HW 3: Malloc

March 10, 2014

1 Setup

Note: This homework is longer than previous assignments so please start early!
Your task in this homework is to implement your own memory allocator in `mm_alloc.c` from scratch. This exposes you to POSIX interfaces, forces you to reason about memory, and poses interesting algorithmic challenges. If your system needs to deal with high memory pressure (which is often th case), improving your memory allocator can have a dramatic impact on performance. That's why custom allocators are ubiquitous (e.g in CPython, C++ Boost, and Linux), so it's worth understanding them well.
This assignment will be due **11:59 pm PST 03/23/2015**

```
cd cs162-hw
git pull staff master
cd hw3
```

You will find a simple skeleton in `mm_alloc.c`. `mm_alloc` defines three functions `mm_malloc`, `mm_free`, `mm_realloc`, which all call their POSIX counter parts right now. You will need to replace this with your own implementation.

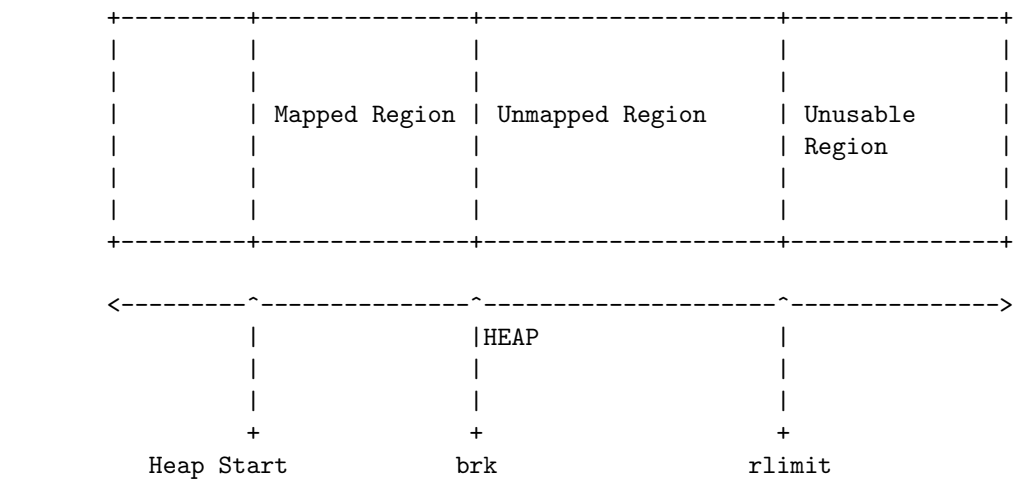
Your version of `malloc()` must use `sbrk()` to request memory from the kernel. The main job of `malloc()` is to carve up this memory into reasonably-sized pieces as needed. Your `free()` routine is responsible for deallocating memory and for releasing it back to the kernel as memory pressure decreases. Also note that any heap-allocated data structures you use in your allocator must use memory *allocated by your allocator*.

The man pages for `malloc`, `mmap`, and `sbrk` are an excellent resource.

2 Getting Memory from the OS

2.1 Process Memory

Each process has its own virtual address space dynamically translated into physical memory address space by the MMU (and the kernel.) This space is divided in several part, all that we have to know is that we found at least some space for the code, a stack where local and volatile data are stored, some space for constant and global variables and an unorganized space for programs data called the heap. The heap is a continuous (in terms of virtual addresses) space of memory with three bounds: a starting point, a maximum limit (managed through `sys/resource.hs` functions `getrlimit(2)` and `setrlimit(2)`) and an end point called the break. The break marks the end of the mapped memory space, that is, the part of the virtual address space that has correspondence into real memory. The below figure marks the organization.



2.2 brk(2) and sbrk(2)

We can find the description of these syscalls in their manual pages:

