Cmpsc 473 Programming Assignment 3- Report

Group members:
Vibhu Patel
Jon Dahl
Sanyukta Baluni

4.1 Workloads- refer to the .txt files on github

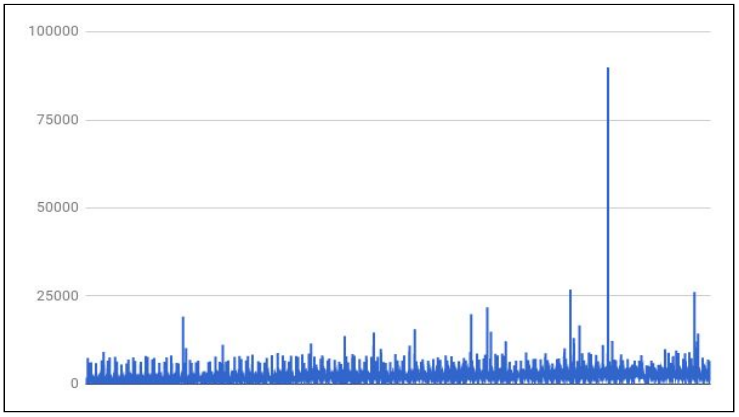
4.2 Measurements

Measure Malloc

For all the graphs: **(x - axis represents processes and y-axis represents the time)**

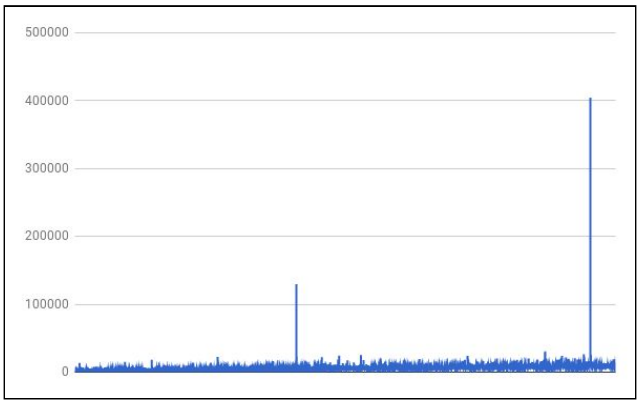
Small - Best

Average = 1991.7526 nanoseconds
Median = 1746 nanoseconds
25th percentile = 1257 nanoseconds
75th percentile = 2444 nanoseconds



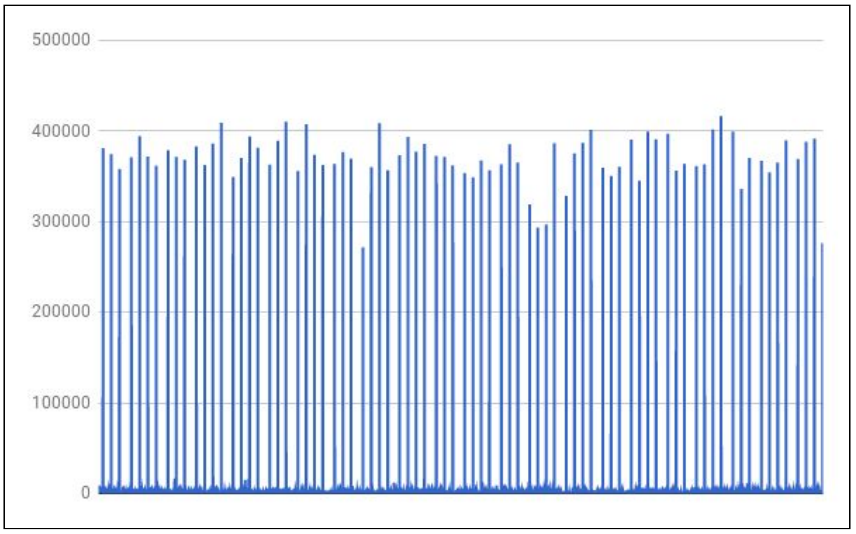
Small - Worst

Average = 5108.3269 nanoseconds
Median = 4819 nanoseconds
25th percentile = 2864 nanoseconds
75th percentile = 6845 nanoseconds



