University of North Texas

CSCE 4600 Project 2

Jason Hoang

Samuel McGregor

Taylor Condrack

Professor Keathly

May 3, 2016

## Introduction

The purpose of this project is to implement and experiment with C’s Malloc and Free system calls alongside our own implementation of a memory management system using our own versions of malloc and free. We will be using the same random number generator that we used and discussed in Project 1 to generate each process with the appropriate number of cycles and memory footprints. The random generator will be properly seeded so that we utilize the same processes for each experiment as needed. All of the processes will enter the system in 50 cycle increments (0,50,100,150,...etc.) and they will be executed contiguously to completion as long as there is memory available to be allocated to each process that requests it. As stated in the project, we will experiment with 64 different processes all of which will initially have a combined total of memory that does not exceed 20MB for parts 1 and 2 of the project. This limit number will change for part 3 when we experiment with lower the limits of 50% and 10% of the original. The final results that we will gather from each experiment will be the time of how long each algorithm takes and the comparison of how the change in available memory will affect the amount of time our implemented memory management system requires when the system does not have more than enough available memory for all of the processes. Throughout implementation we will also observe how each form of memory management operates within a system of process and discover how we can best utilize and optimize our own memory management to output efficient results.

## Malloc and Free System Calls

Since there is no need for scheduling, all of the process that enters the system will be allocated memory if enough exists and immediately starts executing. For use of the Malloc and Free system calls, the algorithm will have no problem allocating enough memory to each process because it is allocated from the cse machine. Since the amount of memory generated for each process totaled across all 64 processes will never exceed 20 MB, there will never be a problem with allocation. The algorithm pulls all of the 64 generated processes into the execution algorithm, but does not execute them all at once.

Within the algorithm that we used for Malloc and Free system calls, all of the processes are checked every cycle for three conditions: if they are arriving into the system, running in the system, or have completed execution. For arriving processes, the algorithm checks the process’s arrival time and compares it with the current system time. If the two times match up, the system call for malloc is appropriately called to allocate memory for the process. The process’s memory size is passed through the function and the function returns an integer pointer to where the memory is located within the system. This pointer is then stored within the process’s class data structure until it is freed by completion. If memory cannot be allocated for the process, then the process is placed in a ready queue and the system continues to cycle through the other executing processes until memory is freed up for the waiting process to allocate. Memory will only be allocated to the first process in the queue that the system has space for which will be checked every cycle. The process arriving section of the algorithm then subtracts 1 cycle from the set number of cycles required for the process to execute, which was randomly generated, sets the process to running and then adds 1 cycle to the overall time of system execution.

The other section of the algorithm is the actual execution of each process. If the process has already entered the system, is running, and has not completed, the process is executed for one cycle and then the algorithm checks to see if the process has completed. If the process has completed after that single execution cycle the process is marked as complete, and the memory that was allocated is freed when the algorithm calls the Free system call. This whole execution process is checked for every single process running in the system. A full unit of time is not complete until all of the 64 processes in the system have been checked through both sections of the algorithm, first arriving into the system or currently executing in the system. If the process has completed execution in the system, then it is continuously skipped from the checking process until the algorithm has completed. This is not the most optimal method, but for our simulation purposes it produces the ability to experiment and gather data correctly. The algorithm will not complete until all of the process have been fully executed. Once all of the processes have completed execution, the total time that the algorithm took to execute all of the processes in the system contiguously is returned to main to be recorded as data.

## My\_Malloc and My\_Free Functions

After we have completed the experiment using the malloc and free system calls every time a process requests memory, we will now use an exact copy of the processes to test our own memory management system using only one malloc and one free system call. We will test this set of processes three times: once with 100% of memory available, once with 50% of memory available, and then once with only 10% of memory available.

