

**Operating Systems**

**40201341**

**R/615/1700**

**Section (2)**

**Assignment 1**

**Submitted to**

Eng. Malik Louzi

**Submitted on**

December 21st, 2023

**Submitted by**

Marwan Tarek Shafiq Al Farah

**Student ID**

21110011

**Fall 2023 - 2024**

**Table of Content**

[***Task 1*** 3](#_Toc154086025)

[***Task 2*** 8](#_Toc154086026)

[***Task 3*** 11](#_Toc154086027)

[***Task 4*** 13](#_Toc154086028)

[***Task 5*** 16](#_Toc154086029)

[***References*** 19](#_Toc154086030)

# ***Task 1***

***Round Robin***

**Gantt Chart:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Process | P1 | P2 | P3 | P4 | P1 | P2 | P3 | P3 |
| Execution Time | 0 4 | 4 8 | 8 12 | 12 16 | 16 20 | 20 22 | 22 26 | 26 28 |

**P1:**

**P2:**

**P3:**

**P4:**

**Averages:**

***Priority Scheduling (Preemptive)***

**Gantt Chart:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | P1 | P2 | P3 | P1 | P4 |
| Execution Time | 0 1 | 1 7 | 7 17 | 17 24 | 24 28 |

**P1:**

**P2:**

**P3:**

**P4:**

**Averages:**

***Shortest Job First (Preemptive)***

**Gantt Chart:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | P1 | P2 | P4 | P1 | P3 |
| Execution Time | 0 1 | 1 7 | 7 11 | 11 18 | 18 28 |

**P1:**

**P2:**

**P3:**

**P4:**

**Averages:**

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

**Gantt Chart:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | P1 | P4 | P2 | P3 |
| Execution Time | 0 8 | 8 12 | 12 18 | 18 28 |

**P1:**

**P2:**

**P3:**

**P4:**

**Averages:**

# ***Task 2***

The code illustrates a solution to the producer-consumer problem using shared memory in C. It demonstrates how multiple processes can safely access and modify a shared data structure, in this case, a circular buffer, to synchronize their actions.

**Shared Buffer Structure**

The shared\_buffer structure is defined to hold an array buffer of size BUFFER\_SIZE (set to 5), which acts as the circular buffer. The in and out indices track the next positions for inserting and removing items, respectively. Additionally, it tracks the number of items produced and consumed.

**Producer Function**

The producer function takes a shared\_buffer pointer and a producer\_id as arguments. Depending on the producer\_id, it produces items in a defined range (1 to 3 or 4 to 6) and places them in the buffer. The producer uses a busy-wait loop to handle the scenario where the buffer is full, and sleeps for 2 seconds after producing an item.

**Consumer Function**

Similarly, the consumer function, with a shared\_buffer pointer and a consumer\_id, continuously consumes items from the buffer. It also uses a busy-wait loop to handle an empty buffer situation and sleeps for 3 seconds after consuming an item.

**Main Function**

* **Initialization:** The program starts by asking the user for the number of items each producer should produce.
* **Shared Memory Segment:** It then creates a shared memory segment and attaches it to the shared\_buffer structure.
* **Process Forking:** The main function forks four child processes, two of which run the producer function and the other two run the consumer function.
* **Synchronization:** Each child process independently produces or consumes items, synchronizing access to the shared buffer using the in and out indices.
* **Completion:** The main process then becomes a consumer, to ensure the utilization of all processes.

**Parent-Child Relationship**

* **Main Process (Parent) (Consumer 3):**
* **Function:** Initializes shared memory, forks child processes, be used later on as a consumer.
* **Responsibilities:**
* Creates and initializes the shared buffer.
* Forks child processes to handle production and consumption.
* Become the 3rd consumer.
* **Child Processes:**
* **Producers (Child Processes):**
  + **Producer 1:**
    - Forged from the main process.
    - Produces items in the range 1 to 3.
  + **Producer 2:**
    - Forged from the main process.
    - Produces items in the range 4 to 6.
* **Consumers (Child Processes):**
  + **Consumer 1:**
    - Forged from the main process.
    - Consumes items from the shared buffer.
  + **Consumer 2:**
    - Forged from the main process.
    - Consumes items from the shared buffer.
* **Relationship:**
* The main process acts as the parent, creating and managing the shared buffer and control logic.
* The child processes, both producers and consumers, are forked by the main process.
* Each child process operates independently, either producing or consuming items, and synchronizes with others via the shared buffer.
* The main process becomes the 3rd consumer itself, thus ensuring that all processes are utilized.

