Embedded Linux Interview Questions

1. What is an Embedded System?

An **embedded system** is a specialized computing system designed to perform a specific task or set of tasks, usually with **real-time constraints**. It is often part of a larger system.

Examples:

- Washing machines, microwave ovens, digital cameras, automotive control units, IoT devices.

2. What is Embedded Linux?

Embedded Linux is the use of the Linux operating system in embedded systems. It's tailored for devices with limited resources (CPU, memory, storage).

Example:

- A smart thermostat running Linux to control temperature and connect to Wi-Fi.

3. Difference Between Embedded Linux and Desktop Linux

Feature	Desktop Linux	Embedded Linux
Hardware	High resources (RAM, CPU)	Low resources (small RAM/CPU)
GUI	Usually has GUI	Often no GUI or minimal GUI
Purpose	General-purpose computing	Specific task/device control
Storage	Large, persistent storage	Limited storage (flash memory)
Boot Time	Seconds to minutes	Fast boot (ms to a few seconds)

4. Main Components of Embedded Linux

1. **Linux Kernel** – Core of OS
2. **Bootloader** – Starts the system
3. **Root File System (RootFS)** – User space applications and libraries
4. **Shell** – Interface for command
5. **Libraries & Utilities** – Like BusyBox for lightweight commands

---

5. What is the Linux Kernel?

The **Linux kernel** is the core of the OS that interacts with hardware and manages processes, memory, and devices.

Example: In a router, the kernel controls network interfaces and packet routing.

---

6. What is a Shell?

A **shell** is a command-line interface for interacting with the OS.

Example:

- `bash` on desktop Linux
- `ash` in embedded Linux (via BusyBox)

---

7. What is BusyBox?

BusyBox is a single binary that provides many common Unix utilities in a lightweight form, ideal for embedded systems.

Example:
Instead of separate binaries for `ls`, `cp`, `mv`, BusyBox handles them all to save storage.

---

8. What is an init Process?

The **init process** is the first process started by the kernel (PID 1) and is responsible for initializing the system, starting services, and launching user applications.

Example:

- `systemd` or `BusyBox init` in embedded Linux.

9. What is Root File System (RootFS)?

RootFS contains the directories, binaries, libraries, and configuration files needed for the system to operate. Essentially, it's the "user space" of Linux.

Example:

- `/bin`, `/etc`, `/lib` directories in embedded Linux root filesystem.

10. What is the Role of initramfs?

initramfs is an initial RAM filesystem loaded by the bootloader into memory to help the kernel initialize before the real RootFS is available.

Purpose:

- Load kernel modules
- Mount RootFS

11. Difference Between initramfs and RootFS

Answer

Feature	initramfs	RootFS
Storage	In RAM	In persistent storage (flash/SD card)
Lifespan	Temporary (during boot)	Persistent (whole runtime)
Purpose	Early system initialization	Main filesystem for applications

12. What is a Bootloader?

A **bootloader** is the first software that runs when a device powers on. It initializes hardware and loads the kernel.

Examples:

- U-Boot, Das U-Boot (common in embedded devices)

13. Linux Booting Process

1. **Bootloader** runs → initializes hardware, loads kernel + initramfs
2. **Kernel** starts → sets up memory, devices
3. **initramfs** runs → prepares environment, mounts real RootFS
4. **init process** runs → starts system services, user applications

---

14. What is Cross Compilation?

Cross compilation is compiling software on one architecture (host) to run on another (target).

Example:

- Compile ARM binary on x86 Linux PC for Raspberry Pi.

---

15. What is BSP (Board Support Package)?

A **BSP** contains all drivers, configuration, and libraries needed to run Linux on a specific hardware board.

Example:

- BSP for BeagleBone Black includes kernel config, device tree, bootloader.

---

16. What is Linux Porting?

Linux porting is adapting the Linux kernel and software to run on new hardware.

Example:

- Porting Linux to a new microcontroller requires:
  1. Writing drivers for hardware peripherals
  2. Creating a device tree
  3. Configuring the kernel and root filesystem

---

17. What is the difference between a process and a thread?

Answer:

Feature	Process	Thread
Memory	Has its own memory space	Shares memory with other threads of the same process
Creation	Heavy and slower	Lightweight and faster
Communication	Requires IPC (pipes, sockets, shared memory)	Can communicate directly via shared memory
Example	A web browser as a process	Each tab or download task as a thread in the browser

Key idea: A thread is like a "lightweight process" inside a process.

