**SPOS**

**Unit – 3**

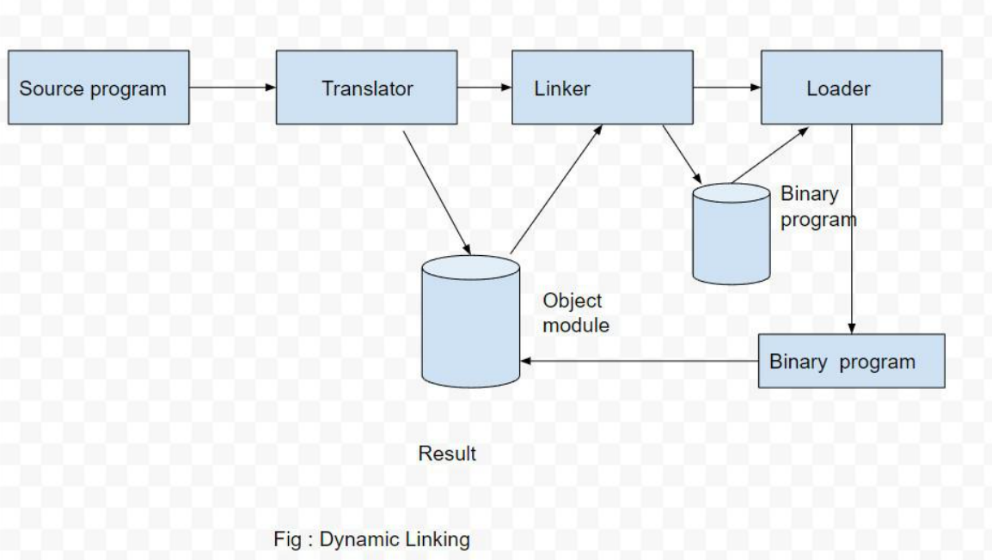
**Q.** **Differentiate between Dynamic and static linking?**

|  | **Static Linking** | **Dynamic Linking** |
| --- | --- | --- |
| Definition | The process of combining all necessary library routines and external references into a single executable file at compile-time. | The process of linking external libraries and references at runtime, when the program is loaded or executed. |
| Linking Time | Occurs at compile-time. | Occurs at runtime. |
| File Size | Generally larger file size, as all required libraries are included in the executable. | Smaller file size, as libraries are linked dynamically at runtime. |
| Flexibility | Less flexible, as any updates or changes to the libraries require recompilation and relinking of the entire program. | More flexible, as libraries can be updated or replaced without recompiling the program. |
| Performance | Faster program startup and direct execution, as all libraries are already linked. | Slightly slower program startup due to the additional linking process, but overall performance impact is minimal. |
| Examples | Executables with file extensions like .exe, .elf, .a, .lib, etc. | Executables with file extensions like .dll, .so, .dylib, etc. |

**Q. DLL ? What is the need of DLL?**

Dynamic linking is a method in system programming and operating systems (SPOs) where the code from libraries is loaded and linked to a program when it is run (at runtime), rather than being combined into the executable file when the program is first built (compile time).

Here are key points about dynamic linking, in simple terms:

* **Runtime Linking:** The linking of libraries happens when the program is loaded or executed by the operating system, not during the compilation phase.
* **Smaller Executables:** Programs (executables) are smaller in size because they only contain a small reference to the needed libraries, not the entire library code itself.
* **Memory Efficiency:** Multiple running programs can share a single, same copy of a common library in the main memory (RAM), which saves significant memory space.
* **Disk Space Savings:** The shared libraries (like .dll files in Windows or .so files in Linux) only need to be stored once on the disk, rather than having a copy embedded in every program that uses them.
* **Easier Updates/Maintenance:** If a bug in a shared library is fixed, only that single library file needs to be updated. All programs using it automatically get the fix the next time they run, without needing to be recompiled or reinstalled.
* **Modular Architecture:** Software can be developed in modules, allowing for easier updates or the addition of new features by simply swapping out a library component without changing the entire core application.
* **Potential Performance Overhead:** The process of resolving function addresses at runtime can make program startup slightly slower compared to static linking, where all addresses are fixed beforehand.
* **Dependency Issues ("DLL Hell"):** If a required shared library is missing or a new version is incompatible with an older program, the program may fail to run, leading to potential compatibility problems known as "DLL hell".

Dynamic linking is a method where a program connects to necessary external code (like libraries) when it is run, rather than when it is first created.

**The process generally involves the following steps:**

* **Source Program to Object Module**: A source program is processed by a translator (like a compiler or assembler) to create an object module.
* **Linking at Runtime**: Unlike static linking where all code is combined upfront, the linker identifies the required external symbol references at "compile time" but does not resolve them until the program loads.
* **Loading**: A loader is responsible for taking the program's machine code (binary program) and placing it into the computer's main memory for execution.
* **Dynamic Resolution**: During the loading or execution phase (runtime), the dynamic loader resolves all the external dependencies by mapping the shared libraries into memory.

This approach saves system resources because multiple running programs can share a single copy of a shared library in memory. It also allows for library updates without needing to re-link the main program.

**Q. Describe the concept of DLL? How dynamic linking can be done with or without import.**

A Dynamic Link Library (DLL) is Microsoft's implementation of a shared library concept [1]. It is a file containing code, data, and resources that multiple programs can use simultaneously [1, 2]. When a program needs the functionality provided by a DLL, it links to it dynamically at run time rather than having the code compiled directly into the executable file [1]. This approach offers several benefits, including reduced memory usage (multiple programs can share a single copy of the DLL in memory) and easier updates (a single DLL can be updated without recompiling the entire application) [1, 2].

**How Dynamic Linking Is Performed**

Dynamic linking can be done in two primary ways: implicit linking (with import libraries) and explicit linking (without import libraries) [1].

**Implicit Linking (With Import Libraries)**

Implicit linking is the most common method and is often referred to as "load-time dynamic linking" [1].

1. **Compiler/Linker Interaction:** When a developer builds an application that uses a DLL, the linker does not embed the DLL's code into the executable. Instead, the linker uses a static library file called an **import library** (typically with a .lib extension) [1, 2].
2. **Import Library's Role:** The import library does not contain the actual function code. It contains information for the system loader, primarily the names of the functions and a list of the DLLs they reside in [1, 2].
3. **Loading Process:** When the application starts, the operating system loader automatically finds and loads the required DLLs into memory. It uses the information in the import library to patch the application's executable code so that function calls resolve to the correct memory addresses within the loaded DLL [1].
4. **Usage:** The application code calls the DLL functions just like it would a normal static library function [1].

**Explicit Linking (Without Import Libraries)**

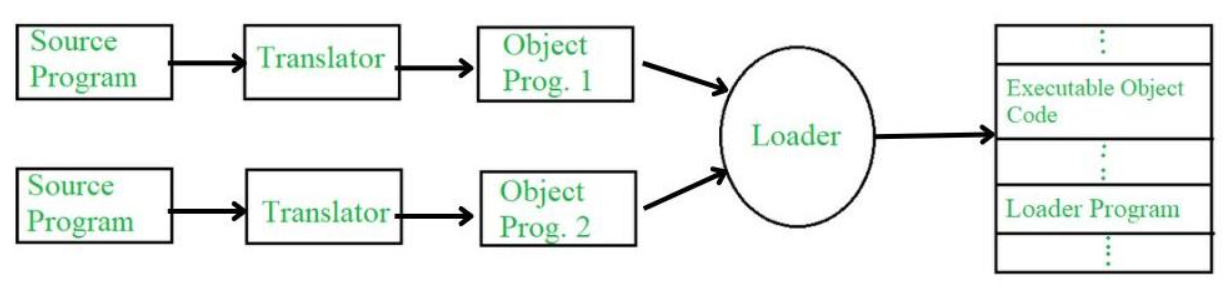
Explicit linking, also known as "run-time dynamic linking," gives the application full control over when the DLL is loaded and unloaded [1].

1. **No Import Library Needed:** This method does not require an import (.lib) file during compilation [1].
2. **Manual Loading:** The application must use specific API functions to load the DLL manually at runtime.
   * In Windows, functions like LoadLibrary or LoadLibraryEx are used to load the DLL into the process's memory [1, 2].
   * GetProcAddress is then used to obtain a pointer to the specific function within the DLL that the application wants to use [1, 2].
   * When the application is finished with the DLL, it uses FreeLibrary to unload it [1].
3. **Usage:** The application calls the DLL functions indirectly through the function pointers it obtained using GetProcAddress. This requires careful handling and error checking in the application code [1].

|  |  |  |
| --- | --- | --- |
| **Feature** | **Implicit Linking** | **Explicit Linking** |
| **Method** | Load-time | Run-time |
| **Import Library** | Required (.lib file) | Not required |
| **Loading** | Automatic by OS loader | Manual using API calls (e.g., LoadLibrary) |
| **Error Handling** | Simpler (load failure crashes app) | More complex (must handle NULL pointers/load failures) |
| **Flexibility** | Less flexible | More flexible (load/unload on demand) |

**Q. List and explain different loader schemes in detail.**

A **loader** is a system program (part of the operating system) that takes an executable program file from storage, places it into the main memory, and prepares it for execution by the CPU. It is the final step in turning your source code into a running program.



**How a Loader Works**

The loader performs several key functions to ensure a program runs correctly in memory:

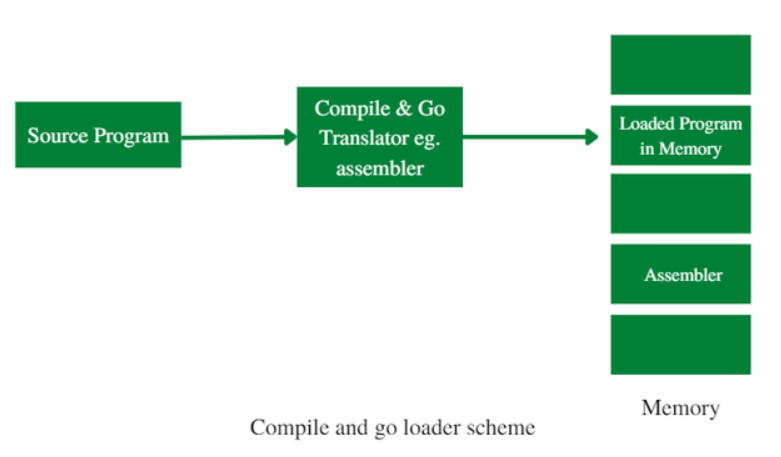
* **Allocation:** The loader determines and reserves the necessary memory space (for code, data, stack, etc.) for the program to run, ensuring it doesn't overlap with other programs.
* **Linking:** Programs often use pre-written functions or libraries (e.g., a printf() function from a standard library). Linking involves resolving these external references by connecting the calls in the main program to the actual memory addresses of those functions in other modules or libraries.
* **Relocation:** Since a program might be loaded into a different memory location each time it runs (not always the same fixed address it was compiled for), the loader adjusts all the program's internal memory references (pointers, addresses) to match its actual current starting address in memory.
* **Loading:** The final step, where the loader physically copies the program's machine code and data from the storage device into the allocated main memory space.
* **Execution Transfer:** After all functions are complete, the loader transfers control (sets the program counter) to the program's starting instruction, initiating its execution.

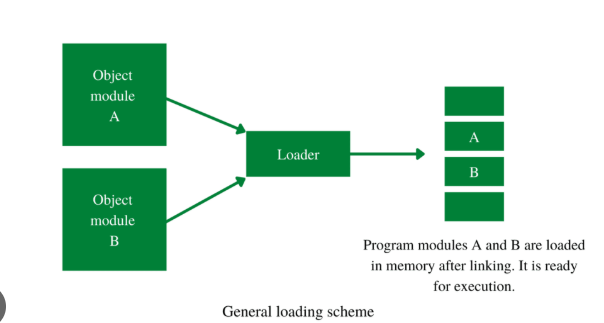
**Key Loader Functions (The "What" Loaders Do)**

Before types, understand the tasks:

* **Allocation:** Giving the program space in memory.
* **Linking:** Connecting different program parts (modules/libraries).
* **Relocation:** Changing memory addresses in the code so it runs in the *allocated* spot, not a fixed one.
* **Loading:** The final step of physically putting the code into memory.

