Assignment 1 Solutions

cpe 453 Winter 2023

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

— /usr/games/fortune

Due by 11:59:59pm, Sunday, January 22nd.
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.

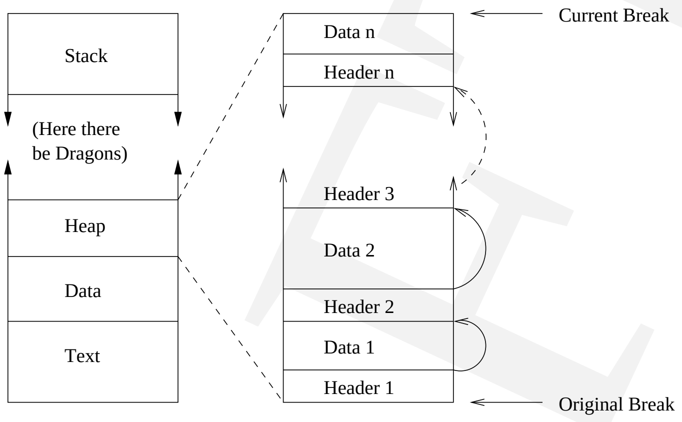

Figure 1: Diagram of memory showing a list-structured heap

- `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)`, and if `realloc(3)` is called with a size of 0 and the pointer is not NULL, it's equivalent to `free(ptr)`.[2]
- 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.

[1]This is the only thing in this specification that differs from "real" `malloc()`
[2]Note that if you opt to have `malloc(0)` return NULL, these are equivalent.

`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

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.

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 ...objn.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 libname.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 ...objn.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[3] 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

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

(libc.so). Assuming LD_LIBRARY_PATH exists[4], 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
$ 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.

Of particular interest for this assignment is that make knows things. One of the two required targets for this assignment is malloc that is supposed to build libmalloc.a and libmalloc.so in the current directory. For this, you will probably create a dependency that looks something like:

```
malloc:  libmalloc.a libmalloc.so
```

You'll be surprised when you run make malloc that make builds the libraries, then goes on to try and make your malloc.c into a program—because you didn't provide a command and it thinks it knows how to make a program out of a .c file—which won't work. To avoid this behavior, you'll

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

have to either provide a harmless command (.e.g "@echo done") or tell make that this isn't a real recipe by adding the following line to the makefile:

```
.PHONY: malloc
```

This will only work for GNU make, but that's the version installed on Linux.

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[5].
- 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 as are the `unix[1-5]` servers.
- 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.

[5]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.

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>
#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 -o tryme tryme.o -lmalloc
$ ./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)
$
```

Solution:

