Course: Operating Systems - Semester 1 of 5781

Assignment 5

Directions

A. Published: 27 January 2021

B. **Due Date: 21 February 2021 at 11:55pm**

C. Groups of up to 2 students may submit this assignment.

D. Code for the assignment must be submitted via GitHub using the private repository opened for you in the OSCourse organization.

- Each student must make at **least one** submit to the repository in order to receive credit.
- The last commit must include the comment "Submitted for grading"

E. There are 100 points total on this assignment.

F. What to turn in the final GitHub repository:

(a) `Makefile` for the whole project (requirements below in Section 3.4). If you choose to reuse the provided Makefile, you must submit it too.

(b) Completed C files and any other associated files (`*.c`) or header files (`*.h`)

(c) Any additional files required for compilation.

(d) A README file (README.pdf, README.md, or REAMDE.txt) that contains the following information:

- The names and ID numbers of each student in the submission team
- The number of hours worked on the assignment
- Any special instructions for compiling or running the program

General Requirements

1. All of the code below must be written in C and compilable and executable in a standard Linux Mint (19+) or Linux Ubuntu (18+) environment.

2. All code must have comments - each function must have an introductory comment detailing its purpose, input parameters, and return value.

Malloc

1 Introduction

Your task in this assignment[1] is to implement your own memory allocator from scratch. This will expose you to POSIX interfaces, force you to reason about memory, and pose interesting algorithmic challenges.

The man pages for `malloc` and `sbrk` are excellent resources for this assignment.

Note: You **must** use `sbrk` to allocate the heap region. You are **not** allowed to call the standard `malloc/free/realloc` functions. Doing so will lead to a grade of 0 on this assignment.

The starter repository on GitHub contains the following files:

- `mm_alloc.c` - This file contains a simple skeleton that you must fill in
- `mm_alloc.h` - This file contains the interfaces for three functions: `mm_malloc`, `mm_free`, and `mm_realloc`. You must implement these functions in the `mm_alloc.c` file. Do not change the names or signatures of the functions.
- `mm_text.c` - This file contains code to load your implementation and test it. You will make changes here, filling in tests.

You must submit all of the above files with your submission, in addition to a `Makefile` and any other files you choose to add and are necessary to compile or run your tool.

2 Background: Getting Memory from the OS

2.1 Process Memory

Each process receives its own virtual address space. Parts of this address space are mapped to physical memory through address translation. To build a memory allocator, we need to understand how the heap in particular is structured.

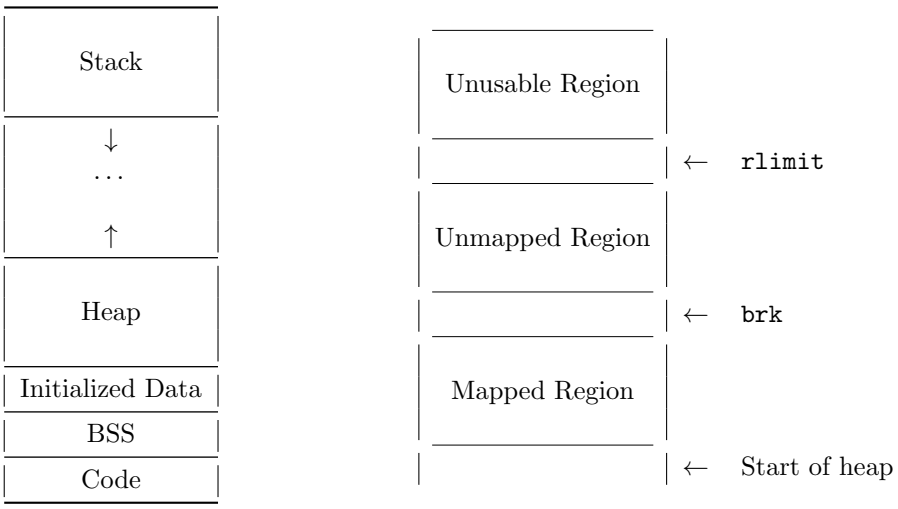

Figure 1: The left figure shows the general layout of the memory space of a process. The right figure shows a breakdown of the heap, including the positions of the two values - `rlimit` and `brk`.

