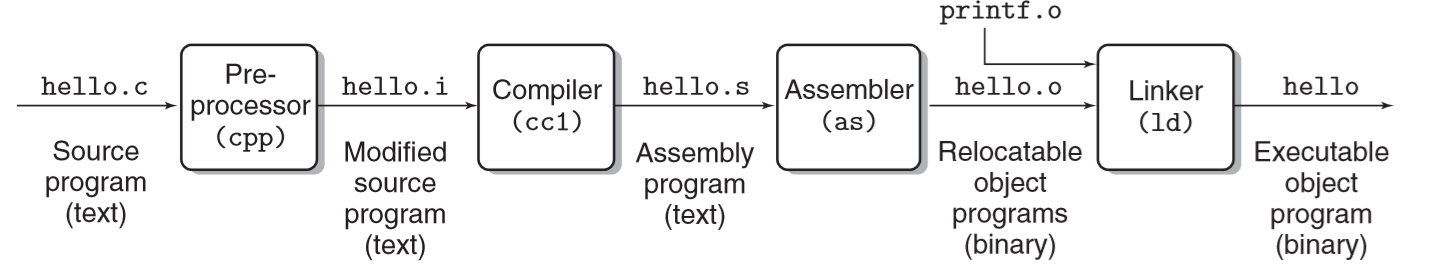
***CS 33 Notes***

**Steps to Compile a Program**

* All data types are stored as bits
* For C programs, created as source program (text) and must be converted into machine-language as executable object program (binary):

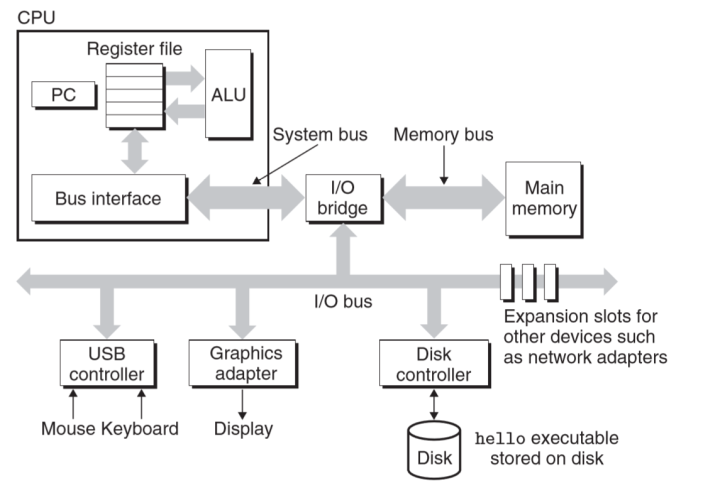
Steps:

1. Preprocessing
   * Modifies program by inserting header file info into program
2. Compilation
   * file.i 🡪 file.s (assembly-language program)
3. Assembly
   * File.s 🡪 file.o (machine-language instructions in binary file w/ machine language intruc.)
4. Linking
   * Merger betw. program files for reserve words and C program



* + Why is this important?
    - Optimize performance, understand linker errors, avoid security holes

**Hardware Organization of a System:**



1. CPU (central processing unit)
   * Interprets instructions in memory and performs simple operations
   * Contains register which holds address of a machine-language instruction, ALU (arithmetic/logic unit) which computes data

**Information Storage:**

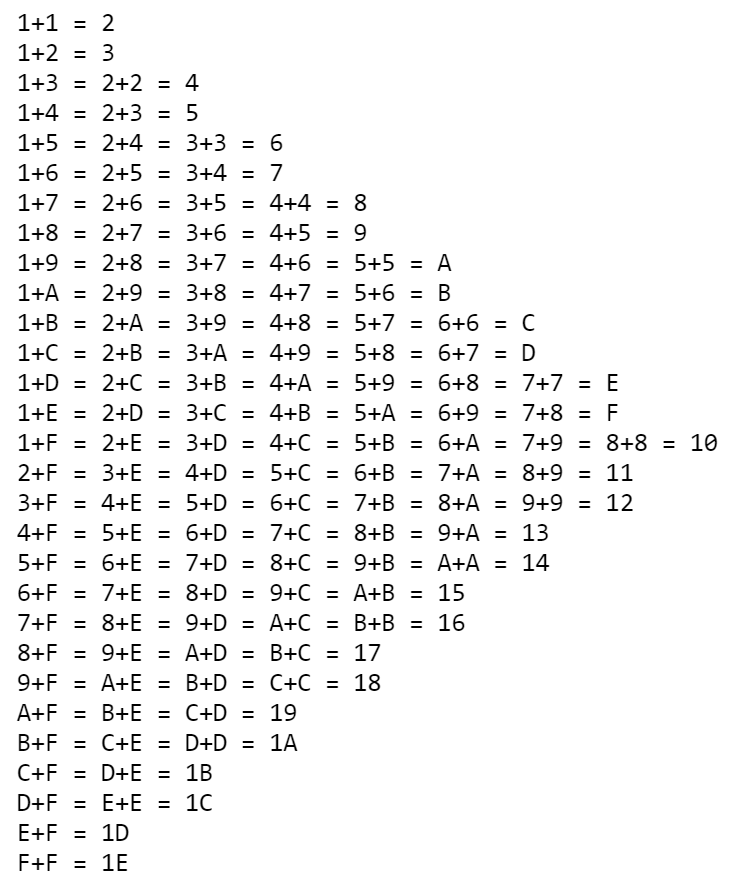
* + Computers store/process info as binary digits (aka bits)

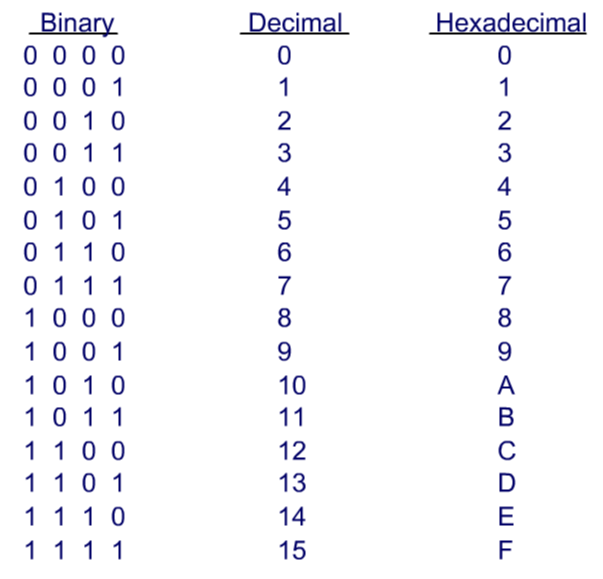
3 Methods to Represent Numbers:

1. Unsigned: >= 0
2. Two’s Complement: signed numbers
3. Floating-point: real numbers
   * Overflow: when number too large to be represented by bits, importance b/c could be source of security breach
   * Computers store memory in blocks of 8 bits = 1 byte
   * Virtual memory: memory as a large array of bytes
   * Every byte has a unique address

Hexadecimal Notation:

* + - Bit patterns w/ base 16 using chars 0-9 and A-F (upper and lower)
    - Constants beginning w/ 0x or 0X are hexadecimal





**How to Convert Between Binary and Hexadecimal:**

Binary 🡪 hex: use above chart

Hex 🡪 binary: split hex number into 4-bit groups and convert

+ if not multiple of 4, make left-most group fewer than 4 bits and pad it with leading 0’s

+ If value is power of 2 (2n), where n = i + 4j (i betw. 0 and 3 where 0 means hex digit 1, 1 is hex digit 2, 2 is hex digit 4, 3 is hex digit 8), hex number then followed by j 0’s

+ ex: 2048 = 211, 11 = 3+4x2 🡺 0x800

**How to Convert Between Decimal and Hexadecimal:**

Decimal 🡪 hex digit:

1. Divide decimal by 16 to get quotient and remainder
2. Divide quotient by 16 again till cannot divide evenly anymore

20 = 1

21 = 2

22 = 4

23 = 8

24 = 16

25 = 32

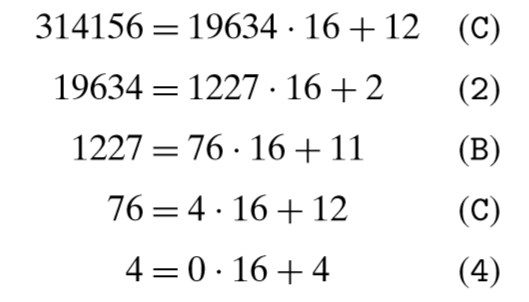
26 = 64

27 = 128

28 = 256

29 = 512

1. For each remainder, convert decimal number to hex digit w/ chart above

Ex: 314156 🡪 0x4CB2C

160 = 1

161 = 16

162 = 256

163 = 4096

164 = 65536

165 = 1048576

166 = 16777216

Hex digit 🡪 decimal:

1. Multiply each hex digit’s decimal equivalent by 16n (n goes from hex digit length to 0)

Ex: given the number 0x7AF, we compute its decimal equivalent as

7 · 162 + 10 · 16 + 15 = 7 · 256 + 10 · 16 + 15 = 1792 + 160 + 15 = 1967

**How to convert between binary and decimal:**

Binary 🡪 Decimal:

1. Start from least significant bit and multiply the 0 or 1 by 2index

Ex: 0110101

= 1x25+1x24+1x22+1x20 = decimal equivalent

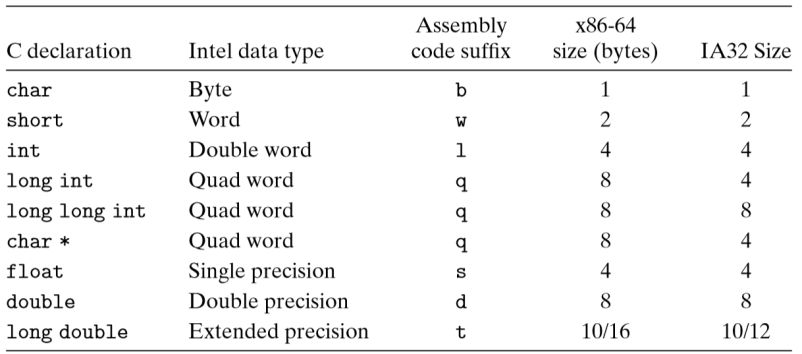
Decimal 🡪 Binary:

* + - 1. divide number by highest 2x repeatedly until reach 0 and for each xth number, a 1 goes in that index location

**Data Sizes:**

Word size: size of int and pointer data, det. maximum address holding space so limits memory

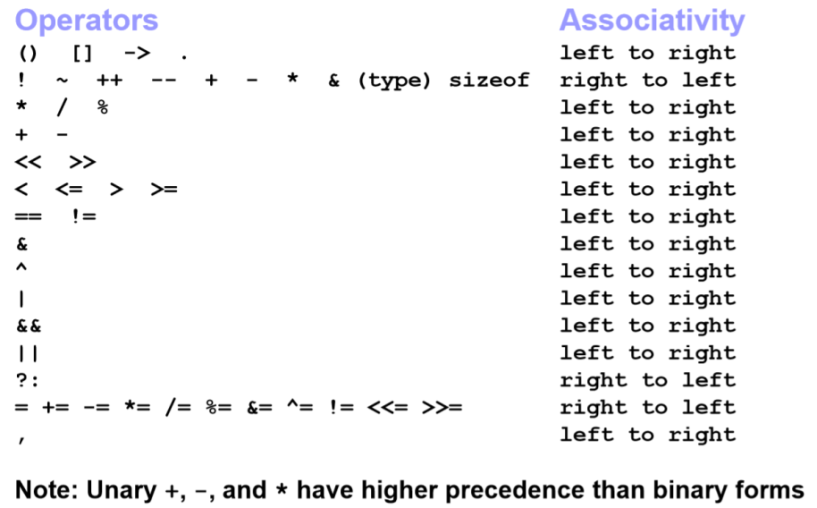
Most computers are 32-bit word size (address space is 4 GB = 4 x 109 bytes)



size\_t = unsigned int at least 16-bit

**Addressing and Byte Ordering:**

* + For multi-byte data types
  + Little endian: reverse indexing where least significant (first part of data) is in furthest address in memory, used in most Intel computers

**Boolean Algebra:**

a ^ b: symmetric difference (exclusive or)

a | b: set union (or)

a & b: set intersection (and)

~a : set complement (not)

-a = ~a + 1

Simplification of Bitwise Operators:

a-b = a&~b

a==b = ~(a^b)

a != b = a^b

**Shift Operations:**

x << n: shift n bits left, fill right end with 0’s, same as multiplying by power of two

x >> n: shift n bits right

1. Logical – fill left end with 0’s (for unsigned data)
2. Arithmetic – fills left end with most significant bit (for signed data)

**Integer Representations:**

Integral Data Types: char, short, long, long long (signed/unsigned)

***Unsigned:***

Range: 0 to 2n – 1

U-min = [00..00] to U-max = [11..11]

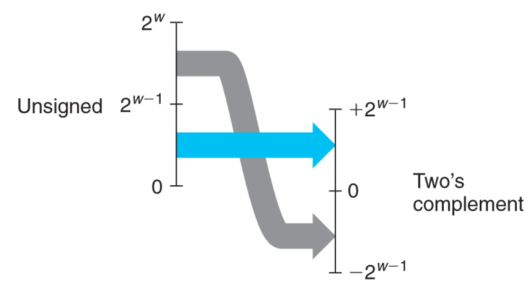
***Signed: Two’s Complement***

Most significant bit holds signed bit

Range: -2n-1 to 2n-1 – 1

T-min = [10..00], -1 = [11..11], 0 = [00..00], T-max = [01..11]

To convert between signed and unsigned: complement each binary digit and add l







***Casting from Signed to Unsigned:***

Extending smaller bit-size to larger:

Original value represented is kept

Unsigned:

Zero extension – add leading 0’s

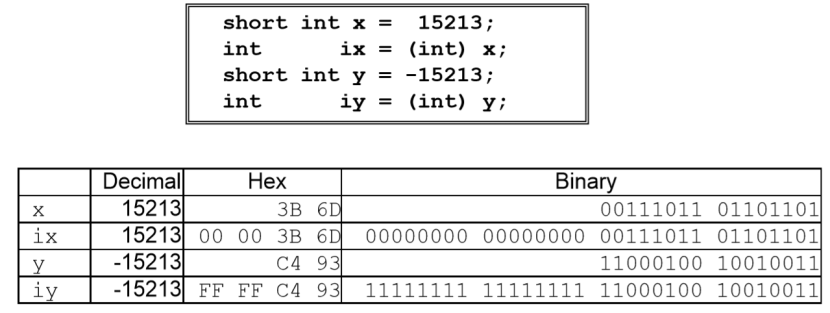
Signed:

Sign extension – add most significant bit to leading side

Truncating larger bit-size to smaller:

Unsigned/Signed:

Discard top bits

Ex:

**Integer Arithmetic:**

Unsigned:

* + Possibility of overflow if two number’s binary representations are too large to fit in same number of bit spots
  + If overflow, truncate first bit which may alter answer if number did overflow
  + how to tell if overflowed: after adding two numbers, their sum is less than either of the two operands

Signed:

* + If overflow, truncate
* If adding signed and unsigned, both operands become unsigned
* a negative number will become a large positive b/c add UINT\_MAX + 1

Arithmetic Operations:

* + Rules of Adding:

0+0 = 0, with no carry  
1+0 = 1, with no carry  
0+1 = 1, with no carry  
1+1 = 0, with carry of 1

* + Overflow happens when doing operation on two addends and getting result that has more bits than the addends

**Usual Arithmetic Conversions:**

1. If either operand is of type **long double**, the other operand is converted to type **long double**.
2. If the above condition is not met and either operand is of type **double**, the other operand is converted to type **double**.
3. If the above two conditions are not met and either operand is of type **float**, the other operand is converted to type **float**.
4. If the above three conditions are not met (none of the operands are of floating types), then integral conversions are performed on the operands as follows:
   * If either operand is of type **unsigned long**, the other operand is converted to type **unsigned long**.
   * If the above condition is not met and either operand is of type **long** and the other of type **unsigned int**, both operands are converted to type **unsigned long**.
   * If the above two conditions are not met, and either operand is of type **long**, the other operand is converted to type **long**.
   * If the above three conditions are not met, and either operand is of type **unsigned int**, the other operand is converted to type **unsigned int**.
   * If none of the above conditions are met, both operands are converted to type **int**.