---

18. What is `fork()`?

Answer:
`fork()` is a system call used to **create a new child process** by duplicating the calling process.

- The parent and child processes run independently.

Example in C:

```c
C/C++
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();
    if(pid == 0)
        printf("Child process\n");
    else
        printf("Parent process\n");
    return 0;
}
```

19. What is `exec()`?

Answer:
`exec()` is a system call that **replaces the current process image with a new program**.

- The process keeps its PID but starts executing a new program.
- Often used after `fork()` in the child process.

Example in C:

```c
#include <unistd.h>

int main() {
    char *args[] = {"/bin/ls", NULL};
    execv("/bin/ls", args); // replace current process with ls
    return 0;
}
```

20. What is a zombie process?

Answer:
A zombie process is a **terminated child process whose exit status has not been read by its parent**.

- It still occupies a slot in the process table until the parent calls `wait()`.

Example command:

```shell
ps aux | grep Z
```

- 'Z' indicates a zombie process.

21. What is an orphan process?

Answer:
An orphan process is a **child process whose parent has terminated**.

- The `init` process (PID 1) adopts it and eventually cleans it up.

22. What is nice value?

Answer:
The nice value controls a process's **CPU scheduling priority**.

- Range: **-20 (highest priority) to +19 (lowest priority)**
- Default: 0

Example:

```shell
nice -n 10 ./my_program  # runs with lower CPU priority
```

---

23. What are system calls?

Answer:
System calls are **interfaces for user programs to request services from the kernel**.

Examples:

- `open()`, `read()`, `write()`, `fork()`, `exec()`, `ioctl()`

Analogy: User programs "ask" the kernel to do things they cannot do directly.

---

24. How does user space communicate with kernel space?

Answer:
Communication methods include:

1. **System calls** – e.g., `read()`, `write()`, `ioctl()`
2. **Device files** – e.g., `/dev/ttyS0`
3. **Procfs and Sysfs** – e.g., `/proc/cpuinfo`, `/sys/class/leds`
4. **Signals** – for asynchronous notifications
5. **Memory-mapped I/O** – using `mmap()`

Analogy: User space is like a tenant requesting services; kernel space is the landlord safely providing them.

---

25. What is a device driver?

Answer:
A **device driver** is a software component that allows the **operating system to communicate with hardware devices**.

- It acts as a bridge between the kernel and hardware.

Example:

- Keyboard driver → kernel can read key presses
- Network card driver → kernel can send/receive network packets

---

26. What are the types of device drivers?

Answer:
Device drivers can be classified based on how they interact with devices:

1. **Character drivers** – handle data as a stream of bytes
2. **Block drivers** – handle data in blocks, usually for storage devices
3. **Network drivers** – manage network interfaces and packet transfer
4. **Other types** – e.g., USB, platform-specific drivers

---

27. What is a character driver?

Answer:
A **character driver** handles devices that can be accessed **one byte at a time**.

- It supports sequential read/write operations.

Examples:

- Serial port (`/dev/ttyS0`)
- Keyboard
- Mouse

---

28. What is a block driver?

Answer:
A **block driver** handles devices that read/write **data in fixed-size blocks**.

- Supports random access, usually for storage devices.

Examples:

- Hard drives (`/dev/sda`)
- SD cards
- USB storage

---

29. What is a network driver?

Answer:
A **network driver** manages communication between the kernel and a network interface card (NIC).

- Responsible for sending and receiving network packets.

Examples:

- Ethernet driver (`eth0`)
- Wi-Fi driver (`wlan0`)

---

30. What are major and minor numbers?

Answer:

- **Major number:** Identifies the **driver associated with the device**.
- **Minor number:** Identifies the **specific device controlled by that driver**.

Example:

```
Shell
ls -l /dev/sda
```

- `8, 0` → 8 is the major number (block driver for disks), 0 is the minor number (first disk).

---

31. What is `ioctl()`?

Answer:
`ioctl()` (Input/Output Control) is a **system call used to perform device-specific operations** that are not covered by standard read/write.

Example in C:

```
C/C++
#include <sys/ioctl.h>

int fd; // file descriptor of device

ioctl(fd, SOME_COMMAND, &argument); // perform custom device operation
```

- Can be used to change baud rate of a serial port, or configure a network device.

