CS485G classnotes

Raphael Finkel

March 13, 2015

1 Intro

1. Handout 1 — My names
2. Plagiarism — read aloud
3. E-mail list: cs485@cs.uky.edu
4. Labs: about five throughout the semester, typically on Fridays. The first one is this Friday. Labs count toward your grade; you must do them during the lab session.
6. Accounts in MultiLab if you want; every student will have a virtual machine as well, at name@name.netlab.uky, where name is your LinkBlue account name.

2 Software tools

Use (client) Spec Programmer Language
Implementation Compiler
3 Abstraction and reality

1. Most CS and CE courses emphasize abstraction; it matches how we think, and it lets us hide implementation details and complexity.

2. But hardware has limits that our abstractions lack (maximum size of an integer, for instance). If we hide implementation details, we are at risk of inefficiency and inability to cooperate with other components.

3. Examples

(a) C int is not an integer: 50000*50000 = 2500000000, but the int is -1794967296.

(b) C float is not real: 1e20 + 3.14 - 1e20 = 3.14, but the float result is 0.0.

(c) Programming languages hide the instructions that are executed
(d) Layout in memory affects performance (caches, pages). Example:

```c
#define BIG 10000

void copyij(int src[BIG][BIG], int dst[BIG][BIG])
{
  int row, col;
  for (row = 0; row < BIG; row += 1) // reorder?
    for (col = 0; col < BIG; col += 1) // reorder?
      dst[row][col] = src[row][col];
}

int from[BIG*BIG], to[BIG*BIG];
copyij(from, to);
```

One experiment shows that in the order given, user time is 0.22 seconds; with interchanged order, user time is 1.13 seconds.

4. We no longer teach assembler-language programming, because compilers are much better and more patient than assembler-language programmers.

5. But you need to understand computation at the assembler level.

(a) When your program has a bug, high-level models can fail.

(b) To improve performance, you need to understand what optimizations the compiler can and cannot do.

(c) When you write system software, what actually runs is machine code.
(d) Operating systems need to deal with the intricacies of machine code as they manipulate processes (keeping track of floating point registers, the program counter, memory mapping tables)

(e) Malware is often written in x86 assembler.

4 Memory-referencing bugs

1. C (and C++) are subject to memory-referencing bugs.

(a) Array references out of bounds. Example (the actual result is architecture-specific; the results shown are on x86_64)

```c
void fun(int index)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[index] = 9223372036854775803L; // index out of bounds?
    printf("%lf\n", d[0]);
}
fun(0); // 3.14
fun(1); // 3.14
fun(-1); // 3.14
fun(-2); // 0.0
fun(2); // nan (not a number)
fun(-3); // 3.14
fun(3); // 3.14 followed by fault during the return
```

Reason: a and d are adjacent on the stack. When you go past the end of a, you might modify d, or you might access saved state on the stack, ruining the return address.

(b) Lecture 2, 1/16/2015: lab 1

(c) Lecture 3, 1/21/2015

(d) Dereferencing invalid pointers

(e) Improper allocating and deallocating memory regions

(f) Unfortunately, the symptoms may be unrelated to the causes, and the effect might be visible only long after it is generated.

(g) Some languages prevent such errors: Java, Ruby, Haskell, ML, but they are not usually used for systems programming.
(h) There are tools to help you detect referencing errors (such as valgrind).

5 Binary representation

1. Bits are represented by two voltages, one for 0 and another for 1. A typical representation would be .3V for 0 and 3.0V for 1, but every architecture bases the values on the particular kind of transistors it uses. When a voltage changes from one value to the other, there is an intermediate time at which its value is indeterminate; hardware carefully avoids inspecting the voltage then.

2. We usually think of numbers in decimal (base 10), but for systems programming, we sometimes need to use binary (base 2) or hexadecimal (base 16) representation.

   (a) Base 2 uses only digits 0 and 1. Represent, for instance, 5.25 as 101.01₂. Some numbers can be represented exactly in decimal but not in binary: 5.3 = 101.0100110011...₂.

   (b) Base 16 uses digits 0...9, A, B, C, D, E, F. The letters are usually written in capital letters. Each hex digit corresponds to four bits. 285.3 = 11D.4CCCCCCC16...

3. A byte is usually 8 bits. (The official name, used in computer networks, is octet, but we’ll just say “byte”). When treated as an unsigned integer, a byte has values ranging from 0 to 255 (or 11111111₂ = FF₁₆).

4. Signed integers using \( n \) bits can store numbers in the range \(-2^{n-1} \ldots 2^{n-1} - 1\). For \( n = 32 \), the range is \(-2147483648 \ldots 2147483647\) or \((-80000000₁₆ \ldots 7FFFFFFF₁₆)\).

5. It’s pretty easy to see how many distinct values you can store in \( n \) bits. Since every bit can be 0 or 1, there are \( 2^n \) possibilities. Luckily, \( 2^{10} \approx 10^3 \), so \( 2^{32} = 2^2 \times 2^{30} = 4 \times (2^{10})^3 \approx 4 \times (10^3)^3 = 4 \times 10^9 = 4 \) billion. Or just remember that

   \[
   2^{10} = 1024 \approx 10^3 = 1 \text{ thousand (kilo or K)}; \\
   2^{20} = 1048576 \approx 10^6 = 1 \text{ million (mega or M)}; \\
   2^{30} = 1073741824 \approx 10^9 = 1 \text{ billion (giga or G)}; \\
   2^{40} \approx 10^{12} = 1 \text{ trillion (tera or T)}; \\
   2^{50} \approx 10^{15} = 1 \text{ quadrillion (peta or P)}.
   \]

   So \( 2^{32} = 4 \text{G} \).
6. What is the largest signed integer you can store in 16 bits? (Answer: $2^{15} - 1 = 32767$)

7. How many bits do you need to store 4893? It’s about $4 \times 10^3$, so about 12 bits. (The right answer is 13 bits, but in a 2’s complement representation, at least 14 bits.)

