→ Canonical Interview.

Linux Operating System.

1) KERNELS

* Core component of linux OS.
* Manages hardware and software interactions.
* enables multitasking and resource management.

kernel subsystem.

→ Process Management ✓
→ Memory Management. ✓ (MMU)
→ Virtual file System. (VFS)
→ Network Stack. ✓
→ Device drivers. ✓
→ IPC unit.

Q. What is a kernel Module?

ans: A piece of code that can be loaded and unloaded into kernel upon demand.

extend funcunality without rebooting systems

eg: device drivers.

→ lsmod

list the module in kernel.

Process of loading.

(a) not present feature
↓
(b) Search for it using name. (modprobe)
↓
(c) Check if it needs dependencies.
↓
(d) load dependency modules
↓
(e) then load main module.

* Ubuntu Core.

Kernel Arch Kernel
↳ Kernel Arch.

→ Centeral Program managing all hardware and software resources.

→ Provides services to user program.

eg: file, scheduling, memory managment

Monolithic kernel. (linux)

→ Most services run in kernel space rather than as a seperate processes.

→ linux also supports loadable kernel module
↳ add/Remove functionality (like drivers)

Shows of kernel is structured & different subsystems interconnect.

→ Linux Kernel development

Introduction to Linux Kernels

* Everything is a file & written in C making Unix highly portable.
* unique fork() System call
* has only (100's) System calls.

→ Linux (1991)
Linus Torvalds. for Intel 80386

Linux System → { kernel ✓, C library ✓, Toolchain ✓, basic System utilities ✓

linux → refers to → kernel

Kernel : Inner most layer of OS

→ components of kernel
1) Interrupt handler — ISR
2) Service Interrupt requests
3) Scheduler
4) Memory management System
5) System Services
   ↳ networking
   ↳ IPC

Intr

Kernel manages System hardware

Hardware → Interrupts (id).
↑ ↓
Interrupt handler — Processor
↓ tells
Kernel

{ Interrupts run in Interrupt context and not processor context }

↳ This exists only to quickly handle interrupts

CPU

A proccesor at any point is doing 1 of 3 things.

* In user-space, executing user code.
* In kernel-space, process context
* In Kernel-Space, ~~kernel~~ interrupt context

eg: idle process

↳ Kernel space, process context.

Imp { Linux usually requires MMU, but it has the capability to run MMU-less embedded systems.

Nowadays (ES) also have MMU.

↓

Complex

Linux is monolithic → executes in sigle address spa
→ entierly in kernel mode.
→ Kernel preemption
→ Kernel threads.
→ dynamically load seperate binaries (kernel modules)
→ kernel is schedulable.

Imp The kernel source is usually installed in

(usr/src/linux) usr/scr/linux

Should remain untouched

→ configuration option are either boolean or tristate.

yes | no | module

Drivers are usually tristate.

Ubuntu kernel

Cross section of kernel features and compile all the drivers as modules

/proc/config.gz

Compressed configuration file is stored here

build it with Make.

Make file handels dependencies.

→ To build kernel with multiple make jobs.

make -jn

/boot : store the kernel

/lib/modules : have all installed modules

→ Extra characterstics.

1) Kernel is coded in C.
2) Can not easily execute floating up. /
3) small pre process fixed size stack.

linux → parallelism
→ concurrency.

* Kernel memory is not pageable
* Memory violation in kernel can result in major kernel error.

→ linux supports symmetrical multiprocessing
↳ preemptive multitasking operating System.

Process Management
→ linux uses a "Completely fair Scheduler" (CFS)
↳ based on (a) Priority
(b) time slicing. RR

Real-time Scheduling.
↳ FIFO ↳ RR.

Memory Management
linux uses virtual memory.
→ paging
→ swapping
→ allocation.

ISR (IDT)

Interrupts

→ Kernel has (ISR) to handle interrupts
  ↳ interrupts are high priority.
    ↓
    Interrupt vector.
    (table to identify the interrupt)

Top half & bottom half
  ↳ minimal work done upon interrupt.
    ↓
    work done later ←
    that requires extensive processing.

