A picture containing logo

Description automatically generated

**Operation System S**

**40201341**

**(R/615/1700)**

**Section (5)**

**Submitted To:**

Dr. Malek Louzi

**Submitted By:**

Farouq Hassan

Fall 2023

**Table Of Contents**

* 1. Task 1 ……………………………………………. (3)

2.1. Task 2 …………………….………………………… (7)

3.1. Task 3 …………………………….…………………….……… (9)

4.0. Task 4 …………………….…………………………………… (10)

4.1. …………………….…………………….………………………… ()

4.2. …………………….………………………………………………… ()

4.3. …………………………………………….………………………… ()

5.1. Task 5 …………………….………………………… (12)

5.1. …………………….………………………………………………… (12)

5.2. …………………………………………….………………………… (14)

10.1. Resources………………………….……………….…... (16)

**------------------------------------------------------------------------------**

**Task 1**

**The Round Robin**

scheduling algorithm, the metrics are as follows:

Completion times for each process:

P1: 20 milliseconds

P2: 22 milliseconds

P3: 28 milliseconds

P4: 16 milliseconds

Turnaround times for each process (Completion Time - Arrival Time):

P1: 20 milliseconds

P2: 21 milliseconds

P3: 26 milliseconds

P4: 13 milliseconds

Waiting times for each process (Turnaround Time - Burst Time):

P1: 12 milliseconds (20 - 8)

P2: 15 milliseconds (21 - 6)

P3: 16 milliseconds (26 - 10)

P4: 9 milliseconds (13 - 4)

Average turnaround time:

20 + 21 + 24 + 13 / 4 = 19.5 milliseconds

Average waiting time:

12 + 15 + 14 + 9 / 4 = 12.5 milliseconds

Ready queue

P1 > P2 > P3 > P4 > P1 > P2 > P3 > P3

Running queue

(0) P1 > (4) P2 > (8) P3 > (12) P4 > (16) P1 > (20) P2 > (22) P3 > (26) P3 (28)

**Priority Scheduling (preemptive)**

scheduling algorithm, the metrics are as follows:

Completion times for each process:

P1: 13 milliseconds

P2: 29 milliseconds

P3: 23 milliseconds

P4: 7 milliseconds

Turnaround times for each process:

P1: 13 milliseconds (Completion Time - Arrival Time)

P2: 28 milliseconds

P3: 21 milliseconds

P4: 4 milliseconds

Waiting times for each process (Turnaround Time - Burst Time):

P1: 5 milliseconds (13 - 8)

P2: 22 milliseconds (28 - 6)

P3: 11 milliseconds (21 - 10)

P4: 0 milliseconds (4 - 4)

Average turnaround time:

16 + 19 + 24 + 9 / 4 = 16.5 milliseconds

Average waiting time:

5 + 22 + 11 + 0 /4 =9.5 milliseconds

(0) P1 > (1) P1 > (2) P1 > (3) P4 > (7) P1 > (13) P3 > (23) P2 (29)

**Shortest Job First (SJF) Non-Preemptive**

scheduling are as follows:

The Completion Times for each process using.

P1: 8 milliseconds

P2: 18 milliseconds

P3: 28 milliseconds

P4: 12 milliseconds

The Turnaround Time for each process using Shortest Job First (SJF) Non-Preemptive scheduling is as follows:

P1: 8 milliseconds

P2: 17 milliseconds

P3: 26 milliseconds

P4: 9 milliseconds

The Waiting Time for each process using Shortest Job First (SJF) Non-Preemptive scheduling is:

P1: 0 milliseconds

P2: 11 milliseconds

P3: 16 milliseconds

P4: 5 milliseconds

The average metrics for Shortest Job First (SJF) Non-Preemptive scheduling are:

Average Turnaround Time:

8 + 17 + 26 + 9 / 4 = 15 milliseconds

Average Waiting Time:

0 + 11 + 16 + 5 / 4 = 8 milliseconds

(0) P1 > (8) P4 > (12) P2 > (18) P3 (28)

**Shortest Job First (SJF) Preemptive**

scheduling are as follows:

The Completion Times for each process using.