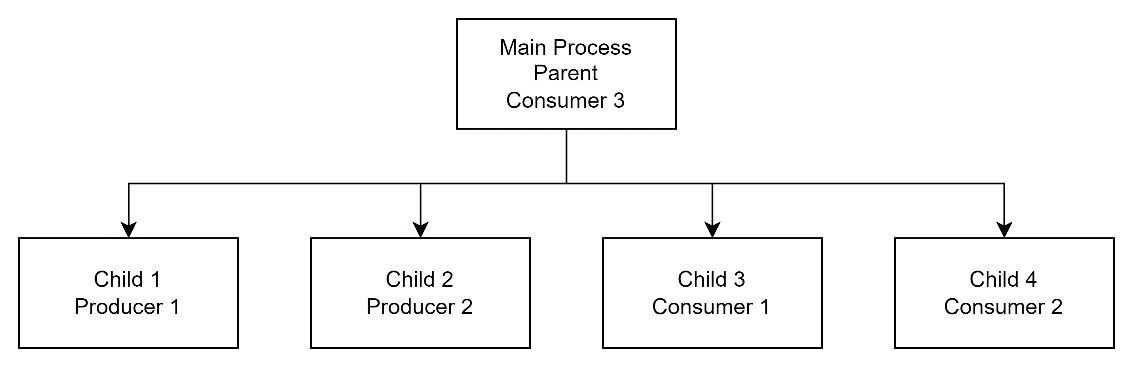


Figure 1 - Parent-Child Relationship

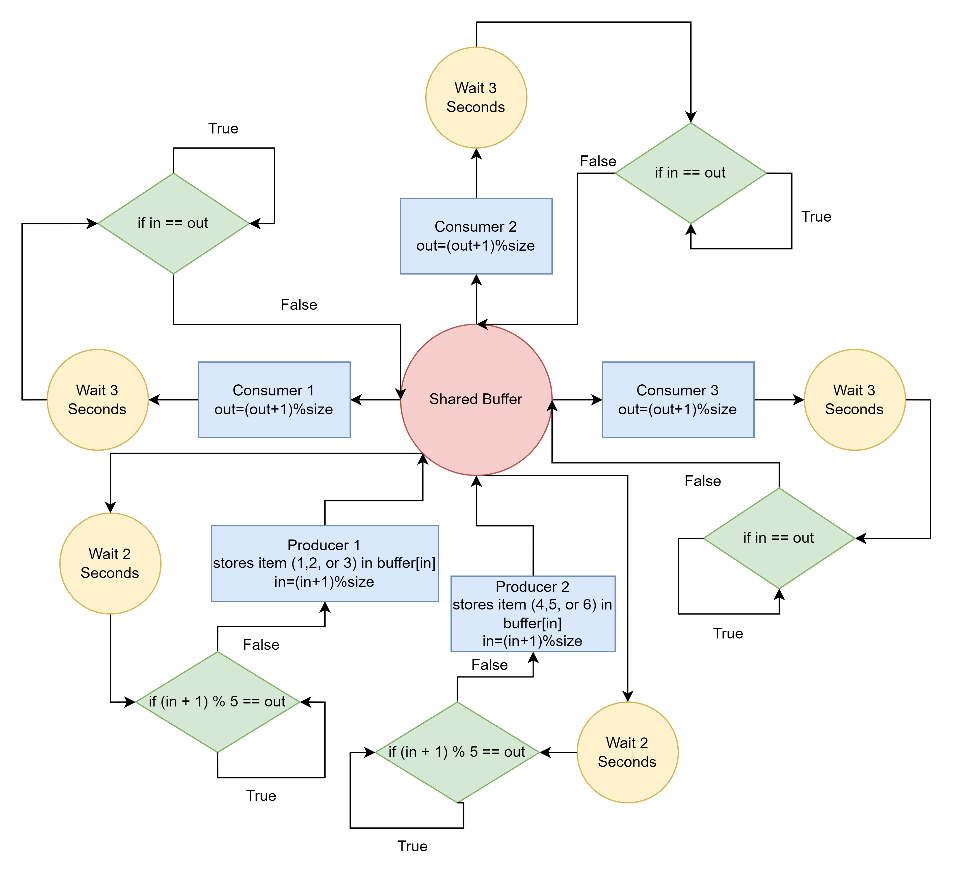


Figure 2 - Producer-Consumer Synchronization with Shared Buffer

Figure 1 depicts the parent-child relationship between all processes, with the main process producing four children using the fork() system call. If the child process was produced in the first two iterations, it is a producer; otherwise, it is a consumer. The main process then becomes a consumer as well.

