

My malloc: mylloc and mhysa

Johan Montelius

HT2016

1 Introduction

So this is an experiment where we will implement our own malloc. We will not implement the world's fastest allocator, but it will work and we will hopefully better understand what is required by the underlying memory manager. Remember that `malloc()` is a library procedure that is running in *user space*, it is therefore easy to direct a program to use our allocator rather than the one provided by the standard library.

2 The interface

If we look at the manual pages of malloc and free, we start to understand what the requirements are.

The `malloc()` function allocates `size` bytes and returns a pointer to the allocated memory. The memory is not initialized. If `size` is 0, then `malloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

The `free()` function frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()`, `calloc()`, or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is `NULL`, no operation is performed.

Also look further down in the man page, to see what the procedures should return and why `malloc()` can fail.

2.1 this is too easy

Ok, how hard can it be, we only need to have a infinite memory and the problem is solved - let's do this as our first experiment. Create a file called `mylloc.c` and buckle your seat belt.

We will make things very easy in the beginning, simply ask the operating system for more heap space when we call `mylloc()` and ignore anything that is freed. Now this solution would not give you any points in the exam but it's a good start for our experiments.

```
#include <stdlib.h>
#include <unistd.h>
```

```

void *malloc(size_t size) {
    if( size == 0 ){
        return NULL;
    }
    void *memory = sbrk(size);
    if(memory == (void *)-1) {
        return NULL;
    } else {
        return memory;
    }
}

void free(void *memory) {
    return;
}

```

Look up the man-pages for `sbrk()`. The procedure will fail only if the operating system fails to increase the heap segment, if this happens it will return `-1` but `malloc()` should return `NULL`. Also note the two versions of changing the size of the heap segment, `sbrk()` and `brk()`, the first is just a convenient way of calling the other.

When we compile the `mylloc.c` program we need to tell GCC that the program should be compiled to an object file but we should not link the object file and try to turn it into an executable. After all there is no `main()` procedure, so it is not a complete program. We do this using the `-c` flag, if we look in the man page for `gcc` we find this description:

`-c` Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

So our command looks like this:

```
$ gcc -c mylloc.c
```

If everything works you should have a file called `mylloc.o` that is the object file that we need.

2.2 the benchmark

To see how well our solution (remember the proposal is close to a stupid solution) works, we implement a benchmark that will allocate and free a sequence of memory blocks. Our first benchmark will not be the best benchmark, but it will at least show that the system is working, or rather why it is not working that well. We use a call to `sbrk(0)` to get the current top of the heap and then track this for each round that we execute the inner loop.

```

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define ROUNDS 10
#define LOOP 100000

int main() {
    void *init = sbrk(0);
    void *current;

    printf("The initial top of the heap is %p.\n", init);

    for(int j = 0; j < ROUNDS; j++) {
        for(int i = 0; i < LOOP ; i++) {
            size_t size = (rand() % 4000) + sizeof(int);
            int *memory;
            memory = malloc(size);
            if(memory == NULL) {
                fprintf(stderr, "malloc failed\n");
                return(1);
            }
            /* writing to the memory so we know it exists */
            *memory = 123;
            free(memory);
        }
        current = sbrk(0);
        int allocated = (int)((current - init) / 1024);
        printf("%d\n", j);
        printf("The current top of the heap is %p.\n", current);
        printf("    increased by %d Kbyte\n", allocated);
    }
    return 0;
}

```

2.3 our first run

We can now compile our benchmark and provide the object file so that the linking works. Note that we provide the object file `mylloc.o` as an argument to `gcc`. The linker will first use the object files that we provide so the real implementation of `malloc()` in the standard library is *shadowed* by our implementation in `mylloc.o`.

```
$ gcc -o bench mylloc.o bench.c
```

Ok, give the benchmark a try to see if it works. If it does, you should see how the heap is increasing as we *myllocate* more memory. The problem is of course that our implementation of `free()` is simply doing nothing. The blocks that we have freed could have been reused when we asked for the next block but we simply ignore old blocks and just ask the kernel for more space.

