MULTILEVEL QUEUE (MLQ) CPU SCHEDULING MODULE

&

MITIGATING CPU STARVATION

Table of Contents

1. INTRODUCTION

A Multilevel Queue (MLQ) CPU Scheduling module is a component of an operating system that manages the allocation of CPU time to different processes based on their priority levels. In a multilevel queue scheduling algorithm, processes are divided into multiple queues, each with a different priority level.

Considering an example of a multilevel queue-scheduling algorithm with five queues:

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes
4. Batch Processes
5. Student Processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue. could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

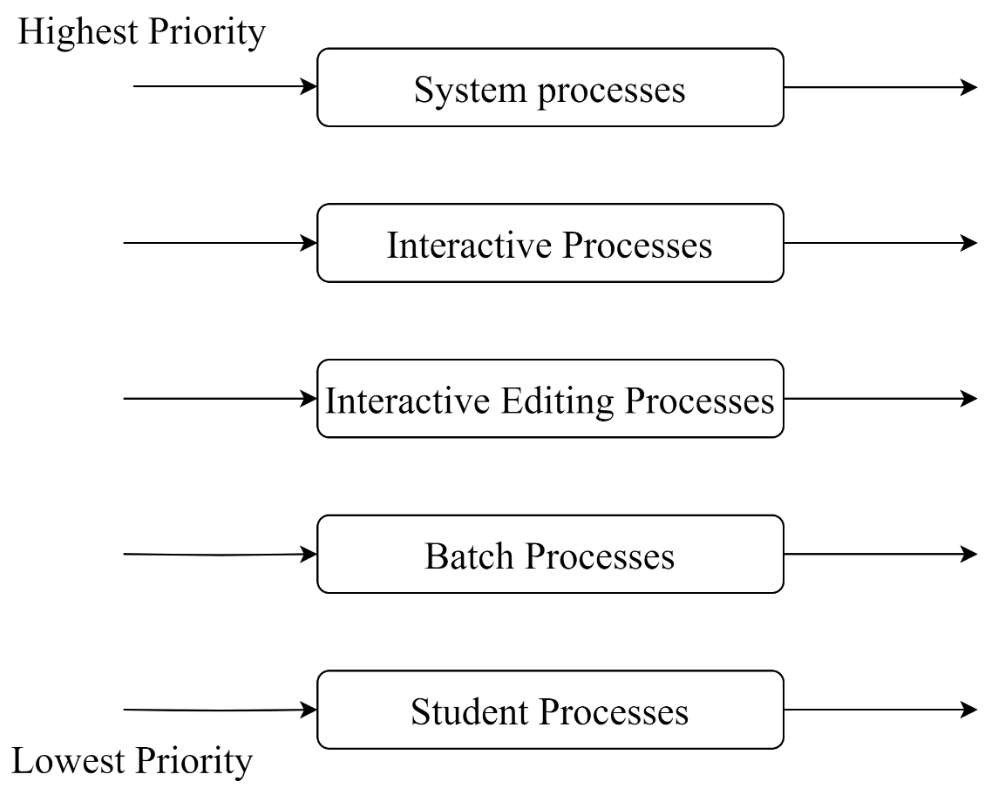


Figure 1.1 - Multilevel queue-scheduling algorithm

In this case, if there are no processes on the higher priority queue only then the processes on the low priority queues will run. Once processes on the system queue, the Interactive queue, and the Interactive editing queue become empty, only then the processes on the batch queue will run.

The Description of the processes in the above diagram is as follows:

- **System Process** - The Operating system itself has its own process to run and is termed as System Process.
- **Interactive Process**- The Interactive Process is a process in which there should be the same kind of interaction (basically an online game).
- **Batch Processes**- Batch processing is basically a technique in the Operating system that collects the programs and data together in the form of a batch before the processing starts.
- **Student Process**- The system process always gets the highest priority while the student processes always get the lowest priority.

The module handles the following tasks:

1. Process Arrival- When a new process enters the system, the MLQ module determines its priority level and places it in the corresponding queue.

2. Queue Selection- The MLQ module selects the highest priority queue that has processes waiting to be executed. The selection can be based on various criteria, such as round-robin, first-come-first-serve, or priority-based scheduling.

3. Process Execution- The MLQ module allocates the CPU to the process selected from the queue. The process is executed until it completes its execution or is interrupted by a higher-priority process.

4. Priority Adjustment- The MLQ module dynamically adjust the priority level of a process based on its behavior or other system conditions. A process that consumes excessive CPU time may be demoted to a lower-priority queue to give other processes a chance to execute.

5. Aging- The MLQ module implements aging mechanisms to prevent starvation of lower-priority processes. Aging involves periodically increasing the priority of processes in lower-priority queues to ensure they eventually get CPU time.

6. Context Switching- The MLQ module manages context switching between processes in different queues. When a higher-priority process becomes ready, the module suspends the execution of the currently running process and switches to the higher-priority process.

7. Queue Maintenance- The MLQ module ensures that each queue is managed efficiently, including handling process arrivals and departures, managing waiting times, and maintaining queue structures.

8. Scheduling Policies- The MLQ module allows the system administrator to define the scheduling policies for each queue, including the quantum time for round-robin scheduling, priority levels, and other parameters.

The MLQ CPU Scheduling module aims to provide an efficient and fair allocation of CPU time to processes with different priority levels, ensuring optimal system performance and responsiveness.