---

32. What is the difference between user space and kernel space?

Answer:

Feature	User Space	Kernel Space
Access	Limited access to hardware	Full access to hardware and memory
Privilege	Runs in **unprivileged mode**	Runs in **privileged mode**
Safety	Crashes affect only the application	Crashes can crash the whole system
Example	Running a program like `vim`	Linux kernel managing processes, memory, and devices

Key idea: User space is for normal applications, kernel space is for OS core operations.

---

33. Why is user–kernel separation important?

Answer:

- **Security:** Prevents user applications from directly modifying hardware or OS structures.
- **Stability:** User crashes don't crash the entire system.
- **Controlled resource access:** Kernel manages resources safely and fairly.

---

34. What is static linking vs dynamic linking?

Answer:

Feature	**Static Linking**	**Dynamic Linking**
Libraries	Linked at compile time	Linked at run time
File size	Larger executables	Smaller executables
Updates	Must recompile to update	Libraries can be updated independently
Example	`gcc main.c -o prog`	`gcc main.c -o prog -ldl`

---

35. What is virtual memory?

Answer:
Virtual memory allows the OS to give **processes the illusion of a large, contiguous memory** even if physical memory is fragmented.

- Uses **page tables** to map virtual addresses to physical addresses.

Example:

- A process can access 4 GB memory on a system with only 1 GB RAM using virtual memory + swap.

---

36. What is swap memory?

Answer:
Swap memory is **disk space used as an extension of RAM** when physical memory is full.

- Helps prevent out-of-memory conditions but is slower than RAM.

Example:

- `/swapfile` or swap partition in Linux.

---

37. What is a page fault?

Answer:
A page fault occurs when a **process tries to access a page not present in RAM**.

- The kernel then loads the page from disk (swap or file-backed memory) into RAM.

Normal part of virtual memory operation.

---

38. What is OOM killer?

Answer:
OOM (Out of Memory) killer is a kernel mechanism that **terminates processes** to free memory when the system runs critically low on RAM.

- Targets processes with high memory usage or low priority.

---

39. What is context switching?

Answer:
Context switching is the process of **saving the state of the current process/thread and loading the state of another**.

- Necessary for multitasking.

Example:

- CPU switches from one process to another every few milliseconds in Linux.

---

40. What is a memory leak?

Answer:
A memory leak occurs when a program **allocates memory but never frees it**, leading to wasted resources and potential system slowdown.

Example:

- Forgetting `free()` after `malloc()` in C.

41. What is kernel space memory allocation?

Answer:
Kernel space memory allocation is **allocating memory that can be used safely by the kernel**.

- Two main types: **contiguous and non-contiguous memory**.
- Functions like `kmalloc()` and `vmalloc()` are used.

42. Difference between kmalloc() and vmalloc()

Answer

Feature	**kmalloc()**	**vmalloc()**
Memory type	Physically contiguous	Virtually contiguous (can be non-contiguous in physical memory)
Speed	Faster	Slower
Usage	Small buffers, DMA	Large memory allocations
Example	`kmalloc(1024, GFP_KERNEL)`	`vmalloc(1024*1024)`

Key idea: Use `kmalloc()` for speed and DMA, `vmalloc()` for large memory needs.

43. What is the difference between interrupt and polling?

Answer:

Feature	Interrupt	Polling
Trigger	Hardware notifies CPU	CPU repeatedly checks device status
CPU usage	Efficient (CPU sleeps until event)	Inefficient (wastes CPU cycles)
Latency	Fast response	Slower response, depends on polling frequency
Example	Keyboard input triggers interrupt	Program constantly checks if a key is pressed

Key idea: Interrupts are event-driven; polling is CPU-driven.

44. What is the difference between mutex and semaphore?

Answer:

Feature	Mutex	Semaphore
Purpose	Ensures **mutual exclusion** for a single resource	Controls access to **multiple resources**
Value	Binary (locked/unlocked)	Can be >1 (counts available resources)
Ownership	Only the thread that locks can unlock	Any thread can signal (V operation)

Key idea: Mutex = binary lock, Semaphore = counting lock.

---

45. What is preemption in Linux?

Answer:
Preemption is the **ability of the kernel to suspend a running task** and switch to a higher-priority task.

- Enables **real-time responsiveness**.

Example:

- A high-priority interrupt-driven task preempts a CPU-bound low-priority task.

