Project 2: Memory Management

*University of North Texas*

*CSCE 4600*

The purpose of this experiment was to create a memory allocation and deallocation function and compare the results to the library functions Malloc and Free.

Connor Bey, Bob Jack, Alexander Hollis

5/8/2015

Project 2: Memory Management

University of North Texas

# Introduction

For this experiment, a memory management system is designed to allocate memory to a predefined memory block. Using 50 processes that run at varying cycles, the memory is partitioned inside the memory block. After each process completes, the allocated memory is returned to the available memory pool, with an adapted version of *free ().* Each process will enter the system every 50 cycles and check if there is room to be allocated. All processes will run until completion and then exit the system. A process will only enter the system if there is room for it to be allocated.

The created memory manager will be compared to the *malloc ()* and *free ()* library functions. Time trials are established with 10 sets of different processes. The same initial queues will be used for the *malloc ()* and *my\_malloc ()* functions per trial. This will give each function the same data and process order to deal with. Having 10 sets will add a variety of memory sizes and positions that each function will have to use.

Initially, we believe that *my\_malloc ()* will run faster because one context switch is called as the memory is first allocated by a *malloc (),* as opposed multiple context switches from using *malloc ()* each time a process needs memory. Overall, the reduction in context switches should cause a significant reduction in runtime.

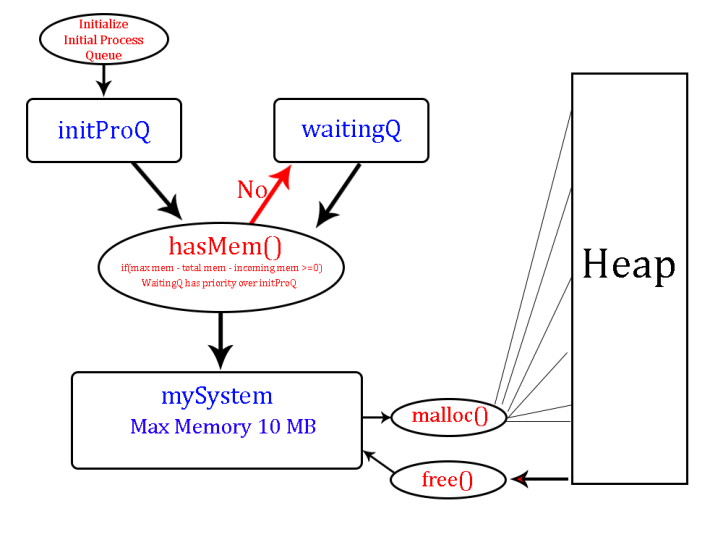
# Problem 1

## Analysis

The first problem establishes a scheme where 50 processes can enter a system that allocates memory using the *malloc ()* function call. Each process has a random cycle and memory size set up in an initial queue. After a process is assigned a memory address, it exits the system when the process cycle count completes. The memory used by the process is returned by a library call to *free ().* A timing system is setup to measure the total time taken to run each trial from the start of the first memory allocation until the last process exiting the system has its memory freed.

## Algorithm Design and Analysis

An initial process queue of 50 processes with varying cycles and memory requirements is established. Every 50 cycle ticks a process will be admitted to the system. If the memory of the admitted process will fit into the maximum size - in this case it is 10 MB, then it will be admitted to enter the system.

If the process does not fit in the memory requirement, then it will be pushed into a waiting queue. This cycle will repeat every 50 ticks. Upon the next iteration, if there is a process sitting in the waiting queue, it will take a priority over a process sitting in the initial queue. This will prevent starvation of processes in the waiting queue. The waiting process will see if there is space for it to enter, if not it will pass to the next process in waiting. If the processes in the waiting queue are too large, then the processes in the initial queue are checked for eligible candidates.

Any process that can fit is allowed to enter the system. Other process that cannot will enter the waiting queue. This system is constructed to handle additional processes placed into the init queue besides the original 50 processes.

### Issues

The algorithm was designed initially to accommodate problem #3 with memory constraints added. There were a fair amount of problems with memory failing to be allocated because the random memory generator was allocating memory too high. Infinite looping occurred because the memory had nowhere to go and the system could not cycle through.

The majority of problems came from trying to find a proper allocation of memory to fit with this algorithm and be uniform through other trials and size constraints. A value was decided upon that fit best was the maximum cap value of 10 MB divided by 16. This worked well in the random memory generator and gave values that worked uniformly in all cases.