```
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`

File	**Where**
Makefile	p.9
malloc.c	p.11
test.c	p.19

```
CC     = gcc

SHELL  = /bin/sh

CFLAGS = -Wall -g -fpic

AR     = ar r

RANLIB = ranlib

ARCHIVE = libmalloc.a

SO     = libmalloc.so

OBJS   = malloc.o

SRCS   = malloc.c

HDRS   =

EXTRACLEAN = core libmalloc.a libmalloc.so lib lib64 malloc32.o malloc64.o

.PHONY: all shared archive allclean clean malloc intel-all test

all:    shared archive

malloc: shared archive

shared: $(SO)

archive: $(ARCHIVE)

intel-all: lib/$(SO) lib64/$(SO)

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

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

lib:
	mkdir lib

lib64:
	mkdir lib64

allclean: clean
	@rm -rf $(EXTRACLEAN)

clean:
	rm -f $(OBJS) *~ TAGS

$(ARCHIVE): $(OBJS)
	$(AR) $@ $(OBJS)
	ranlib $@

$(SO): $(OBJS)
	$(CC) $(CFLAGS) -shared -o $@ $(OBJS)

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

depend:
	@echo Regenerating local dependencies.
	@makedepend -Y $(SRCS) $(HDRS)

tags : $(SRCS) $(HDRS)
	etags $(SRCS) $(HDRS)
```

```
test: intel−all
        ~pn−cs453/demos/tryAsgn1 | tee Test.log 2>&1

# DO NOT DELETE
```

```
#include <errno.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

extern void *_sbrk(intptr_t);
void *sbrk(intptr_t size) {
  return _sbrk(size);
}

/*
 * Vocabulary:
 *     CHUNK: how much to move the break each time
 *     MIN_BLOCK: the smallest allowable allocation unit.   (really
 *                MIN_BLOCK + header_size
 */

#ifndef CHUNK
#define CHUNK (1<<16)
#endif

#define MIN_BLOCK 16

/* the number of hex digits in a pointer or long */
#define PTRWID ((int)(2*sizeof(void*)))


typedef struct mheader *mheader;
struct mheader {
  void *  base;               /* base of allocated segment */
  size_t  size;               /* size in bytes */
  int     free;               /* is this free? */
  mheader prev;               /* pointer to next record */
  mheader next;               /* pointer to previous record */
};

static int debug_malloc=0;        /* flag for verbosity */

/* two useful globals: the head and the tail of the heap list */
static mheader memory=NULL;     /* the hunk o' memory we're allocating */
static mheader memend=NULL;     /* the last chunk in memory            */

/* one useful constant: size of a header, rounded up to
 * preserve alignment.
 */
#define HEADER_SIZE (MIN_BLOCK * (sizeof(struct mheader)/MIN_BLOCK + \
                     ((sizeof(struct mheader)%MIN_BLOCK)?1:0)))


/* useful prototypes */
static size_t lmgt(size_t factor, size_t size);
static mheader find_block(mheader list, size_t size);
static mheader find_block_containing(mheader list, void *ptr);
static mheader expand_break(size_t hint);
static int malloc_init(void);
static void split_block(mheader block, size_t size);
static void merge_blocks(mheader one, mheader two);
static void internal_free(void *ptr);
static void *internal_malloc(size_t size);
void pm();                        /* debugging dump memlist */
static int logmsg(int level, FILE *stream, const char *format, ...); /* log */
static mheader find_block(mheader list, size_t size) {
  /* find a free hunk of size size or bigger in the list */
  mheader block;
  for(block=list; block && ( !block->free || block->size < size ) ;
      block=block->next)
    /* search the list */;

  return block;
}

static mheader find_block_containing(mheader list, void *ptr) {
  /* find the block containing the given pointer */
  while ( list &&
          ( (ptr < list->base) || ((list->base+list->size) <= ptr) ) )
    list = list->next;
  /* this'll be the block, or NULL if not found */
  return list;
}

static mheader expand_break(size_t hint) {
  /* expand the heap and return a pointer to header containing the new
   * space (or NULL on failure).
   * If hint is zero, requests CHUNK bytes.  If non-zero, requests the
   * smallest number of CHUNKs with enough space to hold hint.
   */
  void *brk, *rvalue;
  size_t ask, howmuch;
  int extra;

  if ( hint )  /* this is what we want */
    howmuch = lmgt(CHUNK,hint);
  else
    howmuch = CHUNK;

  ask = howmuch;