The chart illustrates a synchronization scenario referred to as the producer-consumer problem. It shows a shared buffer that acts as an intermediary storage area between producers and consumers. Producers are responsible for generating items and placing them into this buffer, and they must check if there is room available before doing so. Conversely, consumers remove items from the buffer, ensuring that it is not empty when they attempt to do so. The synchronization is visualized through the producers and consumers engaging in busy-waiting loops, ensuring they only interact with the buffer when it is in a state to be read or written to. Index management is also a crucial part of this process, with both parties updating the in and out pointers of the buffer after each operation, employing modulo arithmetic to maintain the circular buffer structure.

However, the charts does not include several aspects of a complete system. The initialization of the shared buffer, which is done by the main process, including the setting of its size and the initialization of index pointers, is not described. The chart simplifies the complex nature of a producer-consumer system, focusing primarily on the logical flow and interaction between producers and consumers. It abstracts away from the detailed setup and management that is present in the actual implementation, presenting a cleaner but less detailed view of the system’s operation.

# ***Task 3***

***(Management et al., 2015; Silberschatz, Galvin and Gagne, 2017; Scaler Topics, 2023; Tutorialspoint, 2023)***

**Sequence of actions:**

1. **Device Driver:** A device driver works as a bridge between the OS and a hardware device. It is a program that provides a communication channel, allowing the OS to interact easily with the device without requiring detailed knowledge of its specifications. The driver manages tasks such as initiating I/O operations and overseeing the device’s resource utilization. Its main role is to offer an interface for the OS to interact with different hardware components.
2. **Interrupt Signal:** The device driver will interrupt the CPU when it wants to transfer data (i.e. the data is ready in the device driver’s local buffer). Cases of device drivers generating an interrupt: if the device has requested input, if the output is complete, or in case of an error. This will send a signal to the CPU notifying it that the operation has been completed and requires attention from it. The OS interrupts through sending signal to the CPU which tells the CPU to stop what it is doing and handle the device request (which involves saving the state of the process that is in progress, and context switching to the new process).
3. **Interrupt Vector Table:** When the CPU receives the interrupt signal, it doesn’t know which specific event triggered it. It then uses the Interrupt Vector Table, which is a table of addresses pointing to different Interrupt Service Routines (ISRs). The OS configures this table so that each type of interrupt is associated with the address of the corresponding ISR. Based on the type of interrupt signal received, the IVT identifies the corresponding ISR address, where each ISR is responsible for handling specific types of interrupt events.
4. **Interrupt Service Routine:** The CPU transfers control to the ISR.It will process the data from the I/O operation, manage the interrupt, and do any necessary steps to service the interrupt, such as reading data from or writing data to the device, using the ISR. When the interrupt handler is done, control is returned to the CPU.
5. **CPU resumes processing of interrupted task:** The CPU continues the execution of the program that was interrupted by the interrupt signal.

The operating system plays a vital role in interrupt handling. It is responsible for maintaining the IVT and dispatching the appropriate interrupt handler when an interrupt signal is received. The operating system also provides a mechanism for device drivers to register themselves and handle interrupts. Here is a more detailed description of the OS role in each step:

1. **Device driver:** The OS is responsible for loading and managing device drivers. The OS also grants APIs to device drivers to allow them to communicate with the computer’s OS. Moreover, resources that are necessary for I/O initiation such as memory, access to the device registers and others are provided by the operating system.
2. **Interrupt Signal:** Interrupt signals handling is also done by an OS which requires a context switching from a current task, which includes saving the process state, and then transition the control into interrupt handler.
3. **Interrupt Vector Table:** The OS not only maintains the IVT, it also dynamically updates it to accommodate new devices or changes in the system configuration. This includes adding new interrupt service routines as new hardware is installed or updating the table when devices are removed or drivers are updated. When the device generates an interrupt signal, the CPU looks up the corresponding address in the IVT and transfers control to the ISR. Additionally, the OS ensures that the addresses in the IVT are correct and that there are no conflicts. A misconfigured IVT can lead to system crashes or improper handling of interrupts.
4. **Interrupt Service Routine:** The OS dispatches the appropriate ISR which is responsible for servicing the device request and after completion, returning control to the CPU. The OS also provides a set of APIs that ISRs can use to communicate with the OS. For example, ISRs can use the OS to allocate and deallocate memory, send and receive messages, and schedule tasks. The OS also provides the ISR with the resources it needs to service the device request, such as access to the device’s registers and memory.