# Problem 2

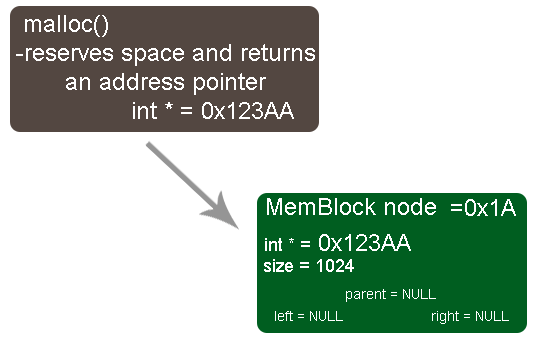
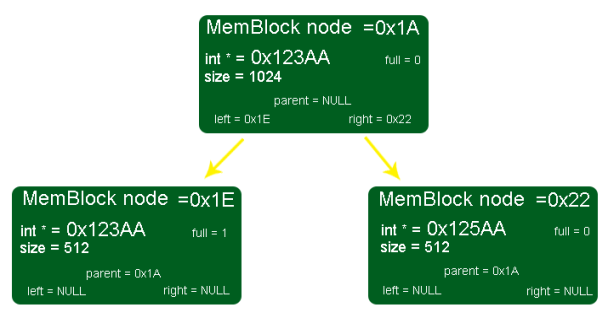
## Analysis

The second problem is to create a memory manager that allocates memory using functions named *my\_malloc ()* and *my\_free ().* A predefined block of 10 MB of memory is first reserved using a call *malloc().* The constructed functions handle the allocation and deallocation of memory inside of the reserved memory space. This should minimize context switches and runtime. Therefore the runtime of *my\_malloc ()* should be faster than the runtime of the created algorithm for *malloc ()* in Problem #1.

## g3.pngAlgorithm Design and Analysis

A buddy system is utilized for memory management. To simplify the implementation of this system, a binary tree data structure handles the allocation of the memory addresses within the initially allocated   
10 MB *malloc ()* block.

Each structure contains an integer pointer to hold the memory addresses. The four-byte size of the integer pointer is taken into consideration when handling memory address sizes.

The first address of the *malloc ()* block is placed into the MemBlock data structure as an integer pointer address. When *my\_malloc ()* is called, it calls a function named *findFit ().* This seeks out a proper space that exactly matches the incoming memory. *FindFit ()* uses a binary search to run down through the memory block tree to see if a block is available. It returns the integer pointer address where the actual memory on the machine is stored.

A round up function is used to round random memory numbers into the next highest power of two for ease in memory block fitting.

If an empty tree node is reached and is a leaf, then *findFit ()* will split that memory block into half. The left node will be passed the integer pointer value from the current node, and the right node will be passed the current node pointer, plus the size of the memory of the node at that level. This represents a data block in the already allocated *malloc ()* chunk.

This splitting process will repeat until a chunk is found. The pointer address will be returned and the node will be marked as full. The address is assigned to the process and placed into the system queue where it runs to completion.

When a process is finished, the memory will be restored using *my\_free ().* The tree accesses the address of the Memory Block, and removes the node. The stored Memory Block location is referenced and that node is marked as available. This returns the allocated chunk of memory back into the system for reuse. A coalescing check is done next. If the node is a leaf, is empty, and its buddy, is an empty leaf, then the two will be merged into the parent.

### Issues

Numerous problems arose from implementing a binary tree for the memory block buddy system. It took a while to sort out all of the issues with splitting and coalescing. Eventually it began to work well. Initially, we saw values that were unusually faster than the *malloc ().* A few bugs were discovered that we were scaling the maximum memory value along with the memory cap. This really did not place any constraint on the trial. Once this was corrected, we observed that the *my\_malloc* () was slower than the *malloc ()* in the normal constraint of 10 MB. It is still not determined if a bug exists to cause this.

Experimental Results

In the original experiment with 50 processes, we observed that malloc () ran faster than my\_malloc. In Part 3 when the constraints are tightened, my*\_malloc ()* becomes faster than the *malloc ()*. We decided to compare these two again under the normal constraints. We wanted to see if a trend would occur as we predicted where the increase in context switches will slow *malloc ()* down.

The graph on the next page shows our experiment to determine if the normal constraints will differ from an increase in the number of processes that enter the system.

The total number of processes was changed from 50 to 100, and then scaled up to 1000. As the process number grew, *malloc ()* stayed faster than *my\_malloc ()* under these conditions.

