Tutoriat 1 (Operating Systems)
Introduction

Contents

1 What is a CPU?

A **processor** is a general term for any hardware component that interprets and executes commands from software and hardware (CPUs, GPUs, sound processors, etc.).

A **CPU (central processing unit)** is a hardware component, also known as the "brain" of a computer, responsible for performing calculations, interpreting, and executing instructions and processing data. The stages of the instruction cycle that the CPU uses to execute instructions from a program are the following:

- **Fetch:** the CPU retrieves the next instruction from memory.
- **Decode:** the instruction is interpreted, determining what operation needs to be performed.
- **Execute:** the instruction is executed.
- **Write back:** the result is written back to a register or main memory.

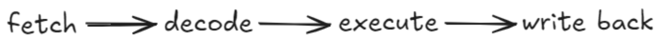


Figure 1: CPU instruction cycle

GHz (gigahertz) is used for measuring the speed of a CPU. 1 GHz = 10^9 Hz, which means that a 3.5GHz processor is capable of executing 3.5 billion instructions per second.

A **CPU core** is an independent processing unit within a CPU that reads and executes program instructions. Each core may handle a task, and therefore a processor containing multiple cores may handle multiple tasks simultaneously. There are two types of CPUs:

- **Single-core:** only one task may be performed at a time, as there is only one core available.
- **Multi-core (dual-core, quad-core, etc.):** two or more tasks may be executed simultaneously, as there are at least two cores.

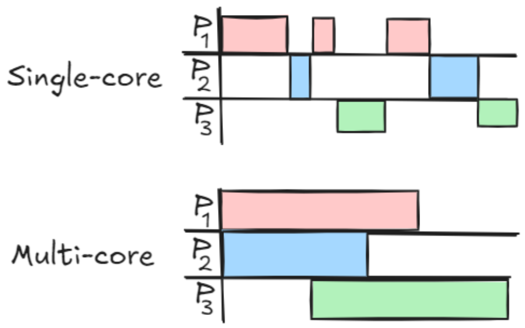


Figure 2: Single-core vs. multi-core processing

Single-core processors work **concurrently**, switching from task to task almost instantly, *simulating parallelism.* A multi-core processor is capable of performing true parallelism, handling two or more tasks at the same time. The problem with having more tasks to handle than there are cores available is having to select the tasks to allocate the CPU to; not only that, they must also be executing for a certain amount of time. This is known as **CPU scheduling**, which will be studied further next time.

2 What is an operating system?

An **operating system** is an intermediary (an interface) between the hardware of a computer and the user. An OS aims to manage the memory of the system, the handling of the CPU, storage space, I/O devices, system and application errors, and so on, so as to create an adequate environment for the user to perform their tasks.

Usually, the way a user interacts with an OS is through a **CLI (command-line interface**, e.g. Windows PowerShell), or through a **GUI (graphical user interface)**.

Each OS has its own advantages and disadvantages, having been built with a certain set of goals in mind. For example, **Windows** is a commercial system, known for being user-friendly, very focused on graphical interfaces; meanwhile, **Linux** distributions are mostly based on using the terminal (which is significantly harder). When building an OS, it would be preferred for it to meet some general requirements:

- **User experience:** interactions with the system must take place in an accessible manner.
- **Executing programs:** it must provide an adequate environment for the user to perform certain tasks.
- **Resource management (efficiently):** it must handle the CPU, system memory, storage space, and I/O devices. Obviously, it would be preferable that the resource handling take place efficiently, maximizing the overall performance of the system.
- **Security:** the integrity and privacy of the user's data must be preserved.
- **Reliability:** the system must continue to operate smoothly, even in the case of an error or an exception.

There are two types of programs on a computer:

- **System programs:** software applications that manage the hardware components; e.g. the actual operating system (Windows, macOS, Linux), drivers (audio drivers, touchpad drivers), compilers, and others.
- **User programs:** notepad, video games, photo editing apps, etc.

