**1 Overview**

This problem features two problems pertaining to the most recent and upcoming lecture topics.

1. EL Malloc implements a simple, explicit list memory allocator. This manages heap memory in doubly linked lists of Available and Used memory blocks to provide el\_malloc() / el\_free(). It could be extended with some work to be a drop-in replacement for malloc() / free()
2. showsym uses mmap() to parse a binary ELF file to print its symbol table. Tools that work with object files like the linker associated with the GCC and the program loader must perform similar though more involved tasks involving ELF files.

**2****Download Code and Setup**

Download the code pack linked at the top of the page. Unzip this which will create a project folder. Create new files in this folder. Ultimately you will re-zip this folder to submit it.

| **File** | **State** | **Notes** |
| --- | --- | --- |
| Makefile | Provided | Build file to compile all programs |
| el\_malloc.h | Provided | Problem 1 header file |
| el\_demo.h | Provided | Problem 1 demo main() |
| el\_malloc.c | COMPLETE | Problem 1 implemented REQUIRED functions |
| showsym.c | COMPLETE | Problem 2 template to complete |
| data/\* | Data | Problem 2 files for input to showsym |
| data/quote\_data.o | Data | Problem 2 ELF object file for input to showsym |
|  |  | Several other ELF and non-ELF files provided |

**3 Problem 1: EL Malloc**

A **memory allocator** is small system which manages heap memory, sometimes referred to as the "data" segment of a program. This portion of program memory is a linear set of addresses that form a large block which an expand at runtime by making requests to the operating system. Solving the allocation problem forms backbone of what malloc()/free() do by keeping track of the space used and released by a user program. Allocators also see use in garbage collected languages like Java where there are no explicit free() calls but the allocator must still find available space for new objects.

One simple way to implement an allocator is to overlay linked lists on the heap which track at a minimum the available chunks of memory and possibly also the used chunks. This comes with a cost: some of the bytes of memory in the heap are no longer available for the user program but are instead used for the allocator's book-keeping.

In this problem, an *explicit list* allocator is developed, thus the name of the system el\_malloc. It uses two lists to track memory in the heap:

* The **Available List** of blocks of memory that can be used to answer calls to malloc()
* The **Used List** of blocks that have been returned by malloc() and should not be returned again until they are free()'d

Most operations boil down to manipulating these two lists in some form.

* Allocating with ptr = el\_malloc(size); searches the Available List for a block with sufficient size. That block is split into two blocks. One block answers the request and is given about size bytes; it is moved to the Used List. The second block comprises the remainder of the space and remains on the Available List.
* Deallocating with el\_free(ptr); moves the block referenced by ptr from the Used List to the Available List. To prevent fragmentation of memory, the newly available block is **merged** with adjacent available blocks if possible.

**3.1 EL Malloc Data Structures**

Several data structures defined in el\_malloc.h should be studied so that one is acquainted with their intent. The following sections outline many of these and show diagrams to indicate transformation the required functions should implement.

**3.2 Block Header/Footer**

Each block of memory tracked by EL Malloc is preceded and succeeded by some bytes of memory for book keeping. These are referred to as the block "header" and "footer" and are encoded in the el\_blockhead\_t and el\_blockfoot\_t structs.

// type which is a "header" for a block of memory; containts info on

// size, whether the block is available or in use, and links to the

// next/prev blocks in a doubly linked list. This data structure

// appears immediately before a block of memory that is tracked by the

// allocator.

typedef struct block {

size\_t size; // number of bytes of memory in this block

char state; // either EL\_AVAILABLE or EL\_USED

struct block \*next; // pointer to next block in same list

struct block \*prev; // pointer to previous block in same list

} el\_blockhead\_t;

// Type for the "footer" of a block; indicates size of the preceding

// block so that its header el\_blockhead\_t can be found with pointer

// arithmetic. This data appears immediately after an area of memory

// that may be used by a user or is free. Immediately after it is

// either another header (el\_blockhead\_t) or the end of the heap.

typedef struct {

size\_t size;

} el\_blockfoot\_t;

As indicated, the blocks use **doubly linked nodes** in the header which will allow easy re-arrangement of the list.

A picture of a block with its header, footer, and user data area is shown below.

Figure 1: Block data preceded by a header (el\_blockhead\_t) and followed by a footer (el\_blockfoot\_t).\_

**3.3 Footers and Blocks Above/Below**

One might wonder why the footer appears. In tracking blocks, there will arise the need to determine a block that is immediately before a given block in memory (not the previous in the linked list). The footer enables this by tracking the size of the user block of memory immediately preceding it.

This is illustrated in the diagram below.

Figure 2: Finding preceding block header using footer (el\_block\_decr(header))

This operation is implemented in the function el\_block\_below(block) and the similar operation el\_block\_above(block) finds the next header immediately following one in memory.

The following functions use pointer arithmetic to determine block locations from a provided pointer.

el\_blockfoot\_t \*el\_get\_footer(el\_blockhead\_t \*block);

el\_blockhead\_t \*el\_get\_header(el\_blockfoot\_t \*foot);

el\_blockhead\_t \*el\_block\_above(el\_blockhead\_t \*block);

el\_blockhead\_t \*el\_block\_below(el\_blockhead\_t \*block);

These functions benefit from macros defined in el\_malloc.h that are useful for doing pointer operations involving bytes.

// macro to add a byte offset to a pointer, arguments are a pointer

// and a # of bytes (usually size\_t)

#define PTR\_PLUS\_BYTES(ptr,off) ((void \*) (((size\_t) (ptr)) + ((size\_t) (off))))

// macro to add a byte offset to a pointer, arguments are a pointer

// and a # of bytes (usually size\_t)

#define PTR\_MINUS\_BYTES(ptr,off) ((void \*) (((size\_t) (ptr)) - ((size\_t) (off))))

// macro to add a byte offset to a pointer, arguments are a pointer

// and a # of bytes (usually size\_t)

#define PTR\_MINUS\_PTR(ptr,ptq) (((size\_t) (ptr)) - ((size\_t) (ptq)))

**3.4 Block Lists and Global Control**

The main purpose of the memory allocator is to track the available and used blocks in explicit linked lists. This allows used and available memory to be distributed throughout the heap. Below are the data structures that track these lists and the global control data structure which houses information for the entire heap.

