Emir Köse
820210309
Kemal Bıçakçı
24 Mart 2025

Computer Operating Systems Homework-1 Report

Scheduling Algorithm Process

1. Introduction: Unveiling the Simulated Process Scheduler

This report presents a comprehensive exploration of a process scheduler simulation, a project developed for COS Homework-1. This simulation endeavors to model the intricate workings of a preemptive priority-based round-robin scheduling algorithm, a fundamental concept within the realm of operating systems. The core objective of this project is to create a functional representation of how a modern operating system manages and allocates CPU time to multiple concurrent processes.

The simulation operates by ingesting job specifications from an external input file, `jobs.txt`. These specifications detail the characteristics of each simulated process, including its arrival time, priority, execution time, and a unique identifier. The scheduler then orchestrates the execution of these jobs, adhering to a predefined time quantum and the principles of the preemptive priority-based round-robin algorithm.

A crucial aspect of this simulation is the meticulous logging of all scheduling events. Every context switch, process creation, termination, and state transition is recorded with precise timestamps in an output file, `scheduler.log`. This log file serves as a detailed audit trail, allowing for thorough analysis and verification of the scheduler's behavior.

This report will delve into the intricacies of the simulation, covering a wide range of topics:

- **Program Architecture and Design:** An examination of the program's structure, including the modular design and the rationale behind the chosen approach.
- **Data Structures:** A detailed analysis of the `Job` structure, which serves as the core data representation for each simulated process.
- **Function Prototypes and Implementation:** A thorough explanation of each function's purpose and implementation, including the logic behind the scheduling algorithm.
- **Data Input and Output:** A description of how the program interacts with external files, including the format of `jobs.txt` and the structure of `scheduler.log`.
- **Makefile and README:** An analysis of these supporting files and their roles in the project's build process and documentation.

- **Testing and Validation:** A discussion of the strategies employed to ensure the program's correctness and robustness.
- **Scheduling Fairness Analysis:** An evaluation of the fairness of the implemented scheduling algorithm and potential improvements.
- **Edge Cases and Failure Scenarios:** An examination of potential failure scenarios and the program's handling of these cases.
- **Discussion and Design Decisions:** A reflection on the design choices made during the project and the reasoning behind them.

This report aims to provide a comprehensive understanding of the process scheduler simulation, from its conceptual design to its practical implementation and analysis. It will serve as a detailed documentation of the project, highlighting its strengths, weaknesses, and potential areas for improvement.

2. Introduction to Process Scheduling: A Deep Dive into the OS's Orchestration

Process scheduling is not merely a technical detail; it is the very heartbeat of a modern operating system. It's the intricate dance of allocating finite CPU resources among a multitude of competing processes, all vying for execution time. This allocation must be performed with precision and efficiency, as it directly impacts the overall performance and responsiveness of the system.

At its core, process scheduling is about resource management. The CPU, the central processing unit, is a critical resource, and the scheduler acts as the traffic controller, determining which process gets to use it at any given moment. This decision-making process is guided by a set of objectives, each contributing to a well-functioning operating system:

- Maximizing Throughput: This objective aims to keep the CPU as busy as possible, minimizing idle time and ensuring that a high volume of work is completed. In real-world scenarios, this translates to more applications running smoothly and efficiently.
- Minimizing Response Time: In interactive systems, such as desktop applications or web servers, quick response times are paramount. Users expect immediate feedback, and delays can lead to frustration. The scheduler strives to minimize the time it takes for a process to produce its first output or react to user input.
- Ensuring Fairness: Fairness is about preventing any single process from monopolizing the CPU, ensuring that all processes receive a reasonable share of processing time. This is particularly important in multi-user systems, where multiple users are running applications simultaneously.
- Balancing Resource Utilization: This objective aims to distribute CPU time in a way that avoids resource starvation and maintains system stability. Resource starvation occurs when a process is perpetually denied access to the CPU, preventing it from making progress.

The simulation developed for this project focuses on a preemptive priority-based round-robin algorithm. This algorithm is a hybrid, combining the fairness of round-robin with the responsiveness of priority-based scheduling.

