Name: UID#: DirectoryID:

CMSC216: Practice Final Exam A

Fall 2024

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.

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. 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 to for the function to succeed?

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

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.

```
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.









}
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.









}
```