Full Name:_______________________________________

Final Exam, Fall 2015 Date: December 22nd, 2015

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/20)	2 (xx/15)	3 (xx/20)	4 (xx/20)	5 (xx/25)	Total (xx/100)

1 Multiple choice questions (25 points):

Circle *all* answers that apply. Each problem is worth 5 points.

A. Which of the following statements are true?

1. each process has its own address space.
2. each thread has its own address space.
3. A parent process can fork a child process who shares the parent's address space.
4. In the statement `int *p = &a;` variable p contains the physical address of variable a.
5. Each physical page can have at most one corresponding virtual page that maps to it.
6. The virtual memory functionality is accomplished by software alone.

B. Which of the following assembly snippets correspond to the function body of
`void foo(int *x) {(*x)++;}`?

1. ```
   addl $0x1, %rdi
   retq
   ```
2. ```
   addl $0x1, (%rdi)
   retq
   ```
3. ```
   addq, $4, (%rdi)
   movl (%rdi) %eax
   retq
   ```
4. None of the above

C. Consider two C files. File `f1.c` contains:

```
float n;
void inc();
void main() {
   inc();
   printf("n=%E\n", n); //%e prints out a float in the style d.dddEdd..
}
```

File `f2.c` contains:

```
int n = 0;
void inc() {
   n++;
}
```

What happens if one types "`gcc f1.c f2.c`" and then run the program?

1. There's a compile-time error that function `inc` has been defined twice in two different places.
2. There's a compile-time error that global variable `n` has been defined twice in two different places.
3. Running the program produces a segmentation fault.
4. The program outputs `n=1.000000E+00`
5. The program outputs some number other than `n=1.000000E+00`.
```

D. Consider the following code snippet,

```
int a[2][3] = {{1,2,3},{4,5,6}};
int *b = a[1];
int *c = b + 1;
(*c)++;
```

What are the values of `a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]`?

1. 1 2 3 4 5 6
2. 1 3 3 4 5 6
3. 1 2 4 4 5 6
4. 1 2 3 5 5 6
5. 1 2 3 4 6 6
6. None of the above

2 Basic C: Integer conversion (15 points)

(a) (5 points) What's the hex representation of decimal number 31?

(b) (10 points) Write a program that accepts an integer (in decimal) as a program argument and prints out its corresponding hex representation. You can use the C library's `atoi` function if needed (see Appendix II). You are *not* allowed to use the `%x` or `%X` formatting option of `printf`. You can assume that the program is given a non-negative integer.

```
//argv[1] contains the first argument of the program
int
main(int argc, char **argv)
{

}
```

3 Processes and Threads (20 points):

Ben Bitddle has written the following C program.

```
void *
print_number(void *arg)
{
   int *p = (int *)arg;
   printf("%d\n", *p);
}

void
main() {
   for (int i = 0; i < 5; i++) {

      if (fork() == 0) {

         print_number(&i);

      }

   }

   exit(0);

}
```

(a) (5 points) When Ben runs the above program, how many processes in total does it produce?

(b) (5 points) Please modify Ben's program so that it prints exactly 5 numbers (0-4) in parallel, with the output containing an arbitary ordering among the five numbers.

Ben has written an alternative program to print out five numbers in parallel by spawning multiple threads instead of multiple processes. Man pages on Pthread library functions `pthread_create` and `pthread_join` are included in Appendix I. Below is Ben's threaded implementation (where function `print_number` is the same as in Ben's old program):

```
void
main() {

   pthread_t th[5];

   for (int i = 0; i < 5; i++) {

      pthread_create(&th[i], NULL, print_number, &i);

   }

   exit(0);

}
```

(c) (5 points) When Ben runs this program, he notices that it often prints out fewer than 5 numbers. Please explain why *and* fix it so that the program always prints out exactly 5 numbers.

