Name: **SOLUTION** UID#: DirectoryID:

CMSC216: Practice Final Exam A SOLUTION

Fall 2025

University of Maryland

Exam period: 20 minutes *Points available:* 40

Problem 1 (10 pts): Pagebo Undary recently wrote a C program that is shown nearby and is startled to find that, despite his code clearly accessing out-of-bounds array indices, a Segmentation Fault does not occur unless the access is "way" out of bounds. Pagebo is confused by this apparent inconsistency but concludes that, so long as his code is only a "little" out of bounds, apparently nothing bad will happen.

```
#include <stdio.h>
int main(){
  int arr[5]={10,20,30,40,50};
  printf("arr[   10]: %d\n",arr[   10]);
  printf("arr[  100]: %d\n",arr[  100]);
  printf("arr[10000]: %d\n",arr[10000]);
  return 0;
}
// >> gcc array_bounds.c
// >> a.out
// arr[   10]: 378735946
// arr[  100]: 1892438979
// Segmentation fault (core dumped)
```

Use your knowledge of the **Virtual Memory System** to educate Pagebo on why some out-of-bounds accesses generate Segmentation faults while others do not. Indicate whether you agree with Pagebo's conclusion (going a little out of bounds is okay) or if there is more to it than this.

SOLUTION: Each time a program accesses a memory address, that address is translated from a Virtual Location to a Physical Location by the Operating System and MMU hardware. Translation is done in chunks called Pages with a Virtual Page mapping to a Physical Page in DRAM or on Disk in a swap space. Segmentation Faults are usually detected when a program attempts to access a Virtual Page that has not been mapped to any Physical Location. If an array happens to fall at the beginning of a valid Virtual Page, then there will be Physical memory that can be accessed after the end of it due to the "chunking" of Pages (typically 4096 byte chunks). However, programmers do not control the placement of variables on memory pages. If an array ends at Page Boundary, then the elements immediately after it may not have a Physical Location and accessing them will trigger a segmentation fault. It is ALWAYS an error to go out of bounds in C arrays as this will lead to unpredictable program behavior: crashes if lucky, corruption of nearby data in unlucky cases.

Problem 2 (10 pts): New programmers are often surprised to learn that once an array is allocated, its size cannot be extended. In C code, this is easily observable as calling `malloc(16)` will yield a block of 16 bytes but there are no simple calls to expand this block of memory and calls like `realloc()` indicate they may move data to another location to find enough space.

Consider a proposed function for EL Malloc called `int el_expand_block(el_blockhead_t *bock)` which would expand a given block.

(A) What conditions need to occur for the function to succeed?

(B) Why is it impossible to expand a block in some cases?

SOLUTION: Expanding an in-use block (one previously granted by a call to `el_malloc() / malloc()`*) could occur if the block "above" (next highest in memory address) was still available. In that case, the block above could be shrunk to a smaller size allowing the existing in-use block to expand. This would be prevented by either the expansion size being larger than the size of the block above OR the block above not being available. Since in-use blocks cannot be moved around without invalidating pointers to them, there would be no way to expand an existing block in those cases and it is likely to arise as heap blocks are usually packed tightly together. The realloc() function attempts to do just this: on reallocating to a larger size, it will expand the existing block if it is possible but move it if it is not possible to expand to the requested size; it is worth knowing about).*

Problem 3 (20 pts): Below are two functions that augment El Malloc with block shrinking. This allows a user to specify that the originally requested size for a memory area can be adjusted down potentially creating open space. Fill in the definitions for these functions.

```
#include "el_malloc.c"
////////////////////////////// SOLUTION //////////////////////////////
el_blockhead_t *el_shrink_block(el_blockhead_t *head, size_t newsize){
// Shrinks the size of the given block potentially creating a new block.  Computes remaining space
// as the difference between the current size and parameter newsize. If this is smaller than
// EL_BLOCK_OVERHEAD, does nothing further and returns NULL. Otherwise, reduces the size of the
// given block by adjusting its header and footer and establishes a new block above it with
// remaining space beyond the block overhead. Returns a pointer to the newly introduced blocks. Does
// not modify any links in lists.

  // NOTE: could simplify considerably using el_split_block()
  size_t remaining = head->size - newsize;
  if(remaining < EL_BLOCK_OVERHEAD){
    return NULL;
  }
  head->size = newsize;                       // adjust size
  el_blockfoot_t *foot = el_get_footer(head); // allows middle foot to be found
  foot->size = newsize;                       // set

  el_blockhead_t *above_head = el_block_above(head); // new header location
  above_head->size = remaining - EL_BLOCK_OVERHEAD;  // set its size
  el_blockfoot_t *above_foot = el_get_footer(above_head); // should be old foot
  above_foot->size = remaining - EL_BLOCK_OVERHEAD;       // set size
  return above_head;
}

int el_shrink(void *ptr, size_t newsize){
// Shrink the area associated with the given ptr if possible.  Checks to ensure that the block
// associated with the given user ptr is EL_USED and exits if not.  Uses el_shrink_block() to
// adjust the block size and create a block for the remaining space.  If not possible to shrink,
// returns 0.  Otherwise moves the current block to the front of the Used List and places the newly
// created block to the front of the Available List after setting its state to EL_AVAILABLE. Returns
// 1 on successfully shrinking.

  el_blockhead_t *head = PTR_MINUS_BYTES(ptr,sizeof(el_blockhead_t));
  if(head->state != EL_USED){              // error check
    printf("Does not appear to be a used block\n");
    exit(1);
  }
  el_blockhead_t *above = el_shrink_block(head, newsize);
  if(above == NULL){
    return 0;                              // could not shrink
  }
  above->state = EL_AVAILABLE;             // now available for use
  el_remove_block(el_ctl->used, head);     // out of used list
  el_add_block_front(el_ctl->used,  head); // to front of used list
  el_add_block_front(el_ctl->avail, above); // to front of available list
  return 1;                                // could shrink
  // likely want to attempt merging above with block above to limit fragmentation
  // in a full implementation of shrinking
}
```