While MLQ CPU scheduling offers advantages such as improved responsiveness and fairness, it also has a potential drawback known as CPU starvation. CPU starvation occurs when processes with lower priorities continuously occupy the CPU, leading to higher-priority processes waiting for extended periods or even indefinitely. To mitigate the issue of CPU starvation in an MLQ system, it is important to implement mechanisms that ensure higher-priority processes receive adequate CPU time. This can be achieved by dynamically adjusting the priority of processes based on their waiting time or other criteria. By periodically boosting the priority of processes that have been waiting for a long time, can prevent them from being starved and ensure a fair distribution of CPU resources. Implementing this MLQ CPU scheduling module with CPU starvation mitigation aims to improve the overall efficiency and fairness of process execution in an operating system. This can lead to better system performance and a more equitable allocation of CPU resources, ensuring that no process is excessively starved of CPU time.

2. REQUIREMENTS, ASSUMPTIONS AND JUSTIFICATIONS

I. REQUIREMENTS

1. The code should be able to schedule multiple processes using the Multilevel Queue (MLQ) CPU Scheduling algorithm.

2. The code should calculate the waiting time and turnaround time for each process.

3. The code should implement an aging mechanism to mitigate CPU starvation.

II. ASSUMPTIONS

1. The user will input the number of processes and their respective burst times.

2. The user will provide a system/user process indicator for each process (0 for system process, 1 for user process).

3. The waiting time threshold for priority boosting is set to 5 units of time.

4. The processes are initially sorted based on the system/user process indicator, with system processes given higher priority.

III. JUSTIFICATIONS FOR THE ASSUMPTIONS

1. The number of processes and their burst times need to be provided by the user as they can vary in different scenarios.

2. The system/user process indicator is required to differentiate between system processes and user processes, as they may have different priorities.

3. The waiting time threshold for priority boosting is set to 5 units of time as an arbitrary value. This value can be adjusted based on specific requirements and observations of the system's behavior.

4. Sorting the processes based on the system/user process indicator, with system processes given higher priority, is a common approach. This assumption allows for a basic prioritization mechanism in the absence of more specific requirements.

IV. SPECIFICATIONS

1. The code will calculate the waiting time and turnaround time for each process using the MLQ CPU Scheduling algorithm.

2. The code will implement an aging mechanism to periodically boost the priority of lower-priority processes that have been waiting for a significant amount of time.

3. The code will sort the processes based on the system/user process indicator and the priority level, giving precedence to processes with higher priority levels.

4. The code will display the system/user process indicator, priority level, burst time, waiting time, and turnaround time for each process.

5. The code will calculate and display the average waiting time and average turnaround time for all processes

3. SYSTEM DESIGN FOR THE PROPOSED SOLUTION

The system design follows a straightforward procedural approach, utilizing arrays, variables, and algorithms to implement the Multilevel Queue CPU Scheduling algorithm with an aging mechanism. It does not incorporate any specific design patterns but organizes the code into logical sections to improve readability and maintainability.

I. SOFTWARE ARCHITECTURE:

The proposed solution follows a procedural programming paradigm. The main function is used as the entry point of the program, where the user inputs the necessary information and the scheduling algorithm is executed. The code consists of several variables, loops, and conditional statements to implement the Multilevel Queue (MLQ) CPU Scheduling algorithm with an aging mechanism.

II. DESIGN PATTERNS:

The code does not explicitly use any design patterns. However, it follows a structured approach, organizing the code into logical sections such as input, sorting, calculation, and output.

III. DATA STRUCTURES:

1. Arrays: The code uses arrays to store information about the processes, including process IDs, burst times, system/user process indicators, priority levels, waiting times, and turnaround times.

2. Variables: Various variables are used to store temporary values and perform calculations during the execution of the algorithm.

IV. ALGORITHMS:

1. Multilevel Queue (MLQ) CPU Scheduling Algorithm: The code implements the MLQ algorithm to schedule processes. It assigns priority levels based on the system/user process indicator and sorts the processes accordingly. It then calculates the waiting time and turnaround time for each process.

2. Aging Mechanism: The code includes an aging mechanism to mitigate CPU starvation. It checks if a process has been waiting for a threshold time and increases its priority level if true.

3. Sorting Algorithm: The code uses a sorting algorithm to sort the processes based on the system/user process indicator and priority level. It swaps the positions of processes to ensure the desired order.

4. IMPLEMENTATION

I. SOFTWARE DEVELOPMENT PROCESS:

The software development process for this solution can follow an iterative and incremental approach. It involves understanding the requirements, designing the solution, implementing the code, testing, and making necessary modifications. The process can be repeated until the desired functionality and performance are achieved.

II. PROGRAMMING LANGUAGE:

The code can be implemented in a programming language such as C, C++, Java, or Python, depending on the developer's preference and the target platform.

III. FRAMEWORKS, TOOLS, AND TECHNOLOGIES:

1. Integrated Development Environment (IDE): An IDE like Visual Studio Code, Eclipse, or IntelliJ IDEA can be used to write, debug, and test the code efficiently.

2. Version Control System: Utilizing a version control system like Git can help track changes, collaborate with other developers, and revert to previous versions if needed.

3. Compiler/Interpreter: The code can be compiled or interpreted using the appropriate compiler or interpreter for the chosen programming language.

4. Debugging Tools: The IDE or standalone debugging tools can be used to identify and fix any issues or errors in the code.

5. Testing Framework: A testing framework like JUnit (for Java) or pytest (for Python) can be employed to write and execute unit tests to verify the correctness of the code.

6. Documentation: Tools like Markdown or a documentation generator (e.g., Doxygen) can be used to create documentation for the codebase, including explanations of the algorithms, functions, and their usage.