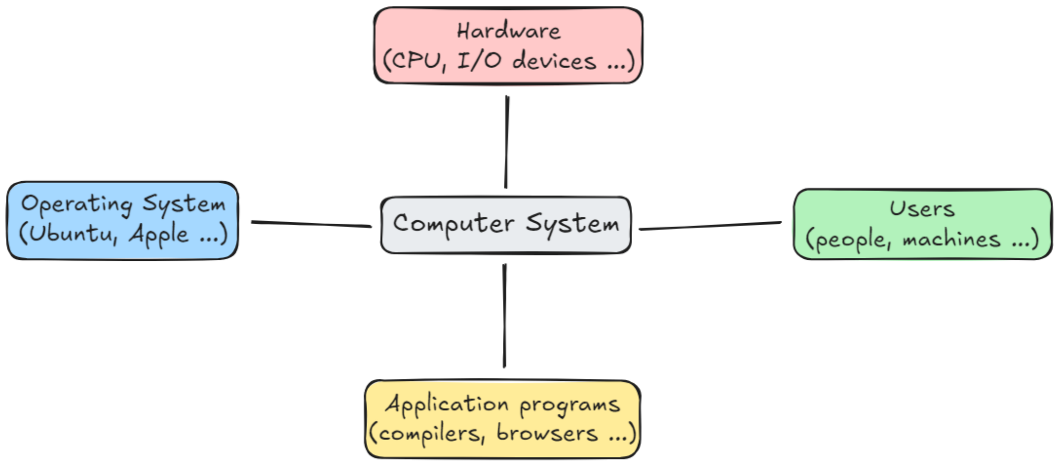


Figure 3: Structure of a computer system

The structure of a computer may be divided into four categories:

- **Hardware:** CPU, memory, I/O devices, etc.
- **Operating system:** Apple, Ubuntu, etc.
- **Programs:** either system programs or user programs.
- **Users:** real people, machines, or other computers.

The **kernel** is the core component of an OS. It is a program that runs permanently once the computer is turned on and has absolute control over the entire system, with the ability to manage hardware, memory, processes (to be studied next time), device drivers, system calls (chapter 4), and allocation of system resources.

There are various approaches to building a kernel. For example, the **monolithic structure** places all the functionality of the kernel into a single static binary file that runs in the same memory space. This approach brings a distinct performance advantage, but is quite difficult to implement and extend; also, changes to one part of the system can have wide-ranging effects on other parts.

The **layered approach** makes the system modular, breaking it into a number of layers (levels). The layers are selected so that each uses functions (operations) and services of only lower-level layers. However, it is challenging to appropriately define the functionality of each layer; in addition, the performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain a service.

A **microkernel** is a minimal operating system that provides only the most essential services, such as memory management, thread management and inter-process communication (to be studied in the future). Other services are implemented as user-level programs that run outside the kernel, resulting in a smaller kernel. One benefit of this approach is that it makes extending the OS easier; furthermore, it provides more security and reliability, since most services will be running in user mode, and if a service fails, the rest of the system remains untouched.

Another methodology involves using **loadable kernel modules (LKMs)**. Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. Linking services dynamically while the kernel is running is preferable to adding new features directly to the kernel, which would require recompilation every time a change was made.

In practice, very few systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in **hybrid systems**.

