CS485G classnotes

Raphael Finkel

May 4, 2015

1 Intro

Lecture 1, 1/14/2015

1. Handout 1 — My names
2. Plagiarism — read aloud
3. E-mail list: `cs485@cs.uky.edu`
4. Labs: about five throughout the semester, typically on Fridays. The first one is this Friday. Labs count toward your grade; you must do them during the lab session.
5. Projects on web at `https://www.cs.uky.edu/~raphael/courses/CS485.html`. First project — Review of the C language
6. Accounts in MultiLab if you want; every student will have a virtual machine as well, at `name@name.netlab.uky`, where `name` is your LinkBlue account name.
7. Text: Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (2nd edition) – required.

2 Software tools

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

3 Abstraction and reality

1. Most CS and CE courses emphasize abstraction; it matches how we think, and it lets us hide implementation details and complexity.
2. But hardware has limits that our abstractions lack (maximum size of an integer, for instance). If we hide implementation details, we are at risk of inefficiency and inability to cooperate with other components.
3. Examples
   (a) C **`int`** is not an integer: 50000*50000 = 2500000000, but the int is -1794967296.
   (b) C **`float`** is not real: 1e20 + 3.14 - 1e20 = 3.14, but the float result is 0.0.
   (c) Programming languages hide the instructions that are executed
   (d) Layout in memory affects performance (caches, pages). Example:

```
#define BIG 10000
void copyij(int src[BIG][BIG], int dst[BIG][BIG])
{
  int row, col;
  for (row = 0; row < BIG; row += 1)     // reorder?
    for (col = 0; col < BIG; col += 1)   // reorder?
      dst[row][col] = src[row][col];
}
int from[BIG*BIG], to[BIG*BIG];
copyij(from, to);
```

   One experiment shows that in the order given, user time is 0.22 seconds; with interchanged order, user time is 1.13 seconds.
4. We no longer teach assembler-language programming, because compilers are much better and more patient than assembler-language programmers.
5. But you need to understand computation at the assembler level.
   (a) When your program has a bug, high-level models can fail.
   (b) To improve performance, you need to understand what optimizations the compiler can and cannot do.
   (c) When you write system software, what actually runs is machine code.

(d) Operating systems need to deal with the intricacies of machine code as they manipulate processes (keeping track of floating point registers, the program counter, memory mapping tables)
(e) Malware is often written in x86 assembler.

4 Memory-referencing bugs

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

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

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

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

   (b) $\boxed{\text{Lecture 2, 1/16/2015}}$: lab 1
   (c) $\boxed{\text{Lecture 3, 1/21/2015}}$
   (d) Dereferencing invalid pointers
   (e) Improper allocating and deallocating memory regions
   (f) Unfortunately, the symptoms may be unrelated to the causes, and the effect might be visible only long after it is generated.
   (g) Some languages prevent such errors: Java, Ruby, Haskell, ML, but they are not usually used for systems programming.

(h) There are tools to help you detect referencing errors (such as valgrind).

5 Binary representation

1. Bits are represented by two voltages, one for 0 and another for 1. A typical representation would be .3V for 0 and 3.0V for 1, but every architecture bases the values on the particular kind of transistors it uses. When a voltage changes from one value to the other, there is an intermediate time at which its value is indeterminate; hardware carefully avoids inspecting the voltage then.
2. We usually think of numbers in decimal (base 10), but for systems programming, we sometimes need to use binary (base 2) or hexadecimal (base 16) representation.
   (a) Base 2 uses only digits 0 and 1. Represent, for instance, 5.25 as 101.01_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. $285.3 = 11\mathrm{D}.4\mathrm{CCCCCCCC}_{16} \ldots$
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. It's pretty easy to see how many distinct values you can store in n bits. Since every bit can be 0 or 1, there are 2^n possibilities. Luckily, $2^{10} \approx 10^3$, so $2^{32} = 2^2 \times 2^{30} = 4 \times (2^{10})^3 \approx 4 \times (10^3)^3 = 4 \times 10^9 = 4$ billion. Or just remember that
   $2^{10} = 1024 \approx 10^3 = 1$ 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 $2^{32} = 4\mathrm{G}$.

6. What is the largest signed integer you can store in 16 bits? (Answer: $2^{15} - 1 = 32767$)
7. How many bits do you need to store 4893? It's about 4×10^3, so about 12 bits. (The right answer is 13 bits, but in a 2's complement representation, at least 14 bits.)

6 C types and their sizes

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

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 !

7 Byte ordering

1. If a word has more than one byte, what is the order?
2. Example: $19088743 = 01234567_{16}$
3. Big-endian: Least significant byte has the highest address. (Sun, PPC, Mac, Internet). The bytes, in order, are 0x01, 0x23, 0x45, 0x67.
4. Little-endian: Least significant byte has the lowest address. (x86). The bytes, in order, are 0x67, 0x45, 0x23, 0x01.
5. You can use the `od` program to show a file in bytes, characters, integers, ...
6. $\boxed{\text{Lecture 4, 1/23/2015}}$ Lab 2
7. $\boxed{\text{Lecture 5, 1/26/2015}}$

