📚 COP4610 Assignment 3: CPU Scheduling Algorithms Implementation

🔑 Table of Contents

📢 Important Note!

Please ensure that your implementation works on the course-designated ocelot server: ocelot-bbhatkal.aul.fiu.edu . Claims such as **"it works on my machine"** will not be considered valid, as all testing and grading will be conducted in that environment.

⚠️ **Important:** You are provided with an autograder to support consistent testing and maintain the highest level of grading transparency. The same autograder will be used by the instructor for final evaluation.

If your implementation passes the test case using the autograder, you are likely to receive full credit for your submission.

You must use these strict gcc compilation flags while building your application:
-std=c17
-D_POSIX_C_SOURCE=200809L
-Wall
-Wextra
-Werror
-g
-O0

⚠️ **There is only one submission - RESUBMISSION IS NOT ALLOWED. You must test your implementation with the provided testing framework on the Ocelot server before the final submission.**

---

1. 📧 TA Contact Information

For any questions or concerns related to assignment grading, please reach out to the TA listed below.

Communication Guidelines:

- Ensure that all communication is polite and respectful
- **You must contact the Graders first for any grading issues**
- If the matter remains unresolved, feel free to reach out to the instructor

Name: Instructor – bbhatkal@fiu.edu (⚠️ via the Canvas inbox only)
Section: COP 4610-U01 (84957)

Name: Angie Martinez – amart1070@fiu.edu (⚠️ via the Canvas inbox only)
Section: COP 4610-U02 (85031)

Name: Srushti Visweswaraiah – svisw003@fiu.edu(⚠️ via the Canvas inbox only)
Sections: COP 4610-UHA (85236)

---

2. 📖 Assignment Overview

This assignment provides **hands-on experience** with fundamental **CPU scheduling algorithms** used in **modern operating systems**. You will implement **six essential scheduling algorithms** that form the backbone of **process management** in operating systems—the same types of algorithms running right now in your laptop, smartphone, and the servers powering the internet.

Through this implementation, you will explore how operating systems **schedule processes**, **allocate CPU time**, **manage process priorities**, and **optimize system performance**. Your implementation must exhibit **deterministic behavior** – for the same input, it must consistently produce the same output, **correct calculation of scheduling metrics**, and **proper handling of tie-breaking rules**—just as real operating system schedulers must.

These CPU scheduling algorithms are not abstract academic concepts—they are the **critical mechanisms** enabling fair CPU allocation, responsive multitasking, and efficient resource utilization in every computing system. Mastering their implementation will deepen your understanding of how operating systems achieve fairness, efficiency, and responsiveness at the process management level.

---

3. 🛠️ Skills You'll Develop

In this assignment, you'll **gain hands-on experience** with both the **technical skills** and **operating system concepts** that are essential for professional systems programming and OS-level development such as:

- **Implement** core CPU scheduling algorithms (FCFS, SJF, SRT, RR, Priority-based).
- **Calculate** scheduling metrics including turnaround time, waiting time, and averages.
- **Apply** tie-breaking rules consistently across different scheduling algorithms.

- **Handle** preemptive and non-preemptive scheduling mechanisms.
- **Optimize** algorithms for correctness and performance in process scheduling.
- **Work** with process control blocks and scheduling queues.
- **Write** deterministic, predictable code for system-critical scheduling functionality.
- **Use** modular design principles for maintainable and reusable code.
- **Test** your code thoroughly against process scheduling scenarios.
- **Produce** professional-quality C code following industry standards.

4. 🎯 Learning Objectives

- **Understand** and implement CPU scheduling algorithms used in real operating systems.
- **Apply** First-Come First-Served (FCFS) for simple, non-preemptive scheduling.
- **Implement** Shortest Job First (SJF) for optimal average waiting time.
- **Design** Shortest Remaining Time (SRT) for preemptive shortest-job scheduling.
- **Build** Round Robin (RR) scheduler with time quantum management.
- **Create** Priority-based schedulers (both preemptive and non-preemptive).
- **Calculate** turnaround time and waiting time for process evaluation.
- **Handle** edge cases including CPU idle time and process starvation scenarios.

