Group 14 Schedule Simulator

# Introduction

The way this paper is organized is according to problem number. Since there were 3 problems, each member of our group worked on a separate problem. Whenever we were stuck on an issue with our problem we called upon each other for help. This ensured we all participated equally and fairly.

# Problem 1

## Problem Analysis

For the first problem, we had to dynamically allocate and de-allocate memory to our generated processes. We modified our processor generator to bring down the total memory required by each process. We left the cycles alone and also decided on 2GHz for our processors. The main issues we needed to deal with was figuring out how to de-allocate from the array. What we decided on was getting the job to allocate memory to itself and have the slot allocated to its slot in the memory array in the simulator be equal to that address. After that job was done running, it frees itself and therefore the slot in the array as well.

## Problem Design

The program first ask how many processors we’re going to use in the simulation. After that it creates the array of jobs and sends the jobs, number of jobs, number of processors and memory counter to a boolean function that’s our simulator. The memory counter is passed by reference in order to display the total amount of memory to the user. The simulator then starts and run a while loop that runs depending on whether or not we’ve gone through the entire job array. It then runs a for loop that runs through each processor and each one a job if it isn’t busy, sets it to busy, and increases the job iterator. The processor figures out how many cycles it’ll take to run the job and sets its job time.

It then runs another for loop to go through the processors and runs an if statement that runs a boolean method inside the processor that subtracts 50 cycles from the jobTime. If the job time is less or equal to 0 it returns true, sets it’s busy flag to false to free it, and frees up the memory used by the process. Afterwards, the while loop runs again and gives jobs to any of the free processors.

The main method uses the C++11 *chrono* header file to in order to use the *high\_resolution\_clock* in order to time how long it takes to run the simulator. It starts the clock before sending the data to the simulator and then another one after we get a return from the simulator. Although it wasn’t necessary since we’ve assumed that the processes take up less than 10Mb of combined memory, there is an error checker in the simulator that returns false if the total memory is over 10Mb. The program displays how long it took to run the processes up until that point.

One limitations this implementation has is that the memory array continues adding memory to the next slot and doesn’t go back and check if a slot is empty or not. It just adds memory to to each slot. Since the memory slot was set to the address of the memory the job asked for, that memory slot should be changed as soon as we free the job memory. One potential fix would be to set the memory array to a fixed set of slots, instead of how ever many jobs there are, and go back through the array one we’ve reached the end and then check each slot to see if it’s empty. The size of each slot would have to be large enough to hold the max ammount of memory each job would require so there would be further limitations regarding how we create the jobs. Alternatively, if the job required more memory than the slot allowed, if would take up two slots or check how large the memory requirement is and give it as many slots that it would need to safely execute. One potential problem would be if we had a job that require the entire array, no job would be able to execute in the meantime or if there were not enough slots and other jobs kept taking up all the slots.

## Presentation and Discussion of Experimental Results

We ran a total of 25 different tests. 5 with 1 processor, 5 with 2 processors, and so forth. The following is the results for those 25 tests.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1 Processor** | | **2 Processors** | | **3 Processors** | | **4 Processors** | | **5 Processors** | |
| Trial Number | Total Runtime (ms) | Trial Number | Total Runtime (ms) | Trial Number | Total Runtime (ms) | Trial Number | Total Runtime (ms) | Trial Number | Total Runtime (ms) |
| 1 | 0.813 | 1 | 1.486 | 1 | 2.13 | 1 | 3.037 | 1 | 3.62 |
| 2 | 0.79 | 2 | 1.424 | 2 | 2.231 | 2 | 2.82 | 2 | 3.738 |
| 3 | 0.077 | 3 | 1.453 | 3 | 2.174 | 3 | 2.76 | 3 | 3.58 |
| 4 | 0.0764 | 4 | 1.453 | 4 | 2.264 | 4 | 2.772 | 4 | 4.106 |
| 5 | 0.761 | 5 | 1.484 | 5 | 2.958 | 5 | 3.032 | 5 | 3.643 |
| **Average Runtime** | **0.50348** | **Average Runtime** | **1.46** | **Average Runtime** | **2.3514** | **Average Runtime** | **2.8842** | **Average Runtime** | **3.7374** |

We did not expect for the Average Runtime to increase as we added more processors. We believe that the reason for the increase in runtime is that when each processor runs it’s jobProgress method, it is not done concurrently. Therefore the runtime is directly proportional to how many processors we have running. To check this we took the results of the 1 Processor experiment and multiplied it by 2,3,4, and 5 we ended up with the following

* 0.50348 \* 2 = 1.0069
* 0.50348 \* 3 = 1.51035
* 0.50348 \* 4 = 2.0138
* 0.50348 \* 5 = 2.51725

This is similar to the results we get for the other processors but not exact. To get a better runtime we would have to make jobProgress execute as a thread so it can run concurrently.