As Figure 1 shows, the heap is a continuous (in terms of virtual addresses) space of memory that grows upward in memory with three bounds:

[1] This homework is based on the HW4 Malloc assignment from UC Berkeley and Professor John Kubiatowicz. The original version can be accessed at: `https://inst.eecs.berkeley.edu/~cs162/fa20/static/homeworks/homework4.pdf`

1. The start (bottom) of the heap.
2. The top of the heap, known as the break. The break can be changed using the `brk` and `sbrk` system calls. The break marks the end of the mapped memory space. Above the break lies virtual addresses that have not been mapped to physical addresses by the OS.
3. The hard limit of the heap, which the break cannot surpass (managed through `sys/resource.h`'s functions `getrlimit(2)` and `setrlimit(2)`).

In this assignment, you'll be allocating blocks of memory in the mapped region and moving the break appropriately whenever you need to expand the mapped region.

2.2 `sbrk`

Initially the mapped region of the heap will have a size of 0. To expand the mapped region, we must manipulate the position of the break. The recommended syscall for doing this is `sbrk`. It has the following signature:

```
void *sbrk(int increment);
```

`sbrk` increments the position of the break by `increment` bytes and returns the address of the previous break (*i.e.* the beginning of newly mapped memory). To get the current position of the break, pass in an increment value of 0. For more information, read the `man` page.

2.3 Heap Data Structure

A simple memory allocator for the heap can be implemented using a linked list data structure as shown in Figure 2. The elements of the linked list will be the allocated blocks of memory on the heap. To structure our data, each allocated block of memory will be preceded by a header containing metadata. For each block, we include the following metadata:

- `prev, next`: pointers to metadata describing the adjacent blocks
- `free`: a boolean describing whether or not this block is free
- `size`: the allocated size of the block of memory (bytes)

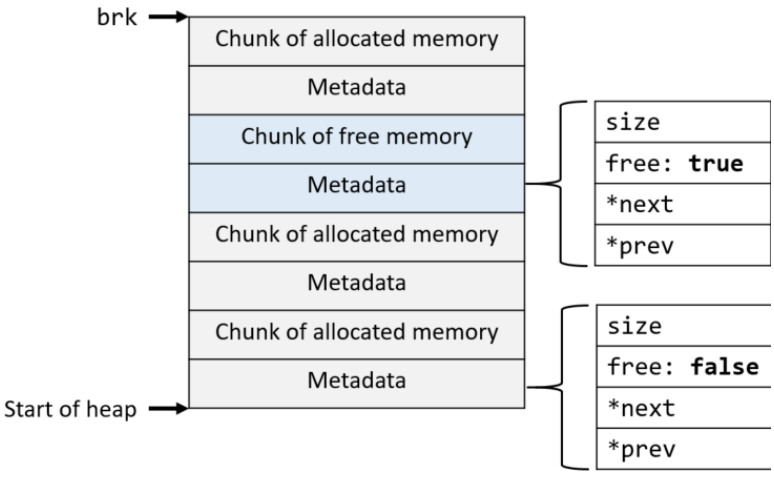

Figure 2: Linked list layout for memory allocation.

The pointer to the memory block could be a `void*` or a zero-length array. You can read more about zero-length arrays at `https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html`.

3 What to do: Memory Allocator

There are many ways to structure a memory allocator. As a start, you will implement the memory allocator using a linked list of memory blocks as described in the previous section. Now, we'll describe how allocation, deallocation, and reallocation should work in the scheme. To make your implementation succeed, you will need to modify `mm_alloc.c`.

3.1 Allocation function

Function signature:

```
void *mm_malloc(size_t size);
```

The user will pass in the requested allocation `size` in bytes. The function must return a pointer to the newly allocated region. Make sure the returned pointer points to the beginning of the allocated space, not your metadata header. Your algorithm will need to traverse the linked list of memory areas and find a region that can supply enough room for the requested size.

3.1.1 First fit

One simple algorithm for finding available memory is called *first fit*. When your memory allocator is called to allocate some memory, it iterates through its blocks until it finds a sufficiently large free block of memory. This is the recommended algorithm to use for this assignment.

Implementation Details

- Iterate over the linked list of memory areas.
- If the first sufficient free block of memory you find is so large that it can accommodate both the newly allocated block and another block after it, split the large block in two. The first part should hold the newly allocated block. The other one must become a residual free block.
- If the first sufficient free block of memory you find is only a bit larger than what you need, but not large enough for a new block (*i.e.* it's not big enough to hold the metadata of a new block), use it anyway, but be aware that you will have some unused space at the end of the newly allocated block.
- If no sufficiently large free block is found, use `sbrk` to create more space on the heap.
- Return NULL if you cannot allocate the new requested size.
- Return NULL if the requested size is 0.
- For grading purposes, please zero-fill your allocated memory before returning a pointer to it.

3.1.2 Other fit algorithms

You may attempt to implement other fit algorithms (best fit, worst fit, buddy), but you must get approval from the instructor first.