### 6 C types and their sizes

1. Unfortunately, C declarations are machine-specific. Here is the size in bytes of various declarations.

<table>
<thead>
<tr>
<th>C declaration</th>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8 !</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16 !</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8 !</td>
</tr>
</tbody>
</table>

### 7 Byte ordering

1. If a word has more than one byte, what is the order?
2. Example: $19088743 = 01234567_{16}$
3. Big-endian: Least significant byte has the highest address. (Sun, PPC, Mac, Internet). The bytes, in order, are $0x01$, $0x23$, $0x45$, $0x67$.
4. Little-endian: Least significant byte has the lowest address. (x86). The bytes, in order, are $0x67$, $0x45$, $0x23$, $0x01$.
5. You can use the `od` program to show a file in bytes, characters, integers, ...

6. [Lecture 4, 1/23/2015](#) Lab 2
7. [Lecture 5, 1/26/2015](#)
8. You can also use a program:

```c
typedef unsigned char *pointer;
void show_bytes(pointer start, int len){
    int i;
    for (i = 0; i < len; i += 1) {
        printf("%p\t0x%.2x\n", start+i, start[i]);
    }
    printf("\n");
}
int a = 15213;
printf("int a = %d;\n", a);
show_bytes((pointer) &a, sizeof(int));
```

The output is:

```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

8. **Memory organization**

1. We usually address memory in **bytes**, although older computers used to measure in “words”, which could be of any length (PDP-10: 36 bits per word).

2. When a program is running, we call it a **process**.

3. From a process’s point of view, memory looks like a long array, starting at byte address 0 and going to some limit determined by the operating system. (On Linux for x86, memory is limited to 3GB.)

4. The operating system creates a separate address space for each process. We say that process address spaces are **virtual**, because when a process refers to address n, it is very likely not at physical address n.

5. Because processes get individual address spaces, they cannot read or write in each other’s address spaces, although the operating system can also arrange for some sharing.
6. The operating system allocates **physical** space, which also looks like an array ranging from address 0 to a limit determined by how much physical memory the machine has.

7. Within a process, the program uses memory for various purposes. The compiler decides where in memory to put various items, including the instructions, initialized data, uninitialized data, stack, and heap.

8. A 32-bit architecture generally means that integers are contained in 32 bits, and that virtual addresses use 32 bits (unsigned). The maximum address is therefore $4G-1$. That memory size is too small for some applications.

9. A 64-bit architecture generally means that integers are contained in 64 bits, and that virtual addresses use 64 bits (unsigned). The maximum address is therefore about $1.8 \times 10^{19}$. The x86_64 architecture supports only 48-bit addresses, which gives 256TB.

10. Architectures generally support multiple data formats. So a 64-bit architecture might be able to manipulate 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit integers.

9 **Strings and Buffers**

1. A C **string** is an array of bytes, each representing a single character, terminated by a null (zero) byte.

2. Declaration

   (a) `char *myString;`
   (b) `char myString[];`
   (c) `char myString[200];`

3. The representation is typically 7-bit ASCII.

4. Some representations, such as UTF-8, might use several bytes for a single Unicode character. So the length of the array is not necessarily the number of characters.

5. There is no need to worry about byte ordering; the start of the string always has the lowest address in memory.
6. **A buffer** is also an array of bytes, typically used to hold data subject to I/O. The bytes hold arbitrary binary values, not necessarily printable values.

7. Declaration

   (a) `char *myBuffer;`
   (b) `char myBuffer[];`
   (c) `char myBuffer[4096];`

8. Buffers are not null-terminated; you need a separate variable to remember how much data is in the buffer.

## Boolean algebra

1. [Lecture 6, 1/28/2015](#)

2. Named after George Boole (1815–1864).

3. A computer’s circuitry uses pieces that accomplish Boolean functions in order to build both combinatorial and sequential circuits.

4. We are familiar with the *truth tables* for **and** (in C: `&`), **or** (|), **not** (~). We might not be familiar with **exclusive or** (xor, ^).

5. When one operates on bytes (or larger chunks such as integers) with Boolean functions, they are applied bitwise. Examples:

   
   
   01101001 105
   &01010101 85
   ========
   01000001 65

6. Instead of interpreting 32 bits as an integer, we can interpret it as a subset of \{0, ..., 31\}. Each 1 bit represents a number in that range that is in the set; every 0 bit represents a number that is not in the set. So 1001 represents the set 0, 3. Then:

   (a) `&` is intersection.
   (b) `|` is union.
   (c) `^` is symmetric difference.
   (d) `~` is complement.
7. One can use the Boolean operators in C and apply them to any integral data type: `char`, `short`, `int`, `long`, `long long`.

8. Don’t confuse these operators with logical operators `&&`, `||`, and `!`. In C, 0 is understood to mean `false`, and any other value is `true`. The logical operators always return either 0 or 1. They use short-circuit semantics.

11 Shifting operators

1. Left shift: `x << y` Left-shifts bits in `x` by `y` positions; new positions on the right are filled with 0. **Warning:** In C (and Java), if `y` is equal to or greater than the number of bits `n` in the type of `x`, the shift distance is `y mod n`. So shifting a 32-bit integer by 34 bits only shifts it `34 mod 32 = 2` bits.

2. Right shift: `x >> y` Right-shifts bits in `x` by `y` positions; new positions on the left are filled with the sign bit. The same warning applies.

12 Encoding integers

1. Given `n` bits, there are `2^n` possible integers.

2. In unsigned form, these range from 0 (represented as all 0 bits) to `2^n − 1` (represented by all 1 bits).

3. In signed form, these range from `-2^{n-1}` to `2^{n-1} − 1`.

   (a) 0 is represented by all 0 bits.

   (b) The most significant bit is 0 for positive numbers. The largest positive number has `n − 1` bits set, representing `2^{n-1} − 1`.

   (c) The most significant bit is 1 for negative numbers. The smallest negative number has only that bit set, representing `-2^{n-1}`.

   (d) To negate a number (either a positive or negative one), invert all the bits and add 1 to the result, ignoring overflow.

   (e) 4-bit examples

   i. `0000_2 = 0`
   ii. `0110_2 = 6`
   iii. `1010_2 = -6`
iv. \(1000_2 = -8\). Try to negate this number.
v. \(0111_2 = 7\).
vi. \(1001_2 = -7\).

### 13 Compilation and disassembly in Linux

2. Command-line options are by Unix convention marked with `-`.
   - `-o filename` put the output of compilation in `filename`
   - `-E` Don’t compile; just run the preprocessor
   - `-S` Compile but don’t assemble; result is `filename.s`
   - `-c` Compile and assemble, but don’t link; result is `filename.o`
   - `-g` Add debugging information to the result.
   - `-O n` Turn on optimization level `n` (from 0 to 3)
3. Tools to inspect compiled code
   - (a) `objdump -d filename`: disassembles `filename`
   - (b) `gdb filename`: runs the debugger on `filename`; can disassemble
   - (c) `nm filename`: shows location and type of identifiers in `filename`
   - (d) `strings filename`: shows all the ASCII strings in `filename`.
   - (e) `od filename`: displays `filename` in numeric or character format.
   - (f) `ldd filename`: tells what dynamic libraries `filename` uses.
   - (g) `dissy filename`: graphical tool to inspect `filename`. You can install `dissy` by using `apt-get`.

### 14 Machine basics

1. Lecture 7, 1/30/2015 Lab 3
2. Lecture 8, 2/2/2015
3. An architecture, also called an instruction-set architecture (ISA), is the part of a processor design that you need to know to read/write assembler code. It includes the instruction set and the characteristics of registers. Examples: x86 (also called IA-32), IPF (also called IA-64: Itanium), x86_64.
4. The **microarchitecture** describes how the architecture is implemented. It includes the sizes of caches and the frequency at which the machine operates. It is not important for programming in assembler.

5. Components of importance to the assembler programmer
   
   (a) **Program counter** (PC): a register containing the address of the next instruction. Called EIP (x86) or RIP (x86_64)
   
   (b) **Registers**, used for heavily-accessed data, with names specific to the architecture. The set of all registers is sometimes called the **register file**.
   
   (c) **Condition codes** store information about the most recent arithmetic operation, such as “greater than zero”, useful for conditional branch instructions.
   
   (d) **Memory**, addressed by bytes, containing code (also called “text” in Unix), data, and a stack (to support procedures).

15 Steps in converting C to object code

1. Say the code is in two files: p1.c and p2.c
2. To compile: gcc p1.c p2.c -o p, which puts the compiled program in a file called p.
3. The gcc compiler first creates assembler files (stored in /tmp, but we can imagine they are called p1.s and p2.s).
4. It then runs the as assembler on those files, creating p1.o and p2.o.
5. It then runs the ld linker to combine those files with libraries (primarily the C library libc) to create an executable file p. Libraries provide code for malloc, printf, and others.
6. Some libraries are dynamically linked when the program starts to execute, saving space in the executable file and allowing the operating system to share code among processes.
7. Sample code:

```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```
8. Generated x86 assembler (using `-O1`)

```
sum:
  movl 8(%esp),%eax
  addl 4(%esp),%eax
  ret
```

9. Interpretation: `x` is at 8(%ebp); `y` is at 4(%ebp); `t` is in register %eax.

10. Output of `objdump`:

```
080483ed <sum>:
     8b 44 24 08    mov 0x8(%esp),%eax
80483f1:  03 44 24 04    add 0x4(%esp),%eax
80483f5:  c3       ret
```

11. Same thing with `gdb`, using command “disassemble sum”:

```
0x080483ed <+0>: mov 0x8(%esp),%eax
0x080483f1 <+4>: add 0x4(%esp),%eax
0x080483f5 <+8>: ret
```

12. Same thing with `gdb`, using command “x/9xb sum”:

```
0x80483ed <sum>: 0x8b 0x44 0x24 0x08 0x03 0x44 0x24 0x04
0x80483f5 <sum+8>: 0xc3
```

13. One can even disassemble .EXE files (from Win32 compilations) with `objdump`

16  **IA32 = x86 architecture**

1. 32-bit architecture.

2. You can compile for it even on an x86_64 with the `-m32` flag.
3. Registers

<table>
<thead>
<tr>
<th></th>
<th>32-bit</th>
<th>16-bit</th>
<th>8-bit</th>
<th>original purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>ax</td>
<td>ah/al</td>
<td>accumulator</td>
<td></td>
</tr>
<tr>
<td>ebx</td>
<td>bx</td>
<td>bh/bl</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>ecx</td>
<td>cx</td>
<td>ch/cl</td>
<td>counter</td>
<td></td>
</tr>
<tr>
<td>edx</td>
<td>dx</td>
<td>dh/dl</td>
<td>data</td>
<td></td>
</tr>
<tr>
<td>esi</td>
<td>source index</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edi</td>
<td>destination index</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>esp</td>
<td>sp</td>
<td>stack pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ebp</td>
<td>bp</td>
<td>base pointer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. Moving data: **movl source dest**

5. Operand types

(a) **Immediate**: integer constant, such as $0x400$ or $-533$. The actual constant is represented in 1, 2, or 4 bytes, depending on size; the assembler chooses the right representation. The source may be immediate, but not the destination.

(b) **Register**: any of the 8 integer registers, such as: `%ecx` (although `%esp` and `%ebp` have special purposes). Either source or destination or both may be register.

(c) **Memory**: 4 bytes of memory whose first byte is addressed by any register, such as `(%eax)` (note the parentheses). Either source or destination, but not both, may be memory.

(d) **Displacement**: 4 bytes of memory whose first byte is addressed by any register plus some constant, such as `8(%eax)`. Either source or destination, but not both, may be memory or displacement.