8. 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");
}
int a = 15213;
printf("int␣a␣=␣%d;\n", a);
show_bytes((pointer) &a, sizeof(int));
```

   The output is:

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

8 Memory organization

1. We usually address memory in **bytes**, although older computers used to measure in "words", which could be of any length (PDP-10: 36 bits per word).
2. When a program is running, we call it a **process**.
3. From a process's point of view, memory looks like a long array, starting at byte address 0 and going to some limit determined by the operating system. (On Linux for x86, memory is limited to 3GB.)
4. The operating system creates a separate address space for each process. We say that process address spaces are **virtual**, because when a process refers to address n, it is very likely not at physical address n.
5. Because processes get individual address spaces, they cannot read or write in each other's address spaces, although the operating system can also arrange for some sharing.

6. The operating system allocates **physical** space, which also looks like an array ranging from address 0 to a limit determined by how much physical memory the machine has.
7. Within a process, the program uses memory for various purposes. The compiler decides where in memory to put various items, including the instructions, initialized data, uninitialized data, stack, and heap.
8. A 32-bit architecture generally means that integers are contained in 32 bits, and that virtual addresses use 32 bits (unsigned). The maximum address is therefore 4G-1. That memory size is too small for some applications.
9. A 64-bit architecture generally means that integers are contained in 64 bits, and that virtual addresses use 64 bits (unsigned). The maximum address is therefore about 1.8×10^{19}. The x86_64 architecture supports only 48-bit addresses, which gives 256TB.
10. Architectures generally support multiple data formats. So a 64-bit architecture might be able to manipulate 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit integers.

9 Strings and Buffers

1. A C **string** is an array of bytes, each representing a single character, terminated by a null (zero) byte.
2. Declaration
   (a) `char *myString;`
   (b) `char myString[];`
   (c) `char myString[200];`
3. The representation is typically 7-bit ASCII.
4. Some representations, such as UTF-8, might use several bytes for a single Unicode character. So the length of the array is not necessarily the number of characters.
5. There is no need to worry about byte ordering; the start of the string always has the lowest address in memory.

6. A **buffer** is also an array of bytes, typically used to hold data subject to I/O. The bytes hold arbitrary binary values, not necessarily printable values.
7. Declaration
   (a) `char *myBuffer;`
   (b) `char myBuffer[];`
   (c) `char myBuffer[4096];`
8. Buffers are not null-terminated; you need a separate variable to remember how much data is in the buffer.

10 Boolean algebra

1. $\boxed{\text{Lecture 6, 1/28/2015}}$
2. Named after George Boole (1815–1864).
3. A computer's circuitry uses pieces that accomplish Boolean functions in order to build both combinatorial and sequential circuits.
4. We are familiar with the *truth tables* for **and** (in C: `&`), **or** (`|`), **not** (`~`). We might not be familiar with **exclusive or** (**xor**, `^`).
5. When one operates on bytes (or larger chunks such as integers) with Boolean functions, they are applied bitwise. Examples:

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

6. Instead of interpreting 32 bits as an integer, we can interpret it as a subset of $\{0, \ldots, 31\}$. 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.

7. One can use the Boolean operators in C and apply them to any integral data type: **`char`**, **`short`**, **`int`**, **`long`**, **`long long`**.
8. Don't confuse these operators with logical operators **`&&`**, **`||`**, and **`!`**. In C, 0 is understood to mean *false*, and any other value is *true*. The logical operators always return either 0 or 1. They use short-circuit semantics.

11 Shifting operators

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

12 Encoding integers

1. Given n bits, there are 2^n possible integers.
2. In unsigned form, these range from 0 (represented as all 0 bits) to $2^n - 1$ (represented by all 1 bits).
3. In signed form, these range from -2^{n-1} to $2^{n-1} - 1$.
   (a) 0 is represented by all 0 bits.
   (b) The most significant bit is 0 for positive numbers. The largest positive number has $n - 1$ bits set, representing $2^{n-1} - 1$.
   (c) The most significant bit is 1 for negative numbers. The smallest negative number has only that bit set, representing -2^{n-1}.
   (d) To negate a number (either a positive or negative one), invert all the bits and add 1 to the result, ignoring overflow.
   (e) 4-bit examples
      i. $0000_2 = 0$
      ii. $0110_2 = 6$
      iii. $1010_2 = -6$

iv. $1000_2 = -8$. Try to negate this number.
v. $0111_2 = 7$.
vi. $1001_2 = -7$.

13 Compilation and disassembly in Linux

1. Compiler for C: `gcc`. Compiler for C++: `g++`.
2. Command-line options are by Unix convention marked with `-`.

`-o` *filename*	put the output of compilation in *filename*
`-E`	Don't compile; just run the preprocessor
`-S`	Compile but don't assemble; result is *filename*.s
`-c`	Compile and assemble, but don't link; result is *filename*.o
`-g`	Add debugging information to the result.
`-O`*n*	Turn on optimization level n (from 0 to 3)

3. Tools to inspect compiled code
   (a) `objdump -d` *filename*: disassembles *filename*
   (b) `gdb` *filename*: runs the debugger on *filename*; can disassemble
   (c) `nm` *filename*: shows location and type of identifiers in *filename*
   (d) `strings` *filename*: shows all the ASCII strings in *filename*.
   (e) `od` *filename*: displays *filename* in numeric or character format.
   (f) `ldd` *filename*: tells what dynamic libraries *filename* uses.
   (g) `dissy` *filename*: graphical tool to inspect *filename*. You can install *dissy* by using `apt-get`.

14 Machine basics

1. $\boxed{\text{Lecture 7, 1/30/2015}}$ Lab 3
2. $\boxed{\text{Lecture 8, 2/2/2015}}$
3. An **architecture**, also called an **instruction-set architecture** (ISA), is the part of a processor design that you need to know to read/write assembler code. It includes the instruction set and the characteristics of registers. Examples: x86 (also called IA-32), IPF (also called IA-64: Itanium), x86_64.

4. The **microarchitecture** describes how the architecture is implemented. It includes the sizes of caches and the frequency at which the machine operates. It is not important for programming in assembler.
5. Components of importance to the assembler programmer
   (a) **Program counter** (PC): a register containing the address of the next instruction. Called EIP (x86) or RIP (x86_64)
   (b) **Registers**, used for heavily-accessed data, with names specific to the architecture. The set of all registers is sometimes called the **register file**.
   (c) **Condition codes** store information about the most recent arithmetic operation, such as "greater than zero", useful for conditional branch instructions.
   (d) **Memory**, addressed by bytes, containing code (also called "text" in Unix), data, and a stack (to support procedures).

15 Steps in converting C to object code

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

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

8. Generated x86 assembler (using `-O1`)

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

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

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

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

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

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

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

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

16 IA32 = x86 architecture

1. 32-bit architecture.
2. You can compile for it even on an x86_64 with the `-m32` flag.

3. Registers

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

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

5. Operand types

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

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

   (c) $\boxed{\text{Lecture 9, 2/4/2015}}$

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

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

6. Example in C: Swap

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

7. Same thing in assembler

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

17 More complex memory-addressing modes

(a) We saw **memory** and **displacement**.

(b) This is the most general form: *D(Rb,Ri,S)*

   i. *D* is the displacement, in bytes, such as 1, 2, 80.
   ii. *Rb* is the base register: any of the 8 integer registers.
   iii. *Ri* is the index register, any register by `%esp`, and you most likely don't want to use `%ebp`, either
   iv. *S* is a scale, which is any of 1, 2, 4, or 8.

(c) The value it references is Mem[Reg[*Rb*]+*S**Reg[*Ri*]+*D*].

(d) Example

   i. `%edx`: `0xf000`
   ii. `%ecx`: `0x0100`
   iii. `0x8(%edx)`: `0xf000 + 0x8 = 0xf008`
   iv. `(%edx,%ecx)`: `0xf000 + 0x0100 = 0xf100`
   v. `(%edx,%ecx,4)`: `0xf000 + 4*0x0100 = 0xf400`
   vi. `0x80(,%edx,2)`: `2*0xf000 + 0x80 = 0x1e080`

18 Address computation without referencing

(a) One can compute an address and save it without actually referencing it.

(b) `leal` *src*, *dest*

(c) Load Effective Address of *src* and put it in *dest*.
(d) Purpose: translate `p = &x[i]`
(e) Purpose: compute arithmetic expressions like `x+k*y` where *k* is 1, 2, 4, or 8.
  i. Example: `x*12`
  ii.
```
leal (%eax,%eax,2), %eax # x = x+2x
sall $2, %eax # x = x << 2
```

19 Arithmetic operations

1. Two-operand instructions

instruction	meaning
`addl`	dest = dest + src
`subl`	dest = dest − src
`imull`	dest = dest × src
`sall`	dest = dest << src
`sarl`	dest = dest >> src (arithmetic)
`shrl`	dest = dest >> src (logical)
`xorl`	dest = dest ⊕ src (bitwise)
`andl`	dest = dest ∧ src (bitwise)
`orl`	dest = dest ∨ src (bitwise)

2. Lecture 10, 2/6/2015
3. Be careful of parameter order for asymmetric operations.
4. There is no distinction between signed and unsigned integers.
5. One-operand instructions

instruction	meaning
`incl`	dest = dest + 1
`decl`	dest = dest − 1
`negl`	dest = −dest
`notl`	dest = ¬ dest (bitwise)

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

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

8. Optimization converts multiple expressions to a single statement, and a single expression might require multiple instructions.
9. The compiler generates the same code for `(x+y+z)*(x+4+48*y)`.
10. Example

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

11. Result of compilation

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

20 Control based on condition codes

1. The four condition codes, each Boolean
   (a) `CF`: Carry flag
   (b) `ZF`: Zero flag
   (c) `SF`: Sign flag
   (d) `OF`: Overflow flag (for signed integers)
2. Arithmetic operations implicitly set these flags, but the `lea` instruction does not set them.
3. Example: addition $t = a + b$
   (a) sets `CF` if there is a carry from most significant bit
   (b) sets `ZF` if the t is zero
   (c) sets `SF` if the top bit of t is set ($t < 0$)
   (d) sets `OF` if there is a two's complement overflow:
   $(a > 0 \wedge b > 0 \wedge t < 0) \vee (a < 0 \wedge b < 0 \wedge t > 0)$
4. The compare instruction (`cmpl b, a`) also sets the flags; it's like computing $a - b$ without modifying the destination.
   (a) sets `CF` if there is a carry from most significant bit
   (b) sets `ZF` if $a = b$
   (c) sets `SF` if $a - b < 0$
   (d) sets `OF` if there is a two's complement overflow:
   $(a > 0 \wedge b < 0 \wedge (a - b) < 0) \vee (a < 0 \wedge b > 0 \wedge (a - b) > 0)$

5. The test instruction (`testl b, a`) also sets the flags; it's like computing $a\&b$ without modifying the destination. Usually, one of the two operands is a mask.
   (a) sets `ZF` if $a \wedge b = 0$
   (b) sets `SF` if $a \wedge b < 0$
6. Many instructions in the `setXX` *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 `jXX` *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. Lecture 11, 2/9/2015
9. Example

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

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

10. Example:

```
int absDiff(int x, int y)
{
  int result;
  if (x <= y) goto elsepoint;
  result = x-y;
  goto exitpoint;
elsepoint:
  result = y-x;
exitpoint:
  return result;
```

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

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

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

21 do `while` loops

1. 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;
}
```

2. Same thing, represented with **`goto`**

```
int countOnes(unsigned x)
{
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

3. Partial assembler listing

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

4. **`for`** loops are very similar
5. The compiler can generate better code by replacing the unconditional jump at the end of the loop with a conditional jump.

22 Procedures

1. $\boxed{\text{Lecture 12, 2/11/2015}}$
2. In order to handle recursion, languages are compiled to use a stack.
3. Each invocation of a procedure pushes a new frame on the stack.
4. When the procedure returns, the frame is popped and the space it occupied is available for another procedure.
5. The frame contains storage private to this instance of the procedure.
   (a) return address

   (b) parameters
   (c) local variables
   (d) temporary locations (that don't fit in registers)

6. On the x86, the `%epb` register points to the start of the frame, and the `%esp` register points to the current top of the stack.
7. The stack (for Unix, at least), grows *downward*.
   (a) `pushl` *src* subtracts 4 from `%esp` and writes the operand at the new address.
   (b) `popl` *dest* puts `(%esp)` in the destination and then adds 4 to `%esp`.
   (c) `call` *label* pushes the return address and then jumps to the label. The **return address** is the address of the instruction after the `call`.
   (d) `ret` pops the return address and then jumps to it.
8. Example:

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

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

9. Linux C frame contents, starting at bottom (right after caller's frame)
   (a) return address (placed by `call`)
   (b) old `%ebp`
   (c) saved registers and local variables
   (d) parameters for the next call ("argument build"), last parameter first
   (e) `%esp` points to the last parameter.
10. Lecture 13, 2/13/2015

11. Linkage at the calling point for `swap(&course1, &course2);`

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

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

location	meaning
`0(%esp)`	old `%ebp`
`4(%esp)`	return address
`8(%esp)`	first parameter
`12(%esp)`	second parameter

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

23 Register-saving conventions

1. The compiler writer determines what registers are meant to survive procedure calls (”non-volatile registers”) and which can be used for temporary storage by the procedure (”volatile registers”).
2. This convention prevents one procedure call from corrupting another’s data.
3. Say A (the **caller**) is calling B (the **callee**).
   (a) If A has been using a volatile register, it must save it (on the stack) before calling B, and pop it when B returns. This situation is called **caller save**.
   (b) If B needs to use a non-volatile register, it should save it (on the stack) before doing its work and pop it before returning. This situation is called **callee save**.
4. The convention that gcc follows for the x86:
   (a) `%eax`, `%ecx`, `%edx` are volatile (caller-save) general-purpose registers.

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

5. Example

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

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

24 Code for local variables, pointers

1. Lecture 14, 2/18/2015

2. Example

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
void incrk(int *ip, int k) {
  *ip += k;
}
```

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

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

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

25 x86_64 registers

1. Eight upgraded 64-bit registers, now with names starting with `r` instead of `e`, such as `%rax`.
2. Eight new 64-bit registers, called `%r8` ... `%r15`.
3. To use only the lower bits of a register, append a suffix to the register name.

suffix	bits
`b`	8
`w`	16
`d`	32

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

26 Data types

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

Intel	C	assembler	bytes
byte	**`char`**	b	1
word	**`short`**	w	2
double word	**`int`**	l	4
quad word	**`long int`**	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 (x86_64)
extended	**`long double`**	t	10/12/16 (x86_64)

27 Arrays

1. $\boxed{\text{Lecture 15, 2/23/2015}}$
2. C declaration: `T myArray[L]`, where T is some type and L is the number of elements (the first is number 0).
3. Contiguously allocated region of $L \times$ **`sizeof`**(T) bytes.
4. Examples

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

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

expression	type	value
`val[4]`	**int**	3
`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+4i

6. 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.

7. Simple example: **`return`** `uky[dig];`

```
# assume d = uky
# assume a = dig
movl (%edx,%eax,4),%eax # a = uky[dig]
```

8. Loop example

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

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

28 Nested arrays

1.

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

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

```
                           # assume a = index
leal (%eax,%eax,4),%eax # a = 5 * index
leal pgh(,%eax,4),%eax  # a  = pgh + (20 * index)
```

8. $\boxed{\text{Lecture 16, 2/25/2015}}$
9. General addressing case: *T* `A[`*R*`][`*C*`];` defines a two-dimensional array of type *T* with *R* rows and *C* columns. Say type *T* requires *k* bytes. Then the location of `A[i][j]` is $A + kiR + kj = A + k(iR + j)$.
10. Example:

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

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

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

29 Structures

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

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

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

```
y   x
i   x+12
n   x+16
end x+20
```

3. The compiler knows all the offsets.

30 Linked lists

1. Example:

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

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

31 Alignment

1. Example:

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

2. The character `c` uses only 1 byte, so the array `y` starts at offset 1, and `v` at offset 9.
3. But the x86 advises that n-byte primitive data should start at an address divisible by n, that is, it should be **aligned** to such an address.
   (a) 1 byte (char): any address

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

4. The x86_64 is stricter
   (a) 8 bytes (double, void *): address ends with 000_2
   (b) 16 bytes (long double): Linux/x86 chooses 000_2.
5. On some machines alignment is mandatory.
6. Motivation
   (a) The CPU accesses memory in chunks of 4 or 8 bytes (architecture-dependent).
   (b) It is inefficient to access a datum that crosses chunk boundaries.
   (c) It is tricky to access a datum that crosses page boundaries (typically every 4KB).
7. The compiler can add *padding* to accomplish this requirement:

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

8. The compiler can also re-order the fields (biggest first, for instance) to reduce the amount of padding.
9. The entire **`struct`** needs to be padded to a multiple of the largest primitive data within the **`struct`**, so that arrays of such **`struct`**s work.

32 Midterm test 3/2/2015

33 Discussion of midterm test

1. $\boxed{\text{Lecture 17, 3/4/2015}}$

34 Unions

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

```
union {
  char c[8];  // 8*1 = 8 bytes
  short s[4]; // 4*2 = 8 bytes
  int i[2];   // 2*4 = 8 bytes
  long l[1];  // 1*8 = 8 bytes (on i86_64)
  float f[2]; // 2*4 = 8 bytes
} dw;
dw.f[0] = 3.1415;
printf("as␣float:␣%f;␣as␣integer:␣%d;\n"
        "as␣two␣shorts:␣%d,␣%d\n",
        dw.f[0], dw.i[0], dw.s[0], dw.s[1]);
```

Result:

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

35 Linux x86 memory layout

1. Simplified version of allocation of 4GB virtual space

start	name	purpose	properties
0	unused	prevent errors	no access
0x08000000	text	program	read-only
	data	initialized data	read, write; static size
	bss	uninitialized data	read, write; static size
	heap	allocatable data	read, write; grows up
	stack	activation records	read, write; grows down
0xC0000000	kernel	kernel code, shared	no access

