Final Exam, Fall 2013  

Date: December 16th, 2013

Instructions:

- This final exam takes 1 hour and 50 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, laptop usage is limited to browsing the e-book only.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (xx/20)</td>
<td>2 (xx/20)</td>
<td>3 (xx/20)</td>
<td>4 (xx/20)</td>
<td>5 (xx/20)</td>
<td>Total (xx/100)</td>
</tr>
</tbody>
</table>
1 Multiple choice questions (20 points):

Answer the following multiple-choice questions. Circle all answers that apply. Each problem is worth 4 points. Each missing or wrong answer costs -2 point.

A. Which of the following C statement calculates the remainder of a divided by 32? Variable a is of int type.
   1. a % 32;
   2. a / 32;
   3. a >> 5;
   4. a << 5;
   5. a | ~0x1f;
   6. a & 0x1f;
   7. None of the above

B. Which of the following creates a 32-bit integer whose left-most i bits are 1 and the rest of the bits are zero?
   1. 1 << i
   2. (1<<i) - 1
   3. ((1<<i)-1) << (32 - i)
   4. 1<<(32-i)-1
   5. ~(1<<(32-i))-1
   6. 0xf8000000
   7. None of the above

C. What is the value of variable b after executing the following code snippet?

float a = -2.05;
int b;
b = *(int *)&a;

   1. -2
   2. -1
   3. 1
   4. 2
   5. 2.05
   6. None of the above
D. What is the value of \( p \) and \( q \) after executing the following code snippet?

```c
int *p;
char *q;
p = (int *)1;
p++;
q = (char *) (p+2)
```

1. \( p = 2, q = 3 \)
2. \( p = 5, q = 13 \)
3. \( p = 5, q = 7 \)
4. \( p = 2, q = 7 \)
5. \( p = 2, q = 13 \)
6. None of the above

E. What is the printf output in the following code snippet?

```c
void swap(int a, int b) {
    a = b;
    b = a;
}
void main() {
    int a = 1, b = 2;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
}
```

1. \( a = 1, b = 2 \)
2. \( a = 1, b = 1 \)
3. \( a = 2, b = 1 \)
4. \( a = 2, b = 2 \)
5. None of the above
2 Basic hardware knowledge (20 points):

Ben Bitdiddle is in charge of building a small-scale data center. He purchased a few hundred machines. Each machine is equipped with one quad-core 3.0GHz Intel CPU and one 1TB 7200 rpm Samsung hard drive. The hard drive is capable of doing 100MBytes/sec sequential transfer.

(a) (10 pts) Please sort the following latencies from the lowest to the highest.

1. Read a random 4-byte from a computer’s local hard disk.
2. CPU’s ALU operation, e.g. add two registers such as addl %eax, %ebx.
3. Read a random 4-byte from main memory. (The memory location is not cached.)
4. Read a 4-byte from L1 cache.
5. The roundtrip time between two computers in a data center. Hint: it’s approximately 100 microseconds.

Your sorted order is:

(b) (5 pts) Please give your estimate of the ratio of access time between 1) a random disk read and 2) CPU’s ALU operation, i.e. \( r = \frac{\text{time of random disk read}}{\text{time of ALU operation}} \).
Ben has written the following program to transform a file into all lower case letters.

```c
/* turn a buffer b of n characters into all lower case ones*/
void turn_into_lower(char *b, int n) {
    int i;
    for (i = 0; i < n; i++) {
        if (isupper(b[i])) {
            b[i] += ('a' - 'A');
        }
    }
}

int main(int argc, char **argv) {
    char buf[1<<24];
    int n_read, n_written;

    in_fd = open(argv[1], O_RDONLY);
    out_fd = open("output", O_CREAT|O_WRONLY);
    while ((n_read = read(in_fd, buf)) > 0) { //while not eof, keep reading
        turn_into_lower(buf, n_read);
        n_written = write(out_fd, buf, n_read);
        assert(n_written == n_read);
    }
}
```

Please estimate the running time of Ben’s program when doing `./a.out X` where X is a 500MByte ASCII file. Briefly explain your calculation.
3 Assembly (20 points):

Ben Bitdiddle is trying to figure out what the following x86-32 assembly does. Alyssa P. Hacker has given him a skeleton C function to describe what the assembly does.