6. Example in C: Swap

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

7. Same thing in assembler
pushl %ebp          # save base pointer
movl %esp,%ebp      # new base pointer
pushl %ebx          # save old contents of %ebx
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *t0 = *xp
movl (%ecx), %eax   # eax = *t1 = *yp
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
popl %ebx           # restore %ebx
popl %ebp           # restore %ebp
ret                 # return

17 More complex memory-addressing modes

(a) We saw memory and displacement.
(b) This is the most general form: $D(Rb,Ri,S)$
   i. $D$ is the displacement, in bytes, such as 1, 2, 80.
   ii. $Rb$ is the base register: any of the 8 integer registers.
   iii. $Ri$ is the index register, any register by %esp, and you most likely don’t want to use %ebp, either
   iv. $S$ is a scale, which is any of 1, 2, 4, or 8.
(c) The value it references is Mem[Reg[Reg[Reg[Rb]]]+S*Reg[Ri]+D].
(d) Example
   i. %edx: 0xf000
   ii. %ecx: 0x0100
   iii. 0x8(%edx): 0xf000 + 0x8 = 0xf008
   iv. (%edx,%ecx): 0xf000 + 0x0100 = 0xf100
   v. (%edx,%ecx,4): 0xf000 + 4*0x0100 = 0xf400
   vi. 0x80(,%edx,2): 2*0xf000 + 0x80 = 0x1e080

18 Address computation without referencing

(a) One can compute an address and save it without actually referencing it.
(b) leal src, dest
(c) Load Effective Address of src and put it in dest.
(d) Purpose: translate \( p = \&x[i] \)
(e) Purpose: compute arithmetic expressions like \( x+k*y \) where \( k \) is 1, 2, 4, or 8.
   i. Example: \( x*12 \)
   ii. leal (%eax,%eax,2), %eax \# x = x+2\times
       sall $2, %eax \# x = x \ll 2

19 Arithmetic operations

1. Two-operand instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>dest = dest + src</td>
</tr>
<tr>
<td>subl</td>
<td>dest = dest − src</td>
</tr>
<tr>
<td>imull</td>
<td>dest = dest \times src</td>
</tr>
<tr>
<td>sall</td>
<td>dest = dest \ll src</td>
</tr>
<tr>
<td>sarl</td>
<td>dest = dest \gg src (arithmetic)</td>
</tr>
<tr>
<td>shrl</td>
<td>dest = dest \gg src (logical)</td>
</tr>
<tr>
<td>xorl</td>
<td>dest = dest \oplus src (bitwise)</td>
</tr>
<tr>
<td>andl</td>
<td>dest = dest \land src (bitwise)</td>
</tr>
<tr>
<td>orl</td>
<td>dest = dest \lor src (bitwise)</td>
</tr>
</tbody>
</table>

2. Lecture 10, 2/6/2015

3. Be careful of parameter order for asymmetric operations.

4. There is no distinction between signed and unsigned integers.

5. One-operand instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl</td>
<td>dest = dest + 1</td>
</tr>
<tr>
<td>decl</td>
<td>dest = dest − 1</td>
</tr>
<tr>
<td>negl</td>
<td>dest = −dest</td>
</tr>
<tr>
<td>notl</td>
<td>dest = \neg dest (bitwise)</td>
</tr>
</tbody>
</table>

6. Example
```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

7. Result of compilation

```
pushl %ebp        # save base pointer
movl %esp,%ebp   # new base pointer
movl 8(%ebp), %ecx # $c = x
movl 12(%ebp), %edx # $d = y
leal (%edx,%edx,2), %eax # $a = 3y
sall $4, %eax     # $a = 48y [t4]
leal 4(%ecx,%eax), %eax # $a = x + 48y + 4 [t5]
addl %ecx, %edx   # $d = x + y [t1]
addl 16(%ebp), %edx # $d = x + y + z [t2]
imull %edx, %eax  # $a = (x+y+z)*(x+48y+4)
popl %ebp         # restore base pointer
ret                # return $a [rval]
```

8. Optimization converts multiple expressions to a single statement, and a single expression might require multiple instructions.

9. The compiler generates the same code for \((x+y+z)*(x+4+48*y)\).

10. Example

```c
int logical(int x, int y)
{
    int t1 = x ^ y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7; // 8185
    int rval = t2 & mask;
    return rval;
}
```
11. Result of compilation

