CS 33 UCLA Spring '18 with Professor Glenn Reinmann

INTRODUCTION TO COMPUTER ORGANIZATION

by Siddharth Joshi

TABLE OF CONTENTS:

Bits and Bytes:

Doubts:
-- Negative binary numbers (how does it know the first bit should be interpreted as negative): 1 bit less used in signed integers, hence their range is half the size - DONE
-- Purpose of Logical/Arithmetic Shift: However, as the logical right-shift inserts value 0 bits into the most significant bit, instead of copying the sign bit, it is ideal for unsigned binary numbers, while the arithmetic right-shift is ideal for signed two's complement binary numbers.

Representing information as Bits
- Everything is Bits
-- Each bit is 0 or 1 (high/low voltage)
-- Why bits: bi-stable elements, reliably transmitted on noisy and inaccurate wires

-Encoding Byte Values
-- Byte = 8 bits
--- Binary 00000000 to 11111111 (Base 2)
--- Decimal 0 - 255 (Base 10)
--- Hexadecimal 00 - FF (15, 15) (Base 16)
---- One Hex digit represent 4 binary bits, so each byte can be represented by 2 hex characters
--- Each memory address in memory stores a single Byte
--- In each byte, we can encode 256 unique values.
---Memory dumps are printed out in hex, but really represent binary.

- Example Data Representations
-- Type: 32bit, 64bit, x86-64
-- Char: 1, 1, 1
-- Short: 2, 2, 2
-- Int: 4, 4, 4,
-- Long: 4, 4, 8
-- Float: 4, 4, 4
-- Double: 8, 8, 8
-- Long Double: - - 10 or 16
-- Pointer: 4, 8, 8

Bit level manipulations
- Boolean Algebra: A AND B = A&B, A OR B (A|B), NOT A (~A), XOR (A^B)(Exclusive or that is either A or B but not A and B)
- Boolean algebra ^^ operates on bit vectors:
-- Operation applied bitwise:

```
E.g   01101001
   &  01010101
      01000001
```

- Representing and Manipulating Representations
-- Width w bit vector represents subsets of {0, ..., w - 1}

-- a[j] = 1 if j belongs to A
E.g.:
A
01101001 {0, 3, 5, 6}
76543210
B
01010101 {0, 2, 4, 6}
76543210
--- Operations: & = Intersection (E.g. {0, 6}), | Union, ^ Symmetric Difference, ~ Complement

- **Bit Level Operations** Available in C for all integral data types (view arguments as bit vector).
--Usage: ~0x41 -> 0xBE
--0x69 & 0x55 -> 0x41
--0x69 | 0x55 -> 0x7D

- **Logical Operators** in C:
-- && - and, || - or, !
-- View 0 as False
-- Anything nonzero as "True"
-- Always return 0 or 1
-- Early termination (short circuiting in conditionals)
---Eg. !0x41 -> 0x00; !0x00 -> 0x01; !!0x41 -> 0x01
---Eg. 0x69 && 0x55 -> 0x01

- **Shift Operators**:
-- Left Shift: x << y (moves by y positions)
--- Shift bit-vector x left by y positions:
---- Throw away extra bits on Left
--- Fill with 0s on right (for all types of shifts i.e. logical and arithmetic)

-- Right Shift: x >> y
--- Shift bit vector x right positions bit on Left
---- Throw away bits on Right
--- **Logical Shift**: Fill with 0s on Left
--- **Arithmetic Shift**: Fill with MSB on the left

Undefined Behavior: Shift Amount < 0 or >= word size.

Integers

Representation of Integers
- Encoding Integers (w is length of words in bits)
-- Bits2Unsigned (B2U): Summation from 0 to w - 1 of ith digit into 2 ^ i
-- Bits2TwosComplement (B2T): -MSB * 2 ^ w - 1 + Summation of ith digit into 2 ^ i from 0 to w - 2

Numeric Range
-- Unsigned Values: 0 (all bits are 0) to (2^w) minus 1 (all bits are 1)
-- Two's Complement Values: (-2^(w-1)) (10000000) to (2^(w-1) - 1) (01111111)
--- Two's complement is -1 if all the bits are 1's.
-- Asymmetric Range -> abs(Tmin) = Tmax + 1
-- (Unsigned integer range) Umax = 2 * Tmax + 1
-- C++ (declares constants in limits.h for ULONG_MAX - unsigned long max)
-- One to one mapping from binary to unsigned/ two's complement and vice-a-versa