P1: 8 milliseconds

P2: 11 milliseconds

P3: 26 milliseconds

P4: 4 milliseconds

The Turnaround Time for each process using Shortest Job First (SJF) Preemptive scheduling is as follows:

P1: 18 milliseconds

P2: 10 milliseconds

P3: 26 milliseconds

P4: 4 milliseconds

The Waiting Time for each process using Shortest Job First (SJF) Preemptive scheduling is:

P1: 10 milliseconds

P2: 4 milliseconds

P3: 16 milliseconds

P4: 0 milliseconds

The average metrics for Shortest Job First (SJF) Preemptive scheduling are:

Average Turnaround Time:

18 + 10 + 26 + 4 / 4 = 14.5 milliseconds

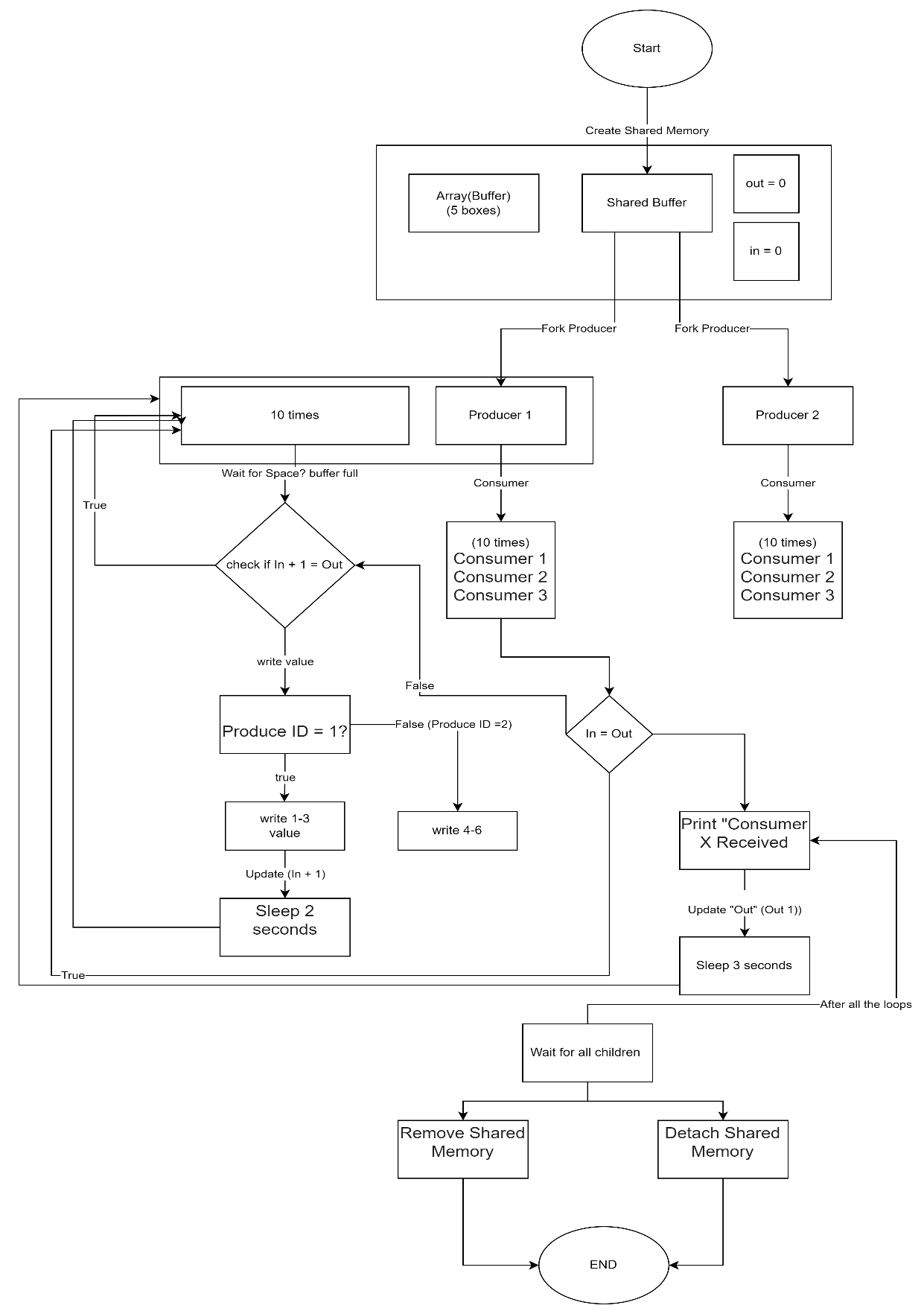
Average Waiting Time:

10 + 4 + 16 + 0 / 4 = 7.5 milliseconds

(0) P1 > (1) P2 > (3) P4 > (7) P2 > (11) P1 > (18 )P3 (28)

**Task 2**

1. **(Code)**
2. Provide a diagram to show relationships between processes. **(Report)**



**Task 3**

In reference to the below figure, rearrange the following points to show their sequence in action and describe what are they and what is the OS role in each one of them:

1. An interrupt signal is one of several that device controllers can send via the interrupt request line (IRL) to tell hardware that it needs to be given CPU attention. These signals, which the CPU picks up during its cycles, signal an occurrence that needs to be dealt with right away. These signals are recognized and prioritized by the OS, which makes sure the CPU works on the most important tasks first. It tracks IRL activity in a methodical manner, identifying the precise hardware source of the interrupt and storing the associated interrupt number for processing.
2. Interrupt Vector Table: Linking interrupt identifiers to their corresponding interrupt source addresses, the Interrupt Vector Table is a crucial directory located in system memory. Every table entry links to an ISR's memory address that is customized for a particular interrupt. When the OS first starts up, it carefully sorts through this database, matching each interrupt to a useful response routine. This makes it easier to quickly retrieve and run the relevant ISR when an interrupt comes in.
3. Interrupt Service Routine (ISR): An OS established procedure that is intended to react to interrupts is called an ISR. The CPU leaves its ongoing job to execute the interrupt-specific routine (ISR) when an interrupt is detected. These routines are effectively managed by the OS, which makes sure every interrupt is recognized and handled in a way that preserves system integrity and ranks tasks based on how important they are to the system's operation.
4. Device Driver: A device driver serves as a mediator, translating OS commands into actions that hardware devices can comprehend. These drivers must be loaded and their functions must be coordinated by the OS, especially when interrupts are involved. By guaranteeing that every driver can efficiently connect with its ISR, it offers a simplified channel for data transfer between hardware and software.

Sequence in Operation:

The procedure starts when an interrupt is generated on the IRL by a device driver in response to a hardware event. The system bus, which serves as the main conduit for component communication inside the computer, carries this activity.

The CPU pauses its ongoing work as soon as it detects an interrupt through the IRL. To determine which ISR to execute, it looks up the interrupt identification in the IVT.

The selected ISR assumes control and saves the required state data before completing the interrupt service tasks—which could include input/output management or processing—that are necessary.

After finishing, the ISR deactivates the interrupt at the device level and gives the CPU back control. The CPU then resumes the job that was stopped, guaranteeing the uninterrupted flow of operations.

**Task 4**

Research the evolution of the operating system scheduler, specifically in Multilevel Queue and Multi level feedback queue, what are their benefits over the traditional scheduling algorithm mentioned in Task 1.

Transitioning from Single-Task Operations to Adaptive Scheduling: Tracing the Progression of OS Schedulers

When computing was just getting started, there was just one user and one task per domain. There was no competition for the lone processor's resources, therefore scheduling wasn't necessary. This simplicity, however, was only temporary, as the development of more powerful computers ushered in a time when multitasking was the norm and the single-task approach was no longer sufficient.

Jobs were gathered and processed in bulk during the 1960s, which also marked the birth of the first batch processing era. During this time, the First-Come-First-Served approach and other basic scheduling systems were developed, allocating CPU time in a linear, arrival-based order. While this improved operational efficiency, it was not responsive enough to the changing needs of the computing environment.

The revolutionary years of the late 1960s and early 1970s gave rise to the ideas of time sharing and multiprogramming. The introduction of concurrent processes competing for CPU time brought about by these developments necessitated a more intricate scheduling strategy. This is where algorithms like Round Robin and Shortest Job Next became important, balancing system agility and equity by giving shorter jobs priority and allocating CPU time in fair chunks. Until a process voluntarily gave up control, the early versions of these schedulers were non-preemptive and allowed a process to run to completion. Preemptive variations, which gave the system the power to proactively grab and reallocate CPU time to more urgent tasks, quickly followed this.

Schedulers changed to accommodate these additional needs as operating system complexity increased and resource demands grew. Let's explore Multilevel Feedback Queue Scheduling (MFQS), a more sophisticated version of Multilevel Queue Scheduling (MQS). MQS ensured that crucial processes were given priority by classifying jobs into various queues according to a predetermined hierarchy of importance. This idea was developed by MFQS, which made it possible for processes to roam freely between queues based on the features of their execution. This led to a system that was more responsive and effective.

