|  |  |
| --- | --- |
| **[Midterm]** | [OPERATING SYSTEMS]  **LIYAN AQEL**  **[21110405]** |

**Table of Contents**

[**Part 1** 2](#_Toc154078823)

[**Part 2** 6](#_Toc154078824)

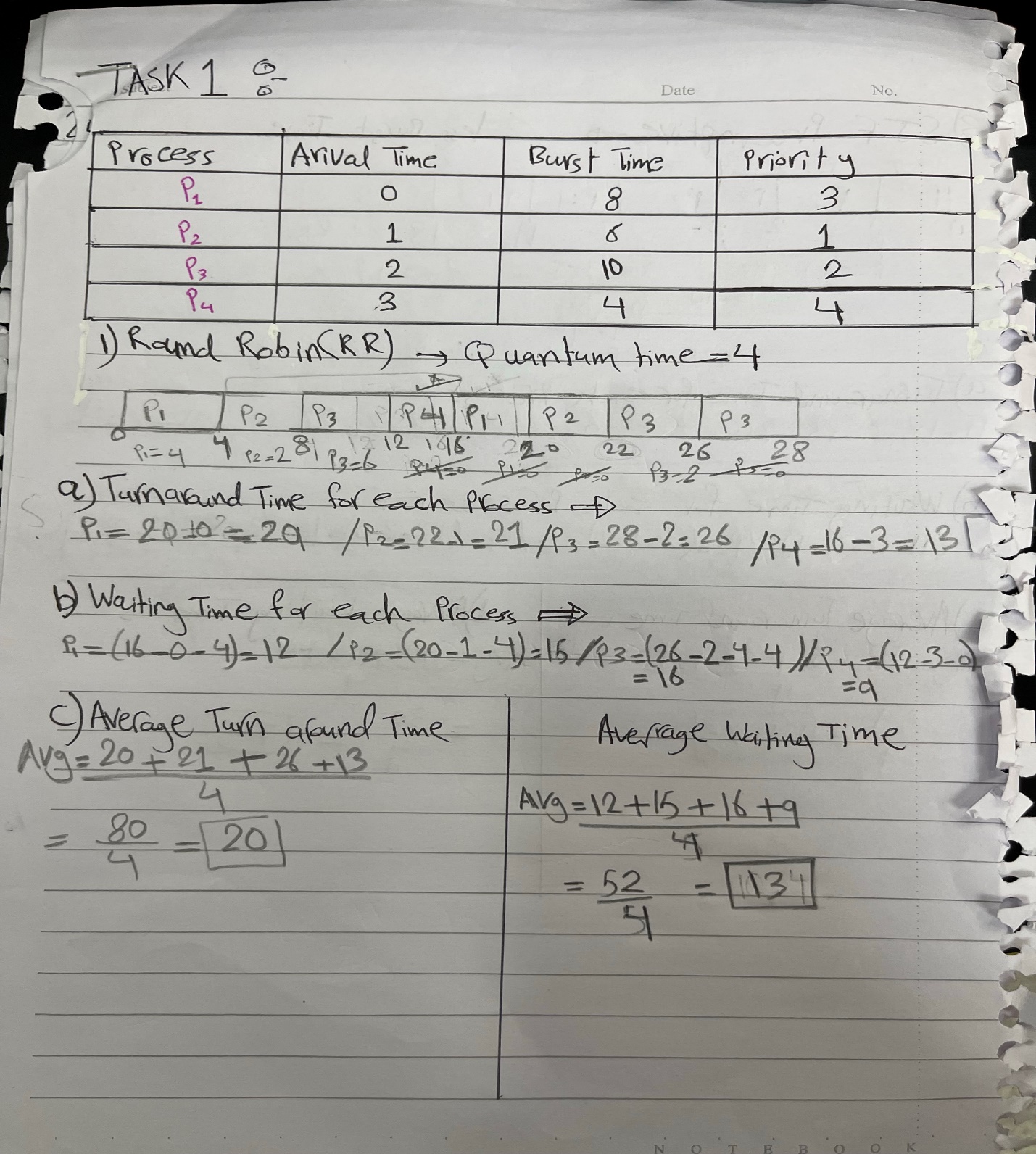
[**Part 3** 6](#_Toc154078825)

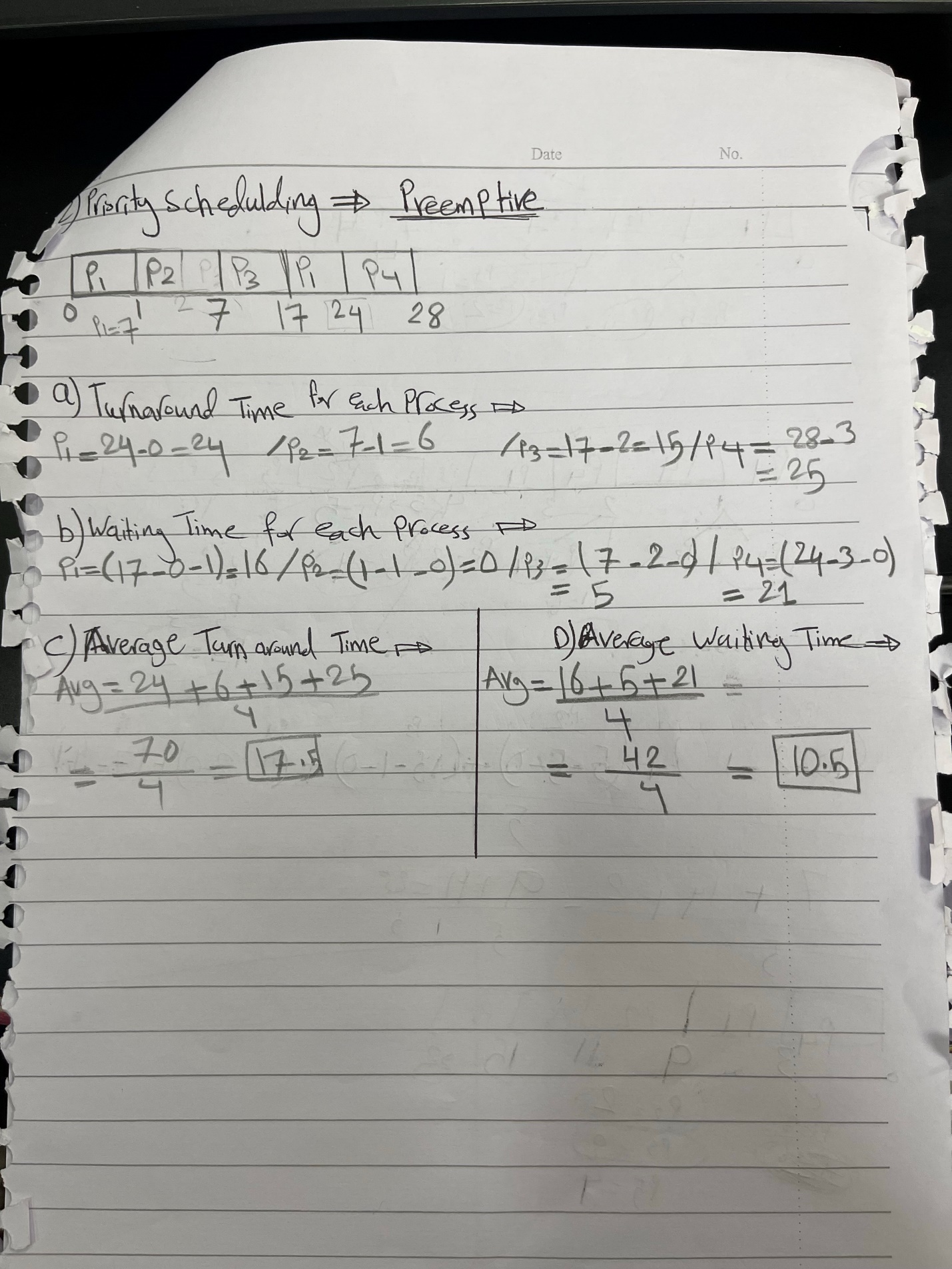
[**Part 4** 8](#_Toc154078826)