Conversion, Casting
- Mapping between Signed & Unsigned:
-- Converting from one to the other maintains same bit pattern, just interpreted differently
- Signed vs Unsigned in C
-- Default is signed int, U as suffix for Unsigned
-Casting
-- Explicit - same as U2T and T2U - treating bit pattern in the other one's form (unsigned) or (int). Eg. tx = (int) ux; uy = (unsigned) ty;
-- Implicit - via procedure calls where one is passed to another (tx = ux)
-- Casting Surprise
--- for a mix of unsigned and signed: signed implicitly casted to unsigned when evaluating expressions. (Eg. -1 is implicitly casted to 1)
--- including comparison operators (<, >, ==< <=, >=)

Binary to Hex conversion
-Split it into 4s (from the byte 00111101). Eg. 3B: 0011 -> 3 | 1011 -> B | hence 3B

Expanding and Truncating
- Expanding (always add to the left)
-- Unsigned - copy 0s into new bits
-- Sign Extension for Signed Ints
--- Convert from smaller to larger data type by copying over MSB in all new bits (C Does this automatically)
- Truncating (always removed from the left)
-- Unsigned/Signed: bits are truncated (chopping)
-- Result is then reinterpretted
-- For unsigned it is essentially a modulo operator with the place one beyond the new MSB which is the same as the place of the last digit that was truncated. That is, number%2^(desired length of w)
-- For signed value it is similar to mod but not really due to the negative value

Addition, Negation, Multiplication, Shifting

- Addition

-- Unsigned: True Sum for w bits could require at max w+1 bits, but assembly truncates resulting in modulo with 2^w if overflow (drops the new MSB). -> (u+v%2^w)

--- Mathematical Addition + possible addition or subtraction of 2^w

-- Addition for two's complement is exactly the same, but:

--- Positive overflow becomes negative value

--- Negative overflow becomes positive value

- Multiplication

-- Unsigned: up to 2w bits

--- Result range: 0 to (2^2w) - (2^(w+1)) + 1

E.g. A * B → true product would be 2*w bits

But discard top w bits when overflow exists (same as (u.v%2^w)

-- Two's complement min (negative): upto 2w-1 bits

--- Result range: 0 to -2^(2w-2) + 2^(w-1)

-- Two's complement max (positive): up to 2w bits, but only for (Tminw)^2

--- Result range: 0 to 2^(2w-2)

-- Maintaining exact results, would need to keep expanding word size with each product computed, which is done by software if needed

-- Power of 2-Multiply

--- Multiplication (left shift): u * 2^k = u << k (True product is w+k bits but upper k bits are discarded)

---Eg. (u << 5) - (u << 3) = 32u - 8u = 24u = u * 24

---- Most machines shift and multiply faster than multiply, hence compiler converts if possible

--- Division is opposite i.e. it uses a right logical shift u >> k = (u / 2^k)

- Summary

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

-- sizeOf (var name) returns unsigned int for size

-- size_t -> Variable that is unsigned and may have length = word size

-- Unsigned - useful in modular arithmetic and when using bits to represent sets as logical right shift used - no sign extension

- Representations in memory, pointers, strings

-- Byte-Oriented Memory Organization

--- Programs refer to data by address

---- memory can be thought of as a massive array of bytes where addresses are the indices and pointer vars store these addresses

--- System provides private address space to each "process"

---- a process = program being executed that has its own memory so that it doesn't clash with other processes' memory

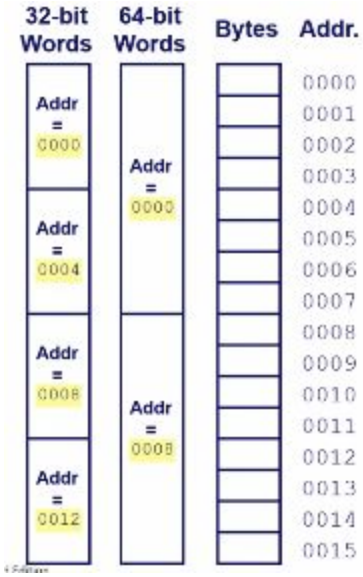
-- Machine Words

--- Every machine has machine words -> Word Size - nominal size of integer-valued data and of **addresses**