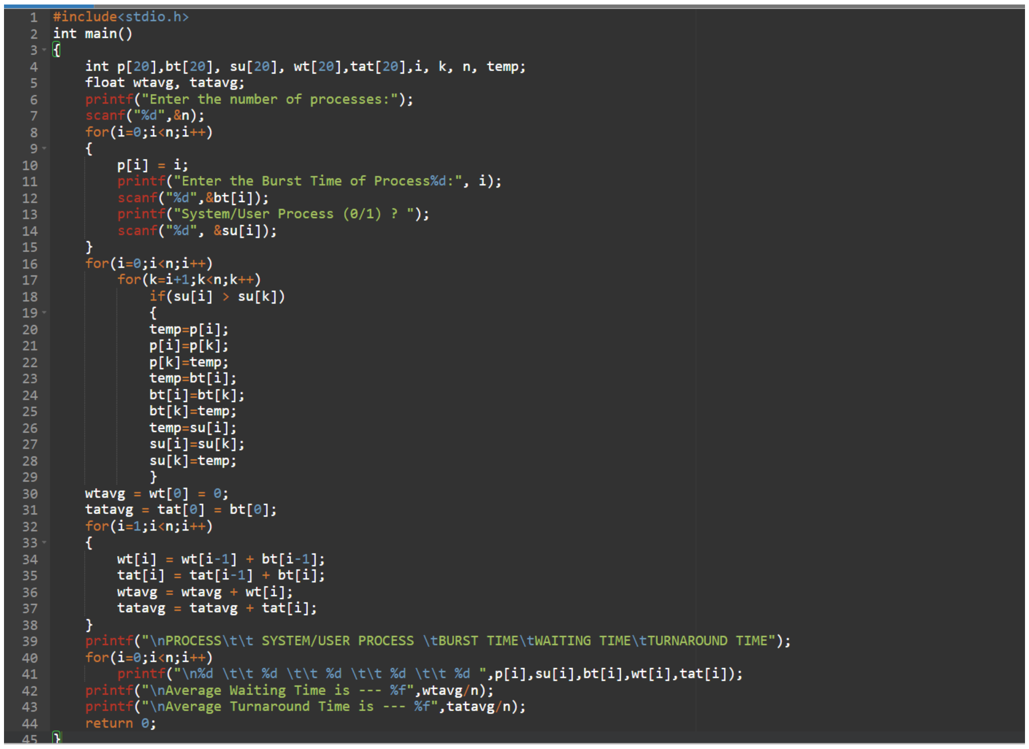
IV. THE C IMPLEMENTATION OF MULTILEVEL QUEUE SCHEDULING:

Code-

```
#include<stdio.h>
int main()
{
```

```
int p[20],bt[20], su[20], wt[20],tat[20],i, k, n, temp;
float wtavg, tatavg;
printf("Enter the number of processes:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
        p[i] = i;
        printf("Enter the Burst Time of Process%d:", i);
        scanf("%d",&bt[i]);
        printf("System/User Process (0/1) ? ");
        scanf("%d", &su[i]);
}
for(i=0;i<n;i++)
        for(k=i+1;k<n;k++)
                if(su[i] > su[k])
                {
                temp=p[i];
                p[i]=p[k];
                p[k]=temp;
                temp=bt[i];
                bt[i]=bt[k];
                bt[k]=temp;
                temp=su[i];
                su[i]=su[k];
                su[k]=temp;
                }
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
        wt[i] = wt[i-1] + bt[i-1];
        tat[i] = tat[i-1] + bt[i];
```

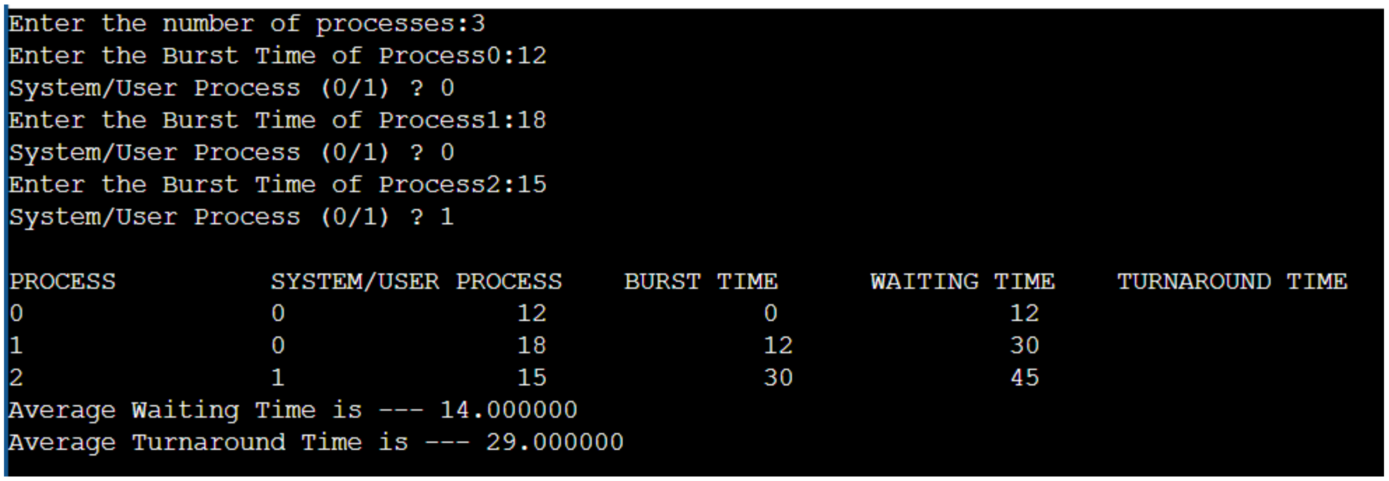
```
		wtavg = wtavg + wt[i];
		tatavg = tatavg + tat[i];
	}
	printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
	for(i=0;i<n;i++)
		printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],su[i],bt[i],wt[i],tat[i]);
	printf("\nAverage Waiting Time is --- %f",wtavg/n);
	printf("\nAverage Turnaround Time is --- %f",tatavg/n);
	return 0;
}
```



```
#include<stdio.h>
int main()
{
    int p[20],bt[20], su[20], wt[20],tat[20],i, k, n, temp;
    float wtavg, tatavg;
    printf("Enter the number of processes:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        p[i] = i;
        printf("Enter the Burst Time of Process%d:", i);
        scanf("%d",&bt[i]);
        printf("System/User Process (0/1) ? ");
        scanf("%d", &su[i]);
    }
    for(i=0;i<n;i++)
        for(k=i+1;k<n;k++)
            if(su[i] > su[k])
            {
            temp=p[i];
            p[i]=p[k];
            p[k]=temp;
            temp=bt[i];
            bt[i]=bt[k];
            bt[k]=temp;
            temp=su[i];
            su[i]=su[k];
            su[k]=temp;
            }
    wtavg = wt[0] = 0;
    tatavg = tat[0] = bt[0];
    for(i=1;i<n;i++)
    {
        wt[i] = wt[i-1] + bt[i-1];
        tat[i] = tat[i-1] + bt[i];
        wtavg = wtavg + wt[i];
        tatavg = tatavg + tat[i];
    }
    printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
    for(i=0;i<n;i++)
        printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],su[i],bt[i],wt[i],tat[i]);
    printf("\nAverage Waiting Time is --- %f",wtavg/n);
    printf("\nAverage Turnaround Time is --- %f",tatavg/n);
    return 0;
}
```

