Rüzgar Ayan 21801984

2020-2021 SPRING CS342 PROJECT 3 Report

Experiment 1. Measuring the Fragmentation Amount

In this experiment, I will try to measure both the internal and the external fragmentation. Actually, we can say something about the internal fragmentation before the experiments. If the requested allocation sizes are totally random, since the actual allocation are the powers of 2, the internal fragmentation should be about 25%.

And in my experiments, I will also use random sizes in allocation request.

Another important point for the measurement is that, if a user requests 300 bytes allocation and library gives them 512 bytes, I will consider this remaining unused 212 bytes as external fragmentation. I called this 512 bytes as real allocation size in the table below.

In the experiment, I continiously allocated memory (without freeing) from the library until I got one NULL pointer.

I have initialized the sbmemlib with 256KB = 262144 bytes shared memory segment. And in the different experiments, I have changed the interval of the random size of allocation request. Results are as follows:

Interval of the Random Allocation Request Sizes	Total User Allocation Size	Total Real Allocation Size	Internal Fragmentation Percent	External Fragmentation Percent
128 - 256	176081	262144	32.83%	32.83%
128 - 512	183085	261888	30.09%	30.16%
128 - 1024	189032	261632	27.75%	27.89%
128 - 2048	192482	261376	26.36%	26.57%
128 - 4096	194727	260608	25.28%	25.72%

Table 1. Experiment 1 results without freeing any blocks.

Although they will be more or less the same, I have repeated the experiments in the following way: This time, I free'd 99% of the allocations that I've made so that the the shared segment will take more time to fill up.

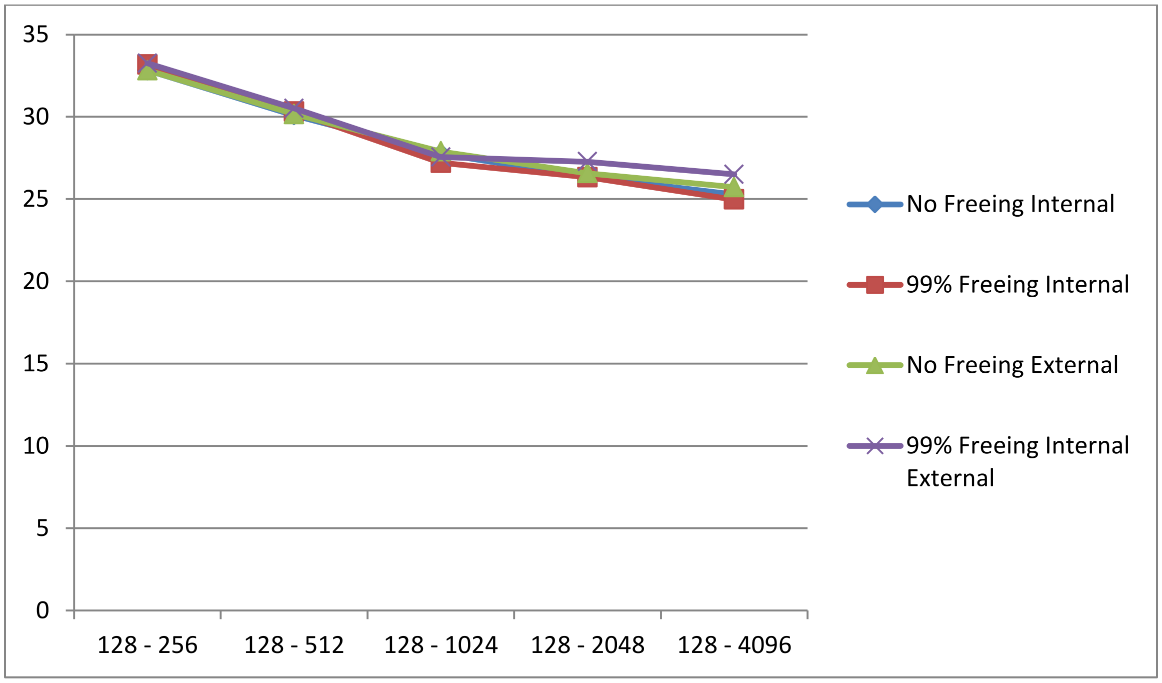
Interval of the Random Allocation Request Sizes	Total User Allocation Size	Total Real Allocation Size	Internal Fragmentation Percent	External Fragmentation Percent
128 - 256	174990	261888	33.18%	33.25%

128 - 512	182165	261376	30.31%	30.51%
128 - 1024	189921	260864	27.20%	27.55%
128 - 2048	190657	258816	26.33%	27.27%
128 - 4096	192663	256768	24.97%	26.50%

Table 2. Experiment 1 results by freeing 99% of the blocks along the way.