---- Most machines used 32 bits i.e. 4 bytes as the word size → machine has at most 2^32 bytes of memory (RAM) because 2^(word size in bit) unique addresses exist

---- Modern machines have 64 bit but use way lesser memory actually

--- Machines can still access fractions or multiples of word size, as long as it's an integral number of bytes

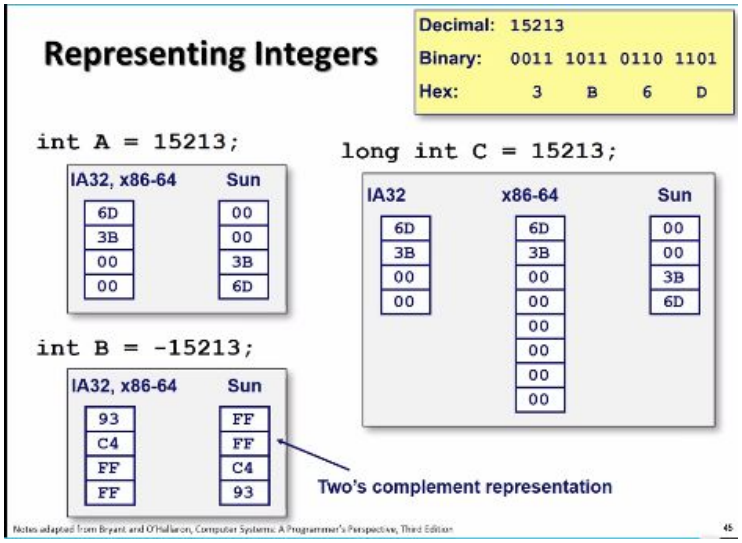


-- Big Endian(SUN, PPC Mac, Internet) - LSB in Word has highest address → read first

-- Little Endian (x86, ARM - for Android, iOS and Windows) - MSB in Word has highest address → read first,

Individual bits inside bytes are not flipped only the whole byte is.

Most Significant Byte and Most Significant Bit - Bit/Byte in the highest magnitude place



-- Examining Data Representations

--- Code to Print Byte Representations of Data (casting pointer to char* → use as array)

Code to Print Byte Representation of Data

Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

Printf directives:

%p: Print pointer

%x: Print Hexadecimal

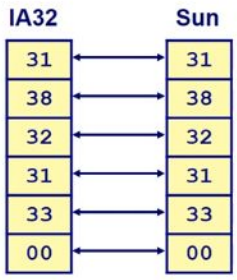
```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

The "Right Shift (Arithmetic)" is useful when used on twos-complement numbers. The sign bit (leftmost bit) is replicated at the high end of the number, instead of bringing in zeroes as with "Right Shift (Logical)". If the number is negative, replicating the sign bit with "Right Shift (Arithmetic)" keeps it negative.

Representing Strings