```assembly
pushl %ebp  # save base pointer
movl %esp,%ebp  # new base pointer
movl 12(%ebp),%eax  # $a = y
xorl 8(%ebp),%eax  # $a = y ^ x [t1]
sarl $17,%eax  # $a = (y ^ x) >> 17 [t2]
andl $8185,%eax  # $a = t2 & mask [rval]
popl %ebp  # restore base pointer
ret  # return
```

20 Control based on condition codes

1. The four condition codes, each Boolean
   (a) CF: Carry flag
   (b) ZF: Zero flag
   (c) SF: Sign flag
   (d) OF: Overflow flag (for signed integers)

2. Arithmetic operations implicitly set these flags, but the lea instruction does not set them.

3. Example: addition $t = a + b$
   (a) sets CF if there is a carry from most significant bit
   (b) sets ZF if the $t$ is zero
   (c) sets SF if the top bit of $t$ is set ($t < 0$)
   (d) sets OF if there is a two’s complement overflow:
       $(a > 0 \land b > 0 \land t < 0) \lor (a < 0 \land b < 0 \land t > 0)$

4. The compare instruction (`cmpl b, a`) also sets the flags; it’s like computing $a - b$ without modifying the destination.
   (a) sets CF if there is a carry from most significant bit
   (b) sets ZF if $a = b$
   (c) sets SF if $a - b < 0$
   (d) sets OF if there is a two’s complement overflow:
       $(a > 0 \land b < 0 \land (a - b) < 0) \lor (a < 0 \land b > 0 \land (a - b) > 0)$
5. The test instruction (\texttt{testl b, a}) also sets the flags; it’s like computing \(a \& b\) without modifying the destination. Usually, one of the two operands is a mask.

(a) sets \(ZF\) if \(a \& b = 0\)

(b) sets \(SF\) if \(a \& b < 0\)

6. Many instructions in the \texttt{setxx dest} family test the condition codes and set the destination (a single byte) to 0 or 1 based on the result.

\begin{align*}
\text{sete} & \quad ZF & \text{Equal/Zero} \\
\text{setne} & \quad \neg ZF & \text{Not Equal / Not Zero} \\
\text{sets} & \quad SF & \text{Negative} \\
\text{setns} & \quad \neg SF & \text{Nonnegative} \\
\text{setg} & \quad \neg (SF \oplus OF) \wedge \neg ZF & \text{Greater (Signed)} \\
\text{setge} & \quad \neg (SF \oplus OF) & \text{Greater or Equal (Signed)} \\
\text{setl} & \quad (SF \oplus OF) & \text{Less (Signed)} \\
\text{setle} & \quad (SF \oplus OF) \vee ZF & \text{Less or Equal (Signed)} \\
\text{seta} & \quad \neg CF \wedge \neg ZF & \text{Above (unsigned)} \\
\text{setb} & \quad CF & \text{Below (unsigned)}
\end{align*}

7. Many instructions in the \texttt{jxx dest} family jump depending on the condition codes.

\begin{align*}
\text{jmp} & \quad \text{true} & \text{Unconditional} \\
\text{je} & \quad ZF & \text{Equal/Zero} \\
\text{jne} & \quad \neg ZF & \text{Not Equal / Not Zero} \\
\text{js} & \quad SF & \text{Negative} \\
\text{jns} & \quad \neg SF & \text{Nonnegative} \\
\text{jg} & \quad \neg (SF \oplus OF) \wedge \neg ZF & \text{Greater (Signed)} \\
\text{jge} & \quad \neg (SF \oplus OF) & \text{Greater or Equal (Signed)} \\
\text{jl} & \quad (SF \oplus OF) & \text{Less (Signed)} \\
\text{jle} & \quad (SF \oplus OF) \vee ZF & \text{Less or Equal (Signed)} \\
\text{ja} & \quad \neg CF \wedge \neg ZF & \text{Above (unsigned)} \\
\text{jb} & \quad CF & \text{Below (unsigned)}
\end{align*}

8. Lecture 11, 2/9/2015

9. Example
1 int absdiff(int x, int y)
2 {
3    int result;
4    if (x > y) {
5        result = x-y;
6    } else {
7        result = y-x;
8    }
9    return result;
10 }

absdiff:
    pushl %ebp               # save base pointer
    movl %esp,%ebp           # new base pointer
    movl 8(%ebp), %edx      # d = x
    movl 12(%ebp), %eax     # a = y
    cmpl %eax, %edx          # x <> y ?
    jle .L6                  # jump if x <= y
    subl %eax, %edx          # x = x - y
    movl %edx, %eax          # a = x - y
    jmp .L7                  # jump
.L6:
    subl %edx, %eax          # a = y - x
.L7:
    popl %ebp                # restore %ebp
    ret                      # return

10. Example:
1 int absDiff(int x, int y)
2 {
3    int result;
4    if (x <= y) goto elsepoint;
5    result = x-y;
6    goto exitpoint;
7 elsepoint:
8    result = y-x;
9 exitpoint:
10    return result;
absDiff:
    pushl %ebp  # save base pointer
    movl %esp,%ebp  # new base pointer
    movl 8(%ebp), %edx  # d = x
    movl 12(%ebp), %eax  # a = y
    cmpl %eax, %edx  # x <> y?
    jle .L6  # jump if x <= y
    subl %eax, %edx  # d = x - y
    movl %edx, %eax  # a = x - y
    jmp .L7  # goto exitpoint
.L6:  # elsepoint
    subl %edx, %eax  # a = y - x
.L7:  # exitpoint
    popl %ebp  # restore %ebp
    ret  # return

11. C can sometimes use a single conditional expression to handle such cases:

```c
val = x>y ? x-y : y-x;
```

21  **do while loops**

1. C code to count how many 1’s in a parameter

```c
int countOnes(unsigned x) {
    int result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

2. Same thing, represented with **goto**
int countOnes(unsigned x) {
    int result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x)
            goto loop;
    return result;
}

3. Partial assembler listing

    movl 8(%esp),%edx  # d = x
    movl $0, %ecx     # result = 0
.L2:                # loop:
    movl %edx, %eax   # a = x
    andl $1, %eax     # a = x & 1
    addl %eax, %ecx   # result += x & 1
    shrl $1, %edx     # x >>= 1
    jne .L2           # If !0, goto loop
    movl %ecx, %eax   # a = result

4. for loops are very similar

5. The compiler can generate better code by replacing the unconditional jump at the end of the loop with a conditional jump.

22 Procedures

1. In order to handle recursion, languages are compiled to use a stack.
2. Each invocation of a procedure pushes a new frame on the stack.
3. When the procedure returns, the frame is popped and the space it occupied is available for another procedure.
4. The frame contains storage private to this instance of the procedure.

    (a) return address
(b) parameters
(c) local variables
(d) temporary locations (that don’t fit in registers)

6. On the x86, the %ebp register points to the start of the frame, and the %esp register points to the current top of the stack.

7. The stack (for Unix, at least), grows downward.

(a) pushl src subtracts 4 from %esp and writes the operand at the new address.
(b) popl dest puts (%esp) in the destination and then adds 4 to %esp.
(c) call label pushes the return address and then jumps to the label. The return address is the address of the instruction after the call.
(d) ret pops the return address and then jumps to it.

8. Example:

```
804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax
...
8048591: c3 ret
```

when | %esp | %eip
---|---|---
before call | 0x108 | 0x804854e
after call | 0x104 | 0x8048b90
before return | 0x104 | 0x8048591
after return | 0x108 | 0x8048553

9. Linux C frame contents, starting at bottom (right after caller’s frame)

(a) return address (placed by call)
(b) old %ebp
(c) saved registers and local variables
(d) parameters for the next call (“argument build”), last parameter first
(e) %esp points to the last parameter.