First observation from these results is that the "Total Real Allocation Size" is always very close to the the total shared segment size which is 256KB = 262144 bytes. Because of that, the internal fragmentation and the external fragmentation becomes almost the same. This is because of the properties of the Buddy allocation algorithm. Even the maximum request size of 4KB is small compared to the total shared segment size. These small requests can easily fill up the empty spots.

Just like the theoretical guess of 25% internal fragmentation, the experiment results gets closer to 25% as we increase the request sizes. When we have smaller request sizes for example in the range 128-256, the overhead of 16 bytes that store the information about every memory block becomes more significant. Because of that, the percentage of the part the user can really use decreases and this increases the fragmentation.



Experiment 2. Performance Comparison with Standard malloc() function

Although the project document was asking for experiments on fragmentation, I was actually wondering whether this will be faster than using the malloc() function. So I did some experiments on this too.

All of the following experiments are done with a 256KB shared memory segment. Normally, the project document states that 4KB is the max request size, but I removed this condition from the code for this experiment to see the results better.

For different request sizes, I have allocated a memory and then immediately free'd it. I have done this for 100,000 times for each request size. First, we can measure the worst case performance of our library as follows:

```
char *p;
 for (int i = 0; i < times; i++)
 {
    p = sbmem_alloc (allocSize);
    sbmem_free (p);
 }
```

If the memory is initially free, sbmem_alloc will cause a lot of splits (according to the Buddy algorithm). And then when we free it, all of these split buddies will be combined together again. This is why this is the worst cate.

However, if we allocate one block in the beginning and keep it, its buddy will be available for the upcoming allocations. So, there will be the minimal amount of splits and combines and best case can be obtained by the following:

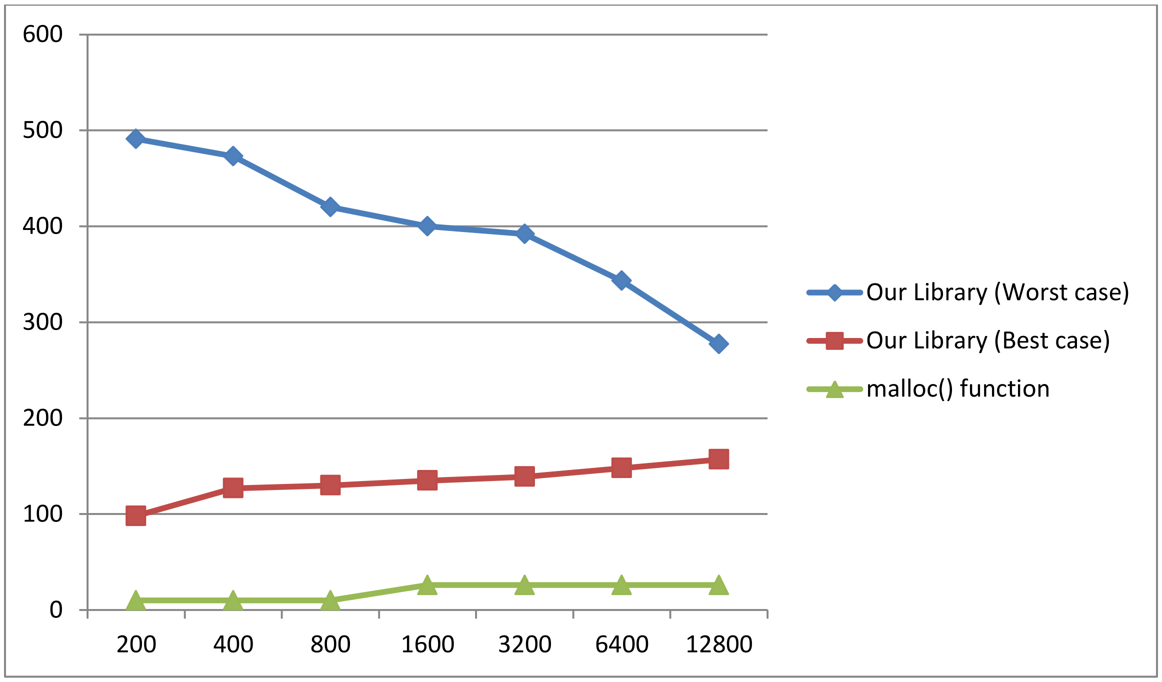
```
char *p, *p2;

p2 = sbmem_alloc (allocSize);
for (int i = 0; i < times - 1; i++)
{
   p = sbmem_alloc (allocSize);
   sbmem_free (p);
}
sbmem_free (p2);
```