// Type for a list of blocks; doubly linked with a fixed

// "dummy" node at the beginning and end which do not contain any

// data. List tracks its length and number of bytes in use.

typedef struct {

el\_blockhead\_t beg\_actual; // fixed node at beginning of list; state is EL\_BEGIN\_BLOCK

el\_blockhead\_t end\_actual; // fixed node at end of list; state is EL\_END\_BLOCK

el\_blockhead\_t \*beg; // pointer to beg\_actual

el\_blockhead\_t \*end; // pointer to end\_actual

size\_t length; // length of the used block list (not counting beg/end)

size\_t bytes; // total bytes in list used including overhead;

} el\_blocklist\_t;

// NOTE: total available bytes for use/in-use in the list is (bytes - length\*EL\_BLOCK\_OVERHEAD)

// Type for the global control of the allocator. Tracks heap size,

// start and end addresses, total size, and lists of available and

// used blocks.

typedef struct {

void \*heap\_start; // pointer to where the heap starts

void \*heap\_end; // pointer to where the heap ends; this memory address is out of bounds

size\_t heap\_bytes; // number of bytes currently in the heap

el\_blocklist\_t avail\_actual; // space for the available list data

el\_blocklist\_t used\_actual; // space for the used list data

el\_blocklist\_t \*avail; // pointer to avail\_actual

el\_blocklist\_t \*used; // pointer to used\_actual

} el\_ctl\_t;

The following diagram shows some of the structure induced by use of a doubly linked lists overlaid onto the heap. The global control structure el\_ctl has two lists for available and used space.

Figure 3: Structure of heap with several used/available blocks. Pointers from el\_ctl lists allow access to these blocks.

The following functions initialize the global control structures, print stats on the heap, and clean up at the end of execution.

int el\_init(int max\_bytes);

void el\_print\_stats();

void el\_cleanup();

**3.5 Pointers and "Actual" Space**

In several structures, there appear pointers named xxx and structs named xxx\_actual. For example, in el\_blocklist\_t:

typedef struct {

...

el\_blockhead\_t beg\_actual; // fixed node at beginning of list; state is EL\_BEGIN\_BLOCK

el\_blockhead\_t \*beg; // pointer to beg\_actual

...

} el\_blocklist\_t;

The intent here is that there will *always* be a node at the beginning of the doubly linked list which just to make the programming easier so it makes sense to have an actual struct beg\_actual present. However, when working with the list, the address of the beginning node is often referenced making beg useful. In any case, beg will be initialized to &beg\_actual as appears in el\_init\_blocklist().

void el\_init\_blocklist(el\_blocklist\_t \*list){

list->beg = &(list->beg\_actual);

list->beg->state = EL\_BEGIN\_BLOCK;

list->beg->size = EL\_UNINITIALIZED;

...

}

Similarly, since there will always be an Available List, el\_ctl\_t has both an avail pointer to the list and avail\_actual which is the struct for the list.

**3.6 Doubly Linked List Operations**

A large number of operations in EL Malloc boil down to doubly linked list operations. This includes

* Unlinking nodes from the middle of list during el\_free()
* Adding nodes to the beginning of a headed list (allocation and free)
* Traversing the list to print and search for available blocks

Recall that unlinking a node from a doubly linked list involves modifying the previous and next node as in the following.

node->prev->next = node->next;

node->next->prev = node->prev;

while adding a new node to the front is typically accomplished via

node->prev = list->beg;

node->next = list->beg->next;

node->prev->next = node;

node->next->prev = node;

You may wish to review doubly linked list operations and do some reading on lists with "dummy" nodes at the beginning and ending if these concepts are rusty.

The following functions pertain to block list operations.

void el\_init\_blocklist(el\_blocklist\_t \*list);

void el\_print\_blocklist(el\_blocklist\_t \*list);

void el\_add\_block\_front(el\_blocklist\_t \*list, el\_blockhead\_t \*block);

void el\_remove\_block(el\_blocklist\_t \*list, el\_blockhead\_t \*block);

**3.7 Allocation via Block Splitting**

The basic operation of granting memory on a call to el\_malloc(size) involves finding an Available Block with enough bytes for **both** the requested amount of memory and a new header/footer combination. The current requirement is that **a block always gets split** on an allocation though a straight-forward optimization would be to not split in the case of a close or exact size match.

This process is demonstrated in the below diagram in which a request for some bytes has been made. The process prior to diagram involves searching the Available List for a block with enough space. Once found, the block is split into the portion that will be used and the remaining portion. A pointer to the user area immediately after theel\_blockhead\_t is returned.

Figure 4: Splitting a block in an allocation request.

The following functions pertain to the location and splitting of blocks in the available list to fulfill allocation requests.

el\_blockhead\_t \*el\_find\_first\_avail(size\_t size);

el\_blockhead\_t \*el\_split\_block(el\_blockhead\_t \*block, size\_t new\_size);

void \*el\_malloc(size\_t nbytes);

**3.8 Freeing Blocks and Merging**

Freeing memory passes in a pointer to the user area that was granted. Immediately preceding this should be a el\_blockhead\_t and it can be found with pointer arithmetic.

In order to prevent memory from becoming continually divided into smaller blocks, on freeing the system checks to see if adjacent blocks can be merged. Keep in mind that the blocks that can be merged are **adjacent in memory**, not next/previous in some linked list. Adjacent blocks can be located using el\_block\_above() and el\_block\_decr().

To merge, the adjacent blocks must both be Available (not Used). A free can then have several cases.

1. The freed block cannot be merged with any others
2. The freed block can be merged with only the block above it
3. The freed block can be merged with only the block below it
4. The freed block can be merged with both adjacent blocks

The diagrams below show two of these cases.

Figure 5: Two cases of freeing blocks. The 2nd involves merging adjacent nodes with available space.