(d) (5 points) Ben had expected his program to print out the 5 numbers 0-4. However, often the program outputs certain numbers more than once. For example, here's the output of an errorneous run: "3 5 3 3 4" Please explain why this happens. Please also fix the program so that it correctly prints out the five numbers 0-4 in parallel.

4 Programming with threads (20 points)

Ben Bitdiddle has implemented a linked list where each list node stores an integer value. To make certain aspects of the coding easier, Ben's linked list always contains a "dummy" node at its head: the dummy node's "next" pointer points to the first "real" node. Ben's implementation is shown below.

```
typedef struct node {
   int v;
   struct node *next;
}node;

//the head node is a dummy node, the real first node is pointed to by head->next
node *head;

void
init() {
   head = malloc(sizeof(node));
   head->next = NULL;
}

void
insert(int v) {
   node *n = malloc(sizeof(node));
   n->v = v;
   n->next = head->next;
   head->next = n;
}
```

(a) (10 points) Assuming the linked list starts out empty. Can the contents of the linked list be anything other than 1, 2 or 2, 1 if thread T1 inserts integer 1 while thread T2 concurrently inserts integer 2? If so, list *all* errorneous outcomes *and* describe the races in terms of the concrete code interleavings that lead to each of them. To make your answer easy-to-read, you should label certain lines in the code and use those labels in your explanation.

This page is intentionally left blank in case more room is needed for Problem 4(a).

Ben decides to add locking to his linked list. To maximize performance, Ben associates each node in the linked list with its own mutex. His implementation is as follows:

```
typedef struct node {
   int v;
   struct node *next;
   pthread_mutex_t m;
}node;

//the head node is a dummy node, the real first node is pointed to by head->next
node *head;

void
init() {
   head = malloc(sizeof(node));
   head->next = NULL;
   pthread_mutex_init(&head->m, NULL);
}

void
insert(int v) {
   node *n = malloc(sizeof(node));
   pthread_mutex_init(&n->m, NULL);

   pthread_mutex_lock(&n->m);
   n->v = v;
   n->next = head->next;
   pthread_mutex_unlock(&n->m);

   pthread_mutex_lock(&head->m);
   head->next = n;
   pthread_mutex_unlock(&head->m);
}
```

(b) (10 points) Does the above code solve the races described in (a)? If so, please explain how. If not, help fix Ben's program. You fix must not add more locks than the ones Ben already added.

5 Memory Allocation (25 points):

Ben Bitdiddle has written a simple memory allocator based on the textbook's implicit list implementation. Ben has also implemented a simple test program to try out his malloc. His test program (`test.cc`) is as follows. Note that function `mm_malloc` is defined in Ben's malloc source file `mm.c`.

```
void main() {
    int *m1;
    int *m2;

L1: m1= (int *)mm_malloc(4);
    for (int i = 0; i < 4; i++) {
L2:    m1[i] = i;
    }
L3: m2 = (int *)mm_malloc(16);
    for (int i = 0; i < 16; i++) {
L4:    m2[i] = i;
    }
}
```

(a) (5 points) Ben runs his test program. To his great dismay, the test program encounters a segmentation fault. Based on your knowledge of the implicit list implementation, where do you think the illegal memory access corresponding to the segmentation fault is at? (Cicle one).

1. In `mm.c`, function `mm_malloc`.
2. In `test.c`, line `L1`.
3. In `test.c`, line `L2`.
4. In `test.c`, line `L3`.
5. In `test.c`, line `L4`.

(b) (5 points) What is the *root cause* of the tester program's segmentation fault? Please also help Ben fix the segmentation fault.

Ben is annoyed to realize that the segmentation fault does not happen where the bug has occurred. He is determined to build a custom malloc library to help prorammmers catch memory bugs as soon as they occur. Since this malloc library is for debugging only, Ben will not worry about its throughput nor memory utilization.