Reduced (RISC) vs. Complex (CISC) Instruction Sets:

RISC – supports simpler computer hardware w/ fewer instructions

CISC – supports larger/faster hardware w/ many different instructions//slower clock speed

**Assembly Language:**

assembly language – textual representation for indiv. instruct. machine gives

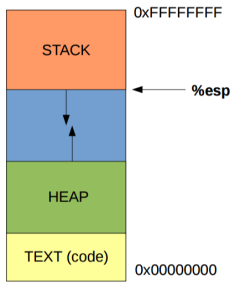
ISA – Instruction Set Architecture, defines format/behavior of machine-level programs

To see assembly language: unix> gcc -O1 -S code.c

To see object code file: unix> gcc -O1 -c code.c (hexadecimal representation)

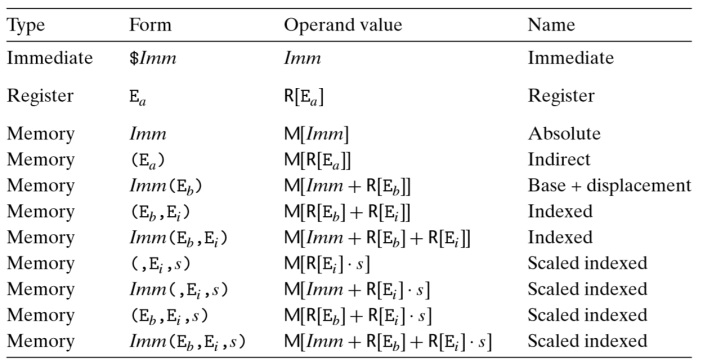
IA32 (instruction set for 32 bit computers)

* + - Instructions range from 1-15 bytes where common/simpler instructions use less bytes

**Accessing Memory:**

* + IA32 CPU has 8 registers storing 32-bit values
  + First 6 are general purpose and last two contain pointers to parts of program stack
    - %esp – stack pointer to top of stack (lowest address in memory)
      1. x86-64 only this, no base ptr
    - %ebp – frame pointer hold address of current stack frame
    - %eip – instruction pointer hold address of next instruction to be called





Operands:

Types:

1. Immediate: constant integer values

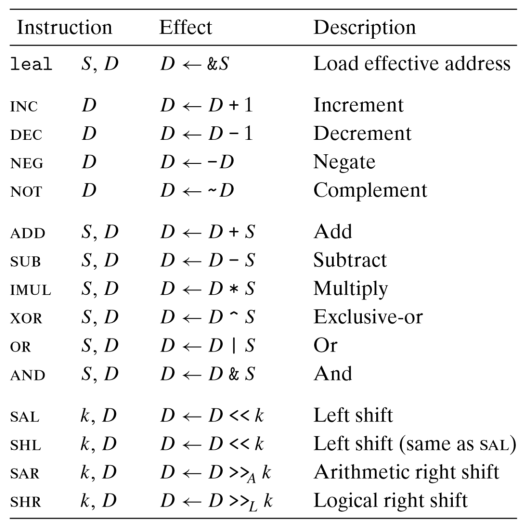
Format: $number

1. Register

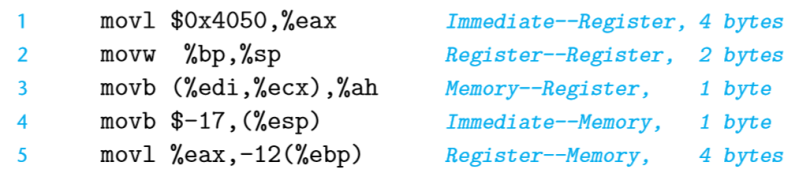
Format: %register name

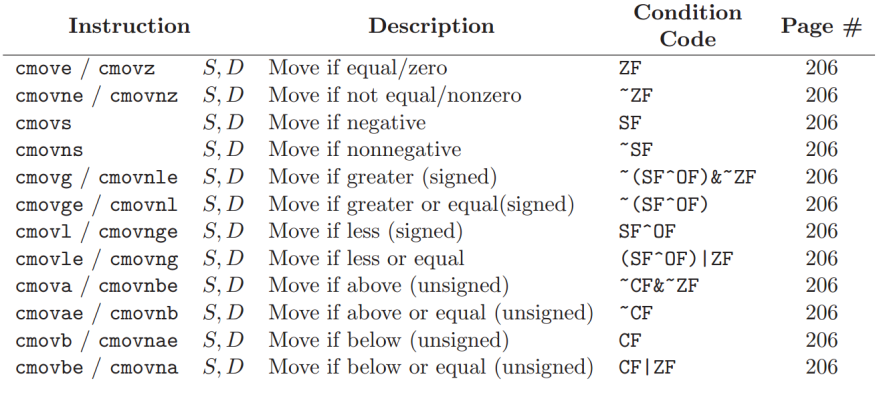
1. Memory: accessed through effective address
   * + scaling factor has to be 1, 2, 4, 8

**x86 Instruction Set:**

***mov:***

1. movb
2. movw
3. movl
4. movslq (mov signed long into quad)
   * copies source values (either immediate, stored in register or stored in memory) into destination (either stored in memory or in register)
   * to mov from memory to memory takes 2 instructions (load source value into register and write register value to destination)





cmov: conditional move

***push:***

* + decrements (subl) %esp by however many bytes of data you are pushing on, move data onto memory stack top

***pop:***

* + move/store data from top of stack, increment (addl) %esp

***leal:***

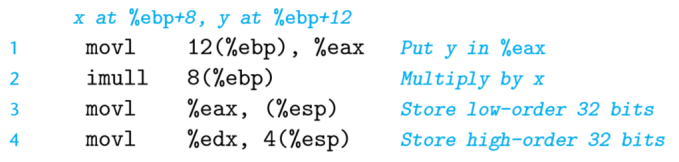
* + load effective address
  + reads from memory to register, but doesn’t access memory at all and just copies address to register
  + no other size variants
  + can do arithmetic:
    1. ie: leal 0x10(%ebx), %eax means eax = ebx + 10
    2. does not dereference memory

***shifts (sal/sar/shr/shl):***

* + left operand (k) can only be immediate or from register %cl
  + right operand can be register or memory
  + sal (arithmetic, signed) and shl (logical, unsigned) are same
  + by default has operand of 1 (shift by 1 bit)

***imul:***

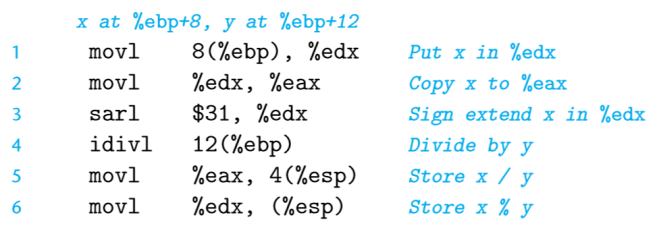
* + if passed 32-bit argument, answer will overflow to span %eax and %edx where lower 32-bits in %eax and higher 32 in %edx

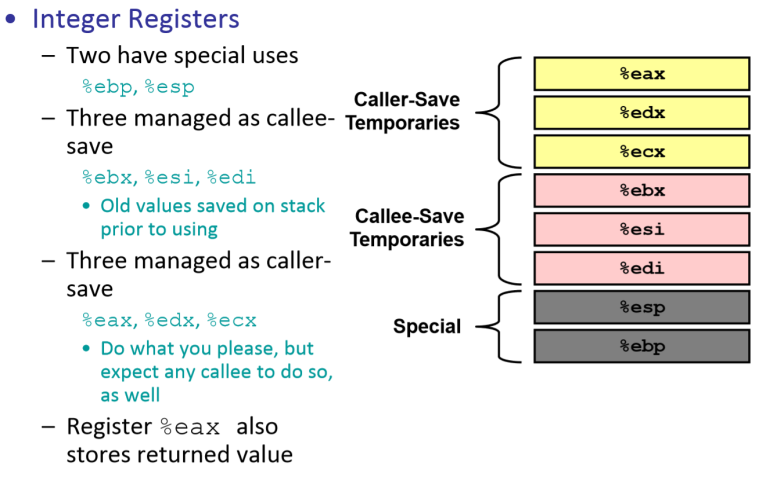
ie:

***idiv:***

* + quotient in %eax, remainder in %edx

ie: sign extend x so %edx and %eax hold 64-bit sign extended version of x for division by 32-bit number



**Calling Conventions:**

Function Argument Storage (x86-64):

%rdi = 1st Argument

%rsi = 2nd Argument

%rdx = 3rd Argument

%rcx = 4th Argument

%r8 = 5th Argument

%r9 = 6th Argument

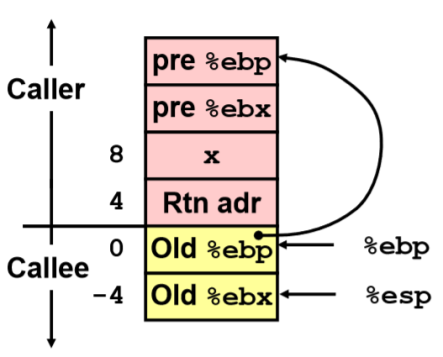
7th Argument+: on stack

return value = %rax

callee saved = %ebx, %esi, %edi, %ebp, %r10, %r12…%r15

* + new frame procedure must save values in registers before overwriting memory and must restore these values before returning

caller saved = %eax, %edx, %ecx

* + when procedure is called, new frame can overwrite the registers w/o destroying data in previous frame
  + old ebp stored at \*ebp
  + to allocate space, decrement %esp
  + to deallocate, increment %esp

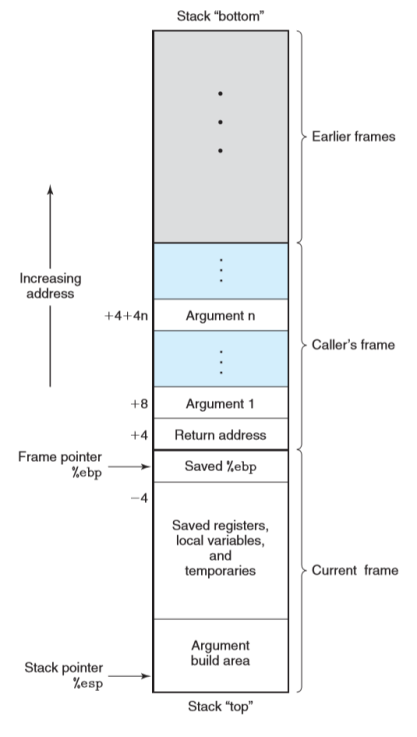
**Parameters:**

* + passed via stack
  + retrieved based on offset from %ebp

**%ebp + 4 = return address of previous frame**

**%ebp + 8 = 1st parameter**

**%ebp + 12 = 2nd parameter**

**Upon call to another frame:**

* + - use “call Label”
      1. pushes return address onto stack (next instruction in current frame) and jumps to call procedure (moves %eip)
      2. direct or indirect like jumps
    - save content of caller-saved registers (%eax, %edx, %ecx)
    - push parameters in inverted order onto stack before calling
    - when use call instruction, return address placed at top of stack
    - new stack frame pushed onto stack

**Upon return from another frame:**

* + - “leave” (prep stack for return by deallocating stack)
    - “ret” (pops return address off stack top and moves %eip to that address)
* **return value: stored in %eax**
* **return address: stored at \*(ebp + 4)**
* ret sometimes preceded by leave instruction (leave ret)
* deallocates stack frame
  + - set %eip to %eip of caller’s next instruction
    - return value (int or pointer) in %eax
    - remove parameters from stack to restore stack
    - restore caller-saved registers

**Control Flow:**

* + usually instructions in sequential order
  + but with loops, if-else, etc., may need jumps (instruction further along program often dependent on to result of a test for condition)

***cmp b,a: a – b***

* + does not store result into a register unlike sub

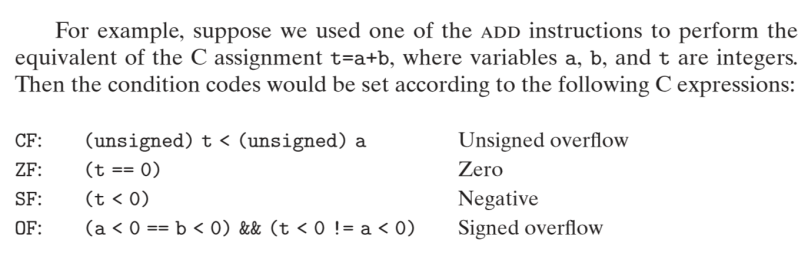
***test b,a: a & b***

* + usually tests for 0

**Flags:**

* + - single bit condition code registers describe results of previous arithmetic or logical operation (1-bit)

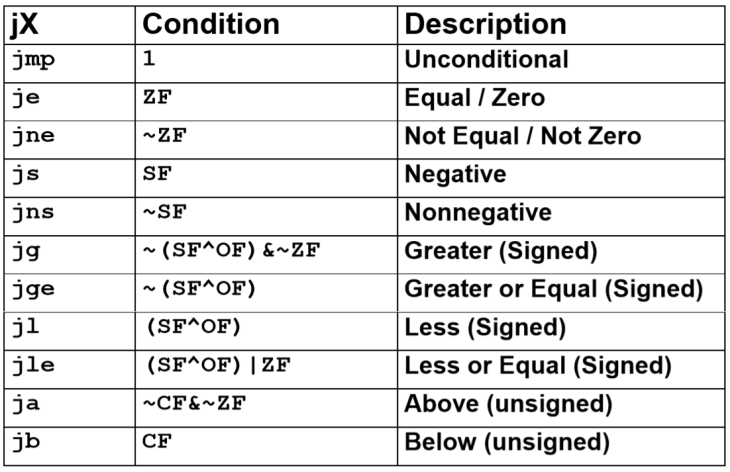
1. CF (Carry Flag): recent operation results in carry (used for overflow)
2. ZF (Zero Flag): recent operation result in zero
3. SF (Sign Flag): recent operation result in negative
4. OF (Overflow Flag): recent operation result in two’s complement overflow (signed overflow)

ie:

**Reading Condition Flags:**

* + - setx intructions:



**JUMPS:**

* + interrupt sequence of instructions and move control unconditionally or after testing result of previous cmp/test
  + moves %eip
  + jump [destination label]
  + Types:
    1. Direct = destination is immediate jump target label (ie jmp .L1 or jmp $0x7FF4078)
       1. label (instruction address) translated to address by linker later
    2. Indirect = destination is actual register or memory
       1. ie jmp \*%eax jump to address in register
       2. jmp \*(%eax) value in memory referred by %eax)

**If-Else:**

movl x, %eax

cmpl y, %eax

jg .L12

.L12:

movl $2, a

ret

movl $3, b

ret

void foo() {

if (x < y)

a = 2;

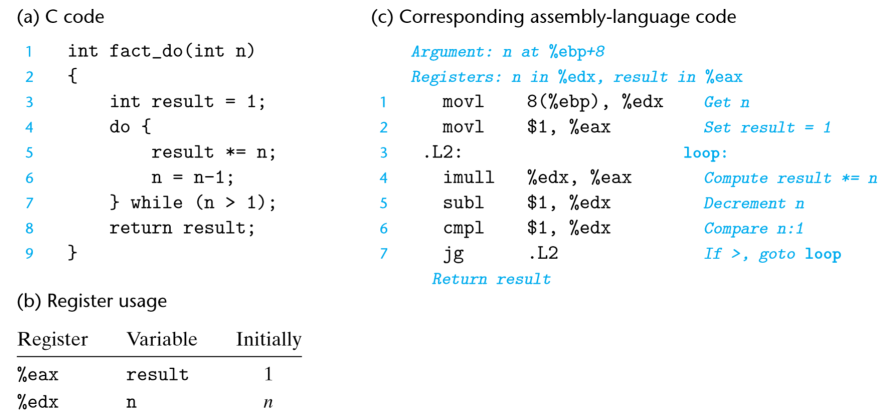
else

b = 3;

}

**Loops:**

**Do-While Loops:**

****C Format:

do

body-statement

while (test-expr)