- Round-Robin: This approach allocates fixed-length time slices (time quanta) to each process in a circular queue. This ensures that all processes get a chance to run, preventing any single process from monopolizing the CPU.
- Priority-Based: This approach assigns priorities to processes, allowing critical tasks to be executed before less important ones. This ensures that important applications are responsive and that system-critical tasks are completed promptly.
- Preemptive: Preemption allows the operating system to interrupt a running process and switch to another process, even if the first process has not finished its time slice. This is crucial for responsiveness, as it allows high-priority tasks to interrupt lower-priority ones.

This simulation seeks to accurately model the workings of this complex scheduling algorithm. By understanding its implementation, we gain valuable insights into the fundamental principles of operating systems and their role in managing concurrent processes.

3. Program Architecture and Design: A Modular Approach for Clarity and Extensibility

The architecture of the process scheduler simulation is designed with a strong emphasis on modularity, clarity, and extensibility. This approach ensures that the codebase is easy to understand, maintain, and adapt to future modifications or enhancements. The simulation is structured into two primary files: `scheduler.c` and `scheduler.h`, each serving a distinct purpose.

3.1. The Role of scheduler.h: Defining the Interface

The `scheduler.h` file acts as the public interface, defining the data structures and function prototypes that form the building blocks of the scheduler. It serves as a contract between the implementation in `scheduler.c` and any other module that might interact with the scheduler.

- **Data Structure Definitions:** The `scheduler.h` file houses the definition of the `Job` structure, which encapsulates all the essential information about a simulated process. This structure is the fundamental unit of data within the scheduler, and its definition in the header file ensures that all parts of the program share a consistent understanding of its layout.
- **Function Prototypes:** The header file declares the prototypes of all functions that are part of the scheduler's public interface. This includes functions for logging events, managing jobs, and implementing the scheduling algorithm. By declaring these prototypes, the header file provides a clear overview of the scheduler's capabilities.
- **Constants and Macros:** Any constants or macros that are relevant to the scheduler's interface are also defined in the header file. This ensures that these values are consistently used across the program.
- **Include Guards:** The header file employs include guards (`#ifndef`, `#define`, `#endif`) to prevent multiple inclusions, which can lead to compilation errors.

3.2. The Implementation in `scheduler.c`: The Engine Room

The `scheduler.c` file contains the implementation of the functions declared in `scheduler.h`. It houses the core logic of the scheduler, including the scheduling algorithm, job management, and event logging.

- **Function Implementations:** This file provides the definitions of all functions declared in `scheduler.h`. These implementations contain the actual code that performs the scheduler's operations.
- **Local Variables and Functions:** The `scheduler.c` file may also contain local variables and functions that are used internally by the scheduler but are not part of its public interface.
- **Data Structures (Local):** Any data structures that are used exclusively within the `scheduler.c` file are defined here.
- **Algorithm Implementations:** The scheduling algorithms, including the preemptive priority-based round-robin logic, are implemented within the functions of this file.

3.3. Rationale for Modular Design

The modular design of the scheduler simulation offers several advantages:

- **Clarity and Readability:** Separating the interface from the implementation makes the code easier to understand and navigate.
- **Maintainability:** Changes to the implementation in `scheduler.c` do not require modifications to the interface in `scheduler.h`, reducing the risk of unintended side effects.
- **Reusability:** The header file can be reused in other projects that require similar scheduling functionality.
- **Testability:** The modular design facilitates unit testing, as individual functions can be tested independently.
- **Extensibility:** New features can be added by modifying the implementation in `scheduler.c` or by adding new functions to the interface in `scheduler.h`.

This modular approach ensures that the scheduler simulation is not only functional but also well-structured and maintainable, contributing to its overall quality and longevity.

3.1. Data Structures: The Job Structure - A Refined Core Entity

The Job structure, central to our process scheduler simulation, serves as the fundamental data representation for each simulated process. It encapsulates all the essential attributes required for the scheduler to manage and orchestrate the execution of concurrent jobs. With a slight adjustment to the name field, we'll explore its significance within the simulation:

```
typedef struct {
    char name[10];
    int arrival_time;
    int priority;
    int execution_time;
    int remaining_time;
    pid_t pid;
    int has_started;
} Job;
```

Let's examine each field of this structure, highlighting its role and implications:

- **name (char[10]):**
  - This field stores the name of the simulated job.
  - It is a character array with a fixed size of 10 bytes, limiting job names to 9 characters plus a null terminator.
  - This size limitation introduces a constraint on job naming conventions, potentially simplifying memory management and string handling within the simulation.
  - The name field remains crucial for human-readable identification and logging, aiding in debugging and analysis.