For how long can this work, we must run out of memory sometime? Increase the number of `ROUNDS` to 100 and see what happens. My guess is that the program will not end successfully, it might stop with an exit message “malloc failed” but it could also be that it is shot down by the kernel and the final message is simply “killed”. The latter happens when the kernel is starting to run out of virtual memory and is looking around for a suitable process to terminate. This is the called *Out Of Memory management* or simply *the OOM-Killer process in action*.

You might wonder what could cause the kernel to be out of virtual memory, if we have a virtual address space of 47 bits, there should be plenty of virtual memory to choose from. Hmm ..., comment out the line where we write 123 to the allocated memory, then increase the number of rounds to a thousand - what is going on?

2.4 a random sequence of size

Ok, so we know that things work and we know what we will have to improve the reuse of freed blocks, but before we change the mylloc procedures we need to improve the benchmark program.

The first thing that we will fix is the random selection of block size. We can assume that a program will request small blocks far more often than large blocks, so this is our goal. The use of `rand() % 4000` in the code gives us a uniform distribution from zero to 4000. We would much rather have an *exponentially decreasing* distribution between for example *MIN* and *MAX*; how can we achieve this? What if the size was calculated like this, where *r* is a random value,:

$$size = MAX/e^r$$

If e^r is from 1 to MAX/MIN , then the size will be in the range from *MIN* to *MAX*. To generate e^r in this range we simply want *r* to be in the range 0 to $\log(MAX/MIN)$. Let's go, we have to keep track of doubles and int:s but it's quite simple.

```
#include <math.h>

#define MAX 4000
#define MIN 8

int request() {
```

```

/* k is log(MAX/MIN) */
double k = log(((double) MAX )/ MIN);

/* r is [0..k[ */
double r = (((double)( rand() % (int)(k*10000)))) / 10000;

/* size is [0 .. MAX[ */
int size = (int)((double)MAX / exp(r)) ;

return size;
}

```

To see the effect of this distribution you can do a quick experiment. Write a small program, **rand.c**, that takes an integer as an argument and generates a sequence of requested block sizes. Print them to **stdout** so that we can pipe the sequence and sort the output.

```

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    if(argc < 2) {
        printf("usage: rand <loop>\n");
        exit(1);
    }
    int loop = atoi(argv[1]);
    for(int i= 0; i < loop ; i++) {
        int size = request();
        printf("%d\n", size);
    }
}

```

When you compile the program you need to include the *math library* (**-lm**) in order for the linker to find the definitions of **log()** and **exp()**. Note that **rand.c** must be given before the link parameter.

```
$ gcc -o rand rand.c -lm
```

Now we generate a sequence and sort the output:

```
$ ./rand 100 | sort -n
```

Looks ok? Now generate a thousand numbers, sort them and save it to a file **freq.dat**.

```
$ ./rand 1000 | sort -n > freq.dat
```

Now we are ready to use gnuplot to take a look at the generated sequence. Start gnuplot and just do a quick and dirty plot of the `freq.dat`.

```
$ gnuplot
:
:
gnuplot> plot "freq.dat" u 1
```

You can try the following if you better want to see what is happening:

```
gnuplot> set logscale y
:
gnuplot> plot "freq.dat" u 1
```

Ok? Now let's continue the work on the benchmark program.

2.5 a buffer of blocks

The benchmark program does not mimic how a typical program would use memory. We simply allocate a memory block and then free the same block immediately. A typical program would probably allocate some blocks, free some and then allocate some more etc. We need to adapt our benchmark program so it produces something that is closer to regular behavior.

To do this we introduce a buffer of allocated blocks. We will randomly select a position in the buffer and if it is empty we allocate a block. If we find a block at the position we first free this block before allocating a new block. When we start, the buffer is all empty but after a while it will be filled with references to allocated blocks.

