OPERATING SYSTEMS – ASSIGNMENT 2
SCHEDULING

Objectives :

1. Learn different process scheduling algorithms.
2. Implement different process scheduling algorithms in C programming language.

You will write a program to implement the following CPU scheduling algorithms.

1. First Come First Serve

2. Shortest Job First

3. Priority

4. Round Robin

5. Priority with Round Robin

The program reads a list of tasks from a file and then schedules them based on the chosen scheduling algorithm.

Each line in the task file describes a single task. Each task has a name, a priority(for algorithms that need priority), arrival time, and a CPU burst separated by commas.

Priority, burst time, and arrival time are represented by an integer number. The higher the number, the higher the priority.

Example task file:

<task id>,<priority>,<task arrival time>,<burst time>

```
1, 4, 0,10

2, 3, 3,8

3, 3, 2,9
```

Function to implement :

1- Build(.....)

The functions build convert each line in the file to a struct called **task** that has the same fields (task id, priority …).

2- Table(....)

The functions return an array of **tasks,** you could assume that the MAX length of tasks is equal to 10. (MAX is macro)

3- Display(....)

The function prints a **table** that shows (like in a practical session) the tasks in the file.

4- schedule(....)

The function print the scheduling of an array of tasks, you need to use **enum (DONT USE NUMBERS FOR ALGORTHIMS) for each algorithm name.**

The output should be:

<P<taskid>,CPU_TIME><P<taskid>,CPU_TIME>...........

Example : (FCFS, array_taks)

Array_taks contains 2 tasks :

0, 4, 0,10

1,3,1,5

The output :

<P0,10><P1,5>

PART2: XV6 Scheduling

Overview

In this part, you will learn how to implement a priority-based scheduler for xv6. To get started, download a new copy of the xv6 source code from here. Do NOT use the source code of previous homework. You'll do two things in this part:

- You'll replace xv6's current round-robin scheduler with a priority-based scheduler.
- You'll add a new syscall for a process to set its own priority.

2. 1: Priority-based Scheduler for Xv6 (50%)

In the first part, you will replace the round-robin scheduler for xv6 with a priority-based scheduler. The valid priority for a process is in the range of 0 to 200, inclusive. **The smaller value represents a higher priority.** For example, a process with a priority of 0 has the highest priority, while a process with a priority of 200 has the lowest priority. The default priority for a process is 50. A priority-based scheduler always selects the process with the highest priority for execution. If there are multiple processes with the same highest priority, the scheduler uses round-robin to execute them in turn to avoid starvation. For example, if process A, B, C, D, E have the priority of 30, 30, 30, 40, 50, respectively, the scheduler should execute A, B, and C first in a round-robin fashion, then execute D, and execute E at last.

For this part, you will need to modify proc.h and proc.c. The change to proc.h is simple: just add an integer field called priority to struct proc. The changes to proc.c are more complicated. You first need to add a line of code in the allocproc function to set the default priority for a process to 50.

Xv6's scheduler is implemented in the scheduler function in proc.c. The scheduler function is called by the mpmain function in main.c as the last step of initialization. This function will never return. It loops forever to schedule the next available process for execution.

In this part, you need to replace the scheduler function with your implementation of a priority-based scheduler. The major difference between your scheduler and the original one lies in how the next process is selected. Your scheduler loops through all the processes to find a process with the highest priority (instead of locating the next runnable process). If there are multiple processes with the same priority, it schedules them in turn (round-robin). One way to do that is to save the last scheduled process and start from it to loop through all the processes.

2.2: Add a Syscall to Set Priority (50%)

The first part adds support to priority-based scheduling. However, all the processes still have the same priority (50, the default priority). In the second part, you will add a new syscall (setpriority) for the process to change its priority. The syscall changes the current process's priority and returns the old priority.

If the new priority is lower than the old priority (i.e., the value of new priority is larger), the syscall will call yield to reschedule.

In this part, you will need to change user.h, usys.S, syscall.h, syscall.c, and sysproc.c

Here is a summery of what to do in each file:

• syscall.h: add a new definition for SYS_setpriority.

• user.h: declare the function for user-space applications to access the syscall by adding:

```
int setpriority(int);
```

• usys.S: implement the setpriority function by making a syscall to the kernel.

• syscall.c: add the handler for SYS_setpriority to the syscalls table using this declaration:

```
extern int sys_setpriority(void);
```

• sysproc.c: implement the syscall handler sys_setpriority.

In this function, you need to check that the new priority is valid (in the range of [0, 200]), update the process's priority. If the new priority is larger than the old priority, call yield to reschedule. You can use the proc pointer to access the process control block of the current process.