  logmsg(1,stderr,"%s:  trying to add %u bytes to break...", __FUNCTION__,
          (unsigned) ask);

  if ( !memory ) {              /* This is the first time */
    brk = sbrk(0);              /* find the current break */
    if ( brk == (void*)-1) {
      logmsg(1,stderr,"FAILED.\n");
      return NULL;
    }

    /* round up to nearest aligned location:  This will be the base
     * of the new memory.  Make sure the heap is aligned, and that
     * the ask contains a full aligned CHUNK (or howmuch, if bigger)
     */
    extra = (uintptr_t)brk % MIN_BLOCK;
    if ( extra ) {
      ask += MIN_BLOCK - extra;
      brk += MIN_BLOCK - extra;
    }
    memory = brk;               /* this is now the head */

    /* move the break */
    brk = sbrk(ask);
    if ( brk == (void*)-1) {
      memory = NULL;
      perror(__FUNCTION__);
      return NULL;
    }
    /* it worked, hook it up.  */
    memory->base = ((void *)memory) + HEADER_SIZE;
    memory->size = howmuch - HEADER_SIZE;
    memory->free = TRUE;
    memory->prev = NULL;
    memory->next = NULL;
    memend = memory;
    rvalue = memory;
  } else {
    /* Memory already exists.  Just move the break */
    brk = sbrk(ask);
    if ( brk == (void*)−1) {
      logmsg(1,stderr,"FAILED.\n");
      return NULL;
    }
    /* it worked, hook it up. */
    memend−>next = brk;
    memend−>next−>base = brk + HEADER_SIZE;
    memend−>next−>size = howmuch − HEADER_SIZE;
    memend−>next−>free = TRUE;
    memend−>next−>prev = memend;
    memend−>next−>next = NULL;
    if ( memend−>free ) {
      merge_blocks(memend,memend−>next);
    } else {
      memend=memend−>next;
    }
    rvalue=memend;
  }

  logmsg(1,stderr,"ok.\n");

  return rvalue;
}

static int malloc_init(void) {
  /* set up the world as we'd like it to be */
  int ok = TRUE;
  char *debug, *end;

  /* check for debugging flag */
  if ( (debug=getenv("DEBUG_MALLOC")) ) {
    debug_malloc=strtol(debug, &end, 0);
    if ( debug_malloc == 0 )
      debug_malloc++;
    if ( debug_malloc < 0 || *end ) {
      /* if this was a bad number, complain and just set it to 1 */
      debug_malloc=TRUE;
      fprintf(stderr,
             "%s:  invalid value for DEBUG_MALLOC.  Setting to %d.\n",
             debug, debug_malloc);
    }
  }

  /* move the break and allocate a hunk of memory to work with */
  memory=expand_break(0);

  if ( !memory )
    ok = FALSE;

  return ok;
}

static void split_block(mheader block, size_t size) {
  /* divide the given block into two blocks, one that can hold size
   * blocks (rounded to a multiple of MIN_BLOCK), and the rest.  Make
   * sure there's enough room to make it worth it.
   */
  mheader rest;

  /* round size up to nearest multiple of MIN_BLOCK */
  size = lmgt(MIN_BLOCK,size);

  if ( block−>size >= size + (2*HEADER_SIZE) + MIN_BLOCK ) {
    /* There's room for a valid fragment; it's worth it; chop off rest. */
    /* create it */
    rest = block−>base + size;

    /* cut it in */
    rest −>prev = block;
    rest −>next = block−>next;
    if ( rest−>next )
      rest−>next−>prev = rest;
    block−>next = rest;
    /* set it up */
    rest −>free = TRUE;
    rest −>base = ((void*)rest) + HEADER_SIZE;
    rest −>size = block−>size − size − HEADER_SIZE;
    block−>size = size;
    if ( memend == block ) /* keep track of the end of the world */
      memend = rest;

    /* check to see if the new block can be merged w/its successor */
    if ( rest−>next && rest−>next−>free )
      merge_blocks(rest, rest−>next);
  }
}

static void merge_blocks(mheader one, mheader two){
  /* Merge the given blocks.  Insist that
   * (a) they be contiguous, and
   * (b) the second block must be free
   */
  if ( two−>free && one−>next == two ){ /* merge 'em */
    one−>size = one−>size + two−>size + HEADER_SIZE;
    one−>next = two−>next;
    if (two−>next)
      two−>next−>prev = one;
    /* keep  track of the end of the world */
    if ( memend == two )
      memend = one;
  } else {                    /* whoops */
    if ( !two−>free )
      fprintf(stderr, "%s:  attempt to merge allocated block", _FUNCTION_);
    else
      fprintf(stderr, "%s:  attempt to merge non-contiguous blocks",