2. The limit program shows per-process limitations; by default, for instance, the stack is limited to 8MB.

36 Buffer overflow

1. Underlying problem: library functions do not check sizes of parameters, because C array types don't specify length.
2. Lecture 19, 3/11/2015
3. Which functions: `gets()`, `strcpy()`, `strcat()`, `scanf()`, `fscanf()`, `sscanf()`.
4. Effect of overflowing a local array (on the stack): overwriting return address.
   (a) If the return is to an address not in text or stack space, causes a segmentation fault.
   (b) The return address can be to code on the stack that is part of the overflowing buffer, leading to execution of arbitrary code.
5. Internet worm (November 1988): the *fingerd* program used `gets()` to read a command-line parameter; by exploiting a buffer overflow, the worm got *fingerd* to run a root shell with a TCP connection to the attacker.
6. There are hundreds of other examples.
7. Avoiding vulnerability
   (a) Use library routines that limit lengths: `fgets()`, `strncpy()`, `scanf(...%ns...)`.
   (b) Randomized stack offsets: allocate a random amount of stack space as the program starts. Then the attacker cannot guess the start of the buffer, so it is harder to fake the return address to jump into the buffer.
   (c) Nonexecutable segments: On the x86, anything readable is executable, including the stack. On the x86_64, there is separate executable permission.
   (d) Stack canaries: put a canary value on stack just beyond each buffer; check for corruption as part of linkage during return. In

gcc, use `-fstack-protector` (adds code to evidently suspicious routines) or `-fstack-protector-all` (adds code to all routines)

```
8048654: 65 a1 14 00 00 00    mov %gs:0x14,%eax
804865a: 89 45 f8             mov %eax,0xfffffff8(%ebp)
804865d: 31 c0                xor %eax,%eax
...
8048672: 8b 45 f8             mov  0xfffffff8(%ebp),%eax
8048675: 65 33 05 14 00 00 00 xor  %gs:0x14,%eax
804867c: 74 05                je   8048683 <echo+0x36>
804867e: e8 a9 fd ff ff       call 804842c <FAIL>
```

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

37 Linking

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

(a) Programs define symbols and reference them:

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

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

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

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

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

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

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

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

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

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

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

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

6. Shared object files

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

(b) Libraries are usually shared object files.

(c) When the linker resolves a identifier by referring to a shared object file, it leaves it dangling (but resolved); full resolution happens when the shared object file is loaded into memory.

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

7. $\boxed{\text{Lecture 21, 3/23/2015}}$
8. Relocation

   (a) The linker combines the object files into a single file.
   (b) All text segments are placed together; similarly for other segment types.
   (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.

38 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. Sections of an ELF file.

   (a) header: word size, byte ordering, object-file type, architecture
   (b) segment header table page size, segment sizes.
   (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 table (`.rel.text` section): addresses in `.text` that need to be modified for relocation and how to relocate them.
   (i) data relocation table (`.rel.data` section): addresses in `.data` that need to be modified for relocation and how to relocate them.
   (j) debugging information (`.debug` section), produced, for instance, with `gcc -g`.
   (k) section header table: offsets and sizes of each section

4. **Global symbols**: defined in this module, may be referenced by other modules. In C: functions (except **`static`**) and file-global variables (except **`static`**).
5. **External symbols**: global symbols referenced by this module but defined in another module.
6. **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. **`extern`**.

39 Unix tools for object files

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

40 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`**.

41 Libraries

1. $\boxed{\text{Lecture 22, 3/25/2015}}$
2. C library (`libc`): about 4MB (static), 2MB (dynamic); about 1600 symbols; routines for I/O, memory allocation, signals, strings, random numbers, integer mathematics.
3. Math library (`libm`): about 1MB; about 400 symbols; floating-point math.
4. The linker only uses the symbols from 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
```

42 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
   ```

43 Operating systems — Introduction

1. $\boxed{\text{Lecture 23, 3/27/2015}}$ Lab 5
2. $\boxed{\text{Lecture 24, 3/30/2015}}$
3. Goals
   (a) abstract the hardware, hiding some features and providing new ones.
   (b) share resources among processes.
4. Layers
   (a) Hardware, including instruction set, registers, memory, and devices.
   (b) OS kernel
   (c) Applications (processes)
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.

44 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 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 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.

(d) When the kernel is ready to return from the exception:
  i. The CPU changes to its previous mode (typically user mode) and its previous location.
  ii. The saved information is popped from the stack.

45 Example: opening a file in Linux x86

1. $\boxed{\text{Lecture 25, 4/1/2015}}$
2. In C, a program invokes `open(filename, options)`.
3. The C library handles this function by executing an interrupt instruction: `int $0x80`.
4. 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 is in a valid location in memory.
   (b) verify that the file exists.
   (c) verify that the file permissions allow this process to open it in th manner specified by the options.
   (d) build a kernel data structure `files_struct`.
   (e) return the index of that structure within the kernel's per-process `fd_array`.
5. 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*.

46 Example: Page fault

1. A program tries to access a memory address that is in its virtual space but which is not currently available.
2. The MMU (memory management unit) notices the problem and generates a **page fault**.
3. The Linux kernel executes a procedure called `fault()`. It takes possibly complex action to make sure the memory address is available.
4. The kernel then returns to the same instruction that caused the fault.

47 System calls invoked by C library routines

1. Many C library routines include system calls.
   (a) `printf()` calls `write()` which executes a system call.
   (b) `malloc()` might call `sbrk()` to increase the space for the heap.

48 Processes

1. A **process** is an instance of a running program.
2. Each process has its own virtual memory, which it can treat as its own private domain.
3. Each process may treat the CPU as its own as well.
4. The kernel hides that fact that the processes are sharing both memory and the CPU.
5. The kernel **schedules** the processes so that they each get a "fair" share of the CPU.
   (a) Each process can pretend that it has the full use of the CPU: disk and I/O operations seem immediate, and other processes don't interfere.
   (b) In fact, the kernel often stops running process A and starts running process B. This **process switch** involves a **context switch** from A to the kernel, a scheduling decision, and then a **context switch** from the kernel to B.
6. $\boxed{\text{Lecture 26, 4/3/2015}}$
7. The kernel arranges memory maps so that the MMU hardware can redirect all memory accesses that each process makes.
8. The `fork()` system call asks the kernel to create a new process.
   (a) Lab 4 showed this material.
   (b) Say A submits the call.
   (c) The kernel builds a new process B.
   (d) B shares all of A's code and has a new copy of all of A's data and stack. B's program counter (PC) is identical to A's.
   (e) Therefore both A and B see a return from **fork()**.

   (f) The only difference is the return value. For A, it is B's process identifier (PID). For B, it is 0.

9. The `exit()` system call terminates the calling process.

   (a) The return value is the status. By convention, 0 means success, and any other value is a failure code.
   (b) The program can register a cleanup function to call at exit by using atexit(); one may register many such functions.

10. The `wait()` system call **reaps** the terminated child, collecting its termination status and freeing its space in kernel data structures.

    (a) Until the parent calls **wait()**, the child is a **zombie**.
    (b) If the parent terminates before reaping its children, the `init` process reaps them.
    (c) Ordinary `wait()` blocks the parent until some child terminates; `waitpid()` waits for a specific PID.
    (d) `wait()` returns an integer that can be queried by `WIFEXITED` and `WEXITSTATUS`, among other macros.

11. A process can completely reload itself to run a different program by using one of the `exec()` family of calls, such as `execl()`.

    (a) The parameters to `execXX()` include a string indicating the file holding the executable object file, a list of 0 or more parameters to that program, and, optionally, an array of pointers to strings indicating environment values of the form *var=value*.
    (b) The initial stack, when `main()` is called, contains the (1) environment strings, (2) the command-line parameters, (3) pointers to 1, (4) pointers to 2, (5) a pointer to 3, (6) a pointer to 4, a count of parameters.

49 Processes in Unix

1. Process 1 is always **init**.

   (a) Starts *login* on each terminal.
   (b) Starts **daemons** to do background work.
   (c) Reaps all orphaned children when they exit.