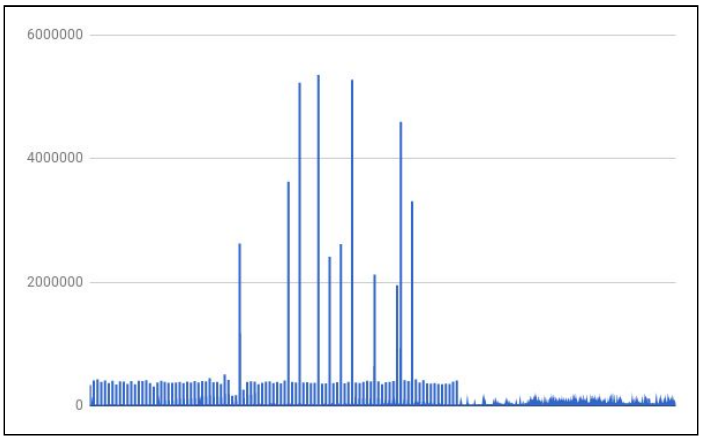
Mixed - Best

Average = 3838.8869 nanoseconds
Median = 908 nanoseconds
25th percentile = 768 nanoseconds
75th percentile = 978 nanoseconds



Mixed - Worst

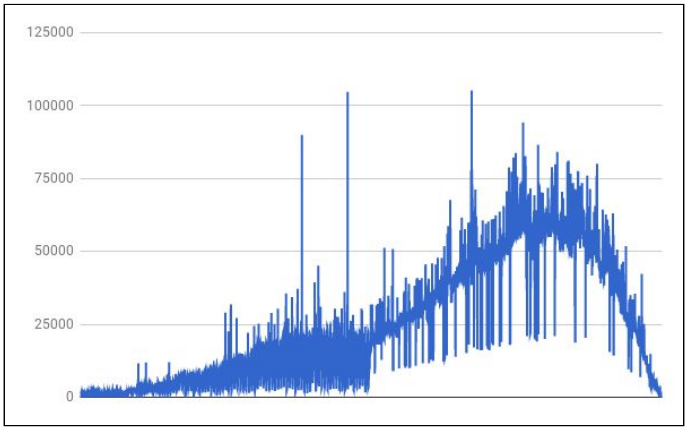
Average = 13778.9167 nanoseconds
Median = 4400 nanoseconds
25th percentile = 2444 nanoseconds
75th percentile = 8939 nanoseconds



Measure Free

Small - Best

Average = 24153.8451 nanoseconds
Median = 18997 nanoseconds
25th percentile = 4959 nanoseconds
75th percentile = 42463 nanoseconds

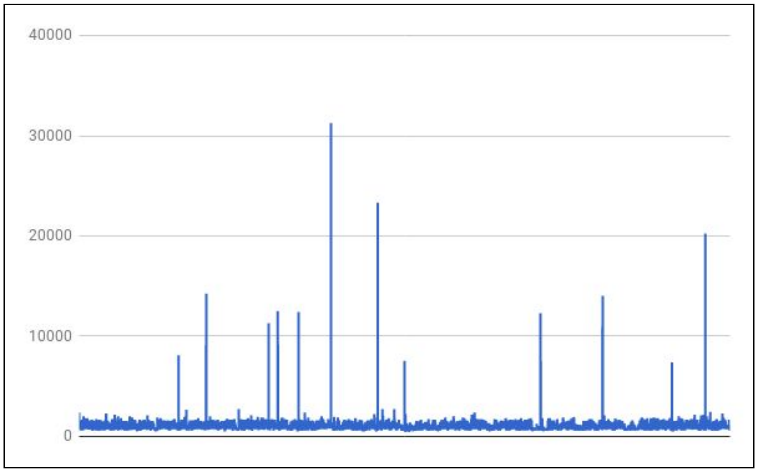


Small - Worst

Average = -41293.5702 nanoseconds
Median = 39111 nanoseconds
25th percentile = 10965 nanoseconds
75th percentile = 99942.5 nanoseconds

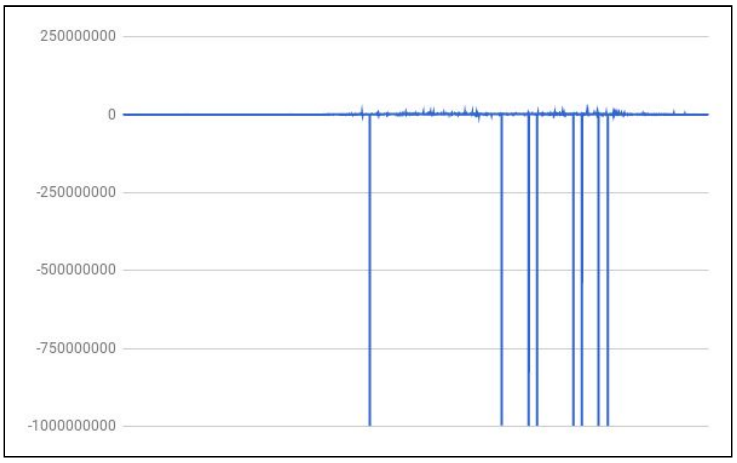
Mixed - Best

Average = 1044.3812 nanoseconds
Median = 1047 nanoseconds
25th percentile = 768 nanoseconds
75th percentile = 1188 nanoseconds