Kernel Boot Process

1) Firmware / BIOS / UEFI
   ↳ intialises the hardware and loads the bootloader.

2) Bootloader (GRUB, LILO)
   ↳ loads kernel image & an optional initial ram disk

Kernel Initialisation
  ↳ kernel decompresses
    ↳ sets up basic hardware and memory.
      ↳ Starts first user space process
        (init or Systemd)
        ↳ handles services and user-level start up scripts
        until system is fully operational

Process Management

Struct task_struct (PCB)

↳ holds process meta data
(PID, state, priority etc)

Process States

Task_running → READY
Task_Interruptable → WAITING_STATE

CREATION fork() exec()

→ fork() → duplicate of parent process.
Copy-on-Write.

Clone() sys call

exec() → replaces the process image with new executable.

Process Scheduling

Completly Fair Scheduler (CFS)
2.6.23

↳ uses red-black tree to track runnable task.

red-black tree

→ Allocate CPU time ∝ to task priority.

System Calls

→ user invokes software interrupt.
→ system call number passed via register

Kernel Data Structure

stored in /proc

linked list
circular, doubly Struct list_head

Queue
for FIFO operations.

Maps.
idr for ID allocation

Red-black tree
↳ self balancing BT for efficient lookups → (CFS)

Interrupts and Interrupt handlers

Top half → immediate
bottom half → minimal work.

Interrupts

↳ run in atomic context, can not sleep or access user-space.

Kernel Synchronization

→ Spinlocks → for shorter critical section.
→ Mutex → for longer sections.
→ Semaphore.
↳ Counting mech for resource access.

~~Timer and Time Management~~

Spinlocks

Memory Management

→ Pages → physical memory divided. 4 KB

Zones
↳ DMA, NORMAL, HIGHMEM.

→ Page allocator.
↳ large chunks.

→ Slab allocator
↳ Cache for frequently used objects.

Virtual memory

address space managed via struct mm_struct.

→ Technical Deep Dive

→ Kernel

* Interface b/w user programs and hardware.
It is also the core part of the OS.

- In linux the linux OS is usually refered to as kernel

→ Subsystems

Process Management
- handles process lifecycle and scheduling of all processes.

Memory Management Unit
- handle memory allocation, physical memory or virtual memory

Virtual File System
- Provides more like API or syscalls for physical storage devices

Networking Stack
- Responsible for how a user processes data packets from user space all the way down to the kernel

* Ubuntu
  → upstream kernel with modifications
everyone contributes to

Device Driver
- handles different devices in the kernel or attached to the system

Arch Specific logic
- processor specific logic present in the kernel code.

Modifying the linux kernel (C, Rust)

enable/disable features
Optimise kernel performance and size
add support for new hardware.
apply security features.

* make **menuconfig** (GUI)....
* make **defconfig** (default configuration)

1) Start by building your kernel. (make -j×)
2) Install the kernel
3) Running your kernel.

Test and debug

* **What is the change you made**
* **dmesg log**
  → Captures everything coming out of the kernel.

Automated testing

→ linux test Project (LTP)
  ↳ 3-4 hrs.