2. Lecture 27, 4/6/2015
3. *login* starts a **shell** for each logged-in user.
   (a) *sh*: original shell, Stephen Bourne, 1977
   (b) *csh*: BSD Unix C shell. *tcsh*: enhanced at CMU.
   (c) *bash*: Bourne-again shell.
   (d) *ash*: Almquist shell, 1997 *dash*: POSIX standard.
4. The shell runs a read-eval loop.
   (a) Primarily it runs commands as processes (using `fork()` and `exec()`.
   (b) It waits for commands to finish (using `wait()`), or not, if they have been started **in the background**.
   (c) A user has a limited number of processes (built-in `limit` command)

50 Signals

1. When a process terminates, the kernel sends its parent a software interrupt.
2. Software interrupts are called **signals**.
3. There are about 30 possible signals, each with a different purpose. see *signal(7)*.
4. Each signal has its default action. For instance, `SIGSEGV` terminates with a dump file; `SIGCHLD` (a child has stopped or terminated) is ignored.
5. A process may explicitly send a signal to another process by the `kill(2)` system call. The sending and target processes must be owned by the same user, or the sending process must be owned by root.
6. The */bin/kill* command sends a given signal to a given target process (or process group).
7. The user can send signals to processes by some keyboard methods: ctrl-c sends `SIGINT`; ctrl-z sends `SIGSTOP`.
8. For most signals, a process can choose in advance what action to take when receiving them, but using `signal(2)`.

   (a) Terminate
   (b) Ignore
   (c) Terminate and create a dump file (called *core*).
   (d) Temporarily stop the process.
   (e) Continue the process from a temporary stop.
   (f) **Catch** the signal by **installing** a procedure in the process called a **signal handler**.
   (g) Temporarily block the signal (the signal becomes **pending**; further identical signals are ignored while it is pending).
   (h) Unblock a pending signal

9. When a signal handler runs, the process is temporarily interrupted; it resumes when the signal handler finishes.

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
void handler(int sig) {
  safe_printf("You␣hit␣ctrl-c\n");
  sleep(2);
  safe_printf("Well...");
  sleep(1);
  printf("OK\n");
  exit(0);
}
main() {
  signal(SIGINT, handler); /* install handler */
  while(1) { // busy loop
  }
}
```

10. If an arriving signal interrupts a long system call (such as `read()`), different versions of Unix have different behaviors when the signal handler finishes: Continue the long system call (Linux), abort the long system call with an `EINTER` error.
11. A function is **async-signal-safe** if all its variables are local (so it is **reentrant**) or non-interruptible.

    (a) Such functions may be called even from inside a handler.

(b) The POSIX standard requires 117 functions, such as `write()`, to be safe. But `printf()` is not safe.

51 Nonlocal jumps

1. Nonlocal jumps provide a powerful but dangerous method to transfer control to an arbitrary location.
2. `int setjmp(jmp_buf jb)`: saves a "snapshot" in `jmp_buf`. It may return multiple times; the first time it returns 0.
3. The snapshot contains register information, including pc and sp.
4. `void longjmp(jmp_buf jb, int val)`: causes a return from the associated **setjmp**, with `val` as the return value.

52 Files

1. Unix files are sequences of bytes. The kernel does not distinguish files built for various purposes (text, data, object) when a process reads or write them, no matter what the file name might be.
2. Even non-file abstractions are made to look like files to allow processes to treat them in a consistent way.

   (a) Even devices look like files. They have names like `/dev/tty1` and `/dev/sda2`. Some are **character special**, like terminals. Others are **block special**, like disks.
   (b) The kernel memory can be seen as a file: `/dev/kmem`.
   (c) Information about processes look like files: `/proc/`
   (d) Directories are special, in that they are not read/written using `read()` and `write()`. Directories contain names of files, building a hierarchical file arrangement.
   (e) **Pipes**, used for inter-process communication, also look like files.
   (f) A **FIFO** is like a pipe, but it has a name in the file system.
   (g) A **socket** is used for cross-machine inter-process communication. Processes usually use `recv()` and `send()`, not `read()` and `write()`.

3. Basic I/O system calls. Always check the return value!

(a) `open()` and `close()`: `open()` returns a **file descriptor**; the kernel allocates and initializes a data structure. `close()` lets the kernel deallocate the structure.

(b) `read()` and `write()`

```
#include <unistd.h> // defines ssize_t: signed integer
char buf[512];
int fd1, fd2;
ssize_t nbytes;
if ((fd1 = open("/etc/hosts", O_RDONLY) < 0)) {
        perror("/etc/hosts");
        exit(1);
}
if ((nbytes = read(fd1, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
if ((fd2 = creat("/tmp/hosts", 0664)) < 0)) {
        perror("/tmp/hosts");
        exit(1);
}
if ((nbytes = write(fd2, buf, nbytes) != nbytes) {
   perror("write");
   exit(1);
}
```

A value of `nbytes` less than `sizeof(buf)` is valid.

(c) `lseek()`

53 Sockets

1. Lecture 28, 4/8/2015
2. A **socket** is an abstraction that lets two processes communicate, even if they are on different machines.
3. Setting up a socket is asymmetric; the **client** and the **server** do different things, after which both can `read()` and `write()`.
4. Client

   (a) `socket()` to create a local file descriptor (like "buy a phone")

   (b) `connect()` to attach the socket to the server (like "call a number").

5. Server
   (a) `socket()` to create a local file descriptor ("buy a phone").
   (b) `bind()` to bind a name to the socket ("get a phone number").
   (c) `listen()` to wait for a client to connect ("plug in the phone")
   (d) `accept()` to create a file descriptor for this connection ("answer the phone")

6. The book provides some nicer routines that handle errors and combine operations.
   (a) Client: `open_clientfd()` (combines `socket()` and `connect()`)
   (b) Server: `open_listenfd()` (combines `socket()`,`bind()`,and `listen()`), then `accept()`

7. Generic socket address structure (used for `connect()`,`bind()`,and `accept()`

```
struct sockaddr {
 unsigned short sa_family;   /* protocol */
 char           sa_data[14]; /* address data */
};
```

8. Specific socket address structure for the internet

```
struct sockaddr_in {
  unsigned short sin_family;  /* AF_INET */
  unsigned short sin_port;    /* net order */
  struct in_addr sin_addr;    /* net order */
  unsigned char  sin_zero[8]; /* pad */
};
```

9. Sample client for ”echo”

```
int client(int argc, char **argv) {
  int toserverfd, port;
  char *host, buf[MAXLINE];
  rio_t rio;
  if (argc != 3) {
    fprintf(stderr, "Usage:␣%s␣host␣port\n", argv[0]);
    exit(1);
  }
  host = argv[1]; port = atoi(argv[2]);
  toserverfd = open_clientfd(host, port);
  Rio_readinitb(&rio, toserverfd);
  printf("type:␣"); fflush(stdout);
  while (Fgets(buf, MAXLINE, stdin) != NULL) {
    Rio_writen(toserverfd, buf, strlen(buf));
    Rio_readlineb(&rio, buf, MAXLINE);
    printf("echo:␣");
    Fputs(buf, stdout);
    printf("type:␣"); fflush(stdout);
  }
  Close(toserverfd);
  exit(0);
} // client
```

10. Implementation of `open_clientfd`

```
int open_clientfd(char *hostname, int port) {
  int toserverfd;
  struct hostent *serverHostEntry;
  struct sockaddr_in serveraddr;
  if ((toserverfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno */
    // AF_INET: internet; SOCK_STREAM: reliable; 0: normal
  /* Fill in the server's IP address and port */
  if ((serverHostEntry = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  bcopy((char *)serverHostEntry->h_addr_list[0],
  (char *)&serveraddr.sin_addr.s_addr, serverHostEntry->h_length);
  serveraddr.sin_port = htons(port);

