Assignment 1

cpe 453 Fall 2015

```
Happiness is good health and a bad memory.
-- Ingrid Bergman
```

— /usr/games/fortune

Due by 11:59:59pm, Friday, October 2nd.
This assignment is to be done individually.

This is a warm-up assignment to get your systems programming skills back up to snuff and, at the same time, to introduce you to the role of the operating system as resource allocator.

Library: malloc

That's it. Implement a memory management system that supports the four C allocation/deallocation functions that you know and love using only the system call `sbrk(2)`. The functions are, of course:

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

You always knew that `malloc(3)` wasn't part of the C language, but rather part of the library. That means we can replace it if we want. In UNIX, the top of data segment is known as the *program break*, and can be moved using one of two system calls, `brk(2)` or `sbrk(2)`. `Sbrk(2)` is the one that is going to be of the most use to us here, because it allows one to adjust the break without previously knowing where it was and returns the old value. This old value is where the new space starts.

Once you have moved the break to get a hunk of memory from the operating system, your task is to parcel it out in response to requests by client programs. There are many ways to do this, but about the simplest is to overlay a linked-list(ish) structure on your heap where each allocated chunk has a header that keeps track of useful information such as its size, whether it is free, etc., and also holds a pointer to the next chunk. This is shown schematically in Figure 1

Once you have such a structure, it is easy to traverse it looking for a suitable portion of memory in response to a `malloc(3)` call. Once you find it, carve it off, update your data structures, and return the pointer to the caller. If there is no suitable chunk, ask the OS for more via `sbrk(2)`. If that fails, return `NULL` and set `errno` to `ENOMEM`.

For the original hunk, you'll have to chose a size. Pick something reasonable that won't have you calling `sbrk(2)` every time someone calls `malloc(3)`, but that also won't be wasteful. For what it's worth, I allocate in 64k chunks. Remember, too, that a request to `malloc(3)` could be bigger than any chunk size you should choose. Be sure to deal with this case correctly.

You'll implement these as both a shared library and a static archive.

Pesky details:

- `malloc(3)` promises that the memory it returns will be properly aligned for any use. For our purposes, this means that all memory returned by your `malloc(3)`, `realloc(3)`, or `calloc(3)` shall be evenly divisible by 16. Intel x86 processors are very forgiving of misaligned data, so you might want to test this on something else if you have access to it.

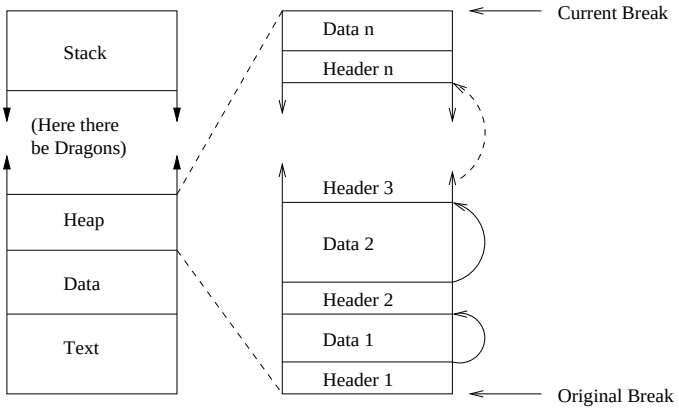

Figure 1: Diagram of memory showing a list-structured heap

- `free(3)` and `realloc(3)` each take a pointer to a block of memory allocated by `malloc(3)`, but the pointer will not necessarily be to the very first byte of region. You must support this ability to discover which allocation unit holds a particular address[1].
- As you go through your heap, cutting off hunks to allocate, fragmentation is going to become a problem. You will have to remember to merge sections of memory if adjacent ones become free.
- `realloc(3)` must try to minimize copying. That is, it must attempt in-place expansion (or shrinking) if it is possible, including merging with adjacent free chunks, if any. If expansion in place is not possible, of course, `realloc(3)` must copy appropriately. If it is unable to allocate new space, it must preserve the original buffer in a safe (allocated) state, but return NULL.
- Also, remember, if `realloc(3)` is called with NULL it does the same thing as `malloc(3)`.
- To facilitate debugging, your library needs to support the environment variable `DEBUG_MALLOC`, which, if set, will cause these functions to narrate their behavior. (See below.)
- Finally, you don't have to support this, but consider the situation where a large hunk of memory becomes free at the high end of the heap. This memory can safely be returned to the operating system to be allocated to any process that needs more memory. It's the right thing to do.

Tricks and Tools

Some potentially useful system calls, library functions, and utilities are listed in Table 1.