<set_buf>:
    pushl  %ebp
    movl  %esp, %ebp
    subl  $16, %esp
    movl  $2, -4(%ebp)
    movl  $0, 0x804a040
    movl  $1, 0x804a044
    jmp  .L2
.L3:
    movl  -4(%ebp), %eax
    subl  $1, %eax
    movl  0x804a040(%eax,4), %edx
    movl  -4(%ebp), %eax
    subl  $2, %eax
    movl  0x804a040(%eax,4), %eax
    addl  %eax, %edx
    movl  -4(%ebp), %eax
    movl  %edx, 0x804a040(%eax,4)
    addl  $1, -4(%ebp)
.L2:
    movl  -4(%ebp), %eax
    cmpl  8(%ebp), %eax
    jl   .L3
    leave

<main>:
    pushl  %ebp
    movl  %esp, %ebp
    subl  $4, %esp
    movl  $10, (%esp)
call  set_buf
    leave
/* The C code skeleton */
int buf[100];

void set_buf(int n)
{
    int i;
    int i;
    buf[0] = 0;
    while (_____________) {

    }
}

void main()
{
    set_buf(10);
}

(a) (8 pts) In the function set_buf’s assembly code, what is being stored at which memory location? Please complete the following table.

<table>
<thead>
<tr>
<th>Memory location</th>
<th>Variable name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804a040</td>
<td></td>
</tr>
<tr>
<td>0x804a044</td>
<td></td>
</tr>
<tr>
<td>8(%ebp)</td>
<td></td>
</tr>
<tr>
<td>-4(%ebp)</td>
<td></td>
</tr>
</tbody>
</table>

(b) (12 pts) Please help Ben complete the C code equivalent of the assembly.
4 Fork and exec (20 points)

Ben Bitdiddle wrote the following C code to test the fork syscall.

```c
int main(int argc, char **argv) {
  int n = atoi(argv[1]);
  if (fork() == 0) {
    n = n - 2;
  } else {
    n = n - 1;
  }
  printf("n=%d\n", n);
  return 0;
}
```

(a) (10 pts) Ben compiles this program into binary executable `a.out` and executes `./a.out 4`. Please give the terminal output of the program.
(b) (10 pts) Alyssa P. Hacker decided to make Ben’s life harder. She added an additional code segment to Ben’s original program (after line 11 and before line 12). This is what the program is like now:

```c
int main(int argc, char **argv)
{
    int n = atoi(argv[1]);
    if (fork() == 0) {
        n = n - 2;
    } else {
        n = n - 1;
    }

    printf("n=%d\n", n);

    if (n <= 1) {
        exit(0);
    } else {
        char buf[100];
        sprintf(buf, "%s", n); /* turn n into an ASCII string */
        argv[1] = buf;
        execv(argv[0], argv); /* see man page for execv in appendix I */
    }
    return 0;
}
```

Now when Ben runs the program “./a.out 4”, what potential output does he see?
5 Heap, malloc, buffer overflow, oh my (20 points):

Ben Bitdiddle implemented a simple malloc library based on explicit-free list. His relevant code fragments are shown below. The actual code of `mm_malloc` is omitted because you do not need to know its details to complete the questions. However, if you are curious, `mm_malloc` is shown in Appendix II.

```c
#define TOTAL 500
typedef struct header {
   int used;
   int sz; /*size of the block's payload*/
   struct header *prev;
   struct header *next;
} hdr_t;

char heap[TOTAL];
hdr_t *root; /* root holds the head of the free list*/

void *mm_malloc(int s)
{
   /* Loop over every block in the free list starting from root,
   As soon as a block's payload is bigger than s, do following and return:
   1) remove the block from freelist
   2) allocate s bytes from the block's payload
   3) turn leftover payload into a new free block with proper header
   4) insert new free block to the front of the freelist
   */
}

void mm_free(void *p)
{
   hdr_t *h = (hdr_t *)((char *)p-sizeof(hdr_t));
   hdr_t *adj_h = (hdr_t *)(p + h->sz);
   if (!adj_h->used) { /* perform coalescing*/
      /* remove h's following adjacent block from freelist*/
      if (adj_h->prev) {
         L4: adj_h->prev->next = adj_h->next;
      } else {
         root = adj_h->next;
      }
      if (adj_h->next)
         adj_h->next->prev = adj_h->prev;
      /* combine h and its following adjacent block into one free block*/
      h->sz += adj_h->sz+sizeof(hdr_t);
   }
   h->used = 0;
   /* insert free block to head of the free list */
   h->next = root;
   h->prev = NULL;
   if (root)
      root->prev = h;
   root = h;
}
```
void mm_init()
{
    hdr_t *h = (hdr_t *)heap;
    h->sz = TOTAL-sizeof(hdr_t);
    h->used = 0;
    h->prev = NULL;
    h->next = NULL;
    root = h;
}

