1 Intro

Class 1, 1/10/2018

1. Handout 1 — My names
2. Plagiarism — read aloud
3. E-mail list: cs270@cs.uky.edu
4. Labs: 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.
5. Projects are available at the course website, https://www.cs.uky.edu/~raphael/courses/CS270.html. First project — Review of the C language
6. Accounts in MultiLab if you want; every student has a virtual machine as well, at name@name.netlab.uky, where name is your LinkBlue account name. The first lab will acquaint you with this facility.

2 A brief introduction to systems programming

2. Programs are written in C, which can do low-level manipulation of data, but is error-prone.
3. Files: `open()`, `read()/write()`, `close()`
4. Memory: `malloc()` gives a pointer to a character array.

### 3 Software tools

<table>
<thead>
<tr>
<th>Use (client)</th>
<th>Spec</th>
<th>Programmer</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Class 2, 1/12/2018] Lab 1

### 4 Abstraction and reality

1. [Class 3, 1/17/2018]
2. Most CS and CE courses emphasize abstraction; it matches how we think, and it lets us hide implementation details and complexity.
3. 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.
4. Examples of hardware limits
   
   (a) C `int` is not an integer: $50000 \times 50000 = 2500000000$, but the int has value $-1794967296$.
   
   (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
int from[BIG*BIG], to[BIG*BIG];

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];
} // copy

int main() {
    copyij(from, to);
    return (0);
} // main
```

One experiment shows that in the order given, user time is 0.3 seconds; with interchanged order, user time is 1.4 seconds.

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

6. 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.

5 Memory-referencing bugs

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

1. Array references out of bounds. Example (the actual result is architecture-specific; the results shown are on the x86_64)
```c
void fun(int index) {
    printf("fun(%d): \n", index);
    double d[1] = {3.14};
    long a[2];
    a[index] = 9223372036854775803L;
    printf("%lf\n", d[0]);
}

int main() {
    fun(0); // 3.14
    fun(1); // 3.14
    fun(2); // 3.14
    fun(-1); // 3.14
    fun(-2); // nan (not a number)
    fun(-3); // 3.14
    fun(3); // 3.14 then fault during the return
         // *** stack smashing detected ***: ./fun terminated
    return(0);
}
```

Reason: \(a\) and \(d\) are adjacent on the stack; \(d\) at a lower address than \(a\). When you go before the start of \(a\), you might modify \(d\); when you go beyond the end of \(a\), you might access saved state on the stack, ruining the return address.

2. Dereferencing invalid pointers
3. Improper allocating and deallocating memory regions
4. Unfortunately, the symptoms may be unrelated to the causes, and the effect might be visible only long after it is generated.
5. Some languages prevent such errors: Java, JavaScript, Python, Perl, Ruby, but they are not usually used for systems programming.
6. There are tools to help you detect referencing errors (such as valgrind).

6 Binary representation

1. Bits are represented by two voltages, one for 0 and another for 1. A typical representation would be 0.3V 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 \(101.01_2\). 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. \(285.3 = 11D.4CCCCCCC\ldots_{16}\)

(c) I used the `bc` program to compute these values:

```
bc
scale = 10
obase = 16
285.3000
```

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 \(11111111_2 = \text{FF}_{16}\)).

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 \((-80000000_{16} \ldots 7FFFFFFF_{16})\).

5. [Class 4, 1/19/2018]: Lab 2

6. [Class 5, 1/22/2018]

7. Modern computers represent signed integers in a format called 2’s complement.

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

(b) In signed form, if the high bit is 0, the number is positive, in the range \(0 \ldots 2^{n-1} - 1\).

(c) In signed form, if the high bit is 1, the number is negative, in the range \(-2^{n-1} \ldots -1\).

(d) To negate a number:
i. flip all the bits (0 becomes 1, 1 becomes 0)
ii. add 1 (ignoring any carry out of the most significant bit).

(e) Example (n = 4): 6_{10} = 0110_2. Flipped: 1001_2. Final result: 1010_2.
(f) To negate again: −6_{10} = 1010_2. Flipped: 0101_2. Final result: 0110_2.
(g) Negating 0 gives 0.
(h) The most negative number (when n = 4) is −8_{10} = 1000_2. Negating this number leaves it unchanged.

8. It’s 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} ≈ 10^3, so 2^{32} = 2^2 \times 2^{30} = 4 \times (2^{10})^3 ≈ 4 \times (10^3)^3 = 4 \times 10^9 = 4 \text{ 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}.

9. What is the largest signed integer you can store in 16 bits? (Answer: 2^{15} − 1 = 32767)

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

7 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>
8 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 \texttt{od} program to show a file in bytes, characters, integers, ... For instance:

\begin{verbatim}
    od -t x4 -t o1 /etc/hosts | head
\end{verbatim}

6. You can also use a program:

\begin{verbatim}
1 typedef unsigned char *pointer;
2 void show_bytes(pointer start, int len){
3     int i;
4     for (i = 0; i < len; i += 1) {
5         printf("%p	0x%.2x\n", start+i, start[i]);
6     }
7     printf("\n");
8 } // show_bytes

9 int main() {
10    int a = 15213;
11    printf("int a = %d (0x%08x)\n", a, a);
12    show_bytes((pointer) &a, sizeof(int));
13 } // main
\end{verbatim}

The output is:

\begin{verbatim}
    int a = 15213 (0x00003b6d);
    0x7ffffff9d80c 0x6d
    0x7fffffff9d80d 0x3b
    0x7fffffff9d80e 0x00
    0x7fffffff9d80f 0x00
\end{verbatim}
9 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_32 memory is limited to 3GB; on our machine, the x86_64, it is limited to 128TB.)

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. [Class 6, 1/24/2018]

8. 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.

9. 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.

10. 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.

11. 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.
10 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.

11 Boolean algebra

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

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

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

4. When one operates on bytes (or larger chunks such as integers) with Boolean functions, they are applied bitwise. Examples:
5. **Class 7, 1/26/2018**: Lab 3

6. **Class 8, 1/29/2018**

7. Instead of interpreting 32 bits as an integer, we can interpret it as a subset of \{0, \ldots, 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.

8. One can use the Boolean operators in C and apply them to any integral data type: `char`, `short`, `int`, `long`, `long long`.

9. 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.

12 **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 (for signed types only). The same warning applies.
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.
     (result goes to standard out)
   - `-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, also `s` for size, also `fast`)

14 Tools to inspect compiled code

1. `objdump -d filename`: disassembles `filename`

2. `gdb filename`: runs the debugger on `filename`; can disassemble

3. `nm filename`: shows location and type of identifiers in `filename`

4. `strings filename`: shows all the ASCII strings in `filename`.

5. `od filename`: displays `filename` in numeric or character format.


7. `dissy filename`: graphical tool to inspect `filename`. You can install `dissy` by using `apt-get`.

15 Machine basics

1. 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 x86_32, IA-32), IPF (also called IA-64: Itanium), x86_64.

2. 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.

3. 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).

16 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. Output of objdump -d on compiled (-O0) file:

   ```
   4004ed: 55      push  %rbp
   4004ee: 48 89 e5  mov  %rsp,%rbp
   ```
4004f1: 89 7d ec mov %edi,-0x14(%rbp)
4004f4: 89 75 e8 mov %esi,-0x18(%rbp)
4004f7: 8b 45 e8 mov -0x18(%rbp),%eax
4004fa: 8b 55 ec mov -0x14(%rbp),%edx
4004fd: 01 d0 add %edx,%eax
4004ff: 89 45 fc mov %eax,-0x4(%rbp)
400502: 8b 45 fc mov -0x4(%rbp),%eax
400505: 5d pop %rbp
400506: c3 retq

9. Interpretation: x is in %edi, then -0x14(%rbp), then %edx; y is in %esi, then -0x18(%rbp), then %eax; t is in %eax, then -4(%rbp), then %eax.

10. Same thing with **gdb**, using command “disassemble sum”:

    0x00000000004004ed <+0>:  push %rbp
    0x00000000004004ee <+1>:  mov %rsp,%rbp
    0x00000000004004f1 <+4>:  mov %edi,-0x14(%rbp)
    0x00000000004004f4 <+7>:  mov %esi,-0x18(%rbp)
    0x00000000004004f7 <+10>: mov -0x18(%rbp),%eax
    0x00000000004004fa <+13>: mov -0x14(%rbp),%edx
    0x00000000004004fd <+16>: add %edx,%eax
    0x00000000004004ff <+18>: mov %eax,-0x4(%rbp)
    0x0000000000400502 <+21>: mov -0x4(%rbp),%eax
    0x0000000000400505 <+24>: pop %rbp
    0x0000000000400506 <+25>: retq

11. Same thing with **gdb**, using command “x/25xb sum”:

    0x4004ed <sum>:  0x55 0x48 0x89 0xe5 0x89 0x7d 0xec 0x89
    0x4004f5 <sum+8>: 0x75 0xe8 0xb 0x45 0xe8 0x8b 0x55 0xec
    0x4004fd <sum+16>: 0x01 0xd0 0x89 0x45 0xcfc 0x8b 0x45 0xcfc
    0x400505 <sum+24>: 0x5d 0xc3

12. Compiling with **-O1**:

    4004ed:  8d 04 37  lea (%rdi,%rsi,1),%eax
    4004f0:  c3     retq

13. [Class 9, 1/31/2018]

14. One can even disassemble .EXE files (from Win32 compilations) with objdump
17 Intel/AMD architectures

1. Architecture: what a programmer needs to write assembler/machine code.

   (a) Machine code: byte-level programs that the CPU executes
   (b) Assembler code: a text representation of the machine code
   (c) x86 32-bit architecture (since 1985; 2nd edition of our textbook)
   (d) You can compile for 32 bits even on an x86_64 with the -m32 flag.
   (e) x86-64: 64-bit architecture (since 2003; 3rd edition of our textbook)
   (f) Others: ARM (used in mobile phones), MIPS, Sparc, ...

2. Registers

   non-volatile means the callee must preserve the value.

<table>
<thead>
<tr>
<th>64-bit</th>
<th>32-bit</th>
<th>16-bit</th>
<th>8-bit</th>
<th>original purpose</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>eax</td>
<td>ax</td>
<td>ah/al</td>
<td>accumulator</td>
<td>C</td>
</tr>
<tr>
<td>rbx</td>
<td>ebx</td>
<td>bx</td>
<td>bh/bl</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>rcx</td>
<td>ecx</td>
<td>cx</td>
<td>ch/cl</td>
<td>counter</td>
<td></td>
</tr>
<tr>
<td>rdx</td>
<td>edx</td>
<td>dx</td>
<td>dh/dl</td>
<td>data</td>
<td></td>
</tr>
<tr>
<td>rsi</td>
<td>esi</td>
<td>sil</td>
<td></td>
<td>source index</td>
<td></td>
</tr>
<tr>
<td>rdi</td>
<td>edi</td>
<td>dil</td>
<td></td>
<td>destination index</td>
<td></td>
</tr>
<tr>
<td>rsp</td>
<td>esp</td>
<td>sp</td>
<td>spl</td>
<td>stack pointer</td>
<td></td>
</tr>
<tr>
<td>rbp</td>
<td>ebp</td>
<td>bp</td>
<td>bpl</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td>r8d</td>
<td></td>
<td></td>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td>r9d</td>
<td></td>
<td></td>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td>r9d</td>
<td></td>
<td></td>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td>r9d</td>
<td></td>
<td></td>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td>r9d</td>
<td></td>
<td></td>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>r13</td>
<td>r9d</td>
<td></td>
<td></td>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>r14</td>
<td>r9d</td>
<td></td>
<td></td>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>r15</td>
<td>r15d</td>
<td></td>
<td></td>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>rip</td>
<td>eip</td>
<td>ip</td>
<td></td>
<td>instruction pointer</td>
<td></td>
</tr>
</tbody>
</table>

3. Moving data: movq source, dest The q means “quad”, which means 64 bits. One can also use movl, where l means “long”, which means 32 bits.

4. 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 16 integer registers, such as: `%rcx` (although `%rsp` and `%rbp` have special purposes). Can also be the lower 4, 2, or 1 bytes of a register (using names above). Either source or destination or both may be register.

(c) **Memory**: 8 bytes of memory, whose first byte is addressed by any register, such as `(%rax)` (note the parentheses). Either source or destination, but not both, may be memory.

(d) **Displacement**: 8 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.

5. Example in C: Swap

