OS-344 Lab - 2

Instructions

- This assignment must be completed in a group.
- Group members should ensure that one member submits the completed assignment before the deadline.
- Each group should submit a report, with relevant screenshots/ necessary data and findings related to the assignments.
- We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through software, and plagiarised submissions will be penalised heavily, irrespective of the source and copy.
- There will be a viva associated with the assignment. Attendance of all group members is mandatory.
- Assignment code, report and viva all will be considered for grading.
- An early start is recommended; any extension to complete the assignment will not be given.
- If any required information is missing, reasonable assumptions may be made. All such assumptions must be mentioned in the report to ensure transparency and consistency.
- Before you start coding, read Chapters 4 and 7 of the xv6 book.

———————————————————

Task 2.1: Weighted Round Robin Scheduler (Moderate)
In this assignment, you will enhance the existing round-robin scheduler in xv6 to support weighted scheduling based on user-defined process priorities.

Objective:
—————
The default xv6 scheduler treats all processes equally. In this task, you will modify it to allow processes with higher priority to get more CPU time, while ensuring lower priority processes are not completely starved.

1. Add new system calls to allow setting and getting process priorities:
   - set_priority(n): sets the priority of the current process to n (where n is a positive integer).
   - get_priority(): returns the priority of the calling process.
   - You may assume a maximum priority value (e.g., 1000) for simplicity.
   - Implement basic error handling and input validation for these system calls.

2. Modify the xv6 scheduler to implement a Weighted Round Robin (WRR) algorithm using the priority values as weights:
   - A process with higher priority should get proportionally more CPU time than one with lower priority.
   - However, no process should be starved — all runnable processes must get some CPU time.
   - Set a default priority for all new processes (e.g., 10), unless explicitly changed.

3. Ensure correctness
   - Protect shared kernel structures with appropriate locking when accessing or updating priority values.

Deliverables:
———————
1. Ensure xv6 compiles, boots, and runs as expected with your changes.
2. Submit:
   - Modified system call implementations (sysproc.c, usys.S, etc.).
   - Changes to the scheduler (proc.c, sched.c, or wherever applicable).
   - Any helper functions you create.
   - At least one test program to demonstrate the weighted scheduling behavior.

Hints:
———
- Start by reviewing where in the kernel the scheduler is implemented.
- Use a simple weighted round robin logic: e.g., run a process for priority number of time slices before switching.
- Make sure you are using locks when updating shared state, especially in a multicore environment.
- Remember to add your test program to UPROGS in the Makefile so it is compiled automatically.

Notes:
———
- This is a moderate-difficulty assignment, requiring careful edits to core scheduling code.
- Focus on writing clean, commented, and readable code.
- Don't forget to recompile xv6 after making your changes:

  make clean
  make qemu

———————————————————————

Task 2.2: Implement a Custom Scheduling Algorithm (Advanced)
In this task, you will replace the existing scheduler in xv6 with a custom scheduling algorithm of your choice. This can be a standard algorithm (like Shortest Job First, Multilevel Feedback Queue, etc.) or a novel algorithm you design yourself.

Objective:
—————
Move beyond round-robin and weighted round-robin scheduling. Analyse and implement a scheduling algorithm that optimises one or more of the following goals:
- Minimise turnaround or response time
- Improve CPU utilisation
- Reduce waiting time
- Avoid starvation
- Ensure fairness

Requirements:
———————
1. Design or select a scheduling algorithm:
   - You may choose from classical algorithms (e.g., SJF, MLFQ, Lottery Scheduling, etc.) or propose your own.
   - Your scheduler must dynamically schedule processes at runtime — i.e., decisions should not be hardcoded or static.
   - Clearly document your algorithm and justify your choice based on its properties or the type of workload it is intended to optimize.

2. Modify the scheduler in xv6:
   - Replace the existing round-robin logic with your custom scheduling policy.

- Ensure that the scheduling decisions are based on the logic you've described and implemented.
- Consider corner cases: What happens with new processes? How are terminated processes handled? How are different process types (CPU-bound vs I/O-bound) treated?

Deliverables:
——————

1. A brief design document (design.txt) describing:
   - The scheduling algorithm you've implemented.
   - Why did you choose this algorithm.
   - What trade-offs does it make, and in which scenarios does it perform well or poorly in.

2. Modified scheduler code (typically proc.c, sched.c).
3. Any new system calls or kernel utilities added.
4. At least two user-level test programs demonstrating the effectiveness of your scheduler.
5. Updated Makefile to include your test programs in UPROGS.

Hints:
———

- Review how process states and scheduling are managed in proc.c and proc.h.
- You may need to track additional information per process (e.g., runtime, wait time, last scheduled time). Modify the struct proc to store this.
- To implement advanced algorithms, consider adding kernel timers or maintaining custom queues.
- Look at how context switching and scheduling are triggered in the scheduler().

Notes:
———

- This is an advanced assignment that tests your understanding of operating system design and trade-offs in process scheduling.
- There is no single correct solution — what matters is that your scheduler is functional, well-justified, and tested.