To simulate our own memory management functions, we use three (3) global functions and an integer array for the blocks of memory (“RAM”) that our functions manage. Initially, the main function calls our memory initializer function, “allocateMEM”, to call the function malloc and allocates 20MB of memory for our integer array. Our function then proceeds to mark every block with a number one (1); this means each memory block is free for a process to allocate and use. We also utilize a global boolean variable signifying if the memory was allocated correctly. In this case, it would be set to true after we make the one time function call malloc. Otherwise, the variable remains false. Lastly, “allocateMEM” returns true if the allocation succeeded, otherwise, it returns false. Only one context switch is necessary throughout this entire function.

Now that we have an entire integer array of “free” blocks, we can use our function called “my\_malloc” to allocate our “RAM” to given processes. This function accepts one argument; the number of blocks to malloc, namely the number of consecutive blocks in “RAM” the process needs in order to run. It returns the offset (starting at 0) of the starting location in memory of the number of consecutive blocks that was found. If a large enough memory block was not found, it returns a negative one (-1). Once a process has started and is ready to run, the process calls our function “my\_malloc” and sends the number of blocks it desires to be allocated. Initially, the function checks if the memory was allocated via the global variable stated in the above paragraph(allocateMEM). If the memory was not initialized correctly, then this function exits immediately by returning negative one (-1). If the number of blocks to allocate that was passed in was less than or equal to zero (0), then it would also exit by returning a negative one (-1). We used the “first fit” method of managing memory, so “my\_malloc” traverses the array of memory blocks from index zero (0) all the way up to the last index in “RAM”. While it traverses the array, we have a temporary counter that counts the number of consecutive memory blocks. This counter is reset to zero (0) every time the current index points to memory location that is in use. At every iteration of the loop, it checks if the temp counter equals the number of blocks requested to malloc. If these two values equal then we have found a block of memory big enough for the process. Since the index is currently pointing to the last element in the block of “RAM,” we loop backwards in the array for the number of blocks requested whilst setting the blocks to used or, in this case, zero (0). Finally, we are left with an index pointing to the start of the block of now used memory. This offset is then returned to the calling function. A context switch is not necessary throughout this entire function.

After a process has no more cycles to complete (i.e. is finished running), it calls our function “my\_free.” This function accepts two arguments; the offset (index starts at 0) of the memory location in “RAM” and the number of blocks to free from memory. It returns negative one (-1) if our memory has not been allocated through the aforementioned allocation function, if the parameters are out of bounds., or if the sum of the offset and number of blocks to free are greater than the total size of our “RAM.” If the parameters are valid, the function loops from the offset to the offset plus the number of blocks to free whilst setting each block to free or one (1) in this case. Lastly, the function returns the number of blocks that were freed. A context switch is not necessary throughout this entire function.

