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).

### Small:

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

|  |  |  |
| --- | --- | --- |
|  | Cycles | Memory KB |
| Average | 5531 | 97 |
| Range | 12094 | 184 |

### Medium:

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.

|  |  |  |
| --- | --- | --- |
|  | Cycles | Memory KB |
| Average | 5514 | 208 |
| Range | 8163 | 398 |

### Large:

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.

|  |  |  |
| --- | --- | --- |
|  | Cycles | Memory KB |
| Average | 5468 | 1109 |
| Range | 10139 | 1920 |

The data shows, MyMalloc/MyFree have problems 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.

# Conclusion

Malloc/free are better equipped to handle scenarios where lots of memory needs to be allocated, particularly when then the memory that needs to be allocated exceeds the total available memory. With medium sized processes (approx. 200kb) MyMalloc/MyFree performed comparably with malloc/free, only in 1 out of 5 runs did our functions perform worse than malloc/free. For the small size processes (approx. 100kb) MyMalloc/MyFree had better results than malloc/free.

# Appendix

### Sample Input for Small

ProcessID NumberOfCycles (Millions) Memory ( kb )

0 2976 88

1 4539 117

2 1589 111

3 4811 102

4 260 110

5 5160 14

6 7841 116

7 1290 193

8 2388 86

9 7918 31

10 7053 180

11 5855 163

12 5175 90

13 12354 56

14 3787 71

15 8206 183

16 5974 110

17 6369 77

18 4990 61

19 3804 11

20 3931 111

21 7710 72

22 870 63

23 7648 47

24 5968 104

### Sample Input for Medium

ProcessID NumberOfCycles (Millions) Memory ( kb )

0 5227 169

1 7549 243

2 6748 158

3 7436 213

4 7145 50

5 4111 149

6 7961 142

7 4275 312

8 5516 102

9 7453 344

10 7295 276

11 4863 143

12 5770 177

13 4509 78

14 3084 183

15 8601 141

16 2380 342

17 8549 206

18 4269 196

19 8030 130

20 5503 254

21 5412 254

22 6785 420

23 4922 178

24 6981 22

ProcessID NumberOfCycles (Millions) Memory ( kb )

25 7554 45

26 7378 82

27 4616 82

28 5116 37

29 7439 127

30 7114 143

31 2329 121

32 773 27

33 5979 165

34 2882 158

35 10598 134

36 4452 139

37 8309 106

38 5229 9

39 3435 68

40 4716 33

41 4569 70

42 5806 81

43 8434 74

44 8252 162

45 6651 109

46 2492 203

47 6400 133

48 7154 38

49 10430 131

ProcessID NumberOfCycles (Millions) Memory ( kb )

25 7722 64

26 6963 305

27 7446 322

28 3488 298

29 3996 264

30 4710 230

31 5710 171

32 7583 200

33 7559 327

34 5658 233

35 3090 102

36 3737 307

37 2666 130

38 3218 43

39 3808 187

40 8294 252

41 1241 316

42 4160 364

43 5923 262

44 7326 23

45 6147 346

46 7405 205

47 438 132

48 1431 185

49 5608 266

### Sample Input for Large

ProcessID NumberOfCycles (Millions) Memory ( kb )

0 8507 1041

1 10309 2080

2 10859 1336

3 5379 1370

4 3156 573

5 1765 1719

6 6082 1636

7 5792 650

8 3481 342

9 6313 873

10 4088 719

11 1934 1437

12 7480 861

13 6323 1051

14 5096 1444

15 5233 1365

16 3809 1674

17 5753 1107

18 4373 1344

19 7231 944

20 5242 1622

21 6607 494

22 5385 265

23 3125 1234

24 6395 1314

ProcessID NumberOfCycles (Millions) Memory ( kb )

25 6257 1084

26 6453 599

27 4702 1451

28 5951 1255

29 5007 1192

30 6925 612

31 6374 1226

32 3304 1326

33 8404 1552

34 3681 578

35 3167 895

36 5763 1554

37 11904 332

38 4168 1486

39 5023 883

40 3564 1517

41 6081 973

42 7296 1257

43 2601 160

44 6369 1768

45 5663 1359

46 2292 577

47 4128 1457

48 4059 1273

49 4523 581