Libraries

Libraries come in two forms: *archives,* used for static linking, and *shared objects,* used for dynamic linking. The principle is the same, the only difference is how they're produced and used.

[1]This is the only thing in this specification that differs from "real" `malloc()`

brk(2) sbrk(2)	Set or adjust the program break.
getenv(3)	Read an environment variable
strtol(3) strtoul(3)	String to integer conversion routines
ar(1)	The archive maker
ranlib(1)	Adds an index to archive files
ld.so(8)	The dynamic linker that loads shared objects
gcc(1)	The GNU Compiler Collection
nm(1)	Lists the names defined by a library or object file
stdint.h(0p)	A header file that provides standard integer types, particularly intptr_t and uintptr_t big enough to treat any pointer as an integer.

Table 1: Some potentially useful tools

Archive (.a) libraries are created from object files using ar(1). First compile the object files, then add them to the archive. The r flag means "replace" to insert new files into the archive:

% ar r libname.a obj1.o obj2.o ...obj$n$.o

If you want, you can add an index to speed up linking with ranlib(1):

% ranlib libname.a

To use such a library file, lib*name*.a, you can do one of two things. First, you can simply include it on the link line like any other object file:

% gcc -o prog prog.o thing.o lib*name*.a

Second, you can use the compiler's library finding mechanism. The -L option gives a directory in which to look for libraries and the -l*name* flag tells it to include the archive file lib*name*.a:

% gcc -o prog prog.o thing.o -L. -lname

For shared (.so) libraries, the process is a little different. First the shared object must be built, then the loader (ld.so(8)) has to be told where to find it when a program is executed.

To create the shared object, first compile the object files, then put them together into a library using gcc's -shared flag. You will also have to use -fpic in your CFLAGS to generate position independent code:

% gcc -shared -fpic -o libstuff.so obj1.o obj2.o ...obj$n$.o

Building programs that use this library is just the same as above. The compiler will verify that the needed functions are in the library, but it will not link them until you try and run the program:

% gcc -o prog prog.o thing.o -L. -lname

But if you try and run this program it won't[2] work because the loader, ld.so(8), doesn't know where the library is. The loader is controlled by several environment variables. Primary among these is LD_LIBRARY_PATH a colon-separated list of directories in which to look for libraries in addition to the standard places. These are searched in order, so if you put your library directory ahead of /usr/lib in the search path, it should grab your malloc(3) before the one in the C library (libc.so). Assuming LD_LIBRARY_PATH exists[3], in [t]csh this would be:

% setenv LD_LIBRARY_PATH /wherever/your/library/is:$LD_LIBRARY_PATH

In [ba]sh it's:

$ LD_LIBRARY_PATH=/wherever/your/library/is:$LD_LIBRARY_PATH

[2]This (non) behavior is demonstrated by accident in the sample runs below.

[3]If not, it's even easier: setenv LD_LIBRARY_PATH /wherever/your/library/is

```
$ export LD_LIBRARY_PATH
```

The advantage of the shared library is, of course, that the library can change even after the application has been built, and any bug-fixes, etc., will be active immediately. (This is also the disadvantage of shared libraries: programs that have been stable for years can have new bugs injected into them by changes in the library.)

Note: If you add libraries for multiple architectures to your `LD_LIBRARY_PATH`, the loader will automatically choose the correct one.

Note, too: If you want to have *every* dynamically linked program you run use your library, set `LD_PRELOAD`. This is a list of libraries to be loaded before anything else. If you include a version of `malloc(3)` in a preloaded library, that's the library your programs will use.

Debugging output

If the environment variable `DEBUG_MALLOC` is defined, the four library functions must narrate their behavior on `stderr` with messages of the following form (where the actual values for the location and size of the allocated region are substituted for the `printf(3)` formats, of course):

```
MALLOC: malloc(%d)      =>  (ptr=%p, size=%d)
MALLOC: calloc(%d,%d)   =>  (ptr=%p, size=%d)
MALLOC: realloc(%p,%d)  =>  (ptr=%p, size=%d)
MALLOC: free(%p)
```

You will find this debugging output particularly useful for making sure that you've linked against the correct version of `malloc(3)`. Remember, if you build the library (or set up your environment variables) incorrectly, "real" `malloc(3)` still exists in the C library, so you could find yourself silently testing the wrong version. If you set `DEBUG_MALLOC` and it starts babbling, you can be confident that you have the right functions.

It is not important to make the debugging output particularly efficient. A little slowing down is ok here.