10. Lecture 13, 2/13/2015
11. Linkage at the calling point for `swap(&course1, &course2);`

```
subl $8,%esp         # make room for two parameters
movl $course2, 4%esp) # parameter 2
movl $course1, (%esp) # parameter 1
call swap            # call
```

12. When a procedure is called, after the procedure pushes `%ebp`, the following values apply.

<table>
<thead>
<tr>
<th>location</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(%esp)</td>
<td>old <code>%ebp</code></td>
</tr>
<tr>
<td>4(%esp)</td>
<td>return address</td>
</tr>
<tr>
<td>8(%esp)</td>
<td>first parameter</td>
</tr>
<tr>
<td>12(%esp)</td>
<td>second parameter</td>
</tr>
</tbody>
</table>

13. If a local register, such as `%ebx`, has then been pushed, the `%esp` register is not as good a base as the `%ebp` register for accessing parameters.

### 23 Register-saving conventions

1. The compiler writer determines what registers are meant to survive procedure calls ("non-volatile registers") and which can be used for temporary storage by the procedure ("volatile registers").

2. This convention prevents one procedure call from corrupting another’s data.

3. Say A (the caller) is calling B (the callee).

   (a) If A has been using a volatile register, it must save it (on the stack) before calling B, and pop it when B returns. This situation is called **caller save**.

   (b) If B needs to use a non-volatile register, it should save it (on the stack) before doing its work and pop it before returning. This situation is called **callee save**.

4. The convention that gcc follows for the x86:

   (a) `%eax, %ecx, %edx` are volatile (caller-save) general-purpose registers.
(b) %ebx, %esi, %edi are non-volatile (callee-save) general-purpose registers.
(c) %esp and %ebp are non-volatile (callee-save) special-purpose registers.

5. Example

```c
int bitCount(unsigned x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + bitCount(x >> 1)
}
```

bitCount:
```asm
pushl %ebp          # save %ebp (non-volatile)
movl %esp, %ebp     # new %ebp
pushl %ebx          # save b (non-volatile)
subl $4, %esp      # room for p1
movl 8(%ebp), %ebx # b = x
movl $0, %eax      # a = 0
testl %ebx, %ebx    # x ==? 0
je .L3             # if (x==0) jump
movl %ebx, %eax     # a = x
shrl %eax          # a = x >> 1
movl %eax, (%esp)   # pl = x >> 1
call bitCount       # a = bitCount(pl)
movl %ebx, %edx     # d = x
andl $1, %edx       # d = x & 1
leal (%edx,%eax), %eax # a = (x&1) + bitCount(pl)
.L3:
addl $4, %esp      # remove pl
popl %ebx           # restore b
popl %ebp           # restore ebp
ret                 # return
```

24 Code for local variables, pointers

1. Lecture 14, 2/18/2015
2. Example

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}

void incrk(int *ip, int k) {
    *ip += k;
}
```

add3:
- pushl %ebp  # save old ebp
- movl %esp, %ebp  # new ebp
- subl $24, %esp  # allocate 24 bytes (6 "chunks")
- movl 8(%ebp), %eax  # a = x
- movl %eax, -4(%ebp)  # localx = x
- movl $3, 4(%esp)  # actual2 = 3
- leal -4(%ebp), %eax  # a = &localx
- movl %eax, (%esp)  # actual1 = &localx
- call incrk  # a = incrk(actual1, actual2)
- movl -4(%ebp), %eax  # a = localx
- addl $24, %esp  # deallocate 24 bytes
- popl %ebp  # restore ebp
- ret  # return

3. right before the call to incrk(), the stack has these “chunks”:

- x at 8(%ebp)
- return address to add3’s caller
- old ebp at (%ebp)
- localx at -4(%ebp)
- unused
- unused
- unused
- actual 2 at 4(esp)
- actual 1 at (esp)
25 x86_64 registers

1. Eight upgraded 64-bit registers, now with names starting with r instead of e, such as %rax.

2. Eight new 64-bit registers, called %r8 ... %r15.

3. To use only the lower bits of a register, append a suffix to the register name.

<table>
<thead>
<tr>
<th>suffix</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>8</td>
</tr>
<tr>
<td>w</td>
<td>16</td>
</tr>
<tr>
<td>d</td>
<td>32</td>
</tr>
</tbody>
</table>

4. Conventions
   (a) The first 6 parameters are passed, in reverse order, in %rdi, %rsi, %rdx, %rcx, %r8, and %r9.
   (b) The remaining parameters are pushed on the stack in call order.
   (c) non-volatile (callee-save) registers: %rbp, %rbx, %r12, %r13, %r14, and %r15.
   (d) Other registers are volatile (caller-save).

26 Data types

1. Integer
   (a) Can be stored in general registers or in memory.
   (b) Signed and unsigned work the same except for shift operations.
   (c) Suffix on instructions indicates how many bits are affected

<table>
<thead>
<tr>
<th>Intel</th>
<th>C</th>
<th>assembler</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>char</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>word</td>
<td>short</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>double word</td>
<td>int</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>quad word</td>
<td>long int</td>
<td>q</td>
<td>8 (x86_64)</td>
</tr>
</tbody>
</table>

2. Floating point
   (a) Can be stored in floating-point registers or in memory.

<table>
<thead>
<tr>
<th>Intel</th>
<th>C</th>
<th>assembler</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>float</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>l</td>
<td>8 (x86_64)</td>
</tr>
<tr>
<td>extended</td>
<td>long double</td>
<td>t</td>
<td>10/12/16 (x86_64)</td>
</tr>
</tbody>
</table>
27 Arrays

1. Lecture 15, 2/23/2015

2. C declaration: \( T \) myArray\([L] \), where \( T \) is some type and \( L \) is the number of elements (the first is number 0).

3. Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes.

4. Examples
<table>
<thead>
<tr>
<th>declaration</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>char string[12]</td>
<td>12</td>
</tr>
<tr>
<td>int val[5]</td>
<td>20</td>
</tr>
<tr>
<td>double a[3]</td>
<td>24</td>
</tr>
<tr>
<td>char *p[3]</td>
<td>12 (on x86)</td>
</tr>
<tr>
<td>char *p[3]</td>
<td>24 (on x86,64)</td>
</tr>
</tbody>
</table>

5. C syntax, given int val[5]; stored starting at location \( x \), containing 1, 2, 3, 4, 5.
<table>
<thead>
<tr>
<th>expression</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int*</td>
<td>x</td>
</tr>
<tr>
<td>val+1</td>
<td>int*</td>
<td>x+4</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int*</td>
<td>x+8</td>
</tr>
<tr>
<td>val[4]</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>garbage</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>val + i</td>
<td>int*</td>
<td>x+4i</td>
</tr>
</tbody>
</table>

6. Using the same type for several arrays
   1. #define ZLEN 5
   2. typedef int myArrayType[ZLEN];
   3. myArrayType cmu = { 1, 5, 2, 1, 3 };
   4. myArrayType mit = { 0, 2, 1, 3, 9 };
   5. myArrayType uky = { 9, 4, 7, 2, 0 };

   It’s possible, but not guaranteed, that the three arrays are consecutive in memory.

7. Simple example: return uky[dig];

   # assume d = uky
   # assume a = dig
   movl (%edx,%eax,4),%eax # a = uky[dig]
8. Loop example