MFQS distinguishes itself with multiple unique benefits. It makes use of dynamic prioritizing to make sure that critical operations receive the resources they require on time. There is an aging process in place in MFQS to ensure that lower-priority jobs are not perpetually neglected. Furthermore, MFQS demonstrates a high degree of adaptability, optimizing resource use as well as overall system performance by adjusting itself to the changing workload demands and user patterns.

The development of OS schedulers, from their beginnings as monotasking systems to their current state of sophisticated adaptive prioritization, is a reflection of the revolution in computing. This evolution is best exemplified by MFQS, which is always responding to the combined needs of end users and technology advancements to provide a smooth and effective computing experience.

Providing more details about the advantages of MFQS over conventional scheduling techniques

Prioritization: In heterogeneous workloads, MFQS distinguishes between jobs, prioritizing user-interactive procedures for instant responsiveness and postponing less important background operations. It skillfully handles the execution of different process durations by prioritizing short-term computing activities over lengthy ones.

Prevention of Process Neglect: MFQS reduces the possibility that important operations will be overshadowed by less important but longer-running processes by employing aging mechanisms to make sure that jobs put on the back burner ultimately come to the forefront.

Optimal Resource Allocation: MFQS dynamically distributes CPU time, giving critical processes priority in contexts like servers where prioritization is critical.

Versatility: By moving commonly used apps to higher-priority queues, MFQS adaptively adjusts queue priorities based on the time of day and the type of jobs. This improves user experience overall.

Replacing Conventional Schedulers: MFQS provides a dynamic scheduling hierarchy, which makes sure that shorter but important tasks are not impeded by longer procedures, in contrast to the inflexible FCFS. MFQS, as opposed to SJN, avoids the possibility of ignoring important but lengthy tasks by means of flexible priority changes and built-in aging mechanisms.

**Task 5**

suppose you have the following two C programs.

**Static Linking and Loading: A Comprehensive Analysis**

The creation and implementation lifecycle of a program in a static system consists of the following interconnected steps:

**Phase of Compilation:** For Program 1, Main.c, Function1.c, Function2.c, and for Program 2, Main.c, Function3.c, Function4.c are the individual source files that are compiled first. High-level code is converted into machine-readable object files (\*.o) during the compilation process, which are code that is not yet ready for execution. The raw code for the functions defined within these object files is contained therein; however, code for library functions such as scanf is absent.

**Linking Phase:** The linker assumes a central role after compilation, merging the object files into a single, stand-alone executable. This executable is larger under a static scheme because it includes copies of the identical library functions for every application, together with all the code it might possibly need. As a result, some executables replicate scanf and other C library routines. This duplication is inherent in the static linking process, while not explicitly shown, and leads to larger executable files for Program 1 and Program 2.

**Phases of Loading and Execution:** The loader moves the complete executable into the system's memory when it's time to run the program. This method is all-or-nothing; all code, whether used or not, is loaded, which could strain system resources and prolong the program's startup time. At that point, the program starts to run, settling into RAM so that the CPU can start carrying out its instructions.