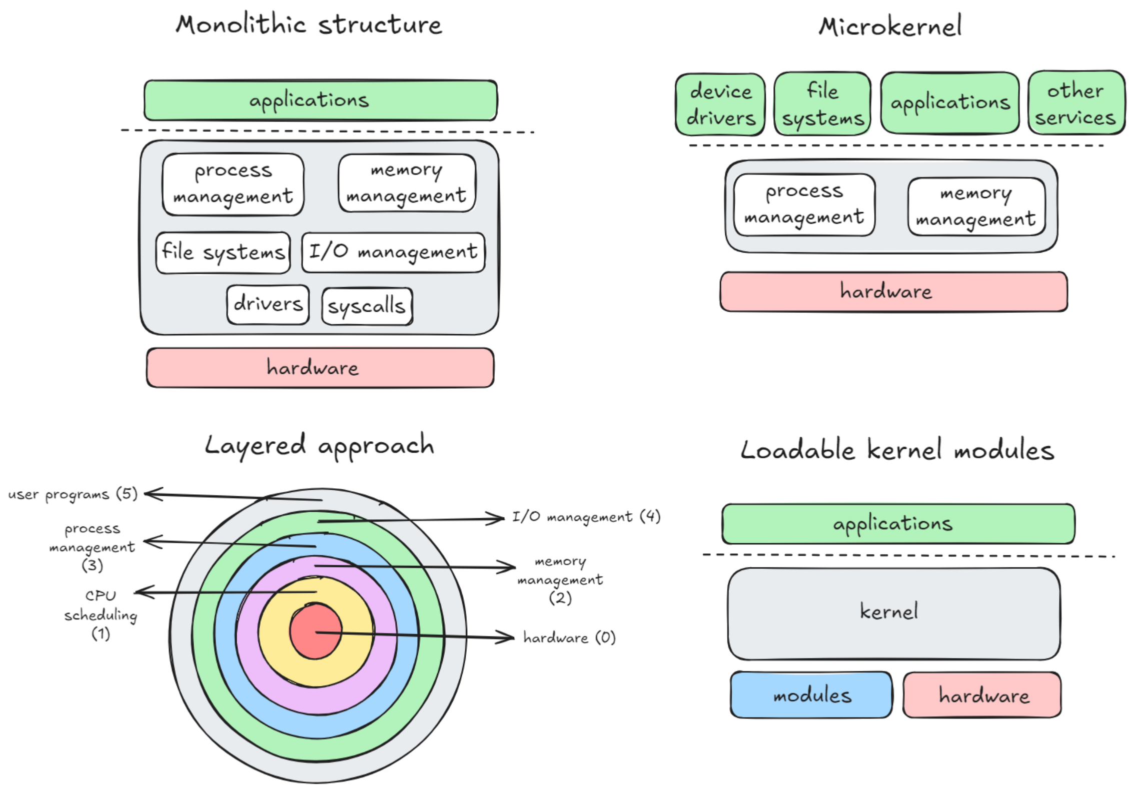


Figure 4: Different types of kernels

3 Linker and loader

The two steps in running a program are producing the executable file and then loading it into RAM. A **linker** is a program that takes object files (and other pieces of code) generated by the compiler and combines them with any necessary libraries to originate an executable file. A **loader** is a program that aims to allocate memory space to programs and load them into memory.

The responsibilities of the linker are as follows:

- **File combining:** the linker takes multiple object files and libraries and merges them into one file.
- **Memory address assignment:** it determines the memory addresses for the code and data segments.

- **Symbol resolution:** it connects different parts of the code, such as a function call in one file with the actual function definition in another file.
- **Code optimization:** the code generated by the compiler is optimized to reduce the program size and improve performance.

The responsibilities of the loader are as follows:

- **Loading:** it brings the executable file into RAM.
- **Memory allocation:** allocates the space for the program to run.
- **Dynamic linking:** connects any necessary external libraries or modules to the program at runtime.

4 Interrupts and exceptions

An **interrupt** is a signal generated by a hardware component or a software application that prompts the CPU to temporarily pause its current task and redirect its attention towards an event of higher priority.

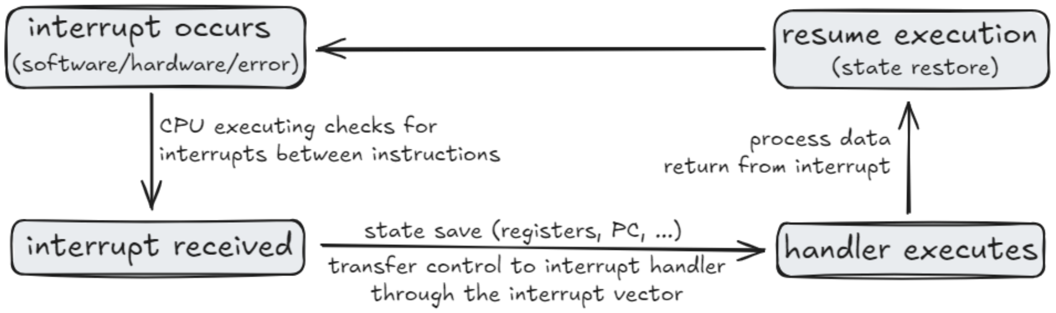

interrupt occurs
(software/hardware/error)
resume execution
(state restore)
CPU executing checks for
interrupts between instructions
process data
return from interrupt
interrupt received
state save (registers, PC, ...)
transfer control to interrupt handler
through the interrupt vector
handler executes

