Kristian Forestier
Zijia Zhai
Kaichuan Chan

Project #2 Report

Introduction:

For our project, we used the process generator, which we developed this semester earlier. The generator generated 50 processes with different memory sizes. In this project we allocated all of the 50 processes into memory. Our processes memory requirements ranged from 1KB to 500KB for each process. We randomly generated a memory requirement per process. We then used FIFO to simulate the processes coming in every 50 cycles and executing using their cycles to track if each process is finished or not. If the process finished, then we would free it.

In the first part, we used system calls malloc and free to dynamically allocate and de-allocate memory to each process. For the second part and third part, every time a process arrived it would allocate some memory from the 10MB block, 5MB block, or the 1MB block. Once the block was full the remaining process would have to wait until a completed process freed the required amount of memory. After then, we measured the system time for each part in our main.

Part #1:

The first part of the project was simple. We simply used malloc and free in order to do the dynamically allocating and de-allocating of memory to our processes. We allocated memory for each process one at a time using one for-loop and after that was done we freed the memory for each process using another for-loop.

Part #2:

For the second part of the project we simply created two functions which simulated the allocation and freeing of memory for the processes. We used a 10MB block and allocated the memory to each process. The allocating function was called My_malloc and the deallocating function was called My_free. Whenever a process needed to be allocated we would call the necessary function. Once all memory for the processes was allocated we simply assumed that the processes completed and freed the memory for all of them.

Part #3:

The third part of the project was the trickiest. For the first part of part 3 we allocated a set 10 MB block of memory for the processes to use. Since 10 MB was more than enough for all of our processes we did not have any processes waiting on other processes to free memory and all processes could complete without a problem. We needed to use an array in order to keep track of the cycles of each process. There was no scheduling but after a process arrived it would run to completion and when it finished (cycles = 0) we would free the memory it allocated regardless of whether or not another process was waiting on that memory.

For the second part of part 3 we allocated a set 5 MB block of memory for the processes to use. We needed to again use an array in order to keep track of the cycles of each process. After a process arrived it would run to completion and when it finished (cycles = 0) we would free the memory it allocated regardless of whether or not another process was waiting on that memory. The difference was that we actually had processes waiting on the memory to become available in certain instances. This was because a prior process could have allocated too much memory and there would not be enough memory available while it is running in order for the next process to allocate its required memory and run.

For the third part of part 3 we allocated a set 1 MB block of memory for the processes to use. This was again very similar to the previous part. We had many more instances of processes waiting for memory to become available in this instance because of the very small amount of memory that was available relative to the amount of memory that each process used. For this reason our runtime was generally the largest for this function.

Shared Assumptions and Limitations:

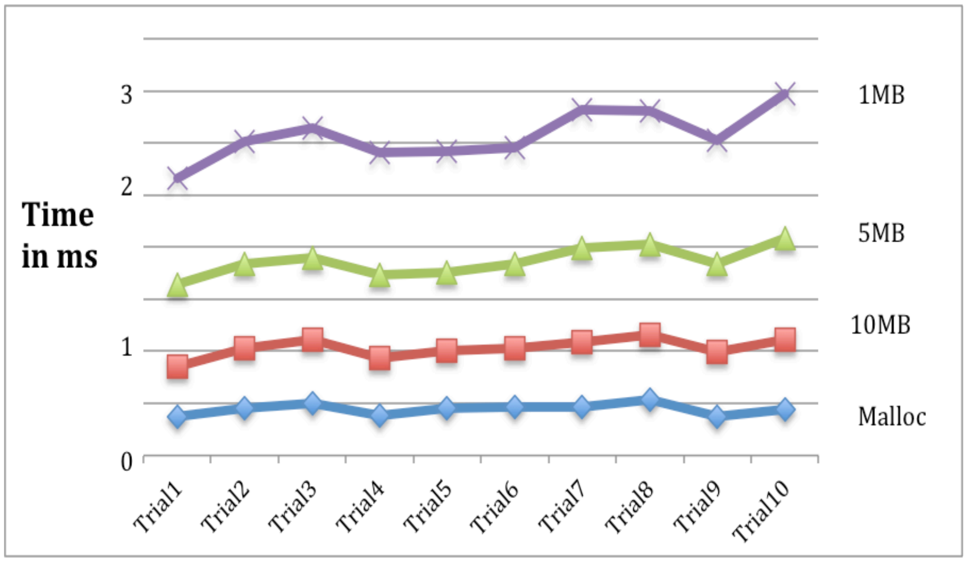
All of the 3 parts have some assumptions and limitations in common. First, our process generator generates 50 processes at random, with the amount of memory size in range between 1KB and 500KB with a mean of 150KB. The range of cycles is between 100 and 1000. Second, all of these processes are expected to arrive in the system every 50 cycles, starting from 0. Finally, it is assumed that there will be no more than one process generated in a certain set of processes that share the same number of cycles because of its low probability. The process to arrive first will be positioned in the CPU before the other process that arrives later, that shares the same cycles as the first one. All of tests are run on the CSE machine which can also cause errors depending on the traffic.

