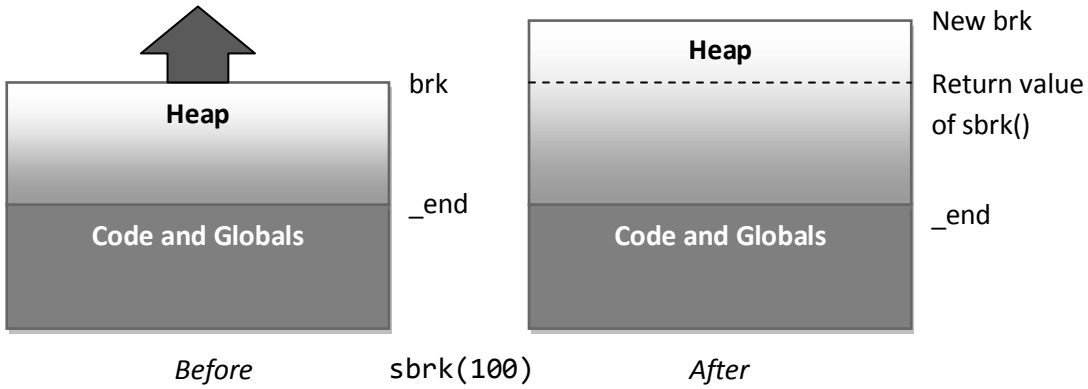
Project 4: A Custom malloc()

Description

In our discussions of dynamic memory management we discussed the operation of the standard C library call, malloc(). Malloc designates a region of a process's address space from the symbol _end (where the code and global data ends) to brk as the heap.

As part of dynamic memory management, we also discussed various algorithms for the management of the empty spaces that may be created after a malloc()-managed heap has had some of it's allocations freed. In this assignment, you are asked to create your own version of malloc, one that uses the worst-fit algorithm.

Details

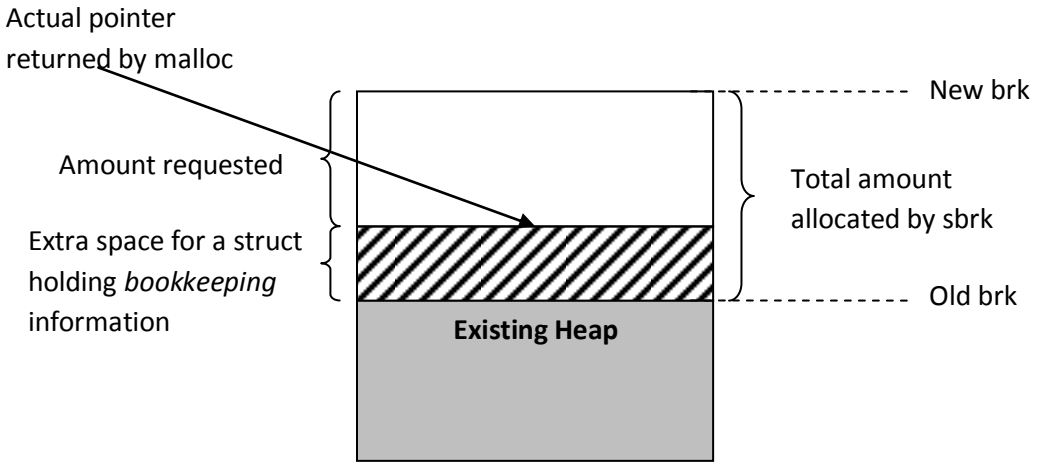


We are programmatically able to grow or shrink the size of the heap by setting new values of brk. The function sbrk() handles scaling the brk value by its parameter:

```
void *sbrk(intptr_t increment);

DESCRIPTION
brk sets the end of the data segment to the value specified by  end_data_segment,
when that value is reasonable, the system does have  enough memory and the process
does not exceed its max data size (see setrlimit(2)).

sbrk increments the program's data space by increment bytes. sbrk isn't a system
call, it is just a C library wrapper. Calling sbrk with an increment of 0 can be used
to find the current location of the program break.

RETURN VALUE
On success, brk returns zero, and sbrk returns a pointer to the start of the new
area. On error, -1 is returned, and errno is set to ENOMEM.
```

A simple place to start then is to create a malloc which, with each request, simply increments brk by the amount requested and returns the old value of brk as the pointer. However, when you want to write a free() function, the parameter to free() is just a pointer to the start of the region, so you have no way to determine how much space to deallocate. In order to know what space is free or used, and how big each region is, we must use one of the techniques from class: Bitmaps or Linked lists.

From our discussion in class, linked lists seem like the better choice, but now we need some place to store this dynamic list of free and occupied memory regions inside of the heap. If we just allocated some fixed-size region, that space may not be adequate for how many nodes in the list we'd need to create. A better idea is illustrated in the figure below:



We can add some additional space to each update of brk in order to accommodate a structure that is a node in our linked list, and this structure can contain useful things like:

- The size of this chunk of memory
- Whether it is free or empty
- A pointer to the next node
- A pointer to the previous node

We then return back a pointer that is in the middle of the chunk we allocated, and thus the program calling malloc() will never notice the additional structure. However, when we get a pointer back to free, we can simply look at the memory before it for the structure that we wrote there with the information we need.

Requirements

You are to create two functions for this project.

1. A malloc() replacement called `void *my_worstfit_malloc(int size)` that allocates memory using the worst- fit algorithm. Again, if no empty space is big enough, allocate more via sbrk().
2. A free() called `void my_free(void *ptr)` that deallocates a pointer that was originally allocated by the malloc you wrote above.

Your free function should coalesce adjacent free blocks as we described in class. If the block that touches brk is free, you should use sbrk() with a negative offset to reduce the size of the heap.

As you are developing, you will want to create a driver program that tests your calls to your mallocs and frees. A week before the due date, I will provide a sample driver program that must work. During grading, we will use a second driver program in addition to the first one in order to test that your code works.

Environment

For this project we will again be working on `thot.cs.pitt.edu`

This machine is a 64-bit machine, and to avoid pointer cast warnings, build using the -m32 option for gcc. This will build a 32-bit program instead of a 64-bit one.

Hints/Notes

- When you manually change brk with sbrk, you may not call malloc or mmap, as they may have unintended consequences. All allocations will have to be done with your new custom malloc().
- In C, the sentinel value for the end of a linked list is having the next pointer set to NULL.
- Make sure you don't lose the beginning or end of your linked list. You may want 2 global variables to keep track of either end
- sbrk(0) will tell you the current value of brk which can help in debugging
- gdb is your friend, no matter what you think after project 2

What to turn in

- A header file named mymalloc.h with the implementations of your two functions
- The test program you used during your initial testing
- Any documentation you provide to help us grade your project