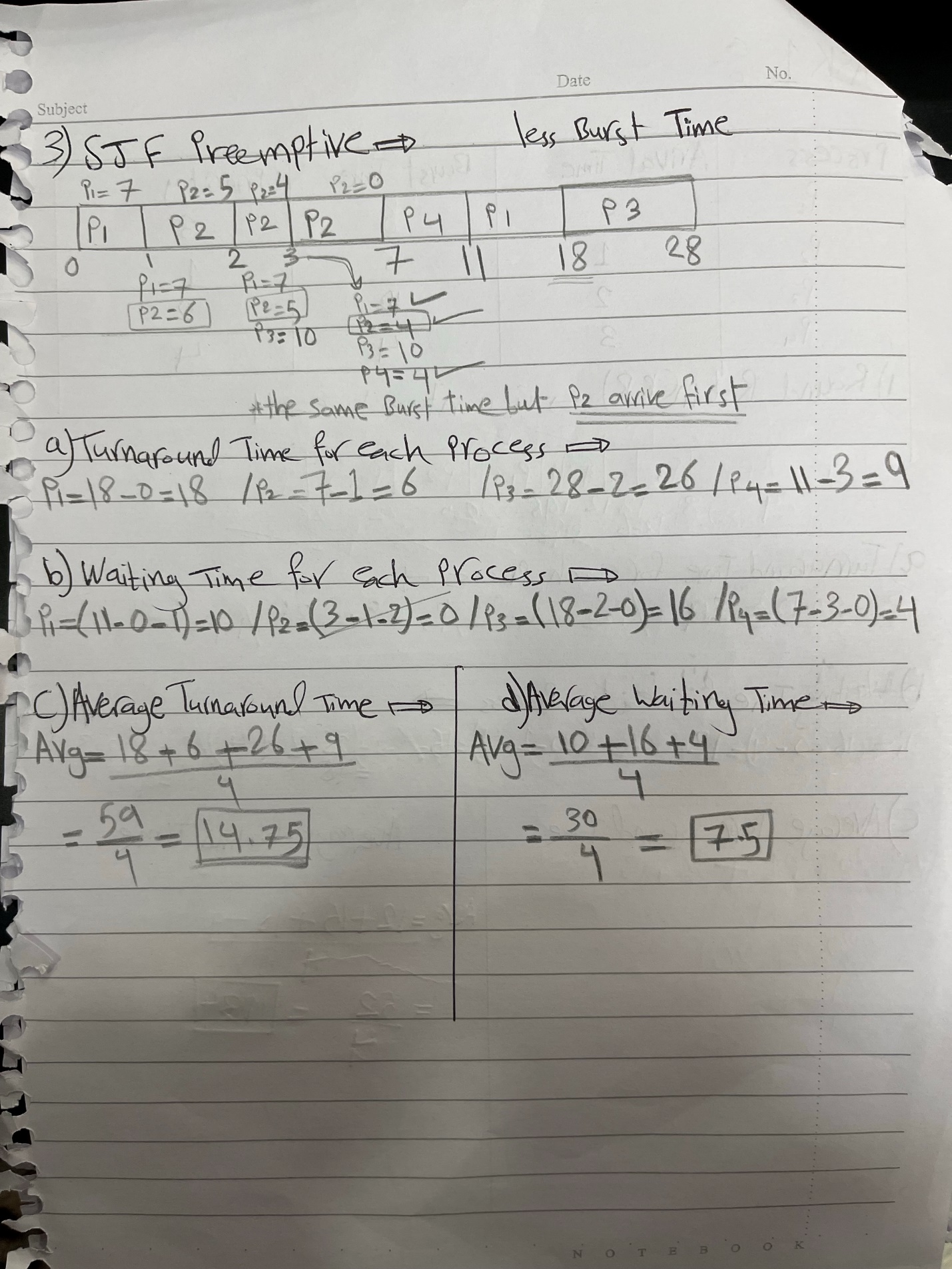
[**Part 5** 11](#_Toc154078827)

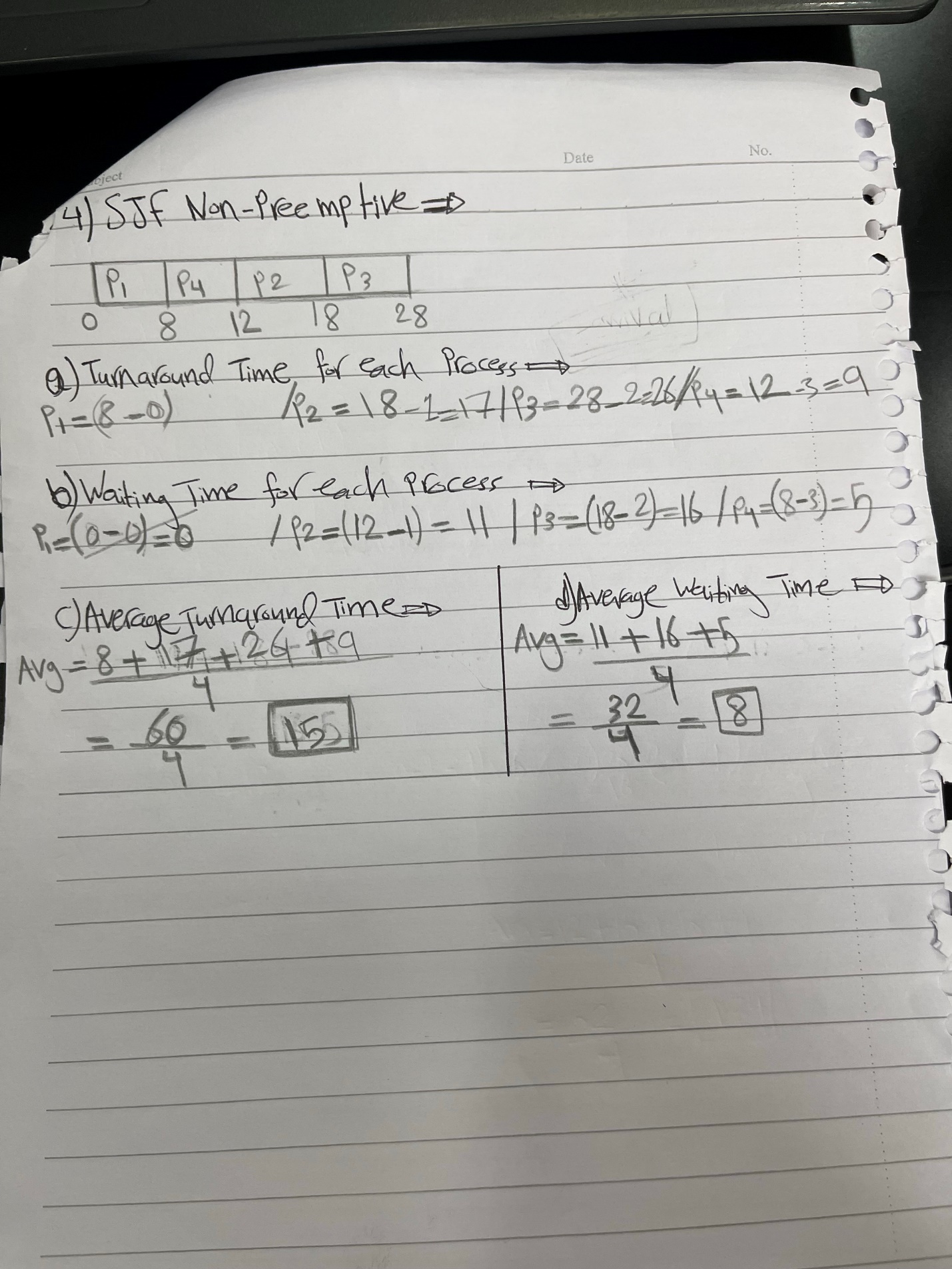
[**References:** 23](#_Toc154078828)