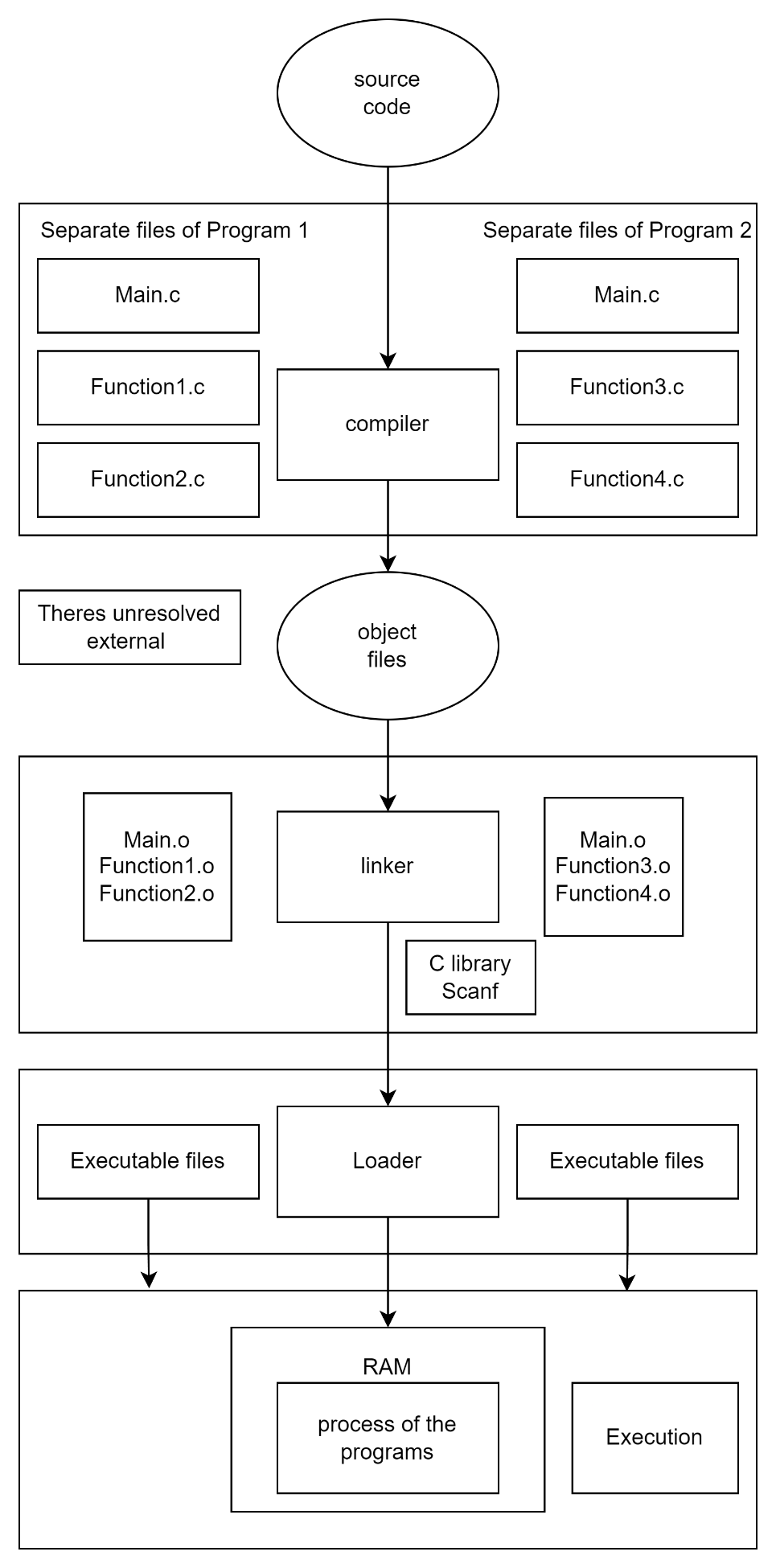
**Combined Perspective on Static Implementation:**

**Increased Memory and Disk Usage:** Because all library code is included in each executable, the static approach results in bigger executable sizes and a larger memory footprint.

**Easier Deployment and Execution:** Deployment is simple because runtime linking is not a complicated process. Because all of their dependencies are included in the executables, they can operate independently on any suitable system without the need for any setup.

**Lack of Flexibility:** Recompiling the entire executable is necessary when updating or patching a statically linked library. As a result, maintenance may become more labor-intensive and less flexible.

**Consistency and Reliability:** Because the executable is self-contained, static linking can provide a more predictable and consistent execution environment, resulting in fewer runtime shocks caused by library version mismatches.



**Dynamic Linking and Loading: An Integrated Framework for Effective Execution**

The process of converting source code into an executable program is improved by the dynamic linking and loading method, which is distinguished by its flexibility and optimization of runtime resources:

**Compilation**: Every program's source files are combined to create its object files. Unresolved external symbols, which are effectively references to functions within shared libraries, are prepared in a unique way in these files. The reason this first step is so important is that it creates the framework for effective memory use by compiling the required application code in a small footprint.

**Linking**: The work of the linker is essential but limited; it builds symbol tables for every object file, identifying the locations of external functions but leaving them unresolved. By doing away with the necessity to duplicate frequently used routines like scanf across several executables, this approach opens the door to a decrease in the amount of memory and disk space used.

**Loading**: Instead of loading all of the shared libraries at once when the application launches, the loader carefully places the executable into memory. This method of selective loading significantly lowers the program's initial memory usage, which helps it start up faster.