**Example:**

A user is typing on a keyboard. The keyboard driver initiates I/O by sending an interrupt signal to the CPU to let it know that a new key has been pressed. The CPU receives the interrupt signal and transfers control to the keyboard ISR. Upon receiving this signal, the CPU pauses its current tasks and consults the IVT. The CPU uses the IVT to find the address of the keyboard ISR. Once located, the CPU transfers control to this ISR. The keyboard’s ISR reads the key that was pressed and sends the corresponding character code to the operating system. The operating system then displays the character on the screen.

The operating system plays a vital role in interrupt handling. It is in charge of loading and managing device drivers, processing interrupt signals, creating and maintaining the IVT, and providing APIs for ISRs to communicate with the OS. Interrupt handling would be impossible without the operating system.

# ***Task 4***

***(Silberschatz, Galvin and Gagne, 2017; GeeksforGeeks, 2018; Javatpoint, 2020; Sharma, 2021; Kumar, 2023)***

**Evolution of Operating System Schedulers: Multilevel Queue and Multilevel Feedback Queue**

Operating system schedulers are critical in controlling and optimizing CPU resource use in a multiprogramming context; therefore, the evolution of the operating system scheduler represents a significant advancement in how computer systems manage processes and tasks. Initially, traditional scheduling methods, such as Round Robin (RR), Shortest Job First (SJF), and Priority Scheduling, have been developed to solve restrictions and increase overall system efficiency. However, as systems became more complex, the need for a more dynamic approach became evident. The Multilevel Queue scheduling was introduced to cater to this need, segmenting tasks into different queues based on priority or task type where each queue would have its own scheduling algorithm which model allowed for greater flexibility and responsiveness, ensuring that critical tasks received the appropriate amount of CPU time while still maintaining overall system efficiency.

The Multilevel Feedback Queue scheduling further evolved from MLQ by adding the capability to adjust the priority of tasks dynamically. The evolution from simple, static schedulers to MLQ and then to the more dynamic MFQ represents a continual effort to optimize resource allocation, reduce response time, and improve the overall efficiency of computing systems, reflecting the growing complexity and demands of modern software and hardware environments.

**Multilevel Queue Scheduling (MLQ)**

MLQ introduces a hierarchical structure to the ready queue, dividing it into multiple queues with different priority levels. Processes are assigned to queues based on their priority, with higher-priority queues receiving preferential access to CPU resources. This approach aims to differentiate between processes based on their resource requirements and importance, ensuring that critical processes receive the necessary attention.

**Benefits of MLQ:**

* **Improved Response Time for Critical Processes:** Higher-priority processes, such as system tasks and interactive applications, receive preferential treatment, ensuring their responsiveness.
* **Reduced Average Waiting Time:** By prioritizing critical processes, MLQ can reduce the average waiting time for all processes, leading to a smoother overall system experience.
* **Fairness for Different Process Types:** MLQ provides a mechanism for balancing the needs of different types of processes, ensuring that resource-intensive processes don’t monopolize the CPU.

**Multilevel Feedback Queue (MLFQ)**

MLFQ builds upon MLQ by introducing a dynamic feedback mechanism. It continuously monitors the CPU usage behavior of processes and adjusts their priority accordingly. This dynamic approach aims to adapt to the changing resource demands of processes, ensuring efficient CPU utilization.

**Benefits of MLFQ:**

* **Adaptability to Process Behavior:** MLFQ’s feedback mechanism allows it to adapt to the changing resource needs of processes, preventing resource starvation and improving overall system efficiency.
* **Improved Fairness and Performance:** By dynamically adjusting priorities, MLFQ can improve fairness among processes and overall system performance.
* **Reduced Context Switching Overhead:** By prioritizing processes that have recently used CPU time, MLFQ can reduce context switching overhead and improve system responsiveness.