Figure 5: Cycle of an interrupt

Interrupts can be categorized into two types:

- **Hardware:** generated by external devices such as the mouse, keyboard, etc.
- **Software:** also known as **traps**. They are generated by software applications either intentionally (such as a program making a system call) or unintentionally (due to errors, such as division by zero).

The mechanism for handling interrupts is the following:

- The signal is received. The CPU finishes executing the current instruction and performs a **state save** (saving the current status, which includes register values and the **program counter** = a special register which holds the memory address of the next instruction to be executed).
- The code of the interrupt signal is used as an index in the **interrupt vector table** in order to find the correct **ISR (interrupt service routine)**, which is a piece of code designed to handle that specific event.
- The ISR is executed. Once it's done, a **state restore** is performed (all of the data saved before the execution of the routine is restored) and the execution of the interrupted task is resumed.

5 Memory hierarchy in an operating system

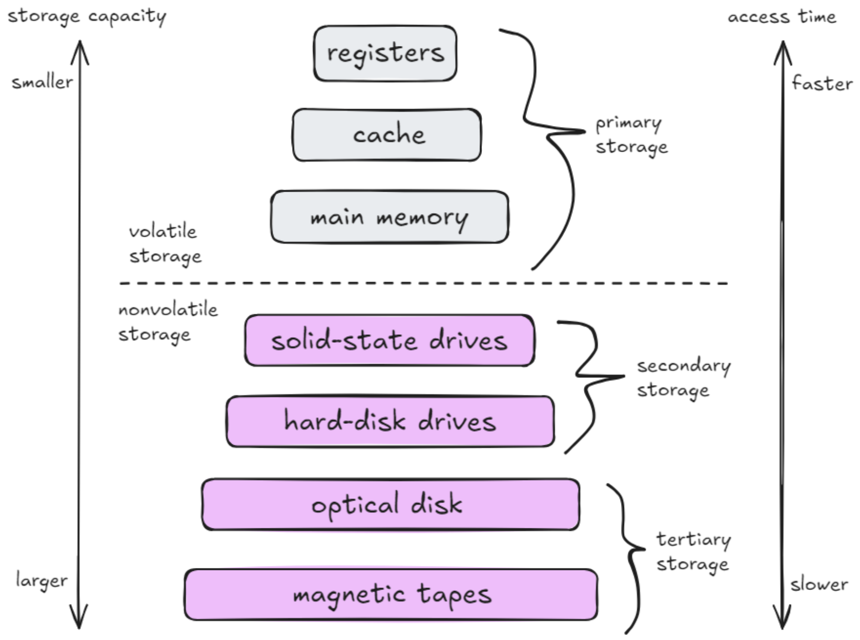


Figure 6: Hierarchy of the memory in an OS

The memory of an OS can be classified into **volatile memory** and **nonvolatile memory (NVM)**. *What is volatile memory?* When a computer shuts down, all of the data stored in volatile memory are permanently lost; thus, it is a short-term memory. Non-volatile memory is for long-term storage of data that does not disappear once the PC is shut down. The operating system itself is stored in NVM (and is loaded into RAM once the computer is turned on).

5.1 Volatile memory

The smaller the dimensions of the memory, the faster it is accessed. With **registers** being the smallest form of memory, they are also the quickest to access. They are stored within the CPU and used for temporary data storage purposes, such as memory addresses and arithmetic results.

Certain data are frequently accessed, such as web pages or the result of a specific arithmetic operation. Thus, in order to gain quicker access times, the resources associated with the web page or the result of the arithmetic operation get stored in a **cache**, which is a quickly accessible memory that contains copies of frequently requested data.