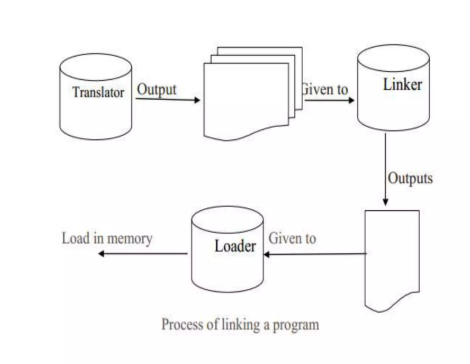
**Types of Loader Schemes (The "How" They Do It)**

1. [**Compile-and-Go Loader**](https://www.google.com/search?q=Compile-and-Go+Loader&sca_esv=cf3d8fcadb69d60c&sxsrf=AE3TifObN9pH3wf3Xg9Xp6geLjdu0yMHsA%3A1765909237246&ei=9aJBadjlDti5vr0PgviImQo&ved=2ahUKEwjiiKGs38KRAxXQ1zQHHSGpEGkQgK4QegQICBAB&uact=5&oq=List+and+explain+different+loader+schemes+in+detail.+in+spos+%2C+give+in+easy+and+simple+words&gs_lp=Egxnd3Mtd2l6LXNlcnAiXExpc3QgYW5kIGV4cGxhaW4gZGlmZmVyZW50IGxvYWRlciBzY2hlbWVzIGluIGRldGFpbC4gaW4gc3BvcyAsIGdpdmUgaW4gZWFzeSBhbmQgc2ltcGxlIHdvcmRzSLgTUABYAHAAeAGQAQCYAcIBoAHCAaoBAzAuMbgBA8gBAPgBAvgBAZgCAKACAJgDAJIHAKAHkAGyBwC4BwDCBwDIBwCACAA&sclient=gws-wiz-serp&mstk=AUtExfCgn-0tJxXB7f2I09ApWK5rwvcadjSP65tKBlT9AUzApBc_jgAQGwzpkq-niBw_MKsUWbCiy_rRIpU_Z-X3LJ4UxF5ommLMieruOQO8ok9b0jK8BYAilFNGLF5SmY511GLw9olXedrRUhKBXOmtZBoskKsRMmXL1yoWme5jwxF7PwyLiAWSRWdIrgtURRNSiRMU9ux4R2pplMRi7pOT72KWyL2Ap1misQsOxyUzN3pvsL-JyPH_Ez0MqBoVzzmmngjwymSXm0UvIMunZLvUAPhHv8BbztroUJTFfd8WUMuMoL3eY2YlmzynryVR8nHERw&csui=3)**:**
   * **Simple:** Reads source code, translates (assembles) line-by-line, and immediately puts the machine code into memory.
   * **Easy to Use:** Good for simple, single-file programs.
   * **Downside:** Re-translates the whole thing every time; inefficient for large or repeated tasks.
2. [**Absolute Loader**](https://www.google.com/search?q=Absolute+Loader&sca_esv=cf3d8fcadb69d60c&sxsrf=AE3TifObN9pH3wf3Xg9Xp6geLjdu0yMHsA%3A1765909237246&ei=9aJBadjlDti5vr0PgviImQo&ved=2ahUKEwjiiKGs38KRAxXQ1zQHHSGpEGkQgK4QegQICBAG&uact=5&oq=List+and+explain+different+loader+schemes+in+detail.+in+spos+%2C+give+in+easy+and+simple+words&gs_lp=Egxnd3Mtd2l6LXNlcnAiXExpc3QgYW5kIGV4cGxhaW4gZGlmZmVyZW50IGxvYWRlciBzY2hlbWVzIGluIGRldGFpbC4gaW4gc3BvcyAsIGdpdmUgaW4gZWFzeSBhbmQgc2ltcGxlIHdvcmRzSLgTUABYAHAAeAGQAQCYAcIBoAHCAaoBAzAuMbgBA8gBAPgBAvgBAZgCAKACAJgDAJIHAKAHkAGyBwC4BwDCBwDIBwCACAA&sclient=gws-wiz-serp&mstk=AUtExfCgn-0tJxXB7f2I09ApWK5rwvcadjSP65tKBlT9AUzApBc_jgAQGwzpkq-niBw_MKsUWbCiy_rRIpU_Z-X3LJ4UxF5ommLMieruOQO8ok9b0jK8BYAilFNGLF5SmY511GLw9olXedrRUhKBXOmtZBoskKsRMmXL1yoWme5jwxF7PwyLiAWSRWdIrgtURRNSiRMU9ux4R2pplMRi7pOT72KWyL2Ap1misQsOxyUzN3pvsL-JyPH_Ez0MqBoVzzmmngjwymSXm0UvIMunZLvUAPhHv8BbztroUJTFfd8WUMuMoL3eY2YlmzynryVR8nHERw&csui=3)**:**

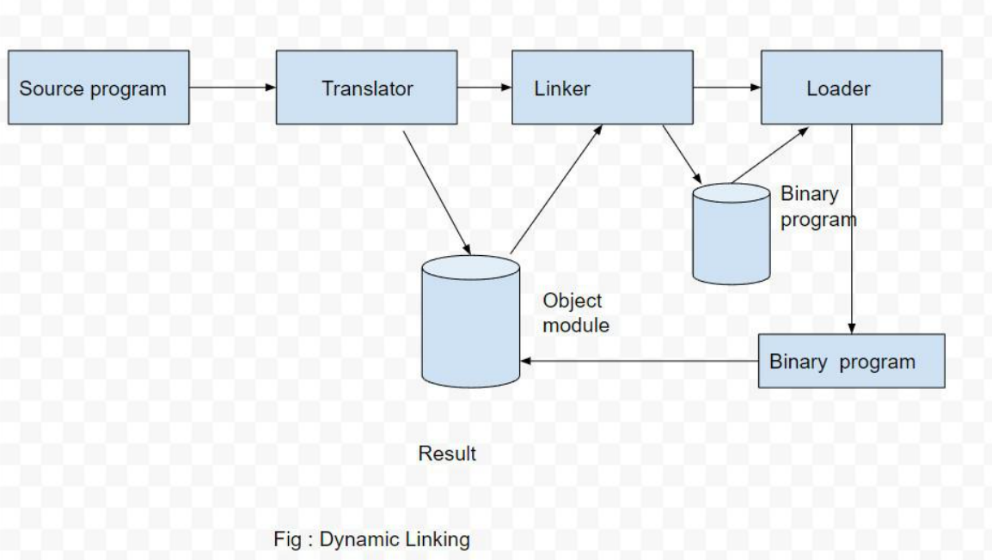


* + **Fixed Addresses:** The programmer or assembler decides *exactly* where in memory the code goes.
  + **Simple Loader:** Loader just places code at the specified address; no relocation needed by loader.
  + **Less Flexible:** Can't easily load programs to different spots.

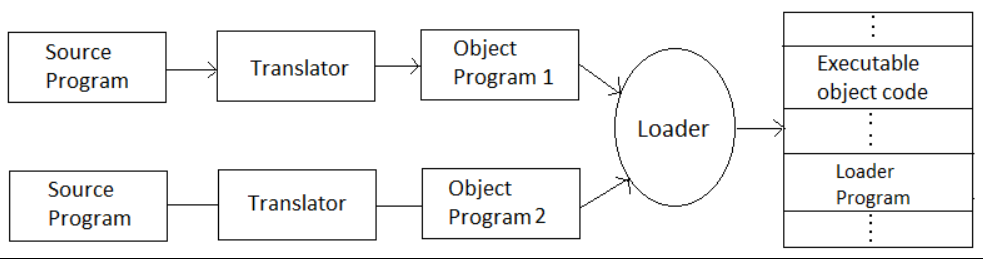
1. [**Relocating Loader**](https://www.google.com/search?q=Relocating+Loader&sca_esv=cf3d8fcadb69d60c&sxsrf=AE3TifObN9pH3wf3Xg9Xp6geLjdu0yMHsA%3A1765909237246&ei=9aJBadjlDti5vr0PgviImQo&ved=2ahUKEwjiiKGs38KRAxXQ1zQHHSGpEGkQgK4QegQICBAL&uact=5&oq=List+and+explain+different+loader+schemes+in+detail.+in+spos+%2C+give+in+easy+and+simple+words&gs_lp=Egxnd3Mtd2l6LXNlcnAiXExpc3QgYW5kIGV4cGxhaW4gZGlmZmVyZW50IGxvYWRlciBzY2hlbWVzIGluIGRldGFpbC4gaW4gc3BvcyAsIGdpdmUgaW4gZWFzeSBhbmQgc2ltcGxlIHdvcmRzSLgTUABYAHAAeAGQAQCYAcIBoAHCAaoBAzAuMbgBA8gBAPgBAvgBAZgCAKACAJgDAJIHAKAHkAGyBwC4BwDCBwDIBwCACAA&sclient=gws-wiz-serp&mstk=AUtExfCgn-0tJxXB7f2I09ApWK5rwvcadjSP65tKBlT9AUzApBc_jgAQGwzpkq-niBw_MKsUWbCiy_rRIpU_Z-X3LJ4UxF5ommLMieruOQO8ok9b0jK8BYAilFNGLF5SmY511GLw9olXedrRUhKBXOmtZBoskKsRMmXL1yoWme5jwxF7PwyLiAWSRWdIrgtURRNSiRMU9ux4R2pplMRi7pOT72KWyL2Ap1misQsOxyUzN3pvsL-JyPH_Ez0MqBoVzzmmngjwymSXm0UvIMunZLvUAPhHv8BbztroUJTFfd8WUMuMoL3eY2YlmzynryVR8nHERw&csui=3)**:**
   * **Flexible:** Loads programs into *any* available memory area by adjusting addresses (relocating).
   * **Separate Steps:** Assembly, linking, and loading happen in separate phases, producing intermediate files (object code, relocation info).
   * **Better for Sharing:** Allows multiple programs to share memory without conflicts.
2. [**Direct-Linking Loader**](https://www.google.com/search?q=Direct-Linking+Loader&sca_esv=cf3d8fcadb69d60c&sxsrf=AE3TifObN9pH3wf3Xg9Xp6geLjdu0yMHsA%3A1765909237246&ei=9aJBadjlDti5vr0PgviImQo&ved=2ahUKEwjiiKGs38KRAxXQ1zQHHSGpEGkQgK4QegQICBAQ&uact=5&oq=List+and+explain+different+loader+schemes+in+detail.+in+spos+%2C+give+in+easy+and+simple+words&gs_lp=Egxnd3Mtd2l6LXNlcnAiXExpc3QgYW5kIGV4cGxhaW4gZGlmZmVyZW50IGxvYWRlciBzY2hlbWVzIGluIGRldGFpbC4gaW4gc3BvcyAsIGdpdmUgaW4gZWFzeSBhbmQgc2ltcGxlIHdvcmRzSLgTUABYAHAAeAGQAQCYAcIBoAHCAaoBAzAuMbgBA8gBAPgBAvgBAZgCAKACAJgDAJIHAKAHkAGyBwC4BwDCBwDIBwCACAA&sclient=gws-wiz-serp&mstk=AUtExfCgn-0tJxXB7f2I09ApWK5rwvcadjSP65tKBlT9AUzApBc_jgAQGwzpkq-niBw_MKsUWbCiy_rRIpU_Z-X3LJ4UxF5ommLMieruOQO8ok9b0jK8BYAilFNGLF5SmY511GLw9olXedrRUhKBXOmtZBoskKsRMmXL1yoWme5jwxF7PwyLiAWSRWdIrgtURRNSiRMU9ux4R2pplMRi7pOT72KWyL2Ap1misQsOxyUzN3pvsL-JyPH_Ez0MqBoVzzmmngjwymSXm0UvIMunZLvUAPhHv8BbztroUJTFfd8WUMuMoL3eY2YlmzynryVR8nHERw&csui=3)**:**



* + **Powerful:** Combines linking and loading in one go.
  + **Handles Modules:** Manages multiple object modules (like different files or libraries) and resolves references between them.
  + **Efficient for Big Projects:** Finds external symbols (from other files) and links them directly.

1. [**Dynamic Linking Loader (Load-on-Call)**](https://www.google.com/search?q=Dynamic+Linking+Loader+%28Load-on-Call%29&sca_esv=cf3d8fcadb69d60c&sxsrf=AE3TifObN9pH3wf3Xg9Xp6geLjdu0yMHsA%3A1765909237246&ei=9aJBadjlDti5vr0PgviImQo&ved=2ahUKEwjiiKGs38KRAxXQ1zQHHSGpEGkQgK4QegQICBAV&uact=5&oq=List+and+explain+different+loader+schemes+in+detail.+in+spos+%2C+give+in+easy+and+simple+words&gs_lp=Egxnd3Mtd2l6LXNlcnAiXExpc3QgYW5kIGV4cGxhaW4gZGlmZmVyZW50IGxvYWRlciBzY2hlbWVzIGluIGRldGFpbC4gaW4gc3BvcyAsIGdpdmUgaW4gZWFzeSBhbmQgc2ltcGxlIHdvcmRzSLgTUABYAHAAeAGQAQCYAcIBoAHCAaoBAzAuMbgBA8gBAPgBAvgBAZgCAKACAJgDAJIHAKAHkAGyBwC4BwDCBwDIBwCACAA&sclient=gws-wiz-serp&mstk=AUtExfCgn-0tJxXB7f2I09ApWK5rwvcadjSP65tKBlT9AUzApBc_jgAQGwzpkq-niBw_MKsUWbCiy_rRIpU_Z-X3LJ4UxF5ommLMieruOQO8ok9b0jK8BYAilFNGLF5SmY511GLw9olXedrRUhKBXOmtZBoskKsRMmXL1yoWme5jwxF7PwyLiAWSRWdIrgtURRNSiRMU9ux4R2pplMRi7pOT72KWyL2Ap1misQsOxyUzN3pvsL-JyPH_Ez0MqBoVzzmmngjwymSXm0UvIMunZLvUAPhHv8BbztroUJTFfd8WUMuMoL3eY2YlmzynryVR8nHERw&csui=3)**:**
   * **On-Demand:** Links libraries only when a function is actually called, not at start.
   * **Saves Memory:** Loads shared libraries once and lets many programs use them.
   * **Modern Approach:** Common in modern operating systems for efficiency.