```c
void swap(long x[2])
{
    long t0 = x[0]; // or *x
    long t1 = x[1];
    x[0] = t1;
    x[1] = t0;
}
```

6. Same thing in assembler

```assembly
1 movq (%rdi), %rax        # t0 = x[0]
2 movq 0x8(,%rdi), %rdx    # t1 = x[1]
3 movq %rax, 0x8(%rdi)     # x[1] = t0
4 movq %rdx, (%rdi)        # x[0] = t1
5 retq                      # return
```

18 More complex memory-addressing modes

1. We saw memory and displacement.

2. This is the most general form: $D(Rb,Ri,S)$

   (a) $D$ is the displacement, in bytes, such as 1, 2, 80.
(b) \( R_b \) is the base register: any of the 8 integer registers.
(c) \( R_i \) is the index register, any register by \%esp, and you most likely don’t want to use \%ebp, either
(d) \( S \) is a scale, which is any of 1, 2, 4, or 8.

3. The value it references is \( \text{Mem}[\text{Reg}[R_b]+S*\text{Reg}[R_i]+D] \).

4. Example (using 32-bit registers)
   (a) \%edx: 0xf000
   (b) \%ecx: 0x0100
   (c) 0x8(\%edx): 0xf000 + 0x8 = 0xf008
   (d) (%edx,%ecx): 0xf000 + 0x0100 = 0xf100
   (e) (%edx,%ecx,4): 0xf000 + 4*0x0100 = 0xf400
   (f) 0x80(,%edx,2): 2*0xf000 + 0x80 = 0x1e080

19 Address computation without referencing

1. [Class 10, 2/2/2018]
2. One can compute an address and save it without actually referencing it.
3. lea src, dest
4. Load Effective Address of src and put it in dest.
5. Purpose: translate \( p = \&x[i] \)
6. Purpose: compute arithmetic expressions like \( x+k*y \) where \( k \) is 1, 2, 4, or 8.
7. Example: \( x*12 \)
   ```asm
   1 leal (%eax,%eax,2), %eax  # x = x+2x
   2 sall $2, %eax             # x = x * 4
   ```
20 Arithmetic operations

1. Two-operand instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>dest = dest + src</td>
</tr>
<tr>
<td>subq</td>
<td>dest = dest - src</td>
</tr>
<tr>
<td>imulq</td>
<td>dest = dest × src</td>
</tr>
<tr>
<td>salq</td>
<td>dest = dest &lt;&lt; src</td>
</tr>
<tr>
<td>sarq</td>
<td>dest = dest &gt;&gt; src (arithmetic)</td>
</tr>
<tr>
<td>shriq</td>
<td>dest = dest &gt;&gt; src (logical)</td>
</tr>
<tr>
<td>xorq</td>
<td>dest = dest ⊕ src (bitwise)</td>
</tr>
<tr>
<td>andq</td>
<td>dest = dest ∧ src (bitwise)</td>
</tr>
<tr>
<td>orq</td>
<td>dest = dest ∨ src (bitwise)</td>
</tr>
</tbody>
</table>

2. Be careful of parameter order for asymmetric operations.

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

4. 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 = ¬ dest</td>
</tr>
</tbody>
</table>

5. 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;
}
```

6. Result of compilation (-O1)
1 lea (%rdi,%rsi,1),%eax # $a = t1 = x+y
2 add %edx,%eax # $a = t2 = x+y+z
3 lea (%rsi,%rsi,2),%edx # $d = t4 = 3y
4 shl $0x4,%edx # $d = t4 = 16*(3y) = 48y
5 lea 0x4(%rdi,%rdx,1),%ecx # $c = t5 = x+48y+4
6 imul %ecx,%eax # $a = rval = t5 * t2
7 retq # return rval

7. Optimization converts multiple expressions to a single statement, and a single expression might require multiple instructions.
8. The compiler generates similar code for $(x+y+z) \times (x+4+48 \times y)$.
9. Example

```c
int logical(int x, int y)
{
    int t1 = xˆy;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7; // 0x1FF9
    int rval = t2 & mask;
    return rval;
}
```

10. Result of compilation (-O1)

```c
xor %esi,%edi # x = t1 = xˆy
sar $0x11,%edi # x = t2 = t1 >> 17
mov %edi,%eax # $a = t2 = t1 >> 17
and $0x1ff9,%eax # $a = rval = t2 & mask
retq # return rval
```

21 Control based on condition codes

1. There are four Boolean condition codes

   (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 \texttt{lea} instruction does not set them.