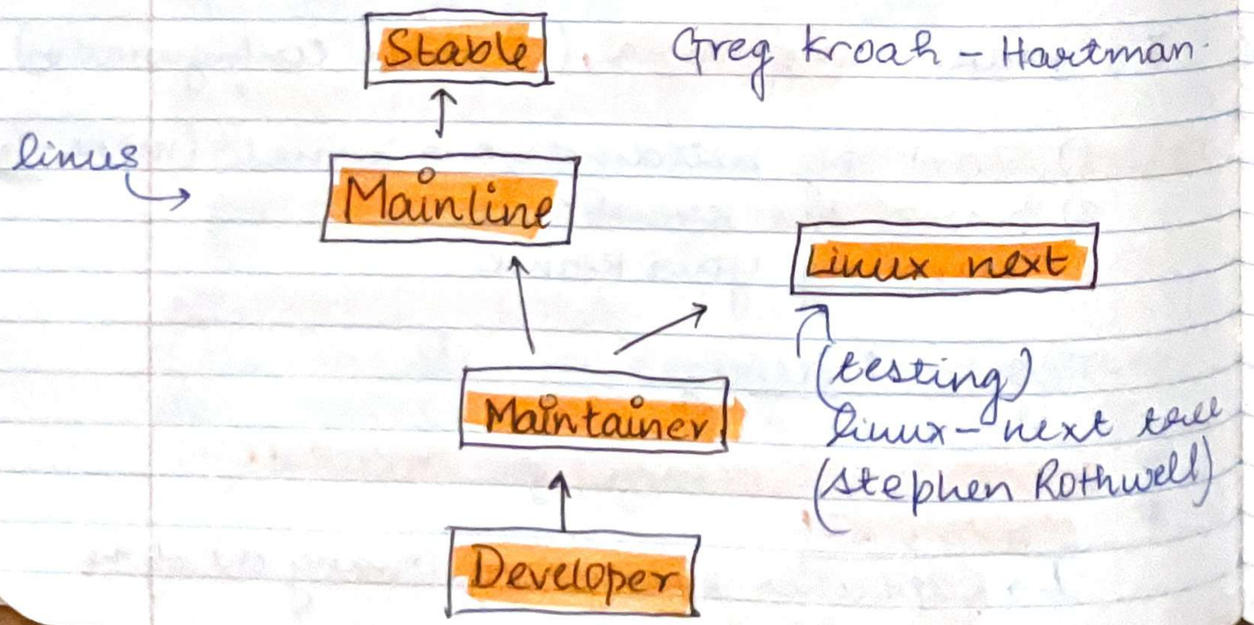
→ kselftest (test)
  ↳ part of linux kernel.

Imp

debugging

- printk().
- ftrace: trace of which func called this one.
- dump stack: prints stack tree.

linux Kernel development Process



* The linux kernel is both monolithic and modular since it can insert and remove kernel modules at runtime.

* Monolithic Kernel
  ↳ All the kernel code is running in a single address space (or) kernel code running in kernel space

Microkernel
  ↳ * Main kernel code is running in kernel space and DD and modules are running in user space.
  Only load (files) → DD and modules when needed

Hybrid

linux (current) → Monolithic + you can add DD & modules

menuconfig
* → built-in
M → load/make as module
. → disable

Steps to configure and compile linux kernel

1) get linux source code (git)
2) install dependencies (apt update)
3) choose a default configuration (defconfig)
4) clean files of old dependencies (make clean)
5) configure the kernel (make menuconfig)
6) Compile the kernel (make -j$(nproc))
7) make modules
8) Install the kernel (sudo make modules_install)
9) ~~Reboot~~ update GRUB.
10) Reboot.

Q What is the role of OOMKiller.

ans: Out-of-memory killer is a feature in linux kernel designed to handle situation when the system runs out of available memory.

Terminates process by weighing pro's and con's
→ less critical
→ uses most memory.

Q. What is kernel panic?

ans:
- fatal error that linux kernel encounters and recover from.
- Happens when the kernel detects an inconsistency or an issue that severely impact system stability.
- Often requires manual restart.

Q What is loadable kernel module (LKM)?

ans: Specific component of kernel that can be dynamically loaded and unloaded runtime.

Doesn't require system to be rebooted. to add or remove at runtime.

Q How does the kernel handle security and permissions?

ans: Uses mechanisms like user account, access control list and security module.

→ Makes sure that users and processes access resources they are authorized to use.

Q What is Kernel Preemption.

ans: Ability of kernel to interrupt a running process to switch to a higher priority task.

Imp because gives kernel access to handle crucial tasks first.

