Project 3: A Custom `malloc()` and `free()`

CS/COE 0449 — Introduction to System Software

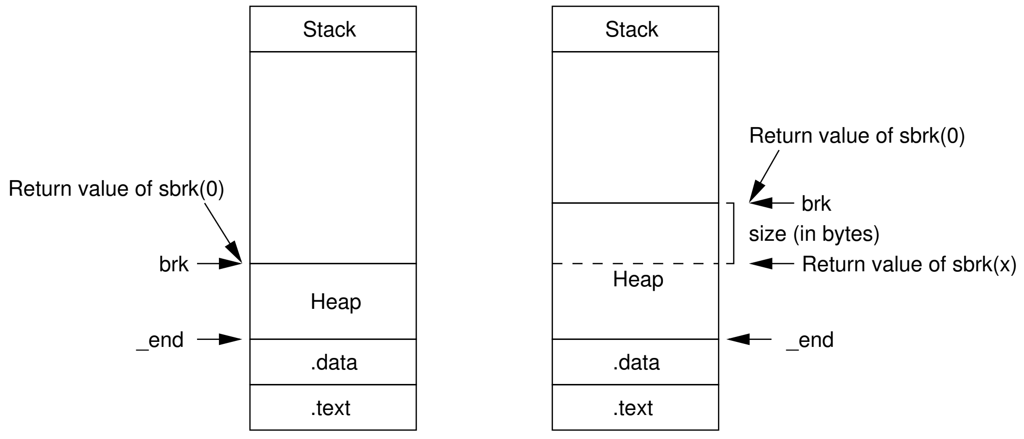
See CourseWeb for Due Date

Description

In our discussions of dynamic memory management we discussed the operation of the standard C library call, `malloc()`. Function `malloc()` designates a region of a processs address space from the symbol `_end` (where the code and global data ends) to `brk` as the heap (see Figure below on left).

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 its allocations freed. In this project, you are asked to create your own version of `malloc()`, one that uses the **first-fit** algorithm.

Details



After call sbrk(x);

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. The following is a description of the funciton `sbrk()`:

```
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)`).

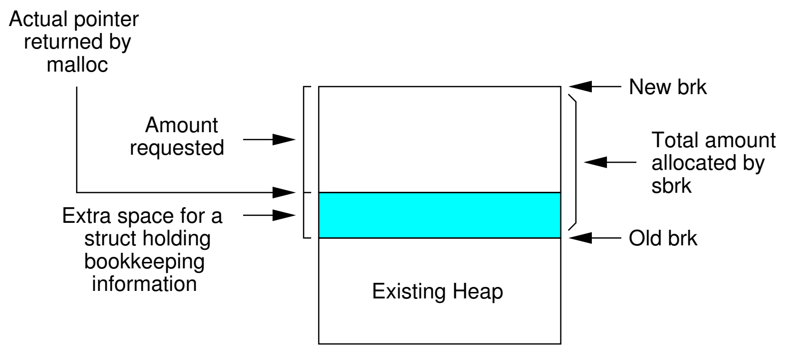
`sbrk` increments the programs data space by increment bytes. `sbrk` isnt a system call, it is just a C library wrapper. Calling `sbrk` with an increment of 0 can be used to find the current location of the program break.

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`.

In other words, a call to `sbrk()` with a positive value will increase the size of the heap by that value (in bytes). As a result, the location of the symbol `brk` will be increased by the same value. A call to `sbrk()` with a negative value results in reverse effect. Note that by calling `sbrk(0)`, it returns the current location of the symbol `brk`.

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 memory management techniques discussed in 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:


Actual pointer returned by malloc
Amount requested
Extra space for a struct holding bookkeeping information
New brk
Total amount allocated by sbrk
Old brk
Existing Heap

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 (requested size + the size of this structure node)

- Whether it is allocated (not free) or unallocated (free)
- A pointer to the next node
- 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

This project will be developed using multi-file environment which will consist of the following files:

1. `mymalloc.h`: The header file of your implementation of `malloc()` and `free()`. This file will consist of function signatures and node structure. Ideally, your `mymalloc.h` should look like the following:

   ```
   #include <unistd.h>

   void *my_firstfit_malloc(int size);
   void my_free(void *ptr);