- **arrival_time (int):**
  - This field represents the time at which the simulated job enters the ready queue.
  - It is an integer value, allowing for precise tracking of job arrival times.
  - The arrival_time field is essential for simulating real-world scenarios where jobs arrive at varying times, influencing the scheduler's decision-making.

- **priority (int):**
  - This field specifies the priority of the simulated job, with lower integer values indicating higher priority.
  - It is an integer value, enabling the scheduler to prioritize critical jobs.

    - The priority field is fundamental to the priority-based aspect of the scheduling algorithm.
- **execution_time (int):**
    - This field represents the total CPU time required to complete the simulated job.
    - It is an integer value, simulating the job's execution duration.
    - The execution_time field is used to model job execution and track progress.
- **remaining_time (int):**
    - This field stores the CPU time remaining for the simulated job's execution.
    - It is an integer value, dynamically updated as the job executes.
    - The remaining_time field is essential for the preemptive nature of the scheduling algorithm.
- **pid (pid_t):**
    - This field stores the process ID (PID) of the forked process simulating the job's execution.
    - It is of type pid_t, a system-defined type for process IDs.
    - The pid field enables the scheduler to control the simulated job's process using system calls.
- **has_started (int):**
    - This field acts as a boolean flag, indicating wether the job has started execution.
    - It prevents double forking of the same job.
    - This field is vital for controling the flow of execution.

The refined Job structure, with its name field limited to 10 characters, continues to serve as the cornerstone of our scheduler simulation. It provides a concise and efficient means of representing simulated processes, enabling the scheduler to effectively manage their execution and track their progress.

3.2. Function Prototypes (scheduler.h) - Declaring the Scheduler's Capabilities

The `scheduler.h` file serves as the public interface for the process scheduler simulation, declaring the functions that define the scheduler's capabilities. These function prototypes act as a contract, specifying the input and output parameters of each function, allowing other parts of the program to interact with the scheduler without needing to know the details of its implementation.

- **`void log_event(const char* message);`**
    - This function is responsible for recording scheduling events in the `scheduler.log` file.

    - It takes a character string `message` as input, which describes the event to be logged.
    - Within the function, the current timestamp is retrieved, formatted, and appended to the log file along with the provided message.
    - This function plays a critical role in debugging and analyzing the scheduler's behavior, providing a chronological record of events.
    - The function implements error handling to gracefully manage situations where the log file cannot be opened, ensuring that the program continues to operate.
- **`char* get_current_time_str();`**
    - This function retrieves the current system time and returns it as a dynamically allocated character string.
    - It utilizes the `time()` and `localtime()` functions to obtain the current time and the `strftime()` function to format it into a human-readable string.
    - The dynamic allocation of memory using `malloc()` ensures that the returned string remains valid even after the function has finished executing.
    - This function is used by `log_event()` and other parts of the program that require timestamp information.
- **`void fork_and_exec(Job* job);`**
    - This function simulates the execution of a job by forking a child process.
    - It takes a `Job` structure as input, which contains the job's execution time and other relevant information.
    - The `fork()` system call is used to create a new process, and the child process then simulates job execution using the `sleep()` function.
    - The parent process logs the forking event, and stores the child processes PID into the job structure.
    - This function simulates the creation and execution of processes within the simulated environment.
- **`void handle_job_completion(Job* job);`**
    - This function handles the completion of a job by terminating its corresponding process.
    - It takes a `Job` structure as input, which contains the job's process ID.
    - The `kill()` system call with the `SIGKILL` signal is used to terminate the process.
    - The function also logs the job completion event.
    - This function ensures that completed jobs are removed from the system, freeing up resources.
- **`void stop_job(Job* job, int run_time);`**
    - This function simulates the suspension of a job by sending a `SIGSTOP` signal to its process.