3.2 Deallocation function

Function signature:

```
void mm_free(void *ptr);
```

When a user is done using their memory, they'll call upon your memory allocator to free their memory, passing in the pointer `ptr` that they received from `mm_malloc`.

Note: Deallocating doesn't mean you have to release the memory back to the OS; you just must be able to allocate that block for future use now. You can mark it as free and leave it in the linked list.

Implementation Details

- As a side-effect of splitting blocks in your allocation procedure, you might run into issues of external fragmentation. That occurs when blocks become too small for large allocation requests, even though you have a sufficiently large section of free memory. To solve this, you must coalesce (combine) consecutive free blocks when freeing a block that is adjacent to other free block(s).
- Your deallocation function should do nothing if passed a NULL pointer.
- Your function should do nothing if passed a pointer that doesn't point to an area you allocated.

3.3 Reallocation function

Function signature:

```
void* mm_realloc(void* ptr, size_t size);
```

Reallocation should resize the allocated block at `ptr` to `size` bytes. Return a pointer to the newly allocated area. A suggested implementation is to first free the block referenced by `ptr`, then `mm_alloc` a block of the specified size, zero-fill the block, and finally `memcpy` the old data to the new block.

Make sure you handle the following edge cases:

- Return NULL if you cannot allocate the new requested size. In this case, do not modify the original block.
- `mm_realloc(ptr, 0)` is equivalent to calling `mm_free(ptr)`. In this case, free the block and return NULL.
- `mm_realloc(NULL, n)` is equivalent to calling `mm_malloc(n)`. Return the pointer to the newly allocated area.
- `mm_realloc(NULL, 0)` is equivalent to calling `mm_malloc(0)`, which should just return NULL.
- Make sure to handle the case where `size` is less than the original size or equal to the original size. In both cases, you don't need to move the block, you can just modify the block metadata in place.

3.4 Makefile

As you did in previous assignments, you must submit a `Makefile` with your code. In order to build and run the tests, you should be able to run the following from the command line:

```
$ make
$ ./mm_test
```

The `Makefile` for the project must include the following rules:

- An `all` rule that compiles the test code into an executable called `mm_test`. The rule should be the *default* rule for the make tool.
- For each `*.c` file: An intermediate rule to compile it to an `*.o` file. Do this *even* if you write the whole program in one `.c` file (which is frowned upon, but not illegal).
- A `clean` rule that erases all `*.o` files and the executable.

I am providing a prepared Makefile that should be sufficient for the above. If it suffices for your needs, you may use it. If not, you may modify it to meet your own needs. In any case, you must submit it with your repository.

4 Test File

Modify the `mm_test.c` file to test your memory allocator and deallocator. You must ensure the test file has the following elements:

1. A `main` function that runs all test functions.
2. A test function called "basic" that performs the following steps:
   (a) Allocates 100 bytes
   (b) Edits a few bytes
   (c) Frees the allocation
   (d) Allocates 0 bytes
   (e) Frees a non-existent pointer
   (f) Reallocates from 0
   (g) Reallocates to 0
3. A second test function that performs 10 allocations, 5 frees, 10 reallocations, 5 frees, and then 10 more allocations. After each allocation, perform reads and writes to the newly allocated space. After reallocation, check that contents are preserved.
4. A third test function that performs 20 allocations, 5 frees, 20 allocations, 20 reallocations, 15 frees, 10 allocations, and 10 reallocations. After each allocation, perform reads and writes to the newly allocated space. After reallocation, check that contents are preserved.
5. Most test functions that you write on your own.

Print out intermediate success messages (*e.g.* allocated 100 bytes successfully, freed 2000 bytes successfully, reallocated 1000 bytes to 2000 bytes successfully) and intermediate failure messages (*e.g.* failed to allocate 800 bytes, failed to free 4000 bytes, failed to reallocate 2000 bytes to 200 bytes).

If you follow the interface defined above, it should be possible to test your code with tests from other students. That will be part of the testing regimen.

Bonus A 10-point bonus will be given for the team with the best and most complete testing program. If there is a tie between multiple groups for the best test regimen, the points will be divided among the groups.

5 Grading and Penalties

The assignment will be graded as follows:

- Allocation: 30%
- Free: 25%
- Reallocation: 25%
- Testing program: 20%

5.1 Penalties

The following is a partial list of penalties that will be assessed on submissions.

- Uses malloc/realloc/free internally **0 points**
- Readme missing. **(-7) points**
- Readme missing important element. **(-3) points**
- Test regimen misses one required element. **(-5) points each**