  /* Establish a connection with the server */
  if (connect(toserverfd, (SA *) &serveraddr, sizeof(serveraddr))
    return -1;
  return toserverfd;
} // open_clientfd
```

11. Sample server for ”echo”

```
int server(int argc, char **argv) {
  int listenfd, connfd, listenPort;
  struct sockaddr_in clientAddr;
  struct hostent *clientHostEntry;
  char *clientIP;
  unsigned short clientPort;
  listenPort = atoi(argv[1]);
  listenfd = open_listenfd(listenPort);
  while (1) {
    socklen_t addrLength = sizeof(clientAddr);
    connfd = Accept(listenfd, (SA *)&clientAddr, &addrLength);
    if (connfd < 1) perror("accept␣error");
    // the following is optional to get info on the client
    clientHostEntry = Gethostbyaddr(
      (const char *)&clientAddr.sin_addr.s_addr,
      sizeof(clientAddr.sin_addr.s_addr),
      AF_INET);
    clientIP = inet_ntoa(clientAddr.sin_addr);
      // net-to-ASCII string in dotted notation
    clientPort = ntohs(clientAddr.sin_port);
    printf("server␣connected␣to␣%s␣(%s)␣on␣my␣new␣clientPort␣%u\n"
      clientHostEntry->h_name, clientIP, clientPort);
    // end of optional part
    echo(connfd);
    Close(connfd);
  } // while(1)
} // server
```

12. Implementation of `open_listenfd`

```c
int open_listenfd(int port) {
  int listenfd, optval=1;
  struct sockaddr_in serveraddr;

  /* Create a socket descriptor */
  if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;
    // AF_INET: internet; SOCK_STREAM: reliable; 0: normal

  /* Allow reuse of address */
  if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
           (const void *)&optval , sizeof(int)) < 0)
    return -1;

  /* let listenfd be an endpoint for all requests
     to port on any IP address for this host */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
  serveraddr.sin_port = htons((unsigned short)port);
  if (bind(
       listenfd,
       (SA *)&serveraddr,
       sizeof(serveraddr)) < 0)
    return -1;

  /* Ready listenfd to accept connection requests */
  if (listen(listenfd, LISTENQ) < 0)
    return -1;
  return listenfd;
} // open_listenfd
```

54 More about files

1. Lecture 29, 4/13/2015
2. Short counts: when the amount of data read/written is not the size of the buffer.

   (a) Read: reached the end of file, or the end of a line with fgets().
   (b) Network sockets or pipes: read or write can give a short count.
   (c) Never from writing files.
   (d) The Rio (robust I/O) package from `csapp.c` handles short counts and retries.

3. File **metadata**: data about the file, but not the contents.

   (a) Maintained by the kernel in its data structures.
   (b) A program can get some of it with `stat()` or `fstat()`:

```
struct stat {
  dev_t     st_dev;
/* ID of device containing file */
  ino_t     st_ino;
/* inode number */
  mode_t    st_mode;
/* protection and file type */
  nlink_t   st_nlink;
/* number of hard links */
  uid_t     st_uid;
/* user ID of owner */
  gid_t     st_gid;
/* group ID of owner */
  dev_t     st_rdev;
/* device ID (if special file) */
  off_t     st_size;
/* total size, in bytes */
  blksize_t st_blksize; /* blocksize for filesystem I/O */
  blkcnt_t  st_blocks;
/* number of 512B blocks allocated */
  time_t    st_atime;
/* time of last access */
  time_t    st_mtime;
/* time of last modification */
  time_t    st_ctime;
/* time of last status change */
};
```

4. Directories used to be just files with a special format, but now one reads them with different system calls: `opendir()`, `readdir()`,

`closedir()`. `readdir()` returns a **struct** `dirent` pointer, which includes the name of the file.

5. Directories associate file names with *inode numbers*, which uniquely index the on-disk metadata for each file.
   (a) Multiple directories can point to the same inode, in which case a single file might have multiple path names.
   (b) The on-disk inode includes a count of how many directories point to it.
   (c) The `ln` command introduces such a **link** between two files.
   (d) To remove a name from a directory, use the `unlink()` system call.
   (e) When the last reference to a file is removed, the kernel can reclaim the file's allocated space on the disk.
6. Each open file includes a *file position* indicator for each open file; it is advanced by `read()` and `write()` and a program can modify it with `lseek()`.
7. $\boxed{\text{Lecture 30, 4/15/2015}}$
8. Multiple processes may have the same file open.
   (a) If they opened the file independently, each has its own file position.
   (b) If they received the file descriptor by inheritance, they share the file position.
9. The standard I/O library (**#include** `<stdio.h>`) provides buffered versions of input and output.
   (a) Instead of file descriptors (small integers), it uses pointers to a `FILE` structure.
   (b) It predefines `stdin`, `stdout`, `stderr`.
   (c) It only submits a system call when a buffer is full (output) or empty (input) or if there is an explicit request to `fflush()`.
10. Choosing which calls to use in the program (low-level, stdio, or Rio):
    (a) low-level: most general, lowest overhead per call, can access file metadata, async-signal-safe (so they can be used in signal handlers). But dealing with short counts is tricky, and I/O is unbuffered, leading to multiple system calls.

   (b) stdio: buffering increases efficiency, and short counts are handled automatically. But no way to read metadata, not async-signal-safe, not a good idea to use on sockets (there are some poorly documented restrictions).
   (c) Rio: intended for use on sockets.

11. Dealing with binary files (like images and object files)
   (a) Don't use line-oriented functions like `fgets()`,`scanf()`,`printf()`, `rio_readlineb()`.
   (b) Don't use string-oriented functions like `strlen()`; data files may contain null bytes.

55 Virtual memory

1. All memory reads and writes are implemented on two data linking the CPU and memory.
   (a) The **data path** is bidirectional, leading from the CPU through a bus to the memory.
   (b) The **address path** is unidirectional. It goes from the CPU through a **memory-management unit (MMU)**, which modifies the address, then to the bus to the memory.
   (c) The MMU uses tables to modify incoming (virtual) addresses to outgoing (physical) addresses.
   (d) The MMU generates a trap if the tables indicate the virtual address is not mapped or the map does not give permission to read/write/execute.
   (e) The kernel sets up those tables, typically one for privileged mode and one for unprivileged mode.
   (f) $\boxed{\text{Lecture 31, 4/17/2015}}$
   (g) Context switches don't require reloading the tables.
   (h) Process switches do require reloading the unprivileged-mode table.
   (i) The tables themselves are typically in memory, but the MMU keeps a cache of recently-used entries; this cache is called a **translation look-aside buffer (TLB)**.
2. Advantages of virtual memory

   (a) Each process gets a uniform linear address space that does not contain any addresses belonging to any other process or to the kernel.
   (b) The kernel can limit access to parts of that linear address space, making some execute/read-only (the text), other parts modifiable (data, stack), other parts inaccessible (gap between data and stack).
   (c) The kernel can let processes share read-only parts of their address space to save space and loading time.
   (d) It is not necessary for all the virtual memory of a process to be in physical memory for it to run.

3. Cache view of memory
   (a) The entire virtual address space of a process is on disk.
   (b) The currently active part of the virtual address space is cached in physical memory.
   (c) The currently active part of physical memory is cached in L3, L2, or L1 caches (closer to the CPU).
   (d) The compiler attempts to keep the most heavily used information in registers, which are in the CPU itself.

4. MMU tables
   (a) Typically the table is a **page table**.
   (b) Virtual memory is chunked into **pages**, typically 4KB long.
   (c) Each page has an entry in the page table. If addresses are 32 bits long, and a page is 4KB (12 bits) wide, a page table has 2^{20} entries. Each entry contains (x86):
      i. Present/missing (1 bit)
      ii. Page frame (20 bits, gives **page frame number**)
      iii. Whether the page has been recently accessed (1 bit)
      iv. Whether the page has been written and is dirty (1 bit)
      v. Does the process have write permission to the page (1 bit)
      vi. Does the CPU have access permission in non-privileged mode (1 bit)
      vii. Caching strategy (2 bits)
      viii. Normal page size (4KB) or extended (4MB) (1 bit)
      ix. TLB caching advice (1 bit)

   (d) Actually, instead of 2^{20} entries, the table is in two layers, but we ignore that detail for simplicity.
   (e) Adjacent addresses within a virtual page all map to the same page frame in physical memory.

5. MMU algorithm
   (a) Separate the virtual address into higher 20 bits (page number) and lower 12 bits (offset).
   (b) Use the page number as an index into the page table.
   (c) If the page is missing, generate a **page fault** trap.
   (d) If the CPU has no access permission to the page or the CPU wants to write and the page is read/execute only, generate an **invalid access** fault.
   (e) Build a physical address from the page frame (left-shifted 12) plus the offset.
   (f) Success is called a **page hit**.
6. $\boxed{\text{Lecture 32, 4/20/2015}}$
7. The kernel must deal with page faults.
   (a) Some are program errors: attempt to access unallocated memory. The kernel sends the process a SIGSEGV signal, which typically terminates the process.
   (b) Some are attempts to grow the stack. The kernel allocates more stack and re-runs the failed instruction.
   (c) Some are valid accesses, but the needed virtual page is not in memory. The kernel allocates a free page frame, brings the virtual page into memory (from backing store, typically disk), then re-runs the failed instruction.
8. In order to make room for incoming pages, the kernel **swaps out** currently unused pages.
   (a) There are various algorithms for deciding which page to swap out. Let's call it the **victim** page.
   (b) The hope is that the victim is a page that will not be needed soon.

   (c) The general idea is to employ the ”recently accessed” bit in the page table to find a victim that has not been recently accessed, hoping that it won’t be accessed soon.
   (d) If the victim is dirty, it must be written to disk. If it is clean, there is already a good copy on disk.

9. Programs exhibit **locality of reference**, which is why swapping in a page works for a long time, and why processes tend not to need all their virtual pages in memory most of the time. The kernel tries to make sure that at least the **working set** of virtual pages are kept in physical memory.
10. Multi-level page tables
    (a) A table of length 2^{20} is unwieldy. The situation is even worse on an x86_64.
    (b) Therefore, the table is built in multiple levels: 2 levels on the x86, 3 levels on the x86_64.
    (c) The virtual address is composed of three pieces: 10 bits (top level), 10 bits (second level), 12 bits (offset). The top-level bits index an entry in a top-level table of length $2^{10} = 1024$ entries, which is in a single physical page pointed to by a hardware register.
    (d) The top-level entry addresses the second-level page table, also of length $2^{10} = 1024$ entries, fitting in a single page.
    (e) There can be a page fault to access that second-level page table.

56 Free-space allocation

1. `malloc()` and friends maintain data structures that implement the heap.
2. When they run out of memory, they can use the `sbrk()` system call to request more data space.
3. Allocation is usually aligned to 8-byte boundaries.
4. There are several data structures that `malloc()` could use.
   (a) Boundary tag method
      i. Every allocated region has a few bytes before and after, called **tags**, that indicate the length of the region.

   ii. Every free region has tags and also is linked into a list of free regions (the link is in the region itself).
   iii. To allocate, traverse the free list until a good region is found. Split it into an allocated region and a free region, adjusting pointers and tags.
   iv. To deallocate, coalesce the region into the ones before and after if they are free; otherwise, place the region on the free list.

(b) Slab allocator

   i. For every size of allocation, maintain a **slab**, which is a large region that can be subdivided into units of the given size.
   ii. Each slab also has a header indicating the size and which units are currently allocated.
   iii. To allocate, find the right slab and allocate one of its units. If the slab is fully allocated or does not exist, start a new slab.
   iv. To deallocate, mark the unit free. If all units are free in a slab, the entire slab may be deallocated.

(c) $\boxed{\text{Lecture 33, 4/22/2015}}$

(d) Buddy system

57 Garbage collection

1. Explicit `free()` calls are fragile.

   (a) One might forget to free space, leading to a **memory leak**.
   (b) One might free space and keep using it, perhaps through an alias, leading to memory corruption.
   (c) Assuming that newly allocated space is initialized to 0. (If you need that assumption, use `calloc()`.
   (d) Returning a pointer to a local variable.
   (e) Freeing a block more than once.

2. Standard alternative: automatic **garbage collection**.

   (a) Part of the run-time library of many languages, like Java, Perl, Macsyma.
   (b) Let allocation continue until there is insufficient free space.

   (c) Then **mark** all allocated regions that are still in use.
   (d) Then reclaim all allocated regions that are not marked.

3. How to know what is in use: **Mark phase**
   (a) All global variables that point to data structures, and the structures they point to, recursively.
   (b) All local variables, both for the current procedure and all its callers, that point to data structures, recursively.
   (c) Recursion may stop when it encounters an allocated region that is already marked.
   (d) Marking requires that the memory manager always know what fields are pointers, and that fields are always initialized (to `NULL` if necessary).
4. Variants on the basic algorithm
   (a) Generational collection: most regions have a very short life; regions that last after a few collections are placed in a less-frequently collected region.
   (b) Parallelized collection: Collect at the same time the program is running, to prevent sudden stoppage.
5. Debugging memory bugs
   (a) Wrapper around `malloc()`
   (b) glibc `malloc()` uses the environment variable `MALLOC_CHECK`.
   (c) run program under the control of a binary translator: *valgrind*, *purify* (people seem to prefer *valgrind*). Very good at catching memory bugs, but the program runs 30 times slower.

58 Networks

1. $\boxed{\text{Lecture 34, 4/24/2015}}$
2. A **network** is a collection of boxes (**nodes**) and wires (**links**).
   (a) System-area network (**SAN**): a single room.
   (b) Local-area network (**LAN**): a building or campus.
   (c) Wide-area network (**WAN**): country or world

3. An **internetwork** is an interconnected set of networks; the Internet is the best example.
4. Lowest level: ethernet segment
   (a) **Hosts** (computers) are connected via wires (CAT-5 twisted pairs, for instance) to a **hub**.
   (b) Each host has a network interface card (**NIC**) that connects to the bus on one side and the wire on the other side.
   (c) The NIC is rated at some **bandwidth**, typically 10Mbps or 100Mbps.
   (d) Each NIC has a unique 48-bit media access control address (**MAC address**), represented in a fashion like `00:16:ea:e3:54:e6`.
   (e) Hosts send information on the lines in chunks called **frames**.
   (f) The hub copies frames from every port to every other port (**broadcasting**).
   (g) A frame includes a destination NIC (by including its MAC address); only the host to which the frame is addressed takes it.
   (h) Hubs have mostly been supplanted with switches and routers, which do not broadcast.
5. A **bridge** connects internet segments, spanning a building or a campus.
   (a) A bridge is typically a **switch**.
   (b) Switches learn which MAC addresses are reachable on each physical port and then selectively copy frames only to the correct port.
6. A **router** connects multiple LANs together into an internet.
   (a) The different LANs might run different protocols, such as Ethernet and Wifi.
   (b) The router bases its port decision on a higher-level address field, such as in internet protocol (**IP**) address and tables.
7. $\boxed{\text{Lecture 35, 4/27/2015}}$
8. Ad-hoc internets have no particular topology, and the routers and links might have a variety of capacities.
   (a) The routers bridge the network, but they may send similar packets along different routes.

   (b) Software on hosts and routers agree on a common internet protocol independent of the capacities of routers and links.

9. IPv4 is a widely used internet protocol.

   (a) Uniform format for addresses: 32 bits, typically expressed as 4 octets: `128.163.146.21`
   (b) Each host and router has at least one IP address; these addresses are unique across the Internet (exceptions discussed later).
   (c) Uniform data-transfer unit: **packet**, consisting of a **header** and a **payload**.
   (d) The header has fields such as packet size (between 20 and 65,535 bytes), source address, destination address.
   (e) The payload is composed of arbitrary data.
   (f) Packets are embedded in frames for transit across individual links.

10. Complexities covered in a course in computer networking

   (a) Different networks may have different maximum frame sizes, leading to packets becoming fragmented, so the protocol must include an ability to put the fragments back together.
   (b) Packets may arrive out of order at the destination, so the protocol must include an ability to put them back in order.
   (c) Packets may be corrupted in transit or otherwise lost, so the protocol must be able to discover the problem (typically by listening for acknowledgements) and retransmit packets.
   (d) Routers need to know how to forward packets, and they need to keep up to date with network-topology changes.
   (e) The transmission protocol needs to deal with congestion by reducing bandwidth use.

11. TCP/IP protocol family

   (a) IP (internet protocol): addressing scheme, host-to-host unreliable, unordered delivery.
   (b) UDP (unreliable datagram protocol): uses IP for process-to-process unreliable, unordered delivery.
   (c) TCP (transmission control protocol): uses IP for reliable, in-order process-to-process communication over established **connections**.

(d) Unix sockets can be established for either UDP or TCP.

12. Internet components

    (a) **Backbone**: collection of routers connected to each other by high-capacity point-to-point networks. The US has about 50 commercial backbones.
    (b) Network Access Point (**NAP**): Router connecting multiple peer backbones. The US has about 12 NAPs connecting commercial backbones.
    (c) Regional network: small backbone covering a city or a state.
    (d) Point of presence (**POP**): machine connected to the Internet.
    (e) Lecture 36, 4/29/2015
    (f) Internet service provider (**ISP**): provides access to POPs, by dial-up, DSL, cable, T1 line, or some other technology.

13. Shortcomings of IPv4

    (a) Messages are insecure: they are visible to routers (no read protection), the source address can be forged (no authentication), and they can be modified in transit (no write protection).
    (b) There aren't enough IP addresses.

14. Techniques to deal with insufficient addresses

    (a) Dynamic (and temporary) allocation of addresses to POPs by ISPs using the dynamic host configuration protocol (**DHCP**). The ISP only needs enough addresses to handle currently connected POPs.
    (b) Network-address translation (**NAT**): router assigns private addresses (such as `192.168.3.3`) to POPs within its domain, and it modifies the headers of packets in both directions.
    (c) NAT typically refuses incoming packets that are not part of an existing conversation.

15. IPv4 address space is divided into classes of regions

    (a) Class A: first octet in range 0-127. 2^{24} addresses in region.
    (b) Class B: first octet in range 128-191; second octet 0-255. 2^{16} addresses in region.

(c) Class C: first octet 192-224; second and third octet 0-255. 2^8 addresses in region.
(d) Special purpose addresses: first octet 225-255.
(e) This division into classes is no longer followed.
(f) A network ID is written in the form `w.x.y.z/n`, where n is the number of bits in a host address. For instance, CS uses the class C region **128.163.146/8**.
(g) Some regions are pre-allocated as private, unrouted addresses: `10.0.0.0/8`, `172.16.0.0/12`, and `192.168.0.0/16`.

16. Human-readable host names

(a) The Domain Name Service (DNS) maps IP addresses into a hierarchy of names.
(b) First-level domain names: `.net`, `.edu`, `.gov`, `.com`
(c) Second level domain names: `uky`, `mit`, `amazon`
(d) Third level: `cs`, `www`.
(e) No limit on levels, and no particular conventions.
(f) The mapping is maintained in a huge distributed database, which can be understood as a collection of host-entry structures.