Q How does kernel enforce memory protection between different processes?

ans: (1) Virtual memory and page table.
(2) Privilege levels
(3) Copy - on - write.
optimise memory sharing b/w forked processes.
(4) Page Permissions
(5) Segmentation faults
↳ block unauthorized access to memory.
(6) IPC.

5) Role of kernel managing i/p and o/p operations.

ans: ~~Disk scheduling algorithms~~.

- ↳ FCFS
- ↳ SSTF
- ↳ SCAN
- ↳ LOOK
- ↳ C-SCAN
- ↳ C-LOOK.

end
last read

ans: By using system calls(), device drivers to access HW. VFS, disk scheduling. buffering and caching, interrupt handling and direct memory access.

Q Handling System Calls in multithreaded environment?

ans: if a system call blocks, only the calling thread is blocked, not the entire process. Some system calls are thread safe that means they can be executed by multiple threads.

eg: Read()

Imp

Q How does the kernel handle real-time processing requirments.

ans: By using RT scheduling policies Rate monotonic scheduling and Earliest deadline first.

Q How does the kernel ensure data integrety in file systems.

ans: kernel uses mechanism such as journaling, checksum, redunding

RAID type stuff.

Imp

Q. How does kernel secure a system against Vulnerabilities?

ans: (1) access control
(2) enforcing memory protection.
(3) Isolating resources
(4) SE linux
(5) Apparmor.

= Kernel updates and patches.
= Kernel address space layout Randomization

Q. How does kernel handle hardware interrupts. reduce int latency

ans: Interrupt handler are used, which are special function that executs in response to an interrupt signal. They ensure timely processing of events.

Interrupt Descriptor table and how the kernel maps interrupts to their resp handler.

imp

Q. How does kernel manage process Isolation and resource allocation in Containers?

ans: namespaces and Cgroups
↳ isolation
↳ of
↳ processid
network interface
file system.

Cgroups
↳ control resource utilisation.

Q. How does kernel handle priority inversion.

ans: Priority Inheritance
↳ low priority task gets high priority

Priority ceiling
↳ resource access restricted to higher priority task.

Kernel → Kernel Arch

Q. Fork() bomb?

ans: Fork bomb ~~boom~~ spawns excessive child processes, depleting system resources.

mitigate this by ulimit -u

Q. Device hotplugging?

ans: ~~The kernel allows~~

This allows hardware component to be added or removed while the system is running without requiring boot.

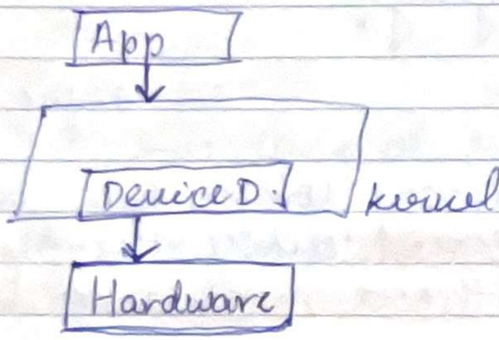
Kernel handles this using
~~dt~~
dynamic device detection
driver loading
resource allocation.

→ Completly fair Scheduler

* It models an ideal, precise multitasking CPU.

Device Drivers

* They are abstraction to a piece of hardware.



Types of Drivers

1) character ✓
2) Block ✓
3) Network ✓
4) Pseudo ✓
5) USB ✓
6) Platform ✓
7) File Systems. ✓

1) character Device Driver

Used for: Device that handle data as a stream of bytes.
eg: Serial ports, keyboards

Interact with kernel:
- register_chrdev() func to register the driver
- file operations
  read(), write(), Open(), close()

eg: /dev/ttyS0
↳ char devices.

2) Block Device Driver

used for: Devices that store and transfer data in fixed size block
eg: SSD, USB device

Interaction with kernel:
- register_blkdev() func to register the driver
- used block request queue to mange read and write op.
- Works with VFS and I/O scheduler

eg: /dev/mmcblk0 (SD card).