int main(int argc, char **argv)
{
    void *p1, *p2;
    mm_init();
    L1:
    p1 = mm_malloc(100);
    p2 = mm_malloc(200);
    mm_free(p1);
    L2:
    mm_free(p2);
    L3:
    return 0;
}

(a) (2 pts) What’s the block header size (in bytes) of Ben’s malloc library when run on a 32-bit x86 machine? For all following questions, we assume the malloc library executes on a 32-bit machine.

(b) (2 pts) At label L1 (i.e. right after executing mm_init), how many free blocks are there in the free list?

(c) (2 pts) At label L2 (i.e. right after the first mm_free), how many free blocks are there in the free list?

(d) (2 pts) At label L3 (i.e. after executing both mm_free), how many free blocks are there in the free list?
Alyssa P. Hacker is worried about the buffer overflow scenario for buffers allocated on the heap. She wrote the following code that uses Ben’s malloc library.

```c
void evil()
{
    printf("you are pwned\n");
}

void foo()
{
    char *p1, *p2;
    p1 = (char *)mm_malloc(4);
    p2 = (char *)mm_malloc(8);

    /*buffer overflow bug*/
    gets(p1);

    mm_free(p1);
}

void main()
{
    mm_init();
    foo();
}
```

(e) (4 pts) What will happen if the program is given an input string of 20 zero-valued bytes?
Alyssa decides to maliciously exploit the buffer overflow bug and hijack the control flow of the program to invoke function `evil()`. Alyssa knows that the address of `evil()` is 0x08abcdef and she has also figured out that the return address of `foo` is stored at stack address 0xbf12345c.

Alyssa’s evil plan is to trick the malloc library to perform coalescing with the adjacent block with a corrupted header. In particular, Alyssa plans to take advantage of the line of code at L4 to overwrite the return address of `foo` to be the address of the function `evil`. If she succeeds, when function `foo` returns, `evil()` will be executed.

Below is the input string that Alyssa has carefully crafted to achieve her goal. Each dotted box contains one byte in hex. As you can see, this input will corrupt the header of the next adjacent block. Specifically, Alyssa sets the `used` field to be zero and sets the `next` field to contain the address of `evil()`. Can you help Alyssa fill in the byte values marked by question marks? (i.e. what should she set the `prev` field to be?)

![Diagram of the header of the next adjacent block]

- `... 00 00 00 xx xx xx xx ?? ?? ?? ?? ef cd ab 08`
- 4 bytes
- Overwrite "used" field to zero
- Overwrite "sz" field with garbage
- Address of `evil()` in little Endian
(g) (4 pts) Alyssa thinks of a way to defend against buffer overflows on the heap. She wrote the following function to wrap around Ben’s `mm_malloc` function. Users of the malloc library are to use `alyssa_malloc` instead of `mm_malloc` to protect themselves against buffer overflow.

```c
void *
alyssa_malloc(int s)
{
    void *p;
    p = mm_alloc(p+PAGE_SIZE);
    /* see attachment for mprotect’s man page, ignore alignment issues*/
    mprotect(p+s,PAGE_SIZE, PROT_NONE);
    return p;
}
```

Can an overflowed buffer corrupt a block header now? If not, what will happen when a heap allocated buffer is overrun? Please explain.
Appendix I: execv

EXEC(3) Linux Programmer’s Manual

NAME
execv - execute a file

SYNOPSIS
#include <unistd.h>

int execv(const char*path, char *const argv[]);

DESCRIPTION
The execv() function replaces the current process image with a new
process image.