Results are as follows where all the exeuction times are in milliseconds:

Allocation Request Size	Execution Time with Our Library (Worst Case)	Execution Time with Our Library (Best Case)	Execution Time with malloc()
200	491	98	10
400	473	127	10

800	420	130	10
1600	400	135	26
3200	392	139	26
6400	343	148	26
12800	277	157	26

And plotted as follows:



The results show that malloc() works the fastest. There were some possible improvements in my code but I don’t think even those improvements would be enough to surpass the performance of malloc().

Note: I haven’t used allocation request size of 2’s powers, instead I used 200, 400 and so on. Because when we request 256 bytes allocation for example, it directly allocates 512 bytes because of the overhead in each blocks. Using 200, 400, ... was fairer for comparison purposes.

Code for Experiment 1.

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

#include "sbmem.h"

int randomUniform(int min, int max)
{
    return rand() % (max - min + 1) + min;
}

int main()
{
    int ret, allocSize;
    char *p;

    srand(time(NULL));

    ret = sbmem_open();
    if (ret == -1)
        exit (1);

    int totalAllocated = 0;
    int totalRealAllocated = 0;

    while (1)
    {
        allocSize = randomUniform(128, 4096);
        p = sbmem_alloc (allocSize);

        //16 is the overhead from the information stored in the beginning of each block.
        int size = allocSize + 16;

        int realAllocSize = 2;
        while (realAllocSize < size)
        {
            realAllocSize *= 2;
        }

        if (p == NULL)
            break;
        else
        {
            if (randomUniform(0, 100) > 0)
            {
                sbmem_free(p);
            }
            else
            {
                totalAllocated += allocSize;
                totalRealAllocated += realAllocSize;
            }
        }
    }
    sbmem_close();

    printf("Total allocated size is %d\nReal allocated size is %d\n", totalAllocated,
totalRealAllocated);
    printf("Internal fragmentation is %.4f\n", (totalRealAllocated - totalAllocated)
/(totalRealAllocated + 0.0));
    printf("External fragmentation is %.4f\n", 1 - (totalAllocated) /(262144 + 0.0));


    return (0);
}
```

Code for Experiment 2.

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

#include "sbmem.h"

void allocateWithLib(int times, int allocSize)
{
    char *p, *p2;

    //p2 = sbmem_alloc (allocSize);
    for (int i = 0; i < times; i++)
    {
        p = sbmem_alloc (allocSize);
        sbmem_free (p);
    }
    //sbmem_free (p2);
}

void allocateWithMalloc(int times, int allocSize)
{
    char *p;

    for (int i = 0; i < times; i++)
    {
        p = malloc(allocSize);
        free(p);
    }
}

int main(int argc, char *argv[])
{
    struct timeval starttime, endtime;
    gettimeofday(&starttime, NULL);

    int ret;
    ret = sbmem_open();
    if (ret == -1)
        exit (1);

    if (argc != 2)
        return 0;
    int allocSize = atoi(argv[1]);

    allocateWithLib(100000,allocSize);

    sbmem_close();

    gettimeofday(&endtime, NULL);
    double elapsedTime1 = (endtime.tv_sec - starttime.tv_sec) +
        (endtime.tv_usec - starttime.tv_usec) / 1000000.0;

    gettimeofday(&starttime, NULL);

    allocateWithMalloc(100000,allocSize);

    gettimeofday(&endtime, NULL);
    double elapsedTime2 = (endtime.tv_sec - starttime.tv_sec) +
        (endtime.tv_usec - starttime.tv_usec) / 1000000.0;

    printf("With lib: %f seconds, with malloc: %f seconds\n", elapsedTime1, elapsedTime2);

    return (0);
}
```