Q. Explain “General loading scheme (using suitable diagram)” with advantages and disadvantages?

The **General Loading Scheme** is an approach in System Programming and Operating Systems (SPOS) where a translator (assembler/compiler) creates a machine-independent object program, which is then processed by a separate loader utility to be placed into main memory for execution. This improves upon simpler methods by decoupling translation from the final loading and execution steps.

* **Translator (Assembler/Compiler):** Converts the source code into an object program (or object modules if modular). This output is stored in secondary storage (e.g., a hard disk).
* **Loader:** A separate system program, much smaller than the assembler, which reads the object program and performs the necessary functions (allocation, linking, relocation, loading) to prepare it for execution in main memory.
* **Execution:** After the program is loaded, control is transferred to its starting address, and the CPU begins execution.

**Advantages**

* **Memory Efficiency:** The assembler or compiler does not need to reside in main memory during the loading/execution phase, freeing up significant memory space for the user's program.
* **No Re-translation:** The source program does not need to be retranslated (compiled/assembled) every time it runs. The object code, once generated and saved, can be loaded multiple times.
* **Modular Programming:** It allows the source program to be written in multiple modules and potentially different source languages (as long as they produce compatible object files in machine language).
* **Flexibility:** The program can be loaded into different memory locations without programmer intervention, as the loader handles the necessary address adjustments (relocation).

**Disadvantages**

* **Increased Complexity:** The loader itself is a more complex program compared to the simple "compile-and-go" loader, as it must handle object files, symbol tables, and relocation information.
* **Secondary Storage Required:** It requires secondary storage (disk, etc.) to save the intermediate object files, whereas the "compile-and-go" scheme does not.
* **Overhead in Loading:** Although faster than re-translating, the process of reading the object file, allocating memory, and performing linking/relocation adds a slight overhead compared to the immediate execution of "compile-and-go".
* **Loader Occupies Memory:** While smaller than the assembler, the loader still occupies a portion of main memory during its operation.

**Q. Differences between static link library and dynamic link library.**

The primary difference between **static link libraries** and **dynamic link libraries** (DLLs or shared objects) is *when* the library code is linked to the program: static linking occurs at **compile time**, while dynamic linking occurs at **run time**.

**Key Differences: Static vs. Dynamic Libraries**

|  |  |  |
| --- | --- | --- |
| **Property** | **Static Library** | **Dynamic Library (Shared Library/DLL)** |
| **Linking Time** | Compile time (during the build process) | Run time (when the program starts or as needed) |
| **Executable Size** | Larger, as library code is copied into the final executable | Smaller, as the executable only stores the library's address/name |
| **Dependency** | Self-contained; no external library files needed at runtime | Dependent on external library files being present on the system at runtime |
| **Memory Usage** | Inefficient if many programs use the same library, as each gets a separate copy in memory | Efficient, as multiple programs can share a single copy of the library in memory |
| **Updates/Maintenance** | Requires the entire application to be recompiled and re-linked if the library is updated | The library can be updated independently without needing to recompile the main application |
| **Portability/Deployment** | Easier to distribute; the single executable can run on any compatible machine | More complex distribution; the correct library files must also be present on the target system (potential "DLL Hell" issues) |
| **Performance** | Generally faster execution speed due to no runtime linking overhead and better code locality | Slightly slower initial load time due to the dynamic linking process |

**Summary**

* **Static libraries** provide a self-contained, robust executable that is simple to distribute but results in larger file sizes and more difficult updates.
* **Dynamic libraries** promote modularity, saving disk space and memory by sharing code among multiple programs, and allow for easier updates to shared components without full application reinstallation. They require the library to be available on the system where the program runs.

**Q. Explain Design of Direct linking loaders.**

The design of a Direct Linking Loader (DLL) involves a two-pass process to combine multiple program segments (code/data) into one executable, allowing flexible inter-segment calls by building a Global External Symbol Table (GEST) in Pass 1 and resolving references in Pass 2 using External Symbol Dictionary(ESD) & Relocation & Linkage Dictionary(RLD) information from object decks. Key components include segment length, symbol tables (Definition/Use), machine code, and RLD cards, enabling independent assembly and loading.

**Core Concepts & Design Goals**

* **Relocatable Loader:** Can load programs anywhere in memory.
* **Multi-Segment Support:** Handles multiple procedure and data segments.
* **Flexible Referencing:** Allows free referencing between segments (e.g., calling a function in another segment).

**Input from Assembler (Object Deck)**

**Each assembled segment provides:**

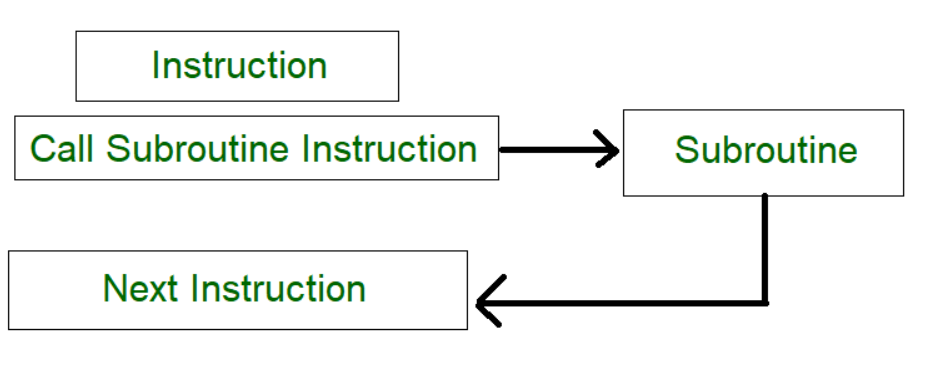
* **Segment Length:** Size of the code/data.
* **External Symbol Dictionary (ESD):** Lists symbols defined in the segment (Definition Table) and used by other segments (Use Table).
* **Text Records (TXT):** Machine code for instructions/data.
* **Relocation & Linkage Dictionary (RLD):** Lists address constants needing modification and which external symbols to use.
* **END Card:** Marks end of segment.

**Q. Write short notes on :**

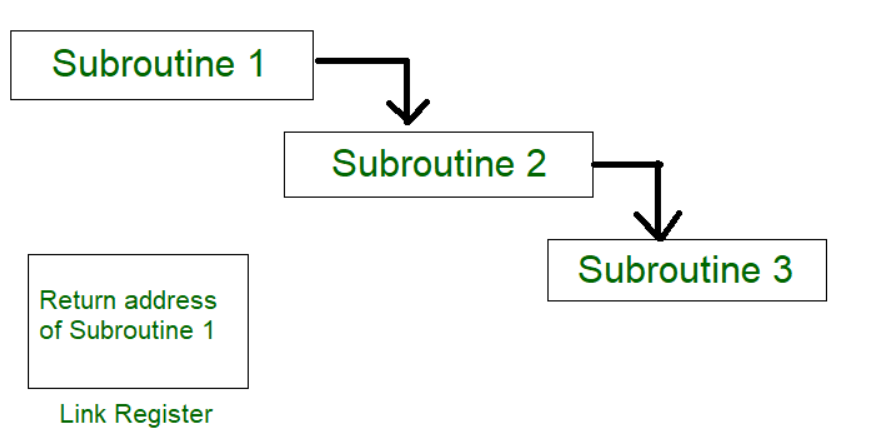
**i) Subroutine Linkage**

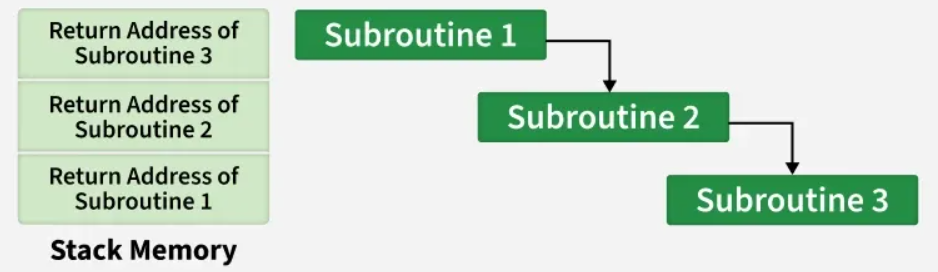
**ii) Overlays**

i) Subroutine Linkage

Subroutine linkage is the mechanism used to transfer control from a calling program to a subroutine (procedure or function) and back to the correct return address after the subroutine's execution is complete. It is a fundamental concept for modular programming, enabling code reusability and more organized program design.

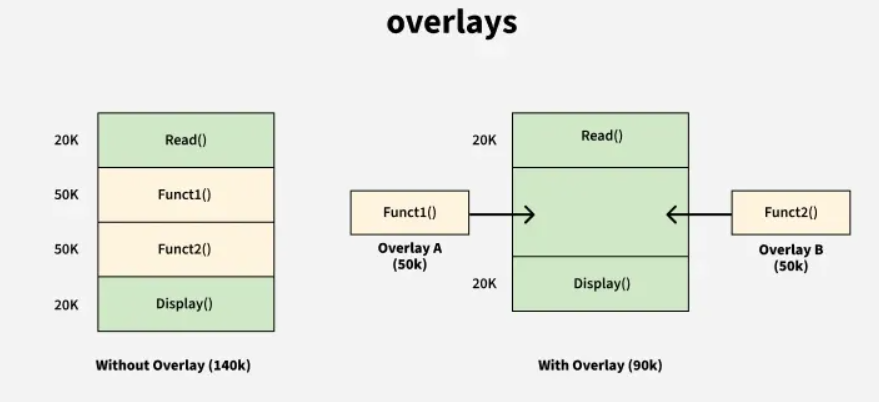
*Process of a subroutine in a program*

* Process:
  + The Call instruction saves the return address (the address of the next instruction in the calling program) in a specific location, such as a dedicated register (link register) or, more commonly in modern systems, a stack.
  + Control is then transferred to the starting address of the subroutine.
  + Parameters or arguments may be passed to the subroutine (via registers, the stack, or specific memory locations).
  + The subroutine executes its tasks using its local variables.
  + A Return instruction at the end of the subroutine fetches the saved return address from its storage location (e.g., pops it from the stack) and transfers control back to the calling program.
* Nesting: Subroutines can call other subroutines (nesting). Using a stack to store return addresses handles this efficiently in a Last-In, First-Out (LIFO) manner, ensuring that each return goes back to the correct immediate caller.

Since the last subroutine called is the first to return (Last In, First Out), a stack is the most efficient way to store these return addresses.

* Advantages: Promotes code reusability, simplifies programming by breaking down complex tasks into smaller modules, and saves memory by avoiding code duplication.

ii) Overlays

Overlays are an older memory management technique used to execute programs that are larger than the available physical main memory. The core idea is that not all parts of a program are needed simultaneously during execution.

* Concept:
  + The program is manually divided into smaller, self-contained modules or segments called overlays, typically organized in a tree structure.
  + Only the essential part of the program (often called the "resident" or "root" module) remains in memory at all times.
  + Other mutually exclusive overlay segments are loaded into a specific, shared region of memory (the "overlay area") as needed.
  + When a needed overlay is not in memory, an "overlay manager" (often part of the program itself, not the operating system) loads it from secondary storage (e.g., a disk) into the overlay area, overwriting what was previously there.

**The basic idea behind overlays is:**

* To load only the portion of a program that is needed for the current task.
* The unused parts of the program are stored on disk or other storage and are loaded into memory as needed.
* Once a part of the program finishes executing, it is unloaded from memory, freeing up space for the next part to be loaded.
* Usage: This technique was crucial in early computer systems with very limited RAM (like the original IBM PC) and is still found in some embedded systems and environments without virtual memory hardware.
* Advantages: Enables the execution of very large programs in resource-constrained systems and makes efficient use of limited RAM.
* Disadvantages: Requires complex design and management by the programmer (the programmer must explicitly define the overlay structure), which increases development complexity and can lead to performance overhead due to frequent loading and unloading of modules. Modern operating systems use more advanced and automatic techniques like paging and segmentation.

**Q. Give complete design of Absolute Loader with suitable example?**

The **absolute loader** is a simple system program that loads object code into memory at fixed, pre-specified addresses without performing relocation or linking. Its design relies on the programmer or assembler knowing the exact memory locations beforehand.

**Design Principles**

The design of an absolute loader is straightforward, typically operating in a single pass. It performs only the fundamental functions of a loader: bringing the program into memory and starting its execution.

The key information is communicated via the object program's records, typically in a specific format like the SIC object program format:

* **Header Record:** Contains the program name, its starting address in memory, and its total length. The loader checks this to ensure the correct program is loaded and can fit in memory.
* **Text Records:** Contain the actual machine code and data, along with the specific memory address where that code should be placed.
* **End Record:** Specifies the absolute address where the loader should jump to begin execution of the loaded program.

