CS270 classnotes

Raphael Finkel

November 15, 2019

1 Intro

Class 1, 8/26/2019

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.cs.uky.edu`, where `name` is your LinkBlue account name. The first lab will acquaint you with this facility.
7. Text: Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (3rd edition) – strongly recommended.

2 A brief introduction to systems programming in Unix

1. Standard input, standard output, standard error.

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.

Class 2, 8/28/2019

3 Software tools

$$\frac{\textbf{Use (client)}}{\textbf{Implementation}}\ \textbf{Spec} \longrightarrow \frac{\textbf{Programmer}}{\textbf{Compiler}}\ \textbf{Language}$$

4 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 of hardware limits
   (a) C **`int`** is not an integer: 50000*50000 = 2500000000, but the int has value -1794967296. examples.c
   (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.

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 assembler.

Class 3, 8/30/2019 Laboratory 1

Class 4, 9/4/2019

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)

   ```
   void fun(int index) {
     printf("fun(%d):␣", index);
     double d[1] = {3.14};
     long a[2];
     a[index] = 9223372036854775803L;
     printf("%lf\n", d[0]);
   } // fun
   void funTest() {
       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 ***: <unknown> terminated */
   } // funTest
   ```

   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 $0 \ldots 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}$. The letters are usually written in capital letters. Each hex digit corresponds to four bits. $285.3 = 11\mathrm{D}.4\mathrm{CCCCCCCC}\ldots_{16}$

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

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 ($-80000000_{16} \ldots 7\mathrm{FFFFFFF}_{16}$).

5. 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 $-2^{n-1} \cdots - 1$.

   (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. Final result: 1010.

(f) To negate again: $-6_{10} = 1010_2$. Flipped: 0101. Final result: 0110.

(g) Negating 0 gives 0.

(h) The most negative number (when $n = 4$) is $-8_{10} = 1000_2$. Negating this number leaves it unchanged.

6. $\boxed{\text{Class 5, 9/6/2019}}$: Laboratory 2

7. $\boxed{\text{Class 6, 9/9/2019}}$

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, $2^{10} \approx 10^3$, 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
$2^{10} = 1024 \approx 10^3 = 1$ thousand (kilo or K);
$2^{20} = 1048576 \approx 10^6 = 1$ million (mega or M);
$2^{30} = 1073741824 \approx 10^9 = 1$ billion (giga or G);
$2^{40} \approx 10^{12} = 1$ trillion (tera or T);
$2^{50} \approx 10^{15} = 1$ quadrillion (peta or P);
So $2^{32} = 4$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 4×10^3, 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.

C declaration	x86	x86-64	
char	1	1	
short	2	2	
int	4	4	
long	4	8	(!)
long long	8	8	
float	4	4	
double	8	8	
long double	12	16	(!)
pointer	4	8	(!)

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 `od` program to show a file in bytes, characters, integers, ... For instance:

   ```
   od -t x4 -t o1 /etc/hosts | head
   ```

6. You can also use a program:

   ```
   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");
   } // show_bytes
   int main() {
     int a = 15213;
     printf("int a = %d (0x%08x);\n", a, a);
     show_bytes((pointer) &a, sizeof(int));
   } // main
   ```

The output is:

```
int a = 15213 (0x00003b6d);
0x7fffcf95d80c  0x6d
0x7fffcf95d80d  0x3b
0x7fffcf95d80e  0x00
0x7fffcf95d80f  0x00
```

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. Class 7, 9/11/2019
5. 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.
6. 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.
7. 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.
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 1.8×10^{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 (American Standard Code for Information Interchange).
4. Class 8, 9/13/2019 Laboratory 3
5. Class 9, 9/16/2019
6. 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.
7. There is no need to worry about byte ordering (even for characters from languages written right-to-left); the start of the string always has the lowest address in memory.
8. 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.
9. Declaration
   (a) `char *myBuffer;`
   (b) `char myBuffer[];`
   (c) `char myBuffer[4096];`

10. 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:

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

5. 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.
6. One can use the Boolean operators in C and apply them to any integral data type: **`char`**, **`short`**, **`int`**, **`long`**, **`long long`**.
7. 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

1. $\boxed{\text{Class 10, 9/18/2019}}$
2. Compiler for C: `gcc`. Compiler for C++: `g++`.
3. 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.
6. `ldd` *filename*: tells what dynamic libraries *filename* uses.
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. Class 11, 9/20/2019

8. Sample code:

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

9. 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
```

10. 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.

11. 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
```

12. Same thing with `gdb`, using command "x/25xb sum":

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

13. Compiling with -O1:

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

14. One can even disassemble `.EXE` files (from Win32 complations) 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.