**Comparison of Traditional and Multilevel Schedulers**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Feature | Round Robin (RR) | Shortest Job First (SJF) | Priority Scheduling | Multilevel Queue (MLQ) | Multilevel Feedback Queue (MLFQ) |
| Queue Structure | Single queue | Single queue | Single queue | Multiple queues with fixed priorities | Multiple queues with dynamic priorities |
| Process Movement | Processes move cyclically | Processes move based on burst time | Processes move based on priority | Processes permanently assigned to queues | Processes move between queues based on feedback |
| Adaptability | Limited adaptability | Not adaptable to process behavior | Adaptable to priority changes | Adaptable to process behavior and priority changes | Highly adaptable to process behavior and priority changes |
| Fairness | Fair for processes with equal job lengths | Can lead to starvation for longer processes | Prioritizes higher-priority processes | Fair for different process types | Fair and adaptable to changing process needs |
| Performance | Average performance | Optimal performance for known job lengths | Good performance for priority-based tasks | Improved performance for critical processes | Improved overall system performance |
| Context Switching Overhead | High context switching overhead | High context switching overhead | Moderate context switching overhead | Moderate to low context switching overhead | Low context switching overhead |

# ***Task 5***

***(HubSpot, 2022; Bhargav, 2023; GeeksforGeeks, 2023b, 2023a; Javatpoint, 2023; Stephanie, 2023)***

***Without Dynamic Linking and Dynamic Loading***

1. **Compilation**

* **All Functions:** During the compilation of program1.c and program2.c, the compiler encounters these functions and identifies them as external. It does not embed the actual code of these functions in the compiled object files. Instead, it uses symbolic addresses or placeholders. These placeholders are critical for the next stage of the build process, as they indicate to the linker that these symbols need to be resolved. The compiler’s role is limited to preparing these object files with these unresolved references, ensuring that the actual resolution of where these functions reside in memory is deferred to the linking stage.

1. **Linking**

* **All Functions:** In this phase, the linker takes the compiled object files and resolves all symbolic references to external functions by searching for their definitions in the provided libraries or object files. For each of these functions (scanf, function1, etc.), the linker finds the actual code and includes it directly into the final executable. This leads to a complete executable, but it also results in larger executable sizes, as all the code for these functions, whether they are executed or not, is included. The problem is that even if certain functions (like function2 in program1.c when n > 0) are never called during a particular execution of the program, their code is still part of the executable and takes up space.

1. **Loading**

* **All Functions:** When the program starts, the operating system’s loader loads the entire executable into memory. This includes all the statically linked code of the external functions. The consequence of this approach is a significant memory footprint, especially for larger applications or those that rely on many external functions. Every function that was statically linked, regardless of its use during the program’s runtime, occupies memory. This can be particularly inefficient in scenarios where only a subset of the available functions is regularly used.

1. **Execution**

* **Execution of Called Functions:** For functions that are actually called during execution, the process is straightforward. Since their code is already loaded into memory, the program execution can proceed without delay. There is no runtime overhead for locating and linking these functions.
* **Memory Occupancy by Uncalled Functions:** However, the downside is evident in the case of uncalled functions. These functions, despite being part of the executable, do not contribute to the program’s functionality in a given run but still occupy precious memory space. This situation is a clear inefficiency, particularly in memory-constrained environments or when the unused portions of the code are substantial.

***With Dynamic Linking and Dynamic Loading***

1. **Compilation**

* **All Functions:** The compilation process for dynamic linking/loading is similar to static linking/loading in that the compiler still identifies external functions and leaves placeholders for them. The key difference is the nature of how these placeholders will be dealt with during the linking stage. They are designed to be resolved dynamically at runtime, not statically at link time. The compiler’s output, therefore, remains a set of object files with unresolved references to external functions, awaiting the dynamic linker/loader’s intervention during program execution.

1. **Linking**

* **User-Defined Functions (function1, function2, function3, function 4):** In dynamic linking, the linker’s job changes significantly. Instead of resolving all external function references by including their code in the executable, the linker includes references for these functions. This approach results in much smaller executables, as they contain only the necessary program code and references to external functions, not the functions’ actual code. These references act as a bridge between the program and the dynamic linker/loader, enabling the latter to resolve these references at runtime. However, if the function was a shared function between processes, when using dynamic linking, it checks whether the function was already loaded on the memory by another process and links it if it was and loads it if not.
* **scanf:** If another process has already loaded the scanf function, by using dynamic linking the operating system allows the new process to use the already-loaded library. This shared use optimizes memory utilization. If it’s not already loaded, the dynamic loader loads the scanf function into memory during execution and resolves scanf’s reference. In the case of dynamic linking, the linker sets up references in the executable for dynamically linked libraries. It does not include the actual code of these libraries in the executable; instead, it prepares the program to find and link these libraries at runtime.

