**Full Name:**______________________________________

**Univ ID:**______________________________________

# Mock Exam (loosely based on the final of Fall 2018)

**Instructions:**

- This exam takes 110 minutes. Read through all the problems and complete the easy ones first.
- This exam is CLOSED BOOK. You may use a A4-sized cheat sheet that you've prepared yourself. All other paper materials, phones, laptops, and calculators are forbidden.

| **1 (30)** | **2 (22)** | **3 (18)** | **4 (30)** | **Bonus (15)** | **Total (100+15)** |
|---|---|---|---|---|---|
| | | | | | |

This exam assumes 64-bit Intel x86 hardware on which all C programs are compiled.

## 1 Multiple choice questions (30 points):

Answer the following multiple-choice questions. Circle **all** answers that apply. Each problem is worth 5 points.

**A.** Suppose register `%rax` corresponds to some C variable `x`. What are `x`'s potential types?

1. `int`
2. `long`
3. `unsigned int`
4. `unsigned long`
5. `int *`
6. `long *`
7. None of the above.

**B.** Given `char x = -20`, what is `x`'s binary representation?

1. $(1001\ 0100)_2$
2. $(1110\ 1011)_2$
3. $(1110\ 1100)_2$
4. $(0001\ 0100)_2$
5. None of the above

**C.** Suppose `x` is of the type `node_t*`, where `node_t` is defined below. If `x` is `0x000000007fff0000`, what are the values of expression `(char *)(x+sizeof(node_t))` and `(char *)x + sizeof(node_t)`, respectively?

```
typedef struct {
   int size;
   int status;
}node_t;
```

1. `0x000000007fff0008 0x000000007fff0008`
2. `0x000000007fff0040 0x000000007fff0008`
3. `0x000000007fff0040 0x000000007fff0040`
4. `0x000000007fff0080 0x000000007fff0008`
5. `0x000000007fff0008 0x000000007fff0080`
6. `0x000000007fff0080 0x000000007fff0080`
7. None of the above

**D.** Consider the following code snippet:

```
char* words[6] = {"we", "are", "wishing", "for", "white", "xmas"};
char** p = words;
p = p + 2;
```

After executing the above code, what is the value of `p[1][1]`?

1. `'i'`
2. `'o'`
3. `'h'`
4. `'m'`
5. `'f'`
6. Undefined
7. None of the above

**E.** Which of the following are valid x86-64 instructions?

1. `movb $0xF, (%eax)`
2. `movq %eax, (%rsp)`
3. `movl %eax, (%rbx)`
4. `movq %rax, $0x123`
5. `movq (%rax, %rbx, 8), %rbx`
6. None of the above

**F.** Which of the following statements are true?

1. The addresses printed out by gdb debugger are physical addresses.
2. The addresses printed out by gdb debugger are virtual addresses.
3. The addresses stored in a Page Table Entry(PTE) are physical addresses.
4. The addresses stored in a Page Table Entry(PTE) are virtual addresses.
5. A forked child process uses the same page table as the parent process.
6. A newly created thread uses the same page table as the main thread.
7. None of the above.

## 2 Cache (22 points):

Ben Bitdiddle bought a 64 bit single-core x86 machine. The technical specification of his machine says that the CPU has a 1KB 2-way cache and that each cache line is 64 bytes.

**A.** (4 points) How many sets does the cache have?

**B.** Assume a cache line is indexed and tagged by the physical address. Consider the rightmost bit position to be 0.

- (4 points) What are the bit positions in the address used to represent the offset in the cache line?

- (4 points) What are the bit positions in the address used to index the cache sets?

**C.** You are given the following type and variable definitions:

```
typedef struct {
  long id;
  long score;
} node;

#define NUM 128
node n[NUM];
```

**1.** (2 points) What is the size of the `node` type in bytes?

**2.** (4 points) Assume the address of `n[0]` is 64-byte aligned (meaning the address is a multiple of 64) and there is no cached data in the beginning. How many cache misses does the following code snippet incur during its execution?

```
for(int i = 0; i < NUM; i++) {
  n[i].id = i;
  n[i].score = i;
}
```

**3.** (4 points) Assume the address of `n[0]` is 64-byte aligned and there is no cached data in the beginning. How many cache misses does the following code snippet incur during its execution? You should assume the least-recently-used cache replacement policy which replaces the oldest cacheline in a set.

```
for(int i = 0; i < NUM; i++) {
  n[i].id = i;
}
for(int i = 0; i < NUM; i++) {
  n[i].score = 0;
}
```

## 3 Virtual Memory (18 points):

Given a 16 bit machine (both virtual address and physical address are 16 bit), it uses 2 level page table for address translation and the page size is 64 bytes.