---

46. What is softirq?

Answer:
Softirq is a **lightweight, deferred interrupt mechanism** in Linux.

- Handles time-sensitive tasks **outside of hard interrupt context**.
- Can run concurrently on multiple CPUs.

Example:

- Network packet processing is often handled via softirq (`NET_RX_SOFTIRQ`).

---

47. What is a tasklet?

Answer:
Tasklets are **built on top of softirqs** and allow **bottom-half processing** in a **serialized, non-preemptible context**.

- Cannot run in parallel on the same CPU.

Example:

- Deferred work after a hardware interrupt, like updating a driver buffer.

---

48. What is a workqueue?

Answer:
Workqueues allow **deferred work to run in process context** instead of interrupt context.

- Can **sleep**, unlike softirqs or tasklets.
- Useful for tasks that may block or require memory allocation.

Example:

- Writing received network data to disk in kernel space.

---

49. What is a threaded IRQ?

Answer:
A threaded IRQ runs the **interrupt handler in a kernel thread** instead of top-half context.

- Can **sleep** and perform long operations safely.
- Top-half (hardware interrupt) schedules the threaded handler.

Example:

- Drivers for network cards or storage devices with long processing.

---

50. What is an interrupt handler?

Answer:
An interrupt handler is a **function executed by the kernel in response to a hardware interrupt**.

- Top-half: Quick, minimal work.
- Bottom-half: Deferred work via softirq, tasklet, or workqueue.

Example:

- Handling a key press from a keyboard or receiving a packet from a network card.

---

51. What is VFS (Virtual File System)?

Answer:
VFS is an **abstraction layer in the Linux kernel** that provides a **common interface for all file systems**.

- Allows applications to use standard file operations (`open`, `read`, `write`) **regardless of the underlying file system**.
- VFS maintains file descriptors, inodes, and superblocks.

Example:

- You can `cat /etc/passwd` whether it's on ext4, FAT, or NFS because VFS abstracts the details.

---

52. What is sysfs?

Answer:
sysfs is a **virtual file system** that exposes kernel objects, drivers, and device information to user space.

- Mounted at `/sys`
- Helps in **querying and configuring hardware and drivers**.

Example:

```shell
cat /sys/class/leds/led0/brightness
```

- Reads or controls the brightness of an LED.

---

53. What is procfs?

Answer:
procfs is a **virtual file system** that provides **information about processes and kernel statistics**.

- Mounted at `/proc`
- Useful for monitoring system resources and debugging.

Example:

```shell
cat /proc/cpuinfo      # CPU details

cat /proc/meminfo      # Memory usage

cat /proc/[pid]/status # Process information
```

---

54. What is udev?

Answer:
udev is the **device manager for Linux**, responsible for:

- Creating device nodes in `/dev` dynamically
- Handling hotplug events (USB, PCI, etc.)
- Assigning permissions and running scripts on device events

Example:

- Plugging in a USB drive automatically creates `/dev/sdb1` and mounts it.

---

55. How do you customize the Linux kernel?

Answer:
Customizing the Linux kernel involves **configuring, compiling, and sometimes patching the kernel** to suit specific hardware or application requirements.

Steps:

1. **Obtain kernel source:**

```
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.x.tar.xz
```

2. **Configure kernel options:**

```
make menuconfig   # interactive configuration menu
```

3. **Compile the kernel:**

```
make -j$(nproc)
```

4. **Install kernel and modules:**

```
make modules_install

make install
```

5. **Boot with the new kernel**

Customizations include:

- Enabling/disabling drivers
- Removing unneeded features to reduce size
- Adding custom patches

---

56. What is Yocto?

Answer:
Yocto is a **build system and framework** for creating **custom Linux distributions for embedded devices**.

- Generates a complete Linux image including kernel, bootloader, and root filesystem.
- Uses recipes and layers to manage packages.

Example:

- Building a Yocto image for Raspberry Pi:

```
source oe-init-build-env

bitbake core-image-minimal
```

---

57. What is Buildroot?

Answer:
Buildroot is a **simpler tool than Yocto** for generating **embedded Linux root filesystems and cross-compiling packages**.

