AUBURN UNIVERSITY

# COMP 3500 Lab 1Assignment

**Instructions** For this assignment, you must implement three CPU scheduling policies in C. The instructor will provide you with a shell in C. Your C program must compile and execute on the Unix machines on the Engineering Network System. If not, no credit will be awarded to you. These machines can be accessed through the host ***gate.eng.auburn.edu***. We will demo in class how to access these machines.

## Objectives of this assignment:

- to work on a Unix based system
- to "*dust off*" your programming skills in C
- to become familiar with the notion of a process control block
- to experience the life cycle of a process
- to "feel" how an operating system manages processes
- to evaluate and compare three fundamental scheduling policies

**IMPORTANT:**

1) *Your code will be tested and graded **REMOTELY** on the Engineering Unix (Tux) machines. If the code does not work on those machines, you will not get any credit even if your code works on any other machine.*
2) *A late submission will get a 50% penalty if submitted right after the deadline. The next day, you cannot submit the lab.*
3) *One submission per group.*
4) *Writing and presentation of your report are considered to grade your lab (30%). Your conclusions **must be supported** by the data/measurements you collect.*
5) *The quality of your code will be evaluated (20%).*
6) ***Questions about this lab must be posted on Piazza if you need a timely answer**.*

**Lab Assignment (Turned in by one group mate)**

It is assumed that by 5:00pm May 25,

**first**, 1) you have an engineering Unix account, 2) you can edit text files, 3) you can compile C programs, and 4) you can execute C programs on the Unix (Tux) machines. You can use any personal computer or computing lab to remotely access the Engineering Unix (Tux) machines.

**second,** you have your group partners signed up on Canvas: 2 points penalty per day late.

**Look at the "How to get started?" section at the very end of the lab.**

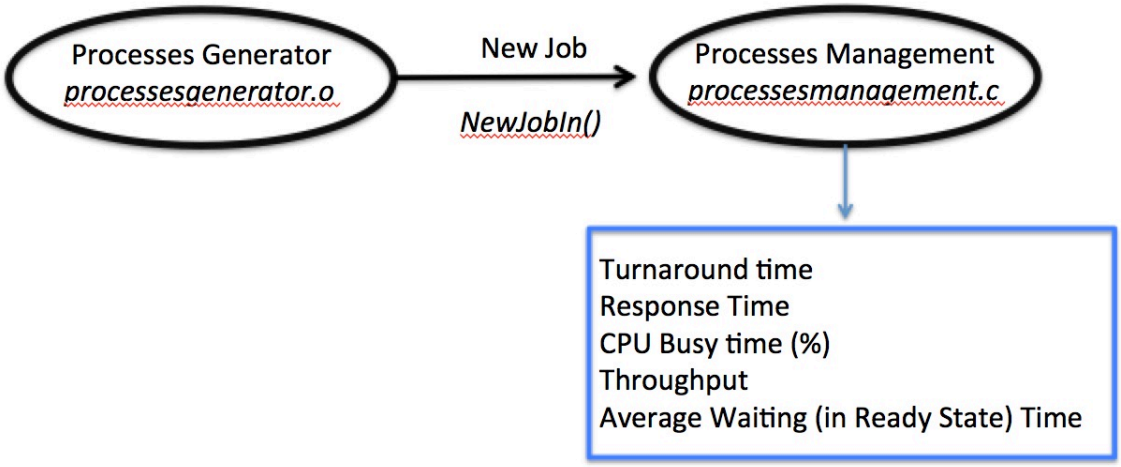
This lab has three parts: 1) Write an efficient code to simulate CPU scheduling policies, 2) evaluate these policies, and 3) analyze and report your results. Efficient code means a code that 1) is correct, , 2) is concise, 3) does not waste memory, and does not waste CPU cycles.

The instructor designed and implemented in C an emulation framework that allows the simulation of processes in order to implement and evaluate different CPU scheduling and memory management strategies.

A process is represented by a Process Control Block defined as follow

```
typedef struct ProcessControlBlockTag{
  Identifier ProcessID;
  State      state;
  Priority   priority;
  Timestamp  JobArrivalTime;    /* Time when job first entered job queue    */
  TimePeriod TotalJobDuration; /* Total CPU time job requires              */
  TimePeriod TimeInCpu;        /* Total time process spent so far on CPU   */
  TimePeriod CpuBurstTime;     /* Length of typical CPU burst of job       */
  TimePeriod RemainingCpuBurstTime; /* Remaing time of current CPU burst   */
  TimePeriod IOBurstTime;      /* Length of typical I/O burst of job       */
  TimePeriod TimeIOBurstDone;  /* Time when current I/O will be done       */
  Timestamp  JobStartTime;     /* Time when job first entered ready queue  */
  Timestamp  StartCpuTime;     /* Time when job was first placed on CPU    */
  Timestamp  TimeEnterWaiting; /* Last time Job Entered the Waiting Queue  */
  Timestamp  JobExitTime;      /* Time when job first entered exit queue   */
  TimePeriod TimeInReadyQueue; /* Total time process spent in ready queue  */
  TimePeriod TimeInWaitQueue;  /* Total time process spent in wait queue   */
  TimePeriod TimeInJobQueue;   /* Total time process spent in job queue    */
  Memory     TopOfMemory;      /* Address of top of allocated memory block */
  Memory     MemorySize;       /* Amount of allocated memory in bytes      */
  struct ProcessControlBlockTag *previous; /* previous element in linked list */
  struct ProcessControlBlockTag *next;     /* next element in linked list */
} ProcessControlBlock;
```

The emulation framework can be viewed as follows:




The system consists of two components: a "***Processes Generator***" and a "***Processes Management***" system. These two componentsconsist of two C programs: *processesgenerator.c* and *processmanagement.c*. In order to facilitate your task, you do not have access to the source of the program *processesgenerator.c*. You will use only the object file *processesgenerator.o* provided by the instructor.

