CS61, Fall 2012
Midterm Review Section
(10/16/2012)

Q1: Hexadecimal and Binary Notation

- Solve the following equations and put your answers in hex, decimal and binary.

				Hexadecimal	Decimal	Binary
15	+	0x15	=	0x24	36	0b100100
99	+	0xAA	=	0x10D	269	0b100001101
11101	+	0xF1	=	0x2c4E	11342	0b10110001001110
15	+	0b1010	=	0x19	25	0b11001

Q2: Address Spaces

- Given the following word sizes, find the largest possible representable memory address in both Little and Big Endian

	Largest Memory Address (Hexadecimal)	
Word Size	Little Endian	Big Endian
2 bytes	0xffff	0xffff
17 bits	0xffff1	0x1ffff
23 bits	0xffff7f	0x7fffff
11 bits	0xff7	0x7ff

Q3: Signed Arithmetic

- All numbers should fit into 6-bits and be expressed in 2's complement

			Binary	Decimal
010101 +	010101	=	0b101010	-22
100001 +	010101	=	0b110110	-10

101101 +	011111	=	0b001100	12
101110 +	100111	=	0b010101	21

Q4: Bitwise Operators

- Use the operators, &, |, ^, <<, >>, &&, || to...

Check if x is even:

```
bool is_odd = x  &  1
```

Check if *x* is a power of two:

```
bool is_pow2 =  0  ==   ( x   &   (x - 1) )
```

Get the absolute value of *x*:

```
int x;
int mask   =   x >> 31;
abs_x   =   (x  +  mask)  ^  mask;
```

Swap the values of *x* and *y*:

```
int x, y;
x  =   x   ^   y;
y  =   y   ^   x;
x  =   x  ^  y;
```

Q5: Struct Size

- Assume a 32-bit machine

```
struct my_struct
{
        char a,b;
        char *c;
        char d[6];
};

struct my_struct list[5];
struct my_struct *ptr = malloc(sizeof(struct my_struct));
```

5.1

What is sizeof(my_struct)?
16 bytes

5.2
How could we reduce this struct's size?

```
struct my_struct
{
        char d[6];
        char a,b;
        char *c;
};
```

Size 12 bytes

5.3
Suppose **list** lives at memory address 0x100. What is at memory address...
i) 0x111
list[1].b

ii) 0x123
padding

iii) 0x14c
list[4].d[4]

5.4
Suppose ptr = 0x1337. What is the address of...
i) ptr + 3
0x135b

ii) (int *) (((char *) ptr) + 5)
0x133c

Q6: Memory Bugs

- Find the problem in the following code that may yield unexpected results.

6.1

```
int* Hamlet() {
        int Ophelia = 42;
        return &Ophelia;
}

void main() {
```

```
        int *Horatio = Hamlet();
        printf("The answer is %d\n", *Horatio);
}
```

Ophelia is a local variable in the function Hamlet(). Hence, when a pointer to Ophelia is returned *after* Hamlet() is done executing, it is not a given that *Horatio = 42.

6.2

```
typedef struct {
        double Jaques;
        int Pages[10];
        char *Duke;
} ExiledCourt;

void main() {
        ExiledCourt *Rosalind;
        int counter = 0;
        Rosalind = (ExiledCourt*) malloc(sizeof(ExiledCourt));

        for (counter = 0; counter < 10;counter++) {
                Rosalind->Pages[counter] = 0x2a;
        }

        strcpy(Rosalind->Duke, "As You Like It");
}
```

strcpy is copying a string to Rosalind->Duke, which is a pointer. But Rosalind->Duke does not necessarily point to a valid memory address. If your C compiler clears memory before you use it, then Rosalind->Duke = 0.