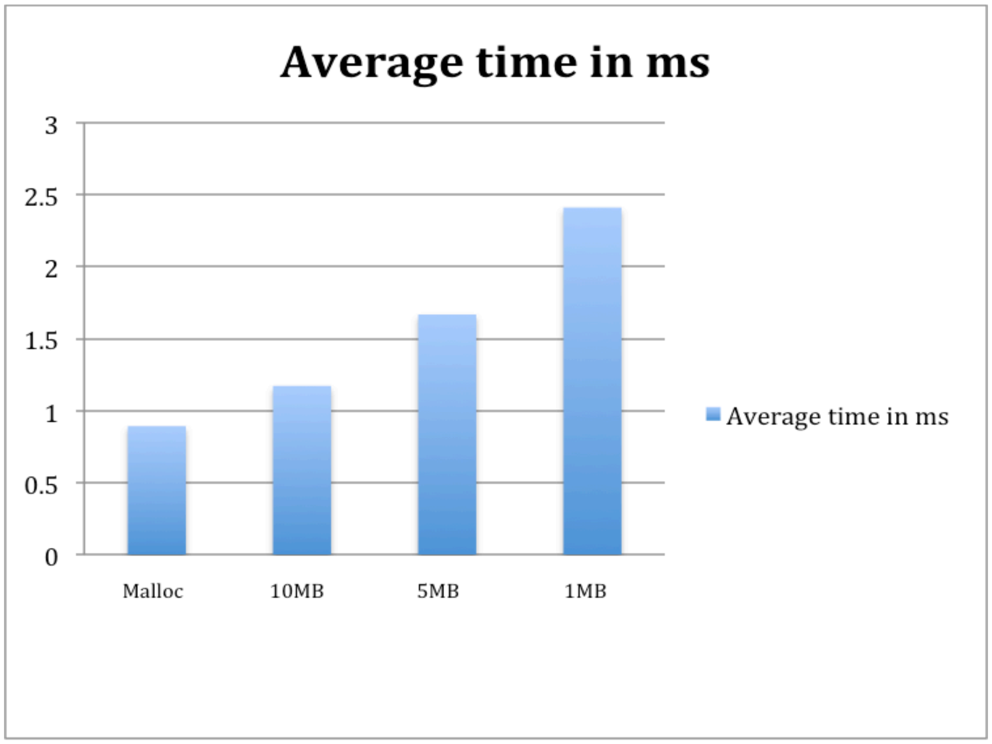
All the information above is shared among using malloc() and free(), 10MB block, 5MB block, and 1 MB block. Unique assumptions and limitations, if there were any, will be described in its own section.

Comparisons/Differences:

After we ran the project 10 times, we got the table below. When comparing the runtime of our 3 allocation constraints and with malloc() function (see the graphs below), we found that the one with the longest runtime was generally the 1MB version because of the fact that not all processes could allocate memory and run because many were waiting for the memory to become available. The second longest runtime was the 5MB version because although It only had 50% of the memory, each process deallocated its memory as soon as it completed. Of course the third largest runtime was 10MB because of it having no processes waiting. The shortest runtime of all was malloc in part 1 because of the fact that we are not waiting for processes to finish and we are not allocating and deallocating when each process finishes, we are only deallocating at the end.

System Time in MS				
Trial	MALLOC	10MB	5MB	1MB
Trial 1	0.749	0.949	1.59	2.03
Trial 2	0.918	1.13	1.64	2.33
Trial 3	0.998	1.23	1.573	2.483
Trial 4	0.773	1.098	1.603	2.345
Trial 5	0.917	1.103	1.496	2.314
Trial 6	0.926	1.139	1.605	2.23
Trial 7	0.926	1.25	1.798	2.657
Trial 8	1.071	1.236	1.743	2.567
Trial 9	0.752	1.241	1.688	2.368
Trial 10	0.889	1.339	1.94	2.783
Average	0.8919	1.1715	1.6676	2.4107





Conclusion:

The system time of each trial might be affected by the environment, such that the other functions which run on the same computer. The function might affect the system time. In addition, because we use the CSE machine to run the project, which means that we need to connect to the CSE machine through internet. The connecting quantity might affect the time also.

After running all of the tests and comparing our data we concluded that the more memory we had the better overall outcome we had in terms of runtime. Each function took longer to execute than the one that had more available memory. This is because of the many deallocations and processes waiting on others to deallocate from memory before running. This proves that more memory is better in almost every case.