    - It takes a `Job` structure and the amount of run time as input.
    - The function then logs the event.
    - This function is essential for implementing the preemptive nature of the scheduling algorithm.
- **`void resume_job(Job* job);`**
    - This function simulates the resumption of a suspended job by sending a `SIGCONT` signal to its process.
    - It takes a `Job` structure as input.
    - The function then logs the event.
    - This function allows the scheduler to resume jobs after their time slice has expired.
- **`int compare_jobs(const void* a, const void* b);`**
    - This function is a comparison function used by the `qsort()` function to sort jobs based on priority, arrival time, and remaining execution time.
    - It takes two pointers to `Job` structures as input and returns an integer value indicating their relative order.
    - This function is crucial for implementing the priority-based scheduling algorithm.
- **`int find_next_job(Job* jobs, int num_jobs, int current_time, int last_run_index);`**
    - This function implements the core job selection logic, determining which job should be executed next.
    - It takes an array of `Job` structures, the number of jobs, the current time, and the index of the last run job as input.
    - It returns the index of the next job to be executed.
    - This function ensures that the scheduler selects the most appropriate job for execution based on the scheduling algorithm.
- **`void run_scheduler(Job* jobs, int num_jobs, int time_quantum);`**
    - This function implements the main scheduling loop, managing the execution of jobs according to the preemptive priority-based round-robin algorithm.
    - It takes an array of `Job` structures, the number of jobs, and the time quantum as input.
    - It iterates through the jobs, selects the next job to execute, simulates context switching, and logs all scheduling events.
    - This is the core control loop of the program.
- **`void schedule_processes(const char* filename);`**
    - This function reads job specifications from the input file and initiates the scheduling process.

    - It takes the filename of the input file as input.
    - It parses the file, creates an array of `Job` structures, and calls the `run_scheduler()` function.
    - This function handles the beginning of the program, and cleans up at the end.

These function prototypes define the scheduler's interface, providing a clear and concise overview of its capabilities. They allow other parts of the program to interact with the scheduler in a well-defined and predictable manner.

4. Code Implementation (scheduler.c) - The Engine Room of the Scheduler

The `scheduler.c` file houses the implementation of the functions declared in `scheduler.h`, providing the concrete logic and behavior of the process scheduler simulation. Let's delve into the implementation of each function, exploring its functionality and underlying mechanisms.

- **`void log_event(const char* message):`**
    - This function initiates by opening the `scheduler.log` file in append mode (`"a"`). This ensures that each new event is added to the end of the existing log, preserving a chronological record.
    - It then retrieves the current system time using `time()` and converts it into a local time structure using `localtime()`.
    - The `strftime()` function is employed to format the time into a human-readable string, which is then combined with the provided `message`.
    - The resulting timestamped message is written to the `scheduler.log` file using `fprintf()`.
    - Crucially, the function includes error handling to gracefully manage situations where the log file cannot be opened. If an error occurs, an appropriate error message is displayed, preventing the program from crashing.
    - This function ensures that all important scheduler actions are logged.
- **`char* get_current_time_str():`**
    - This function retrieves the current system time in a dynamically allocated string.
    - It retrieves the current time, and then formats it.
    - It uses malloc to allocate memory for the string, and returns the pointer to the created string.
    - This function is used to get the current time for the log_event function.
- **`void fork_and_exec(Job* job):`**
    - This function simulates the execution of a job by forking a child process using the `fork()` system call.

    - Within the child process, the function simulates job execution by pausing the process for the specified `execution_time` using the `sleep()` function. This effectively models the CPU time consumed by the job.
    - The parent process records the forking event, including the job's name and process ID, using the `log_event()` function.
    - The `pid` field of the `Job` structure is updated with the child process's ID, enabling the parent process to control the child process later.
    - This function simulates the process running.
- **`void handle_job_completion(Job* job):`**
    - This function handles the completion of a job by terminating its corresponding process.
    - It uses the `kill()` system call with the `SIGKILL` signal to terminate the process identified by the `pid` field of the `Job` structure.
    - The job completion event, including the job's name, is logged using the `log_event()` function.
    - This function cleans up the process after it has finished.
- **`void stop_job(Job* job, int run_time):`**
    - This function simulates the suspension of a job by sending a `SIGSTOP` signal to its process.
    - It uses the `kill()` system call with the `SIGSTOP` signal to suspend the process identified by the `pid` field of the `Job` structure.
    - The function then logs the event.
    - This function is used for context switching.
- **`void resume_job(Job* job):`**
    - This function resumes a suspended job using the `SIGCONT` signal.
    - It uses the `kill()` system call with the `SIGCONT` signal to resume the process identified by the `pid` field of the `Job` structure.
    - The function then logs the event.
- **`int compare_jobs(const void* a, const void* b):`**
    - This function serves as a comparison function for the `qsort()` function, enabling the sorting of jobs based on priority, arrival time, and remaining execution time.
    - It casts the void pointers `a` and `b` to `Job` pointers, allowing access to the job attributes.
    - The function implements a multi-level comparison logic, prioritizing jobs with lower priority values (higher priority), earlier arrival times, and shorter remaining execution times.
    - The function returns an integer value indicating the relative order of the two jobs.
- **`int find_next_job(Job* jobs, int num_jobs, int current_time, int last_run_index):`**