6.3 - Gangnam Style

Somebody stole Psy's Gangnam style! Can you help him track down the thief? To find the culprit, Psy needs a piece of software that will copy a CIA list of known criminals (in the form of a singly-linked list of person structs) to his PDA and search the list for suspects living in Seoul. Unfortunately, programmer who developed this software never took CS61 so he left lots of bugs. Find them all to help Psy finally recover his GANGNAM STYLE!

```
1       typdef struct person_struct {
2               char name[20];
3               int age;
4               char *city;
5               struct person_struct *next;
6       } person;
7
8       person *search_suspects (person* first_person, int n) {
9               //Allocate new memory, copy list of people.10
11              person *new_first = malloc(n*sizeof(person));
12              person *new_current_person = NULL;
13              person *current_person = first_person;
```

```
14              int i = 0;
15              for (i = 0; i <= n; i++) {
16                      new_current_person = new_first + i*sizeof(person);
17                      strcpy(new_current_person->name, current_person->name);
18                      new_current_person->age = current_person->age;
19                      new_current_person->city = malloc(strlen(current_person->city));
20                      strcpy(*new_current_person->city, *current_person->city);
21                      if(current_person->next == NULL) {
22                              new_current_person->next = NULL;
23                      } else {
24                              new_current_person->next =new_current_person+sizeof(person);
25                              current_person = current_person->next;
26                      }
27              }
28
29              //Search new list for people in Los Angeles.
30
31              person *last_person = NULL;
32              new_current_person = new_first;
33              do {
34                      if(strcmp(new_current_person->city, "Los Angeles") == 0) {
35                              strcat(new_current_person->name, " (POSSIBLE SUSPECT!)");
36                              if(last_person == NULL) {
37                                      //This is the new head of the linked list
38                                      new_first = new_current_person;
39                              } else {
40                                      //Update pointer in the last node.
41                                      last_person->next = new_current_person;
42                              }
43                              last_person = new_current_person;
44                      } else {
45                              //Free this struct.
46                              free(new_current_person);
47                      }
48                      new_current_person = new_current_person->next;
49              } while(new_current_person != NULL);
50
51              return new_first;
52      }
```

Line 15 has an off-by-one error, since we process n+1 nodes, but the list only contains n. The check should be <, not <=.

Lines 16 and 24 misuse pointer arithmetic. These should just be
new_current_person = new_first + i;
andnew_current_person->next = new_current_person+1;

Line 20 references the first element of a char array, resulting in a char, when strcpy accepts a char *.

Line 19: the allocated array for the city is too short. Strlen returns the number of bytes

before reaching the null terminator. Thus we malloc one too few bytes to store the nullterminated copy of the city.

Line 21, after the check for the end of the list, should terminate if the end of the list has been reached . Not doing so relies on n being <= to the length of the original list.

Line 34 dereferences either a null pointer or a junk pointer if n is zero (in which case no memory will be allocated by malloc). We should check this condition at the start of the function to be defensive (unless we're really sure that all call will have n > 0.

Line 35 overflows the name array, since we are adding 20 characters to an array that can hold 19 characters and a null terminator (and was probably non-empty to begin with!).

Line 46 is the single worst line in this code. First of all, it is freeing a pointer that was not returned by malloc (malloc returned only the pointer to the beginning of the block in which we are storing the linked list. The entire block must be freed at once.) Even if this were not a problem, line 48 references the memory we would have just freed! Finally, if this code is fixed up, it will still contain unexpected memory leaks, since we must explicitly free the char array pointed to by city. This will not be done by code that simply frees the linked list.

Q7: Garbage Collection

7.1
What are some common memory bugs that garbage collection solves?

Garbage collection stops the programmer from needing to explicitly free memory. This would prevent freeing blocks multiple times, and referencing freed blocks. It may also prevent referencing of nonexistent locations (if garbage collection keeps locations around for as long as they are referred to). It also prevents some (but not all) kinds of memory leaks. (Why is it difficult to prevent all kinds of memory leaks?)