```
struct hostent {
   char   *h_name;
/* official domain name of host */
   char   **h_aliases;
/* null-terminated array of domain names */
   int    h_addrtype;
/* host address type (AF_INET) */
   int    h_length;
/* length of an address, in bytes */
   char   **h_addr_list; /* null-terminated array of in_addr s
}
```

(g) The `gethostbyname()` library call queries the DNS database.
(h) Queries in Unix are first checked in `/etc/hosts`. If that fails, they are sent to a *name server*, as listed in `/etc/resolv.conf`.
(i) Command-line programs: `nslookup`, `dig` (Domain Information Groper)

59 Client-server

1. Clients
   (a) Examples: `ftp`, `telnet`, `ssh`, browsers like `Firefox`.
   (b) $\boxed{\text{Lecture 37, 5/1/2015}}$
   (c) To find a server: use the DNS service to convert a DNS name to an IP address.
   (d) To find the right port: look int `/etc/services` to find the standard port for various services, such as 25 (smtp), 80 (http).
2. Servers
   (a) Typically long-running, started when the operating system boots.
   (b) Can be stopped/started, by distribution-dependent methods (such as `service apache2 stop`.
   (c) Listens for requests on the well-known port and communicates with a standard protocol (typically on top of TCP).
   (d) Standard services. Each uses a
      i. Web (80/HTTP): provides files, can perform server-side computation (SHTML, CGI, PHP, JSP, ...).
      ii. FTP (20, 21/FTP): stores and retrieves files.
      iii. Telnet (23/Telnet): interactive login. Deprecated, because unencrypted.
      iv. SSH (22/SSH): interactive login; encrypted.
      v. Mail (25/SMTP): accepts incoming mail.
3. Web services
   (a) The WWW mostly started around 1993, with the Mosaic browser; there were about 200 WWW servers worldwide.
   (b) Web servers return **content** to clients, which is a sequence of bytes tagged with a MIME (Multipurpose Internet Mail Extension) type, such as `text/html`, `application/pdf`, `image/png`.
   (c) Content can be *static* (contents of a file) or *dynamic* (computed on demand).
   (d) Each content item is associated with a URL (Universal Resource Locator), of a general form `service://host:port/file/path`, specifically `http://www.cs.uky.edu/~raphael/courses/CS485/index.html`

(e) The client uses the `host:port` part to determine the address and of the server and the port to connect to. It uses the `service` part to determine what protocol to use.

(f) The server uses the `file/path` part to determine what file to send, or what program to start. It sends the output of the program to the client.

(g) One can use *telnet* to manually engage in a protocol, if it is conducted in ASCII (most are).

```
unix> telnet www.cs.uky.edu 80
Trying 128.163.146.21...
Connected to bud.cs.uky.edu.
Escape character is '^]'.
GET / raphael/ HTTP/1.1
GET / raphael/finkel2.1.jpg host:  www.cs.edu

HTTP/1.1 200 OK
Date: Fri, 01 May 2015 01:55:58 GMT
Server: Apache/2.2.22 (Ubuntu)
...
<html>
...
Connection closed by foreign host.
```

(h) Various methods to get content (incomplete list)

   i. `GET`: the parameters for dynamic content are part of the URL.
   ii. `POST`: The parameters for dynamic content are in the request body.
   iii. `OPTIONS`: Get server of file attributes

(i) The text describes the code for a tiny web server, which only uses 226 lines of C code.

(j) A **proxy** is an intermediary process between a client and server, acting to each as a partner. The proxy can do caching, logging, anonymization, filtering.

4. SMTP protocol is also ASCII, used for sending mail.

```
unix> telnet mail.cs.uky.edu 25
Trying 128.163.146.23...
Connected to satchmo.cs.uky.edu.
Escape character is '^]'.
220 satchmo.cs.uky.edu ESMTP Postfix (Computer Science)
HELO satchmo.cs.uky.edu
250 satchmo.cs.uky.edu
MAIL FROM:<raphael@cs.uky.edu>
250 2.1.0 Ok
RCPT TO:<raphael@cs.uky.edu>
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
From:  "Foo bar" <foo@bar.org>
to:  "Raphael Finkel" <raphael@cs.uky.edu>
Date:  now
Subject:  test message

Howdy doody!
.
250 2.0.0 Ok: queued as 62EFA280846
QUIT
221 2.0.0 Bye
Connection closed by foreign host.
```