3) Network Devices.

used for: Network interface like ethernet, wifi, bluetooth adp.

Interaction
- register with register_netdev()
- struct net_device interface
- socket buffer for packet transmission

eg: eth0.

4) Pseudo

used for: virtual devices that dont correspond to real hardware

Interaction.
- Implements file operation read(), write()
- often used for testing and debugging

5) USB

used for: USB peripherals like flash drives, printers, webcam.

Interaction
- uses usb_register_driver()
- Communication using ~~URB~~ USB Request block.

6) Platform.

used for: On-board SoC peripherals GPIO controller, SPI/I2C dev

Interaction
- Device tree (.dts)
- registered with platform bus via platform_driver_register()

7) File System.

used for: Managing file systems like ext4, NTFS.

Interaction.
- Registers with the VFS via register_filesystem()
- implement inode op (read_inode() write_inode())

eg: proc, sys, mnt

Imp

How devices interact with the kernel.

1) Drivers must register with the kernel using API.

register_chrdev()
register_blkdev()
usb_register_driver()

2) Handling System Calls

3) Managing hardware via kernel subsystems

I/O scheduler → block devices
network stack → network devices
USB core → USB devices.
Device tree → platform devices.

4) Interrupt handling

Devices generate interrupts and the driver must register an interrupt handler to handle them asynchronously

5) Power Management

Embedded drivers need to support suspend/resume oper for power efficeny

Building device driver / Modules.

→ hello world.

printk() → used to print in kernel seen in logs.

Syslog
↳ System log has the print for hello world.

'tail' /var/log/syslog.

ls -l.

Shows which ~~device~~ file is a chara device or a block device

This command also gives you a major or minor number

Major, Minor.
↙ ↘
Drivers — The devices.
1, 4, 7, 10 — refer to the device.

/dev/null } driver 1
/dev/zero }
serial 4
VCS1 & VCSA 7

Every Module has some operation written in it.

Struct file_operation Scull_fops {

- Owner
- llseek
- read

:

}

each field, points to the function in the driver that implements a specific operation

Implements function for typical System call

write()
↓
doesn't immediatly write to it.

It is stored in the buffer and it is written once fil is being closed or buffer is being flushed.

Q Explain the role of ioctl() in linux device driver?

ans: ~~ioctl() is a system call takes to provide general interface for controlling and configuring a device~~

It is used to implement device specific functions that are not part of standard system calls.

Takes 3 arguments.

Q How do you implement a character device driver in linux?

ans: a) define the device structure (operation, name)
b) implement the device fill operations
c) Register the device
d) Create the device file
e) Implement device specific func
f) Test the driver

Q. Describe the process of allocating memory in linux device driver?

ans: kmalloc() → allocate memory from kernel heap
Vmalloc() → allocate in virtual memory space

Q. Describe the process of cross compiling a device driver for embedded linux?

ans: Install cross compiler.
↳ used to build code for target architecture.
↳ Configure the build env.
↳ compatible with cross compiler and target architecture.
↳ Build driver
↳ build and update. and compile.
↳ Test driver
↳ on target system.

Q. How do you handle real-time constraints in an embedded linux device driver.

ans: (a) interrupts
(b) DMA.
(c) Task scheduling → FIFO, RR, RM, EDF
d) locking → for process synchronisation.

Q. SPI vs I2C

SPI	I2C
✱ 4 wire SCLK, SCL, MISO, MOSI	✱ 2 wire SDA, SCL
✱ full duplex	✱ half duplex
✱ fast	✱ slow.
✱ displays, memory, wireless transceiver	✱ sensors, EEPROM real-time clock.

Q. Describe the use of GPIO in embedded linux device driver.

ans: GPIO device driver uses linux kernels GPIO framework, which provides a consistant interface for device driver to interact with GPIO pins.

- Allocate/deallocate GPIO pins
- Setting the pin direction, value
- registering interrupt handler.