# **Part 1**



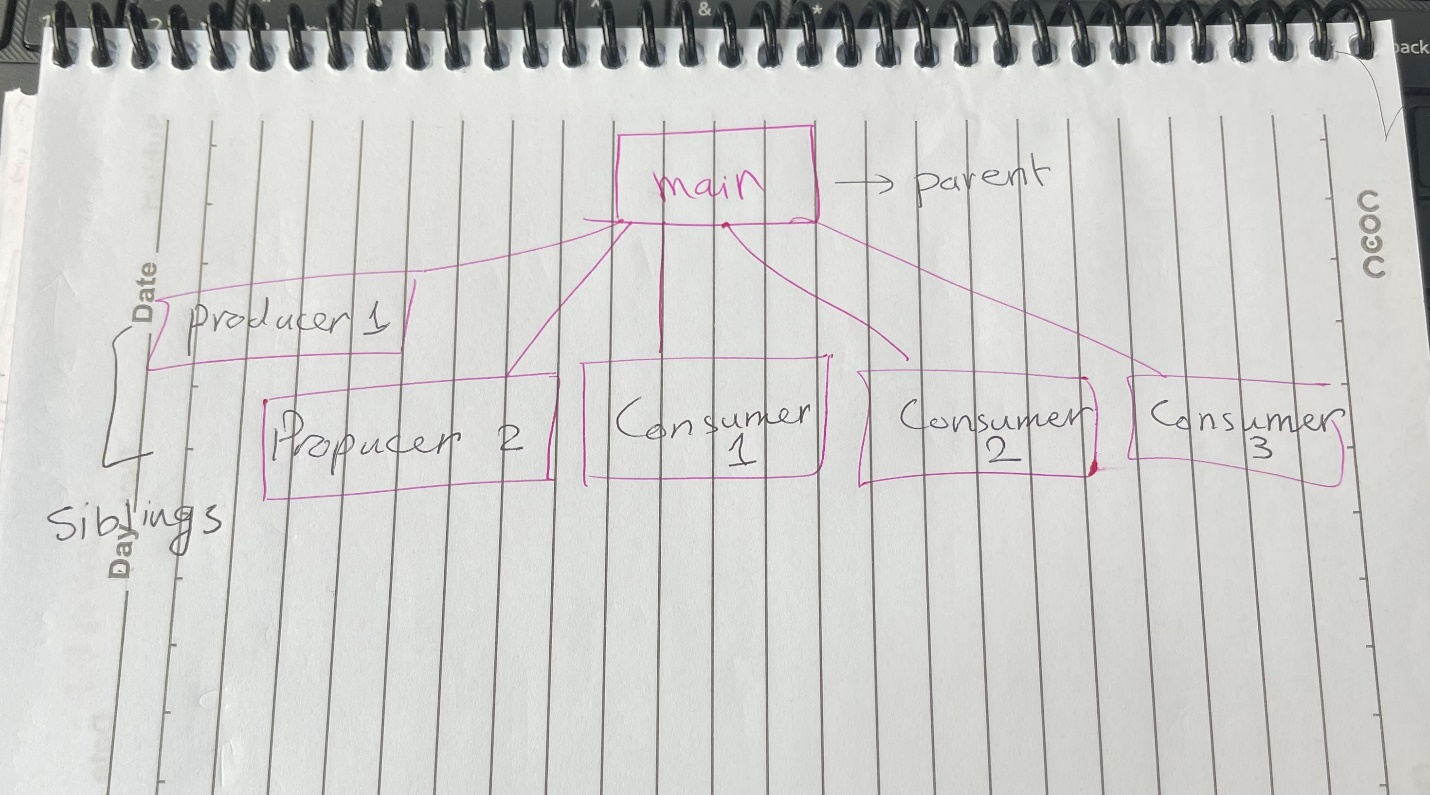






# **Part 2**

* Done



# **Part 3**

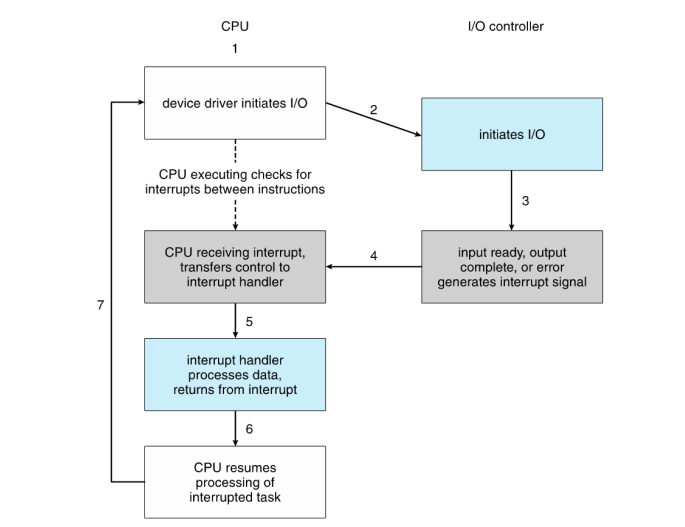
**Rearrange the following points to show their sequence in action:**

Interrupt Service Routine. [4]

Device diver. [1]

Interrupt signal. [2]

Interrupt vector table. [3]



**Describe what are they and what is the OS role in each one of them:**

The I/O operation begins with the CPU sending a request to the I/O device by **the device driver**. The device driver, responsible for managing communication between the CPU and the I/O device, making this interaction easy. Then, the device driver loads the suitable registers on the device controller to start and configure the I/O operation. In addition, the device controller examines the data in the registers, determining the appropriate actions. Also, it starts transferring data from the I/O device to the local buffer. When the local buffer is full, the I/O device signals an **interrupt**, telling the device driver that the operation is complete. Moreover, the device driver informs the operating system (OS) about the interrupt, giving the control to the OS to tell the CPU that there is an interrupt to be executed. The OS, receiving the interrupt, takes responsibility for saving the current state of the CPU before proceeding.

The OS uses the interrupt number to access the **Interrupt Vector Table (IVT).** The IVT contains entries corresponding to different interrupts, these entries store the pointers which point at another place in the memory which saves the **Interrupt service routines (ISRs).** It is executed by the CPU. The ISR provides instructions on how to deal with the specific interrupt. After ISR execution, control returns to the OS, enabling it to determine the next steps and which interrupt to address next.