# Problem 2

## Problem Analysis

For problem two, we had to implement our own version of malloc and free. These versions were supposed to perform a similar task to their original functions in that they had to manage blocks of memory and couldn’t just keep variables in memory to fake it. However, we were allowed to manage this block of memory in whatever way we wanted. This gave us great flexibility in determining our approach to the problem, as well as our eventual solution. In addition, we no longer had to worry about the speed or memory restriction of the processor. With this simplification, we could focus solely on the memory management aspect of the project.

## Problem Design

After deliberation over several memory implementations we decided on a fixed partition scheme with a best fit memory placement approach. This approach proved to drastically reduce code complexity and increase its readability in a few ways.

First, because the memory was partitioned in fixed intervals, we could store those intervals in a vector. This allowed us to easily look up what memory was being used; in addition, we could easily find the partition for best fit as it became a simple loop. This approach greatly simplified code logic as we didn’t have to keep track of changing memory locations for other processes, nor did we have to relocate all the used memory blocks every time we called my\_malloc.

Second, with the memory in fixed intervals, we could arrange the intervals so the smaller partitions were first in the vector with the larger partitions at the end of the vector. This meant that implementing best fit was a simple case of looping over a for loop from the first memory partition to the last partition with the assurance that the first memory block found was the best fit. This again reduced code logic as we didn’t have to look up all the memory partitions and then decide on the right one.

Finally, fixed partition schemes mean we didn’t have to worry about external fragmentation. Because each block was already determined at initialization, in fact, before the program was even compiled, we never had to implement logic that kept track of moving memory blocks.

As a final decision, only one character was written to the start of each memory partition. Because the processes have no code, but rather are variables in our program, we simply needed to indicate whether the process “occupied” a memory slot. This meant that our vector containing the starting location of the memory partitions was actually a vector full of pointers to said memory locations. Upon using the memory, the pointer would be called and a single character would be written to indicate it was occupying that space. This reduced the amount of instructions and process time for our program as a whole.

## Presentation and Discussion of Experimental Results

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1 Processor** | | **2 Processors** | | **3 Processors** | | **4 Processors** | | **5 Processors** | |
| Trial Number | Total Runtime (ms) | Trial Number | Total Runtime (ms) | Trial Number | Total Runtime (ms) | Trial Number | Total Runtime (ms) | Trial Number | Total Runtime (ms) |
| 1 | 16.13 | 1 | 21.966 | 1 |  | 1 | 14.244 | 1 | 16.422 |
| 2 | 21.05 | 2 | 16.305 | 2 |  | 2 | 20.842 | 2 | 13.095 |
| 3 | 17.46 | 3 | 14.761 | 3 |  | 3 | 27.103 | 3 | 16.979 |
| 4 | 20.06 | 4 | 19.327 | 4 |  | 4 | 14.214 | 4 | 14.248 |
| 5 | 18.76 | 5 | 13.783 | 5 |  | 5 | 12.061 | 5 | 14.398 |
| **Average Runtime** | **18.675** | **Average Runtime** | **17.2284** | **Average Runtime** |  | **Average Runtime** | 17.6928 | **Average Runtime** | 15.0284 |

# Problem 3

## Problem Analysis

For the first two problems, We have enough memory to allocate the processes, so we do not need to manage the memory, we only put the processes in it and pull it off. However, for the Problem 3, we only have 50% of memory and 10% of memory, so we have to choose how to manage our memory to get good performance.In the case , we are using fixed-partation scheme.(Best-Fit) to deal with the memory management.

## Problem Design

We set up one vector called ***memorylist*** for storing the information about memory.(ie. how many blocks are available, how many blocks in using, which the starting address and ending address).we have two function control allocate and de-allocte. ***My\_malloc()*** and ***My\_free()***. Each time, we call ***my\_malloc***, we first check ***memorylist***, and find the free space we can use or not. if we have some holes can be used, then we find the smallest one, then update our ***memorylist***. The Interesting part in this implement is ***My\_free()*** ,we try to get the good performance , so we add the compaction in this phase. First we create a ***tmplist*** to storing the blocks which have been used, record their address and memory request. Then we combine the rest of space as a new big hole.

## Presentation and Discussion of Experimental Results

According to the result we have, 10% of the memory required makes the runtime very long. because the rest of processes have to wait until the space has been free. 50% of the memory required looks better than 10%. And the result show us, the runtime also depends on the total memory request of the set of processes, when we have the higher memory request, the runtime will be lower. The processes order also affect the runtime, when big memory request first, its runtime become higher, some one comes first, its runtime become lower.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| implement | 10% | | 50% | |
|  | Memory(byte) | runtime(second) | Memory(byte) | runtime(second) |
| 1 | 609790 | 43.65 | 3065324 | 10.23 |
| 2 | 587426 | 44.12 | 4203134 | 14.11 |
| 3 | 569872 | 42.13 | 3502341 | 10.25 |
| 4 | 619823 | 46.28 | 3892349 | 11.09 |
| 5 | 502431 | 35.10 | 4023768 | 13.82 |