**Question#01: Explain the sub segments of Stack in memory layout for a process.**

**Answer:**

Basically **Stack**  itself is a single memory segment of memory layout, but it is divided into multiple smaller parts based on its usage and content.

It is primarily used to manage function calls and local variables. When a function is called, it creates a **stack frame** to store local variables, function parameters and return address.

**Visual representation of the Stack Frame**



**Understanding Function Stack Frames**

### **1. Caller’s Function**

The caller's function represents the previous stack frame in a function call sequence. It holds essential details about the function that invoked the current function.

### **2. Function Parameters**

Function parameters are stored in dedicated stack components when a function is invoked. These parameters help pass values to the function for processing.

#### **Example:**

**function sayHi(message) {**

**console.log(`Dear Hasan! ${message}`);**

**}**

In this example, message is a parameter of the sayHi function. When this function is called, the message parameter is stored in the function stack frame.

### **3. Return Address**

The Return Address is automatically pushed onto the stack when a function is called. It stores the memory address of the instruction to execute after the function completes. This ensures the correct continuation of program execution after the function returns.

### **4. Local Variables**

All variables declared within a function are stored in this part of the stack. These variables exist only for the duration of the function execution and are removed when the function exits.

### **5. Saved Registers**

Saved Registers store crucial register values required for execution flow, memory access, and function calls. Some essential registers include:

* **EBP (Base Pointer):** Acts as a frame pointer.
* **ESP (Stack Pointer):** Tracks the top of the stack. These registers are preserved during function calls to maintain execution integrity.

### **6. Frame Pointer**

The frame pointer (commonly ebp in x86 architecture) is used to reference local variables and function parameters within the stack frame. It points to a fixed position within the stack, simplifying access to variables and parameters relative to its location.

### **7. Stack Pointer**

The stack pointer (esp) is a CPU register that:

* Tracks the current position of the stack.
* Allocates and deallocates space for stack frames during function calls and returns.

Understanding these stack components is crucial for debugging, optimizing function calls, and ensuring efficient memory management in programming.

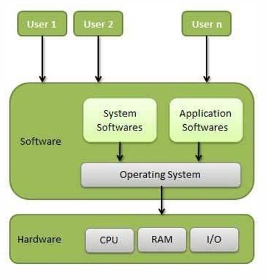
**Question#02: Explain the basic architecture of Operating Systems.**

**Answer:**

**Operating System(OS) Overview**

Operating System is an interface between computer user & hardware. It is the low level software that supports computer’s basic functions. It manages the computer hardware, software resources and provides common services for computer programs.

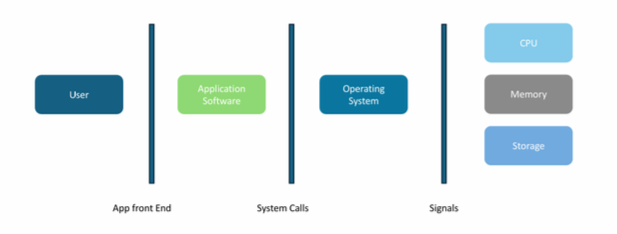
**Operating System Architecture(System View)**



OS architecture from system point view

Let’s take a look at another picture where we will show how the user interacts with the computer through OS.

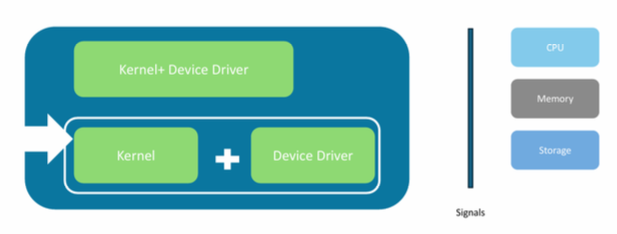
**OS Architecture (User View)**



OS architecture from user point view

When a user interacts with an application, the application communicates with the OS using system calls. The OS then processes the request and interacts with the hardware through signals.

### **Main Components of an OS**

****

An Operating System is primarily composed of **two main components**:

#### **1️⃣ Kernel (The Heart of the OS)**

The **kernel** is the core of the operating system, responsible for managing system resources and communication between hardware and software.

🔹 **Functions of the Kernel:**

* **Process Management** – Scheduling, creating, and terminating processes
* **Memory Management** – Allocating and deallocating memory for processes
* **File System Management** – Managing file storage, access, and security
* **Device Management** – Handling communication with hardware using drivers
* **Security & Access Control** – Managing user permissions and data protection

#### 

#### **2️⃣ Device Drivers**

Device drivers act as a **bridge between the OS and hardware devices** like printers, keyboards, and network adapters. They allow the OS to communicate with hardware components efficiently.

