Project 2 Memory Simulation

CSCE 4600 Operating Systems

Faris Hawamdeh

Dustin Fennessy

Eric Guzman

d

D

# Introduction

The purpose of this project was to develop and employ memory management systems that handle arriving processes. The processes were generated from our process generator which we used in Project 1. There are 50 processes for each run, with the arrival time every 50 cycles. These processes are assumed to run to completion, however, they can only run if they have enough memory.

# Memory Systems

Our memory management systems are written in two ways. Both of them are valid in that they work, but this project attempts to delve into the differences between the systems.

## Malloc()/Free() Management System

The first memory manager of our project

* Has a memory pool of 10MB
* Will only run a process if sufficient memory is available
* Uses calls malloc() and free() to allocate and de-allocate memory
* Relies on OS-specific systems calls, so performance may vary between OS’s

## Custom Management System

The second memory manager of our project

* Has a memory pool of 10MB
* Will only run a process if sufficient memory is available
* Uses custom-built MyMalloc and MyFree functions to allocate and de-allocate memory
* Does not rely on OS system calls
* Employs the buddy system in its implementation

# Test Environment

Our tests were run on the CSE machines. The general system specs for the machines are below.

* Run on Linux
* AMD 2200Mhz processor (with 1-3 cores)
* Barebones from user side
* g++ for compilation

# Memory Allocation Implementation

The buddy system was used for the memory allocation in our program. Memory is first created as a single block. The function initializeBuddy() is used to allocate the specified block of memory to be used to distribute to the processes that request it using the myMalloc() function. In the initializeBuddy() function, it first calculates the size of the array, then allocates the space required to store the array that holds the binary tree. Once the Memory Block has been allocated, the entire memory block is initialized to 0.

The system will partition this block as processes request memory using the myMalloc() function. The myMalloc() function first checks to see if the requested memory is larger than the total memory size, then it enforces the minimum block size by replacing the requested memory value with the minimum block size if the requested memory is less than the minimum block size. The next step the function performs is to calculate on what level of the binary tree the best fit block would reside. Once it has gotten the tree level of the best fit blocks, it searches through that level of the tree to find a node that it can divide to for the request. If it reaches the end of the tree level unable to find a free block, it returns a NULL signaling the request has failed. If there is a block found, it calculates the address of the allocated space using the tree level, block size, node location, and the address of the start of the memory block, and it returns that address. When a block is found it calls the markDivided() function to mark the nodes in the binary tree which identifies those blocks as having been divided.

When the process is done using the block of memory it will call the myFree() function to return the block of memory it is holding back to the memory manager. The function first calculates the offset by subtracting the address passed in from the address of the start of the memory block. If the offset is negative, the function returns false to signal a failure. Next, the function checks the memory size passed in to enforce the same minimum block size rule as in myMalloc. After that, it uses the memory size to determine the tree level the block assigned to. Once the tree level has been calculated, the function calculates the node location in the binary tree using the tree level, offset, and the block size of that tree level. Using the node location, it marks the node as unused in the binary tree, and calls the markMerged() function to merge the unused blocks back together. At the end the function returns true to signal success.

Once the simulator is done with using the buddy memory manager, it calls the function freeBuddy() which frees the memory block and the buddy array.

# Experiment

In order to test the efficiency of the two memory management systems, we run our program 5 times for each system. We compare data such as total process time and cycle distributions between the two systems. The results of our experiment are posted below along with data charts and summaries to explain some of the aspects of the comparisons.

## Total Process Time Comparisons

In the table below we show the comparison between the process times between malloc/free vs MyMalloc/MyFree. We performed three tests five times. The first test used small memory processes (100 bytes), the second used medium memory processes (200 bytes), and the third used large memory processes (1000 bytes).

In the first test, MyMalloc/MyFree consistently beat out malloc/free in total process time.

In the medium memory processes test, MyMalloc/MyFree began to experience higher total process times while malloc/free where consistent with their speeds in the first test. Both were fairly close in total process time in this experiment.

In the third and final test, we used large memory processes. Like before, malloc/free are still consistent in the speeds they achieved. MyMalloc/MyFree gained even more process time during this test and obtained consistently worse process times during this test.

As you can tell from the data, MyMalloc/MyFree has a problem with larger memory size processes. In the first two tests it could keep up with malloc/free, but clearly malloc/free are more optimized to handle memories of varying sizes.

# Appendix