Ben's inspiration comes from the virtual memory mechanism. To catch programmers accidentally running off the end of the buffer, Ben intends to use a *guard page* at the end of each allocated memory block. For example, suppose one uses Ben's library to malloc 10 bytes, i.e. `char *p = (char *)malloc(10);`. Upon successful return, virtual addresses p to $p + 9$ are accessible while addresses $p + 10$ to $p + 10 + PAGESIZE$ are in the guard region whose corresponding page table entry has a `null` mapping. As a result, if the user attempts to read or write at address $p + 10$, his/her program would incur a segmentation fault immediately.

(c) (5 points). The default page size on x86 is 4KB. Suppose the user uses Ben's library to malloc 10 bytes, i.e. `char *p = (char *)malloc(10);`. Please draw a picture of the allocated block including its guard region. In your picture, please indicate the sizes (in bytes) of the accessible and guard region, respectively. Please also indicate the location pointed to by `p`. *Please ignore block headers, as Ben's library stores header information separately from block data.* (Hint: Ben's malloc library has to do allocation on the granularity of pages as the guard region must start at the page boundary.)

(d) (5 points). The previous picture you draw corresponds to allocation in the virtual address space. What is the corresponding number of bytes consumed in the physical memory?

Below is a skeleton implementation of the allocation function for Ben's debugging malloc library.

```
void *
mmdebug_malloc(int size)
{
   int n_pages = size / PAGESIZE;
   if ((size % PAGESIZE) != 0) {
      n_pages++;
   }

   //ask for n_pages + 1 pages (+1 is for the guard page)
   //returned value "start" is a page-aligned address
   char *start;
   start = alloc_free_block(n_pages+1);

   //your code below
   //Note that the address given to mprotect must be aligned to a page boundary










}
```

(d) (5 points) Please help Ben complete `mmdebug_malloc` implementation shown above.

Function `alloc_free_block` (whose implementation is not shown) takes as parameter the total number of consecutive pages requested and returns the the address of the allocated block. You need not worry about the actual block allocation, block splits, keeping track of block sizes etc (we assume they've all been taken care of by `alloc_free_block`). However, you should fill in the code to make the guard page using the `mprotect` system call (see Appendix III) and return the appropriate address.

Appendix I: pthread_create and pthread_join

```
NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.

DESCRIPTION
       The  pthread_create()  function  starts  a new thread in the calling process.  The new thread
       starts execution by  invoking  start_routine();  arg  is  passed  as  the  sole  argument  of
       start_routine().

       The new thread terminates in one of the following ways:

       * It  calls  pthread_exit(3),  specifying  an  exit status value that is available to another
         thread in the same process that calls pthread_join(3).

       * It returns from start_routine().  This is equivalent to calling  pthread_exit(3)  with  the
         value supplied in the return statement.

       * Any  of the threads in the process calls exit(3), or the main thread performs a return from
         main().  This causes the termination of all threads in the process.

       The attr argument points to a pthread_attr_t structure whose contents are used at thread cre
       ation  time  to  determine attributes for the new thread; this structure is initialized using
       pthread_attr_init(3) and related functions.  If attr is NULL, then the thread is created with
       default attributes.

       Before  returning,  a  successful call to pthread_create() stores the ID of the new thread in
       the buffer pointed to by thread; this identifier is used to refer to the thread in subsequent
       calls to other pthreads functions.

RETURN VALUE
       On success, pthread_create() returns 0; on error, it returns an error number,  and  the  con
       tents of *thread are undefined.
```

NAME

pthread_join - join with a terminated thread

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

Compile and link with -pthread.

DESCRIPTION

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

If retval is not NULL, then pthread_join() copies the exit status of the target thread (i.e., the value that the target thread supplied to pthread_exit(3)) into the location pointed to by *retval. If the target thread was canceled, then PTHREAD_CANCELED is placed in *retval.

If multiple threads simultaneously try to join with the same thread, the results are unde fined. If the thread calling pthread_join() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

RETURN VALUE

On success, pthread_join() returns 0; on error, it returns an error number.

