CSC C69: OPERATING SYSTEMS

Tutorial 1
Thursday, Jan 17, 2013
TA: Ioan Stefanovici (ioan@cs.toronto.edu)
HOW DO YOU SUCCEED IN THIS COURSE?

• Show up to lectures & tutorials (way too much material)
• Work on assignments evenly (fill in the other partner...)
• Compiler warnings! (-10% automatic deduction since B09)
• ‘svn add’ files to the repository!
  • Do a complete, clean check-out (especially for OS/161...)
• Read assignments very closely
  • Lots of details, corner cases, and design decisions to make
• Keep things modular (especially in OS/161)
• Use tools & resources available to you and be proactive (good for industry as well)
• Design documents (more than line-by-line description of your code; explain, don't regurgitate)
SOME C REVIEW

• Go through these slides (and try the exercises...) at home!

• Brush up / learn what you don’t know now!

  • Assignments are work-intensive enough as it is...

• Topics: Bit manipulations, pointers, argument-passing, arrays, pointer arithmetic, memory allocation, error handling, etc.
C: BIT MANIPULATION

- Sometimes we need to alter bits in a byte or word of memory directly
  - A 32-bit int is a very compact way to represent 32 different booleans
- C provides bitwise boolean operators
  - "&": AND
  - "|": OR
  - "~": NOT
  - "^": XOR (exclusive OR)
- These are **NOT** the same as the logical operations
# PRACTICE WITH BIT OPS

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a</strong></td>
<td>0110 1001</td>
</tr>
<tr>
<td><strong>b</strong></td>
<td>0101 0101</td>
</tr>
<tr>
<td>~a</td>
<td></td>
</tr>
<tr>
<td>~b</td>
<td></td>
</tr>
<tr>
<td>a &amp; b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a ^ b</td>
<td></td>
</tr>
</tbody>
</table>
BIT MASKS

• A mask is a bit pattern that indicates a set of bits in a word
  • e.g.: 0xFF would represent the least significant byte of a word
    • Happily, this is true regardless of byte ordering
  • For a mask of all 1s, the best way is ~0
    • Portable, not dependent on word size
  • For 32-bit machines, 0xFFFFFFFF will work
    • You may also see -1 used (2s complement, -1 is a bit pattern will all bits set to 1)
PRACTICE WITH BIT MASKS

• Given an integer x, write C expressions for:
  
  • Least significant byte of x, all other bits set to 1
    
    • int y = ________________________________
  
  • Complement of the least significant byte of x, all other bytes unchanged
    
    • int y = ________________________________
  
  • All but least significant byte of x, with least significant byte set to 0
    
    • int y = ________________________________
BIT SHIFTING

• \( x \ll k \): shift the bits of \( x \) by \( k \) bits to the left, dropping the \( k \) most significant bits and filling the rightmost (least significant) \( k \) bits with 0s

  • What if \( k \) is \( \geq \) size of object? (e.g., for int's, on 32-bit machine, \( k \geq 32 \))
    • UNDEFINED! Don’t assume result will be 0

• \( x \gg k \): right shift, logical or arithmetic

  • Logical: fill left end with \( k \) 0s (unsigned types)

  • Arithmetic: fill left end with \( k \) copies of the most significant bit (sign bit)

  • C does not define when arithmetic shifts are used! Typically used for signed data, but not portable
POINTERS

- Every variable has a memory address
  - Can be accessed with "address of" operator: &
- Pointers are variables that store memory addresses
  - int x = 42;
  - int *x_ptr = &x;
  - int *heap_ptr = (int *)malloc(sizeof(int));
- The value a pointer refers to can be accessed with *
  - This is "dereferencing"
  - int y = *x_ptr;
NULL

- NULL is the “0” value for addresses.
  - It’s a good idea to initialize pointers to NULL.
    - Much easier to catch bugs!
  - It’s often used as an error value, too.
PASS BY VALUE/REFERENCE

• C only allows one value (which may be a struct) to be returned.