**The OS is essential in all the steps of I/O operations. It plays many important roles as follows:**

* The OS manages all the interrupt system by creating protocols which recognize and respond to the interrupt signals. Also, it determines the type of interrupt and the ISR appropriate for this interrupt. Also, it saves the current state of the CPU and gives control to ISR appropriate for this interrupt.
* The OS manages and controls the IVT which maps the interrupt numbers with ISR which are corresponding to them. Also, It makes sure that the suitable ISR is executed when the interrupt takes place.
* The OS is responsible for defining and managing the ISR which is a piece of code that deals with a specific piece of interrupt, and it tell the CPU how to deal with the interrupt. When an interrupt happens, it directs the control to the suitable ISR. It is also part of the OS code and carries out tasks such as managing data and dealing with errors or recognizing I/O completion.
* The OS loads and manages device drivers and gives standardized interface to interact with various types of hardware. Also, it involves the device driver which makes communication and control between the hardware and the OS very easy in handling the interrupt. Also, the OS relies on device driver to perform some tasks such as starting I/O operations.

# **Part 4**

**Multilevel Queue:**

It is a scheduling algorithm that is used to make OS fair and efficient as much as possible in managing and handling processes. This technique consists of more than one queue and each queue has its own priority level. Also, each queue has a different category of processes such as real time, system, batch, interactive process, as well as different scheduling algorithms such as round robin, SJF, priority scheduling. Giving priority to any queue depends on the type of processes in the queue. So, the processes will be divided into many queues based on their categories and they will go out from the queue depending on the scheduling algorithm that it uses. **The main idea of this technique** is to execute the processes that have higher priority levels, then executing processes that have lower priority levels. Also, if the processes assignees to a certain queue, it will always belong to the same queue and can’t move to another queue.

Let’s take an example of how this technique works. There are two queues; the first queue includes interactive process (foreground process), and the second queue includes batch process (background process). The interactive processes should be interactive with the user and should be executed faster so it should have higher priority than batch processes. In this technique, there is a scheduler algorithm between the processes and between the queues themselves which is fixed preemptive priority.

An example of preemptive is if background process is being executed and an interactive process arrives, the background process will be in a waiting state and the CPU starts to execute the interactive process and the background process won’t be executed till the interactive process is completely finished.

**Multilevel feedback queue:**

It is a **dynamic** scheduling algorithm that is used to make OS fair and efficient as much as possible in managing and handling processes. MLFQ is more extended than MLQ, which means it enables the processes to change their priorities depending on their behavior which means that they can move between queues. The main goal of this technique is to avoid starvation, offer responsiveness, use system resources efficiently, and improve performance of the systems. For example, if a high priority process takes a long time to be executed, and low priority processes need to be executed in a short time. This technique allows the processes to move between queues. Also, if the process has been waiting for a long time in low priority queue, this process can move to higher priority queue, which will prevent starvation.

**The MLQ & MLFQ have many benefits over Roin Robin (RR) scheduling algorithm:**

**MLQ:** It helps processes to have different priorities to make sure that the processes with high priorities are executed more repeatedly until they are finished. Also, processes are classified into queues depending on their category which determine the priority which makes the technique more organized. However, processes with low priority need to wait until high priority processes are executed first, which is called starvation.

**MLFQ:** It helps to adjust priority processes dynamically depending on their behavior. As a result, it allows (RR) to be more adaptable to different process behavior. Also, using this technique helps to avoid starvation which means that low priority process is not postponed to the end to be executed. As a result, MLFQ works like a clever task organizer. It increases the flexibility of Round Robin scheduling by modifying the priorities of processes according to their behavior. Additionally, it makes sure that less important processes don't go unfinished for a long amount of time.

**The MLQ & MLFQ have many benefits over Priority scheduling algorithm:**

**MLQ:** It gives priority levels with many queues, a more organized and structured approach. Also, in this technique the priority of the process is fixed.

**MLFQ:** It makes sure that executing the processes depending on their priorities is fair, and so it prevents starvation of process with lower priority. Also, it gives dynamic priority to the process, so that they can move freely in the queues.

**The MLQ has many negative effects on SJF Preemptive and non-Preemptive scheduling algorithm:**

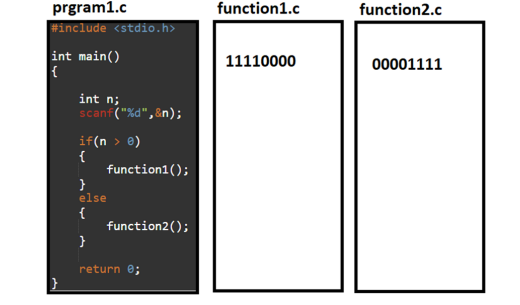
This technique isn’t completely efficient regarding executing processes depending on the least burst time because it leads to starvation, which means processes with high burst time need to wait to be executed until low burst time processes are executed.

**The MLFQ has many benefits over SJF Preemptive and non-Preemptive scheduling algorithm:**

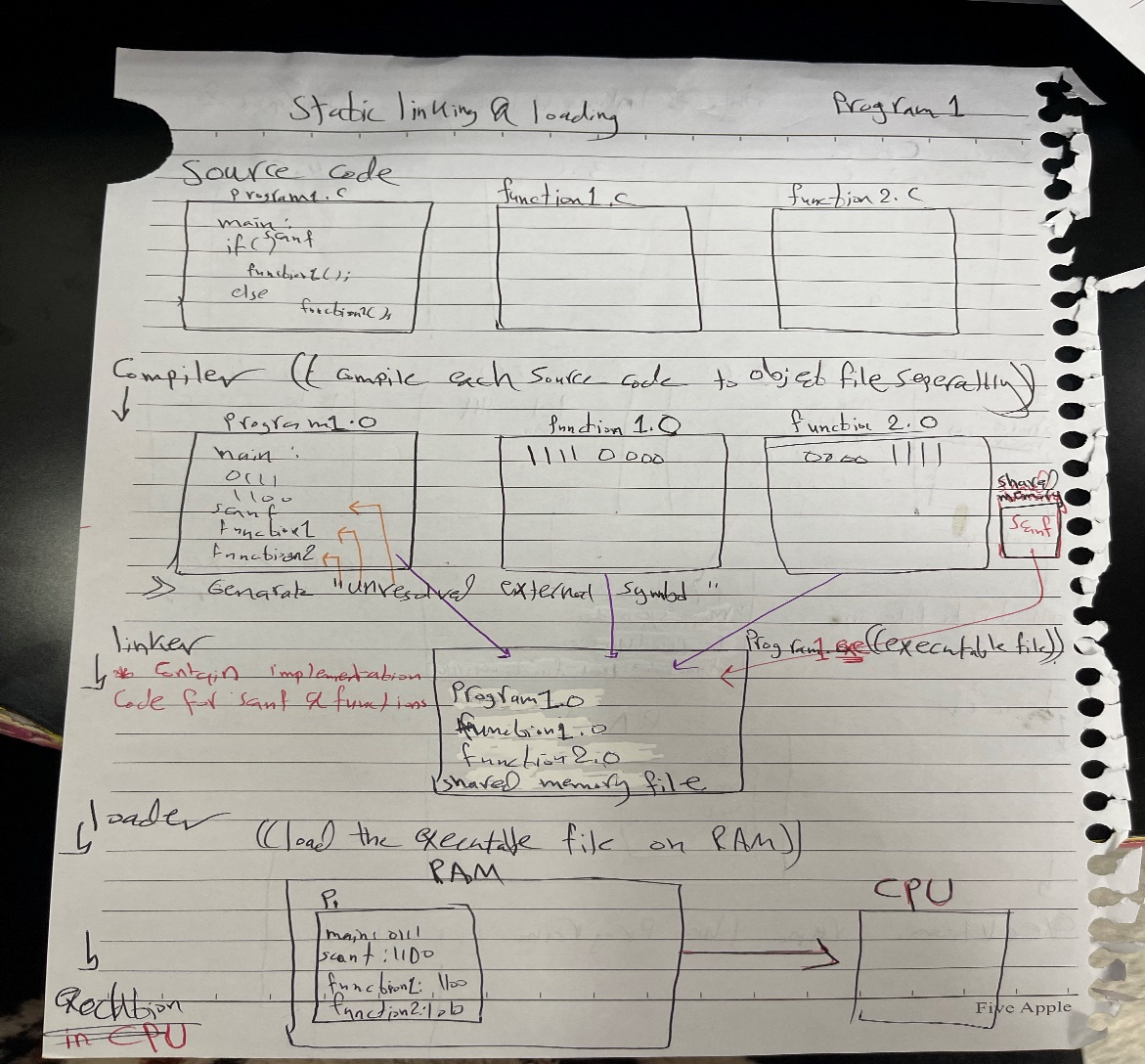
This technique is efficient regarding executing processes depending on the least burst time because it avoids starvation. Also, it ensures responsiveness because it helps processes to cope with different burst times, as well as making responsiveness to the changes in processes behavior faster.