---

5. ⚙️ Assignment Tasks

CPU Scheduling Algorithm Implementation Requirements

You will implement **six fundamental CPU scheduling algorithms** using a provided modular framework:

1). First-Come, First-Served (FCFS):

A non-preemptive scheduling algorithm where processes are executed in the order they arrive

- **Key Characteristics**: Non-preemptive, simple FIFO order, may cause convoy effect
- **Tie-Breaking Rule**: Arrival time → Process ID
- **Usage in OS**: Simple batch processing systems, basic task scheduling
- **Performance**: No overhead, but may have high average waiting time

2). Round Robin (RR):

A preemptive scheduling algorithm using time-sharing with fixed time quantum

- **Key Characteristics**: Preemptive, fair CPU allocation, uses circular queue
- **Tie-Breaking Rule**: Arrival time → Process ID (for initial queue ordering)
- **Usage in OS**: Time-sharing systems, interactive applications, modern OS schedulers
- **Performance**: Fair allocation, responsive, overhead depends on time quantum

3). Shortest Job First (SJF):

A non-preemptive scheduling algorithm that selects the process with shortest burst time

- **Key Characteristics**: Non-preemptive, optimal for minimizing average waiting time
- **Tie-Breaking Rule**: Burst time → Arrival time → Process ID
- **Usage in OS**: Batch processing, background job scheduling
- **Performance**: Optimal average waiting time, but can cause starvation

4). Shortest Remaining Time First (SRT):

A preemptive version of SJF that selects the process with shortest remaining time

- **Key Characteristics**: Preemptive, dynamic priority based on remaining time
- **Tie-Breaking Rule**: Remaining time → Arrival time → Process ID
- **Usage in OS**: Real-time systems, priority-based scheduling
- **Performance**: Better average waiting time than SJF, more responsive

5). Priority Scheduling - Non-Preemptive:

A scheduling algorithm where each process has a priority value (lower number = higher priority)

- **Key Characteristics**: Non-preemptive, priority-based selection
- **Tie-Breaking Rule**: Priority value → Arrival time → Process ID
- **Usage in OS**: System processes, critical task management
- **Performance**: Flexible prioritization, but can cause starvation

6). Priority Scheduling - Preemptive with Round Robin:

A complex algorithm combining preemptive priority scheduling with Round Robin for same-priority processes

- **Key Characteristics**: Preemptive, priority-based with RR tie-breaking
- **Tie-Breaking Rule**: Priority value → Round Robin (time quantum) → Process ID
- **Usage in OS**: Modern operating systems, multi-level feedback queues
- **Performance**: Balanced fairness and priority, responsive to high-priority tasks

Implementation Standards

- Each scheduling algorithm must handle **identical test input** and produce **deterministic output**
- Implement **correct tie-breaking rules** for each algorithm as specified
- Calculate **turnaround time** and **waiting time** accurately using standard formulas:
    - **Turnaround Time (TAT) = Completion Time - Arrival Time**
    - **Waiting Time (WT) = Turnaround Time - Burst Time**
- Handle **CPU idle time** when no process is ready
- Ensure **output format** matches exactly with the reference implementation
- Follow **exact function signatures** provided in skeleton files for autograder compatibility

---

6. 🗂️ Project Framework

This assignment includes a comprehensive development framework comprising **skeleton code files** (*templates for your implementation*), a **driver program**, a **Makefile**, and an **autograder** to support and guide your implementation process.

1) Provided Framework Infrastructure (*Do Not Modify*)

- `CPU_scheduler.h`: Complete header file with data structures, constants, and function prototypes. **(study it thoroughly!)**
- `driver.c`: The main driver program responsible for reading input, coordinating all scheduling algorithms, and managing the overall execution flow.
- `Makefile`: Used to build your application and generate the final executable `scheduler`. You can run it using the following command:

```
./scheduler < Testing/Testcases/input1.txt
```

- `autograder_scheduler.sh`: Automated testing script **(identical to the final grading mechanism)**.
- `batchgrader_scheduler.sh`: Batch processing script for grading multiple student submissions. **(identical to the final grading mechanism)**.