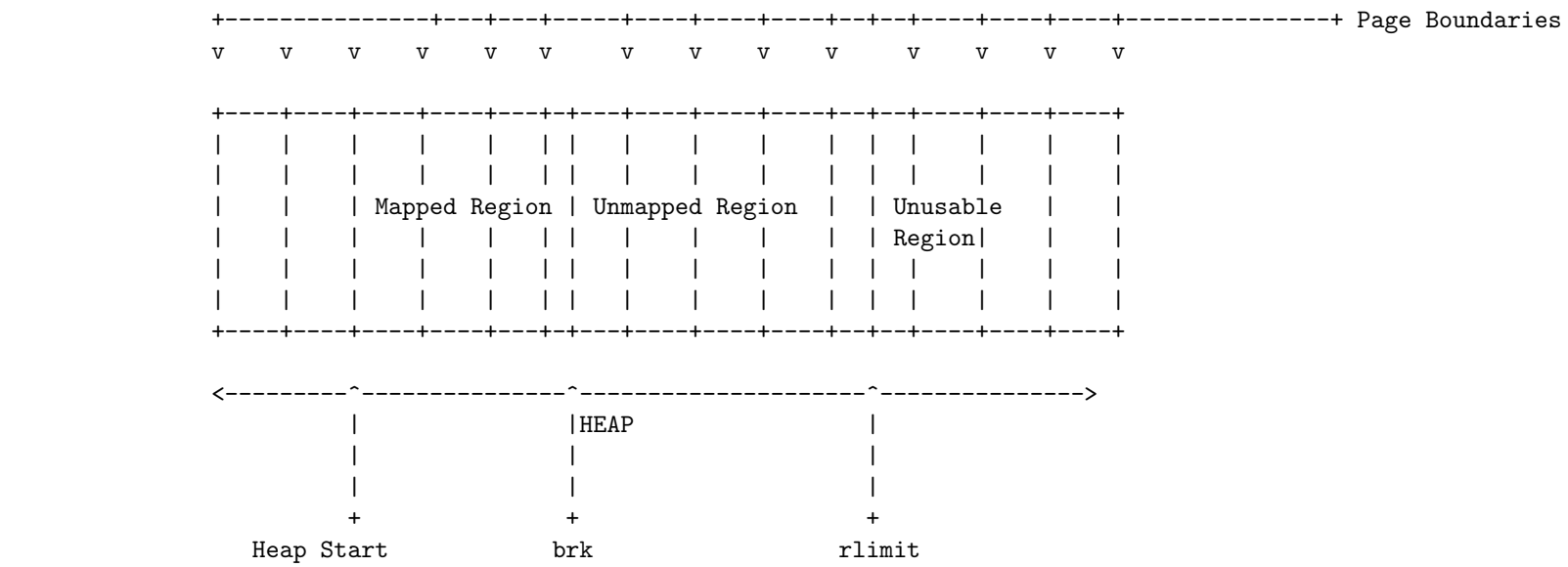
```
int brk(void *addr);
void *sbrk(intptr_t increment);
```

`brk(2)S places the break at the given address \verb`addr— and return 0 if successful, -1 otherwise. The global `errno` symbol indicate the nature of the error. `sbrk(2)` move the break by the given increment (in bytes.) On success it returns the previous address (on Linux systems). On failure, it returns (void *)-1 and set `errno`. On Linux `sbrk` accepts negative values (in order to free some mapped memory.)

We will use `sbrk` as our main tool to implement `malloc`. All we have to do is to acquire more space (if needed) to fulfill the query.

2.3 Unmapped Region and No Man's Land

We saw earlier that the break mark the end of the mapped virtual address space: accessing addresses above the break should trigger a bus error. The remaining space between the break and the maximum limit of the heap is not associated to physical memory by the virtual memory manager of the system (the MMU and the dedicated part of the kernel.) You know that memory is mapped by pages: physical memory and virtual memory is organize in pages (frames for the physical memory) of fixed size (most of the time.) The page size is by far bigger than one byte (on most actual system a page size is 4096 bytes.) Thus, the break may not be on pages boundary.



The above figure presents the previous memory organization with page boundaries (The new lines represent page boundries). We can see the break may not correspond to a page boundary. What is the status of the memory between the break and the next page boundary ? In fact, this space is available ! You can access (for reading or writing) bytes in this space. The issue is that you dont have any clue on the position of the next boundary, you can find it but it is system dependant and badly advise. This no-mans land is often a root of bugs: some wrong manipulations of pointer outside of the heap can success most of the time with small tests and fail only when manipulating larger amount of data.

2.4 mmap(2)

Even you won't use it in this homework you should pay attention to the `mmap(2)` syscall. It has an anonymous mode (`mmap(2)` is usualy used to directly map files in memory) which can used to implement `malloc` (completely or for some specific cases.) mmap in anonymous mode can allocate a specific amount of memory (by page size granularity) and `munmap` can free it. It is often simpler and more efficient than classical `sbrk` based `malloc`. Many `malloc` implementation use `mmap` for large allocation (more than one page.) The OpenBSDs `malloc` uses only `mmap` with some quirks in order to improve security (prefering pages borders for allocation with holes between pages.)

3 Managing Memory

3.1 Organizing the Heap

If we consider the problem outside of the programming context, we can infer what kind of information we need to solve our issues. Lets take analogy: you own a field and partition it to rent portion of it. Clients ask for different length (you divide your field using only one dimension) which they expect to be continuous. When they have finished they give you back their proportion, so you can rent it again.