The size of the buffer is the maximum number of blocks in memory. Since we randomly add or remove blocks we will on average have half of the entries filled. The size of the buffer is a measurement on how memory hungry your benchmark will be and how quickly data structures are freed; choose any number that you think makes sense but realize that it will change the behaviour of the benchmark.

```
| #define BUFFER 100
```

To initialize the buffer we include this code in the beginning of the main procedure:

```
| void *buffer[BUFFER];
| for(int i =0; i < BUFFER; i++) {
|     buffer[i] = NULL;
| }
```

Now change the section where new blocks are allocated to something that looks like this:

```
int index = rand() % BUFFER;
if(buffer[index] != NULL) {
    free(buffer[index]);
}
size_t size = (size_t)request();
int *memory;
memory = malloc(size);

if(memory == NULL) {
    fprintf(stderr, "memory allocation failed\n");
    return(1);
}
buffer[index] = memory;
/* writing to the memory so we know it exists */
*memory = 123;
```

A word of warning; we're happily writing to the allocated memory in the hope that the integer will actually fit. We're thus relying on the fact that `request()` will always return something that is greater than `sizeof(int)`. This is true in our case since we have `MIN` set to 8 but if you set `MIN` to 2 things might break.

So now we should have a nice benchmark program. Try it to see that it works and then do the following: re-compile `bench.c` but now omit to provide `mylloc.o` on the command line. The benchmark program will then use the definition of `malloc()` found in the standard library.

```
$ gcc -o bench bench.c -lm
```

Run the benchmark again and see if there is any difference - we have some work to do, right?

3 Let's keep it simple

We should not complicate things more than necessary, so let's keep it simple. We can save all freed blocks in a linked list and when we're asked for a new block we simply search through the list to see if we can find a block that is large enough.

In order to do this we of course need to keep track of how big the blocks are: this is not something that comes automatically. When the `free()` procedure is called it is given a reference to a block but there is no information on how big the block is. To solve this dilemma we have to cheat a bit.

3.1 You do not owe me your freedom.

Make a copy of `mylloc.c` and call it `mhysa.c`. We will now adapt the procedures to work with a linked list of free blocks. Since we do not want to expose the internals of our implementation we will do a trick. We will allocate more memory than requested and write some hidden information in the beginning of the block. When we hand the block to the user process, we give it a pointer to the first free location.

We create a new data structure `chunk` and initialize a free list pointer. The data structure is called `malloc_chunk` in Linux and holds more information than what we have, but this is fine for now.

```
struct chunk {
    int size;
    struct chunk *next;
}

struct chunk* flist = NULL;
```

We now have to rewrite `free()` and `malloc` to make use of the free list. To free a block is quite simple, we assume that the Reference that we get is a reference to something that we allocated and therefore know that there will be a hidden `chunk` structure just before the given reference.

```
void free(void *memory) {
    if(memory != NULL) {
        /* we're jumping back one chunk position */
        struct chunk *cnk = (struct chunk*)((struct chunk*)memory - 1);
        cnk->next = flist;
        flist = cnk;
    }
    return;
}
```

It is a simple task to let the next pointer of the freed block point to whatever `flist` is pointing to and then update `flist`. You can test to see that it works, but nothing has changed if we do not make use of the blocks in the lists.

3.2 find a free block

The `malloc()` procedure is slightly more code but the task is quite simple. If we're asked for a new block of a given size we will first search the free list for a suitable block. If one is found we can reuse the block and will then of course un-link it from the free list. If no suitable block is found we have to do as we did before and ask the kernel for some more space.


```

void *malloc(size_t size) {
    if( size == 0 ){
        return NULL;
    }
    struct chunk *next = flist;
    struct chunk *prev = NULL;

    while(next != NULL) {
        if (next->size >= size) {
            if(prev != NULL) {
                prev->next = next->next;
            } else {
                flist = next->next;
            }
            return (void*)(next + 1);
        } else {
            prev = next;
            next = next->next;
        }
    }
    :

```

The only tricky thing is to make sure that we request more space from the kernel than the user process asks for. We need some space to write the hidden **chunk** structure. We also initialize this structure with the right size value, since this is something we need to know if we later want to reuse the block.

```

        :
        /* use sbrk to allocate new memory */
        void *memory = sbrk(size + sizeof(struct chunk));
        if(memory == (void *)-1) {
            return NULL;
        } else {
            struct chunk *cnk = (struct chunk*)memory;
            cnk->size = size;
            return (void*)(cnk + 1);
        }
    }
}

```

If everything works you should be able to compile the **mhysa.c** module into an object file and then link this with the benchmark - give it a try.

```

$ gcc -c mhysa.c
$ gcc -o bench mhysa.o bench.c -lm

```

```
$ ./bench
```

How is that? It's dirt simple and of course not the most efficient memory allocator but it looks like it's working right?