              _FUNCTION_);
  }
}

void *malloc(size_t size) {
  /* allocate the given amount of space.  */
  void *rvalue;

  rvalue = internal_malloc(size);

  if ( debug_malloc ){
    mheader b=find_block_containing(memory,rvalue);
    logmsg(1,stderr, "MALLOC: malloc(%ld)\t=> (ptr=%p, size=%ld)\n",
           (long)size, rvalue, (long)(b?b−>size:0));
    if ( debug_malloc > 2 )
      pm();
  }

  return rvalue;
}

static void *internal_malloc(size_t size) {
  /* allocate the given amount of space.  (Rounding takes place in
   * split_block().  This exists because malloc() is called by realloc()
   * and calloc().  We don't want to confuse the diagnostic output.
   * Returns the block containing the newly allocated memory.
   */
  mheader block, rest;
  void *rvalue;

  if ( (size < 0 ) ||                /* if bad size   */
       (!memory && !malloc_init()) ) { /* or initialization fails */
    errno  = ENOMEM;                 /* bummer */
    rvalue = NULL;
  } if ( size == 0 ) {
    rvalue = NULL;        /* It's allowed... */
  } else {                /* find the right size block and return it */

    /* try and find a block.  If not, try to expand the heap until either
     * we find one or the expansion fails
     */
    if ( !(block = find_block(memory,size)) ) {
      do {        /* ask for this and a little bit more.
                   * repeat in case we don't get enough somehow
                   */
        rest = expand_break(size+CHUNK);
        block = find_block(rest,size);
      } while ( rest && !block );
    }

    if ( !block ) {              /* still no dice.   bummer. */
      rvalue = NULL;
      errno  = ENOMEM;
    } else {                     /* woohoo */
      block->free = FALSE;
      rvalue=block->base;
      split_block(block, size);
    }
  }

  return rvalue;
}

static void free_block(mheader block) {
  int ok;
  size_t amount, newsize;

  /* free a block, merging w/neighbors if possible */
  block->free = TRUE;
  /* check for possible merges with neighbors, next first, then
   * prev. This maintains the validity of the block pointer.
   */
  if ( block->next && block->next->free)
    merge_blocks(block, block->next);
  if ( block->prev && block->prev->free)
    merge_blocks(block->prev, block);

  /* If more than two CHUNKs now available at the high end of memory
   * some should be returned to the OS
   */
  if ( memend->free && memend->size > 2 * CHUNK ) {
    newsize = CHUNK + (memend->size % CHUNK);
    amount = memend->size - newsize;

    /* try to return amount to system */
    logmsg(1,stderr,"%s:  trying to return %u bytes from break...",
          _FUNCTION_, (unsigned) amount);

    ok = (sbrk(-amount) != (void*) -1);
    if ( ok ) {
      /* it worked.   Yay.   Adjust our counters */
      memend->size = newsize;
      logmsg(1,stderr,"ok.\n");
    } else {
      /* it didn't work, but it's harmless */
      logmsg(1,stderr,"FAILED.\n");
    }

  }

}

void free(void *ptr) {
  /* Release the given memory, or do nothing if ptr is NULL */

  if ( ptr ) {                    /* if ptr is NULL, do nothing */
    logmsg(1, stderr, "MALLOC: free(0x%p)\n", ptr);
    if ( debug_malloc > 2 )
      pm();
    internal_free(ptr);
  } else if ( debug_malloc) {
    /* sigh.  Special–case this one because snprintf() free()s NULL */
    char *msg="MALLOC: free(NULL)\n";
    write(STDERR_FILENO, msg, strlen(msg));
  }
}

static void internal_free(void *ptr) {
  /* return the block containing this pointer to the wild
   * free() is also called by realloc() so, like malloc, we have
   * internal and external versions
   */
  mheader block;
  block = find_block_containing(memory, ptr);
  if ( block ) {
    free_block(block);
  } else {
    fprintf(stderr, "%s:  freeing unallocated pointer!\n", __FUNCTION__);
  }

  return;
}


void *calloc(size_t nmemb, size_t elem_size){
  /* allocate memb elements of size size, cleared */
  void *rvalue;
  size_t size;

  #ifdef GCC
  if (__builtin_mul_overflow(nmemb,elem_size,&size)) {
    fprintf(stderr,"%s:  overflow detected\n",__FUNCTION__);
    exit(EXIT_FAILURE);
  };
  #else