2) Student Implementation Files

- `skeleton_first_come_first_served.c`: FCFS implementation template with guided TODO sections. Rename it to → `first_come_first_served.c`.
- `skeleton_shortest_job_first.c`: SJF template with guided TODO sections. Rename it to → `shortest_job_first.c`
- `skeleton_shortest_remaining_time_first.c`: SRT template with guided TODO sections. Rename it to → `shortest_remaining_time_first.c`
- `skeleton_round_robin.c`: Round Robin template with guided TODO sections. Rename it to → `round_robin.c`
- `skeleton_priority_non_preemptive.c`: Priority NP template with guided TODO sections. Rename it to → `priority_non_preemptive.c`
- `skeleton_priority_preemptive_rr.c`: Priority Preemptive with RR template with guided TODO sections. Rename it to → `priority_preemptive_rr.c`

3) Testing Infrastructure

- `Testing/Testcases/input1.txt`: Input test case with process data (PID, Burst Time, Priority, Arrival Time).
- `Testing/Expected_output/output1.txt`: Expected results (generated by executing **./A3_sample < Testing/Testcases/input1.txt > Testing/Expected_output/output1.txt**).

4) Instructor's Reference Implementation

`A3_sample`: The instructor-provided reference executable that demonstrates the **complete, correct, and required implementation** of this assignment. 💡 **This executable serves as your primary reference for understanding the expected behavior.**

⚠️ **Important**: Your program's output must **exactly match** that of the instructor-provided executable for equivalent input. Any deviation in behavior, logic, or output formatting may prevent the autograder from evaluating your submission.

5) Scheduling Algorithm Requirements

All scheduling algorithms must use the definitions provided in `CPU_scheduler.h`. Refer to this header file **thoroughly** to understand the process structure, constants, function prototypes, and other framework components required for your implementation.

6) Provided Files Layout

```
A3_PROVIDED_FILES/
├── Sample_Executable/
│   └── A3_sample                 # Reference implementation (study this thoroughly!)
├── Framework/
│   ├── CPU_scheduler.h           # Header file (provided - DO NOT MODIFY)
│   ├── driver.c                  # Application driver (provided - DO NOT MODIFY)
│   ├── Makefile                  # Builds your application (provided - DO NOT MODIFY)
│   ├── autograder_scheduler.sh   # Autograder (provided - DO NOT MODIFY)
│   ├── batchgrader_scheduler.sh  # Batch grader (provided - DO NOT MODIFY)
│   └── skeleton_*.c              # Your implementation files (rename by removing 'skeleton_')
├── Testing/
│   ├── Testcases/
│   │   └── input1.txt            # Test case input file
│   └── Expected_output/
│       └── output1.txt           # Expected output (generate with A3_sample)
├── A3_Specification.pdf          # This detailed specification & requirement document
└── README.txt                    # You must submit team details in the same format
```

7) ✅ Execution Permissions Notice

If you **do not** have execute permissions for the instructor-provided `A3_sample`, `autograder_scheduler.sh`, `batchgrader_scheduler.sh` and any other files, you can manually grant the required permissions using the following command on a **Linux system**.

> **Note:** When files are uploaded or transferred to Ocelot server (`ocelot-bbhatkal.aul.fiu.edu`), execute permissions **may be stripped** due to security restrictions. In such cases, you must explicitly grant **read (r)**, **write (w)**, and **execute (x)** permissions.

Granting Permissions to All Items in the Current Directory

```
# Check current permissions
ls -l

# Grant read, write, and execute permissions (safe and preferred – applies only what's missing)
chmod +rwx *

# Assign full permissions (read, write, execute) to all users for all files in the current directory
# ⚠️ Not preferred – use only as a last resort!
chmod 777 *

# Verify that permissions were applied
ls -l
```

⚠️ **Caution: Use `chmod 777` with care!**
Granting full permissions to everyone can:

- Expose sensitive data
- Allow unintended modifications or deletions
- Introduce security risks, especially in shared or multi-user environments

8) ☑️ YOU MUST!

