CS 237 Meeting 10 — 9/28/12

Announcements

1. None?

Buffer Overrun Bugs

1. If you had never seen a bug like the problems that occurred when we typed in more than 10 arguments to the member program I showed in the last class, you should know that bugs of that sort in (mainly) C programs are one of the major sources of security flaws in computer systems.

   Just Google "buffer overrun bug" or "buffer overflow bug" and you will be amazed at just how common and significant such bugs are.

2. Buffer overrun bugs lead to security errors because they make it possible to put binary data at inappropriate places in a running program. We will see shortly, that one type of data that often ends up stored in memory is the return address that is put in $ra. If one puts some other data in the memory locations used to save $ra, eventually the function that saved it will load it and branch to the replacement value. This means you can make a program execute code that is different from what it was supposed to be executing.

Pointer Arithmetic and Arrays (reviewed)

- Last time, we looked at a simple implementation of bubble sort shown in Figure **??**.
- This code uses the swap function shown in Figure **??** in an interesting way. It always passes it a pair of consecutive elements of the values array.
- In C, it is legal to subtract one pointer value from another or to add integer values to pointer values.
- When C does arithmetic with pointers, it assumes the programmer wants to think abstractly about the things the pointers point to rather

```
int outOfOrder = TRUE;

while ( outOfOrder ) {
  outOfOrder = FALSE;
  for ( p = 0; p < valc - 1; p++ ) {
    if ( values[p] > values[p+1] ) {
          outOfOrder = TRUE;
          swap( &values[p], &values[p+1] );
    }
  }
}
```

Figure 1: bubbleSort

```
// A swap that works!
void swap( int * x, int * y ) {
  int temp;

  temp = *x;
  *x = *y;
  *y = temp;
}
```

Figure 2: Swap using pointers

```
// A swap that works!
void swap( int * x, int y[] ) {
  int temp;

  temp = *x;
  x[0] = *y;
  y[0] = temp;
}
```

Figure 3: Swap using pointers

than about memory addresses. Therefore, if two pointers refer to items in consecutive locations in an array, their difference should be 1 even if the actual addresses for the array elements differ by 4.

- To pull this off:
  - The difference between two pointer values (that point to the same type) will always be the difference of the memory addresses involved divided by the number of bytes required to store a value of the type begin pointed to.
  - When we add/subtract an int constant from a pointer, the constant is first multiplied by the size of the item the pointer refers to.

Pointers and Array indexing

- In C, an array name by itself produces the address of the 0th element of the array.
- As a result of this, and the rules explained above, if a is an array and x is an int, then a + x is the address of array element a[x] and therefore *(a+x) is equivalent to a[x]. An array's name is really viewed as a pointer to the element type of the array.
- This means we can do some really weird things:
  - We can rewrite our swap function as shown in Figure **??**.
  - Worse yet, given that when bubble sort calls swap y always refers to the number right after x, we could also say

    x[0] = x[1]

    instead of

    x[0] = *y

Strings in C

1. You have already seen one glimpse of how C deals with strings. In the declaration of main, we declare argv as "char *argv[]" and you know that argv is an array of strings so "char *" must somehow be the type C uses for strings.
2. Now that you know that pointers and array are somewhat interchangeable, this should actually make sense. A "char *" and a "char[]" are pretty much the same in C and a char is just a single character. Thinking of a string as an array of characters may make sense (at least more sense than thinking of it as a pointer to a single character).
3. We have seen that arrays in C don't know how long they are. For the arrays of characters, C deals with the by sticking an extra array element equal to 0 after the array elements that hold the actual characters for the string. So, to process a string, you tend to write loops that run down the array elements until you find a 0.
4. The program writeDown, which expects one argument and prints the characters in this argument vertically on the screen, gives a simple example of such processing. It is shown in Figure **??**.
   - The program uses a new printf specifier, %c, to output a single character.
   - It also illustrates a C idiom that takes advantage of the fact that anything that is not 0 is true to skip checking whether arg[p] != 0.

Multi-dimensional Arrays

```
#include <stdio.h>

// Print a string one character per line
void printDown( char arg[] ) {
  int p;
  for ( p = 0; arg[p]; p++ ) {
    printf( "%c\n", arg[p] );
  }
}


// This program takes a single argument and prints its text
// in a vertical column on the screen
int main( int argc, char * argv[] ) {
  printDown( argv[1] );
}
```

Figure 4: writeDown.c

1. C also supports multidimensional arrays and arrays of pointer to arrays. Beyond mentioning this, we will not really discuss them at this time.

Accessing Memory in MIPS Assembly Language

1. When we first discussed MIPS assembly language, we explained that in addition to the 32 registers in the machine, it is possible to store and access additional data stored in a much larger memory.

2. We briefly discussed the existence of two instructions name Load Word (lw) and Store Word (sw). Now it is time to examine the details of these instructions and several related instructions.

   - Both of these instructions are encoded in the I-Type format used for instructions with immediate operands like addi.
   - Each instructions encoding includes one register number for the register to/from data should be moved and a register/16-bit immediate operand pair that determines the address of the word in memory from/to which data is moved.
   - The memory address is determined by adding the immediate operand and the value of the associated register. It must be divisible by 4.
   - There are variants of the load and store instructions for shorter data transfers:

     lb, lbu and sb are used to transfer a single byte of data between a register and memory location. For lb, the byte is placed in the lowest 8 bits of the register and the sign bit is extended into the remaining bits. For lbu, the high order bits are set to 0. For sb, the lowest 8 bits of the register are stored in memory.

     lh, lhu, and sh are used to transfer two bytes of data between a register and memory location. The memory address must be divisible by 2. the lowest 16 bits of the register are used for the data.

3. In addition to learning about these instructions, we need to know a few things about support the assembler provides for memory access. These features and the use of lw and sw are illustrated by the (almost complete) assembly language version of the C program we used to show how swap could be written in C in our last class. The code for this is shown in Figure **??**.

   - Before the main method, we associate the labels a and b with words in memory using the .word directive. This makes it easy to access them directly. When we want to print their values, we just use their names in lw instructions.

4. This program also shows how to reference a memory location through a pointer rather than a simple name.

   - In the main method, we pass pointers to a and b to swap using the la (load address) instruction.
   - In swap, we have to load these addresses into register and then combine the value in the register with an immediate offset of 0. The notation "($a0)" in the load and store instruction in swap indicate that the address of the memory work to be accessed is in $a0.

```
.text
      j main
#void swap( int *x, int *y ) {  x in $a0,  y in $a1
#  int temp;    temp in $t0
swap:
#  temp = *x;
      lw $t0,($a0)

#  * x = * y;
      lw $t1,($a1)
      sw $t1,($a0)

#  y = * temp;
      sw $t0,($a1)
#}
      jr   $ra
#
#int main( int argc, char * argv[] ) {   argc in $a0,  argv in $a1
#
.data
#  int a = 10;
a:    .word      10
#  int b = 100;
b:    .word      100

.text
#  swap( a, b );
main:      la      $a0,a
      la      $a1,b
      jal      swap
#
#  printf( "a = %d, b = %d\n", a, b );
      li      $v0,4
      la      $a0,aMess
      syscall
      li      $v0,1
      lw      $a0,a
      syscall
# SOME RELATIVELY BORING CODE IS MISSING HERE

.data
aMess:      .asciiz "a = "
bMess:       .asciiz " b = "
newLine: .asciiz "\n"
#}
```