3. Example: addition $t = a + b$

   (a) sets $\text{CF}$ if there is a carry from most significant bit
   (b) sets $\text{ZF}$ if the result in $t$ is zero
   (c) sets $\text{SF}$ if the top bit of $t$ is set ($t < 0$ if we treat $t$ as a signed integer)
   (d) sets $\text{OF}$ 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 ($\texttt{cmp b, a}$) also sets the flags; it’s like computing $a - b$ without modifying the destination.

   (a) sets $\text{CF}$ if there is a carry from most significant bit
   (b) sets $\text{ZF}$ if $a = b$
   (c) sets $\text{SF}$ if $a - b < 0$
   (d) sets $\text{OF}$ 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. [Class 11, 2/5/2018]

6. The test instruction ($\texttt{test b, a}$) 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 $\text{ZF}$ if $a \land b = 0$
   (b) sets $\text{SF}$ if $a \land b < 0$

7. 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 \text{ZF} & \quad \text{Equal/Zero} \\
\text{setne} & \quad \neg\text{ZF} & \quad \text{Not Equal / Not Zero} \\
\text{sets} & \quad \text{SF} & \quad \text{Negative} \\
\text{setsns} & \quad \neg\text{SF} & \quad \text{Nonnegative} \\
\text{setg} & \quad \neg(\text{SF} \oplus \text{OF}) \land \neg\text{ZF} & \quad \text{Greater (Signed)} \\
\text{setge} & \quad \neg(\text{SF} \oplus \text{OF}) & \quad \text{Greater or Equal (Signed)} \\
\text{setl} & \quad (\text{SF} \oplus \text{OF}) & \quad \text{Less (Signed)} \\
\text{setle} & \quad (\text{SF} \oplus \text{OF}) \lor \neg\text{ZF} & \quad \text{Less or Equal (Signed)} \\
\text{seta} & \quad \neg\text{CF} \land \neg\text{ZF} & \quad \text{Above (unsigned)} \\
\text{setb} & \quad \text{CF} & \quad \text{Below (unsigned)}
\end{align*}
8. Many instructions in the jXX dest family jump depending on the condition codes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>true</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>¬ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>¬SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>¬(SF⊕OF)∧¬ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>¬(SF⊕OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF⊕OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF⊕OF)∨ZF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>¬CF∧¬ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

9. Many instructions in the cmovXX src, dest family move data depending on the condition codes.

10. Example

```c
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```assembly
mov %edi,%eax # $a = x
sub %esi,%eax # $a = x-y
mov %esi,%edx # $d = y
sub %edi,%edx # $d = y-x
cmp %esi,%edi # flags for x:y
cmovle %edx,%eax # if $x <= $y then $a=$d
retq
```

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

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

`gcc -O1` generates the same output for this one line as it does for
the body of `absdiff()` above.

## 22 do while loops

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

   ```c
   1 int countOnes(unsigned x) {
   2     int result = 0;
   3     do {
   4         result += x & 0x1;
   5         x >>= 1;
   6     } while (x);
   7     return result;
   8 }
   ```

2. Assembler listing

   ```assembly
   1 0: mov $0x0,%eax     # $a = result = 0
   2 5: mov %edi,%edx     # $d = x
   3 7: and $0x1,%edx     # $d = x & 0x1
   4 a: add %edx,%eax     # $a += (x & 0x1)
   5 c: shr %edi          # x >>= 1
   6 e: jne 5 <countOnes+0x5> # if x != 0, goto 5
   7 10: repz retq        # return result (ignore repz)
   ```

3. for loops are very similar

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

## 23 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. From the bottom (at higher addresses):
   (a) parameters 7 ... (if necessary; later parameters first)
(b) return address
(c) saved registers (saved by callee)
(d) local variables
(e) temporary locations (that don’t fit in registers)
(f) parameters for the next call (“argument build”), last parameter first

5. Class 12, 2/7/2018

6. On the x86 (32 bits), the %ebp register points to the start of the frame, and the %esp register points to the current top of the stack.

