Full Name:_____

Final Exam, Fall 2016 Date: Dec 21st, 2016

Instructions:

- This final exam takes 110 minutes. **Read through all the problems and complete the easy ones first**.
- This exam is OPEN BOOK. You may use any books or notes you like. However, the use of any electronic devices including laptops, ipads, phones etc. is forbidden.

1 (xx/25)	2 (xx/20)	3 (xx/30)	4 (xx/25)	Total (xx/100)

1 Multiple choice questions (25 points):

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

A. After executing the following code snippet, what are the values of y and z? (We assume `sizeof(int)` is 4 and `sizeof(long)` is 8).

```
long x = 0x1234;
int *y = (int *)x;
y++;
long *z = (long *)x;
z++;
```

1. `0x1235` and `0x1235`
2. `0x1238` and `0x123c`
3. `0x1238` and `0x1238`
4. `0x123c` and `0x123c`
5. None of the above

B. Consider the following code snippet, what is output (if any) of its execution?

```
char str[7] = {'a', 'b', 'c', 'd', 'e', 'f', '\0'};
str[3] = '\0';
printf("%s %s\n", str, str+4);
```

1. `abcd ef`
2. `abc ef`
3. `abcdef ef`
4. `abc def`
5. Segmentation fault
6. None of the above

C. The function `foo()` below prints out `0x40061f`.

```
void foo() {
   printf("%p\n", foo);
}
```

Which of the following statements are true?

1. The output `0x40061f` represents a physical memory address.
2. The output `0x40061f` represents a virtual memory address.
3. The memory location at `0x40061f` readable by the user program.
4. The memory location at `0x40061f` writable by the user program.
5. The memory location at `0x40061f` executable by the user program.

D. Consider the following buggy program that uses a pointer that has previously been malloc-ed and then freed.

```
long *p = (long *)malloc(8);
free(p);
int *r = (int *)malloc(4);
*p = 1;
```

Which are the *likely* outcomes when executing the line `*p = 1`?

1. It causes "segmentation fault".
2. It causes the program to fail to compile.
3. It succeeds but corrupts the malloc library's internal data structure.
4. It causes buffer overflow.
5. None of the above

E. Consider the following piece of code.

```
void foo(int *p) {
   int b;
   printf("%p %p\n", p, &b);
}
void main() {
   int a;
   foo(&a);
}
```

Which of the following are plausible outputs when running the program on your laptop?

1. `0x7fff9ceff148 0x7fff9ceff12c`
2. `0x7fff9ceff12c 0x7fff9ceff148`
3. `0x7fff9ceff148 0x7fff9ceff144`
4. `0x7fff9ceff144 0x7fff9ceff148`

2 C basics (20 points):

Ben Bitdiddle is writing a function to compare the value of two positive numerical strings without converting the strings to their corresponding floats.

We assume that Ben's program is always given strings in a "canonical form". A string in the canonical form always corresponds to a positive value; it does *not* have any leading nor trailing zeros and it always contains the radix point (i.e. the '.' character). For example, "123." "34.1", ".54" are in the canonical form, while "123", "34.10", "0.54" are not.

Note when answering the following questions, you may use C library functions such as `strchr`, `strlen`, `strcmp`, `strncmp` (refer to Appendix I for details on these functions)

(a) (10 points) Ben decides to first write a helper function to find the position of the radix point in the given string in the canonical form. Please fill in his helper function `FindRadixPoint`.

```
//return the position of the radix point found in string s.
//e.g. FindRadixPoint("123.01") should return 3
//FindRadixPoint("1234.") should return 4
int FindRadixPoint(char *s)
{









}
```

(b) (10 points) Next, Ben proceeds to implement a function to compare two numerical strings in the canonical form. Complete the function `CmpNumericalStrings` so that the tests in function `TestCompare` will pass. Hint: you should use `FindRadixPoint()`.

```
//Ben's simple tests
void TestCompare() {
   char s1[10] = ".34";
   char s2[10] = "34.";
   assert(CmpNumericalStrings(s1, s2) < 0);

   char s3[10] = "34.";
   assert(CmpNumericalStrings(s2, s3) == 0);
}

//compare the numerical values of strings s1 and s2, which are in the canonical form.
//if s1 is equal to s2, return 0. If s1 is less than s2, return a negative int.
//if s1 is greater than s2, return a positive int
int CmpNumericalStrings(char *s1, char *s2)
{









}
```

3 Virtual Memory and Memory Allocation (30 points)

Ben Bitdiddle thinks his implicit-list-based Malloc Lab incurs too much memory overhead for small allocations. His good friend Alyssa P. Hacker has come up with the following alternative design.

In Alyssa's design, all malloc requests with size ≤ 96 are to be allocated in a 1GB contiguous area starting at address `small_start`. Bigger requests are allocated elsewhere use Ben's old design.