    - This function implements the core job selection logic, determining which job should be executed next based on the scheduling algorithm.
    - It iterates through the array of `Job` structures, considering factors such as arrival time, priority, and remaining execution time.
    - The function maintains a `last_run_index` to ensure fairness and prevent starvation.
    - It returns the index of the next job to be executed.
- **`void run_scheduler(Job* jobs, int num_jobs, int time_quantum):`**
    - This function implements the main scheduling loop, orchestrating the execution of jobs according to the preemptive priority-based round-robin algorithm.
    - It maintains a ready queue, selecting jobs for execution using the `find_next_job()` function.
    - It simulates context switching by calling the `stop_job()` and `resume_job()` functions.
    - It logs all scheduling events, including job creation, execution, context switching, and completion.
    - It handles the main loop of the program.
- **`void schedule_processes(const char* filename):`**
    - This function reads job specifications from the input file and initiates the scheduling process.
    - It opens the input file in read mode and parses its contents, extracting job attributes such as name, arrival time, priority, and execution time.
    - It dynamically allocates memory for an array of `Job` structures to store the job information.
    - It calls the `run_scheduler()` function to begin the scheduling process.
    - It closes the file, and frees the allocated memory.

5. Data Input and Output: The Interface with the External World

The process scheduler simulation interacts with the external world through two primary files: `jobs.txt` for input and `scheduler.log` for output. These files serve as the channels through which the simulation receives job specifications and communicates its scheduling decisions and events.

5.1. `jobs.txt`: The Job Specification File

The `jobs.txt` file serves as the input to the scheduler, providing the necessary information about the simulated jobs. Its format is crucial for the correct operation of the simulation.

- **File Structure:**
  - The first line of the file specifies the time quantum, an integer value that determines the length of each time slice in the round-robin scheduling algorithm.
  - Subsequent lines each represent a job, with each line containing four space-separated values:
    - `job_name` (string): The name of the job, used for identification and logging.
    - `arrival_time` (integer): The time at which the job enters the ready queue.
    - `priority` (integer): The priority of the job, with lower values indicating higher priority.
    - `execution_time` (integer): The total execution time required to complete the job.

```
TimeSlice 3
jobA 0 1 6
jobB 2 2 9
jobC 4 1 4
```

- **Parsing and Interpretation:**
  - The `schedule_processes()` function is responsible for reading and parsing the `jobs.txt` file.
  - It opens the file in read mode and uses `fscanf()` to extract the values from each line.
  - The extracted values are then used to populate the fields of the `Job` structures, which are stored in an array.
  - Error checking is implemented to handle cases where the file cannot be opened or the data is in the incorrect format.

5.2. `scheduler.log`: The Event Log File

The `scheduler.log` file serves as the output of the scheduler, providing a detailed record of scheduling events. Its format is designed for easy analysis and debugging.

- **File Structure:**
  - Each line of the file represents a scheduling event, with each line containing a timestamp and a message describing the event.
  - The timestamp is formatted using `strftime()`, providing a human-readable representation of the time at which the event occurred.
  - The message provides a textual description of the event, such as job creation, execution, context switching, or completion.

```
[2024-10-27 10:00:00] [INFO] Forking new process for jobA
[2024-10-27 10:00:00] [INFO] Executing jobA (PID: 1234)
using exec
[2024-10-27 10:00:03] [INFO] jobA ran for 3 seconds. Time
slice expired - Sending SIGSTOP
[2024-10-27 10:00:03] [INFO] Forking new process for jobB
```

- **Logging Mechanism:**
    - The `log_event()` function is responsible for writing events to the `scheduler.log` file.
    - It opens the file in append mode, ensuring that each new event is added to the end of the existing log.
    - It retrieves the current timestamp using `time()` and formats it using `strftime()`.
    - It writes the timestamped message to the log file using `fprintf()`.
    - Error handling is included to ensure that the program continues to operate even if there are issues with the log file.
- **Purpose:**
    - Debugging: The log file provides a detailed record of the scheduler's actions, making it easier to identify and resolve errors.
    - Analysis: The log file can be used to analyze the scheduler's performance and behavior, providing insights into its efficiency and fairness.
    - Verification: The log file can be used to verify the correctness of the scheduler's implementation, ensuring that it adheres to the specified scheduling algorithm.