A **cache hit** is when the cache is checked for a certain resource and it does contain it; if it were not to have the respective resource, it would be considered a **cache miss**. Obviously, the performance of a system declines when cache misses consistently take place.

RAM (random access memory) is the main memory, having fast access times, and is used to store data processed by the CPU. The operating system gets loaded into RAM every time the computer is turned on; the software applications that run and the data associated with those applications also get loaded into RAM. *What does random access mean?* Information does not necessarily get stored in a particular order; the data can get stored in (and may be accessed from) any location at any time.

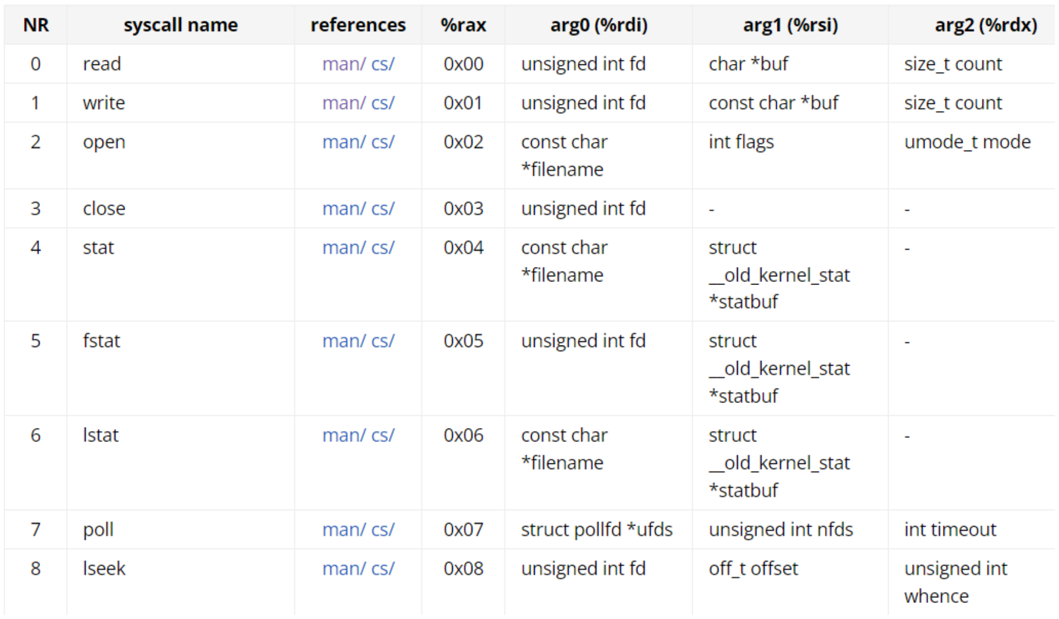
The amount of RAM (measured in GB) is an important computer specification, which majorly affects the performance of a system. The more RAM, the more space you have to run resource-intensive applications (or a larger number of apps). RAM may be implemented in a multitude of ways, such as **DRAM (dynamic random access memory)**, or **SRAM (static random access memory)**.

5.2 Nonvolatile memory

There are various forms of NVM, such as **ROM (read-only memory**; stores **firmware** = software deeply embedded into the memory of hardware), **flash memory** (used by **SSDs** = solid-state drives, **USBs** = universal serial bus, and others), **magnetic storage** (for **HDDs** = hard-disk drives) and **EEPROM (electrically erasable programmable read-only memory**; data may be deleted and rewritten).

Secondary storage is an application of the concept of nonvolatile memory. It represents a category of devices used to store data over a long period of time, e.g. HDDs, SSDs and USBs.

