriate system call service routine. It also validates the arguments if present.   After proper validations, the service routine performs required actions like modify  values at addresses specified in arguments or transfer data between user memory and  kernel memory. After all these actions, service routine returns status of execution to  the system\_call routine   1. Now the handler restores register values from kernel stack and places the system call   return value on the stack   1. Thus handler is returned to wrapper function, simultaneously returning processor to   user mode   1. Just in case if the return value of system call service routine indicated an error, then   wrapper function sets 'errno' a global variable and then returns to caller providing  integer return value that indicates the status of execution |

|  |  |
| --- | --- |
| Linux booting | **Stage 1**  When a system is booted, Processor executed a code from a well-known  location known as BIOS (Basic Input Output System) which is stored in flash  memory of motherboard. Its Job is to find the boot device (floppy/hard disk, cd).  When boot device is detected, it passes control to first stage bootloader.  A master boot record (often shortened as MBR) is a kind of [boot sector](https://www.lifewire.com/what-is-a-boot-sector-2625815) stored on a hard  disk drive or other storage device that contains the necessary computer code to start the  [boot](https://www.lifewire.com/what-does-booting-mean-2625799) process.  **Stage 2**  **The first-stage loader (stage1) is loaded into the RAM and executed by the BIOS from the**  **Master boot record (MBR).** This Boot Loader is 512 bytes in size with 64 bytes partition  table). Its job is to find the SECOND order Boot Loader (grub) and load it into RAM and  passed control to 2nd stage bootloader.  **Stage 3**  Grub1 is is embedded in an MBR (size issue).  Grub2 is knowledge about the Linux file system (ext2,ext3)  Grub2 copy the Linux kernel image into the RAM using /boot/grub/grub.conf..  **Step 4 Kernel stage**  Kernel is in compressed **cpio format** file present in /boot directory .  Mounts the root file system as specified in the “root=” in grub.conf  grub> root (hd0,0)  grub> kernel /vmlinuz-i686-up-4GB root=/dev/hda9  grub> boot  kernel /vmlinuz-i686-up-4GB root=/dev/hda9 - Specifies the kernel location  which is inside the /boot folder. This location is related to the root(hd0,0)  statement.The root partition is specified according to the Linux naming  convention (/dev/hda9/)  initrd/ Initramfs is used by kernel as temporary root file system until kernel is booted  and thereal root file system is mounted. It also contains necessary drivers compiled  inside,which helps it to access the hard drive partitions, and other hardware.  insmod for loading kernel modules, and lvm (logical volume manager tools).  Initramfs/intrd is an image file in /boot containing the basic root file system with all  Kernel modules. The kernel then Mount this image file as a starting memory-based  root file system. The kernel then starts to detect the system’s hardware. The root file  system on disk takes over from the memory. The boot process then starts  INIT (SYSTEMD)  **Step 5 INIT**  The kernel, once it is loaded in step 4, it finds init in sbin (/sbin/init) and  executes it. The first thing init does is reading the initialization file, /etc/inittab.  The program init is the process with process ID 1. |
| **Hardware Security** | Secure Boot is a security standard developed by members of the PC industry to help makesure that your PC boots using only software that is trusted by the PC manufacturer. Whenthe PC starts, the Bios checks the signature of each piece of boot software, including driversand the operating system. If the signatures are good, the PC boots, and the Bios givescontrol to the operating system or else it would halt the boot up process and thrown error. A **Trusted Platform Module (TPM)** is a hardware chip on the computer’s motherboard that  stores cryptographic keys used for encryption. Once enabled, the Trusted Platform  Module provides full disk encryption capabilities. It becomes the "root of trust" for the  system to provide integrity and authentication to the boot process. It keeps hard drives  locked/sealed until the system completes a system verification, or authentication check.  The TPM includes a unique RSA key burned into it, which is used for asymmetric encryption. Additionally, it can  generate, store, and protect other keys used in the encryption and decryption process.  A hardware security module (HSM) are external devices connected to a network using  TCP/IP.encryption capabilities by storing and using RSA keys. |