With careful use of the below functions and handling of NULL arguments, all 4 cases can be handled with very little code. Keep in mind that el\_block\_above()/below()should return NULL if there is no block above or below due to that are being out of the boundaries of the heap.

el\_blockhead\_t \*el\_block\_above(el\_blockhead\_t \*block);

el\_blockhead\_t \*el\_block\_below(el\_blockhead\_t \*block);

void el\_merge\_block\_with\_above(el\_blockhead\_t \*lower);

void el\_free(void \*ptr);

**3.9 Overall Code Structure of EL Malloc**

Below is the code structure of the EL Malloc library. Some of the functions have been implemented already while those marked REQUIRED must be completed for full credit on the problem.

////////////////////////////////////////////////////////////////////////////////

// Global control functions

int el\_init(int max\_bytes);

// Create an initial block of memory for the heap using

// malloc(). Initialize the el\_ctl data structure to point at this

// block. Initialize the lists in el\_ctl to contain a single large

// block of available memory and no used blocks of memory.

void el\_cleanup();

// Clean up the heap area associated with the system which simply

// calls free() on the malloc'd block used as the heap.

////////////////////////////////////////////////////////////////////////////////

// Pointer arithmetic functions to access adjacent headers/footers

el\_blockfoot\_t \*el\_get\_footer(el\_blockhead\_t \*head);

// Compute the address of the foot for the given head which is at a

// higher address than the head.

el\_blockhead\_t \*el\_get\_header(el\_blockfoot\_t \*foot);

// REQUIRED

// Compute the address of the head for the given foot which is at a

// lower address than the foot.

el\_blockhead\_t \*el\_block\_above(el\_blockhead\_t \*block);

// Return a pointer to the block that is one block higher in memory

// from the given block. This should be the size of the block plus

// the EL\_BLOCK\_OVERHEAD which is the space occupied by the header and

// footer. DOES NOT follow next pointer, looks in adjacent memory.

el\_blockhead\_t \*el\_block\_below(el\_blockhead\_t \*block);

// REQUIRED

// Return a pointer to the block that is one block lower in memory

// from the given block. Uses the size of the preceding block found

// in its foot. DOES NOT follow next pointer, looks in adjacent

// memory.

////////////////////////////////////////////////////////////////////////////////

// Block list operations

void el\_print\_blocklist(el\_blocklist\_t \*list);

// Print an entire blocklist. The format appears as follows.

//

// blocklist{length: 5 bytes: 566}

// [ 0] head @ 618 {state: u size: 200} foot @ 850 {size: 200}

// [ 1] head @ 256 {state: u size: 32} foot @ 320 {size: 32}

// [ 2] head @ 514 {state: u size: 64} foot @ 610 {size: 64}

// [ 3] head @ 452 {state: u size: 22} foot @ 506 {size: 22}

// [ 4] head @ 168 {state: u size: 48} foot @ 248 {size: 48}

// index offset a/u offset

//

// Note that the '@ offset' column is given from the starting heap

// address (el\_ctl->heap\_start) so it should be run-independent.

void el\_print\_stats();

// Print out basic heap statistics. This shows total heap info along

// with the Available and Used Lists. The output format resembles the following.

//

// HEAP STATS

// Heap bytes: 1024

// AVAILABLE LIST: blocklist{length: 3 bytes: 458}

// [ 0] head @ 858 {state: a size: 126} foot @ 1016 {size: 126}

// [ 1] head @ 328 {state: a size: 84} foot @ 444 {size: 84}

// [ 2] head @ 0 {state: a size: 128} foot @ 160 {size: 128}

// USED LIST: blocklist{length: 5 bytes: 566}

// [ 0] head @ 618 {state: u size: 200} foot @ 850 {size: 200}

// [ 1] head @ 256 {state: u size: 32} foot @ 320 {size: 32}

// [ 2] head @ 514 {state: u size: 64} foot @ 610 {size: 64}

// [ 3] head @ 452 {state: u size: 22} foot @ 506 {size: 22}

// [ 4] head @ 168 {state: u size: 48} foot @ 248 {size: 48}

void el\_init\_blocklist(el\_blocklist\_t \*list);

// Initialize the specified list to be empty. Sets the beg/end

// pointers to the actual space and initializes those data to be the

// ends of the list. Initializes length and size to 0.

void el\_add\_block\_front(el\_blocklist\_t \*list, el\_blockhead\_t \*block);

// REQUIRED

// Add to the front of list; links for block are adjusted as are links

// within list. Length is incremented and the bytes for the list are

// updated to include the new block's size and its overhead.

void el\_remove\_block(el\_blocklist\_t \*list, el\_blockhead\_t \*block);

// REQUIRED

// Unlink block from the list it is in which should be the list

// parameter. Updates the length and bytes for that list including

// the EL\_BLOCK\_OVERHEAD bytes associated with header/footer.

////////////////////////////////////////////////////////////////////////////////

// Allocation-related functions

el\_blockhead\_t \*el\_find\_first\_avail(size\_t size);

// REQUIRED

// Find the first block in the available list with block size of at

// least (size+EL\_BLOCK\_OVERHEAD). Overhead is accounted so this

// routine may be used to find an available block to split: splitting

// requires adding in a new header/footer. Returns a pointer to the

// found block or NULL if no of sufficient size is available.

el\_blockhead\_t \*el\_split\_block(el\_blockhead\_t \*block, size\_t new\_size);

// Set the pointed to block to the given size and add a footer to

// it. Creates another block above it by creating a new header and

// assigning it the remaining space. Ensures that the new block has a

// footer with the correct size. Returns a pointer to the newly

// created block while the parameter block has its size altered to

// parameter size. Does not do any linking of blocks. If the

// parameter block does not have sufficient size for a split (at least

// new\_size + EL\_BLOCK\_OVERHEAD for the new header/footer) makes no

// changes and returns NULL.

void \*el\_malloc(size\_t nbytes);

// REQUIRED Return pointer to a block of memory with at least the

// given size for use by the user. The pointer returned is to the

// usable space, not the block header. Makes use of find\_first\_avail()

// to find a suitable block and el\_split\_block() to split it. Returns

// NULL if no space is available.