• If variables are passed into a function by value, any changes to them will not be seen outside the function.
  
  • Why? A copy of each parameter is made on the stack, and changes are made to the copy.

• If pointers are passed into a function, any changes made to the value they point to will be seen outside the function -- this is passing by reference.

  • Note that the pointers themselves are still passed by value!
Arrays contain multiple variables of the same type.

Each element can be accessed with the [] notation

```c
int x_arr[10];
for (i = 0; i < 10; i = i+1)
    x_arr[i]=i;
```

Arrays are... just pointers.

```c
int *x_ptr = x_arr;
```

But they're often easier to reason about. Use them!
POINTER ARITHMETIC

• Pointers are just values, so you can manipulate them.

• If x is an array, this is true:

  • x[5] == *(x+5)

• The key? Constants are added to pointers “scaled” by the size of the type. Adding 5 to an (int *) adds 5*sizeof(int).

• And also, strangely, this is true (on most systems):

  • 5[x] == x[5]

  • (but don’t use it...)
ALLOCATING MEMORY

- malloc allocates memory from the heap
  - It allocates by byte, so it requires a size
  - Its return value must be typecast
    - int *heap_ptr = (int *)malloc(sizeof(int)*4);
- Don’t forget to “free” the memory you “malloc”!
- Remember to use “kernel” versions of the calls if you’re working inside the kernel
  - Instead of malloc, kmalloc
  - Instead of free, kfree
POINTERS AND STRUCTS

• Structs are one “aggregate” structure in C.
  • A struct can contain multiple variables in a single package

• Structs have a syntactic quirk:
  • If you have a struct variable, use “.”
    • struct mystruct s = ...;
    • s.myfield = 6;
  • If you have a struct pointer, use “->”
    • struct mystruct *s_ptr = ...;
    • s_ptr->myfield = 6;
    • (*s_ptr).myfield = 6;
C ERROR MESSAGES

• Segmentation Fault
  • A pointer has accessed a location in memory that is not in a segment you own.
  • Maybe an infinite loop: overran an array?
  • Adding two pointers that shouldn’t be?
• Bus error
  • A pointer is not properly aligned
  • Bad casting?
• Incorrect pointer arithmetic?
MORE C QUIRKS TO REMEMBER

- Uninitialized variables have undermined value
  - ...and C will let you use them
  - (good idea to initialize variables to default values; e.g.: pointers to NULL)

- Always check return values
  - ...to make sure an error has not occurred
  - Return values flag errors -- no “exceptions”

- C has no runtime checks
  - Casts are dangerous, but you’ll need some
  - Arrays don’t store their size -- you can easily run over
  - Memory can be corrupted without the program crashing! check your bounds!
STACK ALLOCATION

- Stack allocate!
- Heap allocation isn’t always necessary
- Also, might cause a memory leak (if not careful...)

```c
int foo(){
    struct mystruct z;
    s.x = 1;
    return funcwithmystruct(&z);
} ...NOT

int foo(){
    struct mystruct* z = malloc(sizeof(struct mystruct));
    int rval = -1;
    z->x = 1;
    rval = funcwithmystruct(z);
    free(z);
    return rval;
}
```
DON’T LEAK MEMORY!

• Make sure to free memory you allocate
• This example shows an error case

```c
struct mystruct* sys_mystruct(){
    struct mystruct* first;
    first = malloc(sizeof(struct mystruct));
    if (first == NULL){
        return -1;
    }
    first.other = malloc(sizeof(struct otherstruct));
    if (first.other == NULL){
        free(first);
        return -1;
    }
    return first;
}
```
GENERAL TIPS

- Simplify whenever possible
  - struct mystruct myarray[10][10];
  - is better than
    - struct mystruct **myarray;
- Declare all functions ahead of time
- Use a test-oriented **incremental** development strategy
  - Test first and frequently
ASSIGNMENT I

• Implement a shell
  • printf vs. write call
• fork & wait
• Linked lists