These files serve as the primary means of communication between the scheduler simulation and the external world, enabling the simulation to receive job specifications and communicate its scheduling decisions.

6. Makefile: Automating the Build Process

The `Makefile` is an essential tool for automating the compilation and execution of the process scheduler simulation. It simplifies the build process, ensuring consistency and efficiency.

- **Structure and Targets:**
    - The `Makefile` consists of a series of rules, each defining a target and its corresponding dependencies and commands.
    - The primary targets in the `Makefile` are:
        - `scheduler`: This target compiles the `scheduler.c` source code into an executable named `scheduler`.
        - `run`: This target executes the compiled `scheduler` executable.
        - `clean`: This target removes the generated `scheduler` executable and the `scheduler.log` file.
- **Compilation (`scheduler` Target):**
    - The `scheduler` target specifies that the `scheduler` executable depends on the `scheduler.c` source file.
    - The command associated with this target is `gcc scheduler.c -o scheduler`.
    - This command invokes the `gcc` compiler to compile the `scheduler.c` file and create the `scheduler` executable.

    - The `-o scheduler` option specifies the name of the output executable.

- **Execution (`run` Target):**
    - The `run` target specifies that it depends on the `scheduler` executable.
    - The command associated with this target is `./scheduler`.
    - This command executes the compiled `scheduler` executable.
    - This target simplifies the process of running the simulation.
- **Cleanup (`clean` Target):**
    - The `clean` target specifies that it removes the scheduler executable and the scheduler log file.
    - The command associated with this target is `rm -f scheduler scheduler.log`.
    - This command uses the `rm` command with the `-f` (force) option to remove the specified files, even if they do not exist.
    - This target provides a convenient way to clean up the project directory, removing generated files.
- **Benefits:**
    - Automation: The `Makefile` automates the core build process, reducing the risk of errors and saving time.
    - Consistency: It ensures that the project is built in a consistent manner.
    - Convenience: It simplifies the process of compiling, running, and cleaning the project.

The `Makefile` effectively streamlines the essential tasks, providing a concise and efficient way to manage the project's build process.

7. README File: The Indispensable Guide to Project Understanding and Utilization

The `README` file stands as the essential gateway for anyone seeking to comprehend and effectively utilize the process scheduler simulation. It is far more than a simple text document; it is a meticulously crafted guide designed to bridge the gap between the developer's intent and the user's understanding. Functioning as the primary source of project documentation, it encapsulates crucial information that enables users to compile, execute, and interpret the simulation's behavior. In essence, it transforms the project from a collection of code files into an accessible and usable tool.

At its core, the `README` file aims to provide a clear, concise, and comprehensive overview of the project's purpose and functionality. It should begin with a succinct description of what the scheduler simulation does, outlining the scheduling algorithm it implements and the overall goal of the project. Following this, a detailed description of the project's file structure is vital. Users need to understand the role of each file, how they relate to one another, and where to find specific components.

Crucially, the `README` must provide explicit, step-by-step instructions on how to compile the project using the provided `Makefile`. This section should cater to users with varying levels of technical expertise, ensuring that even those unfamiliar with `make` can successfully build the executable. Similarly, detailed instructions on how to execute the compiled program are indispensable. This includes specifying any required command-line arguments, input files, and expected output.

A thorough explanation of the `jobs.txt` input file format is paramount. Users must understand how to structure their job specifications to ensure that the simulation processes them correctly. This section should include clear examples and explanations of each field within the input file. Likewise, a description of the `scheduler.log` output file format is essential for users to interpret the simulation's results. Examples of log entries and explanations of their meaning are highly beneficial.

Furthermore, the `README` should explicitly list any external dependencies required to build and run the project. This includes compiler versions, libraries, and any other tools that users might need to install. Providing usage examples can significantly enhance the user's understanding of how to apply the simulation to different scenarios. These examples should demonstrate various job configurations and their expected outcomes.

Finally, the `README` must include any relevant notes or limitations of the project. This could include known bugs, unsupported features, or any other information that users should be aware of.