# COMP 3500 Lab 1Assignment

Your task is to "complete" the program *processesmanagement.c*. In order to facilitate your task, the instructor built the template for this program using routines and variables to show you how to use them. You must *augment* this program (*processesmanagement.c*) to implement and evaluate three different CPU scheduling policies: first come first serve (**FCFS**), shortest remaining time first (**SRTF**), and Round Robin (**RR**). Your program must implement these strategies and instrument the code to compute and collect the average turnaround time, the average response time, the CPU Busy time (%), the throughput, and the average waiting Time (in **Ready State**). After you collect these averages for each CPU scheduling policy, analyze, compare, and draw conclusions about CPU scheduling. You must implement/complete these routines:

1) FCFS_Scheduler()
2) SRTF_Scheduler()
3) RR_Scheduler()
4) Dispatcher()
5) BookKeeping()dispatch

The *Processes Generator* generates processes with an inter-arrival time exponentially distributed. Whenever a process is generated, the routine *NewJobIn* (in ***processesmanagement.c***) is called. In order to "start" you, the instructor already included instruction to add every new job to the Job Queue (JOBQUEUE). From this point, you must manage these jobs just like an operating system would do.

You will be provided three files: ***common.h***, ***processesgenerator.o***, and ***processesmanagement.c***. You are not allowed to modify the file ***common.h*** or the ***main*** function in the ***processesmanagement.c*** file. In the file *processesmanagement.c*, you must develop your code ***INSIDE*** the function ***ManageProcesses().*** You may add new global variables or new routines (functions, methods) in the file *processesmanagement.c*. The instructor indicated on the program the routines/functions you need to implement.

To compile your program,
you must type: ***cc -o pm processesgenerator.o processesmanagement.c -lm***
where
***processesgenerator.o*** is the object file that emulates devices generating events
***pm*** is the executable.
***processesmanagement.c*** is the source file you must "complete".

**YOU CANNOT MODIFY *common.h*** (the original file common.h will be used to compile your submitted code)
**YOU CAN create new variables, new types, new routines/functions …. in *processesmanagement*.c .**

2) **Policy Evaluation:**

a) Compile your code with "*cc –o pm processesgenerators.o processesmanagement.c -lm*".
***b)*** Execute your code with "./pm PolicyNumber" where *PolicyNumber* is the CPU scheduling policy. *PolicyNumber* must take the value 1, 2, and 3 for FCFS, SRTF, and RR, respectively. ***The code generates 250 processes and stops.***
c) In order to evaluate your code **for each policy**, you must execute the program until it stops. You must "instrument" your code to collect **for each policy** the average turnaround time (**TAT**), the average response time (**RT**), the CPU Busy time (%) (**CBT**), the throughput (**T**), and the average waiting time (in **Ready State**) (**AWT**).

| Policy | PolicyNumber | TAT | RT | CBT | T | AWT |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| **FCFS** | | | | | | |
| **SRTF** | | | | | | |
| **RR (Q= 1 ms)** | | | | | | |
| **RR (Q= 5 ms)** | | | | | | |
| **RR (Q= 10 ms** | | | | | | |
| **RR (Q= 15 ms** | | | | | | |
| **RR (Q= 20 ms** | | | | | | |
| **RR (Q= 25 ms** | | | | | | |
| **RR (Q= 50 ms** | | | | | | |

For Round Robin, you must collect *TAT, RT, CBT, T,* and *AWT* for the following values for the quantum: 1 ms, 5 ms, 10 ms, 15 ms, 20 ms, 25 ms, and 50 ms.
For RR, plot *TAT, RT, CBT, T,* and *AWT* as a function of the quantum**.**

3) **CPU Scheduling Analysis:**
Based on the measurements for the different policies, discuss and compare the different policies (and the impact of the *quantum* for RR). Do these values match the expected performance of the different policies?

**Get Started**

1) compile the code I provided you by typing:
***cc-o pm processesgenerator.o processesmanagement.c -lm***
2) Execute the code: ./pm 1
3) Observe how the job queue grows
4) Stop the execution with CTRL-C.
5) Execute: ./pm 1 **l**
6) Now, with the parameter 1 (highlighted in red), you should see processes generated: the character 'G' appears just before displaying the process.
7) Stop with CTRL-C…..
8) "Play" with code ***ManageProcesses()*** in processmanagement.c for detecting all processes, then try to manage them.

**What to turn in?**

**1) Electronic copy** of your report and the C source code lab1.c. These two files must be put posted separately on Canvas (not in a zipped folder). **A penalty of 10 points will be applied if these instructions are not followed.**

2) Your report must:
   a. state whether your code works
   b. report/analyze the results (based on the filled table above and the plots). The quality of analysis and writing is critical to your grade.

c. address Part 3) "Performance Analysis" (quality of writing (content and form) is of utmost importance)

Good writing and presentation are expected.