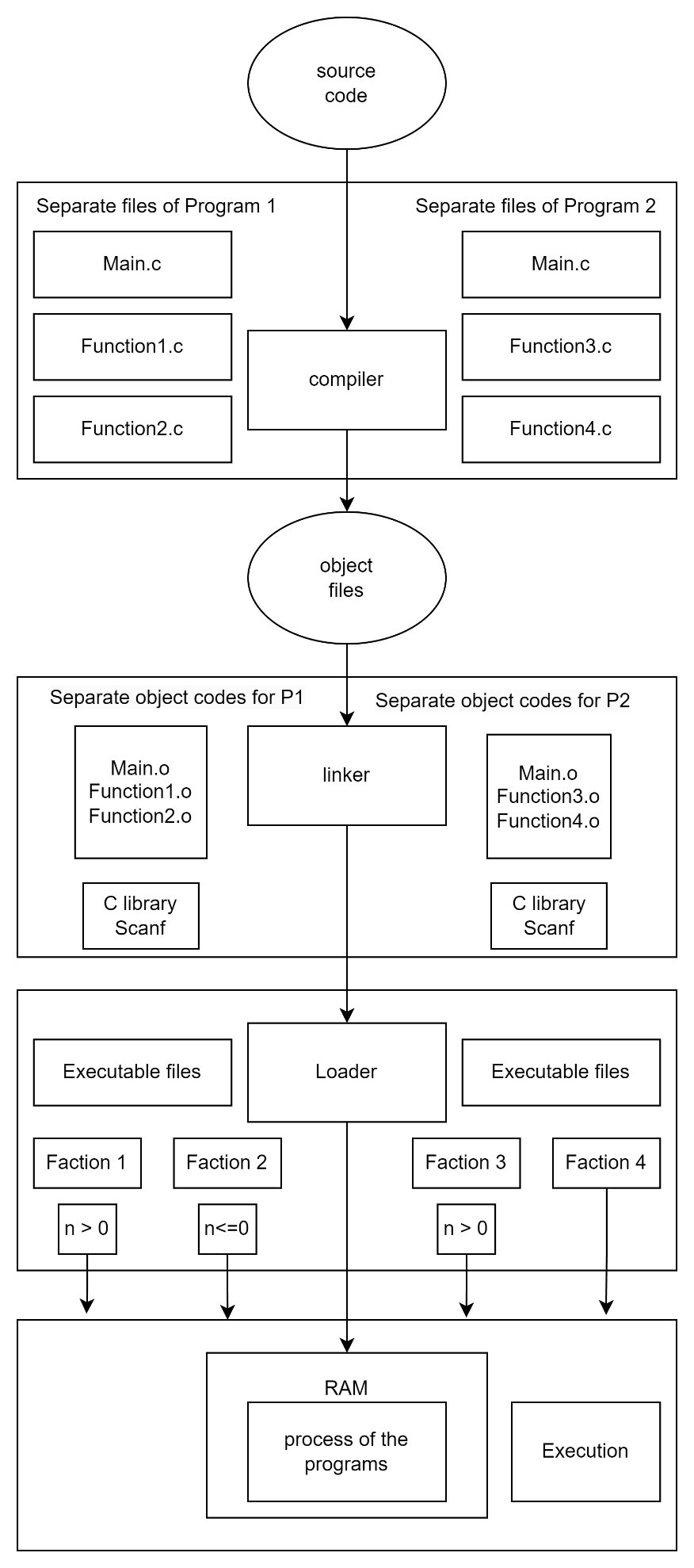
**Execution**: When the program calls on different functions during this phase, the dynamic linker is activated and resolves external symbols in real-time. By guaranteeing that memory is only allotted to functions that are actively using it, just-in-time loading dynamically optimizes the program's memory footprint while it is running.

**Combined Benefits:**

Memory and Startup Optimization: Only the required parts of the software are loaded at startup thanks to dynamic loading, which guarantees a minimal memory footprint and quick access to the main functions of the program.

Adaptive and Efficient Code Sharing: By allowing programs to share common libraries and update and modify programs without requiring complete recompilations, the system greatly lowers code redundancy and boosts efficiency.

On-Demand Function Loading: This technique optimizes memory utilization by dynamically loading functions as needed, which also guarantees that the system's performance is adjusted in real-time to meet the unique requirements of each program.



**Recourses**