```
char S[6] = "18213";
```

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character "0" has code 0x30
      - Digit *i* has code 0x30+*i*
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - Byte ordering not an issue



Strings stored the same irrespective of whether the memory is stored in terms of big endian or little endian (because strings are represented as character arrays in C. Last will be 00 since it must contain the null byte.)

Machine Level Programming: Basics

History of Intel processors and architectures
- Complex Instruction Set Computer
-- Many different instructions with many different formats
-- linux encounters only a smallset
-- Hard to match performance of Reduced Instruction Set Computers

- IA32
-- 32 bit version of the x86

C, assembly, machine code
- Definitions
-- Instruction Set Architecture - interface between software and hardware - the parts of a processor design that one needs to understand or write machine/assembly code
  --- Instruction set specs, register
-- Microarchitecture - implementation of the archicture
  --- Caches sizes and core frequency
-- Code Forms:
  -- Machine Code: byte level programs that a processor executes
  -- Assembly code: A text representation of machine code

-- Other ISAs
  --- ARM, Intel: IA32, Itanium

- Assembly/Machine Code View
  -- CPU
  --- Program Counter (RIP in x86 64)
    ---- Address of next instruction
  --- Register file
    ---- Heavily used program data
  --- Condition Codes
    ---- Stores status info about most recent arithmetic or logical operation
    ---- Used for conditional branching
  -- Memory
    ---- Byte addressable data
    ---- Code and user data
    ---- Stack to support procedures (helps scoping rules work in higher level prog languages)

- Turning C into Object Code
  -- text: C program
    processed by compiler (gcc -Og -S)
  -- text: Asm program (p1.s p2.s)
    processed by assembler (gcc or as)
  -- binary: Object program (p1.o p2.o)
    with the linker
  -- binary: executable program (p) <--- static libraries (.a)

- Compiling into Assembly

-- C Code:
long plus (long x, long y);

void sumstore (long x, long y, long *dest)
{
	long t = plus (x, y);
	*dest = t;
}
obtain w command: gcc -Og -S sum.c
produces file sum.s

-- Generated x86-64 Assembly
sum store:
 pushq %rbx
 movq %rdx, %rbx
 call plus
 movq %rax, (%rbx)
 popq %rbx
 ret

- Assembly Characteristics: Data Types
  -- Integer (data of 1,2,4,8)
    --- Data values
    --- Addresses (untyped pointers)
  -- Floating point data of 4, 8, 10 bytes
  -- Code: bytes sequences encoding series of instructions (cisc variable length)
  -- No aggregate types such as arrays or structures
    --- Just contiguosly allocated bytes in memory

- Assembly Characteristics: Operations
  -- Perform arithmetic function on register or memory data
  -- Transfer data between memory and register
    --- Load data from memory into register
    --- Store register data into memory
  -- Transfer control
    --- Unconditional jumps to/from procedures
    --- Conditional branches

- Object Code
  -- Code for sumstore:
  0x0400595:
	0x53
	0x48
	0x89
	0xd3
	0xe8
	0xf2
	0xff
	0xff

0xff
0xff
0x48
0x89
0x03
0x5b
0xc3

-- Assembler:
--- Translates into .s into .o
--- Binary encoding of each instruction
--- Nearly complete image of executable ode
--- Missing linkages between code in different files

-- Linker:
--- Resolvers references between files
--- Combines with static runtime libraries
---- e.g. code for malloc, printf
--- Some libraries are dynamically linked
---- Linking occurs when programs begins execution

- Machine Instruction Example
*dest = t;

movq %rax, (%rbx) - mov instruction means a copy of the data is being provided to the destination
- q stands for quadword in x86-64 - a 8byte value (movq source, dest)
- % (% indicates register name) rax - current register storage for t - register rax stores t
- rbx is a register that stores the pointer to dest
- dereferencing done w parentheses in movqxc

0x40059e: 48 89 03
-- C Code
--- Store value t where designated by dest
-- Assembly
--- Move 8 byte value to memory
---- Quad words in x86
--- Operands:
---- t: register %rax
---- dest: register %rbx
---- *dest: MemoryM[%rbx]
--- Object Code:
---- this instruction is a 3-byte instruction
---- stored at address 0x40059e

- Disassembling Object Code
-- Disassembled
--- objdump -d sum