Figure 4.1- Multilevel Scheduling code

OUTPUT



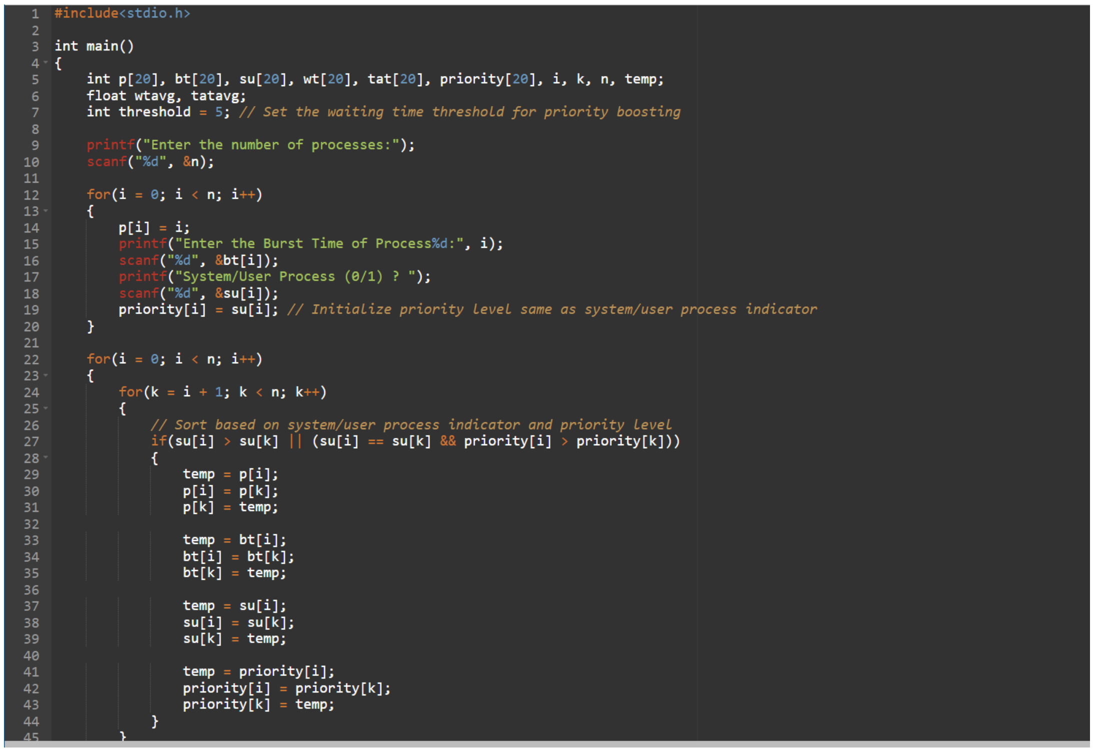
```
Enter the number of processes:3
Enter the Burst Time of Process0:12
System/User Process (0/1) ? 0
Enter the Burst Time of Process1:18
System/User Process (0/1) ? 0
Enter the Burst Time of Process2:15
System/User Process (0/1) ? 1

PROCESS          SYSTEM/USER PROCESS    BURST TIME      WAITING TIME     TURNAROUND TIME
0                0                 12              0               12
1                0                 18              12              30
2                1                 15              30              45
Average Waiting Time is --- 14.000000
Average Turnaround Time is --- 29.000000
```