The importance of the `README` file cannot be overstated. It transforms a potentially opaque code base into an accessible and usable tool, fostering collaboration and facilitating maintenance. By adhering to principles of clarity, completeness, organization, and currency, developers can ensure that their projects are not only functional but also readily understood and utilized by others.

8. Discussion: Reflecting on Design, Findings, and Implementation Choices

This section delves into a reflective analysis of the design decisions, findings, and implementation choices made throughout the development of the process scheduler simulation. It aims to provide insight into the rationale behind our approach and to discuss the advantages, disadvantages, and overall perspective gained from this project.

8.1. Design Decisions and Rationale:

- **Modular Architecture:**
  - The decision to structure the project using `scheduler.h` and `scheduler.c` was driven by a desire for clarity and maintainability. This modular approach allows for a clear separation of interface and implementation, enhancing code readability and facilitating future modifications.
  - We believe this approach is beneficial for larger projects, where many developers work on the same project.

- **`Job` Structure Design:**
    - The `Job` structure was meticulously designed to encapsulate all essential job attributes. The inclusion of `arrival_time`, `priority`, `execution_time`, `remaining_time`, `pid`, and `has_started` was deemed necessary to accurately simulate the behavior of a real-time scheduler.
    - The size limit of the name variable was a design decision that helped to simplify memory usage, and string manipulation.
- **Function Decomposition:**
    - Functions were created to perform specific, well-defined tasks. This decomposition enhances code reusability and testability. For example, `log_event()` handles all logging operations, while `find_next_job()` encapsulates the scheduling algorithm's logic.
    - This approach helped to keep the code clean, and easy to debug.
- **Logging Mechanism:**
    - The implementation of a robust logging mechanism using `scheduler.log` was considered crucial for debugging and analysis. It provides a chronological record of scheduling events, facilitating the verification of the algorithm's behavior.
    - This helped us to see the flow of the program, and to analyze the results.

8.2. Findings and Observations:

- **Complexity of Scheduling Algorithms:**
    - Implementing the preemptive priority-based round-robin algorithm revealed the inherent complexity of process scheduling. Balancing fairness, responsiveness, and efficiency requires careful consideration of various factors.
- **Importance of Input Data:**
    - The accuracy and format of the `jobs.txt` input file significantly impact the simulation's results. Thorough input validation is essential to prevent errors and ensure reliable output.
- **Impact of Time Quantum:**
    - Experimenting with different time quantum values demonstrated its significant influence on the scheduler's behavior. A smaller time quantum increases context switching overhead, while a larger time quantum can lead to longer response times for high-priority jobs.
- **Debugging with Logs:**
    - The scheduler log file was very important for the debuging proccess.

8.3. Advantages and Disadvantages:

- **Advantages:**
  - The modular design enhances code maintainability and reusability.
  - The detailed logging mechanism facilitates debugging and analysis.
  - The simulation provides a valuable tool for understanding process scheduling concepts.
- **Disadvantages:**
  - The simulation's complexity can be challenging for beginners.
  - The use of `sleep()` to simulate job execution introduces inaccuracies.
  - The project does not simulate all aspects of a real operating system.

8.4. Our Perspective:

- This project provided a valuable learning experience, deepening our understanding of operating system concepts and C programming.
- We gained practical experience in designing and implementing a complex algorithm.
- We learned the importance of clear documentation, and clean code.
- We believe that this homework was very useful for us.

8.5. Scheduling Fairness Analysis:

- Our scheduler attempts to ensure fairness by using a round-robin approach within each priority level. However, if there is a lot of high priority jobs, the low priority jobs can be starved.
- To improve fairness, we could implement dynamic priority adjustments, or use weighted round robin.

8.6. Edge Cases and Failure Scenarios:

- **Process Not Responding:**
  - If a simulated process (child process) fails to respond, the parent process (scheduler) could implement timeouts and terminate the unresponsive process.
- **Starvation:**
  - As mentioned earlier, low-priority jobs could be starved if there is a continuous influx of high-priority jobs.
  - We could implement aging, or dynamic priority changes.
- **Deadlock:**
  - Because this program only simulates CPU scheduling, and does not simulate other resources, deadlocks are not possible.
- To handle these cases, robust error handling, timeouts, and resource management strategies would be required.

This discussion provides a comprehensive overview of our design decisions, findings, and perspectives, offering valuable insights into the development of the process scheduler simulation.