- Focuses on **speed and simplicity**
- Generates toolchain, kernel, bootloader, and rootfs

Example:

- Configure Buildroot for ARM board:

```
make menuconfig

Make
```

---

58. How do you reduce kernel size?

Answer:
Techniques to reduce kernel size:

1. **Disable unnecessary drivers and modules** (`make menuconfig`)
2. **Use modular kernel** instead of built-in features
3. **Remove debugging symbols** (`CONFIG_DEBUG_INFO=n`)
4. **Strip the kernel binary**
5. **Enable compression** (`CONFIG_KERNEL_XZ` or `CONFIG_KERNEL_GZIP`)

Result: Smaller memory footprint, faster boot.

---

59. How do you optimize Linux boot time?

Answer:
Techniques for faster boot:

1. **Use initramfs wisely** – only include necessary files
2. **Enable only required services** (disable unneeded daemons)
3. **Parallelize service startup** (`systemd` supports this)
4. **Optimize kernel configuration** – remove unused drivers/features
5. **Use faster storage or preloaded boot images**
6. **Profile boot with tools like `bootchart` or `systemd-analyze`**

Example: On embedded devices, disabling GUI and unnecessary modules can reduce boot time from 10s to 3s.

---

60. What is a device tree?

Answer:
A **device tree** is a data structure that describes the **hardware layout of a system** to the Linux kernel.

- Used in **embedded Linux** for boards where the hardware is not discoverable automatically.
- Provides information like CPU, memory, buses, peripherals, and IRQs.

Example:

- On a Raspberry Pi, the device tree tells the kernel which GPIOs and peripherals are available.

---

61. What are DTS and DTB?

Answer:

- **DTS (Device Tree Source):** Human-readable text file describing hardware in a tree format.
- **DTB (Device Tree Blob):** Compiled binary version of DTS that the kernel can read at boot.

Example:

Shell
```
arch/arm/boot/dts/bcm2710-rpi-3-b.dts  # DTS source

dtc -I dts -O dtb -o bcm2710-rpi-3-b.dtb bcm2710-rpi-3-b.dts  # compile
to DTB
```

---

62. What is device tree binding?

Answer:
Device tree binding defines the **rules and properties for a specific type of device** in the device tree.

- Specifies how the kernel driver should interpret the device tree node.
- Ensures **standardized mapping** between hardware description and driver.

Example:

- UART binding may specify: `compatible`, `reg`, `clocks`, `interrupts` properties that the UART driver reads.

---

63. What is a compatible string in device tree?

Answer:
A **compatible string** is a property in a device tree node that **tells the kernel which driver should handle the device**.

- Kernel searches for a matching driver using this string.

Example in DTS:

None
```
uart0: serial@101f1000 {

    compatible = "arm,pl011", "arm,primecell";

    reg = <0x101f1000 0x1000>;

    interrupts = <0 29 4>;

};
```

- `"arm,pl011"` is the primary compatible string for the UART driver.

64. What is a platform driver?

Answer:
A **platform driver** is a type of Linux device driver for **on-chip or system-on-chip (SoC) devices** that are **not discoverable via hardware buses like PCI or USB**.

- Works with **platform devices** described in **device tree**.
- Typically handles things like GPIOs, timers, UARTs, and I2C controllers.

Example:

- UART driver for an ARM SoC: `drivers/serial/serial_pl011.c`

65. What is the difference between platform driver and PCI driver?

Feature	Platform Driver	PCI Driver
Device discovery	Described in device tree or board code	Discovered automatically via PCI enumeration
Use case	On-chip devices (SoC peripherals)	PCI bus devices like graphics cards or

Key idea: Platform driver = static SoC devices; PCI driver = bus-enumerated devices.

66. What is the purpose of `probe()` function?

Answer:
The **`probe()` function** is called by the kernel **when a matching device is found** for a driver.

- Initializes the device, allocates resources, and registers it with kernel subsystems.

Example:

```
C/C++
static int uart_probe(struct platform_device *pdev) {

    // allocate memory, request IRQ, register device

    return 0;

}
```

67. What is the purpose of `remove()` function?

Answer:
The **`remove()` function** is called when the device is **removed or driver is unloaded**.

- Cleans up resources allocated in `probe()`
- Unregisters device from kernel subsystems

Example:

```
C/C++
static int uart_remove(struct platform_device *pdev) {

    // free memory, release IRQ

    return 0;

}
```

68. How does a driver register with the kernel?

Answer:
Drivers register with the kernel using **specific registration functions** depending on the driver type:

Driver Type	**Registration Function**
Platform	`platform_driver_register()`
PCI	`pci_register_driver()`
USB	`usb_register_driver()`

The kernel calls the driver's `probe()` for devices that match.