64-bit	32-bit	16-bit	8-bit	original purpose	C
rax	eax	ax	ah/al	accumulator	return value
rbx	ebx	bx	bh/bl	base	non-volatile
rcx	ecx	cx	ch/cl	counter	4th parameter
rdx	edx	dx	dh/dl	data	3rd parameter
rsi	esi		sil	source index	2nd parameter
rdi	edi		dil	destination index	1st parameter
rsp	esp	sp	spl	stack pointer	stack pointer
rbp	ebp	bp	bpl	base	base; non-volatile
r8	r8d			general purpose	5th parameter
r9	r9d			general purpose	6th parameter
r10	r10d			general purpose	
r11	r11d			general purpose	
r12	r12d			general purpose	non-volatile
r13	r13d			general purpose	non-volatile
r14	r14d			general purpose	non-volatile
r15	r15d			general purpose	non-volatile
rip	eip	ip		instruction pointer	

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. On our machine, simple **`mov`** means whichever is appropriate given the size of the operands.

4. Operand types

   (a) **Immediate**: integer constant, such as `$0x400` or `$-533` The actual constant is represented in 1, 2, or 4 (not 8) 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 32-bit signed integer constant, such as `8(%eax)`. Either source or destination, but not both, may be memory or displacement.

5. Example in C: Swap

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

6. Same thing in assembler

```
  mov (%rdi),%rax       # a = xs[0]
  mov 0x8(%rdi),%rdx    # d = xs[1]
  mov %rax,0x8(%rdi)    # xs[1] = a
  mov %rdx,(%rdi)       # xs[0] = d
  retq                   # return
```

18 More complex memory-addressing modes

1. Class 12, 9/23/2019
2. We saw **memory** and **displacement**.
3. This is the most general form: *D(Rb,Ri,S)*
   (a) *D* is the 32-bit signed-integer displacement, such as 1, 2, 80.
   (b) *Rb* is the base register: any of the registers. It may be omitted, in which case its value is 0.
   (c) *Ri* is the index register, any register but `%rsp`, and you most likely don't want to use `%rbp`, either
   (d) *S* is a scale, which is any of 1, 2, 4, or 8.
4. The value it references is Mem[Reg[*Rb*]+*S**Reg[*Ri*]+*D*].
5. 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. One can compute an address and save it without actually referencing it.
2. **lea** *src*, *dest*
3. Load Effective Address of *src* and put it in *dest*.
4. Purpose: translate `p = &x[i]`
5. Purpose: compute arithmetic expressions like `x+k*y` where k is 1, 2, 4, or 8.
6. Example: `x*12`

```
leal (%eax,%eax,2), %eax # x = x+2x
sall $2, %eax            # x = x * 4
```

20 Arithmetic operations

1. Two-operand instructions. The source and destination must have the same size (1, 2, 4, or 8 bytes)

instruction	meaning
add	dest = dest + src
sub	dest = dest − src
imul	dest = dest × src
sal	dest = dest << src
sar	dest = dest >> src (arithmetic)
shr	dest = dest >> src (logical)
xor	dest = dest ⊕ src (bitwise)
and	dest = dest ∧ src (bitwise)
or	dest = dest ∨ src (bitwise)

2. Be careful of parameter order for asymmetric operations.
3. There is no distinction between signed and unsigned integers.
4. One-operand instructions

instruction	meaning
inc	dest = dest + 1
dec	dest = dest − 1
neg	dest = −dest
not	dest = ¬ dest (bitwise)