4 How to improve

We have a system that is working but there are of course a lot of things that could be improved. One problem we clearly have is that we're still consuming far more memory than what we would need. The standard library implementation obviously can do away with far less memory. The other problem is that we're probably asking the kernel for memory more often than we would have to.

4.1 system calls

There is a command `strace` that could be fun to use. This command will execute a program but trap all system calls and print them to standard output. Try the following:

```
$ strace ./bench
```

Too much information? Try this (we're redirecting stderr to stdout to be able to pipe it to grep):

```
$ strace ./bench 2>&1 | grep brk | wc -l
```

Try all versions of the benchmark program, one linked with the standard library, one with `mylloc.o` and one with `mhysa.o`. How more often do we request memory from the kernel? Are there ways to avoid this?

4.2 first, best, worst fit

To reduce the amount of memory we use, we could try to be more clever when selecting a free block. If we're looking for a block of 32 bytes it is quite wasteful to use a block of 2000 bytes. The least one could do is to find the smallest block that would work. Finding a good block does however take time, so this is a trade-off.

Even if we find a suitable block we will not always find a perfect fit. There will be space in a block that is unused, this is called *internal fragmentation*. We could probably live with it but if we want to reduce this waste we come to the solution of splitting a block in two if it's too big. Assume we're looking for a block of 32 bytes and the best we can find is a chunk for 78 bytes. We could then divide the chunk into two chunks, one for 32 and the

other for 34 (we loose some bytes since we need a new chunk header). This would be much better, problem solved.

The danger of dividing chunks into too small pieces is that we will have problems finding larger chunks. There might be plenty of free space but it is all divided into small chunks, this is called *external fragmentation*. The solution to this problem is to somehow merge free chunks that are adjacent to each other in memory. If we realize that the two chunks of 32 and 34 bytes that we created are both in the free-list we can merge them and again have a chunk of 78 bytes.

4.3 keep it simple

An alternative to splitting and merging chunks is to allow a certain amount of internal fragmentation to make things simpler. Assume we only allocate chunks of a total size of even powers of 2, for example: 32, 64, 128 etc. A chunk of 32 will consist of a 12 byte header and 20 bytes available for the user (we can improve this, but it's fine for now). Then we keep multiple free lists for the different sizes.

We will in average use only three quarters of the allocated memory, but management of the blocks becomes very simple. The nice thing about even powers of two is that we can reduce the calls to `sbrk`, we always request a memory of multiples of 4096 bytes (a page). The memory is then divided into chunks of the required size and added to the corresponding free list.

Calls to `malloc()` and `free()` are extremely fast and fragmentation is controlled.

4.4 let's complicate things

Just when you thought that things were actually quite simple we have to break it to you - what happens when we have multiple threads accessing the memory allocator concurrently. It's not hard to see that any solution that we have discussed will break. It's of course possible to protect the structures with one big lock, but that would decrease performance.

The challenge with multiple threads is to try to keep things separated while still not using more resources than necessary. If a thread frees a block, then it would be preferred if this block was later returned to the same thread. The thread might still have the block in its cache, and it would therefore have an advantage of being able to use it again.

5 Summary

Implementing a strategy that works well over a range of hardware systems is a problem, and far more complicated than what we have done in this

exercise. The important lesson in this exercise is that it is doable, and I'm sure you realize that you would be able to do so given more time.

I also hope that you better understand the role of `malloc()` and `free()` and that they are working in user space. The kernel should be disturbed as little as possible, and we achieve this by allocating larger blocks of memory than we have to.