AUBURN UNIVERSITY

COMP 3500 Lab 2 Assignment

Instructions This lab assignment explores the performance of three memory management techniques: Best-Fit, Worst-Fit, and Paging.

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 and memory
- to evaluate and compare three simple memory management strategies

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)
Note that the ***blue names in bold and Italic font*** will refer to variables you can access in the files *processesmanagement2.c* or *common2.h*.

Lab2 builds on Lab 1 described below. For Lab 1, memory was not a concern: it was assumed that memory was infinite. Lab 2 differs from Lab 1 by:

1) **Memory for Lab 2 is finite**. ***AvailableMemory*** is a variable contained in the file *common2.h*. It is initialized by the system to some value. You cannot increase ***AvailableMemory*** beyond that initial value. You must accordingly decrease ***AvailableMemory*** as you allocate memory to processes and must increase it as you free the memory when these processes complete. ***AvailableMemory*** is defined in the file ***common2.h***.

```
extern Memory AvailableMemory; // Total Available Memory
```

2) Each process will be created with **memory requirements**: each process control block contains the information :

```
Memory TopOfMemory;      /* Address of top of allocated memory block */

Memory MemoryAllocated;  /* Amount of Allocated memory in bytes      */

Memory MemoryRequested;  /* Amount of requested memory in bytes      */
```

When a process is generated, the variable MemoryRequested will be set to some value that you cannot change. The system must find a free hole larger than MemoryRequested for the process to run it.

3) A process cannot be *admitted* if there is not enough memory to accommodate it. You must keep track of the address where the process is loaded (TopOfMemory) and you must keep track of the list of free memory holes to accommodate future requests.
4) When a process completes, memory must accordingly be updated to eventually admit new processes.

Road Map to a Successful Lab 2

In order to build progressively Lab2 and determine the bounds on the performance you can hope for, you must follow these steps:

Step 1: In addition to the metrics collected for lab 1 (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**)), you must collect also the Average Waiting Time in the Job Queue (**AWTJQ**) and the number of completed processes. *AWTJQ* measures how fast you allocate memory to a process. The better is the memory policy, the lower *AWTJQ* should be (right?). You can keep track of the time in the job queue by updating this information in the process control block:

```
TimePeriod TimeInJobQueue; //Total time process spent in job queue
```

The number of completed processes is a good clue of the performance of the memory allocation policy.

Step 2: Collect all metrics for FCFS, SRTF, and RR (Quantum = 10ms, 20ms, 50ms, 250ms, and 500ms) using the program **processesmanagement2.c** ASIS (i.e., as provided without any modification). This program assumes an infinite memory. Therefore, the values you will get establish the optimal values you can hope for if memory was infinite. Can you predict what *AWTJQ* will be when memory is infinite?

Step 3: Implement the *Optimal Memory Allocation Policy* **(OMAP)**. Do not worry: it is simple. Manage your memory without worrying about **where** in the memory you place the processes or **where** the free memory blocks are. Just manage the memory using the variable ***AvailableMemory***. As long as (***AvailableMemory >= pcb->MemoryRequested***), you can admit the process and accordingly decrease ***AvailableMemory***. When the process completes, free the memory by increasing accordingly ***AvailableMemory***. Collect all metrics for FCFS, SRTF, and RR (Quantum = 10ms, 20ms, 50ms, 250ms, and 500ms) using the program **proccessesmanagement2.c** that you just improved by managing ***AvailableMemory***. The performance you will observe is the best performance you can hope for: it is good to know this *"upper"* bound.

Step 4: Implement the easiest memory allocation strategy. This strategy is ……. **Paging**!!! If you think about it: with paging, you do not have to worry about placing the processes **contiguously** or **where** you place those pages. In this case, all you need to do is to express the ***AvailableMemory*** in terms of ***NumberOfAvailablePages*** and the ***pcb->MemoryRequested*** in terms of

NumberOfRequestedPages. Your implementation of Paging will look very much close to **Optimal Memory Allocation Policy**. For page sizes 256 and 8 KB (8192), collect all metrics for FCFS, SRTF, and RR (Quantum = 10ms, 20ms, 50ms, 250ms, and 500ms) using the program ***proccessesmanagement2.c*** that you just improved by managing ***AvailableMemory*** in terms of pages, rather than bytes. For your analysis, think to compare OMAP with paging with the two diferent page sizes: *OMAP* is simply paging with a page size of one byte.

Step 5: Implement **Best-Fit** memory allocation strategy. I suggest to use a double linked list to manage the free memory holes. Collect all metrics for FCFS, SRTF, and RR (Quantum = 10ms, 20ms, 50ms, 250ms, and 500ms) using the program ***proccessesmanagement2.c*** that you just augmented with the implementation of **Best-Fit**.

Step 6: Redo step 5 with the **Worst-Fit** memory allocation strategy.

Step 7: Write a good report to report and analyze your results.

What to turn in?

1) **Electronic copy** of your report and the C source code lab2.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.