# **Part 5**

**Program 1 🡪**

****

**Static loading and linking**



**Compiler Stage**

In this stage, the complier compiles sources codes (program1.c, function1.c, function2.c) separately to generate the object files (program1.o, function1.o, function2.o) for each source code. The compiling of source code to machine code for each function is based on the code in the function. **Moreover, the source code has high level language, and the object file has machine language.**

During the compiling process, when the compiler finds external symbols (function1, function2), it generates information which is “unresolved external symbol” in the object file, because they are declared in a source file but not defined in the same source file.

Declared🡪 Declare a function but leaving out how it is implemented. For example,

void function1();

Defined🡪 Definition and implementation for the declared function. For example:

void function1() { // Implementation here }.

According to ‘scanf’ function is part of the C library and it is a shared library, which means that more than one program can use it at the same time. It is provided by the operating system. During this stage, the compiler identifies the scanf' function, which is a predefined function included in the standard C library and creates a reference to it in the program1.o object file. Still, at this point, the actual implementation details of scanf' are not included. The compiler does not include the exact instructions for 'scanf' in the object file during the compiler stage, even though it is aware that 'scanf' is a function. When the C library is integrated into the executable at the linking stage, the actual implementation details should be provided.

**Linker Stage**

It is the normal behavior in many compiling processes, and it is responsible for gathering all the object files in one file which is called executable file. In this stage, if there are external symbols such as function1 and function2, ‘scanf’, the linker should look for and resolve them during the linking stage by looking for their actual locations or addresses in the related libraries or object files. When it finds them, it changes the “unresolved external symbol” into the actual address in the executable file, so that they point to their actual addresses where the linker found them. Finally, the linker generates the final executable file which includes the actual addresses for external symbols. By locating and merging the exact addresses of external symbols into the executable, the linker makes sure that all references to those symbols are resolved appropriately.

The linker's job during the static linking stage is to locate the C library's real implementation of "scanf" and combine it with the executable file. Because of this technique, the application is able to call "scanf" during runtime and the executable file contains the right instructions for scanf'.

**Loader Stage**

In this stage, the executable file is loaded into the memory by the loader. The linker creates an executable file, which is then loaded into memory. In this step, all symbol addresses—including those of user-defined functions ('function1' and 'function2') and library functions ('scanf')—are resolved. The exact addresses or locations of each symbol in the executable file are contained in the symbol table that the loader uses, which was created by the linker during the linking stage.

The primary goal of the loader is to make sure that all symbols are given the correct addresses so that the program can run as intended. A linking error occurs if any symbol—whether user-defined or library—cannot be resolved at this point. These errors show that the program cannot be successfully executed because the loader cannot locate the implementations for specific functions.

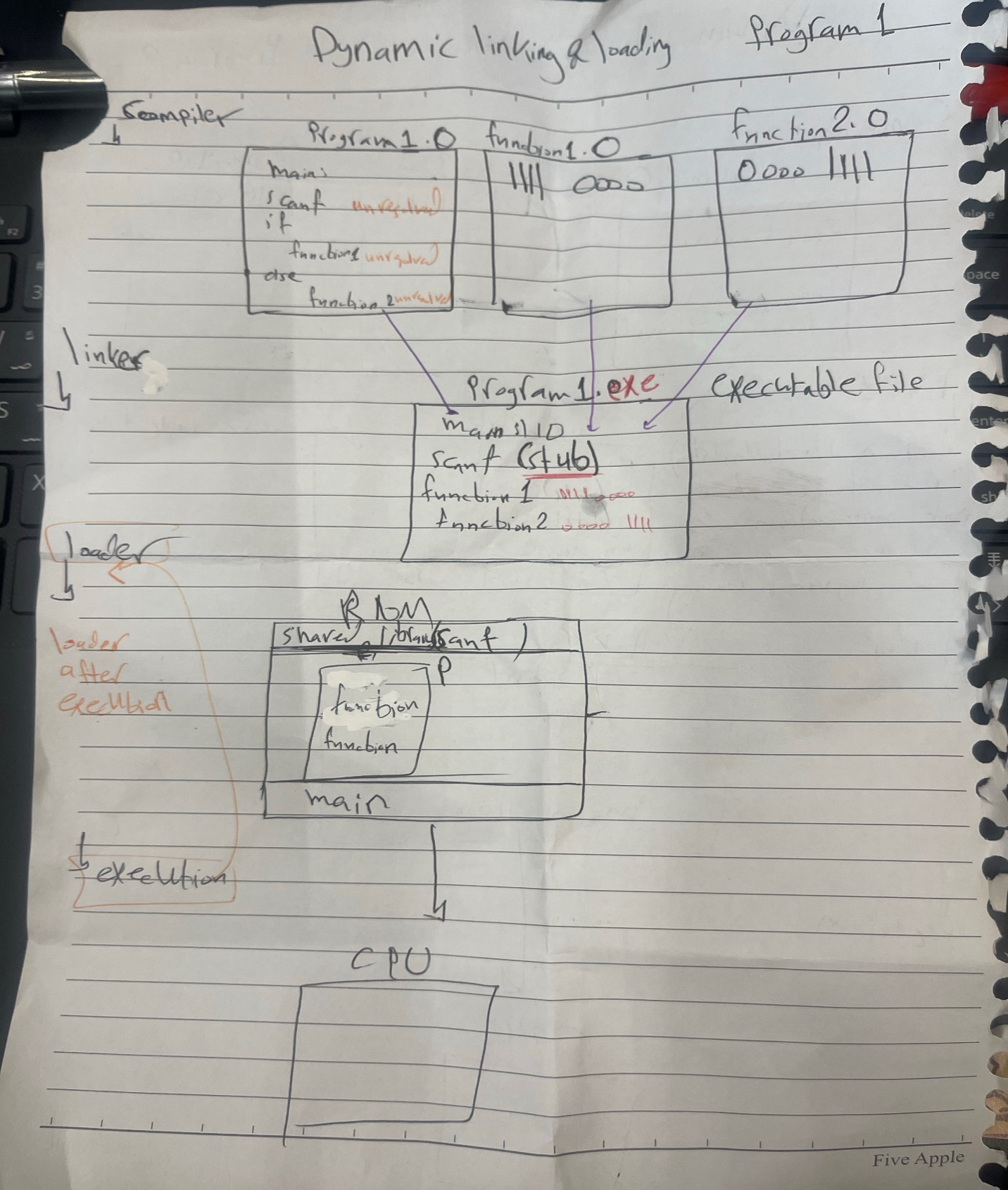
The program runs smoothly when all symbols are successfully resolved during the loading phase. The logic of the program determines which user-defined functions (functions 1 and 2) or library functions (scanf) should be invoked. In order to ensure a seamless transition from the loading to the executable stage, the loader's function is crucial in ensuring that the program's execution has access to the correct addresses for all symbols.