5. Class 13, 9/25/2019

6. Example

```
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 (-O1)

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

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

9. The compiler generates similar code for `(x+y+z)*(x+4+48*y)`.

10. Example

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

11. Result of compilation (-O1)

```
xor     %esi,%edi       # x = t1 = x^y
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 **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 result in t is zero
   (c) sets **SF** if the top bit of t is set ($t < 0$ if we treat t as a signed integer)
   (d) sets **OF** if there is a two's complement overflow:
   $(a > 0 \wedge b > 0 \wedge t < 0) \vee (a < 0 \wedge b < 0 \wedge t > 0)$

4. The compare instruction (**`cmp`** `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:
   $(a > 0 \wedge b < 0 \wedge (a - b) < 0) \vee (a < 0 \wedge b > 0 \wedge (a - b) > 0)$

5. The test instruction (**`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 **`ZF`** if $a \wedge b = 0$
   (b) sets **`SF`** if $a \wedge b < 0$

6. Many instructions in the **`set`**XX *dest* family test the condition codes and set the destination (a single byte) to 0 or 1 based on the result.

`sete`	**ZF**	Equal/Zero
`setne`	¬**ZF**	Not Equal / Not Zero
`sets`	**SF**	Negative
`setns`	¬**SF**	Nonnegative
`setg`	¬(**SF**⊕**OF**)∧¬**ZF**	Greater (Signed)
`setge`	¬(**SF**⊕**OF**)	Greater or Equal (Signed)
`setl`	(**SF**⊕**OF**)	Less (Signed)
`setle`	(**SF**⊕**OF**)∨**ZF**	Less or Equal (Signed)
`seta`	¬**CF**∧¬**ZF**	Above (unsigned)
`setb`	**CF**	Below (unsigned)

7. Many instructions in the **`j`**XX *dest* family jump depending on the condition codes.

`jmp`	true	Unconditional
`je`	**ZF**	Equal/Zero
`jne`	¬**ZF**	Not Equal / Not Zero
`js`	**SF**	Negative
`jns`	¬**SF**	Nonnegative
`jg`	¬(**SF**⊕**OF**)∧¬**ZF**	Greater (Signed)
`jge`	¬(**SF**⊕**OF**)	Greater or Equal (Signed)
`jl`	(**SF**⊕**OF**)	Less (Signed)
`jle`	(**SF**⊕**OF**)∨**ZF**	Less or Equal (Signed)
`ja`	¬**CF**∧¬**ZF**	Above (unsigned)
`jb`	**CF**	Below (unsigned)

8. Many instructions in the **cmov**XX *src*, *dest* family move data depending on the condition codes.
9. Example

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

```
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
```

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

```
        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. Class 14, 9/27/2019
2. C code to count how many 1's in a parameter

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

3. Assembler listing

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

4. **`for`** loops are very similar
5. 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. Class 15, 10/2/2019
5. 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

6. On the x86 (32 bits), the `%epb` 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 (using `%rpb`), but parameters and local variables often fit in the registers. Local variables that must have an address (such as arrays) are always placed on the stack; registers don't have a memory address.
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);`

```
mov $0x601050,%rsi   # address of b
mov $0x601070,%rdi   # address of a
call swap            # call
```

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

location	meaning
`0(%rbp)`	old `%rbp`
`8(%rbp)`	return address
`16(%rbp)`	7th parameter
`24(%rbp)`	8th parameter
...	

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

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

```
      mov    $0x0,%eax   # a = 0
      test   %edi,%edi   # compute x & x
      jne    L1          # if not eq 0, goto L1
      retq               # if eq 0, return
L1:   push   %rbx        # save volatile rbx
      mov    %edi,%ebx   # b = x
      shr    %edi        # x >>= 1
      callq  bitcount    # a = bitcount(x>>1)
      and    $0x1,%ebx   # b = x & 1
      add    %ebx,%eax   # a = a+b
      pop    %rbx        # restore b
      retq
```

Class 16, 10/4/2019 Laboratory 4

25 Code for local variables, pointers

1. Class 17, 10/9/2019
2. The linkage uses a stack frame in some cases
   (a) More than 6 parameters; the extra ones go on the stack.
   (b) If a local variable is dereferenced with the `&` operator, it must have an address (it can't be in a register).
   (c) A local array or struct needs an address.
3. The callee allocates space on the stack by decrementing the stack pointer.
4. Example

