

Topic 5: System Software

Kernel mode, the boot process, linking and containers: what happens beneath your program.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 5.1 Operating systems, 5.2 Language translators. Try the challenge questions on the last pages before you read the answers.

User mode, kernel mode and system calls

The CPU runs in (at least) two privilege levels. Ordinary programs run in **user mode** and cannot touch hardware or other programs' memory. The operating system **kernel** runs in **kernel mode** with full access.

When a program needs a protected service (read a file, send data, create a process) it makes a **system call**. This executes a special instruction that traps into the kernel; the CPU switches to kernel mode, the kernel checks the request is legitimate, performs it, then switches back. A call such as `print()` ends up as a write system call.

This design is what stops a buggy or malicious program from reading another program's passwords or wiping the disk.

The boot process: the chicken-and-egg problem

RAM is empty at power-on, yet the CPU needs instructions to start. The answer is a small program in non-volatile memory:

- **Firmware (BIOS/UEFI)** runs first from flash on the motherboard. It performs the power-on self-test and initialises hardware.
- It finds a **bootloader** on the storage device and loads it into RAM.
- The bootloader loads the **kernel**, which starts drivers and the first user process.
- That process starts services and finally the login screen.

Each stage is small and simple enough to load the next, larger one: hence the word *bootstrap* (pulling yourself up by your bootstraps).

Linking: static vs dynamic

A compiler produces **object code** with gaps where it calls library functions. The **linker** joins object files and libraries into one executable. With **static linking** the library code is copied in; with **dynamic linking** (DLLs on Windows, .so files on Linux) the program just records which library it needs and the loader attaches it at run time.

	Static	Dynamic
Executable size	Larger	Smaller
Library updates / security fixes	Must rebuild every program	Fix once, all programs benefit
Missing or wrong library version	Cannot happen	Program may fail to start

Virtual machines vs containers

A **virtual machine** runs a complete guest operating system on virtual hardware provided by a hypervisor: strong isolation, but heavy (gigabytes, slow to boot). A **container** shares the host's kernel and only isolates the program's files and processes: tiny, starts in milliseconds, but all containers depend on the same kernel.

Challenge questions

1. Why must a user program ask the kernel (via a system call) to write to a disk rather than writing to it directly?
2. Give two advantages and one disadvantage of dynamic linking compared with static linking.
3. Why can a PC not simply load the operating system from the SSD the moment it is switched on?

Worked answers

Only look once you have had a proper go at each question.

1. Direct hardware access in user mode would let any program read or overwrite other programs' or users' data and corrupt the system. Forcing requests through the kernel lets it check permissions, share the device fairly and keep programs isolated.
2. Advantages: smaller executables; one updated library fixes all programs that use it; the library is shared in memory. Disadvantage: the program fails (or behaves differently) if the library is missing or the wrong version.
3. RAM is empty and the CPU has no instructions to run. Firmware in non-volatile memory starts at a fixed address, initialises hardware and knows enough to read the bootloader from storage; the bootloader then loads the kernel.