Figure 4.2 – Output of the code

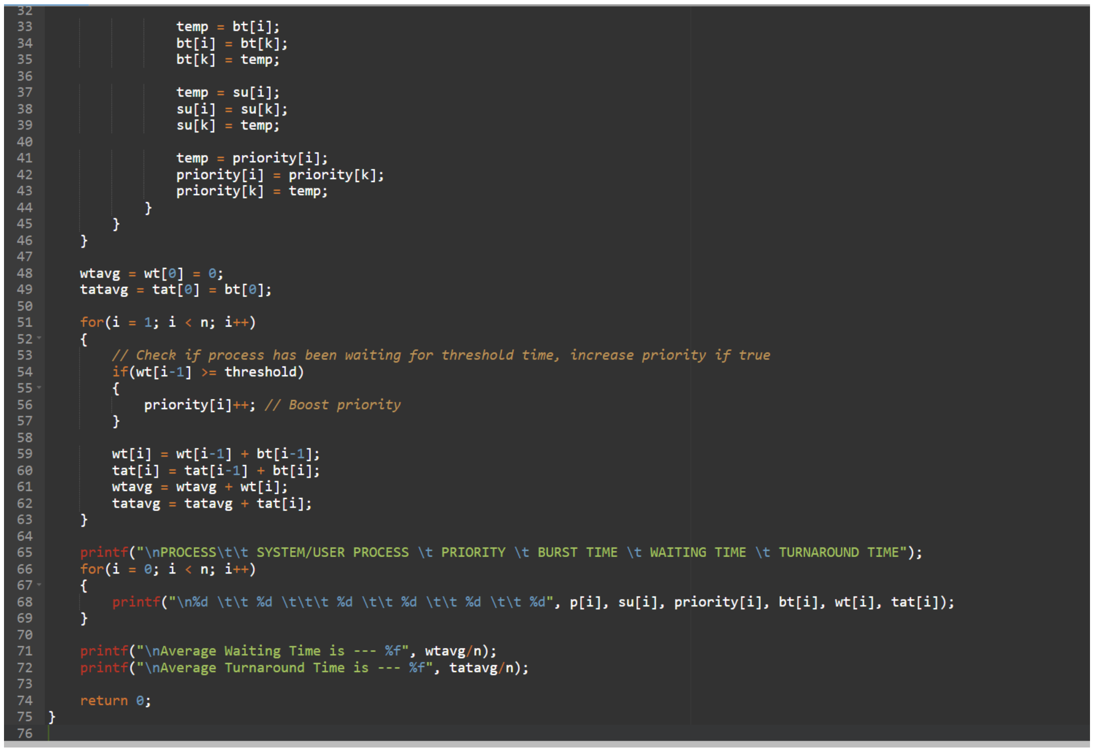
V. CODE AFTER MITIGATING THE ISSUE:

```
#include<stdio.h>


int main()
{
    int p[20], bt[20], su[20], wt[20], tat[20], priority[20], i, k, n, temp;
    float wtavg, tatavg;
    int threshold = 5; // Set the waiting time threshold for priority boosting


    printf("Enter the number of processes:");
    scanf("%d", &n);


    for(i = 0; i < n; i++)
    {
        p[i] = i;
        printf("Enter the Burst Time of Process%d:", i);
        scanf("%d", &bt[i]);
        printf("System/User Process (0/1) ? ");
        scanf("%d", &su[i]);
        priority[i] = su[i]; // Initialize priority level same as system/user process indicator
    }

    for(i = 0; i < n; i++)
    {
        for(k = i + 1; k < n; k++)
        {
            // Sort based on system/user process indicator and priority level
            if(su[i] > su[k] || (su[i] == su[k] && priority[i] > priority[k]))
            {
                temp = p[i];
                p[i] = p[k];
                p[k] = temp;

                temp = bt[i];
                bt[i] = bt[k];
                bt[k] = temp;

                temp = su[i];
                su[i] = su[k];
                su[k] = temp;

                temp = priority[i];
                priority[i] = priority[k];
                priority[k] = temp;
            }
        }
    }

    wtavg = wt[0] = 0;
    tatavg = tat[0] = bt[0];
```

```
    for(i = 1; i < n; i++)
    {
        // Check if process has been waiting for threshold time, increase priority if true
        if(wt[i-1] >= threshold)
        {
            priority[i]++; // Boost priority
        }

        wt[i] = wt[i-1] + bt[i-1];
        tat[i] = tat[i-1] + bt[i];
        wtavg = wtavg + wt[i];
        tatavg = tatavg + tat[i];
    }

    printf("\nPROCESS\t\t SYSTEM/USER PROCESS \t PRIORITY \t BURST TIME \t WAITING TIME \t
TURNAROUND TIME");
    for(i = 0; i < n; i++)
    {
        printf("\n%d \t\t %d \t\t\t %d \t\t %d \t\t %d \t\t %d", p[i], su[i], priority[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time is --- %f", wtavg/n);
    printf("\nAverage Turnaround Time is --- %f", tatavg/n);

    return 0;
}
```