7.2 Which of the memory bugs in the code above would garbage collection have solved?

In Question 2 above, only the error on line 46 will be solved by garbage collection, since garbage collection prevents us from having to explicitly free memory at all. Higherlevel languages with implicit memory management solve most of these problems. For example, many languages include run-time checking of array bounds, which will catch errors such as the error on line 15 and buffer overflow bugs. Many languages also abstract away the use of pointers, preventing pointer arithmetic bugs.

Q8: Basic Assembly

Consider the following assembly code:
x at %ebp+8, n at %ebp+12

```
1       movl 8(%ebp), %esi
2       movl 12(%ebp), %ebx
3       movl $-1, %edi
4       movl $1, %edx
5       .L2:
6       movl %edx, %eax
7       andl %esi, %eax
8       xorl %eax, %edi
9       movl %ebx, %ecx
10      sall %cl, %edx
11      testl %edx, %edx
12      jne .L2
13      movl %edi, %eax
```

The preceding code was generated by compiling C code that had the following overall form:

```
1       int loop(int x, int n)
2       {
3               int result = -1;
4               int mask;
5               for (mask = 0x1; mask != 0; mask = mask << n) {
6                       result ^= (x & mask);
7               }
8               return result;
9       }
```

a. Which registers hold program values x, n, result, and mask?
We can see that result must be in register %edi, since this value gets copied to %eax at the end of the function as the return value (line 13). We can see that %esi and %ebx get loaded with the values of x and n (lines 1 and 2), leaving %edx as the one holding variable mask (line 4.)

b. What are the initial values of result and mask?
Register %edi (result) is initialized to −1 and %edx (mask) to 1.

c. What is the test condition for mask?
The condition for continuing the loop (line 12) is that mask is nonzero.

d. How does mask get updated?
The shift instruction on line 10 updates mask to be mask << n. Note %cl is the lower 8 bits of %ecx.

e. How does result get updated?
Lines 6–8 update result to be result ^ (x&mask).

g. Why are the arguments to loop pushed on the stack in reverse order (i.e., x ends up closer to loop's %ebp than n)? Why can't we do it the other way around?

What if the function doesn't know how many arguments it actually takes; e.g. a variable-argument function like printf? The compiled version of printf relies on the first argument to determine how arguments have been passed to it (e.g. if the first arg were "%d %d %s", it'd expect 3 additional arguments). If that format string were some unknown number of bytes away from %ebp, we'd never be able to figure out what are actually arguments to printf.

Q9: Procedure Calls

- Lets say we are given the following assembly code for a function:

```
1       %edi
2       pushl %esi
3       pushl %ebx
4       sub $0x24, %esp
5       movl 24(%ebp), %eax
6       imull 16(%ebp), %eax
7        movl 24(%ebp), %ebx
8       leal 0(, %eax, 4), %ecx
9       addl 8(%ebp), %ecx
10      movl %ebx, %edx
11      subl 12(%ebp), %edx
…...
20      popl %ebx
21      popl %esi
22      popl %edi
```

9.1

Why are %edi, %esi, and %ebx pushed onto the stack at the beginning of this function and popped off at the end?

Registers %edi, %esi, and %ebx are callee-saved registers. These registers must be saved on the stack by the callee and restored before returning, since the calling function expects them to be the same as when the callee was called.

9.2

What about %eax, %edx, and %ecx? Why aren't they put on the stack?

%eax, %ecx, and %edx are caller-saved registers. This means the callee may use and overwrite these registers without destroying any data required by the caller.

9.3

What do 24(%ebp) and 16(%ebp) refer to?

16(%ebp) refers to the 3rd argument passed to this function. 24(%ebp) refers to the 4th argument passed to this function. Lines 4 and 5 result in the multiplication of the 3rd and 4th arguments passed to this function. (We know that the first four arguments (there may be more) are all 4-bytes long, since we use "addl", "subl", etc. when dealing with them)

9.4

Why do we subtract 0x24 from %esp? What might be put in that area?

Subtracting values from %esp creates space between %ebp and %esp where we could store local variables or place the arguments to a function that we will call.