- **Test your implementation thoroughly** on the **Ocelot** server (`ocelot-bbhatkal.aul.fiu.edu`)
- **Ensure exact output matching** with the instructor sample executable `A3_sample`
- **Validate all calculations** and logic against the sample
- **Test edge cases** including CPU idle time and tie-breaking scenarios
- **Verify compilation** with the specified compiler flags

---

7. 🔧 Development & Testing

Required Files for Development

All the following files must be in the same directory:

1. **Framework files**: `driver.c`, `Makefile`, `CPU_scheduler.h`
2. **Your implementation files**: `first_come_first_served.c`, `shortest_job_first.c`, `shortest_remaining_time_first.c`, `round_robin.c`, `priority_non_preemptive.c`, `priority_preemptive_rr.c`
3. **Testing tools**: `autograder_scheduler.sh`, `batchgrader_scheduler.sh`
4. **Test data**: `Testing/Testcases/input1.txt`, `Testing/Expected_output/output1.txt`

⚠️ **Critical Testing Information: The instructor will use different and more rigorous test cases for final grading that are not provided to students. However, if you pass the provided test case with 100% accuracy, you are highly likely to pass the instructor's final grading test cases.**

Comprehensive Development Workflow

Step 1: Understand Expected Output Format

```
# Generate expected output from instructor sample
./A3_sample < Testing/Testcases/input1.txt > Testing/Expected_output/output1.txt

# View the expected output format to understand correct behavior
cat Testing/Expected_output/output1.txt
```

Step 2: Implement with Incremental Testing

- **Recommended Implementation Order:**
  - FCFS → SJF → SRT → Round Robin → Priority Non-Preemptive → Priority Preemptive with RR
- **Testing Strategy:**
  1. Implement one algorithm at a time
  2. Compile and test after each implementation: `make clean && make`
  3. Run your scheduler: `./scheduler < Testing/Testcases/input1.txt`
  4. Compare your output with expected output
  5. Debug and fix any discrepancies before moving to the next algorithm

Step 3: Compilation and Manual Testing Process

```
# Clean previous builds
make clean

# Compile your implementation
make

# If compilation succeeds, generate the output and store in student_output.txt
./scheduler < Testing/Testcases/input1.txt > student_output.txt

# Compare with expected output
diff student_output.txt Testing/Expected_output/output1.txt
```

Step 4: Use Autograders for Final Validation

```
# Make autograders executable (if needed)
chmod +x autograder_scheduler.sh
chmod +x batchgrader_scheduler.sh

# Run the autograders
./autograder_scheduler.sh
./batchgrader_scheduler.sh
```

⚠️ **Important**: **Please note that instructor will use the same autograder for the final grading.**

Autograder System

The autograder compares your program's output with pre-provided reference output to determine correctness.

Internal Process:

1. **Clean rebuild**: `make clean && make`
2. **Runs your implementation**: `./scheduler < Testing/Testcases/input1.txt > STUDENT_OUTPUT.txt`
3. **Compares outputs**: Extracts each scheduling algorithm section and compares with expected results
4. **Calculates scores**: Awards points based on percentage of matching output lines
5. **Provides detailed feedback**: Shows exactly which lines don't match for each algorithm

---

8. 📩 Assignment Submission Guidelines

- All assignments **must be submitted via Canvas** by the specified deadline as a **single compressed (.ZIP) file**. Submissions by any other means **WILL NOT** be accepted and graded!
- The filename must follow the format: `A3_FirstName_LastName.zip` (e.g., **A3_Harry_Potter.zip**)
- ⚠️ **Important:** Include the Firstname and Lastname of the **team's submitting member**, or your own name if **submitting solo**.
- ⚠️ **Important:** For **group submissions**, each group must designate one **corresponding member**. This member will be responsible for submitting the assignment and will serve as the primary point of contact for all communications related to that assignment.
- ⚠️ **Important Policy on Teamwork and Grading**