6 System calls



NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count
2	open	man/ cs/	0x02	const char *filename	int flags	umode_t mode
3	close	man/ cs/	0x03	unsigned int fd	-	-
4	stat	man/ cs/	0x04	const char *filename	struct __old_kernel_stat *statbuf	-
5	fstat	man/ cs/	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	-
6	lstat	man/ cs/	0x06	const char *filename	struct __old_kernel_stat *statbuf	-
7	poll	man/ cs/	0x07	struct pollfd *ufds	unsigned int nfds	int timeout
8	lseek	man/ cs/	0x08	unsigned int fd	off_t offset	unsigned int whence

Figure 7: Some frequently used syscalls on Linux

A processor can run in at least two modes: **kernel mode**, with the ability to manage any resources related to the computer (e.g. hardware devices and memory), or **user mode**, which imposes restrictions over various sensitive commands, in order to prevent system crashes and security breaches. In addition to these two modes, some systems feature multiple other "rings" (modes) for protection purposes.

Recall that the program which runs permanently on the computer and has absolute control over the entire system is the **kernel** (the "core" of the OS). When a processor runs in *kernel mode*, it has maximum privileges and can execute any of the kernel's functions. Most programs run in user mode (e.g. notepad does not need access to your hardware components or memory).

Usually, an application running in user mode that crashes or returns an error affects only itself, while a kernel mode crash can disrupt the whole system. The way a computer distinguishes between the two modes is through the value of a **mode bit** (a single-bit flag in a computer's processor).

The transition from user mode to kernel mode takes place during system calls; when the system call is finalized, the system goes back to user mode. *What is a system call?* A **syscall** is a call to a special function, a request to the kernel to allow the CPU to run in *kernel mode*, in order to obtain the necessary privileges for handling a specific task. There are all sorts of system calls, such as the ones used for **file management** purposes (creating, opening, reading, and writing files), or the ones used for managing processes (to study next time).

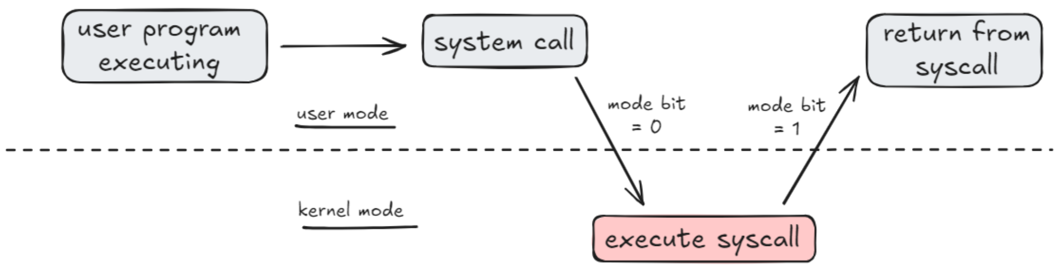
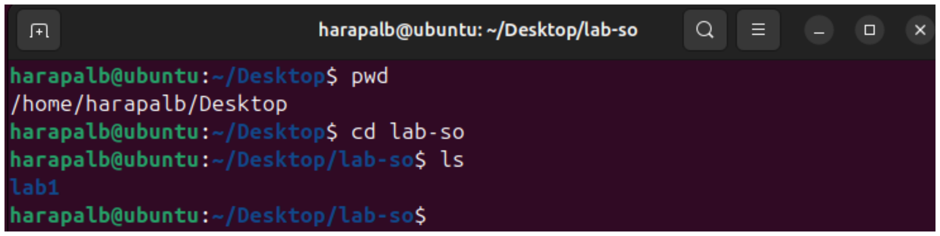


Figure 8: Switching between the two modes

7 Introduction to the Linux terminal

The Linux terminal is a text interface based on the use of commands to interact with the system. It is appreciated and frequently used for its speed, control over the system, and also because of the limitations of GUIs. Furthermore, it allows the possibility of automatizing certain tasks.



```
harapalb@ubuntu:~/Desktop$ pwd
/home/harapalb/Desktop
harapalb@ubuntu:~/Desktop$ cd lab-so
harapalb@ubuntu:~/Desktop/lab-so$ ls
lab1
harapalb@ubuntu:~/Desktop/lab-so$
```

In the terminal, certain symbols appear before each command. In the image above, we can spot the following elements: the username (*harapalb*), the name of the host PC (*ubuntu*), the current working directory (∼ is an abbreviation for */home/your_username*) and the current mode (*$* for user mode, *#* for admin/root mode). Together, these elements form the following structure:

```
[username]@[hostname]:[path][mode]
```