**A.** For each page in the page table, how many entries does it have? (Each entry is the physical address of the page in next level). (2 points)

**B.** How to divide the address to index the entry of the page in different page levels? (4 points)

**C.** Give the virtual address 0x2017, what is its virtual page number (VPN) ? Explain how to get its physical address with the 2 level page table? (8 points)

**D.** In the absence of TLB (and an empty CPU cache), to access some 8-byte data in memory:

1). how many memory accesses is required to perform address translation?

2). How many total memory misses is incurred to access the data (including address translation)? (4 points)

## 4 Malloc (30 points):

Ben Bitdiddle uses an explicit list design for malloc). In his design, each allocated chunk has a 16-byte regular header consisting of the `size` field indicating the chunk size[1], and the `allocated` field indicating whether the chunk has been allocated or not. Additionally, for each free chunk, the regular header is followed by the `prev` and `next` fields which are used to chain all free chunks together into a doubly-linked list.

Ben's design performs block splitting but not coalescing.

The code below (continuing on the next page) shows Ben's malloc implementation. The code returns 16-byte aligned payload pointers. The code has omitted the implementation of the following two helper functions:

1. `first_fit(&freelist, chunk_sz)` finds a block with size $\geq$ `chunk_sz` in the freelist, detaches the block from the freelist, and returns the found block (or NULL).
2. `get_block_from_os(chunk_sz)` grows the heap by `chunk_sz` bytes, initializes the new space as a free block and returns it.

```
// Ben's implementation of an explicit list-based malloc library for Lab 4
typedef struct {
   long allocated; //0: not allocated, non-zero: allocated
   long size; // chunk size including the header
}header_t;

typedef struct free_header {
   long allocated;
   long size;
   struct free_header *prev;
   struct free_header *next;
}free_header_t;

free_header_t* freelist;

//insert a node in the beginning of a doubly linked list
//Head pointer pointed to by headp is changed to the new head
void list_insert(free_header_t** headp, free_header_t* n) {
   n->next = *headp;
   n->prev = NULL;
   if (*headp) {
      (*headp)->prev = n;
   }
   *headp = n;
}
```

[1]chunk size includes the space taken up by the chunk header

```
#define MIN_CHUNK_SZ sizeof(free_header_t)

//split a chunk into two, one is of size csz, the other containing the
// remaining bytes. The remainder chunk is inserted into the freelist.
void split_and_insert_remainder(header_t* h, size_t csz)
{
   if ((h->size - csz) < MIN_CHUNK_SZ)
      return;

   header_t* remainder = ____________________________________________;
   remainder->allocated = 0;
   remainder->size = h->size - csz;
   list_insert(&freelist, remainder);
   h->size = csz;
}

void mm_init() {
   freelist = NULL;
}

void* mm_malloc(size_t size) {
   size_t chunk_sz = sizeof(header) + align(size);
   header_t* h = first_fit(&freelist, chunk_sz);
   if (!h) {
      h = get_block_from_os(chunk_sz);
   }
   split_and_insert_remainder(h, chunk_sz);
   h->allocated = 1;
   return header_to_payload(h);
}

void mm_free(void* p) {
  header_t* h = payload_to_header(p);
  h->allocated = 0;
  list_insert(&freelist, (free_header_t *)h);
}

void* mm_realloc(void* p, size_t newsize) {
  header_t* h = payload_to_header(p);
  size_t  oldsize = h->size;
  void* newp = mm_malloc(newsize);
  memcpy(newp, p, oldsize);
  mm_free(p);
  return newp;
}
```

**A.** (4 points) If Ben's program is tested using a trace sequence consisting only of `mm_malloc` requests but not `mm_free` requests, what is the *worst case* memory utilization of Ben's program? (The memory utilization is calculated as the total amount of user requested space divided by the total heap size)

**B.** (6 points) Complete the `payload_to_header` and `header_to_payload` functions.

```
// Given h pointing to an (allocated) chunk, return
// the pointer pointing to the payload of the chunk.
void* header_to_payload(header_t *h)
{





}

// Given p pointing to a chunk's payload, return
// a pointer pointing to the chunk's header.
header_t* payload_to_header(void *p)
{





}
```