```c
void zincr(myArrayType z) {
    int i;
    for (i = 0; i < ZLEN; i+=1)
        z[i] += 1;
}
```

```assembly
movl $0, %eax # a = 0
.L4: # loop:
    addl $1, (%edx,%eax,4) # z[i] += 1
    addl $1, %eax # i+=1
    cmpl $5, %eax # i:5
    jl .L4 # if < goto loop
```

28 Nested arrays

```c
#define PCOUNT 4
myArrayType pgh[PCOUNT] =
    {{1, 5, 2, 0, 6 },
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

2. The values are placed contiguously from some location $x$ to $x + 4 \times PCOUNT \times ZLEN - 1 = x + 79$. This ordering is guaranteed.

3. `pgh` is an array of 4 elements.

4. Each of those elements is an array of 5 sub-elements.

5. Each of those sub-elements is an integer occupying 4 bytes.

6. Equivalent declaration: `int pgh[PCOUNT][ZLEN]`;

7. C code: `return pgh[index]` (the return type is `int` *)

```assembly
# assume a = index
leal (%eax,%eax,4),%eax # a = 5 * index
leal pgh(,%eax,4),%eax # a = pgh + (20 * index)
```
8. **Lecture 16, 2/25/2015**

9. General addressing case: \( T A[R][C] \); defines a two-dimensional array of type \( T \) with \( R \) rows and \( C \) columns. Say type \( T \) requires \( k \) bytes. Then the location of \( A[i][j] \) is \( A + kiR + kj = A + k(iR + j) \).

10. Example:

```c
int getElement(int n, int x[n][n], int i, int j) {
    return a[i][j];
}
```

```assembly
movl 8(%ebp), %eax # n (param 1)
sall $2, %eax # a = 4n
movl %eax, %edx # d = 4n
imull 16(%ebp), %edx # d = 4 n * i
movl 20(%ebp), %eax # a = j (param 4)
sall $2, %eax # a = 4j
addl 12(%ebp), %eax # a = x + 4j
movl (%eax,%edx), %eax # a = *(x + 4j + 4n*i)
```

11. Walking down a column can be optimized by computing the address of the first element in the column, then repeatedly adding the stride (the number of bytes per row).

29 **Structures**

1. A **struct** is a contiguously allocated region of memory with named **fields**. Each field has its own type.

2. Example:

```c
struct rec {
    int y[3];
    int i;
    struct rec *n;
};
```

Memory layout of **rec**, starting at address \( x \), is based on offsets:

```
y   x
i   x+12
n   x+16
end x+20
```
3. The compiler knows all the offsets.

30 Linked lists

1. Example:

```c
void set_val(struct rec *r, int val) {
    while (r) {
        int i = r->i;
        r->y[i] = val;
        r = r->n;
    }
}
```

```asm
# assume d = r
# assume c = val
.L17:  # loop:
movl 12(%edx), %eax # a = r->i
movl %ecx, (%edx,%eax,4) # r->y[r->i] = val
movl 16(%edx), %edx # r = r->n
testl %edx, %edx # Test r
jne .L17 # If != 0 goto loop
```

31 Alignment

1. Example:

```c
struct S1 {
    char c;
    int y[2];
    double v; // uses 8 bytes
} *p;
```

2. The character c uses only 1 byte, so the array y starts at offset 1, and v at offset 9.

3. But the x86 advises that n-byte primitive data should start at an address divisible by n, that is, it should be aligned to such an address.

   (a) 1 byte (char): any address
(b) 2 bytes (short): address ends with $0_2$
(c) 4 bytes (int, void *): address ends with $00_2$
(d) 8 bytes (double): address ends with $000_2$ (Linux/x86 compilers choose $00_2$).
(e) 12 bytes (long double): Linux/x86 chooses $00_2$.

4. The x86_64 is stricter

(a) 8 bytes (double, void *): address ends with $000_2$
(b) 16 bytes (long double): Linux/x86 chooses $000_2$.

5. On some machines alignment is mandatory.

6. Motivation

(a) The CPU accesses memory in chunks of 4 or 8 bytes (architecture-dependent).
(b) It is inefficient to access a datum that crosses chunk boundaries.
(c) It is tricky to access a datum that crosses page boundaries (typically every 4KB).

7. The compiler can add *padding* to accomplish this requirement:

    c   x
    pad x+1 (pad of 3 bytes)
    y   x+4
    pad x+12 (pad of 4 bytes)
    v   x+16
    end x+24

8. The compiler can also re-order the fields (biggest first, for instance) to reduce the amount of padding.

9. The entire *struct* needs to be padded to a multiple of the largest primitive data within the *struct*, so that arrays of such *structs* work.