The initial argument for this function is the name of a file that is to
be executed.

The const char *argv can be thought of as arg0, arg1, ..., argn.
Together they describe a list of one or more pointers to null-terminated
strings that represent the argument list available to the
executed program. The first argument, by convention, should point to
the filename associated with the file being executed. The list of
arguments must be terminated by a NULL pointer, and, since these are
variadic functions, this pointer must be cast (char *) NULL.

RETURN VALUE
The exec() functions return only if an error has occurred. The return
value is -1, and errno is set to indicate the error.
Appendix II: mm_alloc

void *mm_malloc(int s)
{
    hdr_t *h = root;
    char *p;
    while (h) {
        int sz = h->sz;
        if (sz > s) {
            /* remove this block*/
            if (h->prev)
                h->prev->next = h->next;
            else
                root = h->next;
            if (h->next)
                h->next->prev = h->prev;
            /* allocate s bytes from the block’s payload */
            p = ((char *)h) + sizeof(hdr_t);
            h->sz = s;
            h->used = 1;
            /* turn leftover payload into a new free block*/
            h = (hdr_t *)(p + s);
            h->sz = sz - s - sizeof(hdr_t);
            assert(h->sz > 0);
            h->used = 0;
            /* insert new free block at root*/
            h->next = root;
            h->prev = NULL;
            if (root)
                root->prev = h;
            root = h;
            return (void *)p;
        } else {
            h = h->next;
        }
    }
}
Appendix III: mprotect

#include <sys/mman.h>
int mprotect(const void* addr, size_t len, int prot);

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.

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

P1: A: 1,6  B:3,5  C:6  D:5,13  E:1,1
P2: (a) 2 < 4 < 3 < 5 < 1
(b) A random disk read costs approx. 10ms. An ALU operation is approx. one CPU cycle, so on a 3GHz machine, a cycle costs $1/(3 \times 10^9)$. So the ratio of the two is approx. $3 \times 10^6$. (Note: full scores is given if one is within a factor of 30-50 to the above answer).

(c) The execution time is dominated by the amount of time to read 500MB and to write 500MB from the local disk. Since the sequential transfer speed of the local disk is 100MB/sec, the total time to read and to write is the file $5+5 = 10$ sec. The amount of time for CPU to read 500MB from memory and to do some ALU operations on them is negligible compared to 10 sec.

P3: (a) buf[0], buf[1], n, i
(b) void set_buf(int n) {
    int i;
    i = 2;
    buf[0]=0;
    buf[1]=1;
    while (i < n) {
        buf[i]=buf[i-1]+buf[i-2];
        i++;
    }
}

P4: (a) n=2 n=3 or n=3 n=2
    (b) One possible output is: n=2 n=3 n=2 n=1 n=0 n=1 n=1 n=0
P5: (a) 16
    (b) 1
    (c) 2
    (d) 2 Note Ben’s mm_free does not coalesce with the previous block
    (e) Since the block corresponding to p2 immediately follows p1, the block header of p2 will be completely overwritten with zeros due to the buffer overflow. Subsequently, mm_free will erroneously think p2 is free. In the attempt to remove p2 from the free list, the code will set root to be NULL (in the two lines after L4) because of p2’s zeroed-out next field, thus incorrectly dropping almost all the heap’s free memory.

    (f) L4: adj_h->prev->next = adj_h->next; Here the value of adj_h->prev is obtained from the ?? ?? ?? ?? ?? which is (erroneously) believed to point to the beginning of the block header. Note that the code tries to write the "next" field in the header (not the first field of the header) Since the "next" field is at offset 12 from the beginning, the actual address that will be overwritten with ox08abcdef is ?????????+12. Thus, ????????? should be 0xbf123450 in order for the overflow to cause the return address to be overwritten.

    (g) No. When a heap allocated buffer is overrun, the program touches the region of virtual memory for which any read/write access permission has been denied. This causes the h/w to generate an exception and OS will subsequently kill this process.