**C.** (4 points) Complete the one missing line in `split_and_insert_remainder` on the previous page.

**D.** (6 points) Immediately after initializing the heap with `mm_init`, Ben executes the following function:

```
void buggy_heap_test() {
1:
2:   long* p1 = mm_malloc(10*sizeof(long));
3:   mm_free(p1);
4:   long* p2 = mm_malloc(4*sizeof(long));
5:   for (int i = 0; i < 4; i++)
6:      p2[i] = i+1;
7:   mm_free(p1); // this is a double-free bug
8:   printf("p2 contains %ld %ld %ld %ld\n", p2[0], p2[1], p2[2], p2[3]);
}
```

In the table below, write down the number of all chunks in the heap and the number of chunks in the freelist *after* executing a given line (The first and second lines have been filled out as an example).

| | **# of all chunks** | **# of chunks in the freelist** |
|---|---|---|
| line 1 | 0 | 0 |
| line 2 | 1 | 0 |
| line 3 | | |
| line 4 | | |
| line 7 | | |

**E.** (5 points) What's the result of executing `buggy_heap_test`? (Hint: the double-free bug at line 7 frees `p1` for the second time.)

**F.** (5 points) In Ben's initial `mm_realloc` implementation on the previous page, he allocates a new chunk, copies the payload of the existing chunk to the new one, and then frees the existing chunk. Later, Ben changes his initial implementation to the following code, in which he first frees the existing chunk, then allocates a new chunk, and finally copies the payload of the existing chunk to the new chunk.

```
void* mm_realloc(void *p, size\_t newsize) {
  header_t *h = payload_to_header(p);
  size_t  oldsize = h->size;
  mm_free(p);
  void* newp = mm_malloc(newsize);
  memcpy(newp, p, oldsize);
  return newp;
}
```

Immediately after initializing the heap with `mm_init`, Ben executes the following function to test the new `mm_realloc` implementation. What's the result of the execution?

```
void realloc_test() {
     long* p=mm_malloc(2*sizeof(long));
     p[0] = 1;
     p[1] = 2;
     p = mm_realloc(p, 4*sizeof(long));
     p[2] = 3;
     p[3] = 4;
     printf("p contains %ld %ld %ld %ld\n", p[0], p[1], p[2], p[3]);
     mm_free(p);
}
```

## 5 Bonus question, continuing from the malloc question (15 points):

For the bonus question, you need Ben's implementation of `first_fit` which has been omitted earlier:

```
//Delete a node from the doubly linked list
void list_remove(free_header_t** headp, free_header_t* n)
{
1:   if (n->prev) { //n is not the first in the list
2:      n->prev->next = n->next;
3:   } else {
4:      *headp = n->next;
5:   }
6:   if (n->next)
7:      n->next->prev = n->prev;
}

void first_fit(free_header_t** headp, size_t csz)
{
   free_header_t* n = *headp;
   while (n) {
      if (n->size >= csz)
         break;
      n = n->next;
   }
   if (n)
      list_remove(headp, n);
   return n;
}
```

Alyssa P. Hacker wants to investigate whether overflowing a buffer on Ben's heap can lead to exploits as is the case with buffer overflow on the stack. She wrote the following test function:

```
//This function parses a sequence of long integers (in hex format) separated by
//a space character and stores the resulting longs in array p.  For example,
//if buf is "0x1 0x18", then the function would write p[0]=1 p[1]=24
void parse_hex_longs(char *buf, long* p) {
   //omitted...
}

#define BUFSZ 1000
void malicious_test() {
1:
2:  long* p1 = mm_malloc(2*sizeof(long));
3:  long* p2 = mm_malloc(2*sizeof(long));
4:  mm_free(p2);

    char buf[BUFSZ];
    fgets(stdin, buf, BUFSZ); //read at most 1000 bytes from terminal
    parse_hex_longs(buf, p1); //this function can overflow p1

    long* p3 = mm_malloc(2*sizeof(long));
}
```

**A.** (2 points) Assuming the heap has been initialized immediately prior to invoking `malicious_test`, what is the total number of chunks in the heap, and the number of chunks in the freelist after executing line 4?

**B.** (5 points) Alice inputs the string ”0x1 0x2 0x0 0x20 0x8 0x0” for running `malicious_test`. The program incurs a segmentation fault at line 2 in function `list_remove`. What is the type of memory access causing the segmentation fault (read vs. write vs. execute)? And what is the address of the memory access causing the segmentation fault?

**C.** (8 points) The segmentation fault in **B** gave Alyssa an idea for crafting an exploit that would hijack the program control flow. Specifically, Alyssa plans to overwrite the return address so that program would start executing instructions at address `0xdeadbeefdeadbeef` upon returning from `malicious_test`. Suppose the return address of `malicious_test` is stored at address `0x0000ffff01234568`. What input string should Alyssa give to `malicious_test` to perform her exploit?

— END OF EXAM —