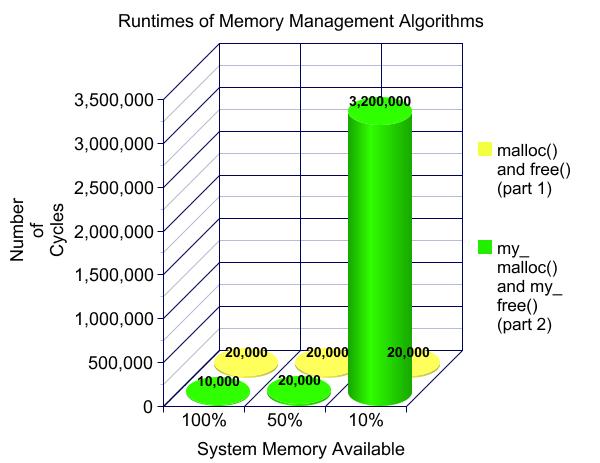
## Data and Analysis

Our experiment measured the runtime of the normal malloc and free method of memory allocation and our own memory management system. The tests represent one execution when the system has 100 percent of the required memory of the processes, when the system has 50 percent of the required memory, and when the system only has 10 percent of the memory required by the 64 processes. We repeated this test on a set of three distinct sets of processes that vary in the average memory footprint for each process. Although the data will be negligible if not the same for the method used in part one for each test, our results include the same experiment performed on the malloc() and free() method for the sake of comparison. The experiments use clock cycles as the unit of measurement to record the runtimes. Clock cycles were used as a convenient unit that was precise enough to present our data with significant values for all tests. Each test was performed using the same set of processes. The average number of cycles for our set one of the 64 processes was 6267. The average size of the memory for each process was 159. For set two the average size of the memory each process requests was 222, and 271 for the third set of processes. Our program assumes that the system is a single core system. The processes are queued to be allocated memory from our memory pool in the order they arrive. Our program uses a nonpreemptive method. We chose to simplify the process arrival so our program’s runtime data would not be influenced by different methods of choosing processes. If a conflict occurs and there is not enough memory for the process, the process is queued and will be allocated memory in the order it was put in the wait queue.

We used a static defined integer to store the amount of memory the system had available. We converted the 20 MB to 20480 KB to work in kilobytes so it would be compatible with our previous programs. To reduce the system's available memory we simply assigned the variable to 50% of the original value and then 10% of the original value. The algorithm makes a single call to malloc to initialize the available resources in our pseudo memory block.

As shown in our data for figure one, the amount of time for our system to allocate memory to all 64 processes given the entire 20 MB was negligible. Our program was able to allocate and deallocate each process as they arrived without conflict. Because the max amount of memory each process ,in the first data set, could potentially have is 320 bytes, each process will not have to wait to receive memory from our allocated block. The processes will finish before there is a backup. Our algorithm calls the function call malloc only once, so this also cuts down on context switches compared to the method used in part one. Hence, the program runs very quickly: within 10000 clock cycles. When the available memory the system could offer the processes was reduced to only 10 MB, the program still ran in only 20000 clock cycles. The amount of time a process had to wait if there were a conflict was very small because the number of conflicts was minimal. When the system only had 10 percent of the 20 MB available it took our program 3200000 clock cycles to finish executing. This was approximately three seconds. With only 2048 KB available, there were many processes that got queued in our wait algorithm.

In figure two, we compare the results with three sets of processes with different sizes of memory requests. The data shows that with a higher average size that each process is requesting, the longer the program will take to execute. The cycles are using up the available block of memory at a faster rate, causing more processes to have to wait for their request to be granted. For an example. When the system had only 10 percent of the available memory the program took 3200000 cycles with a memory footprint average of 159, 5360000 cycles with a memory footprint average of 222, and 6990000 cycles with a memory footprint average of 271. This shows that the more memory that each process requests will lead to the queue being backed up.



-Figure 1

Runtimes of Memory Management Methods for 3 Sets of Processes

|  |  |  |  |
| --- | --- | --- | --- |
|  | 100% memory available | 50% memory available | 10% memory available |
| Process set 1 (avg memory 159) | 10000 clock cycles | 20000 clock cycles | 3200000 clock cycles |
| Process set 3 (avg memory 222) | 10000 clock cycles | 1430000 clock cycles | 5360000 clock cycles |
| Process set 2 (avg memory 271) | 10000 clock cycles | 2510000 clock cycles | 6990000 clock cycles |

-Figure 2

## Conclusion

In our experiment we implemented and tested two methods of memory management. We simulated three consistent sets of 64 processes requesting memory from a memory pool of set size. We compared using the system calls malloc() and free() to allocate and free portions of the memory for each process, and we created our own allocating and deallocating system that reduced the number of context switches needed to assign memory to the arriving processes. As we predicted, our method ran slightly faster than using C’s system calls due to the need to context switch while using malloc and free. We can also conclude from our tests where we reduced the amount of memory available to use for the system of processes that as our memory decreases, the time that it takes for all processes to finish executing within the system greatly increases. The same effect is shown to occur if the processes request a greater amount of memory. The overhead decreases, forcing more processes to wait. Our experiment did have some limitations. Our system does not represent a real life scenario very accurately in which there might be some processes that have priority. We also are not implementing swapping or paging. Nevertheless, our experiment did show that the amount of context switches will affect the runtime and the amount of available memory will also affect the runtime.