### **How the OS Works (Step-by-Step)**

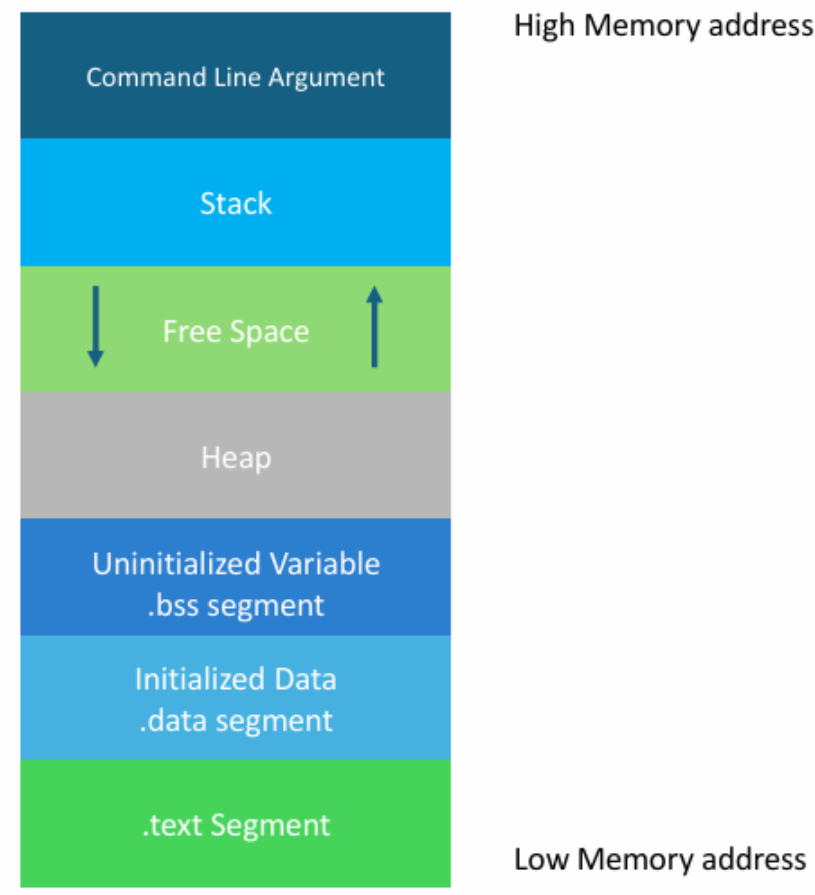
1️⃣ The **User** interacts with an application.  
2️⃣ The **Application** sends requests via **System Calls** to the OS.  
3️⃣ The **Kernel** processes the request and communicates with the hardware.  
4️⃣ The **Hardware** executes the operation and returns the result.  
5️⃣ The **OS** sends the output back to the user through the application.

**Question#03: Explain the Memory layout for a process.**

**Answer:**

The memory layout of a program refers to how the program’s data is stored in the computer memory during its execution.

A typical memory representation of C program consists of the following sections

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Heap
5. Free space
6. Stack
7. Command line argument

**Text Segment**

This segment is also known as code segment. It contains the executable instruction, I mean program code. This segment is often read-only, to prevent from accidentally modifying its instructions.

**Initialized Data Segment**

All of the static & global variables of the program that are declared and initialized by the programmer are stored in this segment.

**Uninitialized Data Segment**

It is also known as **bss(Block Started by Symbol)** segment. All of the static & global variables of the program that are declared but not initialized by the programmers are stored in this segment. The reason behind it is called **bss** segment is, these un initialized variables are initialized by the kernel to symbolic 0 before the program starts execution.

**Heap**

The **Heap** is a dynamically allocated memory region in a process's memory layout. It grows upwards (towards higher memory addresses) and is used for memory allocation at runtime. It is managed by malloc,realloc,and free system calls. It’s open for both read & write operations. All code dependencies like library, header files are stored here. Heap is expandable.

**Stack**

This segement is used to manage all local variables and function call management. It’s open for read & write operations.

**Command Line Argument**

All of the arguments of functions are store in this segement.

**Free space**

**Free Space** is the unallocated memory region between the Stack and Heap in a process's memory layout. It serves as a buffer, allowing both the Stack (growing downward) and Heap (growing upward) to expand as needed. If either exceeds the available Free Space, it can lead to Stack Overflow or Heap Exhaustion errors.

The memory allocation for a process depends on the virtual address space (VA space) and the available physical memory. When the operating system allocates memory for a process, it essentially decides whether to reuse an already allocated memory block or to allocate a new one.

Related Topics. It is not part of any question answer.