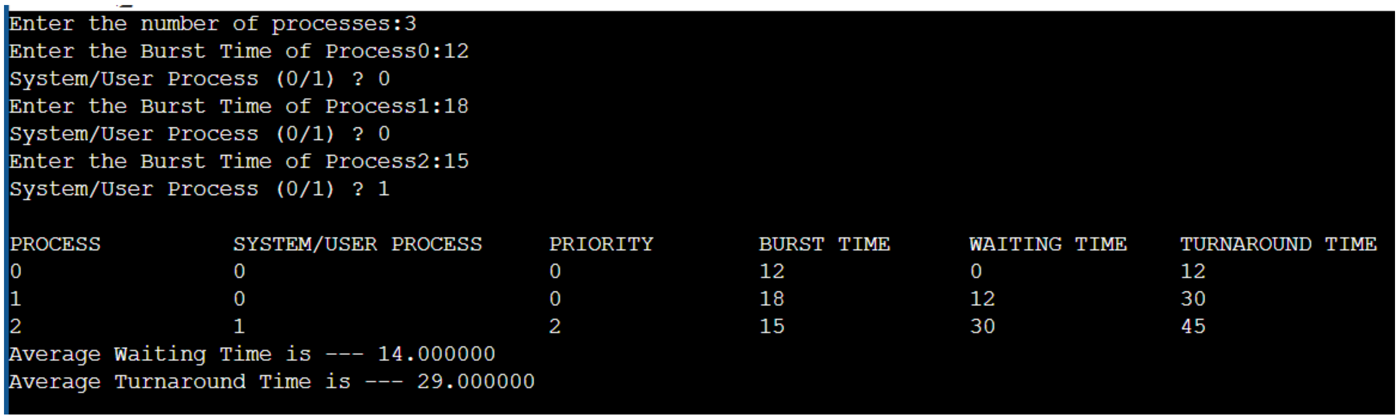


```c
#include<stdio.h>

int main()
{
    int p[20], bt[20], su[20], wt[20], tat[20], priority[20], i, k, n, temp;
    float wtavg, tatavg;
    int threshold = 5; // Set the waiting time threshold for priority boosting

    printf("Enter the number of processes:");
    scanf("%d", &n);

    for(i = 0; i < n; i++)
    {
        p[i] = i;
        printf("Enter the Burst Time of Process%d:", i);
        scanf("%d", &bt[i]);
        printf("System/User Process (0/1) ? ");
        scanf("%d", &su[i]);
        priority[i] = su[i]; // Initialize priority level same as system/user process indicator
    }

    for(i = 0; i < n; i++)
    {
        for(k = i + 1; k < n; k++)
        {
            // Sort based on system/user process indicator and priority level
            if(su[i] > su[k] || (su[i] == su[k] && priority[i] > priority[k]))
            {
                temp = p[i];
                p[i] = p[k];
                p[k] = temp;

                temp = bt[i];
                bt[i] = bt[k];
                bt[k] = temp;

                temp = su[i];
                su[i] = su[k];
                su[k] = temp;

                temp = priority[i];
                priority[i] = priority[k];
                priority[k] = temp;
            }
        }
```



```c
                temp = bt[i];
                bt[i] = bt[k];
                bt[k] = temp;

                temp = su[i];
                su[i] = su[k];
                su[k] = temp;

                temp = priority[i];
                priority[i] = priority[k];
                priority[k] = temp;
            }
        }
    }

    wtavg = wt[0] = 0;
    tatavg = tat[0] = bt[0];

    for(i = 1; i < n; i++)
    {
        // Check if process has been waiting for threshold time, increase priority if true
        if(wt[i-1] >= threshold)
        {
            priority[i]++; // Boost priority
        }

        wt[i] = wt[i-1] + bt[i-1];
        tat[i] = tat[i-1] + bt[i];
        wtavg = wtavg + wt[i];
        tatavg = tatavg + tat[i];
    }

    printf("\nPROCESS\t\t SYSTEM/USER PROCESS \t PRIORITY \t BURST TIME \t WAITING TIME \t TURNAROUND TIME");
    for(i = 0; i < n; i++)
    {
        printf("\n%d \t\t %d \t\t\t %d \t\t %d \t\t %d \t\t %d", p[i], su[i], priority[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time is --- %f", wtavg/n);
    printf("\nAverage Turnaround Time is --- %f", tatavg/n);

    return 0;
}
```

Figure 4.3 – Code after mitigating the issue



```
Enter the number of processes:3
Enter the Burst Time of Process0:12
System/User Process (0/1) ? 0
Enter the Burst Time of Process1:18
System/User Process (0/1) ? 0
Enter the Burst Time of Process2:15
System/User Process (0/1) ? 1

PROCESS          SYSTEM/USER PROCESS     PRIORITY        BURST TIME      WAITING TIME    TURNAROUND TIME
0                0                       0               12              0               12
1                0                       0               18              12              30
2                1                       2               15              30              45
Average Waiting Time is --- 14.000000
Average Turnaround Time is --- 29.000000
```

Figure 4.4- Output of the code

In this modified code, the `priority` array is introduced to store the priority level of each process. The sorting algorithm is updated to consider both the system/user process indicator and the priority level when sorting the processes. Inside the loop that calculates the waiting time and turnaround time, there is an additional check to determine if a process has been waiting for the threshold time. If it has, the priority level of that process is increased by one. The printing section is updated to display the priority level of each process along with other details. With these modifications, the code now incorporates the ability to mitigate CPU starvation by periodically boosting the priority of processes in lower-priority queues that have been waiting for a significant amount of time.

5. USER INTERFACE (UI) DESIGN

Here are some considerations for UI design:

1. Input Form: Creating an input form where users can enter the necessary information, such as the number of processes, burst times, and system/user process indicators. Using appropriate input fields, labels, and error handling to ensure accurate data entry.

2. Buttons and Controls: Include buttons for executing the scheduling algorithm and displaying the results. Provide clear labels for each button and consider using tooltips or icons for additional clarity. If needed, include checkboxes or radio buttons to enable/disable specific functionalities or options.

3. Result Display: Design a section to display the results, including the calculated waiting time and turnaround time for each process. Use a clear and organized layout, such as a table or list, to present the data. Consider using different colors or formatting to highlight important information or distinguish between different types of processes.

4. Error Handling: Implement error handling mechanisms to handle invalid inputs or unexpected errors. Display meaningful error messages to guide users in correcting their inputs or providing additional information.

5. Responsiveness: If the UI is intended to be used on different devices or screen sizes, ensure that the design is responsive and adapts to different screen sizes. Consider using responsive design techniques and frameworks to make the UI accessible on various devices.