--- Analyzes bit pattern of series of instructions
--- Produces an approximate rendition of assembly code
--- Can be run on either a.out(complete executable) or .o file
-- Within GDB Debugger

gdb sum
disassemble sumstore
-- Dissemble procedure
--- x/14xb sumstore
---- examining the 14 bytes starting at sumstore

Assembly Basics: Registers, operands, move
- Integer registers (x86-64)
-- the historical 32bit registers refer to the lower 4 bytes of the new 64bit registers, hence you can't store different values in the both of them

- Moving Data
  -- Operand types
    --- Immediate: constant integer data
        ---- $0x400, $-533
        ---- encoded with 1, 2 or 4 bytes
    --- Register (1 of 16)
        ---- %rax, %r13
        ---- %rsp reserved for special use
    --- Memory (8 consecutive bytes of memory at address given by register)
        ---- (%rax)
        ---- various other "address modes"
- movq Operand Combinations

	Source	Dest
movq	imm	reg/mem
	reg	reg/mem
	mem	reg

Simple Memory Addressing Modes
- Normal: (R) Mem[Reg[R]]
  Register R specifies memory address
  (R) - pointer dereferencing

- Displacement: D(R) Mem[Reg[R]+D]
  movq 8(%rbp), %rdx (pointer + 8)

Complete Memory Addressing Modes
- Most general form
D(Rb, Ri, Rs) --> Mem[Reg[Rb]+S*REg[Ri]+D]
D: Constant displacment 1,2 or 4 bytes
Rb: Base register: any of 16 integers registers
Ri: Index register: any except for %rsp - DOUBT WHY RSP
S: Scale - 1,2,4 or 8

Address Computation INstruction
- leaq Src, Dst
Load Effective Address QuadWord
- Src is the address mode expression
- Set Dst to address denoted by expression

so basically you can do an expression in the first part and then this value is loaded into dest
- Uses
  -- Computing addresses without a memory reference
    --- E.g. translation of p = &x[i]
    ---- Leaq doesn't derefeence pointer
    addq dest = dest + src
    subq dest = dest - src
    imulq dest = dest * src
    salq dest = dest << src
    sarq dest = dest >> src Arithmetic
    shrq dest = dest >> src Logical
    xorq dest = dest ^ src
    andq dest = dest & src
    orq dest = dest | src

    one operand instructions
    incq dest dest = dest+1
    decq dest dest = dest-1
    negq dest dest = -dest
    notq dest dest = ~dest
-*- No distinction between signed and unsigned int

Machine Level Programming: Control

DOUBT: what does the cqto command do in assembly?

Control: Condition Codes

- Processor State (x86-64, Partial)
  -- Information about currently executing program
    --- Temp data (%rax)
    --- Location of runtime stack (%rsp)
    --- Location of current code control point (%rip)
    --- status of recent tests (Carry Flag, Zero Flag, Sign Flad, Overflow flag - helps us create other more complex comparisons like greater/lesser)

- Condition Codes (Implicit Setting)
  -- Single bit registers
    --- CF - unsigned
    --- SF - signed
    --- ZF
    --- OF - signed
  -- Set implicitly by arithmetic operations:
    E.g. addq src, dest --> t = a + b
    --- CF if carry out from MSB (unsigned overflow)
    --- ZF if t == 0
    --- SF if t < 0 (as signed)
    --- if two's complement overflow (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
  -- Not set by leaq instruction

- Condition Codes (Explicit Setting)
These are useful because they don't store value in any destination register
  Set by compare insturuction
  cmpq src2, src
  cmp b, a --> like computing a-b without setting destination
  -- Same as above where t = a - b

  Set by testq src2, src1
  testq b, a --> a&b
  -- ZF set when a&b == 0
  -- SF set when a&b < 0

- Reading Condition Codes
  -- Save condition code value in a general purpose with the setx instrucitons
    --- Set low-ordre byte of destination to 0 or 1 based on combinations of condition codes
    sete ZF (equal/0)
    setne ~ZF (not equal/not zero)
    sets SF (negative) DOUBT
    sets ~Sf (non-negative) DOUBT
    setg ~(SF^OF)&~ZF (greater signed)
    setge ~(SF^OF) greater of equal

setl (SF^OF) less signed
setle (SF6OF) | ZF less or equal signed
seta ~CF&~ZF above (unsigned)
setb CF below (unsigned)

-*- x8664 integer registers - can reference low order byte exclusively (for letters remove first letter and addl for numbered ones add b at end) e.g. %rsp - %spl %r13 - %r13b

Usually combined with another function to set the upper 32 bits to 0 (movezbl from lower byte register to larger representation)