  All members of a team will receive the **same grade** for collaborative assignments. It is the responsibility of the team to ensure that work is **distributed equitably** among members. Any disagreements or conflicts should be addressed and resolved within the team in a **professional manner**. **If a resolution cannot be reached internally, the instructor may be requested to intervene** .
- ⚠️ **Important:** Your implementation **must be compiled and tested on the course designated Ocelot server** (`ocelot-bbhatkal.aul.fiu.edu`) **to ensure correctness and compatibility**. **Submissions that fail to compile or run as expected on Ocelot will receive a grade of ZERO.**
- Whether submitting as a **group or individually**, every submission must include a **README** file containing the following information:
  1. Full name(s) of all group member(s)
  2. **Prefix the name of the corresponding member with an asterisk** `*`
  3. Panther ID (PID) of each member
  4. FIU email address of each member
  5. Include any relevant notes about compilation, execution, or the programming environment needed to test your assignment. **Please keep it simple—avoid complex setup or testing requirements.**
- ⚠️ **Important - Late Submission Policy:** For each day the submission is delayed beyond the **Due date** and until the **Until date** (as listed on Canvas), a **10% penalty** will be applied to the total score. **Submissions after the Until date will not be accepted or considered for grading.**
  - **Due Date**: ANNOUNCED ON CANVAS
  - **Until Date**: ANNOUNCED ON CANVAS

- Please refer to the course syllabus for any other details.

---

9. 📦 Deliverable Folder (`ZIP file`) Structure (⚠️ exact structure required - no additional files / subfolders)

> ⚠️ **DO NOT submit any other files other than these required files. The batch autograder may treat any additional items in the ZIP file as invalid, which will result in a grade of zero: first_come_first_served.c, shortest_job_first.c, shortest_remaining_time_first.c, round_robin.c, priority_non_preemptive.c, priority_preemptive_rr.c, README.txt**
>
> 📁 All other required framework files and the test cases will be supplied by the instructor to the autograder during final grading.

- ⚠️ You are required to submit your work as **a single ZIP archive file** to the Canvas.
- ⚠️ **Filename Format**: your `A3_Firstname_Lastname.zip` (**exact format required**).
- **You will be held responsible for receiving a ZERO grade if the submission guidelines are not followed**.
- ⚠️ **For team submissions** – Include the first name and last name of the team's submitting member in the ZIP file name (e.g., `A3_Harry_Potter.zip`). **For solo submissions** – Use your own first and last name.

```
A3_FirstName_LastName.zip
├── first_come_first_served.c       # Your FCFS implementation
├── shortest_job_first.c            # Your SJF implementation
├── shortest_remaining_time_first.c # Your SRT implementation
├── round_robin.c                   # Your Round Robin implementation
├── priority_non_preemptive.c       # Your Priority NP implementation
├── priority_preemptive_rr.c        # Your Priority Preemptive RR implementation
└── README.txt                      # Team details (see submission guidelines)
```

---

10. 💯 Grading Rubric

Criteria	Points
⚠️ **Program fails to run or crashes during testing**	**0**
⚠️ **Partial Implementation** *(Evaluated based on completeness and at the instructor's discretion)*	**0 - 70**

Detailed Breakdown of 100 Points

Grading Component	Points
1. `First-Come First-Served (FCFS)` implementation	15
2. `Shortest Job First (SJF)` implementation	15
3. `Shortest Remaining Time First (SRT)` implementation	15

Grading Component	Points
4. `Round Robin (RR)` implementation	15
5. `Priority Scheduling - Non-Preemptive` implementation	15
6. `Priority Scheduling - Preemptive with Round Robin` implementation	15
7. Manual Grading - Clear code structure, Meaningful comments, well-written README, Overall readability	10
Total	**100**

🚀 ***Tips for Success:***

- Study the reference implementation thoroughly
- Implement algorithms incrementally following the recommended order
- Ensure **exact output matching** including formatting
- Test tie-breaking rules carefully
- Verify calculations for turnaround time and waiting time
- Handle CPU idle time correctly
- Maintain professional code quality throughout development

Good luck! 💪