7.1 Analyzing an executable file

Consider the following program *test.c*:

```c
#include <stdio.h>

int main() {
    char text[] = "un string";
    printf("%s\n", text);
    return 0;
}
```

The program can be compiled by running the command:

```
gcc test.c
```

Thus, the *test.c* source file is compiled using **GCC (GNU Compiler Collection)**. Since there was no specified name for the output file, it was named *a.out* by default. In order to run the executable file, enter the following command:

```
./a.out
```

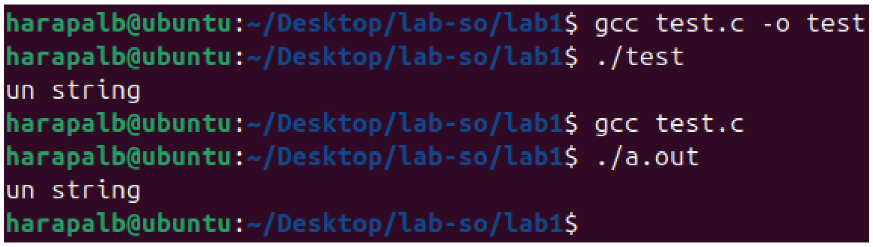
The symbol . represents the current working directory, while ./ is used to execute a specific file in the current directory. If we wish for the compiled file to be named differently, we can rerun the command with the extra parameter *-o*, allowing us to rename the file:

```
gcc test.c -o test
```

Now, the command to run the executable file would be:

```
./test
```

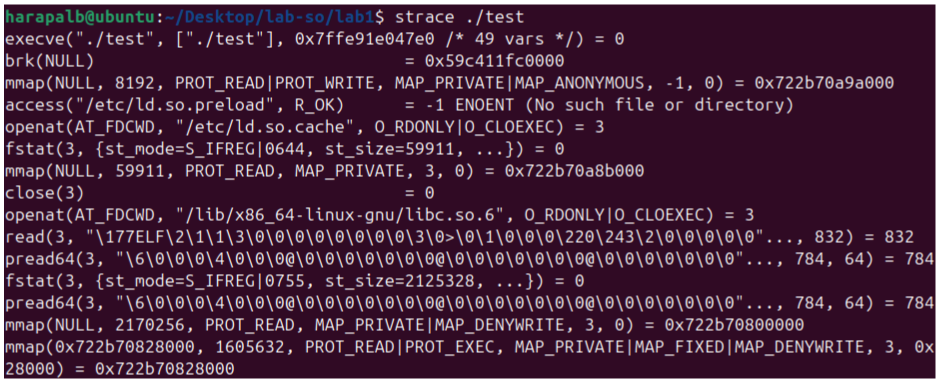
All of the commands shown above would look like this in the terminal:



```
harapalb@ubuntu:~/Desktop/lab-so/lab1$ gcc test.c -o test
harapalb@ubuntu:~/Desktop/lab-so/lab1$ ./test
un string
harapalb@ubuntu:~/Desktop/lab-so/lab1$ gcc test.c
harapalb@ubuntu:~/Desktop/lab-so/lab1$ ./a.out
un string
harapalb@ubuntu:~/Desktop/lab-so/lab1$
```

7.1.1 Listing the used syscalls

Executable files require certain system calls to perform their tasks. Enumerating those calls can be done with help of the command **strace (system call trace)**; thus, *strace ./test* executes the file *test* and lists the used system calls.

