COMP3520 Operating Systems Internals
Programming Exercise 4
FCFS Dispatcher

General Instructions

In this exercise, you will study a First-Come-First-Served (FCFS) dispatcher for a single-processor system.

To help you get started, source codes are provided. Some comments are included to explain various parts of the source codes.

Please bear in mind that this exercise is intended to introduce you the concepts that you will need in order to complete COMP3520 Assignment 2.

This exercise will **not** be marked; assessment for COMP3520 Assignment 2 will be based only on your final source code and discussion document.

First-Come-First-Served Dispatcher

For this exercise, you will have a list of jobs that you will need to schedule for execution depending on the job's time of submission (or arrival) and its allowed service (or execution) time.

The job dispatch list format is defined as follows:

<arrival time>, <service time>

0, 3

2, 6

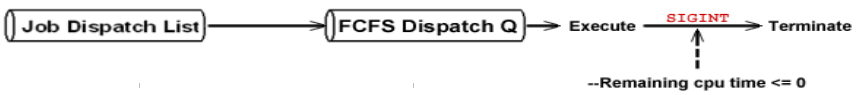
4, 4

6, 5

8, 2

The arrival times and corresponding job execution durations can be randomly generated. The jobs in the list are in the chronological order of arrival.

The job dispatch list will be stored in a file. When the computation starts, all jobs in the file are first loaded to a FCFS dispatch queue. When there is no a job currently running, the first job in the FCFS dispatch queue will be dequeued and dispatched for execution after it has "arrived".



FCFS Dispatcher Algorithm and Implementation

The FCFS dispatcher maintains a queue of jobs, using a single linked list. A single linked list consists of a set of nodes. Each node contains a pointer to the next node to form a chain of nodes with the last node pointing to NULL. A set of functions are used to manipulate the list, such as, to create a new node, insert the node into the list and remove/delete a node form the list.

For the FCFS dispatcher, each node in the linked list is a process control block (PCB) structure such as the following:

```
struct pcb {
    pid_t pid;              // system process ID
    char * args[3];         // program name and args
    int arrival_time;
    int service_time;
    int remaining_cpu_time;
    int status;
    struct pcb * next;      // links for Pcb handlers
};
typedef struct pcb Pcb;
typedef Pcb * PcbPtr;
```

A similar structure will be used in Assignment 2.

For list management, we need a set of functions:

```
PcbPtr createnullPcb(void)
- create inactive Pcb.
returns:
  PcbPtr of newly initialised Pcb
  NULL if malloc failed

PcbPtr enqPcb (PcbPtr headofQ, PcbPtr process)
- queue process (or join queues) at end of queue
- enqueues at "tail" of queue list.
returns:
  (new) head of queue

PcbPtr deqPcb (PcbPtr * headofQ);
- dequeue process - take Pcb from "head" of queue.
returns:
  PcbPtr if dequeued,
  NULL if queue was empty
  & sets new head for the queue
```

These functions can be found in **pcb.c** and **pcb.h**.

A Pseudo code for the FCFS dispatcher is give below.

1) Start dispatcher timer;
2) While there's anything in the queue or there is a currently running process:
   a) If a process is currently running;
      i) Decrement process **remainingcputime**;
      ii) If times up:
         (1) Send **SIGINT** to the process to terminate it;
         (2) Free PCB structure memory

b) If no process currently running && dispatcher queue is not empty && **arrivaltime** of process at head of queue is <= dispatcher timer:
   i) Dequeue process and start it (*fork()* & *execv()*)
   ii) Set it as currently running process;
c) Sleep for one second;
d) Increment dispatcher timer;
e) Go back to 2.

3) Exit.

To implement the above algorithm, we need two more PCB functions to start and terminate processes:

```
PcbPtr startPcb(PcbPtr process)
- start a process
returns:
  PcbPtr of process
  NULL if start failed

PcbPtr terminatePcb(PcbPtr process)
- terminate a process
returns:
  PcbPtr of process
  NULL if terminate failed
```

For this exercise and Assignment 2, you need *fork()*, and *execvp()* to run a real *process* when you start a process. The source program of this *process* is called **sigtrap.c**. It will tell you what signals you have sent to it as well as count the number of seconds that have elapsed since it started running. To get familiar with this program try to compile and run it several times:

gcc –o process sigtrap.c
./process 5

In the above example, the *process* will run for 5 seconds and stop. If the number is not specified, the process will last for 60 seconds.

You also need to use *kill()* function to terminate a process.

Please refer to the relevant Linux *man* page for more information on *fork()*, *execvp()* and *kill()* functions.

The main program for the FCFS dispatcher can be found in **fcfs.c** and **fcfs.h**.

Tasks for the Lab

Task 1: fork, exec and kill functions

In UNIX-based operating systems, processes can be created using the *fork* and *exec* functions; also, there are a set of process control signals that can be sent to processes by using the *kill* function.

sigtrap.c is a program that traps some of the major process control signals and reports them on the terminal. In Assignment 2, you will use this program to simulate jobs.

For process control in this exercise, you need to use SIGINT signal.

Write a short test program (1) to create a process using *fork* and then one of the exec functions, e.g., *execvp* for the created child process to execute `./process` program; and (2) for the parent process to use *kill* function to send a signal (i.e., SIGINT which is the equivalent of the Ctrl-C keyboard interrupt fielded by the shell) to terminate the child process after sleeping a few seconds.

Hints: To use *execvp* function, you may first declare a char string array of size two and fill in

```
args[0] = "./process";
args[1] = NULL;
```

and then, when doing the exec all you need to do is:

```
execvp(args[0], args);
```

To terminate the child process, the parent process uses the child process pid, and calls the function:

```
if (kill(pid, SIGINT))
   fprintf(stderr, "terminate of %d failed", (int) pid);
```

Task 2: Random Jobs Generator

A program called **random.c** is provided that outputs a list of randomly generated jobs to a file whose name is specified by the user.

Compile the source code by using the following command:

gcc –o random random.c -lm

The syntax is *./random* <*filename*>. When this program is started, the user will be asked to enter the following parameters:

- Num_of_jobs – The number of jobs to be dispatched;
- Lambda_arrival – The mean of the Poisson distribution of intervals between job arrivals; and
- Lambda_service – The inverse of the mean of the exponential distribution of corresponding job execution durations.

Recommended values for *lambda_arrival* lie between one and twenty inclusive and may contain decimal fractions. For *lambda_service*, sensible values lie between zero and one inclusive.

The resulting job list file is in a format suitable for use by the dispatcher. Observe that, in general, the values in the job list file will be different for each execution even if the parameters used for each execution are the same.

Compile **random.c** and use this program to generate job list files for various parameters.

Task 3: Calculation of Average Turnaround Time and Average Waiting Time

1. Carefully study the source codes for the FCFS Dispatcher and try to understand all the details. Then compile the source codes by using the following command:

gcc -o fcfs pcb.c fcfs.c

When running the dispatcher program, you need a job dispatch list file as input:

./fcfs you_input_file

2. Modify the *main* program to include the calculation and print out on the screen of the average turnaround time and average waiting time.
3. Compile and run your modified program multiple times with different job dispatch lists to ensure that your program can output correct average turnaround time and average waiting time. (How do you design the job dispatch lists to test the correctness of your calculation?)