---

69. What is DMA (Direct Memory Access)?

Answer:
DMA is a mechanism that **allows hardware devices to transfer data directly to/from memory** without CPU intervention.

- Reduces CPU load and increases performance.

Example:

- A network card transferring received packets to RAM using DMA.

---

70. What is I2C, SPI, UART in Linux context?

Answer:

- **I2C:** Two-wire serial bus for connecting multiple low-speed peripherals.
- **SPI:** High-speed synchronous serial bus for sensors, flash memory, or displays.
- **UART:** Asynchronous serial communication for console, modems, or debug logs.

Example:

- `/dev/i2c-1` → I2C bus
- `/dev/spidev0.0` → SPI device
- `/dev/ttyS0` → UART console

---

71. What is `copy_to_user()` and `copy_from_user()`?

Answer:
These are **kernel APIs to safely transfer data between kernel space and user space**.

- `copy_to_user(dest, src, size)` → kernel → user
- `copy_from_user(dest, src, size)` → user → kernel

Example:

```c
C/C++
char buffer[100];

copy_to_user(user_ptr, buffer, sizeof(buffer));
```

Direct memory access from user space is **forbidden**, so these functions prevent crashes and security issues.

---

72. What is a loadable kernel module (LKM)?

Answer:
A **Loadable Kernel Module** is a **piece of code that can be dynamically loaded/unloaded into the kernel at runtime**.

- Extends kernel functionality **without recompiling the whole kernel**.

Example:

Shell
```
insmod my_driver.ko   # load module

rmmod my_driver.ko    # unload module

lsmod                  # list loaded modules
```

Common uses: device drivers, filesystem modules, network protocol modules.

---

73. Explain Linux kernel architecture

Linux uses a **monolithic kernel architecture with modular design**.

- Core kernel runs in **kernel space**
- User applications run in **user space**
- Kernel provides services like:
  - Process management
  - Memory management
  - File systems
  - Device drivers
  - Networking

Key feature:
Although monolithic, Linux supports **loadable kernel modules (LKMs)**, so drivers can be loaded/unloaded at runtime.

Example:
USB driver can be loaded only when a USB device is plugged in, without rebooting the system.

---

74. How does the Linux scheduler work?

The Linux scheduler decides **which process runs on the CPU and for how long**.

Modern Linux uses **CFS (Completely Fair Scheduler)**.

How CFS works:

- Each process gets a **virtual runtime**
- Scheduler tries to give equal CPU time to all runnable processes
- Process with **lowest virtual runtime** runs next

Other scheduling classes:

- `SCHED_NORMAL` – regular tasks
- `SCHED_FIFO`, `SCHED_RR` – real-time tasks
- `SCHED_DEADLINE` – deadline-based scheduling

Example:
If Process A used less CPU than Process B, A will be scheduled first to keep fairness.

---

75. Explain slab allocator

The **slab allocator** is a **kernel memory management mechanism** used for **efficient allocation of small objects**.

Why slab allocator?

- Reduces fragmentation
- Improves performance
- Reuses pre-initialized objects

How it works:

- Memory divided into **slabs**
- Each slab contains objects of the **same type**
- Objects are reused instead of freed

Example:
Kernel objects like `task_struct` are allocated from slab caches instead of general memory.

---

76. What is RCU (Read Copy Update)?

RCU is a synchronization mechanism used in the Linux kernel.

Purpose:

- Allow **lock-free reads**
- Updates are done by copying data and replacing it safely

How it works:

1. Readers access data without locks
2. Writer creates a copy and updates it
3. Old data is freed only after all readers finish

Example:
Used in networking routing tables where reads are frequent and updates are rare.

---

77. What is SMP (Symmetric Multiprocessing)?

SMP is a system where **multiple CPUs share the same memory and I/O**.

Characteristics:

- All CPUs are equal
- Any CPU can run any task
- Improves performance and scalability

Linux SMP support includes:

- Per-CPU data
- Spinlocks
- CPU affinity

Example:
Quad-core processor running multiple applications simultaneously.

---

78. What is kernel taint?

Kernel taint indicates that the **kernel is in an unsupported or unreliable state**.