### 32  Midterm test 3/2/2015

### 33  Discussion of midterm test

1. Lecture 17, 3/4/2015
34 Unions

1. Snow day: 3/6/2015
2. Lecture 18, 3/9/2015
3. A union type is like a struct, but the fields all start at offset 0, so they overlap.
4. This method allows us to view the same data as different types.
5. It also lets us look at individual bytes of numeric types to see the byte ordering.
6. Example:

```c
union {
    char c[8]; // 8*1 = 8 bytes
    short s[4]; // 4*2 = 8 bytes
    int i[2]; // 2*4 = 8 bytes
    long l[1]; // 1*8 = 8 bytes (on i86_64)
    float f[2]; // 2*4 = 8 bytes
} dw;

dw.f[0] = 3.1415;
printf("as float: %f; as integer: %d; as two shorts: %d, %d\n",
    dw.f[0], dw.i[0], dw.s[0], dw.s[1]);
```

Result:

```
as float: 3.141500; as integer: 1078529622;
as two shorts: 3670, 16457
```

35 Linux x86 memory layout

1. Simplified version of allocation of 4GB virtual space

<table>
<thead>
<tr>
<th>Start Location</th>
<th>Name</th>
<th>Purpose</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unused</td>
<td>prevent errors</td>
<td>no access</td>
</tr>
<tr>
<td>0x08000000</td>
<td>text</td>
<td>program</td>
<td>read-only</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>initialized data</td>
<td>read, write; static size</td>
</tr>
<tr>
<td></td>
<td>bss</td>
<td>uninitialized data</td>
<td>read, write; static size</td>
</tr>
<tr>
<td></td>
<td>heap</td>
<td>allocatable data</td>
<td>read, write; grows up</td>
</tr>
<tr>
<td></td>
<td>stack</td>
<td>activation records</td>
<td>read, write; grows down</td>
</tr>
<tr>
<td>0xC0000000</td>
<td>kernel</td>
<td>kernel code, shared</td>
<td>no access</td>
</tr>
</tbody>
</table>
2. The limit program shows per-process limitations; by default, for instance, the stack is limited to 8MB.

36 Buffer overflow

1. Underlying problem: library functions do not check sizes of parameters, because C array types don’t specify length.

2. Lecture 19, 3/11/2015

3. Which functions: `gets()`, `strcpy()`, `strcat()`, `scanf()`, `fscanf()`, `sscanf()`.

4. Effect of overflowing a local array (on the stack): overwriting return address.
   
   (a) If the return is to an address not in text or stack space, causes a segmentation fault.
   (b) The return address can be to code on the stack that is part of the overflowing buffer, leading to execution of arbitrary code.

5. Internet worm (November 1988): the `fingerd` program used `gets()` to read a command-line parameter; by exploiting a buffer overflow, the worm got `fingerd` to run a root shell with a TCP connection to the attacker.

6. There are hundreds of other examples.

7. Avoiding vulnerability

   (a) Use library routines that limit lengths: `fgets()`, `strncpy()`, `scanf(...%ns...)`.

   (b) Randomized stack offsets: allocate a random amount of stack space as the program starts. Then the attacker cannot guess the start of the buffer, so it is harder to fake the return address to jump into the buffer.

   (c) Nonexecutable segments: On the x86, anything readable is executable, including the stack. On the x86_64, there is separate executable permission.

   (d) Stack canaries: put a canary value on stack just beyond each buffer; check for corruption as part of linkage during return. In
gcc, use `-fstack-protector` (adds code to evidently suspicious routines) or `-fstack-protector-all` (adds code to all routines)

8048654: 65 a1 14 00 00 00 mov %gs:0x14,%eax
804865a: 89 45 f8 mov %eax,0xffffffff8(%ebp)
804865d: 31 c0 xor %eax,%eax
...
8048672: 8b 45 f8 mov 0xffffffff8(%ebp),%eax
8048675: 65 33 05 14 00 00 00 xor %gs:0x14,%eax
804867c: 74 05 je 8048683 <echo+0x36>
804867e: e8 a9 fd ff ff call 804842c <FAIL>

8. Malware
(a) Worm: a program that can run by itself, propagates a fully working version to other computers.
(b) Virus: code that adds itself to other programs, but cannot run independently.

37 Linking
1. Lecture 20, 3/13/2015
2. Basic idea: combine results of one or more independent compilations with libraries.
3. The individual compiled results are called **relocatable object files**; the Unix convention is that their names end “.o”.
4. Benefits
   (a) The programmer can decompose work into small files, promoting modularity.
   (b) Experts (hah!) can program commonly used functions and place them in libraries (C library, math library, ...).
   (c) Changes to one file do not require recompiling the entire suite of files.
   (d) The linker can pick up only those functions that are used from a library, so the entire library need not be part of the executable.
5. Symbol resolution
(a) Programs define symbols and reference them:

```c
void swap() {...} // exported global identifier
extern int myGlobal; // imported global identifier
int myGlobal; // also local, so exported, too
swap(&myGlobal, &myLocal); // reference identifiers
```

(b) The compiler uses an internal data structure called the **symbol table** to keep track of all identifiers.

(c) The symbol table, indexed by the identifier, includes information such as type, location, and global/local flag.

(d) The compiler includes the global symbols as part of the object file it outputs.

(e) The object file marks any reference to an imported global as “dangling”.

(f) The linker resolves dangling references by connecting them to the proper identifier in another object file.

(g) The compiler has already resolved references to local symbols.

(h) It’s a link-time error if the linker discovers multiple possible resolutions.

(i) If desired, the linker then consults libraries to resolve any still-dangling references by adding more object files.

(j) If the linker can resolve all dangling references, the result is an **executable file** that the operating system can load and run.

(k) Otherwise, the result is an **object file** that can be used for further linking steps.

(l) The early Unix convention was to call the executable file “`a.out`”; now it usually has a name without an extension.

6. Shared object files

   (a) If desired, the linker can store its result as a **shared object file** (conventional extension “`.so`”), which the program can load into memory dynamically (typically when the program starts) in such a way that it is shared among all processes that need it.

   (b) Libraries are usually shared object files.

   (c) When the linker resolves a identifier by referring to a shared object file, it leaves it dangling (but resolved); full resolution happens when the shared object file is loaded into memory.
(d) Windows calls shared object files **Dynamic Link Libraries** (DLLs).

7. Relocation

(a) The linker combines the object files into a single file.

(b) The linker **relocates** global identifiers exported from each object file to account for the space occupied by the identifiers in previous object files in its list.

(c) The linker updates all references to relocated global identifiers.