Alyssa structures the 1GB area as a set of 16KB regions. Each region holds chunks of the same size belonging to one specific size class. There are 12 total size classes whose ranges are $(0, 8]$, $(8, 16]$, $(16, 24]$, ..., $(88, 96]$. For example, suppose the malloc request is for size $s = 45$. Since s belongs to the class $(40, 48]$, the allocation is done in a memory region where all chunks are 48 bytes in size. The resulting allocation consumes a single 48-byte chunk. The last 3 bytes $(48 - 45)$ of the chunk are not used.

Alyssa suggests that Ben use an array of one-byte entries to keep track of the size class and the "initialized" status for each memory region. A region is "initialized" if it has been set up to hold chunks of a specific size class. The most significant bit of the entry contains the "initialized" status. The rest 7 bits contain the region's size class.

Ben rolls up his sleeves and gets busy on implementing Alyssa's design.

```
#define GB (1<<30)
#define KB (1<<10)
void *small_start; //starting address of the 1GB area for small allocation
const size_t region_size = 16 * KB;
//pinfo contains one entry for each 16KB region in the GB area
L1: unsigned char pinfo[                          ];

void mm_init() {
   //The sbrk syscall asks the OS to map 1GB area in the process' address space and
   //return the starting address of the resulting area.  Unlike in your malloc lab,
   //the OS does not immediately allocate 1GB physical memory here.
   //Rather, as addresses in this 1GB area are de-referenced later, the OS handles the
   //corresponding page faults by allocating physical pages on demand (demand-paging).
   small_start = sbrk(GB);

   //initialize the entries for all regions in the 1GB area
L2:for (int i = 0; i <                 ; i++) {
      pinfo[i] = 0;
    }
}
```

(a) (5 points) The above skeleton code shows the global variables and `mm_init` of Ben's implementation. How many regions are there in the 1GB area to be used for small allocations? Please complete the code at location `L1` and `L2`.

(b) (5 points) Next, help Ben write the helper function `is_initialized`. The function returns 1 if the most significant bit of the entry is 1 and zero otherwise.

```
//return 1 if the most significant bit of info is 1, 0 otherwise.
int is_initialized(unsigned char info) {









}
```

(c) (5 points) Write the helper function `get_size` to return the size information in the least significant 7-bits of the entry.

```
//return size class encoded in the least significant 7 bits of info.
size_t get_size(unsigned char info) {









}
```

(d) (6 points) In Ben's implementation of `mm_allocate`, he first decides the size class of the request. Then, he scans all region entries to find a region with the matching size. Upon encountering an un-initialized region, he initializes the region to be of the requested size class. Lastly, a chunk is allocated from the region. Please complete location `L3` and `L4` in the skeleton code below.

The code skeleton below uses your helper functions `is_initialized` and `get_size`. We explain what functions `get_size_class` and `allocate_chunk` do, but have omitted their implementations for brevity.

```
void *mm_malloc(size_t s) {
   if (s > 96)
      return mm_malloc_implicit_list(s);

   // get_size_class returns the size class for s
   // e.g. if s = 45, its size class is 48. if s = 20, its size class is 24.
   s = get_size_class(s);

   for (int i = 0; i < ...; i++) {
      if (is_initialized(pinfo[i]) && get_size(pinfo[i])!=s)
         continue;

      if (!is_initialized(pinfo[i])) {
         //code for initializing the region is omitted
         ...
L3:      //set pinfo[i] to be "initialized" and of size class "s"
         pinfo[i] =


      }
L4:   //calculate the starting address of the region whose entry is pinfo[i]
      void *region_start =


      //allocate_chunk allocates a chunk from the region at addr region_start whose
      //size class is s. If there is no free chunk in the region, NULL is returned
      void *p = allocate_chunk(region_start, s);
      if (p) {
         assert(p>=region_start && p<(region_start+region_size));
         return p;
      }
   }
   return NULL;
}
```

(e) (4 points) To free a chunk given a pointer in `mm_free`, Ben must find out the size class and the starting address for the region containing the underlying chunk. Please complete the code at location `L5` and `L6` of the skeleton code.

```
void *mm_free(void *p) {
   if (p < small_start || p >= (small_start+GB)) {
      return mm_free_implicit_list(p);
   }

L5://calculate the start address of the region containing p
   void *region_start =



L6: //find the size class s of the chunk pointed to by p
   size_t s =



   // free the chunk pointed to by p in its corresponding region (at addr region_start)
   // whose size class is s.
   free_chunk(region_start, p, s)
}
```

(f) (5 points) Discuss how Ben can keep track of the allocation status of each chunk in a region with low overhead. You do **not** to write down the code, although you are welcome to do so (e.g. by completing `allocate_chunk` and `free_chunk`).

(g) (5 points) Suppose a user program performs 2^{20} malloc's, each of which is of size 7 bytes. What is the total amount of *physical memory* consumed by Ben's allocator using Alyssa's design (including the overhead of your scheme in question (e))?

Note that the physical memory consumed by Alyssa's design for the above example should be much smaller than that of the implicit list design.