Assembly Format

loop:

body-statement

t=test-expr

if (t)

goto loop

**While Loop:**

C Format:

while (test-expr)

body-statement

Assembly Format:

t=test-expr;

if (!t)

goto done

loop:

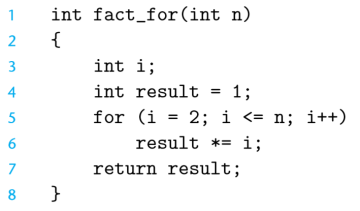
body-statement

t=test-expr;

if (t)

goto loop

done:

**For-Loops:**

look like do-while loops

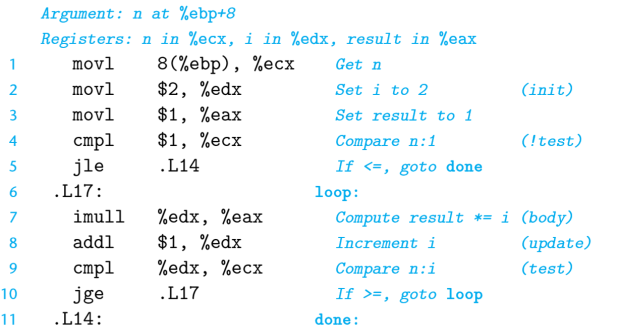
C Format:

for (init-expr; test-expr; update-expr)

body-statement

Assembly Format:

init-expr;

t=test-expr;

if (!t)

goto done;

loop:

body-statement

update-expr;

t=test-expr;

if (t)

goto loop;

done:

**Switch Statements:**

* + implemented with jump table (array of jump targets containing instruction addresses)
  + uses indirect jump to find instruction address in index of jump table

**Recursion:**



recursive factorial example:

* + tail recursion – optimization to avoid pushing onto a stack and allocating memory, just replace stack frame with whatever parameters are different, uses jmp instead of a call to function

**Arrays:**

* + store static arrays on stack as contiguous bytes
  + 1-D arrays

int x = arr1[i]

&x[i] = &arr1 + i\*sizeof(int)

* + multi-dimensional datatype arr2[n][m]
    1. how many bytes? n \* m \* sizeof(datatype)
    2. how to access? &arr[row][col] = &arr\_start + sizeof(datatype) \* (# columns(m) \* row + col)

**Struct:**

* + data in struct take up memory contiguously where member variables differ by offset in bytes from start of struct in memory
  + Data Alignment:
    1. restriction on address of certain types, require address is be multiple of 2, 4, or 8 to make memory access faster
    2. usually goes by data type’s size
       1. 1-byte char: any alignment
       2. 2-byte short: aligned at even address/where divisible by 2
       3. 4-byte int/float: aligned where divisible by 4
       4. 8-byte long/double: aligned where divisible by 8
       5. 12-byte long double
    3. for struct instances alignment must conform with alignment of largest type (for use in arrays of structs)

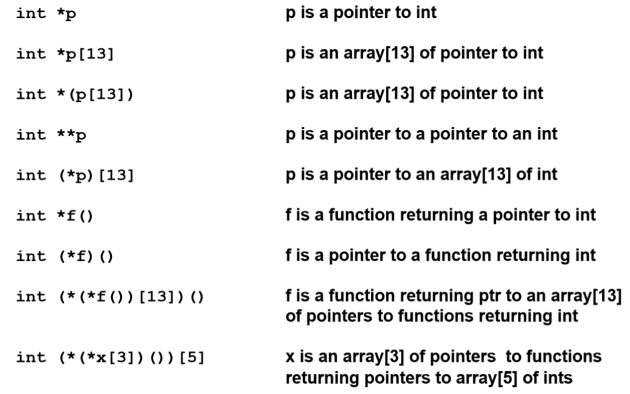
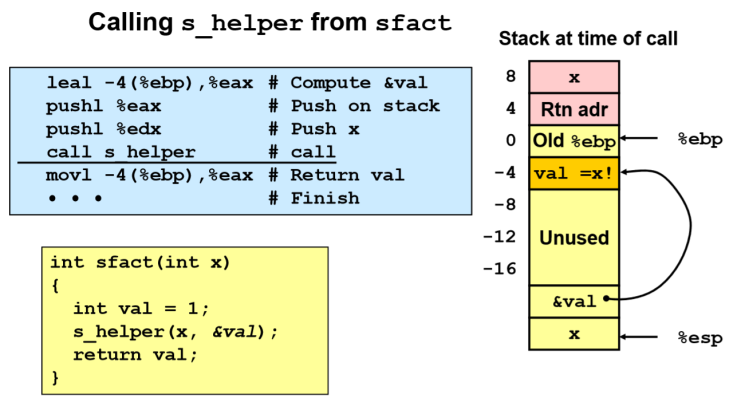
**Unions:**

* + similar to structs except multiple data members share same memory space
  + can only access one data member at a time, allowing for single object to be referenced by multiple types

**Pointers:**

* + simply 32-bit number in x86 and 64-bit number in x86-64
  + malloc – “new” memory allocation
  + uses lea instruction to compute address and then is pushed onto stack

Pointer Declarations:



**GDB Debugger:**

* + let us see what’s going on in a program or what happened upon program crash
  + offsetof(a,b)
    1. ((size\_t)(&(a\*)(0))->b) – place structure at memory address 0 and find address of wanted internal field
       1. defined in <stddef.h>

(size\_t)( // 4. cast address to size\_t to get offset

&( ( // 3. get address of this field

(a\*)(0) // 1. cast a to 0

)->b ) // 2. get field of b (illegal)

)

* + alignof(t)
    1. #**define** **ALIGNOF**(type) offsetof (struct { char c; type member; }, member)

**x86 Hardware Breakpoint Registers:**

breakpoint v. watchpoint:

breakpoint: stops at particular line of source code

watchpoint: stops when contents of a field/struct change

* + - 1. DR0

address registers (tell where want breakpoints/watchpoints)

* + - 1. DR1
      2. DR2
      3. DR3
      4. DR6 – status if program reached breakpoint yet
      5. DR7 – control
         1. tells hardware how to interpret DR0-3
         2. has bits for:

execute: stops program w/o changing instruction

write: put watchpoint

data read/write: stop program whenever access data

GDB commands by function - simple guide

---------------------------------------

More important commands have a (\*) by them.

Startup

% gdb -help print startup help, show switches

\*% gdb object normal debug

\*% gdb object core core debug (must specify core file)

%% gdb object pid attach to running process

% gdb use file command to load object

Help

\*(gdb) help list command classes

(gdb) help running list commands in one command class

(gdb) help run bottom-level help for a command "run"

(gdb) help info list info commands (running program state)

(gdb) help info line help for a particular info command

(gdb) help show list show commands (gdb state)

(gdb) help show commands specific help for a show command

Breakpoints

\*(gdb) break main set a breakpoint on a function

\*(gdb) break 101 set a breakpoint on a line number

\*(gdb) break basic.c:101 set breakpoint at file and line (or function)

\*(gdb) info breakpoints show breakpoints

\*(gdb) delete 1 delete a breakpoint by number

(gdb) delete delete all breakpoints (prompted)

(gdb) clear delete breakpoints at current line

(gdb) clear function delete breakpoints at function

(gdb) clear line delete breakpoints at line

(gdb) disable 2 turn a breakpoint off, but don't remove it

(gdb) enable 2 turn disabled breakpoint back on

(gdb) tbreak function|line set a temporary breakpoint

(gdb) commands break-no ... end set gdb commands with breakpoint

(gdb) ignore break-no count ignore bpt N-1 times before activation

(gdb) condition break-no expression break only if condition is true

(gdb) condition 2 i == 20 example: break on breakpoint 2 if i equals 20

(gdb) watch expression set software watchpoint on variable

(gdb) info watchpoints show current watchpoints

Running the program

\*(gdb) run run the program with current arguments

\*(gdb) run args redirection run with args and redirection

(gdb) set args args... set arguments for run

(gdb) show args show current arguments to run

\*(gdb) cont continue the program

\*(gdb) step single step the program; step into functions

(gdb) step count singlestep \fIcount\fR times

\*(gdb) next step but step over functions

(gdb) next count next \fIcount\fR times

\*(gdb) CTRL-C actually SIGINT, stop execution of current program

\*(gdb) attach process-id attach to running program

\*(gdb) detach detach from running program

\*(gdb) finish finish current function's execution

(gdb) kill kill current executing program

Stack backtrace

\*(gdb) bt print stack backtrace

(gdb) frame show current execution position

(gdb) up move up stack trace (towards main)

(gdb) down move down stack trace (away from main)

\*(gdb) info locals print automatic variables in frame

(gdb) info args print function parameters

Browsing source

\*(gdb) list 101 list 10 lines around line 101

\*(gdb) list 1,10 list lines 1 to 10

\*(gdb) list main list lines around function

\*(gdb) list basic.c:main list from another file basic.c

\*(gdb) list - list previous 10 lines

(gdb) list \*0x22e4 list source at address

(gdb) cd dir change current directory to \fIdir\fR

(gdb) pwd print working directory

(gdb) search regexpr forward current for regular expression

(gdb) reverse-search regexpr backward search for regular expression

(gdb) dir dirname add directory to source path

(gdb) dir reset source path to nothing

(gdb) show directories show source path

Browsing Data

\*(gdb) print expression print expression, added to value history

\*(gdb) print/x expressionR print in hex

(gdb) print array[i]@count artificial array - print array range

(gdb) print $ print last value

(gdb) print \*$->next print thru list

(gdb) print $1 print value 1 from value history

(gdb) print ::gx force scope to be global

(gdb) print 'basic.c'::gx global scope in named file (>=4.6)

(gdb) print/x &main print address of function

(gdb) x/countFormatSize address low-level examine command

(gdb) x/x &gx print gx in hex

(gdb) x/4wx &main print 4 longs at start of \fImain\fR in hex

(gdb) x/gf &gd1 print double

(gdb) help x show formats for x

\*(gdb) info locals print local automatics only

(gdb) info functions regexp print function names

(gdb) info variables regexp print global variable names

\*(gdb) ptype name print type definition

(gdb) whatis expression print type of expression

\*(gdb) set variable = expression assign value

(gdb) display expression display expression result at stop

(gdb) undisplay delete displays

(gdb) info display show displays

(gdb) show values print value history (>= gdb 4.0)

(gdb) info history print value history (gdb 3.5)

Object File manipulation

(gdb) file object load new file for debug (sym+exec)

(gdb) file discard sym+exec file info

(gdb) symbol-file object load only symbol table

(gdb) exec-file object specify object to run (not sym-file)

(gdb) core-file core post-mortem debugging

Signal Control

(gdb) info signals print signal setup

(gdb) handle signo actions set debugger actions for signal

(gdb) handle INT print print message when signal occurs

(gdb) handle INT noprint don't print message

(gdb) handle INT stop stop program when signal occurs

(gdb) handle INT nostop don't stop program

(gdb) handle INT pass allow program to receive signal

(gdb) handle INT nopass debugger catches signal; program doesn't

(gdb) signal signo continue and send signal to program

(gdb) signal 0 continue and send no signal to program

Machine-level Debug

(gdb) info reg (registers) print registers sans floats

(gdb) info all-registers print all registers

(gdb) print/x $pc print one register

(gdb) stepi single step at machine level

(gdb) si single step at machine level

(gdb) nexti single step (over functions) at machine level

(gdb) ni single step (over functions) at machine level

(gdb) display/i $pc print current instruction in display

(gdb) x/x &gx print variable gx in hex

(gdb) info line 22 print addresses for object code for line 22

(gdb) info line \*0x2c4e print line number of object code at address

(gdb) x/10i main disassemble first 10 instructions in \fImain\fR

(gdb) disas (disassemble addr) dissassemble code for function around addr

History Display

(gdb) show commands print command history (>= gdb 4.0)

(gdb) info editing print command history (gdb 3.5)

(gdb) ESC-CTRL-J switch to vi edit mode from emacs edit mode

(gdb) set history expansion on turn on c-shell like history

(gdb) break class::member set breakpoint on class member. may get menu

(gdb) list class::member list member in class

(gdb) ptype class print class members

(gdb) print \*this print contents of this pointer

(gdb) rbreak regexpr useful for breakpoint on overloaded member name

Miscellaneous

(gdb) define command ... end define user command

\*(gdb) RETURN repeat last command

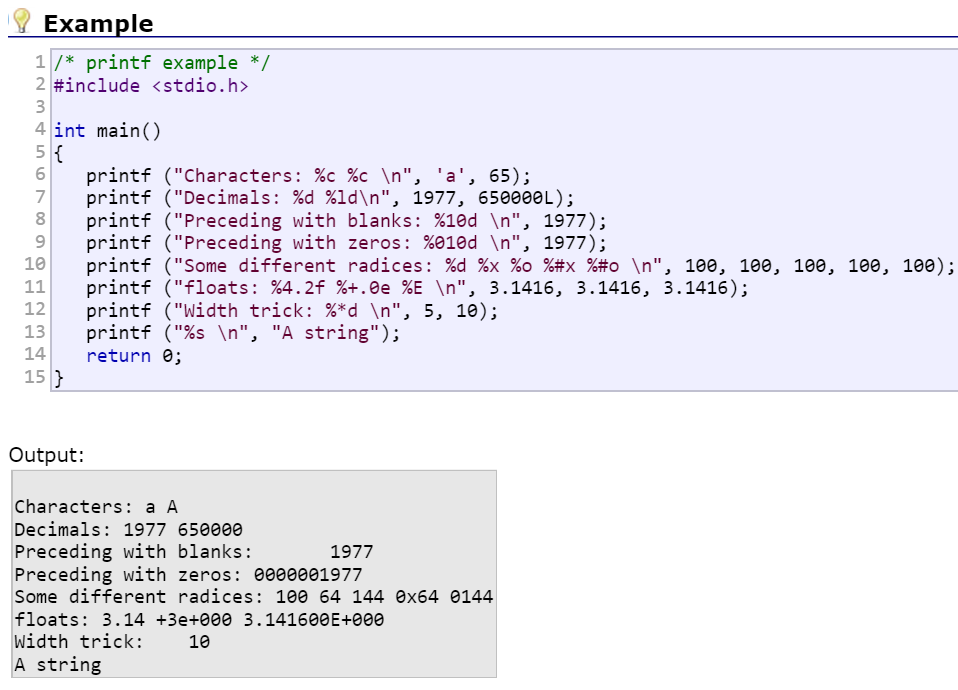
\*(gdb) shell command args execute shell command

\*(gdb) source file load gdb commands from file

\*(gdb) quit quit gdb

INT # betw 0-255:

* + 2 byte instruction crashes program
  + goes to location and replaces instruction there w/ INT
  + problems if instruction replacing is only 1 byte (ie ret)
    1. use INT 3 instead for replacing 1-byte instructions



**Buffer Overflow:**

* + array out of bounds overwrites memory on the stack, corrupting state of stack
  + register won’t be restored properly so what register holds cannot be trusted
  + return address may be overwritten and cause program to jump to unexpected place
  + leads to security issues

How to defend against:

* + don’t use C/C++ b/c too fast
  + gcc subscript-checking (but gcc doesn’t have b/c slow)
  + put variables in safer spot with randomized gap sizes between variables
  + randomized canaries (when destroyed by overflow stops program)

NX-bit:

* + marks each region in memory as being allowed or disallowed for read, write, execution

|  |  |  |  |
| --- | --- | --- | --- |
| R |  | X | Text |
| R | W |  | Heap |
| R | W |  | Stack |

Return Oriented Programming (ROP) Hacking:

* + victim = large programs with bugs
  + hackers gain control of call stack by write own program to cause buffer overflow and overwrite return addresses back to caller’s control
* defense: randomize program

position-independent code (PIC): no addresses in code b/c can run anywhere

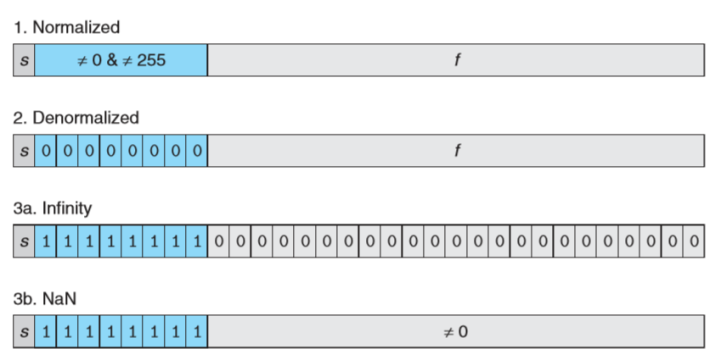
**Floating Point:**

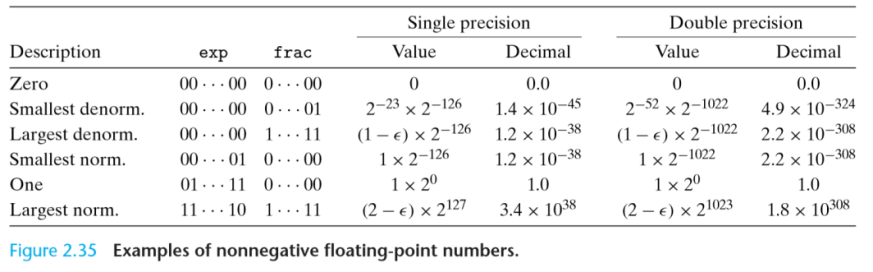
* representing fractional numbers letting bit positions be weighted differently
* addition/multiplication: commutative, not associative
* multiplication does not distribute over addition
* IEEE Standard: x·2y decimal rep // 2E·M binary rep
  + 3 Components:
    - ***sign bit*** (S): 1 bit -- 1 negative or 0 positive
    - ***exponent*** (e): 8 bits(single-precision) 11 bits(double-precision)
      * to represent huge numbers and tiny numbers
      * e rep. unsigned form of number
      * use 2’s Complement to define bias(B)
        + B = 2n-1 – 1 so that half rep positive and other half negative where n = # bits of e

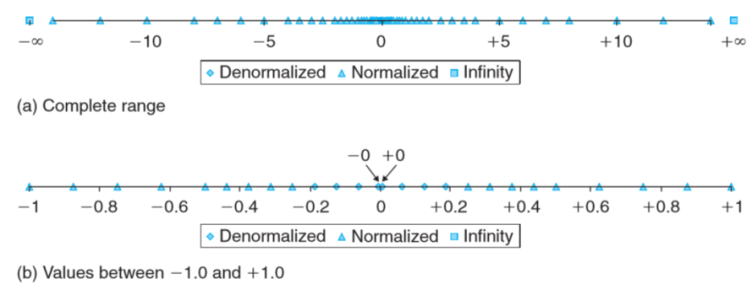
127 (single precision) 1023 (double precision)

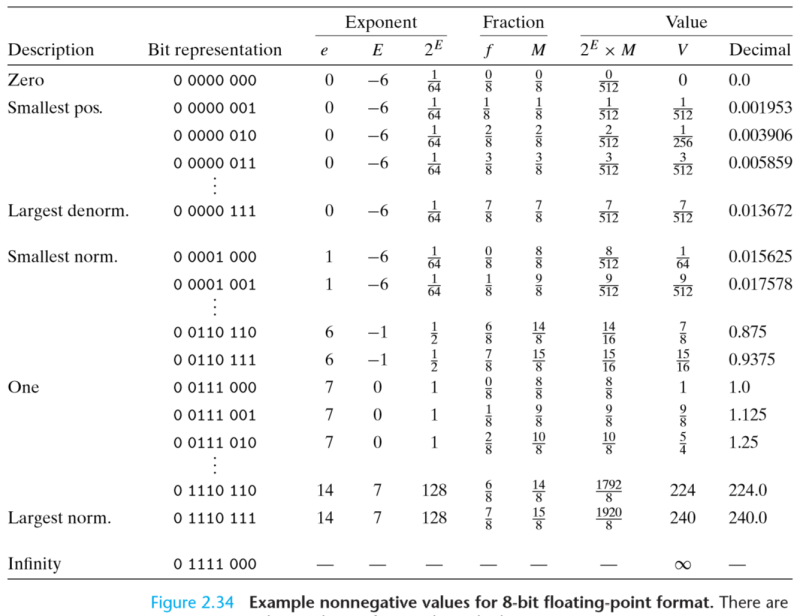
ranges: denorm -126-127 infinite and -1022-1023

* + - ***fraction*** (F): 23 bits(single precision) 52 bits(double precision)
      * contained in mantissa (M); if F = 0011000...0 then M = 1.0011000...0
  + Normalized Format: for numbers NOT near 0
    - represents integer in “biased” form
    - e != all 0’s or all 1’s
    - 0 <= f < 1
    - E = e - B
    - M = 1 + f (implied leading 1 so M >= 1); can adjust E so 1 <= M < 2
    - smallest: M = 1 E = -2k-1+2 = -126 (k-bit exponent) value = 2E
    - biggest: f = 1 – 2-n M = 2-2-n E = 2k-1-1 = 127 value = (1-2-n-1)x22^(k-1)
  + De-normalized Format: for numbers very close to 0
    - e = all 0’s
    - E = 1 – B = -126 (single precision)
    - M = f
    - allows for positive and negative 0 depending on sign bit
    - gradual underflow – numbers very close to 0 are evenly spaced
    - smallest: f = 2-n E = -2k-1+2=-126 value = 2-n+E
    - biggest: f = 1-2-n E = -2k-1+2 value = (1-2-n)x2E
  + Special Cases:
    - e = all 1’s
      * if f = 00..0, rep infinity + or – depending on sign bit
      * if f != 0, NaN (ie sqrt(-1))









Casting values between int, float, and double formats (assuming a 32-bit int):

* int to float
  + the number cannot overﬂow, but it may be rounded
* int/float to double
  + exact numeric value preserved because double has greater range and greater precision
* double to float
  + value overﬂow to positive/negative infinity because smaller range and smaller precision
* float/double to int
  + can overflow
  + value rounded toward zero
  + ie: 1.999 converted to 1, while −1.999 converted to −1

Converting from Decimal to Floating Point:

* + X = (-1)S \* 2E \* M
    - Step 1: Write decimal number in fixed point

21.8125 = 16 + 4 + 1 + 0.5 + 0.25 + 0.0625🡨1/16

1 0 1 0 1 . 1 1 0 1

16 8 4 2 1 -1 -2 -3 -4 🡨 powers

* + - Step 2 : Determine Exponent

1.01011101·24

E = 4

* + - Step 3 : Determine bit-representation of E

E = e - bias

e = E + bias

e = 4 + 127 = 131 = 1000011

* + - Step 4 : Write mantissa

M = 1.01011101 so

F = 01011101 00...00 (padding 0’s)

* + - Step 5 : Combine all 3 components

0(sign) 1 0 0 0 0 0 1 1(e) 0 1 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0(F)

Integers vs. Floating Point Problems:

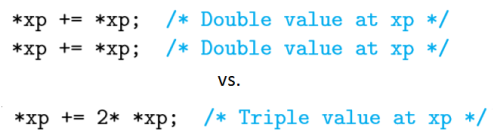
|  |  |
| --- | --- |
| Integers (2’s Complement) | Floating Point (IEEE 754) |
| * + - 1. overflow       2. division by 0 | 1. overflow 2. division by 0 3. inexact 4. underflow (very small numbers) 5. invalid operand (sqrt(-1))   BUT milder overflow problem |

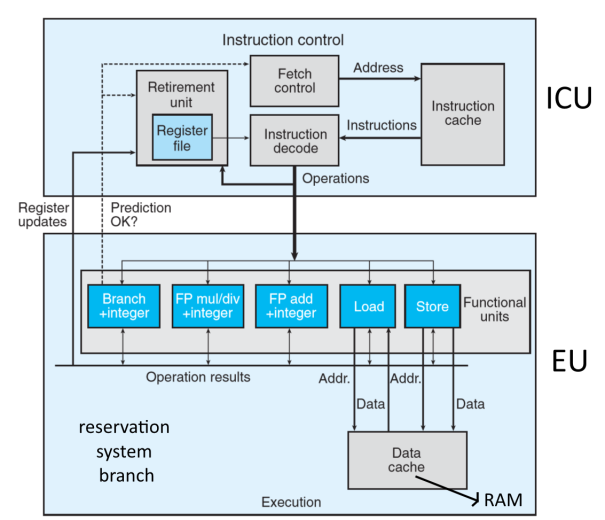
**Optimization:**

Performance Improvement:

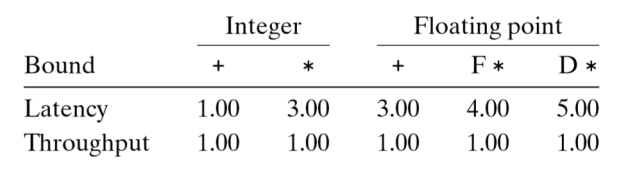
1. don’t do work if there is simple answer
2. use better algorithm to reduce Big-O
3. parallelism
4. sequential constant factor (easiest)
   1. machine-independent
   2. machine-dependent

aliasing:

* problem: 2 pointers point to same address in memory (unplanned) and change value unexpectedly 🡪 optimization blocker (compiler will avoid any chance for this so will not optimize)

Microprocessor:

* + instruction control unit (ICU) – read instructions fr instruction cache (memory contain recent instructions)
    1. issue with conditional branches 🡪 branch prediction + speculative execution
    2. micro-operations made that convert instructions to lower-level instructions
       1. can be multiple for 1 instruction ie. addl %eax, 4(%edx) takes a load, add, and store
       2. allows for parallelism
    3. retirement unit – watches ongoing processes to ensure stays in order of machine-level program w/ FIFO queue
       1. when conditional branches confirms, instruction either:
          1. retired – all changes updated in registers
          2. flushed – all results discarded (when branch prediction wrong)
  + execution unit (EU) – executes instructions from instruction fetch unit out of order (unless is W🡪W or W🡪R)
    1. load/store unit handle RW of memory, access memory through data cache (contain most recent accessed data values)
    2. register renaming
       1. to improve parallelism at hardware level b/c only finite number of registers
       2. to control communication betw. EUs
       3. form of data forwarding to operations instead of RW fr. registers to be more efficient, registers updated after all operations
       4. less collisions, more parallelism

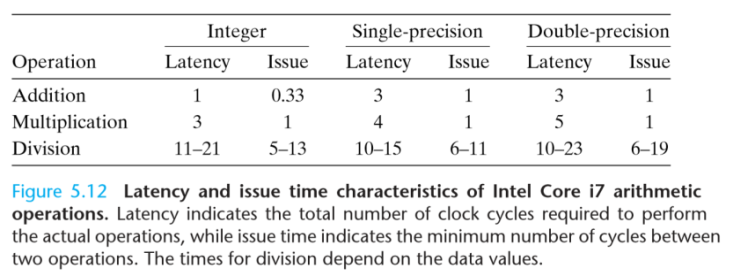
cycles per element(CPE)

elements per second(EPS)

cycles per second/CPE = TFLOPS (trillion floating point operations per second)

throughput – # operations per second

latency – time to do 1 operation from decision to completion

* important for real time efficiency
* inherently sequential
* increase with word size increase

issue time – min. # cycles betw. 2 operations of same type

* addl/mull issue time = 1 b/c of pipelining

Sequential (Loop Inefficiency) Optimization:

* + - 1. “hoisting”/”strength reduction”/code motion – moving as much of loop outside as possible to reduce repeated computation (ie calling vector length etc.)
      2. inlining – optimizing function calls by reducing overhead
      3. loop unrolling – rewrite code to do actual computation than less important instructions
         1. improves performance by:

reduces operations that don’t contribute to program result (overhead)

exposes ways to improve code to reduce ops

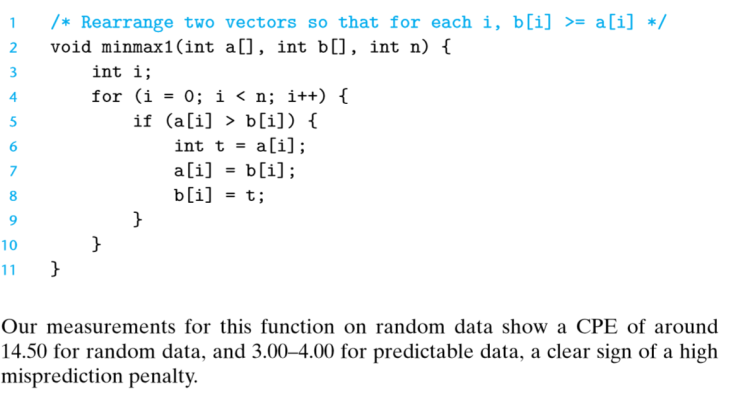
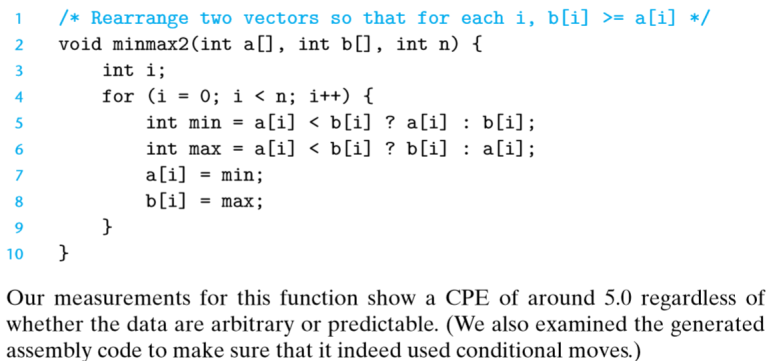
* + - * 1. helps add and mul of non-floating point
        2. problem: have to assume number of loops divisible by some number

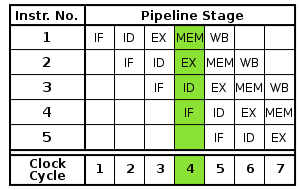
Parallelism:

* + decreases wall time, but NOT energy, or CPU cycles
  + Techniques:
    1. multiple computers (“distributed computing”) – expensive, slower to share data
    2. multiple processors/CPU/cores in 1 computer – communicate via shared memory; but too much going on at once 🡪 racing conditions
    3. multiple instructions executing at same time in 1 CPU (“instruction level parallelism ILP”)
       1. best option b/c cheap + efficient; programs oblivious of it
       2. lower level faster b/c allow multiple op at once
       3. better than compiler b/c know hardware + has dynamic behavior, but worse b/c small
       4. only good up to some point, then performance decrease again
       5. problems:
          1. register spilling – when many local variables, not enough registers to store them 🡪 stored on stack (slow) and rearranged to make fit
          2. branch misprediction penalty

bad for switch statements’ inheritance b/c cause mispredicted jmp

slow when data is random/hard to predict 🡪 conditional move instructions



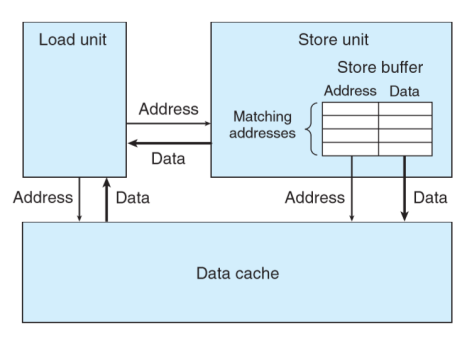
Pipelining (type of parallelism):

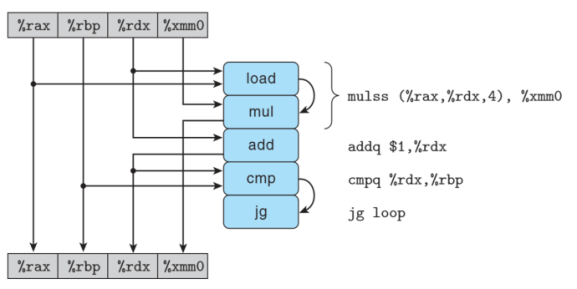
* + technique to increase throughput (# instructions done per cycle) by breaking up basic instruction cycle to run different steps concurrently/in parallel (combined w/ superscalar execution)
  + arrangement of functions so that different portions of an operation flow through a particular set of sub-functions, with the sub-functions happening in parallel
  + fully pipelined = max throughput of 1 op/cycle
    1. IF (fetch instruction fr memory)
    2. ID (decode instruction)
    3. EX (execution)
    4. MEM (memory access)
    5. WB (write back temp computation to registers)
  + problem: conditionals