**Algorithm**

The simple algorithm for an absolute loader is as follows:

1. **Read Header Record:** The loader reads the header record to get the program name, starting address, and length.
2. **Verify Program:** It verifies the program information, potentially checking if the memory space is available.
3. **Read Text Records in a Loop:**

* Read the next text record from the object file.
* While the record type is not 'E' (End record):
  + - Extract the specified load address and the object code/data from the record.
    - Move the object code (represented as hexadecimal characters in the file) into the indicated memory location, converting character representation to actual machine code bytes.
    - Read the next object program record.

1. **Terminate Loop:** Exit the loop when the end record is encountered.
2. **Jump to Execution:** Jump to the starting address specified in the End record to begin program execution.

**Example**

Consider a simple object program with a Header, Text, and End records. All addresses are absolute (pre-determined).

|  |  |  |
| --- | --- | --- |
| **Record Type** | **Content Description** | **Example Object Code** |
| **H** | Program Name, Start Address, Length | H^COPY^1000^1020 |
| **T** | Start Address, Record Length, Object Code | T^1000^0A^140033^4810105C |
| **E** | Execution Start Address | E^1000 |

**Loader's Action:**

1. The loader reads the H record, identifying the program COPY starting at address 1000 with a length of 1020 (hex).
2. It reads the T record:
   * It identifies the starting address for this block as 1000.
   * It takes the object code 140033 and 4810105C (represented as character pairs) and places them directly into memory locations starting from 1000. E.g., the byte 14 (hex) goes to memory address 1000, 00 to 1001, and so on.
3. It reaches the E record.
4. The loader transfers control (jumps) to the address 1000 to start the execution of the program.

The absolute loader is simple because it does not have to manage complex tasks like dynamic memory allocation or resolving external references between different program modules; these are handled manually by the programmer during assembly time. A prime example of an absolute loader is the **bootstrap loader**, which loads the operating system when a computer starts up.

**Unit – 4**

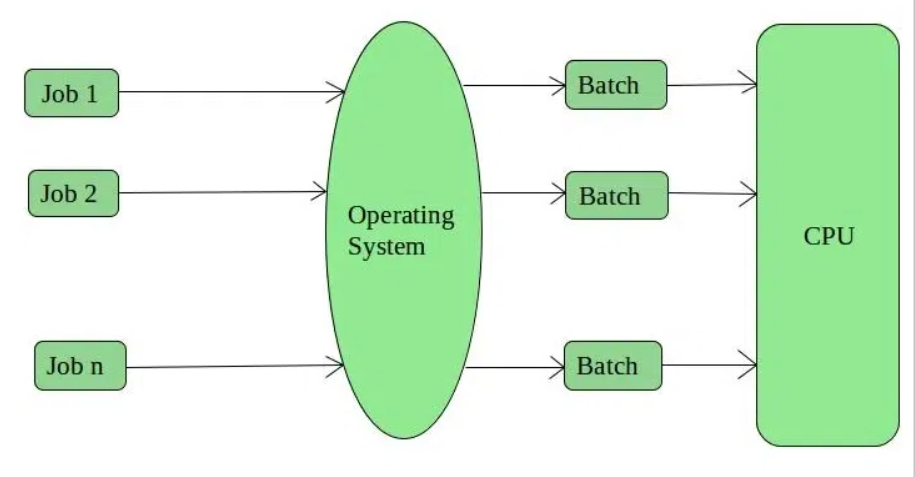
**Q. List different types of Operating Systems? Describe any two of them**

An operating system (OS) is software that manages computer hardware and software resources. It acts as a bridge between users and the computer, ensuring smooth operation.

**Different types of OS serve different needs :**



**1. Batch Operating System**

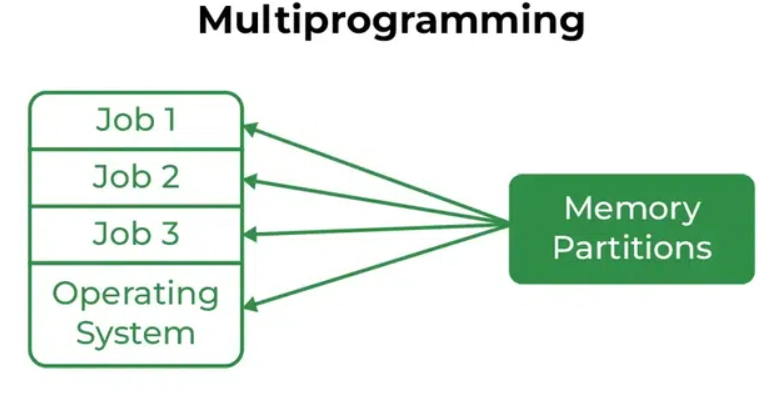
A [Batch Operating System](https://www.geeksforgeeks.org/operating-systems/batch-processing-operating-system/) is designed to handle large groups of similar jobs efficiently. It does not interact with the computer directly but instead processes jobs that are grouped by an operator. These jobs are queued and executed one after the other, without user interaction during the process.

**Advantages of Batch Operating System**

* **Minimal Idle Time**: The system minimizes idle time by processing jobs in a continuous sequence without human intervention.
* **Handling Repetitive Tasks**: Ideal for managing large, repetitive tasks, such as payroll and billing, with minimal effort.

## **2. Multi-Programming Operating System**

In a [Multi-Programming Operating System](https://www.geeksforgeeks.org/operating-systems/multiprogramming-in-operating-system/), multiple programs run in memory at the same time. The CPU switches between programs, utilizing its resources more effectively and improving overall system performance.



**Advantages**

* **Better CPU Utilization:** CPU stays busy by switching to another job during I/O wait.
* **Improved Throughput:** Multiple jobs run concurrently, increasing work done per unit time.
* **Efficient Resource Use:** CPU, memory, and I/O devices are shared effectively among processes.

**Disadvantages**

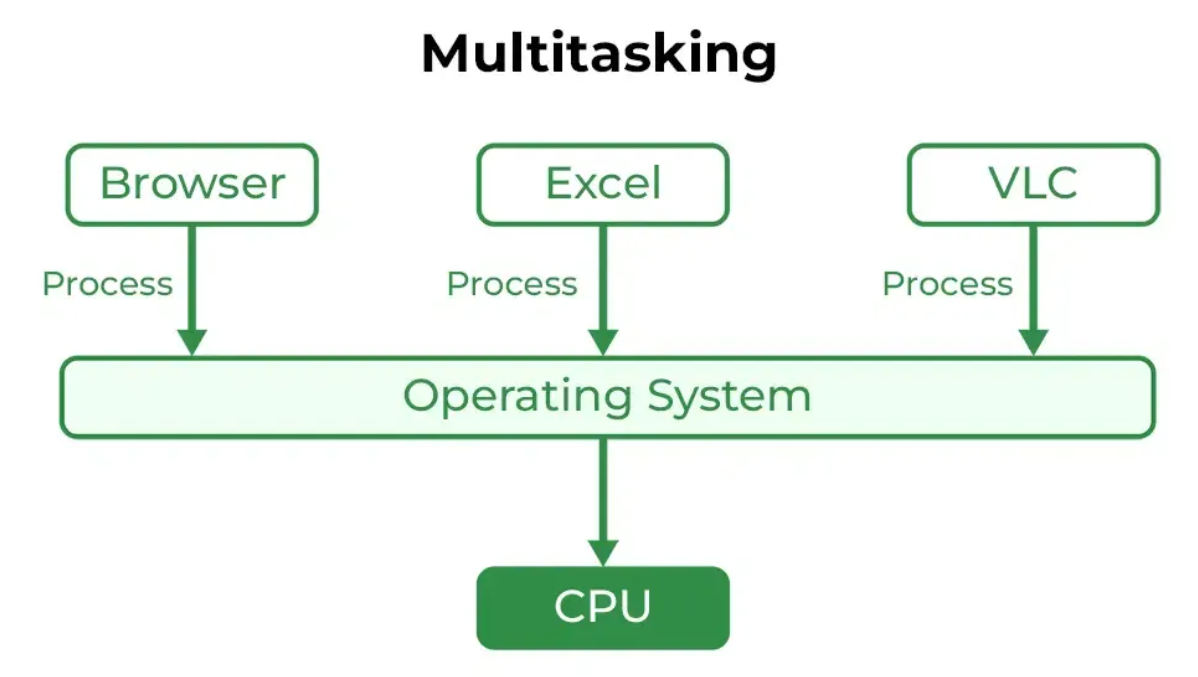
* **Complex Design:** Requires advanced memory management and CPU scheduling.
* **Security Issues:** More programs in memory increase chances of unauthorized access.
* **High Memory Requirement:** Needs larger RAM to run multiple programs together.

Example :

* **Banking systems**
* **Railway servers**

**3. Multi-tasking/Time-sharing Operating systems**

[Multitasking OS](https://www.geeksforgeeks.org/operating-systems/multitasking-operating-system/) is a type of multiprogramming system where each process runs in a **round-robin** manner. Every task gets a fixed time slice called a **quantum**. After the quantum ends, the OS switches to the next task, allowing multiple tasks—whether from one user or many—to run smoothly on a single system.



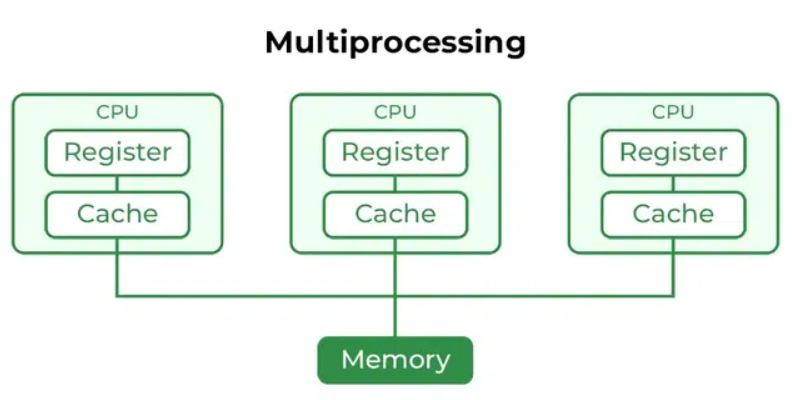
**Advantages**

* **Equal CPU Access:** Each task gets a fair share of CPU time.
* **Reduced Software Duplication:** Many users can run the same software without needing separate copies.
* **Low CPU Idle Time:** Efficient scheduling keeps the CPU busy.

**Disadvantages**

* **Lower Reliability:** System failures affect all users.
* **Security Concerns:** Multiple users increase risks to data integrity and privacy.
* **Communication Issues:** Data sharing between users can cause conflicts.

## **4. Multi-Processing Operating System**

[A Multi-Processing Operating System](https://www.geeksforgeeks.org/operating-systems/difference-between-multiprocessing-and-multiprogramming/) is a type of Operating System in which more than one CPU is used for the execution of resources. It betters the throughput of the System.

**Advantages**

* **Faster Processing:** Multiple CPUs work simultaneously, increasing overall system speed.
* **High Reliability:** If one processor fails, others can continue working (fault tolerance).
* **Supports Heavy Tasks:** Ideal for computation-intensive applications like scientific or industrial tasks.

**Disadvantages**

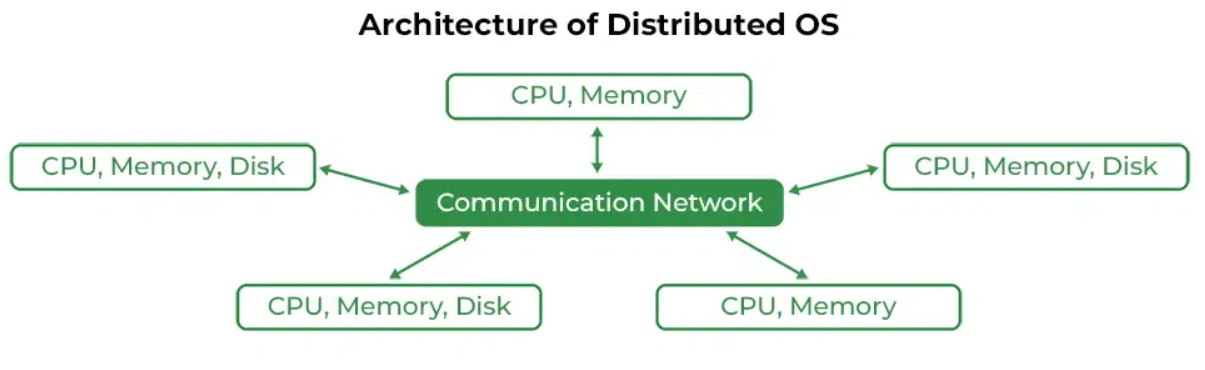
* **High Cost:** Multiple processors and complex hardware increase system cost.
* **Complex Design:** Requires advanced OS support for communication and task distribution.
* **Not Always Efficient:** Poor task distribution can lead to idle processors and wasted resources.

**Example :**

* ***UNIX***
* ***Linux (Ubuntu, Red Hat, Debian)***
* ***macOS***

## **5. Distributed Operating System**

[Distributed operating systems](https://www.geeksforgeeks.org/operating-systems/what-is-a-distributed-operating-system/)connects multiple independent computers through a shared communication network. Each system has its own CPU and memory but works together as a single unit. The main benefit is remote access, allowing users to use files and software stored on other connected systems.