```
// example from book p. 249
long caller() {
    long arg1 = 0x216;
    long arg2 = 0x421;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
} // caller()
```

```
push   %rbp                 # save rbp
mov    %rsp,%rbp            # rbp = top of stack
sub    $0x20,%rsp           # room for 32 bytes on stack
movq   $0x216,-0x20(%rbp)   # arg1 = 0x216
movq   $0x421,-0x18(%rbp)   # arg2 = 0x421
lea    -0x18(%rbp),%rdx     # d = &arg2
lea    -0x20(%rbp),%rax     # a = &arg1
mov    %rdx,%rsi            # param 2 = d
mov    %rax,%rdi            # param 1 = a
callq  swap_add             # a = swap_add(a,d)
mov    %rax,-0x10(%rbp)     # sum = a
mov    -0x20(%rbp),%rdx     # d = arg1
mov    -0x18(%rbp),%rax     # a = arg2
sub    %rax,%rdx            # d = arg1 - arg2
mov    %rdx,%rax            # a = arg1 - arg2
mov    %rax,-0x8(%rbp)      # diff = a
mov    -0x10(%rbp),%rax     # a = sum
imul   -0x8(%rbp),%rax      # a = sum * diff
leaveq                      # rsp = rbp; pop rbp
retq                        # return
```

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

Intel	C	assembler	bytes
byte	**char**	b	1
word	**short**	w	2
double word	**int**	l	4
quad word	**long**	q	8 (x86_64)

2. Floating point

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

Intel	C	assembler	bytes
single	**float**	`s`	4
double	**double**	`l`	8
extended	**long double**	`t`	12 (32-bit) /16 (x86_64)

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 $L \times$ **sizeof**(T) bytes.
3. Examples

declaration	length (bytes)
char `string[12]`	12
int `val[5]`	20
double `a[3]`	24
char `*p[3]`	12 (x86) / 24 (x86_64)

4. C syntax, given **int** `val[5];` stored starting at location `x`, containing 1, 2, 3, 4, 5.

expression	type	value
`val[0]`	**int**	1
`val`	**int** *	`x`
`val + 1`	**int** *	x+4
`&val[2]`	**int** *	x+8
`val[4]`	**int**	5
`val[5]`	**int**	garbage
`*(val+1)`	**int**	2
`val + i`	**int** *	x+4`i`

5. Using the same type for several arrays

```
#define ZLEN 5
typedef int myArrayType[ZLEN];
myArrayType cmu = { 1, 5, 2, 1, 3 };
myArrayType mit = { 0, 2, 1, 3, 9 };
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];`

```
mov  0x6010b0(,%rdi,4),%eax # a = M[uky + 4*ind]
retq
```

7. Loop example

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

```
   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. Example

```
#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];
}
```

2. The values are placed contiguously from some location `x` to `x` + 4*`PCOUNT`*`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`** *)

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

8. Class 18, 10/11/2019 Midterm will be on Oct 16. Quick review.
9. Class 19, 10/16/2019 Midterm exam.
10. Class 20, 10/18/2019 Discussion of midterm exam.
11. Class 21, 10/23/2019
12. 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 $A + kiC + kj = A + k(iC + j)$.
13. Example:

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

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

14. 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).
15. If you want a very efficient loop to zero out an array, you can do something like this:

```
void zeroArray(long n, long x[n][n]) {
  long *ptr;
  for (ptr = &x[n-1][n-1]; ptr >= &x[0][0]; ptr -= 1) {
    *ptr = 0;
  }
}
```

16. But there is a faster C routine to do the same: `bzero(3)`

29 Structures

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

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

Memory layout of `foo`, starting at address `x`, is based on offsets:

code	address	type
`foo`	x	**struct rec**
`foo.y`	x	**int ***
`foo.i`	x+12	**int**
`foo.n`	x+16	**struct rec ***
`foo.y[1]`	x+4	**int**
`foo.y`	x	**int ***

3. The compiler knows all the offsets.

30 Linked lists

1. Example:

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

2. Class 22, 10/25/2019

3. Generated code