```
int gt(long x, long y)
cmpq %rsi, %rdi # Compare x:y
setq %al   #set when >
movezbl %al, %eax # Zero rest of %rax
ret
```

- Jumping (like a goto statement)
jX instructions jump to different parts of code depending on condition codes
jmp 1 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)&~ZF greater(Signed)
jl (SF^OF) less (signed)
jle (SF^OF) | ZF (less or equal (Signed))
ja ~CF&~ZF above
jb CF below

Conditional Branch Example

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

absdiff:
        cmpq %rsi, %rdi #x:y
        jle .L4 (go to label L4)
        movq %rdi, %rax
        subq %rsi, %rax
.L4: (will be replaced later with an actual memory address)
    movq %rsi, %rax
    subq %rdi, %rax
```

General condiitional expression:
val = test > if true_expression : else_expression;

- Conditional move
  -- moves from src to dest only if test passes
  -- avoids the need to actually branch out, gcc tries to use them when known to be save
  -- branches are very disruptive to instruction flow through pipelines (DOUBT)
 C code: val = test ? true_expression : false_expression;

 absdiff:
 movq %rdi, %rax #x
 subq %rsi, %rax # result = x - y
 movq %rsi, %rdx
 subq %rdi, %rdx #eval = y - x
 cmpq %rsi, %rdi
 cmovele %rdx, %rax # if <= result = eval
 ret