Q Optimise performance of system call?

ans: (a) Minimise system calls.
(b) use interrupt handler
(c) use DMA.
(d) use better data structure.
(e) Optimise memory.

Snap vs AppImage vs Flatpacks.

Package

→ dpkg

low level package manager.

Cons:

1) doesn't install dependencies.

- download the .deb file
- sudo -i discord.deb.
  ↳ fails because of dependencies.

dpkg -l : list all package.

highlevel

→ apt (Advanced Package tool)
sudo apt install <name-pkg>

if anything is broken (dependencies)

sudo apt --fix broken install.

APT
has a repository
↳ collection of software.

list
show <pkg>
remove → remove app but not unneeded
list -- installed.
purge → remove everything.
update → up-to-date list/Repo
upgrade → upgrade the list.
full-upgrade → remove previously installed & ~~no~~ upgrade.

lightweight

→ Aptitude (GUI)

More like apt on steroids.

→ Snap

Snap → uses store.

makes it easier to get apps on-to a ~~repository~~. store.

getting it onto a repo is difficult.

→ language based / git

pip

Git

git clone <Repo url.>

pip

sudo apt install python3-pip

pip3 -r req

SNAPS

Build once
distrobute everywhere.
↳ do not have to make pkg for every distro or arch.

— Auto update.
— containerized!!!

→ Snap sandbox

Allows sharing of data & unix sockets between snaps.

Why?

Share common libraries
reduce size by avoiding duplication.

uses Apparmor by default.

(LSM)
linux Security Module.

Other distro like Fedora

SE linux
↳ degrade SNAPS. Sandbox.

Security

→ Linux Security Module

They provide MAC, mandatory access control.

MAC : limits access based ~~on~~ what ~~system~~ user wants or defines

DAC : Security policies that gives user access to decide who to grant permissions to.
chmod
filepermission eg: google docs.

1) SE Linux — MAC, RBAC.
Security enhanched linux
↳ developed by NSA.
↳ default in Redhat, Fedora.

Deny everything

uses labels to define policies for every file, process and user

3 modes
↳ enforcing → block action
↳ Permissive → log ~~but~~ but allow acti
↳ disabled.

* complex
* can cause unexpected ~~dinia~~ denial.

→ embedded → lightweight

2) AppArmor MAC, RBAC,

↳ Developed by Canonical (ubuntu)
↳ debian and ubuntu.

uses profiles to define rules for application.

Profiles stored in /etc/apparmor.

- ✱ easier
- ✱ less fine grain control

allow everything

Q. What are LSM; how they work?

ans: LSM are security module that enforce MAC. ~~at this~~

LSM hook into linux kernel and enforce security policies.

Q. Securing an embedded linux system.

ans: remove unwanted access
LSM → to implement MAC
no Root SSH access
asymetric encryption
disk encryption
secure boot.

Q. What is secure boot?

ans: Secure boot is a security feature that ensure cryptogra... signed bootloader and kernels are executed.

Verify bootloader
↳ digital signature

{ Verify kernel
  init ramfs } bootloader verifies.

prevents Rootkit modifications
bootloader tampering

Q. Security risks of loading kernel module?

ans: unsigned module → malicious code
kernel crash → kernel code not written well.

Backdoor into kernel →
kernel modules can even undetected

* (Restrict who and what module can be loaded).

Q linux namespace and cgroups.

namespace

- isolate system resource.
- Prevents 1 container from seeing the other.
- containers → used in

eg: Process in 1 namespace cant see the other.

Cgroups

- control resource usage.
- Prevents resource exhaustion attacks.
- used in container, systemd services

eg: limit memory usage of 1 container.

Q. What is stack canary Protection? How does linux prevent buffer overflow

ans: stack canarys are random values placed on stack to detect buffer overflows before they can overwrite return address.

* random value placed b/w local and return add.
* before returning from a function the canary value is checked
* if value is changed due to overflow then abort execution.