On one side of the field you have a road with a programmable car: you enter the distance between the begin of the field and the destination (the beginning of your portion.) So we need to know where each portion begin (this the pointer returned by `malloc`) and when we are at a beginning of a portion we need the address of the next one.

A solution is to put a sign at the beginning of each where we indicate the address of the next one (and the size of the current one to avoid unnecessary computing.) We also add a mark on free portion (we put that mark when the client give it back.) Now, when a client want a portion of a certain size we take the car and travel from sign to sign. When we find a portion marked as 5 free and wide enough we give it to the client and remove the mark from the sign. If we reach the last portion (its sign have no next portion address) we simply go to the end of the portion and add a new sign.

Now we can transpose this idea to the memory: we need extra-information at beginning of each chunks indicating at least the size of the chunk, the address of the next one and whether its free or not.

3.2 Finding a fit

Finding a free sufficiently wide chunk is quite simple: We begin at the base address of the heap, test the current chunk, if it fits our need and its free we mark it as not free and just return its address, otherwise we continue to the next chunk until we find a fitting one or the end of the head. The only trick is to keep the last visited chunk, so the `malloc` function can easily extends the end of the heap if we found no fitting chunk.

You may have notice that we use the first available block regardless of its size (provide that its wide enough.) If we do that we will loose a lot of space (think of it: you ask for 2 bytes and find a block of 256 bytes.) A first solution is to split blocks: when a chunk is wide enough to held the asked size plus a new chunk (at least BLOCK SIZE + 4), we insert a new chunk in the list.

3.3 Data Structures

Consider using a simple data structure to manage memory (these go from simple to more complicated and efficient):

- A free list, a linked list of free chunks of memory
- A list of memory sizes, each of which contain a linked list of free chunks of memory of that size. (This is essentially a list of memory buckets)
- a free interval tree. These let you represent every memory allocation as an extent `(start, length)`. The leaves of the tree correspond to regions of unused memory. If N bytes are requested, it should be possible to scan the interval tree for $(> N)$-sized pieces of memory in $O(\log n)$ time, so long as it's properly balanced.

All of the above and more are acceptable data structures to manager your memory.

4 Deallocation

Calls to `free(3)` give memory back to the process to hand out to other allocations. It is important to understand that this is not the same as handing memory back to the operating system!

You must figure out which chunk corresponds to the given pointer, and mark the appropriate chunk as "free", so future allocations can use that chunk.

4.1 Fragmentation

Deallocating memory can cause fragmentation, which reduces the amount of available contiguous memory and increases lookup times. Your allocator **must** serve a request for N bytes of memory without a call to sbrk if at least N bytes of continuous memory exist in your sbrk-ed slab! In other words your allocator must be intelligent enough to combine two contiguous free chunks to achieve its allocation needs, and avoid calls to sbrk whenever necessary. A simple solution is the following: when we free a chunk, and its neighbors are also free, we can fusion them in one bigger chunk.

5 Realloc

Realloc is pretty straight forward. It returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If `realloc()` fails the original block is left untouched; it is not freed or moved.

You can get away with a simple implementation that calls `malloc` and then `memcpy`.

6 Memory Layout

We will give you complete creative freedom in designing your allocator (as long as it works!). But to make autograding simpler your allocated memory must have the following layout. Your allocated memory should also be 0 filled (we know this isn't the most efficient decision, but it makes our life easier)

Let `char*` $p =$ `malloc`$(size)$, then $p - 4 \equiv 0 \bmod 4$ in the required layout;

```
-----------------------------------------
| uint32_t size | <size> zero-filled bytes |
-----------------------------------------
(p - 4)         p
```

7 ε Bonus

ε Bonus doesn't have any points.
There many things you can do to improve your allocator.

- One of the most interesting/challenging of which is to make it theadsafe! This doesn't mean putting a lock around all calls to malloc, but actually protecting the datastructures such that multiple threads can make allocations at the same time. A good way to do this (not the only way) is to lock certain sizes of allocations, so two threads asking for `4kb` would block, but two thread askings for `4kb` and `32kb` would not block.
- You can defer allocations larger than a page size to `mmap`

- You can improve your allocation algorithm, first fit is one of the simplest, a good one to try out is the buddy allocator
- Implement `realloc` properly to extend the current allocation chunk if possible.

8 Autograder & Submission

Push your code to the autograder branch `ag/hw3` on github to test your code

```
git add -u .
git commit
git push personal master
git checkout -b ag/hw3
git push personal ag/hw3
```

Push the final code release to the branch `release/hw3` on github to submit

```
git add -u
git commit
git push personal master
git checkout -b release/hw3
git push personal release/hw3
```