In summary, the program starts to execute if the linker resolves all the external symbols in the previous stage, but if these are unresolved external symbols, the program can’t execute, and the loader makes an error because it is unable to find the implementations for the functions.

**Executable Stage**

The executable file successfully runs and function1 or function2 is called during this stage depending on the value that the user enters by ‘scanf’ method.

**Dynamic loading and linking**



**Compiler Stage**

In this stage, the complier compiles sources codes (program1.c, function1.c, function2.c) separately to generate the object files (program1.o, function1.o, function2.o) for each source code. The compiling of source code to machine code for each function is based on the code in the function. **Moreover, the source code has high level language, and the object file has machine language.**

During the compiling process, when the compiler finds external symbols (function1, function2), it generates information which is “unresolved external symbol” in the object file, because they are declared in a source file but not defined in the same source file.

Declared🡪 Declare a function but leaving out how it is implemented. For example,

void function1();

Defined🡪 Definition and implementation for the declared function. For example:

void function1() { // Implementation here }.

According to ‘scanf’ function is part of the C library and it is a shared library, which means that more than one program can use it at the same time. It is provided by the operating system. During this stage, the compiler identifies the scanf' function, which is a predefined function included in the standard C library and creates a reference to it in the program1.o object file. Still, at this point, the actual implementation details of scanf' are not included. The compiler does not include the exact instructions for 'scanf' in the object file during the compiler stage, even though it is aware that 'scanf' is a function. When the C library is integrated into the executable at the linking stage, the actual implementation details should be provided.

**Linker Stage**

The dynamic linker stage is useful when the code has a shared function in shared library. The linker is responsible for gathering all the object files in one file which is called executable file. In this stage, if there are external symbols for user defined functions such as function1, function2 the linker looks for and resolves them during the linking stage by looking for their actual locations or addresses in the related libraries or object files. When it finds them, it changes the “unresolved external symbol” into the actual address in the executable file, so that they point to their actual addresses where the linker found them.

But if there are external symbols for predefined functions such as scanf function the linker will not look for and resolve them during the linking stage. Instead, it leaves them unresolved and creates an executable file with placeholders (stub) for these external symbols which is piece of code that use to point at the actual address. The actual addresses are found when the program is running, and these placeholders(stub) act as references that are dynamically resolved at runtime which means that these stubs replace itself with the actual address in order to be executed at runtime. Then, the OS makes sure that if the actual address isn’t in the memory, it will be added.

**Loader and execution Stage**

The dynamic loader stage is useful when the code has a certain function which is executed depending on a condition and it isn’t necessarily executed each time. This will reduce the size of processes which are loaded on the memory because only the needed functions are loaded.

In this stage, the executable file is loaded into the memory by the loader. The main idea of the dynamic loader is that loading happens after execution. First, the loader loads the main function then the program starts to be executed. During execution, if the program needs to use a function (function1 or function2) that hasn’t already been loaded, the control goes back to the loader and loads the implementation of the functions and then executes it.

On the other hand, If the program needs to use a shared function (scanf), there are two cases.

The implementation of scanf function doesn’t exist in the executable file, so the loader knows that scanf is a shared function which means that it is part of the program that isn’t included in the executable directly, but it is located during runtime.

The first case is that there is the implementation of the shared function in shared library is ready in the memory. This means that another process used this function before, so that the loader doesn’t need to load it into the memory again.

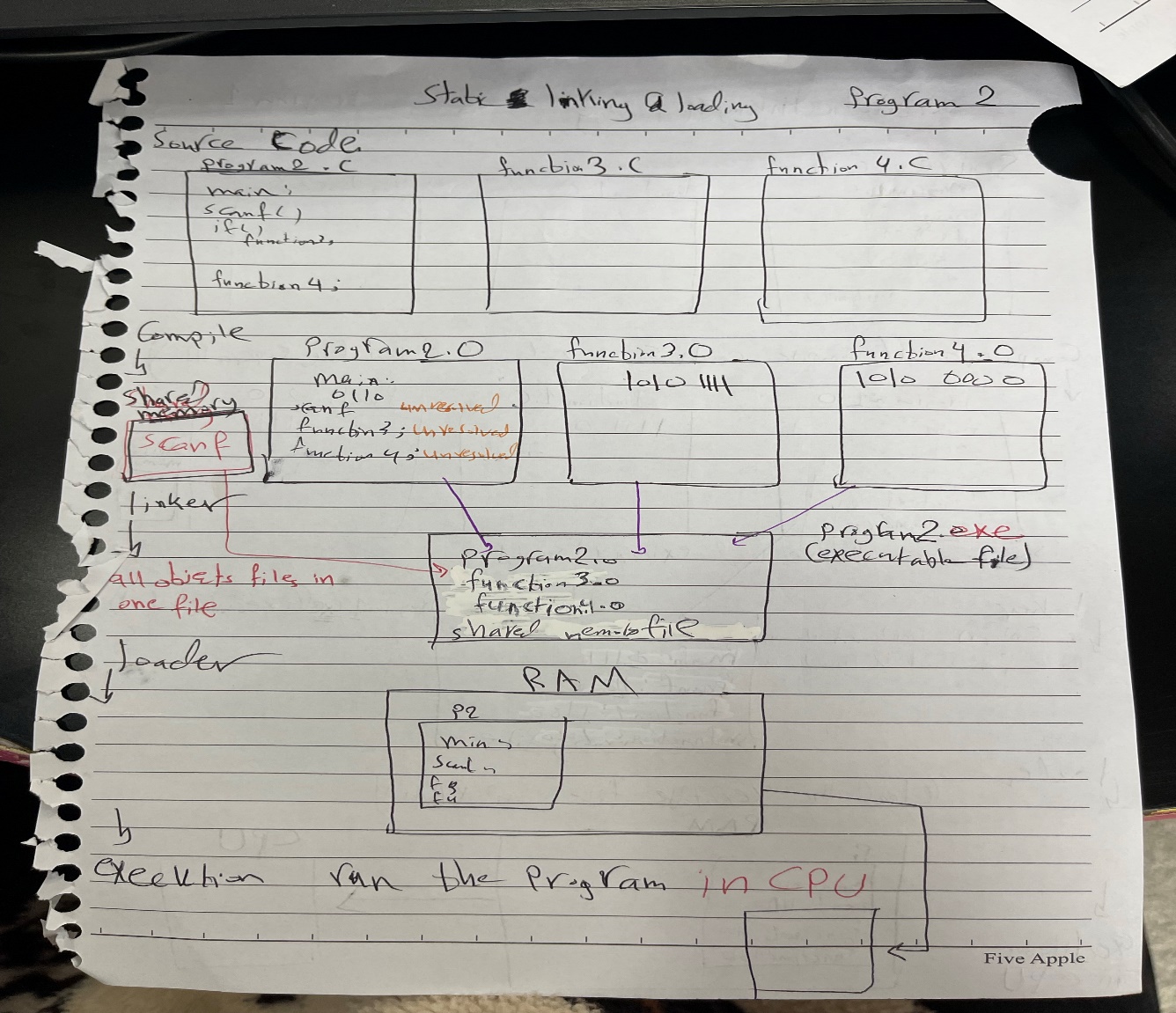
The second case is that there isn’t a shared function in the memory. This means that this function wasn’t used by another process before, so that the **stub** will look for the shared function and make linking for it. Then, the loader loads this shared library into the address space of the program which means bringing both the data and the code of the shared library into the memory at runtime. In other words, in this process the loader is as bridge that links the program and the shared libraries, and it makes sure that the shared functions are available to be used by the program.