**VA Space**

There are two key decisions the OS typically makes during memory allocation:

1. **Reuse previously used memory blocks**: If there are memory blocks that have been freed by the process and are available for reuse, the OS may choose to reuse those blocks to avoid allocating new physical memory. This is generally done through memory management techniques like *heap management* or *paging*. The operating system tracks freed memory blocks, and if they are available, they can be reused without needing new allocations.
2. **Allocate new memory blocks**: If there are no suitable previously used memory blocks (i.e., when the process requires more memory than has been freed or reused), the OS will allocate new memory blocks. This is typically done by mapping additional pages to the process's virtual address space, and these pages may come from physical memory or be swapped in from disk if physical memory is full.

The operating system uses several techniques to manage this process:

* **Paging**: In paging, the virtual address space is divided into fixed-size blocks called "pages," which are mapped to physical memory. When a process needs more memory, the OS can map additional pages to the virtual address space of the process.
* **Memory Fragmentation**: Over time, memory can become fragmented, where free memory is split into small non-contiguous blocks. To optimize memory usage, the OS may compact memory or move data around, but this can incur additional overhead.
* **Demand Paging**: When a process accesses a page that is not currently in physical memory, the OS will load it into memory (this is also known as a page fault). If there is no free memory, the OS might swap out some pages to disk (virtual memory).
* **Buddy System**: Some OS memory allocators use the buddy system, where memory is allocated in powers of two, making it easier to manage fragmentation and reuse blocks efficiently.

These strategies are designed to provide efficient memory management while balancing the needs of multiple processes and the available system memory.

**In VA space there are 2 values 0 & 2.**

**0 means fixed**

**2 means randomize**

**Question#04: How many cycles are done by the CPU in 1 second?**

**Answer**

The number of CPU cycles completed in one second depends on the CPU's clock speed (also known as the clock frequency), which is measured in Hertz (Hz). The clock speed indicates how many cycles the CPU performs in one second.

For example:

* A CPU with a clock speed of 3.0 GHz (gigahertz) performs **3 billion cycles per second**.
* A CPU with a clock speed of 1.0 GHz performs **1 billion cycles per second**.

To calculate the number of cycles in one second, you can use this formula:

Number of cycles = Clock speed (in Hz)

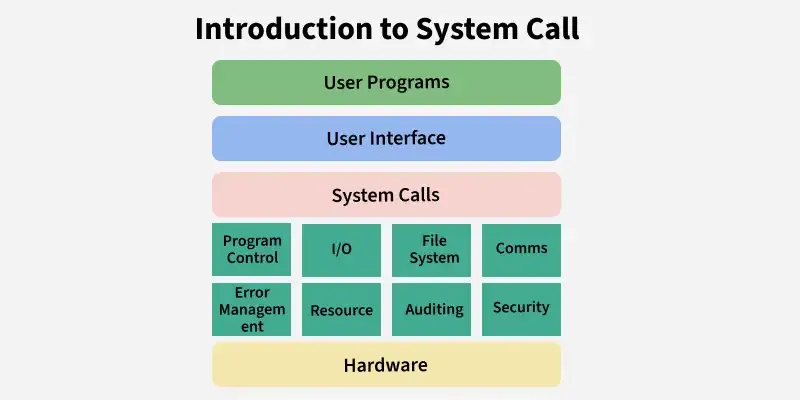
So if you have a CPU running at a clock speed of 2.5 GHz:

= 2.5 billion cycles per second.

Thus, a CPU with a 2.5 GHz clock speed completes 2.5 billion cycles in one second.

**Question#04: What are System Calls & Signals?**

**Answer:**

**System Calls:** System is an interface between application software & OS. It is a mechanism to allow computer programs to request a service from the kernel of the OS.

**Types of System Calls:**

There are 5 important system calls.

1. File System
2. Process Control
3. Memory Management
4. Interprocess Communication
5. Device Management

Examples of system calls

* kill()
* sleep()
* wait()
* mnlock()
* mlock()
* exit()
* exec()
* open()
* pipe()
* writeConsole()

**Signal:** Signal is an asynchronous notification sent to a process or thread in an OS to indicate specific event or condition.

It is generated by the OS itself, by other processes or by the process itself. In microprocessor language, the signal is called an interrupt.

In the motherboard there is a device which is called **System Time**. Its value is 0. It is highly prioritized to the microprocessor to interrupt something.

***Types***

Various types of signals. Each signal is identified by a unique number.

Examples:

SIGINT(interrupt), SIGKILL(kill)

When we press CTRL+C in the terminal it sends a SIGINT to the foreground process.

**OS talk with hardware by signal.**

* 0x01 = write
* 0x80 = invoke system
* 0x04 = representing a specific function

**Question#05: Explain Stack Pointer & Control Pointer.**

**Question#06: When the computer monitor goes to the blue screen?**

**Answer:**

When unsupported data is placed in the stack, which the microprocessor cannot execute by itself, it results in a blue screen.

**Reasons:**

* Unsupported / Not relevant data
* Out of boundary access
* Segmentation fault

**Question#07: What is the function of process management?**