Q10: Flags

-For each one of the following, determine which flags are set by the add instruction and why.

10.1

```
movl $0x40, %eax
movl $0xffffffc0, %ebx
addl %eax, %ebx
```

We're performing 64 – 64. The result, stored in %ecx, is = 0. Hence, the ZF flag is set to 1. When the operands are interpreted as signed integers (64 and -64), the arithmetic operation does not overflow. Hence, the OF flag is set to 0. When operands are interpreted as unsigned integers (64 and 4294967232), the arithmetic operation overflows. Hence, the CF flag is set to 1. When the result is interpreted as a signed integer, the result is non-negative. Hence, the SF flag is set to 0.

10.2

```
movl $0x2a, %eax
movl $0xffffffc0, %ebx
addl %eax, %ebx
```

This is 42 – 64. The result is non zero. Hence the ZF flag is set to 0.
When the operands are interpreted as signed integers (42 and -64), the arithmetic operation does not overflow. Hence the OF flag is set to 0. When the operands are interpreted as unsigned integers (42 and 4294967232), the arithmetic operation does not overflow. Hence, the CF flag is set to 0. When the result is interpreted as a signed integer, the result is negative. (We know this since the MSB of the result from the addl instruction is 1). Hence, the SF flag is set to 1.

10.3

```
movl $0x7FFFFFF0, %eax
movl $0x2c, %ebx
addl %eax, %ebx
```

The result is nonzero, and so the ZF flag is set to 0. When the operands are interpreted as signed integers, the arithmetic operation overflows. Hence the OF flag is set to 1. When the operands are interpreted as unsigned integers, the arithmetic operation does not overflow. Hence, the CF flag is set to 0. When the result is interpreted as a signed integer, it is negative. Hence the SF flag is set to 1.

Q10: Tail Recursive Optimizations

```
int main(void) {
        printf( " %d\n ",  getSum( 5 ) )
        return 0;
}
int getSum(int num) {
        if ( num == 1 )
```

```
        return 1;
    else
        return getSum(num-1) + num;
}
```

10.1

What does the following code do?

Adds all the numbers from 1 to 5 and prints them to the screen.

10.2

Why is this code inefficient?

Suppose instead of passing 5 into getSum, we passed 5,000,000. Since the return statement needs the result of getSum(num - 1) before it can return, it must extend the stack frame for each call of getSum. As you can see, with 5,000,000 the stack will become very large.

10.3

Rewrite this function so it has the same behavior but performs more efficiently.

```
int main(void) {
    printf( " %d\n ",  getSum( 0, 5 ) )
    return 0;
}
int getSum(int sum, int num) {
    if ( num == 0 )
        return sum;
    else
        return getSum(sum+num, num-1);
}
```

Here we make use of tail recursion. Since the final return value of getSum relies completely on the next call, you do not need to extend the stack frame. The frame of the current procedure is not needed any more so it can be replaced by the frame of the tail call. This is an optimization the compiler makes.

Q11: Buffer Overflow

- Consider the following function

```
struct client {
  char name[9];
  int money;
};

struct client joinBank ( int initialDeposit );

int main (void) {

  int joiningReward = 50;
  struct client newClient = joinBank( joiningReward );

  printf("%d\n", newClient.money);
```

```
   saveClientToDataBase(newClient);
}

struct client joinBank( int joiningReward ) {
  struct client newClient;

  newClient.money = joiningReward;
  printf("What is the new client's name?\n");
  gets(newClient.name);

  return newClient;
}

void deleteAllBankUsers() {
   // deletes records of all bank clients!
}
```

12.1
What input would you input in order to instantly make $1,000,000?
The string, “ninecharspadaaaa” will make you a million plus change.

12.2
Suppose the instruction for deleteAllBankUsers() was on line 0x0806578c. What could a malicious user input to delete the records of all the bank clients?

A malicious could input a string that overwrites the return address to be to 0x0806578c.