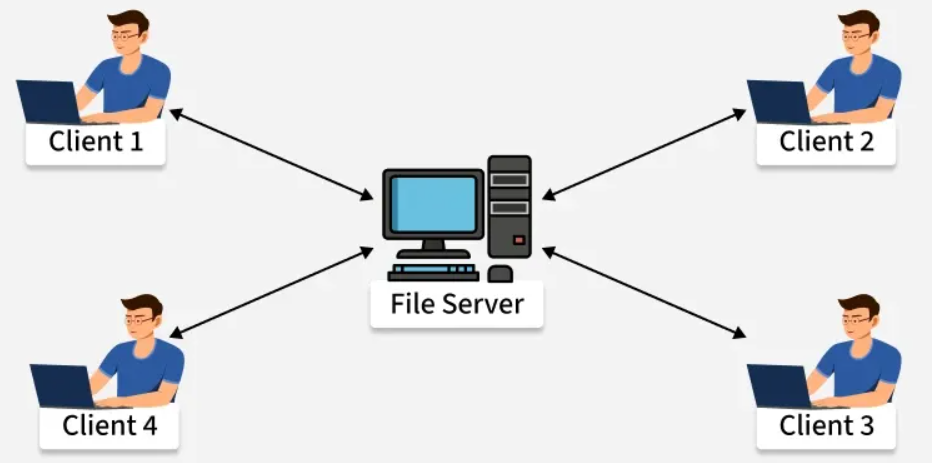
**Advantages**

* **Independent Systems:** Failure of one machine does not affect others.
* **Easily Scalable:** New systems can be added to the network easily.
* **Lower Processing Delays:** Tasks are handled faster across multiple machines.

**Disadvantages**

* **Network Dependency:** If the main network fails, communication stops.
* **Lack of Standardization:** No well-defined language or model for building such systems.
* **High Cost & Complexity:** Hardware is expensive, and the software is complex and not widely understood.

## **6. Network Operating System**

A Network Operating System (NOS) runs on a server and manages data, users, security, applications, and other network functions. It allows shared access to files, printers, and resources within a small private network. Users can see the configuration and connections of other users, which is why these systems are considered [tightly coupled systems](https://www.geeksforgeeks.org/computer-organization-architecture/difference-between-loosely-coupled-and-tightly-coupled-multiprocessor-system/).

**Advantages**

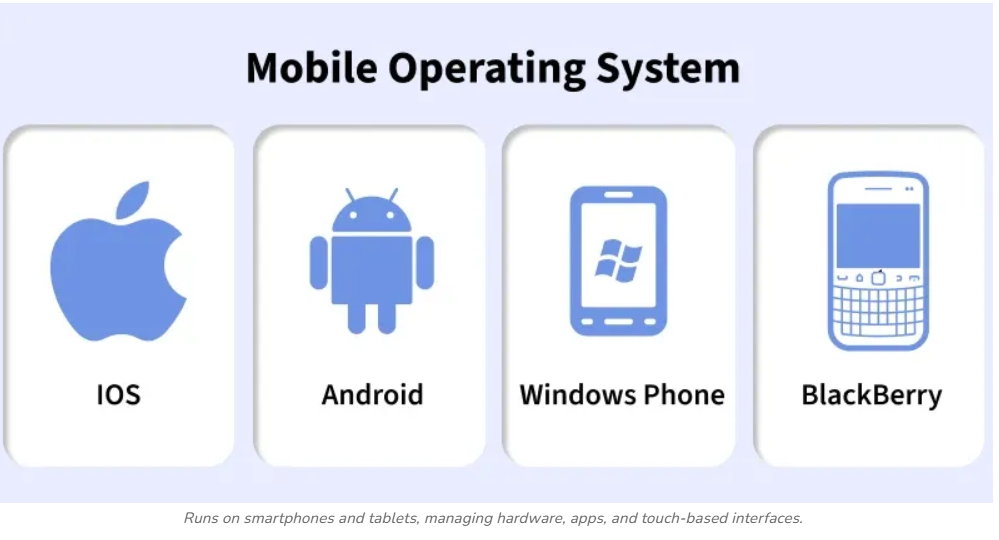
* **Centralized and Stable Servers:** Provide reliable management of resources.
* **Easy Upgrades:** New hardware and technologies can be added without difficulty.
* **Remote Access:** Users can access the server from different locations and devices.

**Disadvantages**

* **High Server Cost:** Setting up and maintaining servers is expensive.
* **Dependency on Server:** Most operations rely on a central server.
* **Regular Maintenance Needed:** Frequent updates and technical support are required.

**8. Mobile Operating Systems**

Mobile operating systems are designed specifically for mobile devices such as smartphones and tablets. Examples of such operating systems are Android and iOS. These operating systems manage the hardware and software resources of the device, providing a platform for running applications and ensuring a seamless user experience.



**Advantages**

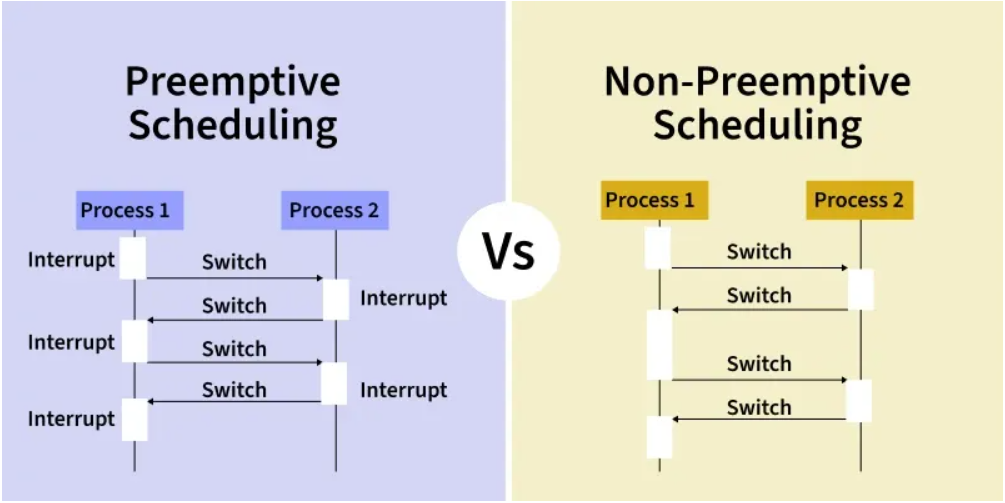
* **User-Friendly Interfaces**: Mobile operating systems are designed to be intuitive and easy to use, making them accessible to a wide range of users.
* **Extensive App Ecosystems**: The availability of a vast number of applications allows users to customize their devices to meet their specific needs.
* **Connectivity Options**: Mobile operating systems support multiple connectivity options, enabling users to stay connected wherever they go.

**Disadvantages**

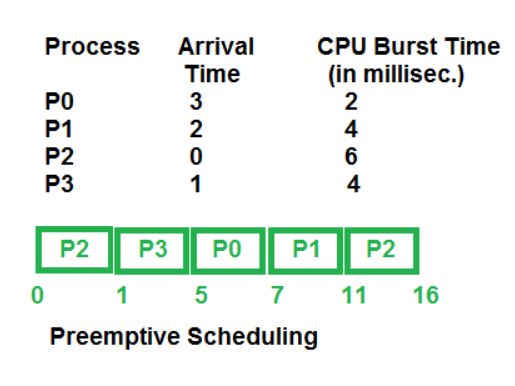
* **Battery Life Constraints**: Despite advancements in power management, battery life remains a challenge for mobile devices, especially with heavy usage.
* **Security Risks**: Mobile devices are susceptible to various security threats, such as malware and phishing attacks, which can compromise user data.
* **Fragmentation**: In the case of Android, the wide range of devices and customizations can lead to fragmentation, making it difficult for developers to ensure compatibility across all devices.
* *Android*
* *iOS*
* *Blackberry*

Q. Differentiate Preemptive and non-preemptive scheduling. Explain one example of each.

| **Preemptive Scheduling** | **Non-Preemptive Scheduling** |
| --- | --- |
| In this resources(CPU Cycle) are allocated to a process for a limited time. | Once resources(CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting state |
| Process can be interrupted in between. | Process can not be interrupted until it terminates itself or its time is up |
| If a process having high priority frequently arrives in the ready queue, a low priority process may starve | If a process with a long burst time is running CPU, then later coming process with less CPU burst time may starve |
| It has overheads of scheduling the processes | It does not have overheads |
| Average process response time is less | Average process response time is high |
| Decisions are made by the scheduler and are based on priority and time slice allocation | Decisions are made by the process itself and the OS just follows the process's instructions |
| More as a process might be preempted when it was accessing a shared resource. | Less as a process is never preempted. |
| Examples of preemptive scheduling are Round Robin and Shortest Remaining Time First | Examples of non-preemptive scheduling are First Come First Serve and Shortest Job First |



**Preemptive Scheduling**

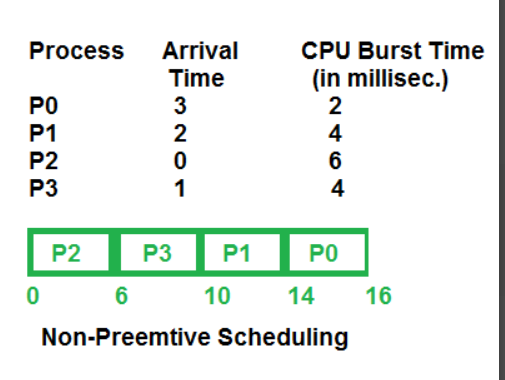
In preemptive scheduling, the operating system can interrupt a running process to allocate the CPU to another process usually due to priority rules or time-sharing policies. A process may be moved from Running → Ready state before it finishes.

**Examples:**

* *Round Robin*
* *Shortest Remaining Time First (SRTF)*
* *Priority Scheduling (preemptive version)*

**Non-Preemptive Scheduling**

In non-preemptive scheduling, once a process starts using the CPU, it runs until it finishes or moves to a waiting state. The OS cannot forcibly take away the CPU.

*Below is the table and Gantt Chart according to the First Come First Serve (FCFS) Algorithm: We can notice that every process finishes execution once it gets CPU.*

**Examples:**

* *First Come First Serve (FCFS)*
* *Shortest Job First (SJF)*
* *Priority Scheduling (non-preemptive version)*

**Q.What is time quantum and its significance in Round robin scheduling**

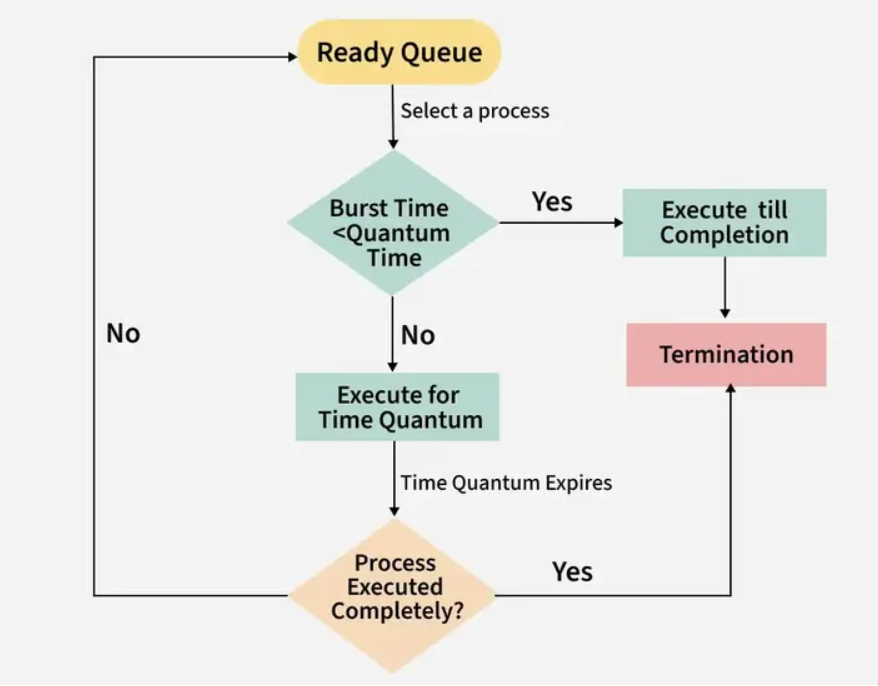
The **time quantum (or time slice)** in Round Robin (RR) scheduling is a small, fixed unit of CPU time given to a process before it's preempted and moved to the back of the ready queue, ensuring fair, cyclic access to the CPU. Its significance lies in balancing responsiveness, fairness, and overhead; a good quantum improves interactivity and prevents starvation, while a poor choice (too big or too small) can degrade performance by causing excessive context switching or mimicking less efficient algorithms like FCFS.