Note: The 64-bit version of glibc's `printf(3)` calls `malloc(3)`. A trick to get around this would be to use something like `snprintf(3)` that uses a fixed-size buffer so `printf(3)` has no reason to want memory, then use `fputs(3)` or `write(2)` to do the actual writing. Also, `snprintf(3)` calls `free(NULL)` at the end. Be sure your library can cope with that. If you're not careful you'll put yourself into an infinite recursion reporting on that call.

Coding Standards and Make

See the pages on coding standards and make on the cpe 453 class web page.

What to turn in

Submit via `handin` in the CSL to the `asgn1` directory of the `pn-cs453` account:

- Your well-documented source file(s).
- A makefile (called `Makefile`) that will build your libraries when given either "`make malloc`" or just "`make`".

  For testing purposes, a special target, "`intel-all`" is also required that will produce 32- and 64-bit versions of the shared library (`libmalloc.so`) in subdirectories of the current directory

called "`lib`" and "`lib64`" respectively[4].

- A README file that contains:
  - Your name.
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

  The README file should be **plain text,** i.e, **not a Word document**, and should be named "README", all capitals with no extension.

Testing

For testing purposes, I have published a test harness, `~pn-cs453/demos/tryAsgn1`, that will attempt to build your library and run it against a set of test files. This is not a complete set of test cases by any means, but if you can't pass these, there's clearly something wrong. A couple of notes:

- It tests both the 32- and 64-bit versions of the libraries. This will only work if you're on a 64-bit machine. Most of the desktops are 64-bit, but not the servers with the exception of `unix[11-15]`.
- Some of these tests allocate quite a bit of memory. It is possible to actually run out of memory, which is indistinguishable from errors in the library. Your error checking output should be able to tell you, though, if you report failures of `sbrk(2)`.
- Finally, there's no reason to copy the script. Simply run "`~pn-cs453/demos/tryAsgn1`" from the directory where your source lives.

Sample runs

Below are some sample runs of building and testing this library. I have also included my version in `~pn-cs453/demos/lib`, `~pn-cs453/demos/lib64`, and `~pn-cs453/demos/libSparc` (as appropriate) if you want to try linking against them. That version responds to numeric values of `DEBUG_MALLOC` by getting more and more verbose (up to 2 as of this writing).

```
$ make clean
rm -f malloc.o  *~ TAGS
$ make intel-all
mkdir lib
gcc -Wall -g -fpic  -m32 -c -o malloc32.o malloc.c
gcc -Wall -g -fpic  -m32 -shared -o lib/libmalloc.so malloc32.o
mkdir lib64
gcc -Wall -g -fpic  -c -o malloc64.o malloc.c
gcc -Wall -g -fpic  -shared -o lib64/libmalloc.so malloc64.o
$ cd Test/
$ cat tryme.c
#include<string.h>
```

[4]How, you ask? You can force gcc to compile in 32- or 64-bit mode by using the `-m32` or `-m64` switches, respectively, or copy the Makefile excerpt from Figure 2. **Note:** Do not simply cut and paste from this PDF. It uses characters that *look like* hyphens but are not.

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

int main(int argc, char *argv[]) {
  char *s;
  s = strdup("Tryme");   /* should call malloc() implicitly */
  puts(s);
  free(s);
  return 0;
}
$ make
gcc -Wall -g    -c -o tryme.o tryme.c
gcc -L ~pn-cs453/demos/lib -lmalloc -o tryme tryme.o
$ ./tryme
./tryme: error while loading shared libraries: libmalloc.so: cannot open
shared object file: No such file or directory
$ LD_LIBRARY_PATH=$HOME/demos/lib:$LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH
$ ./tryme
Tryme
$ DEBUG_MALLOC=
$ export DEBUG_MALLOC
$ ./tryme
MALLOC: malloc(6)       => (ptr=0x01a2d030, size=16)
Tryme
MALLOC: free(0x01a2d030)
$
```

```
CC = gcc

CFLAGS = -Wall -g -fpic

intel-all: lib/libmalloc.so lib64/libmalloc.so

lib/libmalloc.so: lib malloc32.o
	$(CC) $(CFLAGS) -m32 -shared -o $@ malloc32.o

lib64/libmalloc.so: lib64 malloc64.o
	$(CC) $(CFLAGS) -shared -o $@ malloc64.o

lib:
	mkdir lib

lib64:
	mkdir lib64

malloc32.o: malloc.c
	$(CC) $(CFLAGS) -m32 -c -o malloc32.o malloc.c

malloc64.o: malloc.c
	$(CC) $(CFLAGS) -m64 -c -o malloc64.o malloc.c
```

Figure 2: Make dependencies for `intel-all`