Reasons for kernel taint:

- Loading proprietary (closed-source) modules
- Hardware errors
- Forcing module load
- Kernel crashes

Why it matters:

- Helps developers identify issues
- Tainted kernel bugs may not be accepted upstream

Example:
Loading a proprietary NVIDIA driver taints the kernel.

---

79. What is PREEMPT_RT?

PREEMPT_RT is a Linux patch that converts Linux into a **real-time operating system**.

Key features:

- Nearly full kernel preemption
- Interrupt handlers run as threads
- Reduced scheduling latency

Used for:

- Industrial automation
- Robotics
- Audio processing

Example:
Robot controller using PREEMPT_RT to respond within microseconds.

---

80. What are real-time constraints in embedded systems?

Real-time constraints define **timing deadlines** that must be met.

Types:

- **Hard real-time** – missing deadline causes system failure
- **Soft real-time** – performance degrades but system survives

Key constraints:

- Deterministic response
- Low latency
- Predictable scheduling

Example:

- Airbag system → hard real-time
- Video streaming → soft real-time

---

81. How is power management handled in Embedded Linux?

Power management in Embedded Linux is handled through **kernel frameworks, device drivers, and user-space tools** to reduce power consumption while maintaining functionality.

Key components:

- **CPU power management** (frequency & idle states)
- **Device power management** (suspend/resume)
- **System sleep states**
- **Runtime power management**

The Linux kernel coordinates power states based on **system activity**, while drivers inform the kernel when devices can sleep.

Example:
When an embedded device's screen turns off, Linux lowers CPU frequency and puts unused peripherals (Wi-Fi, USB) into low-power mode.

---

82. What power management techniques are used in Embedded Linux?

Embedded Linux uses several power-saving techniques:

1. **Dynamic Voltage and Frequency Scaling (DVFS)**
   - Adjusts CPU voltage and frequency based on load
   - Implemented using `cpufreq` governors
   - Example: CPU runs at 400 MHz during idle, 1 GHz under load
2. **CPU Idle States (cpuidle)**
   - Puts CPU into low-power states when idle
   - Deeper idle → more power saving, higher wake-up latency
   - Example: ARM Cortex-A cores entering WFI (Wait For Interrupt)
3. **Runtime Power Management**
   - Devices are powered down when not in use
   - Example: Camera sensor powered off when no app uses it
4. **System Sleep States**
   - Suspend-to-RAM, Hibernate
   - Example: Smartwatch entering suspend mode when not worn
5. **Tickless Kernel (NO_HZ)**
   - Reduces periodic timer interrupts
   - Helps CPU stay in sleep mode longer

---

83. How do you analyze memory fragmentation?

Memory fragmentation occurs when free memory exists but is **not contiguous**, causing allocation failures.

Types of fragmentation:

- **External fragmentation** – free memory scattered
- **Internal fragmentation** – allocated memory not fully used

Analysis techniques in Embedded Linux:

1. **/proc/buddyinfo**
   - Shows fragmentation in physical memory
   - Helps identify lack of large contiguous blocks
2. **/proc/meminfo**
   - Displays overall memory usage statistics
3. **SLAB/SLUB debugging tools**
   - `slabtop` command
   - Shows kernel object cache usage
4. **Kernel tracing & debugging**

- `ftrace`, `kmemleak`
- Identify memory leaks and allocation patterns

Example:
If a camera driver fails to allocate a large DMA buffer, `/proc/buddyinfo` may show no high-order free pages even though total free memory exists.

---

84. How do you debug race conditions in Linux?

Race conditions occur when **multiple threads or CPUs access shared data incorrectly**.

Steps to debug:

- Enable **lock debugging** (`CONFIG_DEBUG_MUTEXES`, `CONFIG_PROVE_LOCKING`)
- Use **lockdep** to detect deadlocks
- Add **tracepoints / printk** with CPU & PID info
- Use **KCSAN (Kernel Concurrency Sanitizer)**

Example:
Two drivers update the same buffer → crash on SMP system → lockdep reports missing spinlock.

---

85. How do you debug bootloader issues?

Bootloader issues occur **before the kernel loads**.

Steps:

1. Check **UART/serial logs**
2. Verify **bootloader environment variables**
3. Confirm **kernel image & DTB addresses**
4. Enable **bootloader debug prints**
5. Verify memory map

Example:
U-Boot loads kernel but hangs → wrong load address overlaps with initrd.

---

86. How do you debug device tree problems?

Device tree issues cause **hardware not detected or misconfigured**.