Out of Order Execution (OoO)

* + increases throughput more than pipelining

Load and Store:

* + load CPE >= 1 (can be fully pipelined)
  + load performance depends on pipelining capability and latency of load unit
  + store CPE >= 1 (can be fully pipelined)
  + write-read dependency – load has to wait for store to finish
  + Store Unit:
    1. store buffer – has addresses/data of store operations that haven’t been completed (completion updates data cache) 🡪 faster b/c updates registers at end instead of 1 by 1
    2. when loading, refers to store buffer for matching address
    3. along w/ load unit talks to ICU for instructions and registers for address/data and data cache for address/data

Modelling Processor Operations:

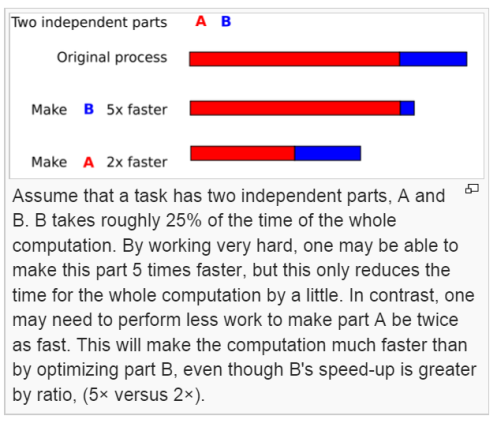
* + use data-flow graphs to find critical path (path of instrns slowing us down the most in terms of updating registers, that waits longest)

Performance Improvement Techniques:

1. Better algorithms/data structures
2. avoid optimization blockers
   1. avoid excessive function calls and memory references w/ temp variables, do code hoisting
3. low-level optimization
   1. loop unrolling
   2. ILP
   3. conditional data transfers, not conditional control transfers

* when should intro optimization in C code even though compiler does a lot for us? compiler guarantees code will be functionally same to avoid volatile data(register data changes every cycle)/memory aliasing
  + when does loop unrolling not help? if increased critical path
* re-association – FP not associative, must be explicit for compiler, could be used with loop unrolling
  + can we use commutativity like loop unrolling and re-association?
    - no b/c hardware already uses commutativity to improve performance and switching order would not doing anything

Performance Bottlenecks:



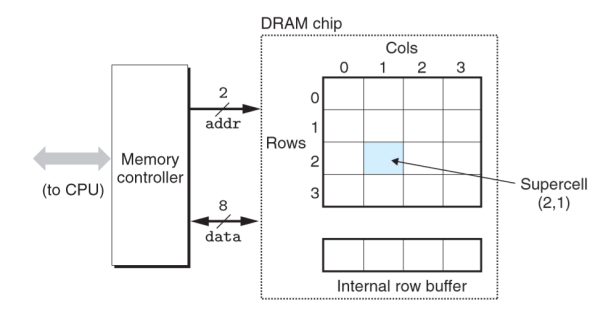
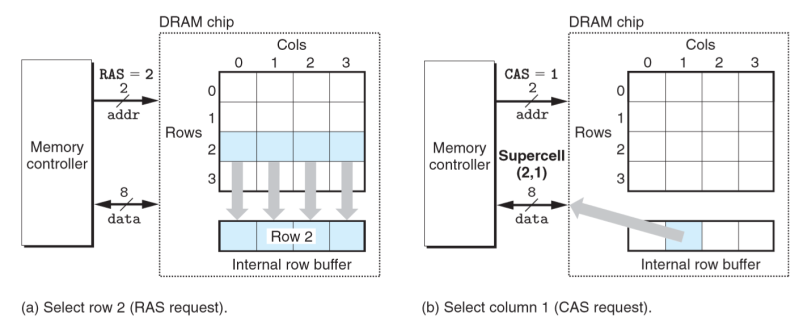
**|**

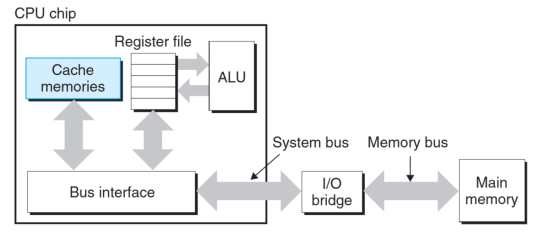
**|**

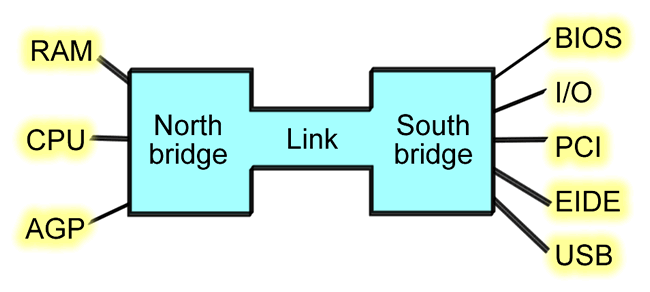
**|**

* + code profilers w/ GPROF (unix> gcc -O1 -pg prog.c -o prog)
    1. collect performance data during execution
    2. det. how much CPU time certain functions of program takes, count of how many times function called
  + Amdahl’s Law
    1. used to find maximum performance improvement by changing 1 part of sys
    2. when improve part by factor, overall performance improve by less than that factor
    3. models expected speed up when problem size stays same

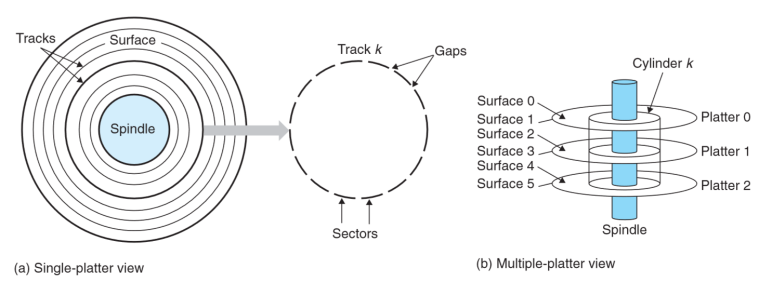
**Storage Technologies:**

1. RAM (random access memory)
   1. assertion about performance that speed accessing memory same everywhere, but this is false
   2. VRAM (volatile) – needs power or else lose info
      1. SRAM (static)
         1. faster
         2. 6 transistors/bit
         3. stable
         4. used for cache memories
      2. DRAM (dynamic)
         1. cheaper
         2. 0 transistors/bit
         3. 1 capacitor/bit
         4. leaky, unstable, decays
         5. 10-100 ms
         6. must refresh by read out and rewrite memory b/c memory disappears fast
         7. used for main memory
   3. Non-volatile Memory
      1. ROM (read-only memory)
      2. PROM (programmable exactly once)
      3. EPROM (erasable programmable ROM)
      4. flash memory – fast and durable
      5. firmware – programs in ROM

Memory Hierarchy:

1. CPU registers (highest)
2. caches
3. main memory
4. Accessing Main Memory
   1. data flows betw DRAM and processor through buses by bus transaction (read (DRAM🡪CPU) or write (CPU🡪DRAM))
      1. system bus – CPU 🡪 I/O bridge
      2. memory bus – I/O 🡪 main memory
   2. bus interface create read transaction on bus
      1. CPU place address on system bus
      2. I/O bridge pass signal to memory bus
      3. main memory reads address
      4. write data to memory bus

more modern computer architecture

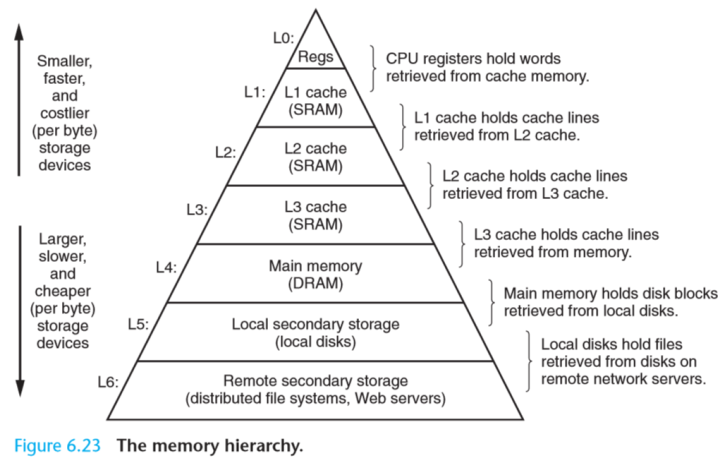
1. in disk (lowest)
   1. RW data in sector sized blocked
   2. access time (3 components)
      1. seek time – read contents
      2. rotational latency – waiting for first bit of target sector to pass under head
      3. transfer time – actual RW process

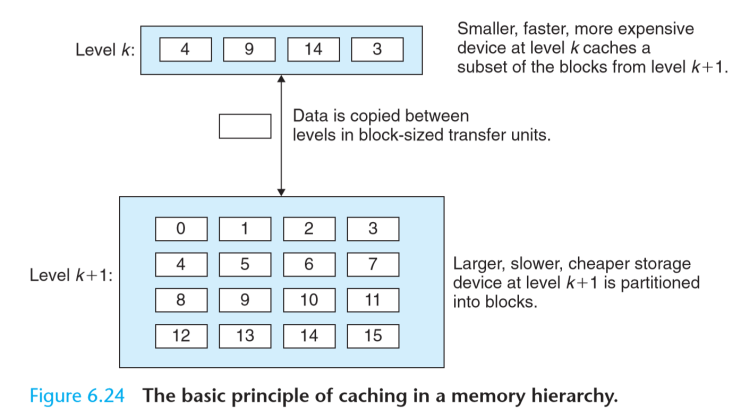
Locality

* property describing access of memory in memory hierarchy
  + good = reference data near other recently accessed data
* temporal – nearby time, memory location likely to be referenced again in near future
* spatial – nearby location, memory likely to be referenced near previously referenced location

Memory Hierarchy

* different storage tech has diff access time
* each level up serves as cache for lower level



1. Caches
   1. small fast storage device for program data/instructions from main memory invisible to program
   2. storage at level k+1 partitioned into blocks and level k has copies of subset of k+1 blocks
      1. larger cache means higher hit rate (fraction memory that hit) but slower
      2. larger blocks means more spatial locality but hurt hit rate
   3. block – fixed sized packet of info moves betw cache and main memory (or a lower-level cache)

line – container in cache stores block

set – collection of few lines

* 1. transfer units – units of data transfer betw 2 levels
  2. how to find item: hash function mapped to cache line #
  3. updating cache: decision of updating RAM at same time or after cache to reflect whats in cache
     1. cache hit
        1. finding needed data from k+1 when look at level k first (faster than direct k+1)
        2. read hit – easy
        3. write hit
           1. if update higher level, update all lower levels by:

write-through

write cache to next lower level immediately

con: cause bus traffic

pro: better latency if multicore, simpler

write-back

wait as long as possible before writing, has “dirty bit” to show if new value in cache is different than memory

only updates lower level when cache line block is evicted

avoids unnecessary stores, complex

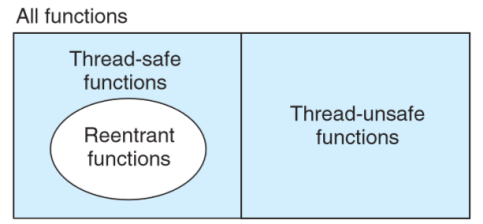
has to remember to modify RAM

no-write-allocate

bypass cache and update lower level

* + 1. cache misses
       1. finding needed data from k+1 but not found in k, so must fetch from k+1 (possible overwrite of block)
          1. aka replacing/evicting
       2. if level k empty, cold miss
       3. read miss
          1. what want to read not in cache b/c cache exhausted or hashing didn’t work well
       4. write miss
          1. write allocate: load and store
          2. no write allocate: write data directly to RAM w/o caching
  1. Types of Caches
     1. i-cache: only instructions (R), close to ICU
     2. d-cache: only program data (RW)
     3. unified cache – holds both instruction + program data

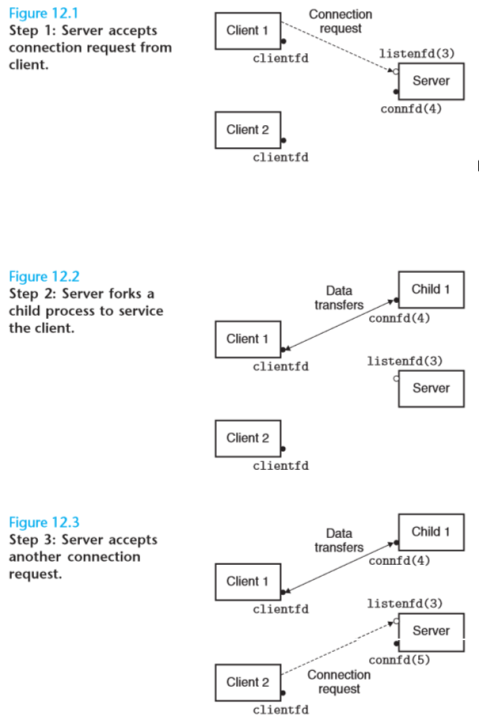
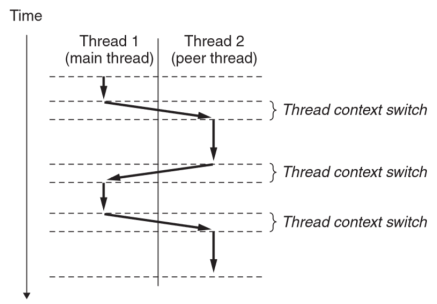
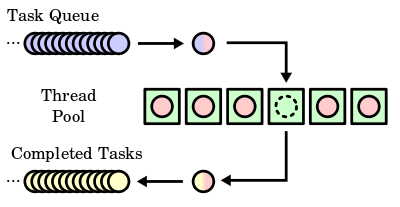
**Concurrency:**

* multiple logical control flow overlap in time
* problems:
  + thread-safe:
    - always produce right results when called repeatedly fr multiple concurrent threads
    - classes:
      * functions don’t protect shared variables
        + solution: semaphore synchronization
      * functions keep state across multiple invocations
        + ex: rand() current call result depends on last call
        + solution: rewrite so does not use static data
      * functions return pointer to static data (ptr val could be overwritten by other threads)
        + solution: mutex
      * functions call thread-unsafe functions
    - reentrant functions: 2 instances of function call in threads won’t collide even w/ signal handlers, don’t reference any shared data, can be interrupted in mid-exec and called again before completing
      * explicitly reentrant if all arguments passed by value
      * implicitly reentrant if arguments passed by reference are sure to be not shared
      * ie malloc not reenetrant b/c share heap
  + race conditions: how logical flows are scheduled
  + deadlocks: when flow is waiting for event that will never happen
* speed-up:
* efficiency:
* scaling
  + strong: add cores, oriented for real-time + latency
  + weak: add cores + work proportional to cores added, oriented for servers + throughput

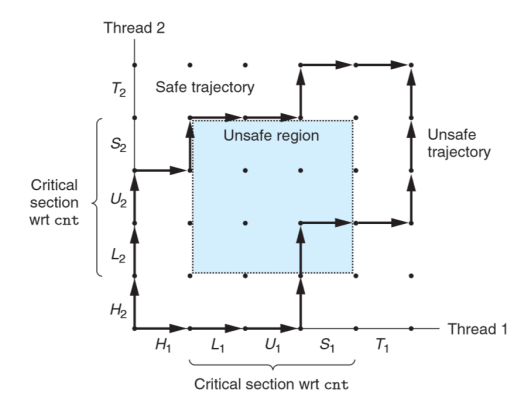
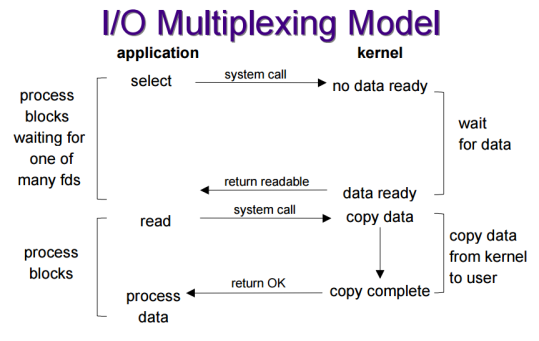
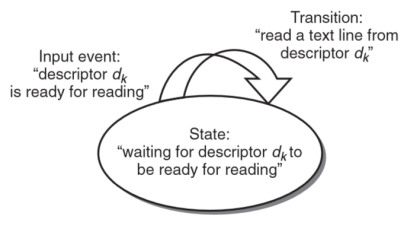
Synchronization:

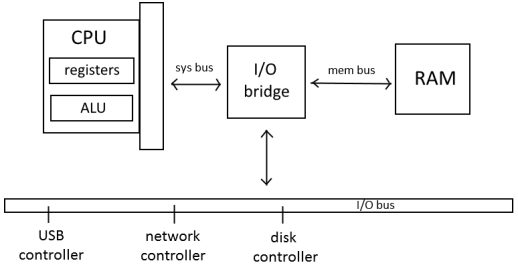
* multiple processes/threads running concurrently, hopefully not racing or conflicting about shared variables

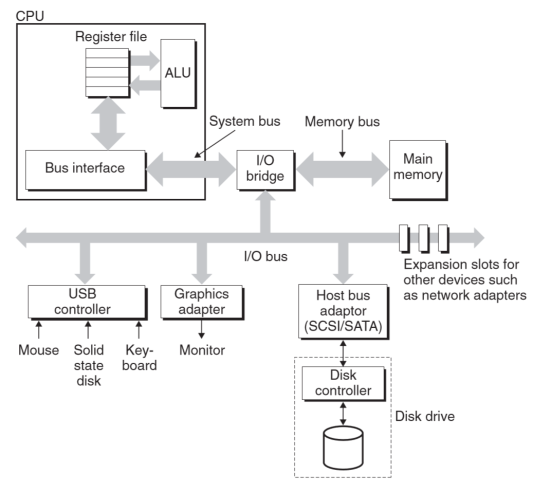
Processes + Threads + I/O Multiplexing:

* process: model/abstraction of execution, running program w/ exclusive access to hardware
  + has private address space, doesn’t share memory w/ other processes
  + unique set of registers, stack
  + managed and scheduled by OS (indep of other processes)
    - to create new process needs privileged instructions
    - unix system call fork() to create new process
      * process makes copy of itself w/ same parent stack and registers
      * fork returns process ID (PID) of spawned child to parent
        + tell parents and child apart by return val of fork (child PID=0)
    - when killing process
      * parent notified of child process’s data and child process PID removed fr process table
      * parent must reap child data or else child PID no leave process table (now zombie) and must be reaped or else take up resources
      * waitpid used for process to wait till children are reaped/terminate
  + interprocess communication (IPC)
    - hard for communication b/c not aware of other processes
    - IPC channels:
      * OS allocated shared memory segments, signals, files 🡪 expensive b/c have to go through OS (lots of overhead)
  + making a web server:
    - accept client connection requests in the parent, and then create a new child process to service each new client
      * server forks child after accepting a request
      * child closes listening descriptor
      * parent closes connected descriptor
        + imp or else entry for connected descriptor never released 🡪 memory leak and crash
      * must have handler reap zombie children after process terminates and remove it from process table
      * connection won’t end until connfd of parent and child closed
* solution: threads
  + logical flow sharing context of program, have minimal things to be executable, own thread context (stack/registers/thread ID (TID))
  + has own stack, stack pointers rsp/rip, registers, memory(program, static data, heap)
  + favored over processes to avoid OS even though processes have own memory
  + start off w/ main thread and forks into peer threads (but no hierarchy as in process parent/child)
  + difference w/ process:
    - multiple threads can exist in 1 process, share single address space so stack addressable by other threads but registers not accessible
    - diff address space for process, same address space for threads
  + POSIX Threads
    - standard interface to manipulate threads in C programs
    - thread routine – contains code and local data for thread
    - Functions:
      * int pthread\_create(pthread\_t \*tid, pthread\_attr\_t \*attr, func \*f, void \*arg);
        + attr: changes attributes of new thread, usually NULL
        + new thread starts w/ execution of function f
        + arg: input argument for function f
        + tid: on return has ID of new thread = 0 if success
      * pthread\_t pthread\_self
        + to get own PID
      * void pthread\_exit(void \*thread\_return);
        + explicit termination (implicit when top-level thread dies)
        + if main thread calls this, waits for peer threads to terminate before self-terminating *along with process*
        + return value = 0 if success
      * int pthread\_cancel(pthread\_t tid);
        + terminate current thread
        + returns 0 if success
      * int pthread\_join(pthread\_t tid, void \*\*thread\_return);
        + default unsafe behavior joint (can be killed by other threads)
        + waits until tid terminates, **reaps threads**
        + thread\_return points to value passed into pthread\_exit(parameter) to terminate
        + returns 0 if success
      * int pthread\_detach(pthread\_t tid);
        + detached thread cannot be killed by other threads (autonomous), memory is freed by system on program completion
  + thread scheduling
    - scheduled by process in userspace (OS doesn’t know they exist)
      * but pthreads scheduled by OS
    - can’t be parallelized if not OS managed
    - can share global variables and system resources w/in same process
    - local variables can be accessed by diff thread 🡪 thread safety
      * safe if can’t be accessed/changed
    - OS no guarantee order of execution across threads or exclusion
      * causes race condition b/c multi-threaded programs could overwrite variables of one another and have race condition
    - usually have master thread with worker threads to complete tasks to prevent making too many threads
  + making a web server:
    - main thread waits for connection request and makes peers thread to handle request
    - cannot pass in pointer to connection descriptor or else will cause race condition between assignment and accept (depending on order executed could mess up what is connected)
    - must detach thread to reap
  + **Semaphores(s)**
    - to protect against shared variable problems
    - global int variable used only to control access to other global variables
    - keeps track of how many available resources, but if only 1 called binary semaphore aka mutex
    - w/ value 1 = locks; val = 0 = unlocked
    - operations:
      * P – if s!=0, s-- and return; if s==0 wait till !=0 to read and restart w/ V then do P
      * V – s++; restarts thread to make available, not in order 🡪 ***can’t predict thread order***
      * P and V must be atomic (sequential execution) so they don’t get race conditions themselves
      * critical section = period in between P and V
      * ensure program can never enter state w/ s having negative val (semaphore invariant)

#include <semaphore.h>

* + - * int sem\_init(sem\_t \*sem, 0, unsigned int value);
        + initializes sem to value
      * int sem\_wait(sem\_t \*s); /\* P(s) wait \*/
      * int sem\_post(sem\_t \*s); /\* V(s) signal \*/
      * all return 0 if success
    - readers-writers problem: many readers can read value at once, but writing value needs exclusive access (locked only if thread writing into global variable)
      * solution: keep track of # of readers and whether writing lock is used (0 or 1) 🡪 favor reader
        + if reader wants write lock, wait, decrement reader count, write, then release write lock
        + if writer wants write lock, wait to get lock, write, release lock
        + if readers keep coming, writers never get turn 🡪 starvation
    - Usage:
      * associate 1 semaphore per shared variable and surround critical section with P and V ops
  + volatile: value can be changed in between accesses, cant optimize, must load and store every time accessed
  + static: variable lifetime extends whole program run
* if threads < processes: too many CPUs, wasted computer, but easy to tune
* if threads > processes: many parts want to run but not enough CPU, some threads have to wait to run(decided by OS), hard to tune/need optimization
  + **context switch**: OS take away CPU and give to another to run in between instructions 🡪 mutated code, stores registers into memory and restores other threads registers from saved area
* process graph
  + models execution of concurrent threads that have shared variable
  + shows each thread should have mutually exclusive access to variable
  + unsafe region: intersection of critical section (where shared variable accessed) for 2 threads
* I/O Multiplexing
  + server responding to user input for:
    - making connection request
    - user typing command
    - problem: which to wait for first?
    - solution: I/O multiplexing
  + use select to get kernel to suspend process and return control to app after I/O events
    - int select(int n, fd\_set \*fdset, NULL, NULL, NULL);
      * wait for/detect event to occur (ready to R/W)
      * waits for descriptor set called read set (fdset) to be ready for reading and manipulates them by (allocating them, assign variable, read and modifiy)
      * fd\_set – set of descriptors want to wait on for events
      * n = number elements
      * returns when ready for I/O
  + if input ready use command to read, parse, and respond to command
  + if listening descriptor ready use accept to get connected descriptor and echo line from client till close connection
  + making an event-driven web server
    - model controls flow as state machines (map states to events) and events trigger transitions to next state
    - for each new client, new state machine made with connection descriptor
      * each state machine has 1 state, 1 input event, 1 transition
    - select function detect input events
    - when each connected descriptor ready for reading, server executes transition for corresponding state machine, and reading/echoing a text line from descriptor
    - pros: more control, access to entire address space to share data, more efficient than processes
    - cons: complex, can’t fully use multi-core processors

**I/O**

* process of copying data between memory and external devices
* To read to CPU fr disk:
  + CPU sends command to disk controller
  + disk controller copies data to RAM
  + disk controller signals CPU that completed
* at HW level, 2 tiers:
  + betw. CPU/device controller
  + betw. device controller/physical device
* Components of Latency for Disk
  + queueing time – set-up (time betw. request fr CPU and action by disk, electrical)
  + seek time (mechanical) – read contents
  + rotational delay (mechanical) -- waiting for first bit of target sector to pass under head
  + transfer time (mech/elec) – actual RW process
* flash
  + not mechanical, faster, expensive
  + How to make faster?
    - faster arms (improve seek time)
    - rotate spindles faster (improve rot. delay)
    - denser recordings (most effective)
    - larger cache/flash
    - faster bus (doesn’t work b/c fast enough)
    - more drives to read in parallel
    - use smarter controller (exec requests in optimal order aka out of order)
* CPU – I/O Interfacing
  + memory-mapped I/O (CPU commands to I/O)
    - I/O bus share addr as mem; talks to device on bus, easier to program
    - ie: movl %eax, 0x\_\_\_\_\_(special I/O address)
  + special I/O insn
    - use load/store insn that’s also used for RAM for I/O, errors less likely
      * ie: &x = 0x\_\_\_\_\_; movl $3, x; volatile int x = \_\_attribute(\_location(&x))
    - ie: outl $3, 0x\_\_\_\_\_; inl 0x\_\_\_\_\_, %eax
  + Direct Memory Access (DMA)
    - device performs RW bus transaction w/o CPU by talking to mem directly
    - can do fr:
      * I/O device🡪RAM
      * RAM🡪I/O device
      * I/O🡪I/O
      * RAM🡪RAM
    - issues
      * device must be programmable
      * know source/destination address
      * transfer/block size
      * how to notify CPU when done (interrupts)
    - problems w/ cache coherency (consistency of shared resource data)
      * if DMA access RAM, RAM may not hold updated cache value which has not been written-back yet
      * fix:
        + 2 bits in cache (dirty+valid)
        + explicit insn saying invalid cache b/c CPU knows what device contains

bad b/c violate rule that program has to work even w/o cache

* + - * + cache snooping

cache monitor address for memory accesses, look at other device stores to RAM

don’t need to change prog

works for multiple CPUs b/c indep bus masters

CPU can keep track + invalidate min. amount mem.

* Buses
  + types:
    - control line (0 or 1, ie clock)
    - address line (RAM addresses)
    - data line (values stored into RAM)
  + more buses: faster, more expensive, more power needed
    - choose betw multiplex(cheaper) vs. separate lines(faster)
  + split transaction bus
    - normal bus: when read data fr RAM usually CPU READs and mem respond w/ val.
    - split transaction:
      * CPU can send multiple read request to RAM w/ tag (int)
      * RAM respond w/ assignment tag # to data
      * allows parallelization 🡪 answers come back out of order
      * tags maps request with response
  + burst requests – 1 command send several bytes to CPU at once (saves on communicating address)

1. Opening
   1. int open(char \*filename, int flags, mode\_t mode); (ret -1 if error)

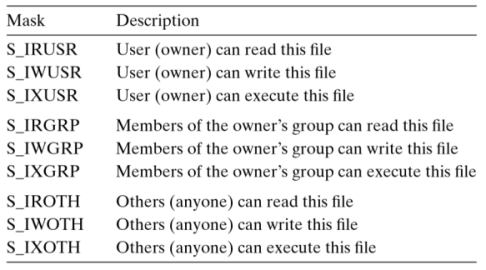
Flags:

* + 1. O\_RDONLY: Reading only
    2. O\_WRONLY: Writing only
    3. O\_RDWR: Reading and writing

CAN BE OR’ED WITH

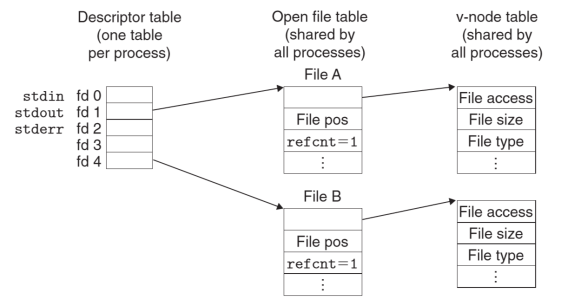
1. O\_EXEC: exec only access
2. O\_CREAT: if ﬁle doesn’t exist create a truncated (empty) version of it
3. O\_TRUNC: restart/discard before output; if ﬁle exists, truncate it
4. O\_APPEND: write only to end w/o modify file

ie: fd = Open("foo.txt", O\_WRONLY|O\_APPEND, 0)

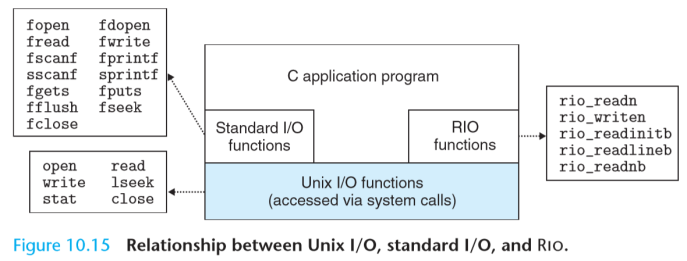
 Mode: permissions

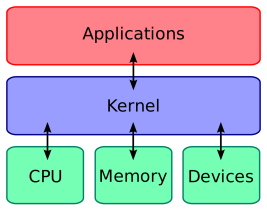
1. Closing
   1. int close(int fd);
      1. ret -1 on error, 0 if ok
2. Read
   1. ssize\_t read(int fd, void \*buf, size\_t n);
      1. returns number of bytes read if OK, 0 on EOF, −1 on error
3. Write
   1. ssize\_t write(int fd, const void \*buf, size\_t n);
      1. returns number of bytes written if ok, -1 on error
4. Robust I/O (RIO):

* solution to short counts for network applications (R/W fewer bytes than application requested b/c of EOF on read, R fr terminal, RW fr network socket)
* I/O package handle repeated calls to RW to transfer all requested bytes
* functions:
  + unbufferred in/outputs -- transfer data directly betw memory/file w// no buffering
    - Calls to rio\_readn and rio\_writen can be interleaved arbitrarily on the same descriptor (thread safe) each function manually restarts the read or write function if it is interrupted by the return from an application signal handler
  + bufferred input -- read text lines/data fr file cached to buffer (thread safe b/c can read text lines then data then text lines)
    - The rio\_readinitb function is called once per open descriptor. Calls to rio\_readlineb and rio\_readnb can be interleaved arbitrarily on the same descriptor (thread safe). However, calls to these buffered functions should not be interleaved with calls to the unbuffered rio\_readn function.
* reentrant: dont reference shared variables, can be interrupted in middle of exec and re-called safely

1. Unix I/O

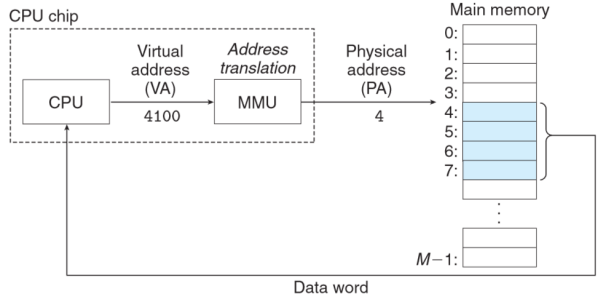
* descriptor – ID file working with, positive int
* descriptor table – entries point to entry in file table (set of open files)
* each process begins w/ standard input (STDIN\_FILENO), output (STDOUT\_FILENO), error (STDERR\_FILENO)