////////////////////////////////////////////////////////////////////////////////

// De-allocation/free() related functions

void el\_merge\_block\_with\_above(el\_blockhead\_t \*lower);

// REQUIRED

// Attempt to merge the block lower with the next block in

// memory. Does nothing if lower is null or not EL\_AVAILABLE and does

// nothing if the next higher block is null (because lower is the last

// block) or not EL\_AVAILABLE. Otherwise, locates the next block with

// el\_block\_above() and merges these two into a single block. Adjusts

// the fields of lower to incorporate the size of higher block and the

// reclaimed overhead. Adjusts footer of higher to indicate the two

// blocks are merged. Removes both lower and higher from the

// available list and re-adds lower to the front of the available

// list.

void el\_free(void \*ptr);

// REQUIRED

// Free the block pointed to by the give ptr. The area immediately

// preceding the pointer should contain an el\_blockhead\_t with information

// on the block size. Attempts to merge the free'd block with adjacent

// blocks using el\_merge\_block\_with\_above().

**3.10 Demo Run using EL Malloc**

Below is a run showing the behavior of a series of el\_malloc() / el\_free() calls. They are performed in the provided el\_demo.c program.

**Source for el\_demo.c**

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include "el\_malloc.h"

void print\_ptr\_offset(char \*str, void \*ptr){

printf("%s: %lu from heap start\n",

str, PTR\_MINUS\_PTR(ptr,el\_ctl.heap\_start));

}

int main(){

printf("EL\_BLOCK\_OVERHEAD: %lu\n",EL\_BLOCK\_OVERHEAD);

el\_init(1024);

printf("INITIAL\n"); el\_print\_stats(); printf("\n");

void \*p1 = el\_malloc(128);

void \*p2 = el\_malloc(48);

void \*p3 = el\_malloc(156);

printf("MALLOC 3\n"); el\_print\_stats(); printf("\n");

printf("POINTERS\n");

print\_ptr\_offset("p3",p3);

print\_ptr\_offset("p2",p2);

print\_ptr\_offset("p1",p1);

printf("\n");

void \*p4 = el\_malloc(22);

void \*p5 = el\_malloc(64);

printf("MALLOC 5\n"); el\_print\_stats(); printf("\n");

printf("POINTERS\n");

print\_ptr\_offset("p5",p5);

print\_ptr\_offset("p4",p4);

print\_ptr\_offset("p3",p3);

print\_ptr\_offset("p2",p2);

print\_ptr\_offset("p1",p1);

printf("\n");

el\_free(p1);

printf("FREE 1\n"); el\_print\_stats(); printf("\n");

el\_free(p3);

printf("FREE 3\n"); el\_print\_stats(); printf("\n");

p3 = el\_malloc(32);

p1 = el\_malloc(200);

printf("RE-ALLOC 3,1\n"); el\_print\_stats(); printf("\n");

printf("POINTERS\n");

print\_ptr\_offset("p1",p1);

print\_ptr\_offset("p3",p3);

print\_ptr\_offset("p5",p5);

print\_ptr\_offset("p4",p4);

print\_ptr\_offset("p2",p2);

printf("\n");

el\_free(p1);

printf("FREE'D 1\n"); el\_print\_stats(); printf("\n");

el\_free(p2);

printf("FREE'D 2\n"); el\_print\_stats(); printf("\n");

el\_free(p3);

el\_free(p4);

el\_free(p5);

printf("FREE'D 3,4,5\n"); el\_print\_stats(); printf("\n");

el\_cleanup();

return 0;

}

**Output of Program**

make: 'el\_demo' is up to date.