4 Multi-process and Multi-threading (25 points)

Ben Bitdiddle is writing a program to construct a linked list of numbers by inserting nodes into the list one by one. After attending the lecture on multi-threaded programming, Ben decides to try to speed up his program by using two threads, each inserting half of the nodes. The code below is Ben's implementation.

```
#define TESTSZ 2

typedef struct node {
   int val;
   struct node *next;
}node_t;

node_t *head; //head of the linked list

void list_insert(node_t *n) {

   n->next = head;
   head = n;

}

void list_print() {

   node_t *p = head;
   while (p) {
      printf("%d ", p->val);
      p = p->next;
   }
   printf("\n");

}

void insert_half(void *p) {
   node_t *nodes = (node_t *)p;
   for (int i = 0; i < TESTSZ/2; i++)
      list_insert(&nodes[i]);
}

void parallel_insert(node_t *nodes) {

   pthread_t tid;
   pthread_create(&tid, NULL, insert_half, (void *)(nodes+TESTSZ/2));
   insert_half((void *)nodes);
   pthread_join(tid, NULL);
   print_list();

}

void main() {

   node_t nodes[TESTSZ];
   for (int i = 0; i < TESTSZ; i++) {
     nodes[i].val = i;
     nodes[i].next = NULL;
   }
   parallel_insert(nodes);

}
```

(a) (10 points) Ben expects the output of his multi-threaded program to print out either ”0 1” or ”1 0”. What are **all** the potential outputs of Ben’s program? For each output that differs from the expected, please explain why that is the case.

(b) (5 points) Please fix Ben’s multi-threaded implementation such that it always produces the expected output. (You can directly add/modify the code on page 11).

(c) (10 points) Ben decides to switch to using multiple processes instead of multiple threads. He changes function `parallel_insert` to the following new implementation while leaving the rest of the code unchanged. What are **all** the potential outputs of his program? For each output that differs from the expected, please explain why that is the case.

```
void parallel_insert(node_t *nodes) {
   pid_t pid = fork();
   if (pid != 0) {
     insert_half((void *)nodes);
   } else {
     insert_half((void *)(nodes+TESTSZ/2));
     exit(0);
   }
   waitpid(&pid, NULL, 0);
   print_list();
}
```

— END OF EXAM—

Appendix I: strlen, strcmp, strchr

```
NAME
       strlen - calculate the length of a string

SYNOPSIS
       #include <string.h>
       size_t strlen(const char *s);

DESCRIPTION
       The  strlen()  function calculates the length of the string s, excluding the terminating null
       byte ('\0').

RETURN VALUE
       The strlen() function returns the number of bytes in the string s.


--------------------------------
NAME
       strcmp, strncmp - compare two strings

SYNOPSIS
       #include <string.h>
       int strcmp(const char *s1, const char *s2);
       int strncmp(const char *s1, const char *s2, size_t n);

DESCRIPTION
       The  strcmp()  function compares the two strings s1 and s2.  It returns an integer less than,
       equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be
       greater than s2.

       The  strncmp() function is similar, except it compares the only first (at most) n bytes of s1
       and s2.

RETURN VALUE
       The strcmp() and strncmp() functions return an integer less than, equal to, or  greater  than
       zero  if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match,
       or be greater than s2.


------------------------------
NAME
       strchr - locate character in string

SYNOPSIS
       #include <string.h>
       char *strchr(const char *s, int c);

DESCRIPTION
       The  strchr()  function  returns  a pointer to the first occurrence of the character c in the
       string s.

RETURN VALUE
       The strchr() function returns a pointer to the matched character or NULL if the
       character  is not found.  The terminating null byte is considered part of the string, so that
       if c is specified as '\0', these functions return a pointer to the terminator.
```

1. Multiple choices
A: 2
B: 2
C: 2, 3, 5
D: 3
E: 1

2.
```
int FindRadixPoint(char *s)
{
   char *s1 = strchr(s, '.');
   return s1-s;
}

int ComNumericalStrings(char *s1, char *s2)
{
   int r1 = FindRadixPoint(s1);
   int r2 = FindRadixPoint(s2);
   if (r1 < r2)
      return -1;
   else if (r1 > r2)
      return 1;
   else
      return strcmp(s1,s2);

}
```

3
```
(a) L1: unsigned char pinfo[GB/(16*KB)];
    L2: for (int i = 0; i < GB/(16*KB); i++) {

(b) int is_initialized(unsigned char info) {
       return (info>>7);
    }

(c)    size_t get_size(unsigned char info) {
       return (size_t)(info & 0x7f);
    }

(d) L3: pinfo[i] = 0x80 | s;
    L4: region_start = small_start + page_size * i;

(e) L5: int i = ((p - small_start)/ page_size);
        void *region_start = small_start + i * page_size;

    L6:  s = get_size(pinfo[i]);
```

(f) Ben can designate a bitmap in the beginning of each region to keep track of the allocation status for each
The size of the bitmap large enough for the number of chunks in the region.