**Program 2 🡪**

A screenshot of a computer code

Description automatically generated

**Static loading and linking**



**Compiler Stage**

In this stage, the complier compiles sources codes (program2.c, function3.c, function4.c) separately to generate the object files (program2.o, function3.o, function4.o) for each source code. The compiling of source code to machine code for each function is based on the code in the function**. Moreover, the source code has high level language, and the object file has machine language.**

During the compiling process, when the compiler finds external symbols (function3, function4), it generates information which is “unresolved external symbol” in the object file, because they are declared in a source file but not defined in the same source file.

**Declared🡪** Declare a function but leaving out how it is implemented. For example,

void function3();

**Defined🡪** Definition and implementation for the declared function. For example:

void function3() { // Implementation here }.

According to ‘scanf’ function is part of the C library and it is a shared library, which means that more than one program can use it at the same time. It is provided by the operating system. During this stage, the compiler identifies the scanf' function, which is a predefined function included in the standard C library and creates a reference to it in the program1.o object file. Still, at this point, the actual implementation details of scanf' are not included. The compiler does not include the exact instructions for 'scanf' in the object file during the compiler stage, even though it is aware that 'scanf' is a function. When the C library is integrated into the executable at the linking stage, the actual implementation details should be provided.

**Linker Stage**

It is the normal behavior in many compiling processes, and it is responsible for gathering all the object files in one file which is called executable file. In this stage, if there are external symbols such as function3 and function4, the linker should look for and resolve them during the linking stage by looking for their actual locations or addresses in the related libraries or object files. When it finds them, it changes the “unresolved external symbol” into the actual address in the executable file, so that they point to their actual addresses where the linker found them. Finally, the linker generates the final executable file which includes the actual addresses for external symbols. By locating and merging the exact addresses of external symbols into the executable, the linker makes sure that all references to those symbols are resolved appropriately.

The linker's job during the linking stage is to locate the C library's real implementation of "scanf" and combine it with the executable file. Because of this technique, the application is able to call "scanf" during runtime and the executable file contains the right instructions for scanf'.

**Loader stage**

In this stage, the executable file is loaded into the memory by the loader. The linker creates an executable file, which is then loaded into memory. In this step, all symbol addresses—including those of user-defined functions ('function3' and 'function4') and library functions ('scanf')—are resolved. The exact addresses or locations of each symbol in the executable file are contained in the symbol table that the loader uses, which was created by the linker during the linking stage.

The primary goal of the loader is to make sure that all symbols are given the correct addresses so that the program can run as intended. A linking error occurs if any symbol—whether user-defined or library—cannot be resolved at this point. These errors show that the program cannot be successfully executed because the loader cannot locate the implementations for specific functions.

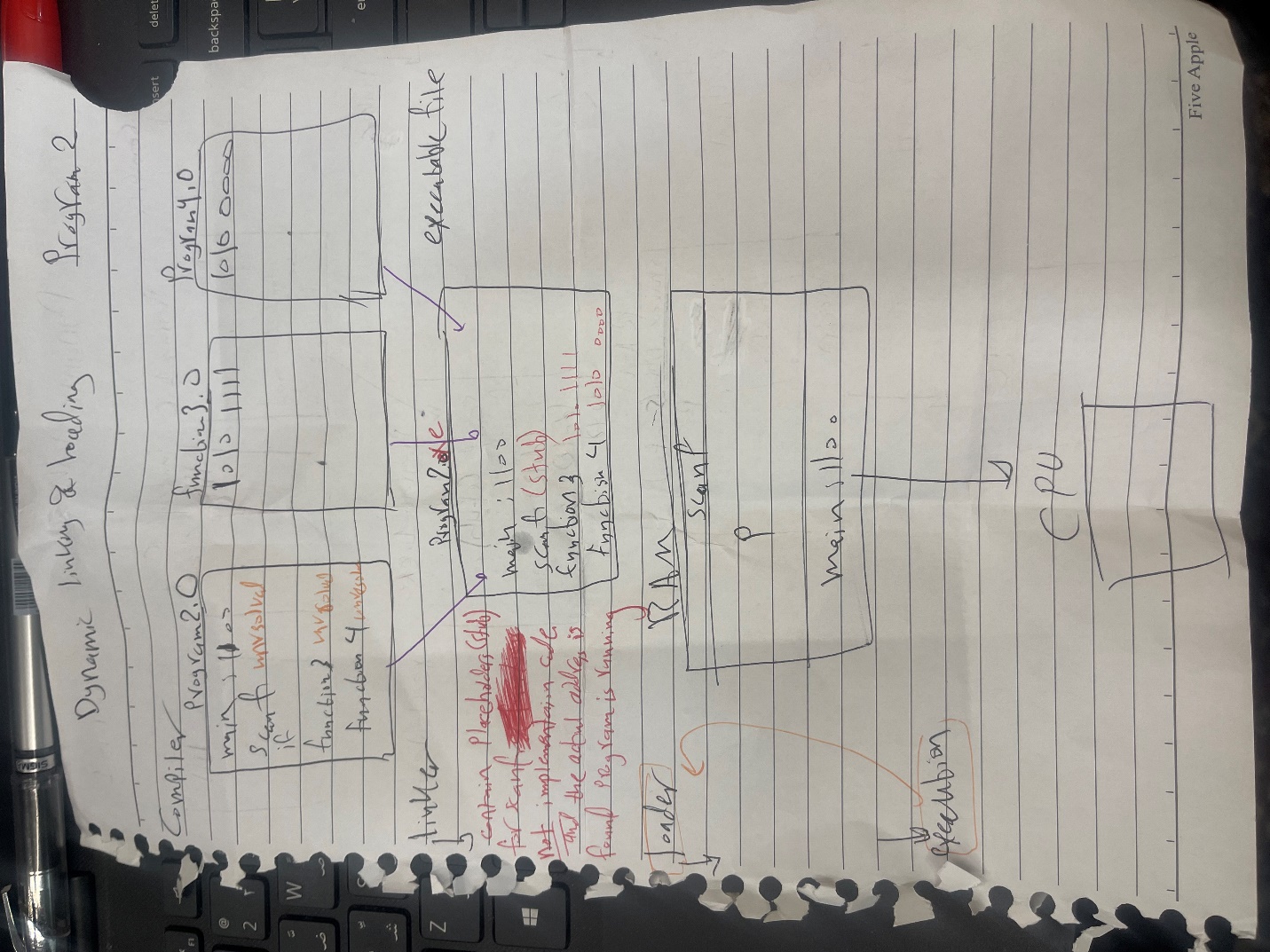
The program runs smoothly when all symbols are successfully resolved during the loading phase. The logic of the program determines which user-defined functions (functions 3 and 4) or library functions (scanf) should be invoked. In order to ensure a seamless transition from the loading to the executable stage, the loader's function is crucial in ensuring that the program's execution has access to the correct addresses for all symbols.

In summary, the program starts to execute if the linker resolves all the external symbols in the previous stage, but if these are unresolved external symbols, the program can’t execute, and the loader makes an error because it is unable to find the implementations for the functions.

**Executable file**

The executable file successfully runs and function3 is called during this stage depending on the value that the user enters by ‘scanf’ method. Function4 is called in this stage in all cases because it is out the range of if statement.