EL\_BLOCK\_OVERHEAD: 40

INITIAL

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 1 bytes: 1024}

[ 0] head @ 0 {state: a size: 984} foot @ 1016 {size: 984}

USED LIST: blocklist{length: 0 bytes: 0}

MALLOC 3

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 1 bytes: 572}

[ 0] head @ 452 {state: a size: 532} foot @ 1016 {size: 532}

USED LIST: blocklist{length: 3 bytes: 452}

[ 0] head @ 256 {state: u size: 156} foot @ 444 {size: 156}

[ 1] head @ 168 {state: u size: 48} foot @ 248 {size: 48}

[ 2] head @ 0 {state: u size: 128} foot @ 160 {size: 128}

POINTERS

p3: 288 from heap start

p2: 200 from heap start

p1: 32 from heap start

MALLOC 5

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 1 bytes: 406}

[ 0] head @ 618 {state: a size: 366} foot @ 1016 {size: 366}

USED LIST: blocklist{length: 5 bytes: 618}

[ 0] head @ 514 {state: u size: 64} foot @ 610 {size: 64}

[ 1] head @ 452 {state: u size: 22} foot @ 506 {size: 22}

[ 2] head @ 256 {state: u size: 156} foot @ 444 {size: 156}

[ 3] head @ 168 {state: u size: 48} foot @ 248 {size: 48}

[ 4] head @ 0 {state: u size: 128} foot @ 160 {size: 128}

POINTERS

p5: 546 from heap start

p4: 484 from heap start

p3: 288 from heap start

p2: 200 from heap start

p1: 32 from heap start

FREE 1

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 2 bytes: 574}

[ 0] head @ 0 {state: a size: 128} foot @ 160 {size: 128}

[ 1] head @ 618 {state: a size: 366} foot @ 1016 {size: 366}

USED LIST: blocklist{length: 4 bytes: 450}

[ 0] head @ 514 {state: u size: 64} foot @ 610 {size: 64}

[ 1] head @ 452 {state: u size: 22} foot @ 506 {size: 22}

[ 2] head @ 256 {state: u size: 156} foot @ 444 {size: 156}

[ 3] head @ 168 {state: u size: 48} foot @ 248 {size: 48}

FREE 3

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 3 bytes: 770}

[ 0] head @ 256 {state: a size: 156} foot @ 444 {size: 156}

[ 1] head @ 0 {state: a size: 128} foot @ 160 {size: 128}

[ 2] head @ 618 {state: a size: 366} foot @ 1016 {size: 366}

USED LIST: blocklist{length: 3 bytes: 254}

[ 0] head @ 514 {state: u size: 64} foot @ 610 {size: 64}

[ 1] head @ 452 {state: u size: 22} foot @ 506 {size: 22}

[ 2] head @ 168 {state: u size: 48} foot @ 248 {size: 48}

RE-ALLOC 3,1

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 3 bytes: 458}

[ 0] head @ 858 {state: a size: 126} foot @ 1016 {size: 126}

[ 1] head @ 328 {state: a size: 84} foot @ 444 {size: 84}

[ 2] head @ 0 {state: a size: 128} foot @ 160 {size: 128}

USED LIST: blocklist{length: 5 bytes: 566}

[ 0] head @ 618 {state: u size: 200} foot @ 850 {size: 200}

[ 1] head @ 256 {state: u size: 32} foot @ 320 {size: 32}

[ 2] head @ 514 {state: u size: 64} foot @ 610 {size: 64}

[ 3] head @ 452 {state: u size: 22} foot @ 506 {size: 22}

[ 4] head @ 168 {state: u size: 48} foot @ 248 {size: 48}

POINTERS

p1: 650 from heap start

p3: 288 from heap start

p5: 546 from heap start

p4: 484 from heap start

p2: 200 from heap start

FREE'D 1

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 3 bytes: 698}

[ 0] head @ 618 {state: a size: 366} foot @ 1016 {size: 366}

[ 1] head @ 328 {state: a size: 84} foot @ 444 {size: 84}

[ 2] head @ 0 {state: a size: 128} foot @ 160 {size: 128}

USED LIST: blocklist{length: 4 bytes: 326}

[ 0] head @ 256 {state: u size: 32} foot @ 320 {size: 32}

[ 1] head @ 514 {state: u size: 64} foot @ 610 {size: 64}

[ 2] head @ 452 {state: u size: 22} foot @ 506 {size: 22}

[ 3] head @ 168 {state: u size: 48} foot @ 248 {size: 48}

FREE'D 2

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 3 bytes: 786}

[ 0] head @ 0 {state: a size: 216} foot @ 248 {size: 216}

[ 1] head @ 618 {state: a size: 366} foot @ 1016 {size: 366}

[ 2] head @ 328 {state: a size: 84} foot @ 444 {size: 84}

USED LIST: blocklist{length: 3 bytes: 238}

[ 0] head @ 256 {state: u size: 32} foot @ 320 {size: 32}

[ 1] head @ 514 {state: u size: 64} foot @ 610 {size: 64}

[ 2] head @ 452 {state: u size: 22} foot @ 506 {size: 22}

FREE'D 3,4,5

HEAP STATS

Heap bytes: 1024

AVAILABLE LIST: blocklist{length: 1 bytes: 1024}

[ 0] head @ 0 {state: a size: 984} foot @ 1016 {size: 984}

USED LIST: blocklist{length: 0 bytes: 0}

**3.11****Grading Criteria for El Malloc   GRADING**

Both binary and shell tests can be run with make test-p1

| **Weight** | **Criteria** |
| --- | --- |
|  | **Automated Tests** |
| 10 | Passing automated tests in test\_el\_malloc.sh |
| 10 | Lack of memory errors reported by Valgrind on test\_el\_malloc.sh |
|  | Points will be deducted from Valgrind scores if significant functions are not implemented |
|  | 2 point deduction if the test-p2 target is not added to the Makefile |
|  | **Manual Inspection** |
| 5 | el\_get\_header() and el\_block\_below() |
|  | Use of pointer provided macros for pointer arithmetic |
|  | Correct use of sizeof() operator to account for sizes |
|  | el\_block\_below() checks for beginning of heap and returns NULL |
| 5 | el\_add\_block\_front() and el\_remove\_block() |
|  | Sensible use of pointers prev/next to link/unlink nodes **efficiently; no looping used** |
|  | Correct updating of list length and bytes |
|  | Accounts for EL\_BLOCK\_OVERHEAD when updating bytes |
| 5 | el\_split\_block() |
|  | Use of el\_get\_foot() to obtain footers for updating size |
|  | Clear evidence of placing a new header and footer for new block |
|  | Accounting for overhead EL\_BLOCK\_OVERHEAD when calculating new size |
| 5 | el\_malloc() |
|  | Use of el\_find\_first\_avail() to locate a node to split |
|  | Use of el\_split\_block() to split block into two |
|  | Clear state change of split blocks to Used and Available |
|  | Clear movement of lower split blocks to front of Used List |
|  | Clear movement of upper split blocks to front of Available lists |
|  | Use of pointer arithmetic macros to computer user address |
| 5 | el\_merge\_block\_with\_above() |
|  | NULL checks for argument and block above which result in no changes |
|  | Clear checks of whether both blocks are EL\_AVAILABLE |
|  | Use of el\_block\_above() to find block above |
|  | Clear updates to size of lower block and higher foot |
|  | Movement of blocks out of available list and merged block to front |
| 5 | el\_free() |
|  | Error checking that block is EL\_USED |
|  | Movement of block from used to available list |
|  | Attempts to merge block with blocks above and below it |
| 5 | Clean, readable code |
|  | Good indentation |
|  | Good selection of variable names |
| 55 | Problem Total |

**4 Problem 2: Printing ELF Symbol Tables**

The [Executable and Linkable (ELF) File Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) is the Unix standard for binary files with runnable code in them. By default, any executable or .o file produced by Unix compilers such as GCC produce ELF files as evidenced by the file command.

> gcc -c code.c

> file code.o

code.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

> gcc program.c

> file a.out

a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0

This problem explores the file format of ELF in order to show any **symbol table** present. The symbol table contains information on publicly accessible items in the file such as functions and global data. The standard utility readelf shows human readable versions of ELF files and the -s option specifically prints out the symbol table section.

a5-code> readelf -s data/quote\_data.o

Symbol table '.symtab' contains 17 entries:

Num: Value Size Type Bind Vis Ndx Name

0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND

1: 0000000000000000 0 FILE LOCAL DEFAULT ABS quote\_data.c

2: 0000000000000000 0 SECTION LOCAL DEFAULT 1

3: 0000000000000000 0 SECTION LOCAL DEFAULT 3

4: 0000000000000000 0 SECTION LOCAL DEFAULT 4

5: 0000000000000000 0 SECTION LOCAL DEFAULT 5

6: 0000000000000000 0 SECTION LOCAL DEFAULT 6

7: 0000000000000000 0 SECTION LOCAL DEFAULT 9

8: 0000000000000000 0 SECTION LOCAL DEFAULT 10

9: 0000000000000000 0 NOTYPE LOCAL DEFAULT 5 .LC0

10: 0000000000000000 0 SECTION LOCAL DEFAULT 8

11: 0000000000000000 11 FUNC GLOBAL DEFAULT 1 max\_size

12: 0000000000000000 8 OBJECT GLOBAL DEFAULT 6 choices

13: 000000000000000b 60 FUNC GLOBAL DEFAULT 1 list\_get

14: 0000000000000047 30 FUNC GLOBAL DEFAULT 1 get\_it

15: 0000000000000010 16 OBJECT GLOBAL DEFAULT 6 choices\_actual

16: 0000000000000020 3960 OBJECT GLOBAL DEFAULT 6 nodes

This problem re-implements this functionality in the showsym program to instruct on some the format of the ELF file. It has the following output.

a5-code> gcc -o showsym showsym.c # compile showsym

a5-code> ./showsym data/quote\_data.o # run on provided data file

Symbol Table # output of program

- 4296 bytes offset from start of file # location and size of symbol table

- 408 bytes total size

- 24 bytes per entry

- 17 entries

[idx] TYPE SIZE NAME # symbol table entries

[ 0]: NOTYPE 0 <NONE>

[ 1]: FILE 0 quote\_data.c

[ 2]: SECTION 0 <NONE>

[ 3]: SECTION 0 <NONE>

[ 4]: SECTION 0 <NONE>

[ 5]: SECTION 0 <NONE>

[ 6]: SECTION 0 <NONE>

[ 7]: SECTION 0 <NONE>

[ 8]: SECTION 0 <NONE>

[ 9]: NOTYPE 0 .LC0

[ 10]: SECTION 0 <NONE>

[ 11]: FUNC 11 max\_size

[ 12]: OBJECT 8 choices

[ 13]: FUNC 60 list\_get

[ 14]: FUNC 30 get\_it

[ 15]: OBJECT 16 choices\_actual

[ 16]: OBJECT 3960 nodes

The output of showsym is a similar but abbreviated version of the readelf -s showing information on public symbols in the ELF file.

**4.1 ELF File References**

You may wish to do some reading on the structure of the ELF format as it will assist in coming to grips with this problem. As you encounter parts of the below walk-through of how to find and print the symbol table, refer to the following resources

* [Manual Pages](http://man7.org/linux/man-pages/man5/elf.5.html): The manual page on ELF (found online or by typing man elf in any terminal) gives *excellent* coverage of the structs and values in the file format. Essentially the entire format is covered here though there may be a few ambiguities.
* [Wikipedia](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format): A good overview of the file format and has some extensive tables on the structs/values that comprise it.
* [Oracle Docs](https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-43405.html#scrolltoc): A somewhat more detailed and hyper-linked version of the manual pages.

Note that we will use the **64-bit ELF format only** which means most of the C types for variables should mention ELF64\_xxx in them though use of 32-bit integers may still apply via uint32\_t.

**4.2 Overall Approach**

ELF files are divided into sections. Our main interest is to identify the Symbol Table section but this is done in several steps.

1. Parse the File Header to identify the positions of the Section Header Array and Section Header String Table
2. Search the Section Header Array and associated string table to find the section named .symtab which is the symbol table and .strtab which contains the string names of the symbol table. Note the position in the file of these two
3. Iterate through the Symbol Table section which is also an array of structs. Use the fields present there along with the associated string names in .strtab to print each symbol and some associated information.

Since this is a binary file with a considerable amount of jumping and casting to structs, it makes sense to use mmap() to map the entire file into virtual memory. **It is a requirement to use mmap() for this problem**. Refer to lecture notes, textbook, and lab materials for details on how to set up a memory map and clean it up once finished.

**4.3 ELF Header and Section Header Array**

The initial bytes in an ELF file always have a fixed structure which is the ELF64\_Ehdr type. Of primary interest are the following

* Identification bytes and types in the field e\_ident[]. These initial bytes identify the file as ELF format or NOT (will check for this).
* The Section Header Array byte position in the field e\_shoff. The Section Header Array is like a table of contents for a book giving positions of most other sections in the file. It usually occurs near the end of the file.
* The index of the Section Header String Table in field e\_shstrndx. This integer gives an index into the Section Header Array where a string table can be found containing the names of headers.

The following diagram shows the layout of these first few important parts of an ELF file.

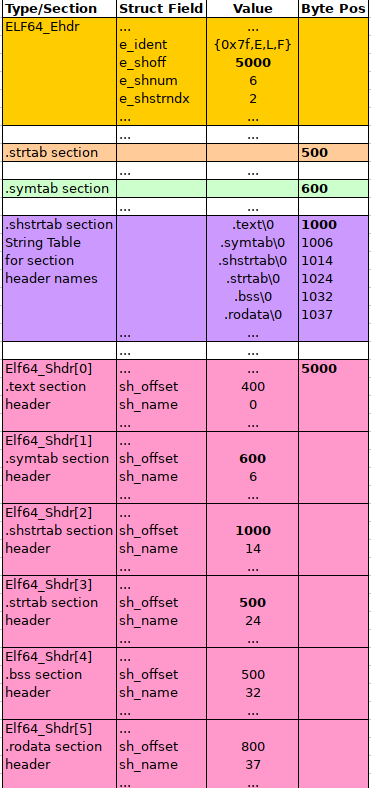


Figure 6: ELF File Header with Section Header Array and Section Header String Table.

**4.4 String Tables, Names, Section Headers**

To keep the sizes of structures fixed while still allowing variable length names for things, all the names that are stored in ELF files are in **string tables**. You can see one of these laid in the middle purple section of the diagram above starting at byte offset 1000. It is simply a sequence of multiple null-terminated strings laid out adjacent to one another in the file.

When a name is required, a field will give an offset into a specific string table. For example, each entry of the Section Header Array has an sh\_name field which is an offset into the .shstrtab (the sh is for "section header").. The offset indicates how far from the start of the string table to find the require name.

* The .shstrtab section begins at byte 1000 so all name positions are 1000 + sh\_name
* The 0th .text section has sh\_name = 0; the string .text\0 appears at position 1000.
* The 1th .symtab section has sh\_name = 6; the string .symtab\0 appears at byte position 1006.
* The 4th .bss section has sh\_name = 32; the string .bss\0 appears at byte position 1032.