```
    test   %rdi,%rdi           ; test r&r
    je     L2                  ; if r == 0 go to L2 (done)
L1: movslq 0xc(%rdi),%rax      ; ind = r->i (extend sign)
    mov    %esi,(%rdi,%rax,4)  ; *(r+ind) = val
    mov    0x10(%rdi),%rdi     ; r = r->n
    test   %rdi,%rdi           ; test r&r
    jne    L1                  ; if r != 0 go to L1 (repeat)
L2: repz retq
```

31 Alignment

1. Example:

```
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_32 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`** 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 architectures 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:

field	type	address	length
`c`	**`char`**	x	1
	pad	x+1	3
`y`	**`int`**[2]	x+4	8
	pad	x+12	4
`v`	**`double`**	x+16	8
`d`	**`char`**	x+24	1
	pad	x+25	7
end		x+32	

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 **`struct`**s 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:

```
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␣float:␣%f;␣as␣integer:␣%d;\n"
        "as␣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 are problematic: `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. Class 23, 10/28/2019
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 network connection to the attacker.

6. There are hundreds of other examples.
7. Mitigating vulnerability

   (a) Over-allocated character arrays. The compiler used to do this in Lab 4, allocating 16B for each 4-character array.

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

   (c) Randomized stack offsets: place the stack at a random location 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 phases.

   (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 phases.

   (e) Stack **canaries**: Push a canary value on the stack at the beginning of the procedure, and then 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)

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

   (f) Address space layout randomization: As it loads each program, let the operating system randomize the starting address of each region (code, data, stack) to make return-oriented programming more difficult.

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.

34 Linux x86_86 memory layout

1. Class 24, 10/30/2019

2. Simplified version of allocation of virtual space

start	name	purpose	properties
0	unused	prevent errors	no access
0x0040000	text	program	read, execute
	data	initialized data	read, write; static size
	bss	uninitialized data	read, write; static size
	heap	allocatable data	read, write; grows up
0x7fe000000000	libs	groups of 4 regions	
	stack	frames	read/write; grows down; to 2^{47}
0x800000000000	kernel	kernel code, shared	no access; starts at 2^{47}
0x1000000000000	none	none	not addressable; starting at 2^{48}

3. The libraries are each composed of four regions

   (a) text (read, execute)

   (b) pad (no access)

   (c) constants (read)

   (d) data (read, write)

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

5. To show 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 25, 11/1/2019
2. Basic idea: combine results of one or more independent compilations with libraries.
3. Class 26, 11/4/2019
4. The individual compiled results are called **relocatable object files**; the Unix convention is that their names end "`.o`".
5. 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:

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

3. Relocation

   (a) The linker combines the object files into a single file.
   (b) All text segments can placed together; similarly for other segment types, although for Linux on the x86_64, each text segment gets its own data segment; there is plenty of virtual space available.
   (c) 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.
   (d) The linker updates all references to relocated global identifiers.

(e) Jump and call instructions use **relative addressing** so that the resulting code is **relocatable**, that is, it can be placed anywhere in memory.

37 Format for object files

1. Since about 1990, Unix variants have standardized to a single format for object files: **Executable and Linkable Format (ELF)**.
2. ELF format applies to relocatable object files, executable object files, and shared object files; together, we call them **ELF binaries**.
3. See `objdump -h /bin/sh`, or better: `readelf -a /bin/sh | less`
4. Sections of an ELF file (simplified)
   (a) header: word size, byte ordering, object-file type, architecture, entry point
   (b) segment header table
   (c) code (`.text` section)
   (d) read-only data, such as jump tables (`.rodata` section)
   (e) initialized global variables (`.data` section)
   (f) uninitialized global variables (`.bss` section): only length, no content. "bss" stands for "block started by symbol".
   (g) symbol table (`.symtab` section), including procedures and static variables, section names, each with location.
   (h) text relocation tables (`.rela.dyn` and `.rela.plt` sections): addresses that need to be modified for relocation and how to relocate them.
   (i) debugging information (`.debug_*` sections), produced, for instance, with `gcc -g`.
   (j) section header table: offsets and sizes of each section
5. **Global symbols**: defined in this module, may be referenced by other modules. In C: functions (except **`static`**) and file-global variables (except **`static`**).
6. **External symbols**: global symbols referenced by this module but defined in another module.

7. **Local symbols**: symbols defined and referenced only within this module. In C: functions and file-global variables defined **`static`**. The object file does not even name variables declared within functions.

38 Unix tools for object files

1. Class 26, 11/4/2019
2. `ar`: creates static libraries
3. `strings`: lists printable strings in any file
4. `strip`: deletes symbol-table information from an object file, so it is no longer linkable.
5. `nm`: list the symbol table of an object file
6. `size`: show the names and sizes of the sections of an object file
7. `readelf`: display the content of an object file; try `readelf -a /usr/bin/tcsh`.
8. `objdump`: show the names and sizes of the sections of an object file
9. `ldd`: show the dynamically-linked libraries this object file needs

39 Resolving multiple definitions of the same symbol

1. **Strong** symbols are procedures and initialized global variables.
2. **Weak** symbols are uninitialized global variables.
3. Multiple strong symbols are an error.
4. A single strong symbol is equated to all weak occurrences. This choice can lead to errors if the variable is declared of different types in different compilation units.
5. Multiple weak symbols are allowed; one is chosen. This choice is also error-prone.
6. Advice:

   (a) Multiple compilation units should share global declarations via a `.h` file.

(b) Use **`static`** if possible to prevent conflicts.
(c) Initialize global variables (to make them strong).
(d) Declare shared global variables **`extern`**.

Class 27, 11/6/2019: Laboratory 5

40 Libraries

1. Class 28, 11/11/2019
2. C library (`libc`): about 4MB (static), 2MB (dynamic); about 2200 symbols; routines for I/O, memory allocation, signals, strings, random numbers, integer mathematics.
   `objdump -T /lib/x86_64-linux-gnu/libc-2.27.so`
3. Math library (`libm`): about 1MB; about 1000 symbols; floating-point math.
4. The linker only resolves the symbols in the library that are unresolved when it gets to that library in the command-line order, so libraries should be listed at the end of the command line, and special libraries (like `libm`) before general ones (like `libc`).
5. Shared libraries that are dynamically loaded reduce duplication in executable object files and allow for quick incorporation of bug fixes.
6. Shared libraries are usually loaded into memory when a program starts, but it is possible to load them later.