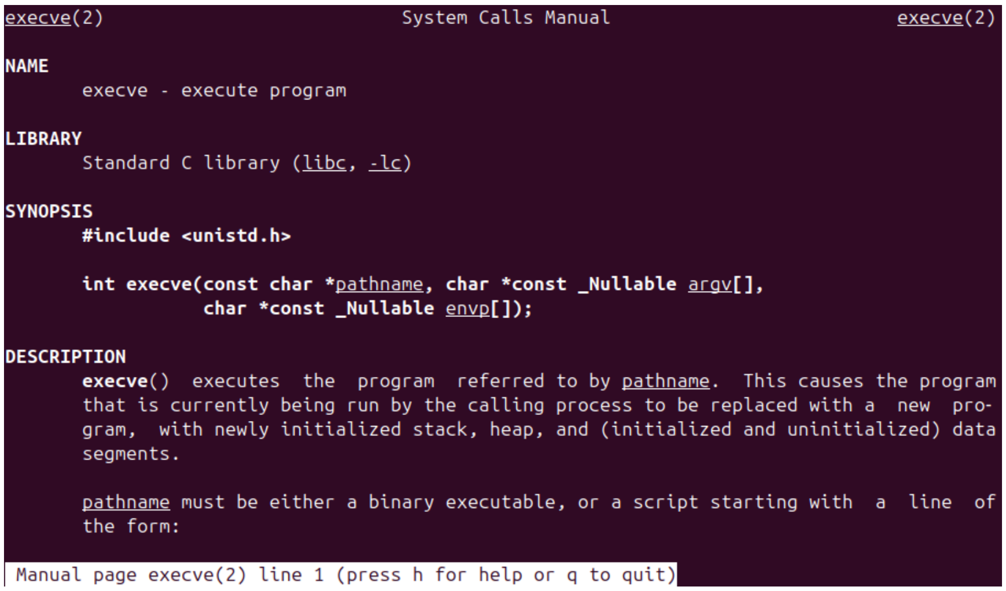


```
harapalb@ubuntu:~/Desktop/lab-so/lab1$ strace ./test
execve("./test", ["./test"], 0x7ffe91e047e0 /* 49 vars */) = 0
brk(NULL)                               = 0x59c411fc0000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x722b70a9a000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=59911, ...}) = 0
mmap(NULL, 59911, PROT_READ, MAP_PRIVATE, 3, 0) = 0x722b70a8b000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x722b70800000
mmap(0x722b70828000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x
28000) = 0x722b70828000
```

Each line of the output above follows the same structure:

```
syscall(arguments...) = return_value
```

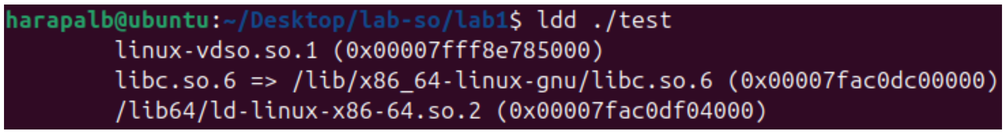
Some of the syscalls listed above which will be further studied are *execve*, *fstat*, and *mmap*. The identified syscalls can be further analyzed with the help of the command *man*. For example, if we wish to find the workings of the *execve* syscall, we can run **man 2 execve**, obtaining the following output:



```
execve(2)                        System Calls Manual                       execve(2)

NAME
       execve - execute program

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <unistd.h>

       int execve(const char *pathname, char *const _Nullable argv[],
                  char *const _Nullable envp[]);

DESCRIPTION
       execve()  executes  the  program  referred to by pathname.  This causes the program
       that is currently being run by the calling process to be replaced with a  new  pro-
       gram,  with newly initialized stack, heap, and (initialized and uninitialized) data
       segments.

       pathname must be either a binary executable, or a script starting with  a  line  of
       the form:

 Manual page execve(2) line 1 (press h for help or q to quit)
```

7.1.2 Listing the used libraries

glibc (GNU C Library) is the standard C library for Linux systems, providing essential functions such as *printf* and *malloc*. A program that makes use of these functions will have to include this library (and possibly other libraries). Listing the libraries required for a program to run can be done with the help of the command *ldd (list dynamic dependencies)*:



```
harapalb@ubuntu:~/Desktop/lab-so/lab1$ ldd ./test
        linux-vdso.so.1 (0x00007fff8e785000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fac0dc00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fac0df04000)
```

The output consists of the required library and its assigned virtual memory address in the process' address space (these concepts will be explained more in depth at a later time). The first library, **Linux vDSO (virtual dynamic shared object)**, helps the program speed up system calls. The second is **glibc**. The last one is a dynamic linker and loader, resolving library dependencies and loading the libraries needed by the program.