The Section Header Array is an array of Elf64\_Shdr structs. By iterating over this array, fishing out the names from .shstrtab, and examining names using strcmp(), the positions for the two desired sections, .symtab and .strtab can be obtained via the associated sh\_offset field.

Note that one will need to also determine length of the Section Header Array from the ELF File Header field e\_shnum.

Also, on finding the Symbol Table section, not its size in bytes from the sh\_size field. This will allow you to determine the number of symbol table entries.

**4.5 Symbol Table and .strtab**

Similar to the Section Header Array, the Symbol Table is comprised of an array of Elf64\_Sym structs. Each of these structs has a st\_name field giving an offset into the .symtab section where a symbol's name resides. The following diagram shows this relationship.

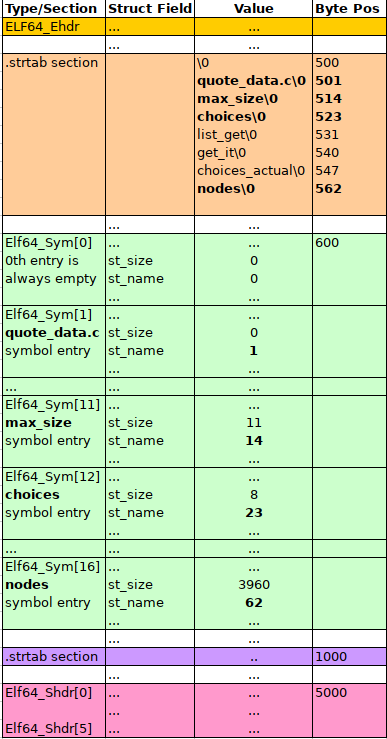


Figure 7: ELF Symbol Table and associated String Table

While iterating over the table, print the following information.

* The index starting at 0 (note that index 0 will always contain a blank entry)
* The type of symbol which can be determined using the methods below
* The size from the st\_size field. This corresponds to the number of bytes a variable will occupy or the number of bytes of instructions for a function.
* The name of the symbol or <NONE> if the symbol's name has length 0. Use the st\_name field which is an offset into the .strtab where the name of the symbol is located.

To determine the symbol's type, make use of the following code

unsigned char typec = ELF64\_ST\_TYPE(symtable\_entry[i].st\_info);

This macro extracts bits from the st\_info field and assigns them to typec which will be one of the following defined variables.

- STT\_NOTYPE : print type "NOTYPE"

- STT\_OBJECT : print type "OBJECT"

- STT\_FUNC : print type "FUNC"

- STT\_FILE : print type "FILE"

- STT\_SECTION : print type "SECTION"

An if/else or switch/case block to determine the type is best here.

**4.6 showsym Template**

A basic template for showsym.c is provided in the code pack which outlines the structure of the code along with some printing formats to make the output match examples. Follow this outline closely to make sure that your code complies with tests when the become available.

**4.7 Grading Criteria for El Malloc   GRADING**

Both binary and shell tests can be run with make test-p1

| **Weight** | **Criteria** |
| --- | --- |
|  | **Automated Tests** |
| 10 | Passing automated tests in test\_showsym.sh |
| 10 | Lack of memory errors reported by Valgrind on test\_showsym.sh |
|  | Points will be deducted from Valgrind scores if significant functions are not implemented |
|  | 2 point deduction if the test-p2 target is not added to the Makefile |
|  | **Manual Inspection** |
| 5 | Correctly sets up a memory map using open(), fstat(), mmap() |
|  | Correct unmap and close of file description at end of execution |
| 5 | Sets a pointer to the ELF File Header properly |
|  | Checks identifying bytes for sequence {0x7f,'E','L','F'} |
|  | Properly extracts the Section Header Array offset, length, string table index |
| 5 | Sets up pointers to Section Header Array and associate String Table |
|  | Loops over Section Header Array for sections named .symtab / .strtab |
|  | Properly uses SH String Table to look at names of each section while searching |
|  | Extracts offsets and sizes of .symtab / .strtab sections |
| 5 | Prints information on byte position of symbol table and its size |
|  | Sets up pointer to Symbol Table and associated String Table |
|  | Loops over entries in Symbol Table printing name, size, type |
|  | Uses ELF64\_ST\_TYPE() to extract symbol type from st\_info field |
| 5 | Clean, readable code |
|  | Good indentation |
|  | Good selection of variable names |
| 45 | Problem Total |

**4.8 Testing Notes**

**Testing Files**

| **File/Directory** | **Notes** |
| --- | --- |
| test\_showsym.sh | Testing script to run tests |
|  | Run single tests with ./test\_showsym.sh 5 to run test #5 |
| test\_showsym\_data.sh | Testing script data/definitions |
| test-input/ | Directory with required data files |
| test-data/ | Directory created by running the test script, it can be deleted safely |
|  |  |

**Early Exit Conditions and Messages**

The following two conditions were somewhat vague in the original specification but are checked in tests. Make sure to honor the behavior indicated below.

1. showsym should check the "magic bytes" (first elements of e\_ident[] in header) and if they are incorrect, print the message:
2. Magic bytes wrong, this is not an ELF file

and exit.

1. During the search for the symbol table, it is possible that it is not found. Such objects are usually executables that have been "stripped" of a symbol table. If it is detected that there is no symbol table, print the message
2. Couldn't find symbol table

and exit.

**Makefile Target**

Add the following target to the project Makefile.

test-p2: showsym

chmod u+x ./test\_showsym.sh

./test\_showsym.sh

The tests can then be run with the invocation

make test-p2

**5 Zip and Submit**

**5.1 Submit to Canvas**

Once your are confident your code is working, you are ready to submit. Ensure your folder has all of the required files. Create a zip archive of your lab folder and submit it to Canvas.

On Canvas:

* Click on the *Assignments* section
* Click on the appropriate link for this lab/assignment
* Scroll down to "Attach a File"
* Click "Browse My Computer"
* Select you Zip file and press OK

**5.2 Late Policies**

You may wish to review the policy on late project submission which will cost you late tokens to submit late or credit if you run out of tokens. **No projects will be accepted more than 48 hours after the deadline.**

<http://www-users.cs.umn.edu/~kauffman/2021/syllabus.html#late-projects>

**6****Test Release**

The tests for all problems are available for download and linked at the top of the spec.