**Significance of Time Quantum in Round Robin**

1. **Fairness & No Starvation**: Ensures every process gets a turn, preventing any single process from monopolizing the CPU and ensuring all processes make progress.
2. **Responsiveness**: A small quantum allows interactive processes to get frequent CPU access, improving system responsiveness, as users quickly see their actions reflected.
3. **Context Switching Overhead**: Directly impacts overhead; a very small quantum leads to frequent context switches, wasting CPU cycles, while a very large quantum reduces this overhead but hurts responsiveness.
4. **Performance Trade-off**:
   * **Large Quantum**: Behaves like First-Come, First-Served (FCFS), potentially causing long response times for short jobs.
   * **Small Quantum**: Increases context switching overhead, reducing overall throughput.
5. **System Efficiency**: The choice of time quantum is crucial for balancing user experience (responsiveness) and system efficiency (throughput), often requiring optimization for specific workloads.

In essence, the time quantum defines the "slice" of time each process gets, making Round Robin a preemptive, time-sharing algorithm that prioritizes fairness and quick interactions.

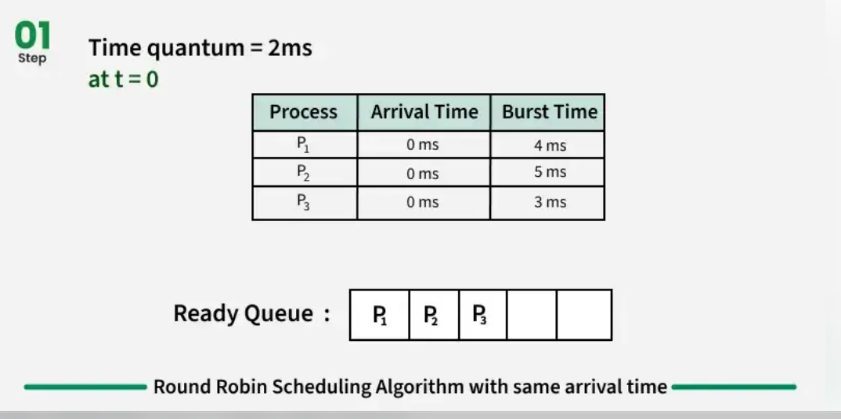
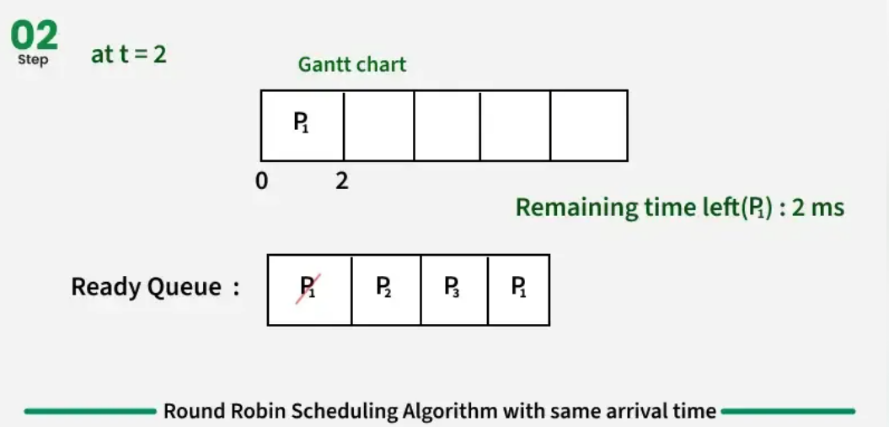
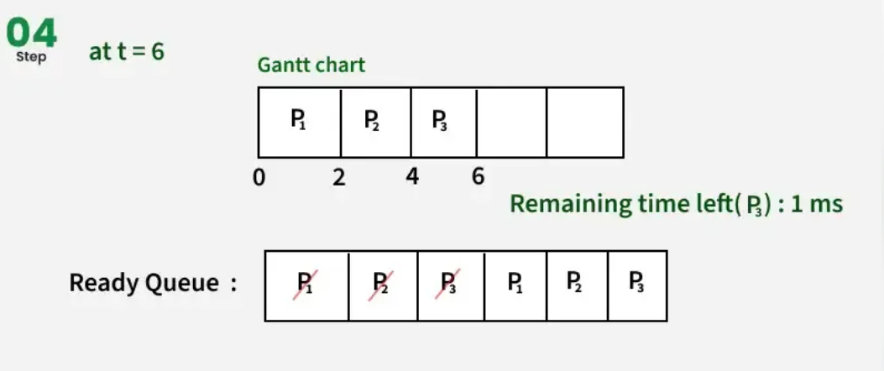
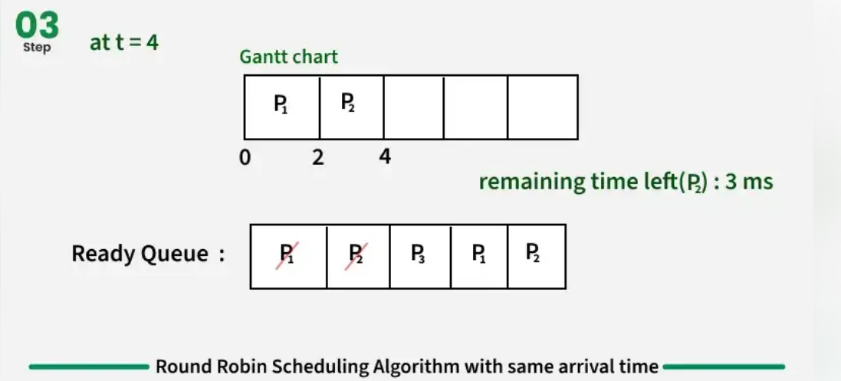
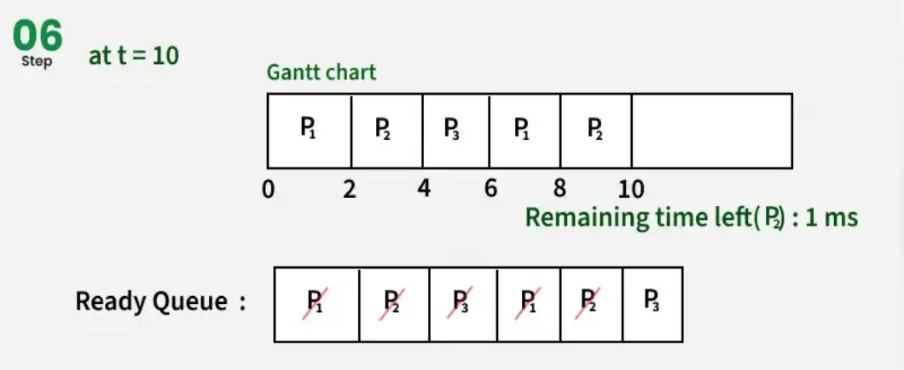
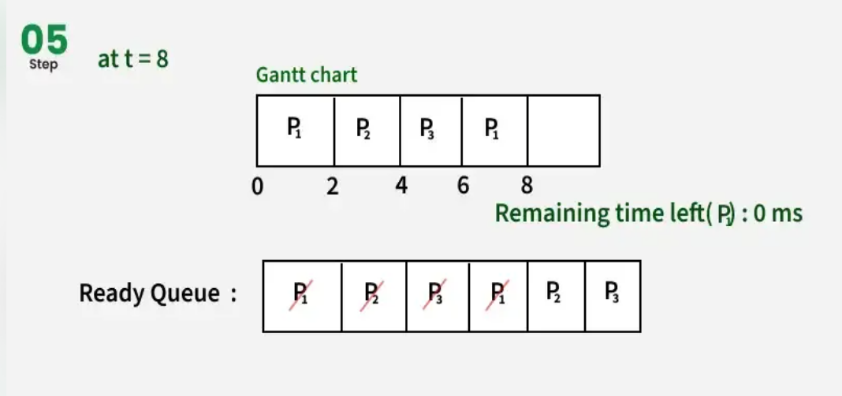
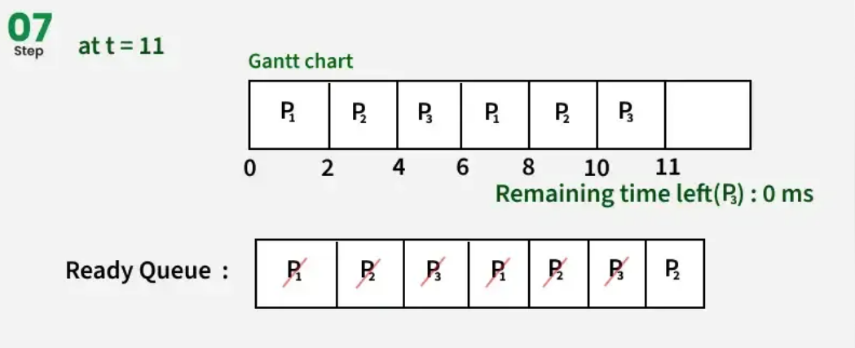
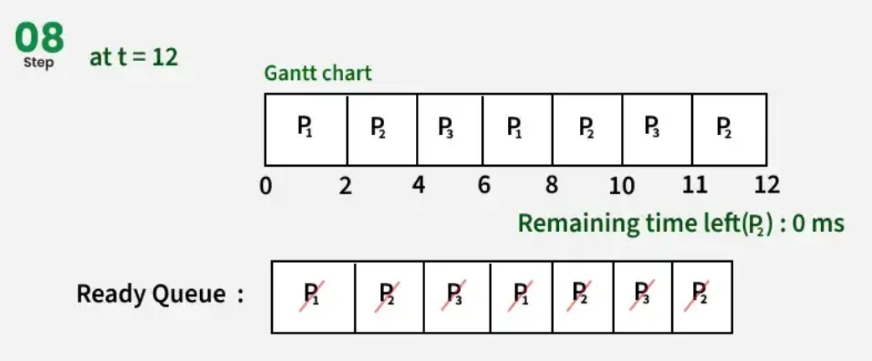
The primary goal of this scheduling method is to ensure that all processes are given an equal opportunity to execute, promoting fairness among tasks.

* **Process Arrival:** Processes enter the system and are placed in a queue.
* **Time Allocation:** Each process is given a certain amount of CPU time, called a quantum.
* **Execution:** The process uses the CPU for the allocated time.
* **Rotation:** If the process completes within the time, it leaves the system. If not, it goes back to the end of the queue.
* **Repeat:** The CPU continues to cycle through the queue until all processes are completed.

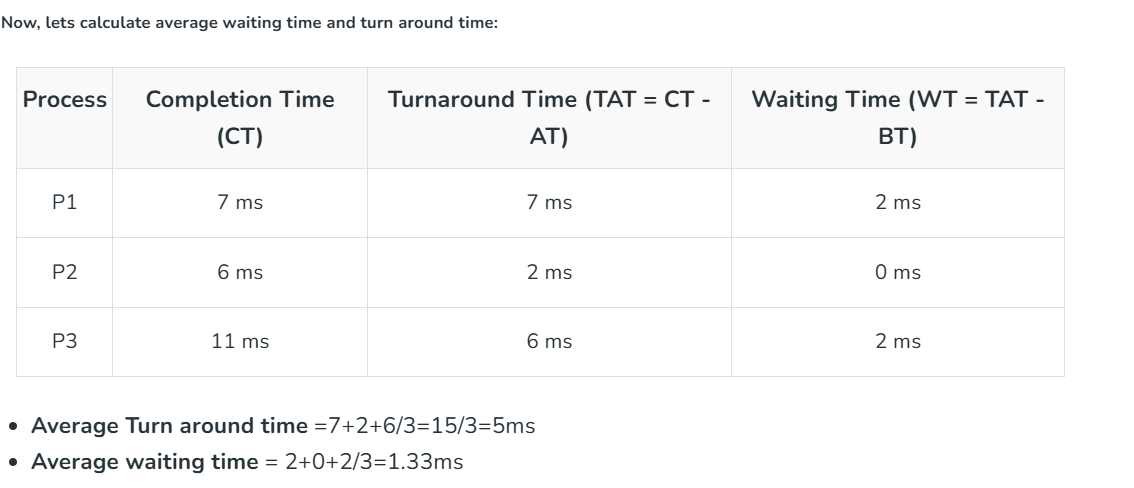
**Example:**

To understand the Round Robin Scheduling algorithm, let’s consider the following two scenarios:

**Scenario 1: Processes with Same Arrival Time**

Consider the following table of arrival time and burst time for three processes P1, P2 and P3 and given **Time Quantum = 2 ms**

| **Process** | **Burst Time** | **Arrival Time** |
| --- | --- | --- |
| P1 | 4 ms | 0 ms |
| P2 | 5 ms | 0 ms |
| P3 | 3 ms | 0 ms |



**Q. Explain multithreaded model and Process Control block in detail.**

Multithreaded mode is an execution model allowing multiple streams of instructions (threads) to run concurrently within a single process, improving efficiency and responsiveness.

A **Process Control Block (PCB)** is a data structure used by the operating system to store all the essential information needed to manage a process.

**Multithreaded Mode**

Multithreading is a technique that enables a single process to have multiple threads of control, each representing a separate path of execution.