1. **Loading**

* **All Functions:** The operating system’s loader loads the executable into memory, but the crucial difference here is that the executable is much lighter. It contains only the program’s own code and the references for external functions. The actual code for functions remains in their respective dynamic libraries or files, not in the executable itself. This approach leads to a more efficient use of memory, as only the required parts of the program are loaded, and the rest of the functions are loaded when needed/if needed during execution.

1. **Execution**

* **User-Defined Functions (function1, function2, function3, function 4):** These functions are dynamically loaded into memory upon their first invocation. By doing this, memory is guaranteed to be used just for operations that are actually required for that specific program run.
* **scanf:** During the first call to scanf, if it’s not already loaded, the dynamic linker/loader loads the C standard library into memory and resolves scanf’s reference. If already loaded, it simply points to the existing code in memory.

***Summary***

* **Usage of scanf with Dynamic Linking:** Dynamic linking can potentially save storage space in our programs by utilizing an already loaded scanf function from another process. If scanf is not pre-loaded by any process, it will be loaded when required. This approach conserves storage space in both program1.c and program2.c, as they don’t need to include separate copies of scanf.
* **User-Defined Functions with Dynamic Loading:** In program1.c, dynamic loading enables loading only one of the two functions (function1 or function2) based on conditional execution, rather than loading both. This results in storage space savings. For program2.c, storage space efficiency varies; function3 may or may not be loaded depending on an ‘if’ condition, but function4, being independent of any condition, will always be loaded. This approach optimizes storage space usage by only loading necessary functions.

# ***References***

Bhargav, N. (2023) *Static vs. Dynamic Linking, Baeldung on Computer Science.* Available at: <https://www.baeldung.com/cs/static-dynamic-linking-differences>

GeeksforGeeks (2018) *Multilevel Queue Scheduling, GeeksforGeeks.* Available at: <https://www.youtube.com/watch?v=_Kcqumoy4VY>

GeeksforGeeks (2023a) *Difference Between Static and Dynamic Loading in Operating System - GeeksforGeeks, GeeksforGeeks.* Available at: <https://www.geeksforgeeks.org/difference-between-static-anddynamic-loading-in-operating-system/>

GeeksforGeeks (2023b) *Static and Dynamic Linking in Operating Systems, GeeksforGeeks.* Available at: <https://www.geeksforgeeks.org/static-and-dynamic-linking-in-operating-systems/>

HubSpot (2022) *Static Versus Dynamic Linking: When to Use Each, HubSpot.* Available at: <https://blog.hubspot.com/website/static-vs-dynamic-linking>

Javatpoint (2020) *Multilevel Queue Scheduling in Operating System, javatpoint.* Available at: <https://www.javatpoint.com/multilevel-queue-scheduling-in-operating-system>

Javatpoint (2023) *Static and Dynamic Loading in Operating System, javatpoint.* Available at: <https://www.javatpoint.com/static-and-dynamic-loading-in-operating-system>

Kumar, A. (2023) *Multilevel Feedback Queue Scheduling, Coding Ninjas.* Available at: <https://www.codingninjas.com/studio/library/multilevel-feedback-queue-scheduling>

Management, M. et al. (2015) *Operating System - Quick Guide, TutorialsPoint.* Available at: <https://www.tutorialspoint.com/operating_system/os_quick_guide.htm>

Scaler Topics (2023) *Introduction to Interrupts in OS, Scaler Topics.* Available at: <https://www.scaler.com/topics/operating-system/interrupts-in-os/>

Sharma, A. (2021) *Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling, Geeks for Geeks.* Available at: [https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/%0Ahttps://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/?ref=rp](https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/%0Ahttps:/www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/?ref=rp)

Silberschatz, A., Galvin, P.B. and Gagne, G. (2017) *‘Operating System Concepts Essentials’, WILEY [Preprint].*

Stephanie (2023) *Static Loading vs. Dynamic Loading: The Surprising Forces in Warehouses –, Quality Material Handling Inc. Blog.* Available at: <https://www.qmhinc.com/static-loading-vs-dynamic-loading-the-surprising-forces-in-warehouses/>

Tutorialspoint (2023) *What is Interrupt I/O Process?, TutorialsPoint.* Available at: [https://www.tutorialspoint.com/what-is-interrupt-i-o-process#:~:text=An interrupt I%2FO is,the attention of the CPU](https://www.tutorialspoint.com/what-is-interrupt-i-o-process%23:~:text=An%20interrupt%20I%2FO%20is,the%20attention%20of%20the%20CPU)