 -- Bad cases for conditional move
    val = Test (x) ? Hard1 (x) : Hard2 (x);
    --- Expensive Computations (because both values get computed)

    --- Risky copmutations
    val = p ? *p : 0
          --- Both values get computed may have undesirable effects

    --- Computations with side effects
    val = x > 0 ? x*= 7: x+= 1
          --- x gets altered by both and hence incorrect value given if the x+= 1 is the desired answer

        -- Do/While Loop condiiton
        long pcount (unsigned long x)
        {
                long result = 0;
                do
                {
                        result += x & 0x1
                        x >> 1;
                }
                while (x);
                return result;
        }

        Copy code from slide

        -- For loop
          --- For while conversion
          for (init; test; update)

```
    body
  init;
  while (Test)
        body;
        update
  --- Usually can be optimized with a do while loop avoiding the initial check in assembly

-- Switch statements
long swtich_eg (long x, long y, long z)
{
        long w = 1;
        switch (x)
        {
                case 1:
                w = y*z;
                break;
                case 2:
                w =  y * z;
                /* Fall through */
                case 3:
                w += z;
                break;
                default:
                w = 2;
        }
        return w;
}

Jump Table (like an array of labels)
Targ 0
Targ 1
Targ 2
Targ n - 1

.section        .rodata
        .align 8
.L4:
        .quad .L8 # x = 0
        .quad .L3 # x = 1
        .quad .L5 # x=  5
        .quad .L9 # x = 3
        .quad .L8 # x = 4
        .quad .L7 # x = 5
        .quad .L7 # x = 6

switch_eg:
        movq %rdx, %rcx
        cmpq $6, %rdi # x:6
        ja .L8 #Use Default
```

```
        jmp *.L4 (, %rdi, 8) --> INDIRECT JUMP TO * (JTab[x])
           -- start of jump table: .L4
           -- must scale by 8 (addreses are 8 bytes)
           -- Fetch target from effective addres .L4 + x*8
              -- only for 0 <= x <= 6


.L3:
    moq %rsi, %rax #y
    imulq %rdx %rax #y * z
    ret

 .L5: #case 2
     movq %rsi, %rax
     cqto #DOUBT
     idivq %rcx
     jmp .L6
 .L9:
     movl         $1, %eax # w = 1
 .L6:
     addq %rcx, %rax # w += z
     ret
 .L7: #Case 5, 6
     movl $1, %eax # w = 1
     subq %rdx, %rax # w -= z
     ret
 .L8: #default
     movl $2, %eax
     ret
-*-
Register:
%rdi   argument x
%rsi   argument y
%rdx   argument z
%rax   return value
-*-
```

Machine Procedures

- Mechanisms in procedures
  -- Passing control
    --- To beginning of procedure code
    --- Back to return point
  -- Passing data
    --- Procedure arguments
    --- Return value
  -- Memory Management
    --- Allocate during procedure execution
    --- Deallocate upon return
  -- Mechanisms all implemented with machine instructions
  -- x86-64 implementation of a procedure uses only those mechanisms required

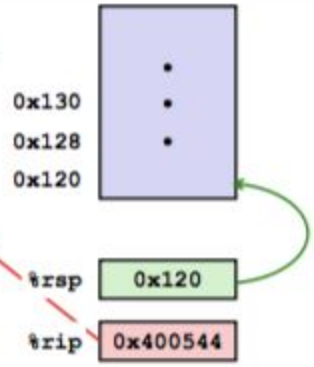
- Stack
  -- Region of memory managed with stack discipline
  -- Grows toward lower addresses
  Stack - Bottom (grows downwards)
  -- Register %rsp contains lowest (top) stack address
  -- pushq src
    --- Fetch operand at src
    --- decrement %rsp by 8
    --- write operand at address given by %rsp
  -- popq dest (must be register)
    --- read value at address given by
    --- increment %rsp by 8
    --- store value at dest

Calling Conventions
- Passing Control
  -- use stack to support procedure call and return
  -- Procedure call: call label
    --- push return address (i.e. next address after call)
    --- jump label
  -- Procedure return: ret
    --- pop address from stack
    --- jump to said address
%rip (register instructional pointer)