* **Resource Sharing:** All threads within the same process share the same memory space, code section, data section, and operating system resources (like open files and I/O devices). This sharing makes thread creation and context switching more lightweight and faster than managing separate processes.
* **Independent Contexts:** Although they share most resources, each thread maintains its own unique execution context, stored in a **Thread Control Block (TCB)**. This includes:
  + A unique Thread ID.
  + A Program Counter (PC) to track the next instruction to execute.
  + Its own set of CPU registers.
  + Its own stack for function calls and local variables.
  + A pointer to the parent Process Control Block (PCB).
* **Benefits:**
  + **Responsiveness:** If one thread is blocked (e.g., waiting for I/O), other threads in the same process can continue running, ensuring the application remains responsive.
  + **Efficient CPU Utilization:** Allows programs to take advantage of multi-core processors by running threads in true parallel on different cores.
  + **Economy:** Threads are more cost-effective to create and manage than processes, as they require fewer resources.

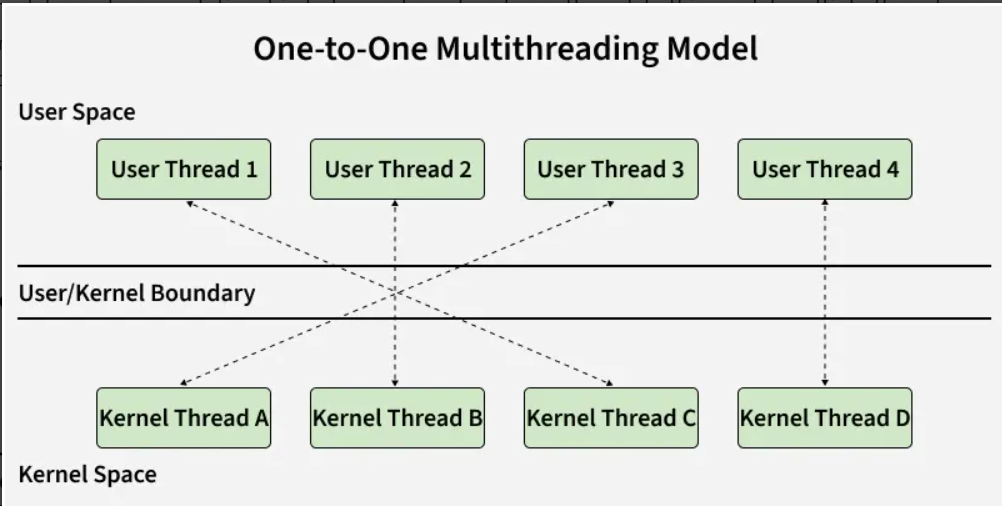
## Mapping Models of Threads

### Many-to-Many Model:

* Multiple user threads map to multiple kernel threads.
* If one thread blocks, others can continue.
* Provides high concurrency and is the most efficient model.

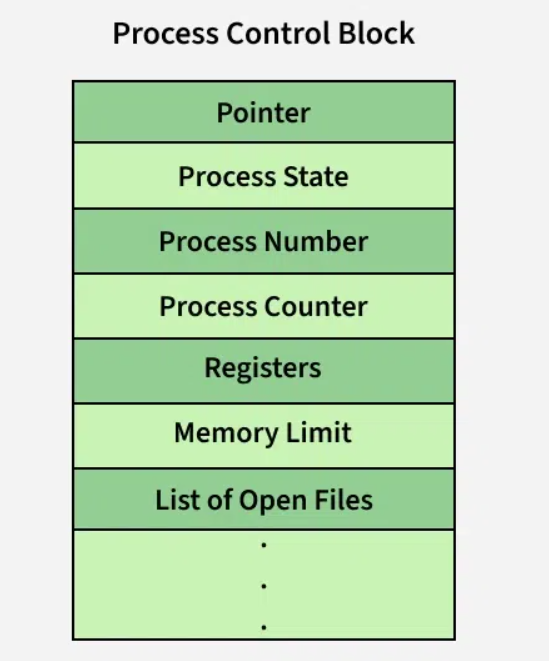
### Many-to-One Model:

* Multiple user threads map to a single kernel thread.
* Blocking in one thread blocks the entire process.
* Efficient user-level management but poor multiprocessing utilization.

1. **One-to-One Model:**

* Each user thread maps to a unique kernel thread.
* Multiple threads can run on multiple processors.
* Blocking in one thread does not affect others.
* Overhead is higher because each user thread requires a corresponding kernel thread.

**Process Control Block (PCB)**

The Process Control Block (PCB), also known as a task control block or process descriptor, is a critical data structure used by the operating system to store all relevant information about a process. Each process has exactly one PCB, and the operating system maintains a process table containing all active PCBs.

The PCB stores the "context" of a process, allowing the operating system to stop a running process and resume it later from the exact point it left off (context switching). The specific information stored in a PCB includes:

* **Pointer**: It is a stack pointer that is required to be saved when the process is switched from one state to another to retain the current position of the process**.**
* **Process state:**It stores the respective state of the process.
* **Process number:**Every process is assigned a unique id known as process ID or PIDwhich stores the process identifier.
* **Program counter:**[Program Counter](https://www.geeksforgeeks.org/operating-systems/what-is-program-counter/)stores the counter, which contains the address of the next instruction that is to be executed for the process.
* **Register:**[Registers](https://www.geeksforgeeks.org/computer-organization-architecture/different-classes-of-cpu-registers/)in the PCB, it is a data structure. When a processes is running and it's time slice expires, the current value of process specific registers would be stored in the PCB and the process would be swapped out. When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the registers in the PCB.
* **Memory limits:**This field contains the information about [memory management system](https://www.geeksforgeeks.org/operating-systems/memory-management-in-operating-system/)used by the operating system. This may include page tables, segment tables, etc.
* **List of Open files:**This information includes the list of files opened for a process.

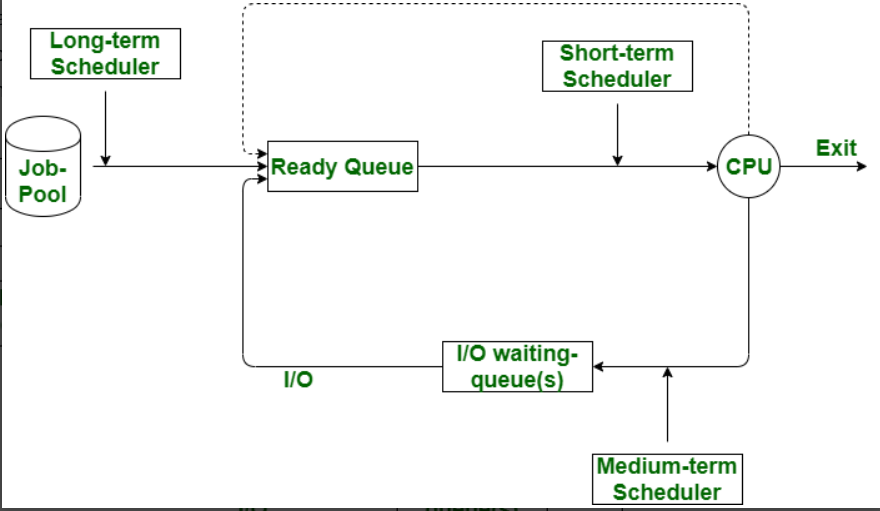
***“ The Process Control Block (PCB) is stored in a special part of memory that normal users can't access. This is because it holds important information about the process. “***

**Q.** **Explain the following types of Schedulers.**

**i) Short Term**

**ii) Long Term**

**iii) Medium Term**

Operating systems use three types of schedulers to manage the execution of processes.

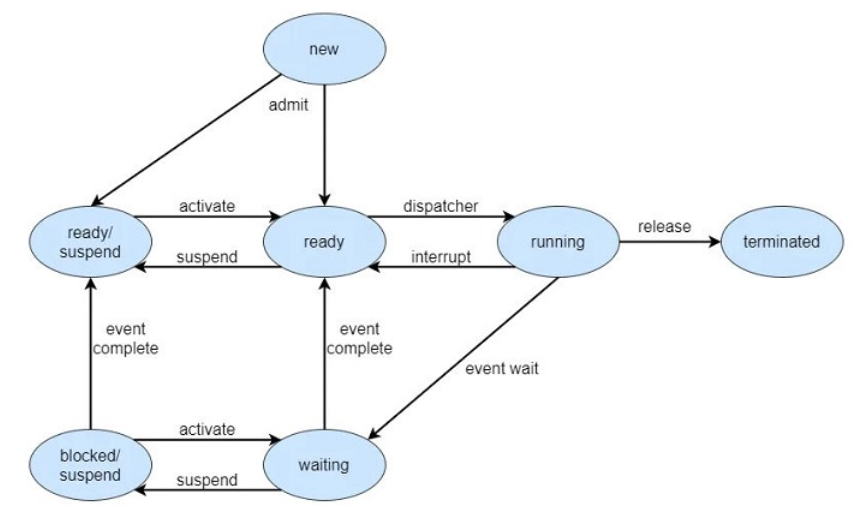
* **Long-Term Scheduler (Job Scheduler):**
  + **Function:** Decides which processes (jobs) from the job pool (secondary memory) are to be loaded into main memory (RAM) for execution.
  + **Frequency:** Invoked infrequently, typically when a current process terminates or a new job arrives.
  + **Control:** Controls the **degree of multiprogramming** (the number of processes in memory).
  + **Goal:** To select a balanced mix of I/O-bound and CPU-bound processes to ensure optimal system performance and prevent the CPU from being consistently idle or overworked.
* **Medium-Term Scheduler (Swapper):**
  + **Function:** Handles **swapping**, which involves temporarily removing processes from main memory to secondary storage (suspending) and later bringing them back into main memory (resuming).
  + **Frequency:** Invoked with moderate frequency.
  + **Control:** Dynamically reduces the degree of multiprogramming to manage memory usage and improve the process mix.
  + **Goal:** To manage memory overcommitment, reduce thrashing, and improve overall system throughput and utilization.
* **Short-Term Scheduler (CPU Scheduler):**
  + **Function:** Selects which process among the ones ready in main memory will be assigned to the CPU for execution next.
  + **Frequency:** Invoked very frequently (milliseconds), often after events like clock interrupts or I/O completion, so it must be very fast and efficient.
  + **Control:** Has a direct, short-term effect on system performance, determining the immediate allocation of the CPU.
  + **Goal:** To maximize CPU utilization and performance metrics like response time, turnaround time, and waiting time using various scheduling algorithms (e.g., FCFS, SJF, Round Robin).

**Difference Between Short-Term,  Medium Term, And Long-Term Schedulers**

| **Basis** | **Short-Term Scheduler** | **Medium-term Scheduler** | **Long-Term Scheduler** |
| --- | --- | --- | --- |
| 1. Alternate Name | It is also called a CPU scheduler. | It is also called a process swapping scheduler. | It is also called a job scheduler. |
| 2. Degree in programming | It provides lesser control over the degree of multiprogramming. | It reduces the control over the degree of [multiprogramming.](https://www.geeksforgeeks.org/operating-systems/multiprogramming-in-operating-system/) | It controls the degree of multiprogramming. |
| 3. Speed | The speed of the short-term scheduler is very fast. | Speed of medium scheduler between the short-term and long-term scheduler | The speed of a long-term term scheduler is more than medium-term scheduler. |
| 4. Usage in time- sharing system sharing system | It is minimal in the time-sharing system. | It is a part of the [time-sharing system.](https://www.geeksforgeeks.org/operating-systems/time-sharing-operating-system/) | It is almost absent or minimal in a sharing system. |
| 5. Purpose | It selects the processes from among the process that is ready to execute. | It can reintroduce the from among the process into memory that executes and its execution can be continued. | It selects processes from the pool and loads them into memory for execution. |
| 6. Process state | Process state is ready to running | Process state is not present | Process state is new to ready. |
| 7. Selection of process | Select a new process for a CPU quite frequently. | Select that process, which is currently not need to load fully on RAM, so it swap it into swap partition. | Select a good process , mix of I/O bound and CPU bound. |

**Q. Explain seven state process model with diagram? Also explain difference between Five state process model & Seven state process model?**