Mixed - Worst

Average = -144780.016 nanoseconds
Median = 342325.5 nanoseconds
25th percentile = 39442.5 nanoseconds
75th percentile = 1132053.5 nanoseconds



Table

(Times in nanoseconds)	**Average**	**Median**	**25th percentile**	**75th percentile**
Malloc(small-best)	1991.7526	1746	1257	2444
Malloc(small-worst)	5108.3269	4819	2864	6845
Malloc(mixed-best)	3838.8869	908	768	978
Malloc (mixed-worst)	13778.9167	4400	2444	8939
Free(small-best)	24153.8451	18997	4959	42463
Free(small-worst)	-41293.5702	39111	10965	99942.5
Free(mixed-best)	1044.3812	1047	768	1188
Free(mixed-worst)	-144780.016	342325.5	39442.5	1132053.5

The malloc_fail.txt is currently an empty file (not shown on Github currently as it is an empty file) But if reset.sh is run, there will be a new malloc_fail.txt file will be generated (as referenced in the README)

-> Number of occasions when a psumalloc() could not be satisfied due to lack of memory- there will be a malloc_fail.txt and it will display “lack of memory has occurred”.

-> Number of occasions when a psumalloc() could not be satisfied due to external fragmentation - there will be a malloc_fail.txt and it will display “ external fragmentation has occurred” and it occurs inside the allocated memory for the process.

5.1

In this project we used a linked list to store data. The possible data structures were: Array, Red-Black Tree, and Linked List. With link lists we can insert and remove elements with constant time complexity, where the same operations have a linear time complexity in an array and a logarithmic time complexity in a Red-Black tree. To insert elements into an array, all the elements after that point need to be shifted. This would require the implementation of another array structure or complex operations. In an array, you can access an element directly by specifying the array index so the time complexity is O(1) whereas in a linked list an element is accessed sequentially ie traversing starting from the first node in the list by the pointer so the time complexity is O(n). But when you look at the insertion and deletion of elements in an array, that takes O(n) time whereas a linked list only takes O(1) so it is very fast. Although an array has constant access time, the common case of insertions and deletions should be prioritized. Red-Black tree operations are all of log n time complexity, but implementing this structure in C would be very difficult: not to mention the complexity in organizing the structure. In order to prioritize the common case and maintain efficiency, a linked list fulfilled all these requirements.

The project contains a psumemory.c and a psumemory.h file and the psumemory.c has the functions:

int psumeminit(int algo, int sizeOfRegion) it is called one time by the process to use the memory allocator. And algo represents which memory allocation policy should be used -0 for best fit and 1-worst fit. sizeOfRegion is the bytes of memory you request from the OS using mmap

void* psumalloc(int size) it takes as input the size in bytes of the object to be allocated and returns a pointer to the start of that object.

void* splitting(node_t* p, node_t* op, int size) it splits larger memory into smaller ones and the requested space is returned while the leftover space remains on the free list

void coalesce() it looks for memory chunks that can be combined

int psufree(void* ptr) this function frees the memory that the pointer points to

void psumemdump() this function is simply for debugging purposes

We are using a function **clock_gettime(CLOCK_MONOTONIC, &s_begin)** to measure performance of malloc and free.

Ambiguities in the specification:

Refer to the README.md

We could not measure internal fragmentation without additional information regarding the use of the allocated space. Internal fragmentation occurs when there is free space in the regions our function allocated. External fragmentation occurs when there is free space between the allocated regions. Our psumalloc function tracks the amount of free space between the allocated regions and upon failure will place a flag in a the appropriate file (malloc_fail.txt). If there is not enough free space to allocate a request, a different flag will be placed in the same file indicating that error.

All the features in the project work correctly and there are no bugs

All the graphs and tables are included in 4.2

Free small-worst and free mixed-worst have a very high average time and it is giving a negative value due to an overflow