CSC369 Tutorial 1

Some review material
How to succeed in this course

• Show up to lectures & tutorials
  – More material to cover than lecture time available
• Work on assignments evenly and collaborate
  – “Fill your partners in” and make sure you all understand *everything*.
• Compiler warnings!
  – In the past, automatic 10% penalty on assignments.
• SVN
  – `svn add`; do a clean checkout and build (from scratch) before you submit your assignments
How to succeed in this course

• Read assignments carefully; lots of corner cases & design decisions to make
• Read the documentation
• Keep things modular
  – Make this part of your initial design
• Use the tools available to you & be proactive in learning them
  – Good for industry as well
• Design documents
  – More than line-by-line descriptions of your code
• Explain the design (how/why); don’t regurgitate the code
Architecture Review
CPU

• The Program Counter (PC)
• The Stack Pointer (SP)
• Data Registers
• Flow of normal execution
  – Memory address and load/store instructions
• Interrupts!
CPU

INC A  0011 1100  3C
CPU

INC A  0011 1100  3C
CPU

INC A  0011 1100  3C
CPU

INC A    0011 1100    3C

End of FETCH
INC A  0011 1100  3C

DECODE
CPU

INC A  0011 1100  3C

EXECUTE
CPU

INC A 0011 1100 3C

EXECUTE
Memory Hierarchy and Trade-off

- Can’t have the fastest memory, largest capacity, and be the cheapest...
- OS must do smart things to efficiently use different types of memory (Caching)
Memory

• Program sees linear address space, segmented
  – Code
  – Data
  – Stack
  – Heap

• Where does the OS go?

• Do programs share the same space?
The Stack grows down!

Every function puts a "frame" onto the stack. The frame contains local variables and then lots of system registers. (More on this later.)

The SP (stack pointer) points to the TOP of the stack -- the next free piece of memory. The FP (frame pointer) points to a specific location in the frame -- like the beginning.

The Heap grows up!

Constant section (.data):
Constants and global variables go here.

Code section (.text):
Your program goes here!

2^i - 1, where i is 32 on a 32-bit machine. Note that this is a small lie: the OS needs some space, so you don't actually get all of this memory. We'll talk about this more in 389.

The PC (program counter) is a CPU register that stores a pointer to the next instruction. It points into this section.

00000000 is an illegal address. Why?
Stack Frame of Function Call
Stack Frame of Function Call

Function 1

Function 2

Function 3

Function 4

SP

FP

P2 Local Variable 2
P2 Local Variable 1
Return address of P1
Parameter 1
Parameter 2
P1 Local Variable 2
P1 Local Variable 1
Stack Frame of Function Call

Function 1

Function 2

Function 3

Function 4

Stack Frame:
- P3 Local Variable 2
- P3 Local Variable 1
- Return address of P2
- Parameter 1
- Parameter 2
- P2 Local Variable 2
- P2 Local Variable 1
- Return address of P1
- Parameter 1
- Parameter 2
- P1 Local Variable 2
- P1 Local Variable 1

SP
FP
Stack Frame of Function Call

Function 1

Function 2

Function 3

Function 4

SP

FP

P4 Local Variable 2
P4 Local Variable 1
Return address of P3
Parameter 1
Parameter 2
P3 Local Variable 2
P3 Local Variable 1
Return address of P2
Parameter 1
Parameter 2
P2 Local Variable 2
P2 Local Variable 1
Return address of P1
Parameter 1
Parameter 2
P1 Local Variable 2
P1 Local Variable 1
Stack Frame of Function Call

Returning

Function 1

Function 2

Function 3

Function 4

Parameter 1
Parameter 2
P3 Local Variable 2
P3 Local Variable 1
Return address of P2
Parameter 1
Parameter 2
P2 Local Variable 2
P2 Local Variable 1
Return address of P1
Parameter 1
Parameter 2
P1 Local Variable 2
P1 Local Variable 1

SP
FP
Stack Frames

- First 4 arguments: $a0-$a3
- Return value (or pointer to it): $v0
- Return address: $ra
- Frame pointer: $fp

### Registers for O32 Calling Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
<th>Caller must preserve?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>constant 0</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>assembler temporary</td>
<td>No</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>$2-$3</td>
<td>values for function returns and expression evaluation</td>
<td>No</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>$4-$7</td>
<td>function arguments</td>
<td>No</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>$8-$15</td>
<td>temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>$16-$23</td>
<td>saved temporaries</td>
<td>Yes</td>
</tr>
<tr>
<td>$s8-$s15</td>
<td></td>
<td>temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>$26-$27</td>
<td>reserved for OS kernel</td>
<td>N/A</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>global pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>return address</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Diagram

- $sp$ before call
- $fp$ during call
- Extra Arguments
  - $ra$
  - $fp$
  - Preserved registers
  - Padding
  - Local data (and $a0$-$a3$)
  - Extra outgoing arguments

http://www.cs.ucsb.edu/~franklin/30/spim/BookCallConvention.htm
C REVIEW
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.
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.
    • Including fields of structures
    • Check pointers for NULL before dereferencing
    • 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 values they point to will be seen -- this is passing by reference.
  – Note that the pointers themselves are still passed by value!
Arrays

- Arrays contain multiple variables of the same type.
- Each element can be accessed with [] notation.
  