   struct node
   {
      :
   };
   ```

2. `mymalloc.c`: The implementation file that contains source code of your `malloc()` and `free()`. Note that this file will not have the `main()` function. This file will consist of two functions as follows:

   (a) A `malloc()` replacement called `void *my_firstfit_malloc(int size)` that allocates memory using the **first-fit** algorithm. Again, if no empty space is big enough, allocate more via `sbrk()`.

   (b) 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. This outline of this file should look like the following:

   ```
   #include "mymalloc.h"

   struct node *firstNode = NULL;

   void *my_firstfit_malloc(int size)
   ```

```
{
    :
}

void my_free(void *ptr)
{
    :
}
```

3. `mytester.c`: This will the main program that you write to test your version of `malloc()` and `free()`. That is the `main()` function will be located in this file. For this file you have to write your own tester that tries to test every possible situation. The outline of this file should look like the following:

```
#include "mymalloc.h"
#include <stdio.h>

// additional function signatures

// additional functions

int main(void)
{
   :
}
```

4. `mallocdrv.c`: This file is another tester file which we provide. This program will use your version of `malloc()` and `free()` many times. Note that if your implementation of `malloc()` and `free()` fail, this driver program will not be able to detect what is wrong with your program. It usually causes segmentation fault. That is why you need your own tester program. If you are able to execute the `malloctest` program without any segmentation fault, you will see an output that look like the following:

```
original val brk: 0x99e7000
148 134 64 236 87 189 63 221 119 96 81 190 139 40 103 59 117 246 57 239 14
...
91 227 93 150 10 217 158 241 254 15 35 66
brk after  test1: 0x99e7000
1 5 8 11 12 12 12 12 12 13 14 16 16 17 17 17 18 19 20 21 24 24 25 26 27 28
...
238 239 240 243 244 247 248 250 250 251 251 252 255
brk after  test2: 0x99e7000
```

   Note that the value of `brk` after `test1` and `test2` must be exactly the same as the original `brk`. How to build the test program will be explained in the next item.

5. `Makefile`: This is a simple text file for building multi-file environment. I will also supply this file for you. The content of this file is as follows:

```
mytester: mymalloc.o mytester.o
        gcc -m32 -o mytester mytester.o mymalloc.o

mytester.o: mytester.c mymalloc.h
        gcc -m32 -c mytester.c

malloctest: mymalloc.o mallocdrv.o
        gcc -m32 -o malloctest mymalloc.o mallocdrv.o

mymalloc.o: mymalloc.c mymalloc.h
        gcc -m32 -c mymalloc.c

mallocdrv.o: mallocdrv.c mymalloc.h
        gcc -m32 -c mallocdrv.c
```

This file must be located in the same directory of all files above. Note that to create an executable program from your `mytester.c`, type the following command:

```
make mytester
```

If all went well, this will create an executable file named `mytester`. If you want to use supplied driver file, type the following command:

```
make malloctest
```

An executable file named `malloctest` will be created.

Environment

For this project we will again be working on thoth.cs.pitt.edu or on your own Ubuntu machine. Note that in the `Makefile`, we use the flag `-m32` to build 32-bit program instead of 64-bit program. This will prevent pointer cast warnings.

For this project, create a new directory called `project3` under your `cs449` directory and create your program there:

```
mkdir project3
cd project3
```

You can copy two files `mallocdrv.c` and `Makefile` to your directory using the following command:

```
cp /afs/cs.pitt.edu/usr0/tkosiyat/public/cs0449/mallocdrv.c .
cp /afs/cs.pitt.edu/usr0/tkosiyat/public/cs0449/Makefile .
```

For those who use your own Ubuntu machine, these two files are available on the CourseWeb.

Hints/Notes

- When you manually change `brk` with `sbrk`, you may not call `malloc()`, 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 dont lose the beginning of your linked list.
- `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 prototypes of your two functions
- A C file named `mymalloc.c` with the implementations of your two functions
- A C file named `mytester.c` which is the test program you created during your initial testing
- Any documentation you provide to help us grade your project
- All in a tar.gz file, named with your user id (`USERNAME_project3.tar.gz`)
- Copy your archive to the appropriate directory:

  ```
  /afs/cs.pitt.edu/public/incoming/CS0449/tkosiyat/sec1
  ```

  See previous labs or projects for detail instruction how to create the `USERNAME_project3.tar.gz` and how to copy the file to the above submission directory.