**Dynamic loading and linking**



**Compiler stage**

In this stage, the complier compiles sources codes (program2.c, function3.c, function4.c) separately to generate the object files (program2.o, function3.o, function4.o) for each source code. The compiling of source code to machine code for each function is based on the code in the function**. Moreover, the source code has high level language, and the object file has machine language.**

During the compiling process, when the compiler finds external symbols (function3, function4), it generates information which is “unresolved external symbol” in the object file, because they are declared in a source file but not defined in the same source file.

**Declared🡪** Declare a function but leaving out how it is implemented. For example,

void function3();

**Defined🡪** Definition and implementation for the declared function. For example:

void function3() { // Implementation here }.

According to ‘scanf’ function is part of the C library and it is a shared library, which means that more than one program can use it at the same time. It is provided by the operating system. During this stage, the compiler identifies the scanf' function, which is a predefined function included in the standard C library and creates a reference to it in the program1.o object file. Still, at this point, the actual implementation details of scanf' are not included. The compiler does not include the exact instructions for 'scanf' in the object file during the compiler stage, even though it is aware that 'scanf' is a function. When the C library is integrated into the executable at the linking stage, the actual implementation details should be provided.

**Linker stage**

The dynamic linker stage is useful when the code has a shared function in shared library. The linker is responsible for gathering all the object files in one file which is called executable file. In this stage, if there are external symbols for user defined function such as function3, function4 the linker looks for and resolves them during the linking stage by looking for their actual locations or addresses in the related libraries or object files. When it finds them, it changes the “unresolved external symbol” into the actual address in the executable file, so that they point to their actual addresses where the linker found them.

But if there are external symbols for predefined functions such as scanf function the linker will not look for and resolve it during the linking stage. Instead, it leaves them unresolved and creates an executable file with placeholders (stub) for this external symbol which is piece of code that is used to point at the actual address. The actual addresses are found when the program is running, and these placeholders(stub) act as references that are dynamically resolved at runtime which means that these stubs replace itself with the actual address in order to be executed at runtime. Then, the OS makes sure that if the actual address isn’t in the memory, it will be added.

**Loader and execution Stage**

The dynamic loader stage is useful when the code has a certain function which is executed depending on a condition and it isn’t necessarily executed each time. This will reduce the size of processes which are loaded on the memory because only the needed functions are loaded. This will help to increase the degree of multi programming.

In this stage, the executable file is loaded into the memory by the loader. The main idea of the dynamic loader is that loading happens after execution. First, the loader loads the main function then the program starts to be executed. During execution, if the program needs to use a function (function3) that hasn’t already been loaded, the control goes back to the loader and loads the implementation of the function and then executes it. The function4 will be executed in all cases because it is not in the if statement range. Therefore, since function4 is out of if statement range, so the control will go back to the loader every times it needs to execute function4.

On the other hand, If the program needs to use a shared function (scanf), there are two cases.

The implementation of scanf function doesn’t exist in the executable file, so the loader knows that scanf is a shared function which means that it is part of the program that isn’t included in the executable directly, but it is located during runtime.

The first case is that there is the implementation of the shared function in shared library is ready in the memory. This means that another process used this function before, so that the loader doesn’t need to load it into the memory again.

The second case is that there isn’t a shared function in the memory. This means that this function wasn’t used by another process before, so that the **stub** will look for the shared function and make linking for it. Then, the loader loads this shared library into the address space of the program which means bringing both the data and the code of the shared library into the memory at runtime. In other words, in this process the loader is as bridge that links the program and the shared libraries, and it makes sure that the shared functions are available to be used by the program.

# **References:**

* Sharma, A. (2022). *Difference Between MLQ And MLFQ CPU Scheduling Algorithms*. [online] PrepBytes Blog. Available at: https://www.prepbytes.com/blog/operating-system/difference-between-mlq-and-mlfq-cpu-scheduling-algorithms/#:~:text=Conclusion%20In%20conclusion%2C%20MLQ%20(Multi [Accessed 21 Dec. 2023].
* GeeksforGeeks. (2023). *Difference Between Static andDynamic Loading in Operating System*. [online] Available at: <https://www.geeksforgeeks.org/difference-between-static-anddynamic-loading-in-operating-system/>.
* www.javatpoint.com. (n.d.). *Static and Dynamic Loading in Operating System - javatpoint*. [online] Available at: https://www.javatpoint.com/static-and-dynamic-loading-in-operating-system#:~:text=Static%20loading%20refers%20to%20loading [Accessed 21 Dec. 2023].

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#define BUFFER\_SIZE 5

#define NUM\_PRODUCERS 2

#define NUM\_CONSUMERS 3

int shmid;

//struct as class

typedef struct {

int data[BUFFER\_SIZE];

int in;

int out;

} SharedBuffer; //name of struct

void produce(SharedBuffer \*buffer, int \*item\_list , int item\_size) {

//SharedBuffer buff; --> object

//buff.in; to access the object we use (.)

//(\*buffer).in; --> pointer

//buffer -> in; to access attributes related to object from its pointer which indicate of the object(object from shared buffer struct)

// Check if buffer is full

while (((buffer->in + 1) % BUFFER\_SIZE) == buffer->out); // Wait while buffer is full (not stop)

int item = item\_list[buffer->in % item\_size]; //add item from producer, item\_size for based on item\_list size of producer, size of in (0-5) based on buffer size

buffer->data[buffer->in] = item; //add item to buffer

printf("Producer produced item: %d\n", item);

buffer->in = (buffer->in + 1) % BUFFER\_SIZE; //incremental

}

int consume(SharedBuffer \*buffer) {

// Check if buffer is empty

while (buffer->out == buffer->in); // Wait while buffer is empty (not stop)

int item = buffer->data[buffer->out];

printf("Consumer consumed item: %d\n", item);

buffer->out = (buffer->out + 1) % BUFFER\_SIZE;

return item;

}

void producer(int \*item\_list, int item\_size) {

SharedBuffer \*buffer = shmat(shmid, NULL, 0); //by shmid we connected the buffers

while (1)

{

// Produce item

produce(buffer, item\_list,item\_size);

// Sleep for producer speed

sleep(2);

}

}

void consumer() {

SharedBuffer \*buffer = shmat(shmid, NULL, 0);

while (1)

{

// Consume item

int item = consume(buffer);

// Sleep for consumer speed

sleep(3);

}

}

// 6 buffers (2 producer, 3 consumer, 1 main) --> in same shared memory by all of them have the same shmid

int main() {

// Item lists for each producer

int producer1\_items[] = {1, 2, 3};

int producer2\_items[] = {4, 5, 6};

// Create one shared memory for 6 buffers

shmid = shmget(500, sizeof(SharedBuffer), IPC\_CREAT | 0666); //almost 28 bytes

// Initialize buffer

SharedBuffer \*buffer = shmat(shmid, NULL, 0);

buffer->in = 0;

buffer->out = 0;

// Create producers

if (fork() == 0) {

producer(producer1\_items,3);

exit(0); //to stop the peoccess (child)

}

if (fork() == 0) {

producer(producer2\_items,3);

exit(0);

}

// Create consumers

for (int i = 0; i < NUM\_CONSUMERS; ++i) {

if (fork() == 0) {

consumer();

exit(0);

}

}

//main (parent) Wait for all child processes to finish

for (int i = 0; i < NUM\_PRODUCERS + NUM\_CONSUMERS; ++i) {

wait(NULL);

}

// Clean up

shmdt(buffer);

shmctl(shmid, IPC\_RMID, NULL);

return 0;

}