The 7-State Process Model adds two crucial memory management states, [**Ready Suspend**](https://www.google.com/search?q=Ready+Suspend&sca_esv=cf3d8fcadb69d60c&lns_surface=44&hl=en-US&cs=1&biw=1536&bih=730&sxsrf=AE3TifO90PS6I5xImANMY4hrOiJFZiDsnw%3A1765918117914&ei=pcVBab7IN96nseMPqZ718QQ&ved=2ahUKEwj1kY2j_8KRAxUUS2wGHTtNCEkQgK4QegQIAhAC&uact=5&oq=Q.+Explain+seven+state+process+model+with+diagram%3F+Also+explain+difference+between+Five+state+process+model+%26+Seven+state+process+model%3F&gs_lp=Egxnd3Mtd2l6LXNlcnAiiAFRLiBFeHBsYWluIHNldmVuIHN0YXRlIHByb2Nlc3MgbW9kZWwgd2l0aCBkaWFncmFtPyBBbHNvIGV4cGxhaW4gZGlmZmVyZW5jZSBiZXR3ZWVuIEZpdmUgc3RhdGUgcHJvY2VzcyBtb2RlbCAmIFNldmVuIHN0YXRlIHByb2Nlc3MgbW9kZWw_MgcQIxgnGOoCMgcQIxgnGOoCMg0QIxjwBRgnGMkCGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMhcQABiABBiRAhi0AhjnBhiKBRjqAtgBATIXEAAYgAQYkQIYtAIY5wYYigUY6gLYAQEyFxAAGIAEGJECGLQCGOcGGIoFGOoC2AEBMhcQABiABBiRAhi0AhjnBhiKBRjqAtgBATIXEAAYgAQYkQIYtAIY5wYYigUY6gLYAQEyFxAAGIAEGJECGLQCGOcGGIoFGOoC2AEBSJoXUNYLWNYLcAF4AZABAJgBAKABAKoBALgBA8gBAPgBAfgBApgCAaACDqgCEJgDDvEFXXS8Jpt8Apa6BgYIARABGAGSBwExoAcAsgcAuAcAwgcDMy0xyAcKgAgA&sclient=gws-wiz-serp) and [**Blocked Suspend**](https://www.google.com/search?q=Blocked+Suspend&sca_esv=cf3d8fcadb69d60c&lns_surface=44&hl=en-US&cs=1&biw=1536&bih=730&sxsrf=AE3TifO90PS6I5xImANMY4hrOiJFZiDsnw%3A1765918117914&ei=pcVBab7IN96nseMPqZ718QQ&ved=2ahUKEwj1kY2j_8KRAxUUS2wGHTtNCEkQgK4QegQIAhAD&uact=5&oq=Q.+Explain+seven+state+process+model+with+diagram%3F+Also+explain+difference+between+Five+state+process+model+%26+Seven+state+process+model%3F&gs_lp=Egxnd3Mtd2l6LXNlcnAiiAFRLiBFeHBsYWluIHNldmVuIHN0YXRlIHByb2Nlc3MgbW9kZWwgd2l0aCBkaWFncmFtPyBBbHNvIGV4cGxhaW4gZGlmZmVyZW5jZSBiZXR3ZWVuIEZpdmUgc3RhdGUgcHJvY2VzcyBtb2RlbCAmIFNldmVuIHN0YXRlIHByb2Nlc3MgbW9kZWw_MgcQIxgnGOoCMgcQIxgnGOoCMg0QIxjwBRgnGMkCGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMgcQIxgnGOoCMhcQABiABBiRAhi0AhjnBhiKBRjqAtgBATIXEAAYgAQYkQIYtAIY5wYYigUY6gLYAQEyFxAAGIAEGJECGLQCGOcGGIoFGOoC2AEBMhcQABiABBiRAhi0AhjnBhiKBRjqAtgBATIXEAAYgAQYkQIYtAIY5wYYigUY6gLYAQEyFxAAGIAEGJECGLQCGOcGGIoFGOoC2AEBSJoXUNYLWNYLcAF4AZABAJgBAKABAKoBALgBA8gBAPgBAfgBApgCAaACDqgCEJgDDvEFXXS8Jpt8Apa6BgYIARABGAGSBwExoAcAsgcAuAcAwgcDMy0xyAcKgAgA&sclient=gws-wiz-serp), to the 5-state model (New, Ready, Running, Blocked, Exit), allowing swapping processes to disk, providing finer control over main memory usage by distinguishing between in-memory and swapped-out processes for better multitasking and efficiency, unlike the simpler 5-state model that treats all non-running processes similarly.

**7-State Process Model**

**States Explained:**

1. **New:** Process creation phase; in secondary memory.
2. **Ready (In-Memory):** In main memory, ready for CPU.
3. **Running (In-Memory):** Currently executing on CPU.
4. **Blocked/Waiting (In-Memory):** Waiting for an event (I/O, resource) in main memory.
5. **Exit/Terminated:** Process finished execution.
6. **Ready Suspend:** Ready process swapped out to disk (due to memory pressure).
7. **Blocked Suspend:** Blocked process swapped out to disk (waiting for event + memory).

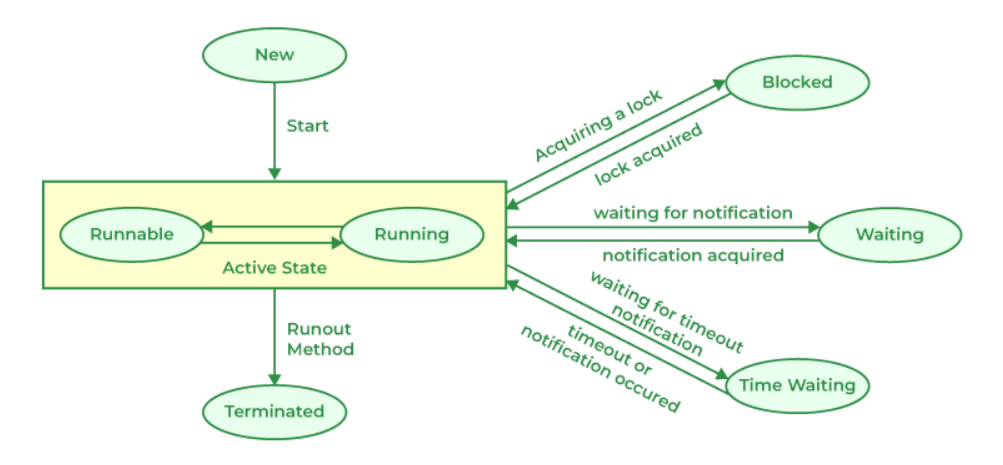
**5-State vs. 7-State Model**

|  |  |  |
| --- | --- | --- |
| **Feature** | **5-State Model** | **7-State Model** |
| **States** | New, Ready, Running, Blocked/Waiting, Exit/Terminate. | Adds **Ready Suspend** & **Blocked Suspend** to the 5 states. |
| **Memory Mgmt** | Less granular; doesn't differentiate swapped-out processes from in-memory blocked ones. | More granular; distinguishes between processes in main memory and those swapped to disk. |
| **Suspension** | Treats all waiting processes (in-memory or not) similarly. | Explicitly handles processes suspended in secondary storage (disk). |
| **Efficiency** | Lower efficiency in memory management for multiprogramming. | Higher efficiency, better resource (memory) utilization via swapping. |

**Key Difference**

The fundamental difference is the **introduction of suspended states (Ready Suspend, Blocked Suspend)** in the 7-state model, which allows the Operating System to manage memory better by swapping processes out to disk when main memory is low, improving overall system performance and multitasking, a capability lacking in the simpler 5-state model.

**Q. What is mean by Threads, Explain Thread lifecycle with diagram in detail?**

A **thread** is a lightweight, independent path of execution within a process, allowing a program to perform multiple tasks concurrently, like multiple chefs in a kitchen. The thread lifecycle describes its journey through states: [**New**](https://www.google.com/search?sca_esv=cf3d8fcadb69d60c&lns_surface=44&hl=en-US&cs=1&sxsrf=AE3TifNL96l0oNSFkVyZot8I5JgWGhPO4Q%3A1765918847679&q=New&source=lnms&fbs=AIIjpHxU7SXXniUZfeShr2fp4giZ1Y6MJ25_tmWITc7uy4KIesp4JNzLpfTaPhMa2AVaJKqFtcfmRTQLRkZioTgjDHIUS_a7o6TghI7UW0M4rb_D91Hsk9mcFHq1Yehnm3pfhqwVnRQjZMpIgtY7UoKymnOiFdwOudUDjDACqQX5gNqWOt6I9MvD9l4AMR00XbxC_-ZJNSQ9iJCsaD6tC-6_zRu1afQRnnrkz7XV3QA9YvgtQQoDNKk&sa=X&ved=2ahUKEwjD_7iwgMORAxWfTGwGHeHIIaUQgK4QegYIAAgAEBE&biw=1536&bih=730&dpr=1.25) (created, not started), [**Runnable**](https://www.google.com/search?sca_esv=cf3d8fcadb69d60c&lns_surface=44&hl=en-US&cs=1&sxsrf=AE3TifNL96l0oNSFkVyZot8I5JgWGhPO4Q%3A1765918847679&q=Runnable&source=lnms&fbs=AIIjpHxU7SXXniUZfeShr2fp4giZ1Y6MJ25_tmWITc7uy4KIesp4JNzLpfTaPhMa2AVaJKqFtcfmRTQLRkZioTgjDHIUS_a7o6TghI7UW0M4rb_D91Hsk9mcFHq1Yehnm3pfhqwVnRQjZMpIgtY7UoKymnOiFdwOudUDjDACqQX5gNqWOt6I9MvD9l4AMR00XbxC_-ZJNSQ9iJCsaD6tC-6_zRu1afQRnnrkz7XV3QA9YvgtQQoDNKk&sa=X&ved=2ahUKEwjD_7iwgMORAxWfTGwGHeHIIaUQgK4QegYIAAgAEBI&biw=1536&bih=730&dpr=1.25) (ready for CPU), [**Running**](https://www.google.com/search?sca_esv=cf3d8fcadb69d60c&lns_surface=44&hl=en-US&cs=1&sxsrf=AE3TifNL96l0oNSFkVyZot8I5JgWGhPO4Q%3A1765918847679&q=Running&source=lnms&fbs=AIIjpHxU7SXXniUZfeShr2fp4giZ1Y6MJ25_tmWITc7uy4KIesp4JNzLpfTaPhMa2AVaJKqFtcfmRTQLRkZioTgjDHIUS_a7o6TghI7UW0M4rb_D91Hsk9mcFHq1Yehnm3pfhqwVnRQjZMpIgtY7UoKymnOiFdwOudUDjDACqQX5gNqWOt6I9MvD9l4AMR00XbxC_-ZJNSQ9iJCsaD6tC-6_zRu1afQRnnrkz7XV3QA9YvgtQQoDNKk&sa=X&ved=2ahUKEwjD_7iwgMORAxWfTGwGHeHIIaUQgK4QegYIAAgAEBM&biw=1536&bih=730&dpr=1.25) (executing), [**Blocked/Waiting**](https://www.google.com/search?sca_esv=cf3d8fcadb69d60c&lns_surface=44&hl=en-US&cs=1&sxsrf=AE3TifNL96l0oNSFkVyZot8I5JgWGhPO4Q%3A1765918847679&q=Blocked%2FWaiting&source=lnms&fbs=AIIjpHxU7SXXniUZfeShr2fp4giZ1Y6MJ25_tmWITc7uy4KIesp4JNzLpfTaPhMa2AVaJKqFtcfmRTQLRkZioTgjDHIUS_a7o6TghI7UW0M4rb_D91Hsk9mcFHq1Yehnm3pfhqwVnRQjZMpIgtY7UoKymnOiFdwOudUDjDACqQX5gNqWOt6I9MvD9l4AMR00XbxC_-ZJNSQ9iJCsaD6tC-6_zRu1afQRnnrkz7XV3QA9YvgtQQoDNKk&sa=X&ved=2ahUKEwjD_7iwgMORAxWfTGwGHeHIIaUQgK4QegYIAAgAEBQ&biw=1536&bih=730&dpr=1.25) (paused for I/O, sleep, or lock), and [**Terminated**](https://www.google.com/search?sca_esv=cf3d8fcadb69d60c&lns_surface=44&hl=en-US&cs=1&sxsrf=AE3TifNL96l0oNSFkVyZot8I5JgWGhPO4Q%3A1765918847679&q=Terminated&source=lnms&fbs=AIIjpHxU7SXXniUZfeShr2fp4giZ1Y6MJ25_tmWITc7uy4KIesp4JNzLpfTaPhMa2AVaJKqFtcfmRTQLRkZioTgjDHIUS_a7o6TghI7UW0M4rb_D91Hsk9mcFHq1Yehnm3pfhqwVnRQjZMpIgtY7UoKymnOiFdwOudUDjDACqQX5gNqWOt6I9MvD9l4AMR00XbxC_-ZJNSQ9iJCsaD6tC-6_zRu1afQRnnrkz7XV3QA9YvgtQQoDNKk&sa=X&ved=2ahUKEwjD_7iwgMORAxWfTGwGHeHIIaUQgK4QegYIAAgAEBU&biw=1536&bih=730&dpr=1.25) (finished). These transitions, managed by a thread scheduler, are vital for efficient multitasking.

1. **New (Born)**: A Thread object is created, but start() hasn't been called. The thread is alive but not yet active.
2. **Runnable (Ready)**: After start() is called, the thread moves to the runnable state, ready to be scheduled for CPU time.
3. **Running**: The Thread Scheduler picks the thread, and the CPU executes its run() method.
4. **Blocked/Waiting/Timed Waiting**: The thread pauses due to:
   * **I/O Operations**: Waiting for input/output.
   * **sleep()**: Pauses for a specific duration (Timed Waiting).
   * **wait()/join()**: Waiting for another thread or resource lock (Waiting).
5. **Terminated (Dead)**: The thread finishes execution (its run() method exits) or is stopped, ending its life.

**Key Transitions**

* **New -> Runnable**: Calling thread.start().
* **Runnable -> Running**: CPU scheduling.
* **Running -> Blocked/Waiting**: sleep(), wait(), I/O.
* **Blocked/Waiting -> Runnable**: Sleep ends, I/O completes, notify().
* **Running/Runnable -> Terminated**: run() method finishes.

**Q.Draw Gantt chart and calculate Avg. turnaround time, Avg. Waiting time for the following processes using SJF non preemptive and round robin with time quantum 0.5 Unit**