7. On the x86_64, the same convention holds, but parameters and local variables often fit in the registers.

8. The stack grows downward.

(a) push src subtracts 8 from %rsp and writes the operand at the new address. (There is no 32-bit equivalent on the x86_64.)
(b) pop dest puts (%rsp) in the destination and then adds 8 to %rsp.
(c) callq label pushes the return address and then jumps to the label. The return address is the address of the instruction after the call.
(d) retq pops the return address and then jumps to it.

9. Linkage at the calling point for swap(&a, &b);

1. mov 0x601050,%esi # address of b
2. mov 0x601070,%edi # address of a
3. call swap # call

10. When a procedure is called, after the procedure pushes %rbp, it copies %rsp into %rbp. Then the following values apply.

<table>
<thead>
<tr>
<th>location</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(%rbp)</td>
<td>old %rbp</td>
</tr>
<tr>
<td>8(%rbp)</td>
<td>return address</td>
</tr>
<tr>
<td>16(%rbp)</td>
<td>7th parameter</td>
</tr>
<tr>
<td>24(%rbp)</td>
<td>8th parameter</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

11. If the saved register area is in use, the %rsp register is not as good a base as the %rbp register for accessing parameters.
24 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 needs the value currently in a volatile register, A must save that register (on the stack) before calling B and pop it when B returns. This situation is called **caller save**.
   - (b) If A needs the value currently in a non-volatile register, it needs to take no special action. The value will survive the call to B.
   - (c) If B needs to use a non-volatile register, B should save the previous value (on the stack) before doing its work and pop it before returning. This situation is called **callee save**.
   - (d) If B needs to use a volatile register, it needs to take no special action. A doesn’t expect the value to be preserved.

4. The convention that gcc follows for the x86 (32 bit):
   - (a) Special-purpose: `%eip`, `%esp`, `%ebp`
   - (b) Non-volatile: `%ebx`, `%esi`, `%edi`
   - (c) Volatile: `%eax`, `%ecx`, `%edx`

5. The convention for the x86_64:
   - (a) Special-purpose: `%rip`, `%rsp`
   - (b) Non-volatile: `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15`
   - (c) All other registers are volatile general-purpose; parameters are passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`.

6. Example

   1. int bitCount(unsigned x) {
   2.     if (x == 0)
   3.         return 0;
   4.     else
   5.         return (x & 1) + bitCount(x >> 1);
   6. }
25 Code for local variables, pointers

1. The linkage uses a stack frame in some cases

   (a) More than 6 parameters; the extra ones go on the stack.
   (b) A local variable is dereferenced with the & operator, so it must
        have an address (it can’t be in a register).
   (c) A local variable is an array or struct, so it needs an address.

2. The callee allocates space on the stack by decrementing the stack
   pointer.

3. Example

```
1 // example from book p. 249
2 long caller() {
3      long arg1 = 0x216;
4      long arg2 = 0x421;
5      long sum = swap_add(&arg1, &arg2);
6      long diff = arg1 - arg2;
7      return sum * diff;
8 } // caller()
```
### Data types

1. **Class 13, 2/9/2018**

2. 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</td>
<td>q</td>
<td>8</td>
</tr>
</tbody>
</table>

3. 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</td>
</tr>
<tr>
<td>extended</td>
<td>long double</td>
<td>t</td>
<td>12 (32-bit) /16 (x86_64)</td>
</tr>
</tbody>
</table>

27 Arrays

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

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

3. Examples:

<table>
<thead>
<tr>
<th>declaration</th>
<th>length (bytes)</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 (x86) / 24 (x86_64)</td>
</tr>
</tbody>
</table>

4. C syntax, given \( \text{int} \) 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[0]</td>
<td>int</td>
<td>1</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>

5. 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.