Appendix II: atoi

```
NAME
       atoi, atol, atoll, atoq - convert a string to an integer

SYNOPSIS
       #include <stdlib.h>

       int atoi(const char *nptr);
       long atol(const char *nptr);
       long long atoll(const char *nptr);
       long long atoq(const char *nptr);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       atoll():
           _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 600 || _ISOC99_SOURCE ||
           _POSIX_C_SOURCE >= 200112L;
           or cc -std=c99

DESCRIPTION
       The atoi() function converts the initial portion of the string pointed to  by  nptr  to  int.
       The behavior is the same as

           strtol(nptr, NULL, 10);

       except that atoi() does not detect errors.

       The atol() and atoll() functions behave the same as atoi(), except that they convert the ini
       tial portion of the string to their return type of long or long long.  atoq() is an  obsolete
       name for atoll().

RETURN VALUE
       The converted value.
```

Appendix III: mprotect

NAME

mprotect - set protection on a region of memory

SYNOPSIS

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

DESCRIPTION

mprotect() changes protection for the calling process's memory page(s) containing any part of the address range in the interval [addr, addr+len-1]. addr must be aligned to a page boundary.

If the calling process tries to access memory in a manner that violates the protection, then the kernel generates a SIGSEGV signal for the process.

prot is either PROT_NONE or a bitwise-or of the other values in the following list:

PROT_NONE The memory cannot be accessed at all.

PROT_READ The memory can be read.

PROT_WRITE The memory can be modified.

PROT_EXEC The memory can be executed.

RETURN VALUE

On success, mprotect() returns zero. On error, -1 is returned, and errno is set appropriately.

ERRORS

EACCES The memory cannot be given the specified access. This can happen, for example, if you mmap(2) a file to which you have read-only access, then ask mprotect() to mark it PROT_WRITE.

EINVAL addr is not a valid pointer, or not a multiple of the system page size.

ENOMEM Internal kernel structures could not be allocated.

ENOMEM Addresses in the range [addr, addr+len-1] are invalid for the address space of the process, or specify one or more pages that are not mapped. (Before kernel 2.4.19, the error EFAULT was incorrectly produced for these cases.)

6 Solution

```
1. Multiple choice questions
A. 1
B. 2
C. 5.
D. 5.

2.
(a) 0x1e

3.
(a) 32
(b) add exit(0); after print_number(\&i);
(c) The program exits before all threads have finished printing. add the following
block of thread before exit(0)
for (int i = 0;i < 5; i++) {
   pthread_join(th[i], NULL);
}
(d) This happens because while some thread is accessing stack variable i (via print_number(\&i))
the main thread is concurrently incrementing it (for...i++)
The fix can be
void
main() {
  int numbers = {1, 2, 3, 4, 5}
  for (int i = 0; i < 5; i++) {
    pthread_create(\&th[i], print_number, numbers[i], NULL)
  }
}

4.
(a) The list can contain just 1 or just 2.
Below is the interleaving that causes the list to contain only 1.
T1                                    T2
...                                   ...
n->next = head->next;
                              n->next = head->next
                              head->next = n;
head->next = n;
(b) No. The same interleaving can happen
one must lock head->m before reading the head variable (n->next = head->next)

5. (a) 1. Because L1 did not allocate enough space, L2 has overwritten block header and/or footer.
As a result, Ben's program is going to seg fault in mm.c when it tries to traverse the
implicit list of blocks to satisfy the malloc request at L3.
(b) Root cause is L1 and L2 did not allocate enough space.
L1: m1 = (int *)mm_malloc(4 *sizeof(int))
L3: m2 = (int *)mm_malloc(16 *sizeof(int))
(c)
|____________________|10-bytes|_______guard page____________|
x                    p       x+4K                         x+8K
(d) 4KB, because guard page does not consume physical memory. it just corresponds to
a null page table entry
(e)
mprotect(start + n_pages*PAGESIZE, PAGESIZE, PROT_NONE);
return start + n_pages*PAGESIZE - size;
```