  ```
  int x_arr[10];
  for (i = 0; i < 10; i=i+1)
    x_arr[i] = i;
  ```
- Arrays are ... almost the same as pointers.
  
  After “int *x_ptr = x_arr;” x_ptr[i] is just like x_arr[i]
  
  - Differences:
    - `sizeof(x_ptr) = 4 (sizeof(int*))`, whereas `sizeof(x_arr) = 40 (10*sizeof(int))`
    - You can’t change an array var. to point to a different array
  - Note: arrays are passed to a function as a **pointer**, not an array-typed variable
• Pointers are just values, so you can manipulate them.
• If x is an array, this is true:
  \[ x[5] == *(x + 5) \]
• The key? Constants added to pointers are “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]
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;
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” 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
Stack Allocation

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

```c
int foo() {
    struct mystruct z;
    z.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;
}
```
Stack versus Heap trade-off

• Stack allocation is “easy,” but stack sizes are limited. (1-4MB for a “regular” system, and only 4KB for a kernel thread running on sys161)
  – This means any array or struct with more than a handful of elements should be heap allocated.
  – Cannot return a pointer to a stack-allocated variable
  – Additionally, no recursion in kernel threads!

• Heap allocation is “harder,” but gets around these limitations. Why is it harder?
  – Have to remember to free any malloc/calloc’d mem.!
  – Can’t free a memory location more than once!
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) {
        return -1;
    }
    return first;
}
```
More C Quirks to Remember

- Uninitialized variables
  - ... have undetermined value (and C won’t complain)
- Array bounds
- Runtime exceptions
  - ... don’t exist!
  - Instead, functions return, e.g., “-1” or “0”
- Type casts
  - ... are not checked at runtime! (can cast char to int*)
  - “Dangerous,” but you’ll need to do it sometimes.
- Memory can be corrupted without the program crashing: check your bounds!
- Use assert() to check invariants
  - but not error conditions that are actually possible!
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?
  – Forgot to initialize a pointer and dereferenced it?
  – Adding two pointers that shouldn’t be?
  – Note: segfaults can be sporadic, since you have to step outside the (rather large) segment to get one.

• Bus Error:
  – A pointer is not properly aligned.
  – Bad casting? Bad pointer arithmetic?
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
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 boolean values
- C provides bitwise boolean operators
  - “&” : AND
  - “|” : OR
  - “~” : NOT (complement)
  - “^” : XOR (exclusive OR)
Practice with bit ops

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0110 1001</td>
</tr>
<tr>
<td>b</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></td>
</tr>
<tr>
<td>a ^ b</td>
<td></td>
</tr>
</tbody>
</table>
## Practice with bit ops

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0110 1001</td>
</tr>
<tr>
<td>b</td>
<td>0101 0101</td>
</tr>
<tr>
<td>~a</td>
<td>1001 0110</td>
</tr>
<tr>
<td>~b</td>
<td>1010 1010</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>0100 0001</td>
</tr>
<tr>
<td>a</td>
<td>0111 1101</td>
</tr>
<tr>
<td>a</td>
<td>0011 1100</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
  – For a mask of all 1’s, the best way is ~0
    • Portable, not dependent on word size
    • For 32-bit machines, 0xFFFFFFFF will work
    • You may also see -1 used (2’s complement, -1 is a bit pattern with all bits set to 1)
Practice with bit masks

• Given an integer x, write C expressions for:
  – Set n-th bit of y:
    • int y |= 1 << n
  – L.s.b unchanged, toggle all other bits of y:
    • int y ^= 0xffffffff00
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 l.s.b. of x, all other bytes unchanged:
    • int y = ________________
  – All but l.s.b. of x, with l.s.b. set to 0
    • int y = ________________
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 = (x & 0xFF) | 0xFFFFFFFF00
  – Complement of the l.s.b. of x, all other bytes unchanged:
    • int y = x ^ 0xFF
  – All but l.s.b. of x, with l.s.b. set to 0
    • int y = x & 0xFFFFFFFF00
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 0

• Example: $6 \ll 1 = 12$

Before: 00000000 00000000 00000000 00000110
After:   00000000 00000000 00000000 00001100

Equivalent to multiplying by $2^k$
Bit Shifting

• Shifting is *non circular*

• E.g 3,758,096,384 << 1

  • Before: 11100000 00000000 00000000 00000000
  • After : 11000000 00000000 00000000 00000000

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

- $x >> k$ right shift, logical or arithmetic
  - logical right shift - fill left end with $k$ 0's (unsigned types)
  - arithmetic right shift (care about signed bit) - fill left end with $k$ copies of the most significant bit
    - C does not define when arithmetic shifts are used! Typically used for signed data, but not portable

- Example -2,147,483,552 >> 4

  - Before: 10000000 00000000 00000000 01100000
  - Arithm: 11111000 00000000 00000000 00000110
  - Logical: 00001000 00000000 00000000 00000110
Exercise 1

In groups (max 3)