Lab 1: Shell and Scheduling

The purpose of this lab is to gain experience working with system calls, process management, and scheduling policies. There are two independent exercises in the lab: the first exercises is to implement a basic shell in C/C++ and the second requires you to implement/simulate four different scheduling policies given a process workload.

Part 1: Simple Shell in C++

The goal of this part of the lab is to implement a simple shell in C++. This requires you to use system calls directly in your code. In class, we touched upon how a few system calls, (notably fork and exec) can be used to implement a command shell. In this exercise, you will implement **tsh** (trivial shell), a simple shell.

Like a real shell, **tsh** will take the name of a program as an argument. The program can be the name of any executable in the current directory where your shell program resides. The shell should run the specified program with the arguments before prompting for a new user command.

The command **"quit"** should terminate your shell.

Here is a simple example:

```
elnux25$ ./tsh
tsh> ls
foo.txt
tsh> quit
elnux25$
```

To get started, download the starter code associated with this documentation. Do not change any file names or function names that are provided to you. Simply implement the real process logic.

Shell Notes

- Useful functions and system calls include fork, exec (specifically the execvp variant, in conjunction with the `cmdTokens` variable), sleep, and waitpid.
- If you're trying to use waitpid and get a warning like "warning: implicit declaration of function 'waitpid'", you probably need to include an additional system header file. Add the line:

  ```
  #include <sys/wait.h>
  ```

  to the top of your file above or below the other `#include` statements.

- Be careful when adding calls to fork -- if you write an infinite loop with a fork in it, a fork bomb will result. If in doubt, you can add a `sleep(1);` before each fork during debugging, which will slow the rate of process creation.

- **tsh** can execute a program with arguments, but cannot execute multiple programs using Bash constructs (e.g., `'sleep 3 && echo hello'` to sleep for 3 seconds, then print hello). However, you can accomplish the same by making a new Bash file (e.g., `'sleephello'`) and calling that from within tsh (e.g., `'./sleephello'`). If you do this, make sure the script you are trying to call is executable (`'chmod +x sleephello'`).

- Since each OS has its own (different) system call interface, you should use linux machines since fork and exec are Unix/linux system calls. You can use your own machine to write the code, but you must ensure that it runs on linux.

- Do not use cygwin/windows for this lab as fork/exec are not supported on windows.

Part 2: Scheduling

The goal of this part of the assignment is to implement four different scheduling algorithms in a scheduling simulator. In particular, you are to implement:

- First-In, First-Out (FIFO)
- Shortest Job First (SJF)
- Shortest Time to Completion First (STCF)
- Round Robin (RR)

If you need to brush up on these algorithms before starting you should review the chapter in the book that discusses CPU scheduling and/or review the slides covering this material from class.

You are given some starter code that defines the interface to the scheduling simulator and some basic data structures to get you going. You cannot change the interface or the data structures. If you do the automated tests may fail and you will be heavily penalized. The files for this part of the lab include:

- `src/scheduling.h`
- `src/scheduling.cpp`
- `src/main_scheduling.cpp`

You need not touch `scheduling.h` or `main_scheduling.cpp`. All the work that you need to do can be found inside `scheduling.cpp`.

Scheduling Notes

- The files inside the workloads directory describe a workload or processes and their start time and execution duration. The definition is two integers per line:

  ```
  0 100
  ```

```
10 10
10 10
```

The first integer is the start time and the second integer is how long the process needs to run (the duration).

- The **`Process`** struct records all the information you need to know about a process before and after it runs through a scheduling algorithm. Only the `arrival` time and `duration` are known before a process executes. The `first_run` field is updated to indicate when a process first begins executing and `completion` is the time a process completes executing.

- The **`ArrivalComparator`** and **`DurationComparator`** are two classes that define an ordering on processes. The former is used to order processes by their arrival time. The later is used to order processes by their duration. These classes are used as part of the definition of priority queues that are used to manage processes.

- To make it easier to define priority queues we define two new types using typedef: **`pqueue_arrival`** and **`pqueue_duration`**. The former is used to define priority queues that order processes by arrival time and the later is used to define priority queues that order processes by duration (how long they have left to execute).

- You are to implement a function to read in a workload file and return a priority queue ordered by arrival time:

```
pqueue_arrival read_workload(string filename)
```

- You are to implement 4 CPU scheduling algorithms:

```
list<Process> fifo(pqueue_arrival workload);
list<Process> sjf(pqueue_arrival workload);
list<Process> stcf(pqueue_arrival workload);
list<Process> rr(pqueue_arrival workload);
```

Each CPU scheduling function takes a priority queue of processes ordered by their arrival time and returns a list of processes ordered by their completion time. In addition, the processes that are returned have their `first_run` and `completion` fields filled in. You can assume a time slice of 1 time unit for the round robin scheduler and all processes are added to the end of the process queue upon arrival.

TIP: you are likely to need both a queue containing the entire workload and another queue for processes that have already arrived for most of the above algorithms.

- You are to implement two metric producing functions:

```
float avg_turnaround(list<Process> processes)
float avg_response(list<Process> processes)
```

The first returns the average turnaround time given a list of processes. The second returns the average response time given a list of processes.

- You are free to choose how you implement each of the above functions. However, our recommendation is to not use references and pointers. Our implementation did not use any references or pointers which made it easier to reason about what we were doing. If you need to modify anything, it is simply easier to make a copy of it. See the `show_workload` and `show_processes` for examples on how we copied priority queues.