  size = nmemb*elem_size;
  #endif

  if ( (rvalue = internal_malloc(size)) ) { /* get it  */
    memset(rvalue,0,size);                  /* wipe it if non–NULL */
  }

  if ( debug_malloc ) {
    if ( rvalue ) {
      mheader b=find_block_containing(memory, rvalue);
      logmsg(1, stderr, "MALLOC: calloc(%ld,%ld)\t=> (ptr=%p, size=%ld)\n",
             (long)nmemb, (long)elem_size, rvalue, (long)(b?b–>size:0));
    } else {
      logmsg(1,stderr, "MALLOC: calloc(%ld,%ld)\t=> NULL\n", (long)nmemb,
             (long)elem_size);
    }
    if ( debug_malloc > 2 )
      pm();
  }

  return rvalue;
}

void *realloc(void *ptr, size_t size) {
  /* reallocate the given pointer to the given size, copying, if necessary.
   * (rounding takes place in split_block())
   */
  void *rvalue=NULL, *new;
  mheader block;

  if ( ptr == NULL ) {
    rvalue = internal_malloc(size);
  } else if ( size == 0 ) {
    internal_free(ptr);
    rvalue=NULL;                /* it's just neater this way */
  } else {
    block = find_block_containing(memory, ptr);
    if ( block && (size > 0) ) {
      /* This was allocated by us (and the new size makes sense).
       * Otherwise the default NULL will be returned.
       */

      /* try to extend this block, if possible (it can't hurt) */
      while ( block->next && block->next->free )
        merge_blocks(block, block->next);

      /* now, will it fit? */
      if ( size <= block->size ) {
        /* yes, woohoo! we don't have to move*/
        split_block(block,size);
        rvalue = block->base;
      } else {
        /* nope.  Get a new block and copy.  Block's size is smaller,
         * so copy block->size rather than size */
        if ( (new = internal_malloc(size)) ) {
          memcpy(new, block->base, block->size);
          rvalue=new;
          free_block(block);
        } else {                /* no dice */
          rvalue = NULL;
        }
      }
    }
  }

  if ( debug_malloc ) {
    if ( rvalue ) {
      mheader b=find_block_containing(memory,rvalue);
      logmsg(1,stderr,
             "MALLOC: realloc(%p,%ld)\t=> (ptr=%p, size=%ld)\n",
             ptr, (long)size, rvalue, (long)(b?b->size:0));
    } else {
      logmsg(1,stderr, "MALLOC: realloc(%p,%ld)\t=> NULL\n", ptr,
             (long)size);
    }
    if ( debug_malloc > 2 )
      pm();
  }


  return rvalue;
}

#define powOf2(x) (((x != 0) && !(x & (x - 1))))

static size_t lmgt(size_t factor, size_t size) {
  /* LMGT--least multiple greater than.  Rounds size up
   * to the nearest multiple of factor
   */
    size_t extra;
    if ( powOf2(factor) )
      extra = size & (factor-1); /* mask if it's a power of two */
    else
      extra = size%factor;


    if (extra) {
      size = size-extra + factor;
    }
    return size;
}


/****************************************************************/
void print_memlist(FILE *where) {
  /* debugging code to print the memory list.  Since we only do thi
```

```
   * while debugging, it should be ok to use logmsg(1,...) here to
   * avoid printf() using malloc() and changing things.
   */
  mheader l;
  int entry=0;
  static int count=0;

  logmsg(1,where,"  Memory Map (time=%d  memory=%p  memend=%p):\n",
         count++, memory, memend);
  for(l=memory; l; l=l->next )
    logmsg(1, where,
           "    %0d-%p) {base=%p,size=%*ld,free=%d,prev=%p,next=%p}\n",
           entry++, l, l->base, PTRWID, (long int) l->size, l->free,
           l->prev, l->next);
}

void pm() {
  print_memlist(stderr);
}



#define LOGMSGSIZE 1024
static int logmsg(int level, FILE *stream, const char *format, ...) {
  /* print out logging messages.  This uses a fixed-size buffer
   * to be sure that snprintf() has no reason to call malloc()
   * (Although it still free()s NULL.  Why?)
   */
  char msg[LOGMSGSIZE];
  int rval=0;
  va_list ap;

  if ( debug_malloc >= level ) {
    va_start(ap, format);
    vsnprintf(msg, LOGMSGSIZE, format, ap);
    va_end(ap);
    rval = fputs(msg, stream);
  }

  return rval;
}
```

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

int main(int argc, char *argv[]) {
  char *c;
  c = malloc(10);
  return 0;
}
```