```
#include <dlfcn.h>
void *handle;
void (*addvec)(int *, int *, int *, int);
char *error;
handle = dlopen("./libvector.so", RTLD_LAZY);
// should verify handle != NULL
addvec = dlsym(handle, "addvec");
// should verify addvec != NULL
// may now invoke addvec()
dlclose(handle); // should verify return value >= 0
```

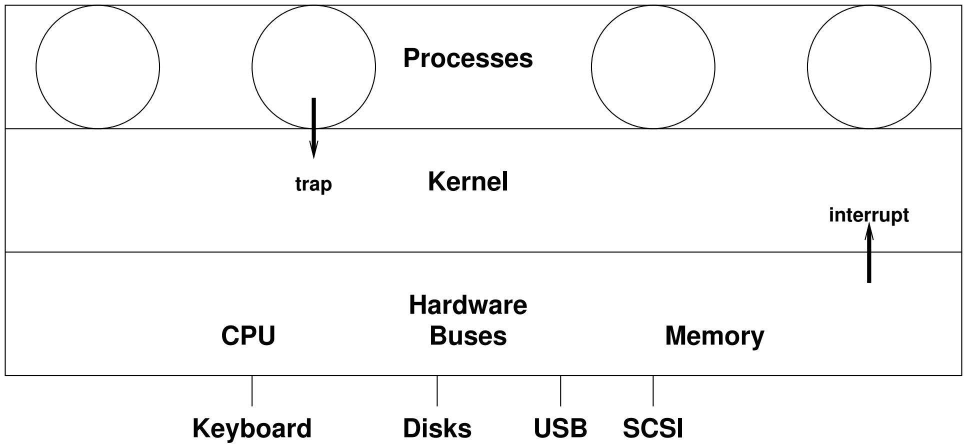
41 Interpositioning

1. Replace standard routine with a special one in order to monitor (for example, to find memory leaks), profile (for improving efficiency), add facility (like encryption), reduce facility (sandboxing).
2. At compile time: Build replacement, and use `#define` to force calls to go to the replacement.
3. At link time: build replacement version `__wrap_foo()` that calls `__real_foo()` when it needs it. Call the linker with `--wrap,foo`. `Gcc` can pass this flag to the linker: `-Wl,--wrap,foo`. Then the linker resolves calls to `foo()` as `__wrap_foo()` and calls to `__real_foo()` as `foo()`.
4. At load time: tell dynamic linker to resolve symbols by going first to a special library. The dynamic linker uses the `LD_PRELOAD` environment variable:

```
LD_PRELOAD="/usr/lib64/libdl.so ./mymalloc.so"
./myProg
```

42 Operating systems — Introduction

1. Goals
   (a) abstract the hardware, hiding some features and providing new ones.
   (b) share resources among processes.
2. Class 29, 11/13/2019
3. Layers
   (a) Hardware, including instruction set, registers, memory, and devices.
   (b) OS kernel
   (c) Applications (processes)

4. 

5. Kernel design alternatives

   (a) Monolithic: many functions, all linked together.
   (b) Other: object-oriented, layered, dynamically loaded modules
   (c) Linux: Mostly monolithic, written in C (and some assembler), but with some object-based data structures, layers for utilities such as memory management, and dynamically loaded modules, mostly for device control.

6. The kernel code runs only when

   (a) A process requests assistance via a **system call**.
   (b) A process executes an invalid instruction.
   (c) A device generates an interrupt indicating it needs service.

7. The kernel executes in a **privileged mode** that lets it execute any instruction, access any memory location, access any device.
8. Processes execute in **user mode**, which limits the instructions, memory access, and device access.

43 Hardware exception handling

1. Hardware **exceptions** are of two types:

   (a) **interrupts** are caused asynchronously by devices; examples are clock interrupts and device-completion interrupts. Some are unrecoverable, such as power fail.

   (b) **traps** are caused by faults during executing an instruction, such as division by zero or attempt to access memory in an invalid way. They are also intentionally caused by processes in order to invoke a system call.

2. When an exception (either a trap or an interrupt) occurs:
   (a) The CPU saves the current state (at least the PC, usually other information as well such as the current processor state) in a standard place (often on the stack).
   (b) The CPU changes to privileged (note spelling!) mode.
   (c) The CPU jumps to a standard location (based on the particular exception, typically through a jump table called an **interrupt vector**). That location is in the kernel.

3. When the kernel is ready to return from the exception, it executes a "return from exception" instruction:
   (a) The CPU changes to its previous mode (typically unprivileged (user) mode) and its previous location.
   (b) The CPU pops saved information from the stack, if that is where the CPU placed it.

44 Example: opening a file in Linux x86_86

1. In C, a program invokes `open(filename, options)`.
2. The C library handles this function by executing a `syscall` instruction with the number of the system call in `%rax`.
3. The `syscall` instruction causes a trap, switching context to the kernel.
4. Actually, many system calls first go to the `vdso` (virtual dynamically linked shared object) region mapped into virtual memory to avoid such context switches in some cases.
5. The Linux kernel runs a procedure called `sys_open()`. Any error it encounters causes it to return a specific negative number indicating the error, such as `EFAULT`.
   (a) verify that the filename string is in a valid location in the process's memory.

   (b) verify that the file exists.
   (c) verify that the file permissions allow this process to open it in the manner specified by `options`.
   (d) build a kernel data structure `files_struct`.
   (e) return the index of that structure within the kernel's per-process `fd_array`.

6. The C library sees if the return value R is negative. If so, it stores R in the global variable `errno` and returns –1. Otherwise, it returns R.