**6.1 Testing Files**

| **File/Directory** | **Notes** |
| --- | --- |
| test-input/ | Problem 1 and 2: Directory with required data files |
| test-data/ | Directory created by running the test script, it can be deleted safely |
| test\_el\_malloc.sh | Problem 1: Testing script to run tests |
|  | Run single tests with ./test\_el\_malloc.sh 5 to run test #5 |
| test\_el\_malloc\_data.sh | Problem 1: Testing script data/definitions |
| test\_showsym.sh | Problem 2: Testing script to run tests |
|  | Run single tests with ./test\_showsym.sh 5 to run test #5 |
| test\_showsym\_data.sh | Problem 2: Testing script data/definitions |

**6.2 Problem 2 showsym Early Exit Specifications**

The following two conditions were somewhat vague in the original specification but are checked in tests. Make sure to honor the behavior indicated below.

1. showsym should check the "magic bytes" (first elements of e\_ident[] in header) and if they are incorrect, print the message:
2. Magic bytes wrong, this is not an ELF file

and exit.

1. During the search for the symbol table, it is possible that it is not found. Such objects are usually executables that have been "stripped" of a symbol table. An example of this is the file test-input/ls which is the standard directory listing program which is stripped on most systems.

If it is detected that there is no symbol table, print the message

Couldn't find symbol table

and exit.

**6.3 Makefile Targets**

Add the following targets to the project Makefile.

# TESTING TARGETS

test-p1: el\_malloc.o

chmod u+x ./test\_el\_malloc.sh

./test\_el\_malloc.sh

test-p2: showsym

chmod u+x ./test\_showsym.sh

./test\_showsym.sh

The tests can then be run with the invocations make test-p1 and test-p2 as shown below.

> make test-p1

gcc -Wall -g -Og -c el\_malloc.c

chmod u+x ./test\_el\_malloc.sh

./test\_el\_malloc.sh

Loading tests from test\_el\_malloc\_data.sh... 10 tests loaded

Running 10 tests

RUNNING NORMAL TESTS

TEST 1 single\_alloc : test-data/el\_malloc\_test\_01.c Compiling Running : OK

TEST 2 three\_allocs : test-data/el\_malloc\_test\_02.c Compiling Running : OK

TEST 3 reqd\_basics : test-data/el\_malloc\_test\_03.c Compiling Running : OK

TEST 4 alloc\_free : test-data/el\_malloc\_test\_04.c Compiling Running : OK

TEST 5 four\_alloc\_free\_1 : test-data/el\_malloc\_test\_05.c Compiling Running : OK

TEST 6 four\_alloc\_free\_2 : test-data/el\_malloc\_test\_06.c Compiling Running : OK

TEST 7 four\_alloc\_free\_3 : test-data/el\_malloc\_test\_07.c Compiling Running : OK

TEST 8 alloc\_fail : test-data/el\_malloc\_test\_08.c Compiling Running : OK

TEST 9 el\_demo : test-data/el\_malloc\_test\_09.c Compiling Running : OK

TEST 10 stress1 : test-data/el\_malloc\_test\_10.c Compiling Running : OK

Finished:

10 / 10 Normal correct

RUNNING VALGRIND TESTS

TEST 1 single\_alloc : test-data/el\_malloc\_test\_01.c Compiling Running : Valgrind OK

TEST 2 three\_allocs : test-data/el\_malloc\_test\_02.c Compiling Running : Valgrind OK

TEST 3 reqd\_basics : test-data/el\_malloc\_test\_03.c Compiling Running : Valgrind OK

TEST 4 alloc\_free : test-data/el\_malloc\_test\_04.c Compiling Running : Valgrind OK

TEST 5 four\_alloc\_free\_1 : test-data/el\_malloc\_test\_05.c Compiling Running : Valgrind OK

TEST 6 four\_alloc\_free\_2 : test-data/el\_malloc\_test\_06.c Compiling Running : Valgrind OK

TEST 7 four\_alloc\_free\_3 : test-data/el\_malloc\_test\_07.c Compiling Running : Valgrind OK

TEST 8 alloc\_fail : test-data/el\_malloc\_test\_08.c Compiling Running : Valgrind OK

TEST 9 el\_demo : test-data/el\_malloc\_test\_09.c Compiling Running : Valgrind OK

TEST 10 stress1 : test-data/el\_malloc\_test\_10.c Compiling Running : Valgrind OK

Finished:

10 / 10 Valgrind correct

=====================================

OVERALL:

10 / 10 Normal correct

10 / 10 Valgrind correct

> make test-p2

gcc -Wall -g -Og -c showsym.c

gcc -Wall -g -Og -o showsym showsym.o

chmod u+x ./test\_showsym.sh

./test\_showsym.sh

Loading tests from test\_showsym\_data.sh... 10 tests loaded

Running 10 tests

RUNNING NORMAL TESTS

TEST 1 x.c not elf : OK

TEST 2 small x.o : OK

TEST 3 coins\_funcs.o : OK

TEST 4 coins\_funcs\_asm.o : OK

TEST 5 coins\_main.o : OK

TEST 6 coins\_main : OK

TEST 7 cppvector : OK

TEST 8 quote\_main : OK

TEST 9 ls stripped : OK

TEST 10 warsim : OK

Finished:

10 / 10 Normal correct

RUNNING VALGRIND TESTS

TEST 1 x.c not elf : Valgrind OK

TEST 2 small x.o : Valgrind OK

TEST 3 coins\_funcs.o : Valgrind OK

TEST 4 coins\_funcs\_asm.o : Valgrind OK

TEST 5 coins\_main.o : Valgrind OK

TEST 6 coins\_main : Valgrind OK

TEST 7 cppvector : Valgrind OK

TEST 8 quote\_main : Valgrind OK

TEST 9 ls stripped : Valgrind OK

TEST 10 warsim : Valgrind OK

Finished:

10 / 10 Valgrind correct

=====================================

OVERALL:

10 / 10 Normal correct

10 / 10 Valgrind correct