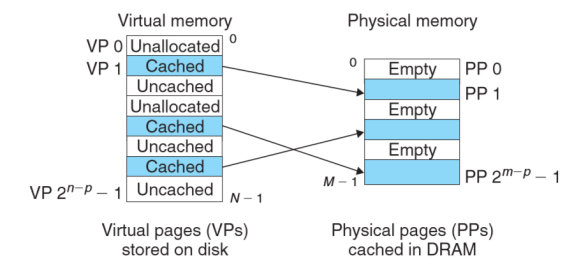
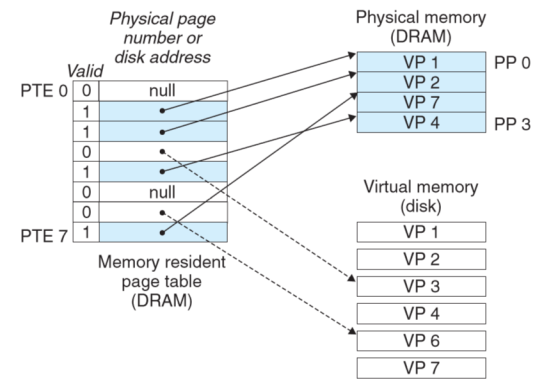
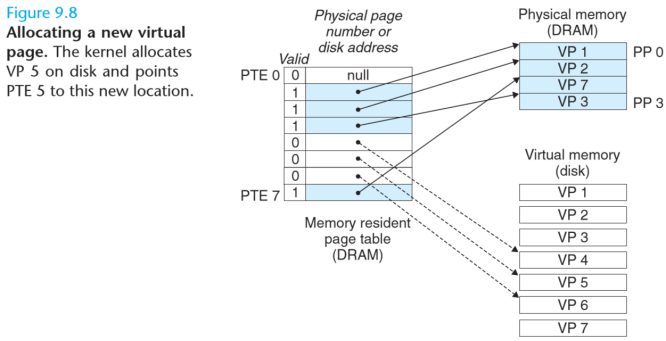
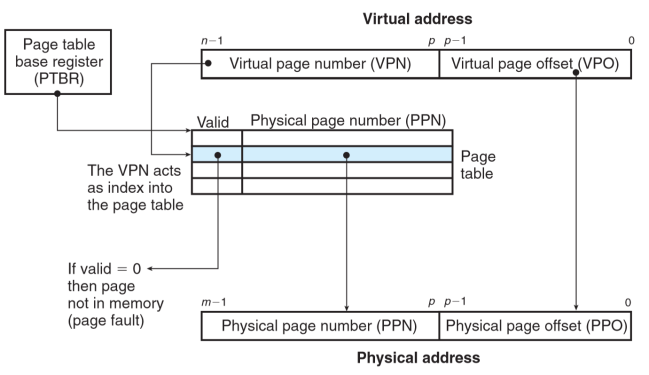
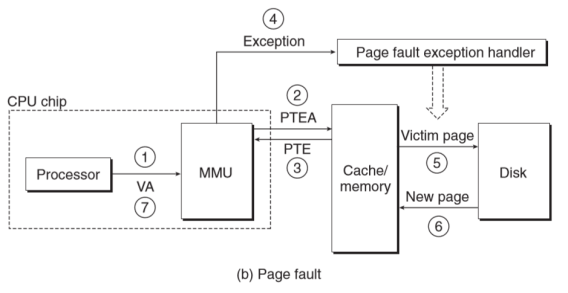
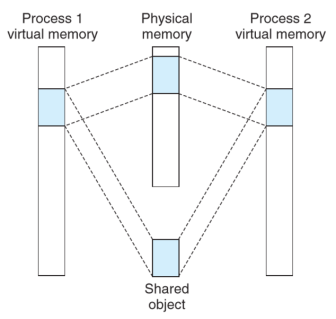
Summary



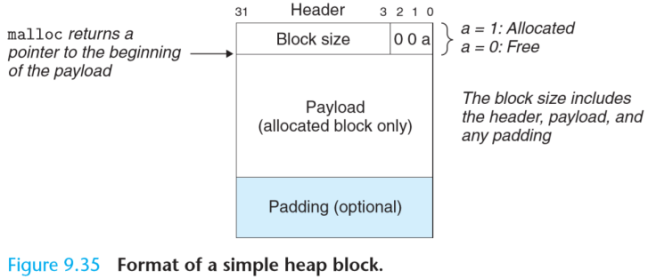
Kernel:

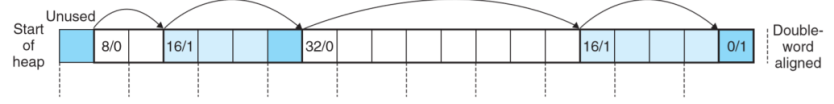
part of OS that controls all programs in computer, manages I/O requests and translates into insn for CPU

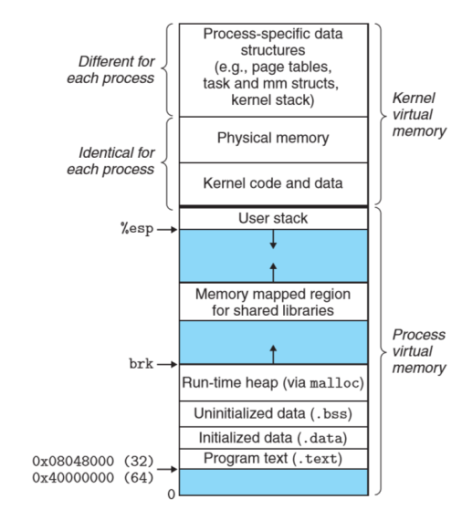
**Virtual Memory**

* Capabilities:
  + uses main memory efficiently by treating it as cache for address space on disk, keeps only actively used memory, transfers memory when needed
  + provides each process with own address space
  + protects address space of processes
* temporal locality
  + virtual memory efficient enough b/c working with small set of active pages
  + thrashing: constant swapping of pages b/c bad temporal locality
* Physical/Virtual Addressing
  + RAM has physical addresses which are translated between virtual addresses
  + memory management unit (MMU) – translates virtual addresses
  + virtual address space = 2n where n = bits on computer
* How virtual addressing is used:
  + virtual and physical memory split into memory chunks (aka virtual pages)
  + virtual pages are either
    - unallocated
    - cached in physical mem
    - uncached in physical mem
  + **page tables** (map/cache virtual to physical mem), MMU, and OS det. if page is cached or not in DRAM
    - if virtual page cached somewhere in DRAM find physical page, if cache miss find memory on disk and copy virtual page to DRAM
    - page tables:
      * array of page table entries (PTE)
        + valid bit – if page currently cached in DRAM
        + address field – start of physical page in DRAM where virtual page cached
      * each page in virtual address space has fixed offset in page table
  + swapping/paging – transferring page betw disk and memory
    - swapped/page in – disk to DRAM
    - swapped/page out – DRAM to disk
    - demand paging – wait until last moment to swap in when miss occurs
  + multiple processes
    - virtual space + page table for each process
    - multiple virtual pages can be mapped to same physical space
* Allocating Pages (making new virtual page)
* role in memory management
  + simplifies linking
    - separate address space keeps memory arrangement the same for each process 🡪 independence of actual physical location
  + simplifies loading
    - memory mapping virtual pages to arbitrary locations
  + sharing code/data
    - for IPC, multiple processes can share same physical page even w/ diff virtual pages
  + allocating mem.
    - allocate physical pages in separate spots in RAM, does not have to be contiguous
* role in memory protection
  + PTE includes extra info on permissions of that page
  + if violated 🡪 segmentation fault
* Translation
  + page table base register – points to page table
  + virtual address has:
    - virtual page offset
    - virtual page number
  + MMU uses virtual address to find corresponding PTE
  + physical address has:
    - physical page number from PTE
    - same offset as in virtual page number
  + page hit
    - CPU makes/sends virtual address to MMU
    - MMU requests PTE address from RAM
    - RAM accepts and sends PTE to MMU
    - MMU makes physical address and sends to RAM
    - RAM sends back data in that location
  + page fault
    - when program accesses virtual address space but not loaded into physical memory; response to make physical page available; cache miss
    - Steps:
      * CPU makes/sends virtual address to MMU
      * MMU requests PTE address from RAM
      * RAM accepts and sends PTE to MMU
      * valid bit in PTE 0
      * page fault exception handler
        + control transferred to kernel
        + choose diff victim virtual page (that is cached to memory) and modifies to show its not cached in DRAM
        + swaps places with current PTE (copies original virtual page looking for into corresponding physical page, updates page entry table)
        + restart insn (now original page cached in memory)
  + Translation Lookaside Buffer (TLB)
    - cache of PTEs in MMU that speeds up translation
    - Steps:
      * hit:
        + CPU makes virtual addr
        + MMU gets PTE fr TLB
        + MMU translates virtual addr to physical, sends to cache/RAM
        + cache/RAM sends back data to CPU
      * miss:
        + MMU get PTE fr L1 cache
        + PTE overwrites entry in TLB
  + Nehalem Core i7:
    - sparse tree representation b/c lots of empty space
    - VPN uses multi-level page tables
* memory mapping
  + allows shared objects in virtual memory that refer to one physical address space for commonly used code instead of having to make copies for each process
  + memory initialized by associating it w/ object on disk
  + makes new areas of virtual memory/maps objects to them:

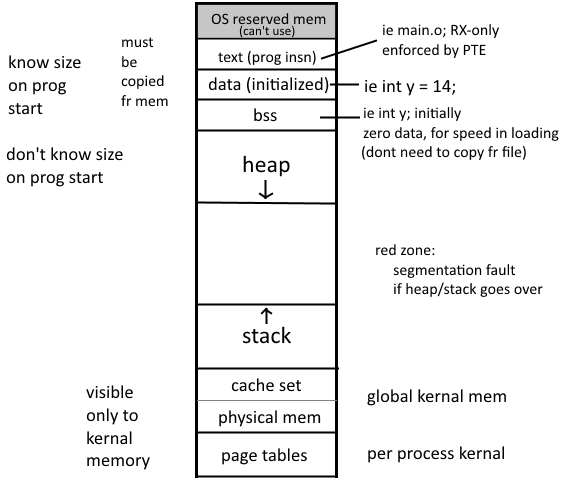
void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);

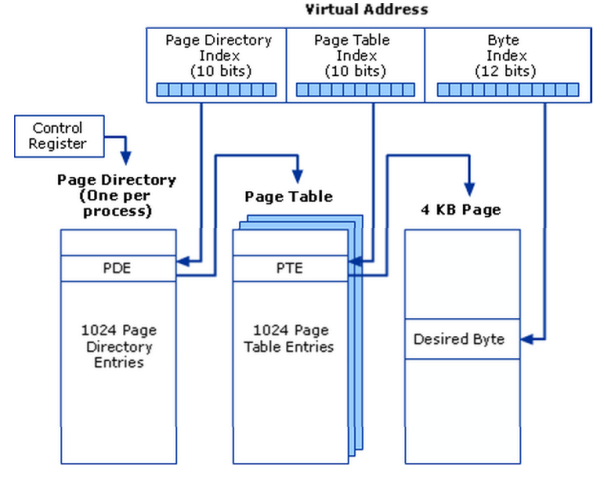
* + - protects page table w/ permissions
    - new memory begins at start, continues length bytes, starts at offset
      * fd – file descriptors, where in storage mem stored
      * prot options: protection/permissions
        + PROT\_EXEC – pages may be exec
        + PROT\_READ
        + PROT\_WRITE
        + PROT\_NONE – pages can’t be accessed
      * flag options: what to do with pages
        + MAP\_ANON – demand 0 mem (like bss)
        + MAP\_SHARED – shared among processes
        + MAP\_PRIVATE – unshared among processed, copy on write when want to modify
* dynamic memory allocation
  + allocation issues:
    - free block organization – keep track of free blocks
    - placement – how choose block to place
    - splitting – after placing allocated block in free block what do w/ freed block
    - coalescing – what to do with block when freed
  + as implicit free list
    - various sized blocks allocated in heap linked implicitly by header size field
      * header – holds block size, free or not
      * payload – size that was requested during malloc
      * padding for alignment
    - advantages:
      * simple
    - disadvantages:
      * linear search of blocks
      * minimum block size required b/c of alignment
    - if running low on memory, use mmap
    - placing blocks:
      * first fit – search fr beginning for space
        + pro: leave large free block at end
        + con: leaves small gaps between blocks
      * next fit – continues where last search left off
        + pro: faster
        + con: worse mem utilization
      * best fit – look at every free block and choose smallest size that fits
        + pro: better mem utilization
        + con: long search
    - splitting free block: use whole block or split it into 2 if malloc request not big enough
    - coalescing
      * freeing mem cause fragmentation (lots of free small chunks of scattered memory)
      * options:
        + immediate coalescing – merge adjacent free blocks when freeing
        + deferred coalescing – wait to merge later



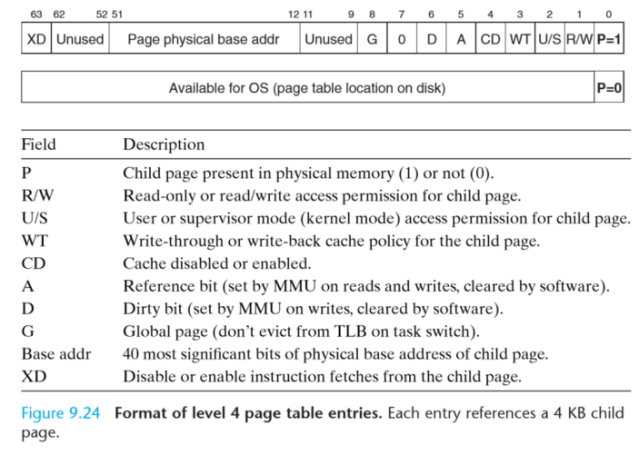


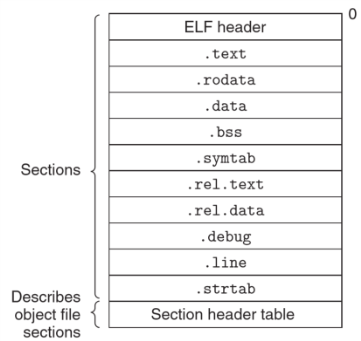
* Virtual Memory Layout

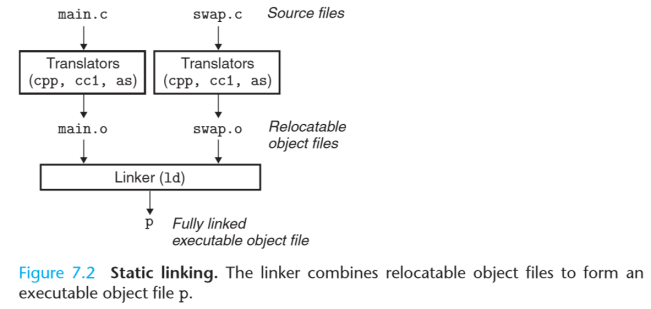
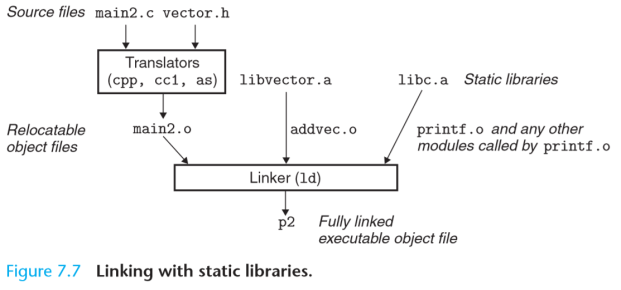