Steps:

- Enable `CONFIG_OF`
- Check **dmesg** for DT errors

- Verify `compatible`, `reg`, `interrupts`
- Decompile DTB using `dtc`
- Compare with reference DTS

Example:
Ethernet not working → wrong PHY address in device tree.

---

87. System is not booting — what will you do?

Systematic approach:

1. Check **power & clocks**
2. Observe **serial console output**
3. Identify **boot stage failure** (ROM / bootloader / kernel)
4. Verify kernel, initrd, DTB
5. Enable early kernel logs (`earlyprintk`)

Example:
Boot stops after "Starting kernel" → invalid DTB or kernel image.

---

88. System boot is slow — how will you fix it?

Steps:

- Measure boot time using `systemd-analyze`
- Disable unused services
- Optimize device initialization
- Use parallel init
- Reduce kernel modules

Example:
Wi-Fi service delays boot by 10s → disable auto-start.

---

89. Kernel panic occurs — how will you debug it?

Steps:

- Capture **panic log**
- Decode stack trace using `addr2line`
- Identify faulting function
- Enable `CONFIG_KALLSYMS`
- Use crash dump (`kdump`)

Example:
NULL pointer dereference in scheduler → corrupted task structure.

---

90. Kernel panic during driver load — how will you debug it?

Steps:

- Enable driver debug prints
- Check `probe()` function
- Validate memory allocations
- Verify device tree bindings
- Load module with `insmod -v`

Example:
Driver panics due to accessing uninitialized pointer in `probe()`.

---

91. Random crashes after hours — what steps will you take?

Steps:

- Suspect **memory leaks or race conditions**
- Enable `kmemleak`
- Monitor memory over time
- Stress test system
- Enable watchdog

Example:
Crash after 6 hours → slab cache exhaustion due to leaked buffers.

---

92. High CPU usage — how will you analyze it?

Steps:

- Use `top` / `htop`
- Identify process or kernel thread
- Use `perf` or `ftrace`
- Check interrupts using `/proc/interrupts`

Example:
IRQ storm causes 90% CPU usage.

---

93. High CPU usage — how will you fix it?

Fix depends on cause:

- Optimize code loops
- Fix busy-waits
- Add sleep or blocking calls
- Fix interrupt flooding
- Adjust scheduler priority

Example:
Polling loop → replace with interrupt-based design.

---

94. Memory usage keeps increasing — how will you find the memory leak?

Steps:

- Monitor `/proc/meminfo`
- Use `top`, `ps`
- For kernel: enable `kmemleak`
- For user space: use `valgrind`
- Track allocations

Example:
Daemon forgets to free buffers after socket disconnect.

---

95. Driver works intermittently — how will you debug it?

Likely causes:

- Timing issues
- Missing locks
- Hardware reset problems

Steps:

- Add debug logs
- Test on SMP & UP
- Validate power management callbacks
- Check error handling paths

Example:
Driver fails after suspend → resume path missing reinitialization.

---

96. Device is not detected after boot — how will you debug it?

Steps:

- Check device tree
- Verify driver is loaded
- Check `dmesg`
- Verify bus (I2C/SPI/PCI)
- Probe device manually

Example:
I2C sensor not detected → incorrect I2C bus number.

---

97. Application crashes frequently — how will you debug it?

Steps:

- Enable core dumps
- Use `gdb`
- Check stack trace
- Run with `valgrind`
- Verify input handling

Example:
Crash due to buffer overflow in string handling.

---

98. What is systemd, and how does it manage services in Linux?

systemd is the **init system and service manager** in modern Linux.

Features:

- Parallel service startup
- Dependency management
- Logging via `journald`
- Service monitoring

Example:
`systemctl start ssh` starts SSH service and its dependencies.

---

99. What is the difference between a kernel module and built-in kernel code?

Answer

Kernel Module	**Built-in Code**
Loadable at runtime	Compiled into kernel
Can be unloaded	Cannot be unloaded
Smaller kernel	Larger kernel

Example:
USB driver as module vs scheduler built-in.

---

100. How do you debug memory leaks in kernel space versus user space?

Kernel space:

- `kmemleak`
- `slabtop`
- Debug alloc/free paths

User space:

- `valgrind`
- `AddressSanitizer`
- Heap profiling tools

Example:
Kernel leak → unreleased `kmalloc`
User leak → missing `free()`