6. Help and Documentation: Provide help text or tooltips to guide users in understanding the purpose and usage of each input field and functionality. Additionally, consider creating documentation or a user guide that explains the algorithm, assumptions, and specifications to assist users in understanding the solution.

The UI design should focus on simplicity, clarity, and ease of use to enhance the user experience and facilitate the interaction with the scheduling algorithm.

6. FUNCTIONALITY AND FEATURES

1. Process Input: Allow users to input the number of processes and their respective burst times. Provide validation to ensure that the inputs are valid integers or floats.

2. System/User Process Indicator: Provide an option for users to indicate whether each process is a system process or a user process. This information is used to assign priority levels to the processes.

3. Multilevel Queue (MLQ) Scheduling: Implement the MLQ CPU Scheduling algorithm to schedule the processes. The algorithm assigns priority levels based on the system/user process indicator and sorts the processes accordingly.

4. Waiting Time and Turnaround Time Calculation: Calculate the waiting time and turnaround time for each process based on the scheduling algorithm. The waiting time is the total time a process has spent waiting in the queue, while the turnaround time is the total time from the arrival of the process to its completion.

5. Aging Mechanism: Implement an aging mechanism to periodically boost the priority of lower-priority processes that have been waiting for a significant amount of time. This helps mitigate CPU starvation and ensures fair scheduling.

6. Result Display: Display the system/user process indicator, priority level, burst time, waiting time, and turnaround time for each process. Present the results in a clear and organized format, such as a table or list.

7. Average Waiting Time and Turnaround Time: Calculate and display the average waiting time and average turnaround time for all processes. This provides an overall measure of the efficiency of the scheduling algorithm.

8. Error Handling: Implement appropriate error handling to handle invalid inputs, such as non-numeric values or negative burst times. Display meaningful error messages to guide users in correcting their inputs.

9. User-Friendly Interface: Design a user-friendly interface that is easy to navigate and understand. Use clear labels, tooltips, and error messages to provide guidance and feedback to the users.

10. Documentation: Provide documentation or a user guide that explains the functionality, assumptions, and specifications of the solution. This helps users understand how to use the program effectively.

7. CODE STRUCTURE AND DOCUMENTATION

To ensure maintainability and readability, it is important to structure the code in a well-organized manner and provide adequate documentation. Here are some guidelines for code structure and documentation:

I. CODE STRUCTURE:

- Use meaningful variable and function names: Choose descriptive names that accurately represent the purpose and functionality of variables and functions.

- Modularize the code: Break down the code into logical modules or functions, each responsible for a specific task. This promotes code reusability and makes it easier to understand and maintain.

- Follow coding conventions: Adhere to coding conventions for the chosen programming language, such as indentation, naming conventions, and commenting styles.

- Limit code duplication: Avoid duplicating code blocks by encapsulating common functionality in reusable functions or classes.

II. DOCUMENTATION:

- Function and method documentation: Include comments before each function or method to describe its purpose, input parameters, return values, and any exceptions it may throw. This helps other developers understand how to use the code and its expected behavior.

- Inline comments: Use inline comments sparingly to clarify complex or non-obvious parts of the code. Comments should explain the reasoning behind the code or any potential pitfalls.

- Algorithm explanation: Include comments or separate documentation sections that explain the Multilevel Queue (MLQ) CPU Scheduling algorithm and the aging mechanism in detail. Describe the steps involved, any assumptions made, and the expected behavior.

- Input and output format: Document the expected format of the input (e.g., number of processes, burst times) and the format of the output (e.g., displayed table with process details and calculated metrics).

- Assumptions and limitations: Document any assumptions made during the implementation process and any limitations or constraints of the solution.

III. README FILE:

- Create a readme file that provides an overview of the project, including its purpose, functionality, and instructions for running the code.

- Include any necessary setup instructions, such as installing dependencies or configuring the environment.
- Provide examples and usage instructions to guide users on how to interact with the program effectively.
- Mention any known issues or caveats, and provide contact information for support or further inquiries.

By following these guidelines, the code structure and documentation will enhance the maintainability and understandability of the solution, making it easier for other developers to work with and for users to understand how to use the program effectively.

8. GITHUB EPOSITORY

https://github.com/PeshalaDanushki/MLQ-Scheduling.git

9. TESTING RESULTS

When testing the scheduling algorithm, the following scenarios can be considered:

1. Test with Different Input Sizes: Run the algorithm with varying numbers of processes, ranging from small to large inputs. This will help identify any performance issues or unexpected behavior that may arise when dealing with different input sizes.

2. Test with Different Burst Times: Create test cases with different burst times for the processes. This will help evaluate how the algorithm handles different process execution times and whether it produces correct results.

3. Test with Different Priority Levels: Test the algorithm with different combinations of system and user processes to assess if the priority levels are assigned correctly and if the scheduling behavior is as expected.

4. Test with Aging Mechanism: Run the algorithm for a longer duration to observe the effects of the aging mechanism. Verify if lower-priority processes are boosted appropriately and if the scheduling remains fair.

5. Test Edge Cases: Include test cases with extreme or boundary values, such as very high or very low burst times, to ensure that the algorithm handles such cases without errors or unexpected behavior.

6. Verify Correctness of Results: Validate that the waiting time, turnaround time, and average metrics calculated by the algorithm are correct for each test case.

When analyzing the testing results, the following can be considered:

1. Accuracy: Verify that the algorithm produces correct results, such as accurate waiting time, turnaround time, and average metrics for each process.

2. Performance: Assess the algorithm's performance in terms of execution time and memory usage. Ensure that it performs efficiently, even with larger input sizes.

3. Stability: Check if the algorithm consistently produces the same results for the same input, indicating stability and determinism.