6. Simple example: return uky[ind];
7. Loop example

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

```assembly
mov $0x0, %eax  # a = 0
L: addl $0x1, (%rdi, %rax, 1)  # z[a] += 1
    add $0x4, %rax  # a += 1
    cmp $0x14, %rax  # 5 : a
    jl L  # if a < 5 goto L
    repz retq  # return
```

28 Nested arrays

1. [Class 14, 2/12/2018]

2. Example

```c
#define ZLEN 5
typedef int myArrayType[ZLEN];

#define PCOUNT 4
myArrayType pgh[PCOUNT] =
    {{1, 5, 2, 0, 6 },
     {7, 11, 8, 7, 9 },
     {11, 15, 12, 11, 17 },
     {19, 23, 20, 20, 19 }};

int *showPgh() {
    int index;
    scanf("%d", &index);
    return pgh[index];
}
```
3. The values are placed contiguously from some location $x$ to $x + 4*PCOUNT*ZLEN - 1 = x + 79$. This ordering is guaranteed.

4. $pgh$ is an array of 4 elements.

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

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

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

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

    ```
    1     # a = index
    2    lea (%rax,%rax,4),%rax # a = 5*index (start of row)
    3    lea pgh(,%rax,4),%rax # a = pgh + 4*5 index
    ```


10. Example:

    ```
    1   int getElement(long n, long x[n][n], long i, long j) {
    2       return x[i][j];
    3   }
    ```

    ```
    1     shl $0x3,%rdx # d = 8i
    2     imul %rdx,%rdi # t = n*8i
    3    lea (%rsi,%rcx,8),%rax # a = x + 8j
    4    mov (%rax,%rdi,1),%rax # a = Mem[x + n*8i + 8j]
    5   retq
    ```

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).

12. If you want a very efficient loop to zero out an array, you can do something like this:

    ```
    1   void zeroArray(long n, long x[n][n]) {
    2       long *ptr;
    3   for (ptr = &x[n-1][n-1]; ptr >= &x[0][0]; ptr -= 1) {
    4           *ptr = 0;
    5       }
    6   }
    ```
13. But there is a faster C routine to do the same: \texttt{bzero(3)}

## 29 Structures

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

2. Example:

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

   Memory layout of \texttt{foo}, starting at address \texttt{x}, is based on offsets:

<table>
<thead>
<tr>
<th>code</th>
<th>address</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>x</td>
<td>struct rec</td>
</tr>
<tr>
<td>foo.y</td>
<td>x</td>
<td>int *</td>
</tr>
<tr>
<td>foo.i</td>
<td>x+12</td>
<td>int</td>
</tr>
<tr>
<td>foo.n</td>
<td>x+16</td>
<td>struct rec *</td>
</tr>
<tr>
<td>foo.y[1]</td>
<td>x+4</td>
<td>int</td>
</tr>
<tr>
<td>foo.y</td>
<td>x</td>
<td>int *</td>
</tr>
</tbody>
</table>

3. The compiler knows all the offsets.

## 30 Linked lists

1. \texttt{Class 15, 2/14/2018}
2. Example:

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

3. Generated code

```assembly
test %rdi,%rdi # test r&r
je L2 # if (r==0) jump to L2
L1: movslq 0xc(%rdi),%rax # a = ind = r[12] = r.i (extend sign)
mov %esi,(%rdi,%rax,4) # r->y[ind] = val
mov 0x10(%rdi),%rdi # r = r->n
test %rdi,%rdi # test r&r
jne L1 # if (r != 0) jump to L1
L2: repz retq
```

31 Alignment

1. Example:

```c
struct S1 {  
    char c;
    int y[2];
    double v; // uses 8 bytes
    char d;
} *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 \texttt{02}
   (c) 4 bytes (int, void *): address ends with \texttt{002}
(d) 8 bytes (**double**): address ends with $000_2$ (Linux/x86 compilers choose $00_2$).

(e) 12 bytes (**long double** on x86.32): gcc chooses $00_2$.

4. The x86.64 is stricter

(a) 8 bytes (**long, 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 data that crosses chunk boundaries.

(c) It is tricky to access data that crosses page boundaries (typically every 4KB).

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

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>address</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>char</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>pad</td>
<td></td>
<td>x+1</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>int[2]</td>
<td>x+4</td>
<td>8</td>
</tr>
<tr>
<td>pad</td>
<td></td>
<td>x+12</td>
<td>4</td>
</tr>
<tr>
<td>v</td>
<td>double</td>
<td>x+16</td>
<td>8</td>
</tr>
<tr>
<td>d</td>
<td>char</td>
<td>x+24</td>
<td>1</td>
</tr>
<tr>
<td>pad</td>
<td></td>
<td>x+25</td>
<td>7</td>
</tr>
<tr>
<td>end</td>
<td></td>
<td>x+32</td>
<td></td>
</tr>
</tbody>
</table>

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 Unions

1. A **union** type is like a **struct**, but the fields all start at offset 0, so they overlap.

2. This method allows us to view the same data as different types.
3. It also lets us look at individual bytes of numeric types to see the byte ordering.

4. Example:

```c
union {
  char ch[8]; // 8*1 = 8 bytes
  short sh[4]; // 4*2 = 8 bytes
  int in[2]; // 2*4 = 8 bytes
  long lo[1]; // 1*8 = 8 bytes (on i86_64)
  float fl[2]; // 2*4 = 8 bytes
} dw;

dw.fl[0] = 3.1415;
printf("as \textit{float}: %f; as \textit{integer}: %d; \n"
  "as \textit{two shorts}: %d, %d \n",
  dw.fl[0], dw.in[0], dw.sh[0], dw.sh[1]);
```

Result:

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

33 Buffer overflow

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

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

3. 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.

4. 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.
5. Class 16, 2/16/2018

6. Class 17, 2/19/2018

7. There are hundreds of other examples.

8. Mitigating vulnerability

   (a) Over-allocated character arrays. We saw the compiler do this in Lab 4.

   (b) Library routines that limit lengths: `fgets()`, `strncpy()`, `scanf(...%ns...)`.

   (c) Randomized stack offsets: assign a random start 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. Project 3 randomizes stack offset for the last two levels.

   (d) Non-executable segments: On the x86, anything readable is executable, including the stack. On the x86_64, there is separate executable permission. Project 3 marks the stack non-executable for the last two levels.

   (e) Stack canaries: push a canary value on stack just after the return address; 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)

   ```
   1   0: push %rbp
   2   1: mov %rsp,%rbp
   3   4: sub $0x330,%rsp # room for locals
   4   b: mov %fs:0x28,%rax # canary
   5  14: mov %rax,-0x8(%rbp)
   6 ...
   7  1f: mov -0x8(%rbp),%rdx
   8  23: xor %fs:0x28,%rdx
   9  2c: je 33
   10 2e: callq <error>
   11 33: leaveq
   12 34: retq
   ```

9. 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.

34 Linux x86_64 memory layout

1. Simplified version of allocation of virtual space

<table>
<thead>
<tr>
<th>start</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>0x0040000</td>
<td>text</td>
<td>program</td>
<td>read, execute</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>0x7fe000000000</td>
<td>libs</td>
<td>groups of 4 regions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>stack</td>
<td>frames</td>
<td>read/write; grows down; to 2^{47}</td>
</tr>
<tr>
<td>0x8000000000000</td>
<td>kernel</td>
<td>kernel code, shared</td>
<td>no access; starts at 2^{47}</td>
</tr>
<tr>
<td>0x1000000000000</td>
<td>none</td>
<td>none</td>
<td>not addressable; starting at 2^{48}</td>
</tr>
</tbody>
</table>

2. The libraries are each composed of four regions
   (a) text (read, execute)
   (b) pad (no access)
   (c) constants (read)
   (d) data (read, write)

3. Try the command `less /proc/self/maps`.

4. To find shows per-process limitations, use the command (in the `bash` shell) `ulimit -a`. In `csh`, use the command `limit`. You will find, for instance, the stack is limited to 8MB, and that a process can have 1048576 files open at once (!).

35 Linking

1. [Class 18, 2/21/2018]

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.

### 36 Linking details

1. Symbol resolution

   (a) Programs define symbols and reference them:

   ```
   1 void swap() {...} // exported global identifier
   2 extern int myGlobal; // imported global identifier
   3 int myGlobal; // local and exported
   4 static int myLocal; // local, not exported
   5 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.

2. Shared object files

(a) The linker can store its result as a **shared object file** (conventional extension “`.so`”), which a program can load into memory dynamically, typically when the program starts.

(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. At that time, the entire library is brought into (virtual) memory.

(d) Shared object files are shared with all processes that are using the same library.

(e) Windows calls shared object files **Dynamic Link Libraries (DLLs)**.