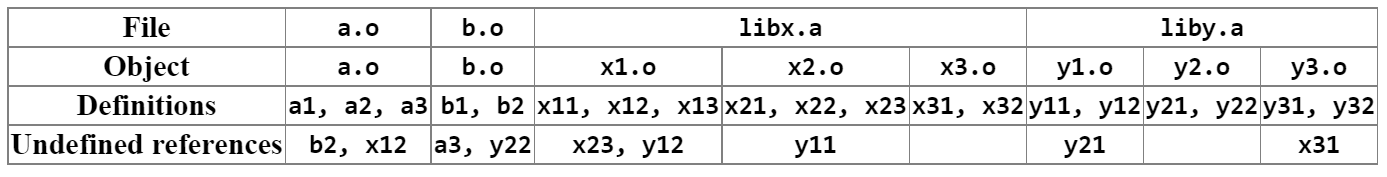
****

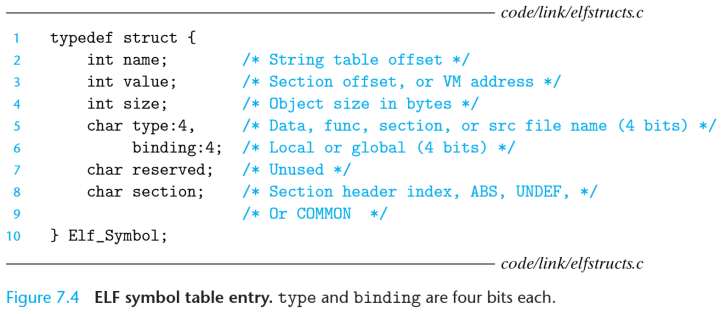
Multi-level Page Tables:



**Linking:**

* static linking at compile time, dynamic linking during runtime
* object files
  + relocatable (static) – has binary code/data to be combined into executable
    - header: word size/byte ordering of sys
    - .text: machine code of program
    - .rodata: read-only data
    - .data: initialized global variables (RW)
    - .bss: uninitialized global variables (RW)
    - .symtab: symbol table w/ info about functions/variables
    - .rel.text: locations in .text that will be modified after linking
    - .rel.data: relocation info for global variables
    - .debug: symbol table for local variables
    - .line: maps program line numbers w/ assembly code
    - .strtab: string table for symbols in symtab/debug
  + executable – binary code/data that can be copied into mem and executed
  + shared (dynamic)
* linker functions:
  + make executable
  + symbol/name resolution – object files define/reference symbols; fill in missing pieces of code for undefined symbols; gives error if reference not properly found/linked
  + relocation – moves around code blocks by mapping mem to symbol to reference data
* **STATIC LINKING**
  + share code by reusing object files; copies libraries used in program into executable
  + links relocatable object files and arguments into an executable object file w/ section of instructions, initialized global variables, uninitialized global variables
    - ***static library*** (ie libm.a stored in archive) made from multiple relocatable object files and input to linker, but only use obj files referenced in program
      * saves memory and don’t have to make copies of same relocatable file per process, but does for multiple processes
      * but when make changes have to re-build and compile



* + disadvantages:
    - have to be maintained/updated and relinked when changes made
    - keeps multiple copies of code for each process (waste memory)
    - order of linking matters (definition follow reference; libraries linked last)
    - includes object files of libraries for every symbol even if library may have many undefined symbols that not used in program and will need to find 🡪 pull in more library object files
* relocatable object files (.o)
  + Symbol
    - Types:
      * global: non-static variables, defined by current module and referenced by others
      * local
        + defined and referenced in current module
        + local variables no entry in symtab, managed on stack during runtime
      * extern keyword
        + variable/functions defined somewhere else in program
        + implicit for functions
        + variable usage:

declares w/o defining (**not allocated**):

ie extern int var;

declared + defined normally:

ie int var;

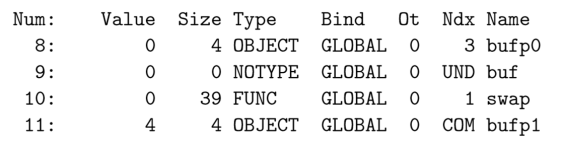
* + - * static keyword
        + variable/function exist throughout program runtime but only accessible in function declared
    - Symbol Resolution
      * each reference associated w/ 1 symbol table entry
      * can have multiple declarations (no allocation of memory), not multiple definitions (allocates memory)
      * local symbols defined once per module
      * global symbols if not defined in current module assumed to be defined in another, makes empty symbol table entry, lets linker handle
        + if linker can’t find, error
      * if multiple globals defined multiple places:
        + each symbol **strong** (functions/initialized global variables) or **weak** (uninitialized global variables)
        + Rules:

multiple strong symbols disallowed

choose strong over weak symbol

if multiple weak, choose any

* + - Symbol Table



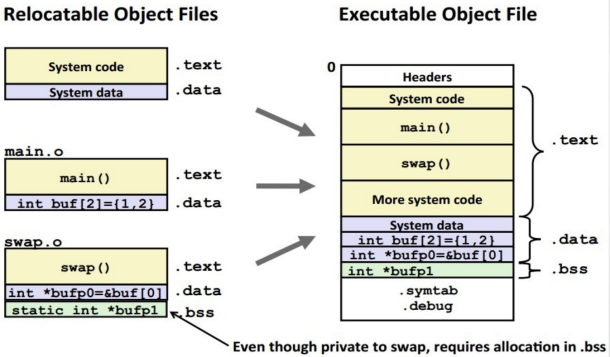
* + - Relocation
      * assign addresses to symbols code/data so they can be referenced
      * Steps:
        + Relocation sections + symbol defs

merge all sections of same type (ie .data)

unique addresses given to each symbol

* + - * + Relocating symbol references w/in sections

get symbol references in code to point to address of corresponding symbol

relies on relocation entries composed of:

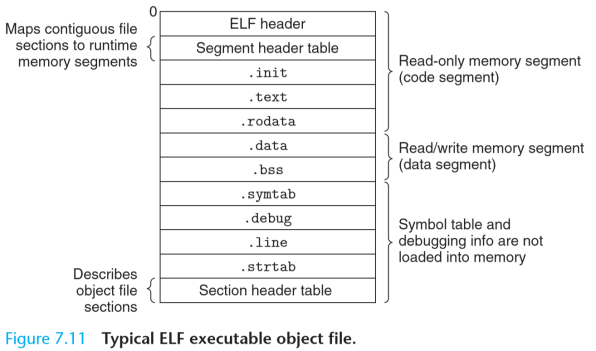
offset – location of insn need to be relocate to

symbol – index in symtab where insn ptr is

type

R\_386\_32 – 32-bit absolute address

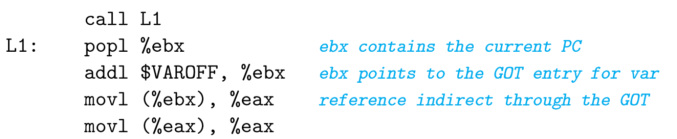
R\_386\_PC32 – 32-bit relative address

* executable object files
  + same as relocatable object file but no relocation segments, more symbols (ie \_init, \_bss\_start, \_fini)
  + loader copies data/code into mem and passes control to program
  + entry point: address of first insn to execute
* **DYNAMIC LINKING** (.so file)
  + shared library/object:
    - libraries placed in executable and during run-time will reference address of library code/data module **when needed** (not copied even by multiple processes)



view of program from linker POV:

* + - linker notes down symbol name + library come fr
  + program modifies self to link to new libraries not available when building program
    - pros:
      * allows fixing/altering many programs at once by updating the single copy of library which will be reflected in the references to that library
      * applications during run-time can request to load/link libraries
      * no need to re-compile, already linked files
    - con:
      * easy to break program if update library with incorrect/incompatible info
      * slows exec
  + position independent code (type of shared library)
    - solution to having multiple processes refer to shared block of memory for code (problems in memory management/allocation)
    - compile library code so can be loaded/exec at any addr
    - PIC Data References
      * b/c data segment always after code section can make ***global offset tables*** (GOT) (start of .data section)
        + GOT has entry for each global data object referenced and its relocation record
        + compiler relocates each GOT entry to absolute addr
        + each global var referenced using following code

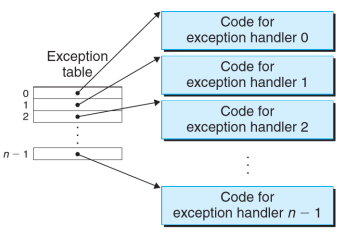
loads address of current location plus some offset into ebx and gets global variable data fr GOT

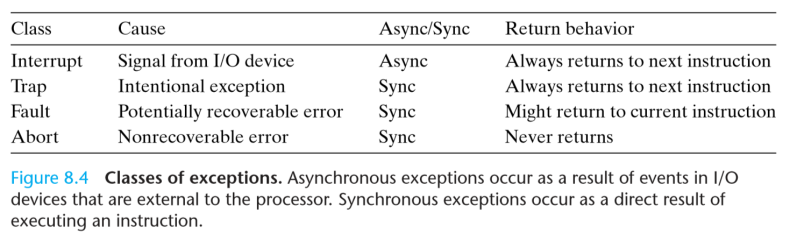
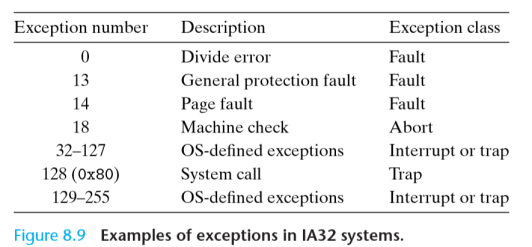
* + - * + performance disadvantages:

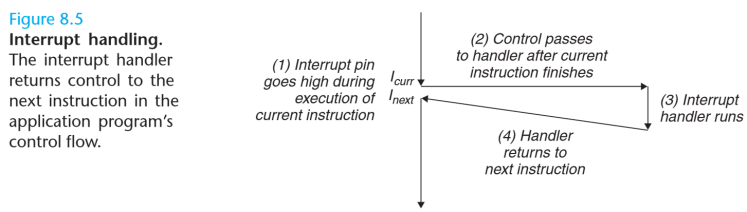
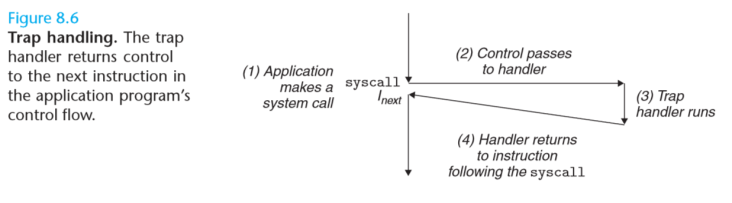
global variance reference needs more instructions, memory accesses, uses extra register 🡪 may cause register spill

* + - lazy binding/linking of shared lib
      * delay loading of obj till needed, GOT entry doesn’t know addr
      * call function in other shared library (doesn’t need to be PIC)
      * implementation:
        + call addr fr entry in procedure linkage table (PLT) (in .text section) instead of absolute addr
        + PLT addr jumps to entry in GOT

***Exceptional Control Flow:***

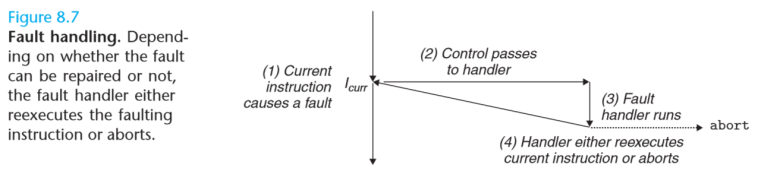
* reaction to abrupt transfer of control (ie jmps/call/ret or smooth in adjacent mem transfers)
* Exceptions:
  + General Routine:
    - abrupt change in control flow in response to change in processor state (aka event)
    - when event detected
      * processor det. exception # and finds on exception table
        + exception table holds addresses for all exception handlers mapped to their exception #
        + exception table base register stores starting addr of exception table
      * indirect procedure call (exception) to OS exception handler for that exception
    - when exception handler done,
      * return control to insn occurring when exception took place
      * return control to next insn
      * abort program

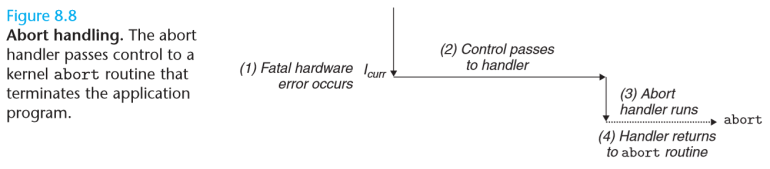


* + Types of Exceptions:
    - Asynchronous (b/c of I/O signal external to processor, not caused by execution of insn)
      * interrupts
    - Synchronous (b/c of a faulting insn)
      * traps
        + intentional exceptions like sys calls requesting services fr kernel/OS
        + sys call runs in kernel mode

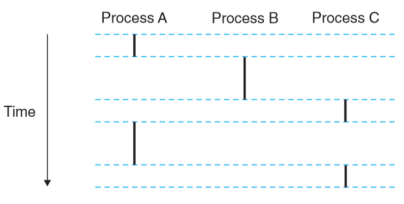
can access any mem location

can exec any insn

* + - * + more state saved
        + ie INT insn
      * faults
        + caused by error conditions handler can correct
        + control transferred to fault handler

if handler fixes, returns control to current insn; else aborts program

* + - * aborts
        + unrecoverable fatal errors
  + Relation to Processes
    - processes give application independent control flow and private address space per process
    - logical control flow = sequence of program counter values corresponding to insns of program
    - multi-process control flow:
      * time slice: portion of time process executing

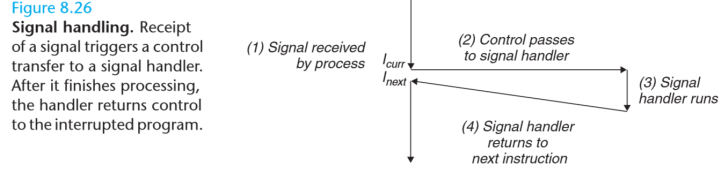
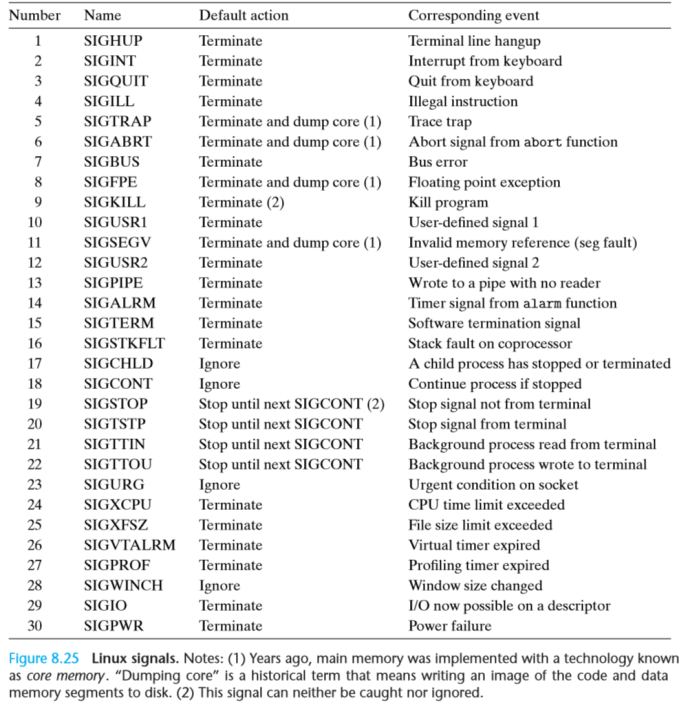


* + - * concurrent flow: multiple logical flows overlapping (run concurrently)
      * parallel flow: 2 flows running on diff cores/processors (run in parallel)
      * context switch:
        + kernel scheduler can decide when to context switch to another process (usually when waiting for something else to finish, use time efficiently and do something else)
        + saves context of current process
        + restores saved context of previous switched process
        + pass control to new process
        + Problems:

can cause race conditions

can negatively affect caching (have data for old process 🡪 cache miss)

overhead

* Signals
  + allow kernel/process to interrupt other processes
  + Sending:
    - kernel sends signal to process b/c detected synchronous event or kill function called
  + Receiving:
    - process can ignore, terminate, or catch signal by exec signal handler
    - can only be at most 1 pending signal 🡪 repeated signals will be discarded

|  |  |
| --- | --- |
| CISC (Complex Insn Set Computer) | RISC (Reduced Insn Set Computer) |
| * less insns but longer cycle/insn time * variable length (common insn short, complex long) * diverse addressing modes * complex insn decoding method * buggier/hard to write compiler | * small set of optimized insn reduce cycles/insn but more insns * fixed length * one addressing mode * simple decode/compilers * faster |