4. Error Handling: Observe if the algorithm handles invalid inputs gracefully and provides meaningful error messages when necessary.

5. Compliance with Requirements: Evaluate if the algorithm meets the specified requirements and adheres to the expected behavior, such as correct priority assignment and scheduling.

By thoroughly testing and analyzing the results it can ensure that the scheduling algorithm functions correctly and meets the desired criteria.

10. DEPLOYMENT AND INSTALLATION

1. Packaging the Application: Bundle all the necessary files and dependencies of the software into a single package. This can be a compressed file or an installer package, depending on the target platform.

2. Documentation: Include clear and concise documentation that guides users through the installation process. This should include step-by-step instructions, system requirements, and any prerequisites or dependencies needed for the software to run.

3. Platform Compatibility: Ensure that the software is compatible with the target platform where it will be deployed. This includes checking the operating system version, architecture (32-bit or 64-bit), and any specific dependencies or libraries required.

4. Installation Process: Provide an intuitive and user-friendly installation process. This may involve creating a graphical installer with prompts, checkboxes for optional components, and progress indicators. Alternatively, if it is a command-line application, provide clear instructions on how to install and run the software.

5. Configuration: If the software requires any configuration, such as database connection settings or API keys, provide clear instructions on how to set up and configure these components.

6. Testing the Installation: Before releasing the software, thoroughly test the installation process on different target platforms to ensure that it works as expected. This includes verifying that all dependencies are installed correctly and that the software runs without any issues.

7. Deployment Options: Consider different deployment options based on the requirements of your software, such as local installation, cloud-based hosting, or containerization using technologies like Docker.

8. Release and Distribution: Determine how you will distribute the software to end-users. This can be done through direct downloads from a website, distribution through app stores, or integration into existing software distribution platforms.

9. Updates and Maintenance: Plan for future updates and maintenance of the software. Consider implementing mechanisms for automatic updates or providing a clear upgrade process for users.

11. CONCLUSION

In conclusion, developing a scheduling algorithm with the necessary functionality and features can greatly optimize the management of processes in a system. By implementing a Multilevel Queue (MLQ) CPU Scheduling algorithm with an aging mechanism, you can ensure fair and efficient scheduling of processes, mitigating CPU starvation and improving overall system performance.

To ensure maintainability and understandability of the code, it is important to structure the code in a modular and organized manner. Additionally, providing comprehensive documentation, including comments within the code and a separate readme file, will assist other developers in understanding and working with the code effectively.

Thorough testing of the scheduling algorithm is essential to validate its accuracy, performance, stability, and compliance with requirements. By creating various test cases and analyzing the results, you can ensure that the algorithm functions correctly and meets the desired criteria.

If applicable, packaging the software into a deployable package and providing clear installation instructions will enable users to easily install and set up the software on their systems. Considering platform compatibility, configuration requirements, and testing the installation process will ensure a smooth deployment experience for users.

With proper functionality, code structure, documentation, testing, and deployment, it can develop and deliver a robust and efficient scheduling algorithm that enhances system performance and improves the user experience.

12. FUTURE ENHANCEMENTS

1. Preemptive Scheduling: Consider adding preemption support to the scheduling algorithm. This would allow higher-priority processes to interrupt the execution of lower-priority processes, ensuring more responsive and dynamic scheduling.

2. Dynamic Priority Adjustment: Implement a mechanism to dynamically adjust process priorities based on factors such as process behavior, resource usage, or system load. This would allow for more intelligent and adaptive scheduling decisions.

3. Support for Real-Time Processes: Extend the scheduling algorithm to handle real-time processes that have strict timing constraints. This could involve incorporating priority-based scheduling policies such as Earliest Deadline First (EDF) or Rate Monotonic Scheduling (RMS).

4. Multiple CPU Support: Enhance the algorithm to support systems with multiple CPUs or cores. This could involve load balancing techniques to distribute processes across multiple CPUs, improving overall system performance.

5. Interactive Process Handling: Improve the algorithm's handling of interactive processes that require low response times. Consider implementing techniques such as Shortest Remaining Time First (SRTF) or Round Robin (RR) scheduling to ensure fair allocation of CPU time to interactive processes.

6. Energy Efficiency Considerations: Integrate energy efficiency considerations into the scheduling algorithm. This could involve optimizing resource allocation to minimize power consumption or incorporating sleep states for idle processes.

7. Visualization and Monitoring: Develop a graphical user interface or monitoring tool to visualize the scheduling algorithm's behavior and provide real-time insights into process execution, priorities, and system performance.

8. Advanced Metrics and Analysis: Extend the algorithm to calculate and provide additional performance metrics such as throughput, fairness index, and response time. This would enable more comprehensive analysis and optimization of the system's scheduling behavior.

9. Machine Learning Techniques: Explore the integration of machine learning techniques to predict process behavior and optimize scheduling decisions. This could involve using historical data or real-time monitoring to make intelligent scheduling choices.

10. Integration with Other System Components: Consider integrating the scheduling algorithm with other system components, such as memory management or I/O scheduling, to achieve better system-wide performance and resource utilization.

These suggestions aim to enhance the scheduling algorithm's functionality, adaptability, and performance in different scenarios. Prioritize the enhancements based on the specific requirements and constraints of your target system and consider the potential impact on system stability and complexity.

13. REFERENCES

https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/

https://www.studytonight.com/operating-system/multilevel-queue-scheduling

https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/

14. APPENDIX

https://www.youtube.com/watch?v=fvkSXMZaBNY

https://www.youtube.com/watch?v=JDsrcoLKKhg

https://www.youtube.com/watch?v=1S_CekP8lMw