| **ebpf** | eBPF is a revolutionary technology with origins in the Linux kernel that can run  sandboxed programs in an operating system kernel. It is used to safely and efficiently  **extend the capabilities of the kernel** without requiring changing kernel source code  or load kernel modules. Seguridad de red de microservicios basada en eBPF - Code World |
| **DPDK** | **DPDK (Data Plane Development Kit)** is a framework (under the Linux Foundation)  comprised of various userspace libraries and drivers for fast packet processing.  Originally developed by Intel to run on x86 based CPUs, DPDK now supports other  CPU types. DPDK leverages existing Intel Processor technologies like SIMD instructions  (Singles Instruction Multiple Data), Huge-pages memory, multiple Memory channels  and Caching to provide acceleration with its own libraries.  Though DPDK uses a number of techniques to optimise packet throughput, how it  works (and the key to its performance) is based upon Fast-Path and PMD.  **Fast-Path (Kernel bypass)** - A fast-path is created from the NIC to the  application within user space, in turn, bypassing the kernel. This eliminates  context switching when moving the frame between user space/kernel space.  Additionally, further gains are also obtained by negating the kernel  stack/network driver, and the performance penalties they introduce.  **Poll Mode Driver** - Instead of the NIC raising an interrupt to the CPU when a frame is  received, the CPU runs a poll mode driver (PMD) to constantly poll the NIC for new  packets. However, this does mean that a CPU core must be dedicated and assigned to  running PMD. |
| **KSM** | KSM (kernel samepage merging) is a Linux kernel feature that allows share identical  memory pages among different process or virtual machines on the same server.  **User kernel tracing with ftarce to get back trace**  **https://blog.selectel.com/kernel-tracing-ftrace/** |
|  | The Intel VT-d extensions provides hardware support for directly assigning a  physical devices to guest. The main benefit of the feature is to improve the performance  as native for device access.  The VT-d extensions are required for PCI passthrough with Red Hat Enterprise Linux.  The extensions must be enabled in the BIOS. Some system manufacturers disable these  extensions by default.  These extensions are often called various terms in BIOS which differ from manufacturer  to manufacturer  Activate Intel VT-d in the kernel by appending the intel\_iommu=on parameter to the  kernel line of the kernel line in the /boot/grub/grub.conf file.  he AMD IOMMU extensions are required for PCI passthrough with Red Hat Enterprise Linux. |
| **barrier** | A memory barrier, also known as a membar, memory fence or fence instruction, is a type of  barrier instruction that causes a central processing unit (CPU) or compiler  to enforce an ordering constraint on memory operations issued before and after the  barrier instruction. This typically means that operations issued prior to the barrier  are guaranteed to be performed before operations issued after the barrier. he following two-processor program gives an example of how such out-of-order execution can affect program behavior:  Initially, memory locations x and f both hold the value 0. The program running on processor #1 loops while the value of f is zero, then it prints the value of x. The program running on processor #2 stores the value 42 into x and then stores the value 1 into f. Pseudo-code for the two program fragments is shown below. The steps of the program correspond to individual processor instructions.  Processor #1:  while (f == 0);  // Memory fence required here  print x;  Processor #2:  x = 42;  // Memory fence required here  f = 1; |
| **fork vs/ vfork** | During the fork() system call the Kernel makes a copy of the parent process’s address space  and attaches it to the child process.  But the vfork() system call do not makes any copy of the parent’s address space, so it is  faster than the fork() system call. The child process as a result of the vfork() system call  executes exec() system call.  The child process from vfork() system call executes in the parent’s address space  (this can overwrite the parent’s data and stack ) which suspends the parent process  until the child process exits. |
|  | Tasklet Priority Softirq Description  HI\_SOFTIRQ 0 High-priority tasklets  TIMER\_SOFTIRQ 1 Timer bottom half  NET\_TX\_SOFTIRQ 2 Send network packets  NET\_RX\_SOFTIRQ 3 Receive network packets  SCSI\_SOFTIRQ 4 SCSI bottom half  TASKLET\_SOFTIRQ 5 Tasklets  DMA controller is a device which directly drives the data and address bus during data  transfer. So, it is purely Physical address. (It never needs to go through MMU & Virtual  addresses). |