We initially thought that the context switches from *malloc ()* would bog it down, it appears that our algorithm could not overcome the increase in context switching with   
normal constraints.

# Problem 3

## Analysis

The third problem is to ensure that the system can handle tighter constraints on memory allocation requirements. Two extra settings are introduced where the maximum memory block available is reduced to 50%, then to 10% of its normal size. The random memory assignments are to remain the same.

## Algorithm Design and Analysis

The *malloc ()* and *my\_malloc ()* were separated by an integer type. This would alert the program on which one to run. For each trial, constraints are placed on the memory requirements. The memory is running normally at first, and then the cap will be reduced to 50%. The third run the cap is reduced to 10%. Ten trials are timed and then averaged. The results of these trials are shown below.

### Issues

Rounding up the random memory was also needed because of the random memory sizes. Random memory setup was scrutinized to ensure that each case (normal, 50%, and 10%) were close to being equivalent and only the memory cap was reduced. As a result of this rounding, internal fragmentation will not be truly represented, but would exist outside of  
this experiment.

## Experimental Results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Normal | | 50% Memory | | 10% Memory | |
|  | Malloc | my\_malloc | Malloc | my\_malloc | Malloc | my\_malloc |
| Trial | Time (ms) | Time (ms) | Time (ms) | Time (ms) | Time (ms) | Time (ms) |
| 1 | 328 | 343 | 406 | 375 | 2000 | 2055 |
| 2 | 330 | 359 | 312 | 312 | 1937 | 1890 |
| 3 | 250 | 281 | 374 | 377 | 1953 | 1328 |
| 4 | 344 | 343 | 375 | 373 | 2078 | 1203 |
| 5 | 326 | 341 | 406 | 453 | 1765 | 1187 |
| 6 | 265 | 265 | 375 | 375 | 1968 | 1328 |
| 7 | 262 | 260 | 359 | 359 | 2234 | 1680 |
| 8 | 281 | 281 | 390 | 383 | 1625 | 1687 |
| 9 | 297 | 312 | 390 | 390 | 1766 | 1203 |
| 10 | 284 | 311 | 359 | 328 | 1749 | 1375 |
| Total | 2967 | 3096 | 3746 | 3725 | 19075 | 14936 |
| Average | 296.7 | 309.6 | 374.6 | 372.5 | 1907.5 | 1493.6 |

Initially, it was believed that the *my\_malloc ()* function would have a faster average runtime than the *malloc ()* calls due to the context switching overhead. According to the trials that we ran, when the maximum memory size of 10 MB was used, the *malloc ()* calls were slightly faster. As constraints to memory requirements were applied, the experiment began to fit the initial expectations. At 10% memory, the difference was more significant. *my\_malloc ()* became faster with the constraint.

The above graph reflects our observations. The increase in memory constraints is measured by a factor of n that is applied to the cap that tightens the memory cap of 10 MB. As this factor shrinks the available memory, the runtime of the trials increase for both algorithms. At a point just after 50%, *my\_malloc ()* becomes faster and stays faster.

As the constraints are applied, both *malloc ()* and *my\_malloc ()* are having to wait a bit longer to assign the appropriate space for a process to run. At 10%, there is little room for a process to run, so many processes are waiting for others to finish. *my\_malloc ()* is handling the allocation much better than the *malloc (*) at this point. This could be due to the average runtime of the binary search of O (log2 n).

Since the function *my\_malloc ()* is handling the 10% memory constraint faster, the binary buddy implementation is more cost effective than the library function *malloc ()*. Meaning, if you want to optimize performance on a system, writing a *my\_malloc ()* is better than depending on a library function for tighter memory conditions. This holds true for an increase in processes as well.

# Conclusion

The *my\_malloc function* outperformed the *malloc function* under certain conditions. Also *my\_malloc ()* effectively can be used by any program to acquire an actual physical memory address located in the system memory. To reduce page swapping, having the memory addresses located in a tight area would be more efficient overall. Also, due to the reduction of context switches, time is saved. *My\_malloc ()* allows the user to control the storing of heap space instead of letting *malloc ()* decide.

If a simple system is used, or there is little overhead in space or processes, then *malloc ()* would be the optimal choice. However, external fragmentation is present when using *malloc*. If the system becomes involved and complex, *my\_malloc* would be the optimal choice. However, some internal fragmentation will be present when using my\_malloc.