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

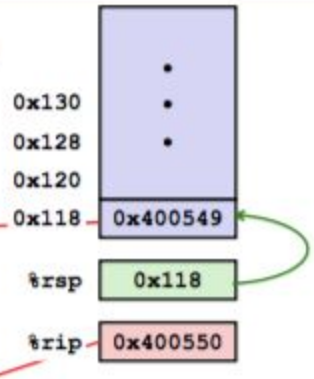
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

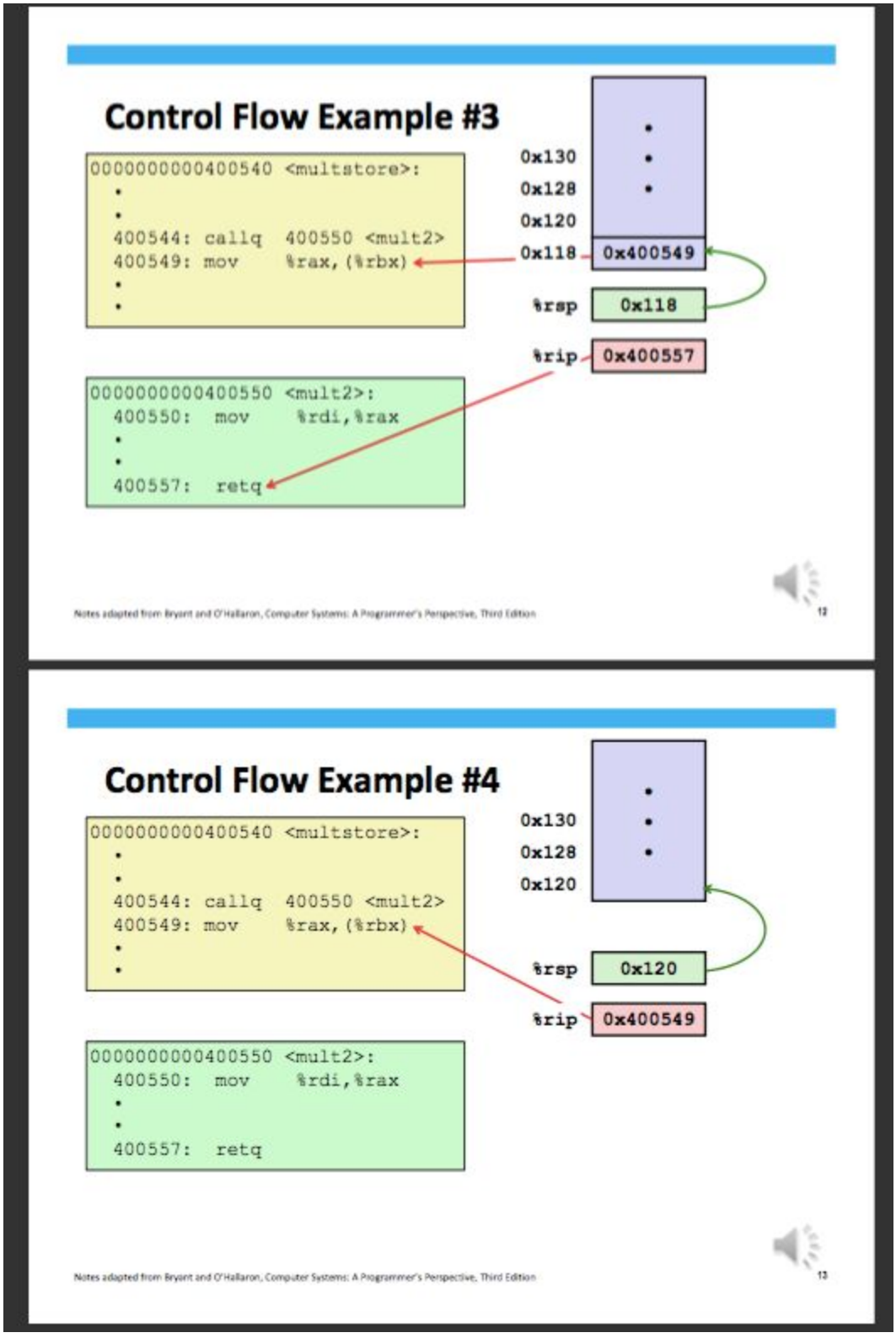


Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



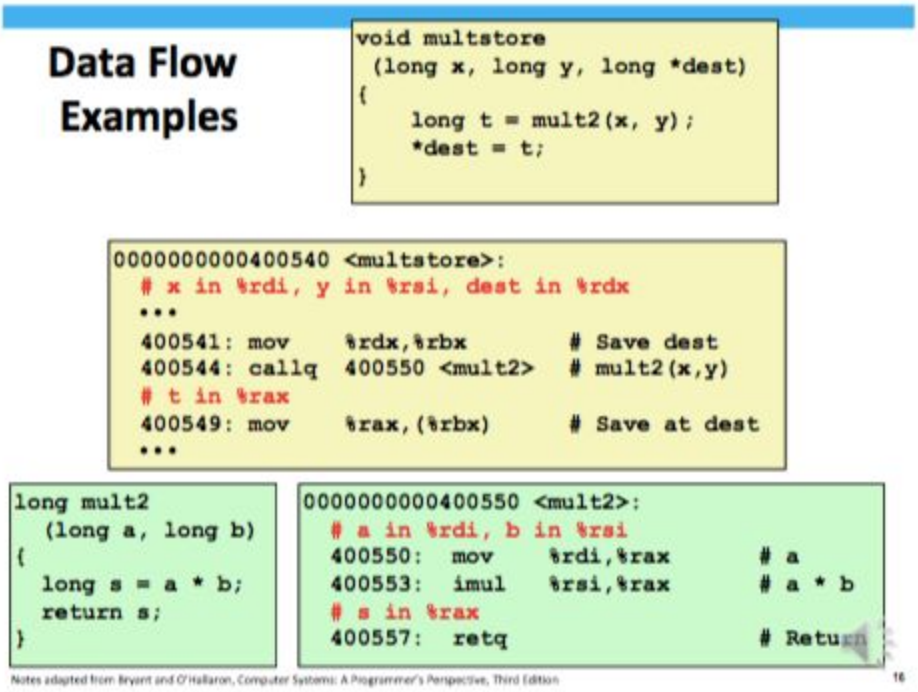


- Passing data

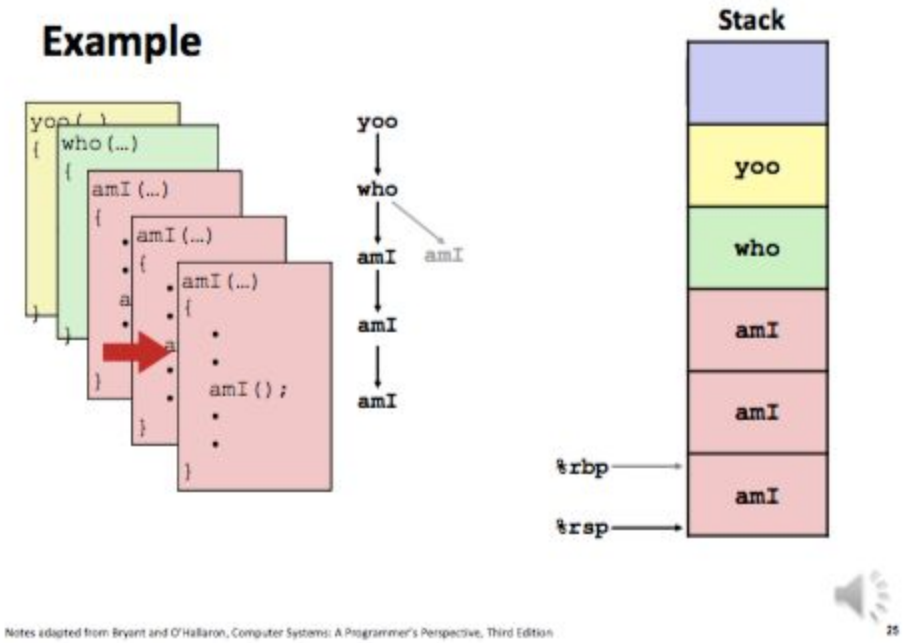
Registers:	Stack
First 6 args	Arg n
%rdi	....
%rsi	Arg 8
%rdx	Arg 7
%rcx	
%r8	
%r9	

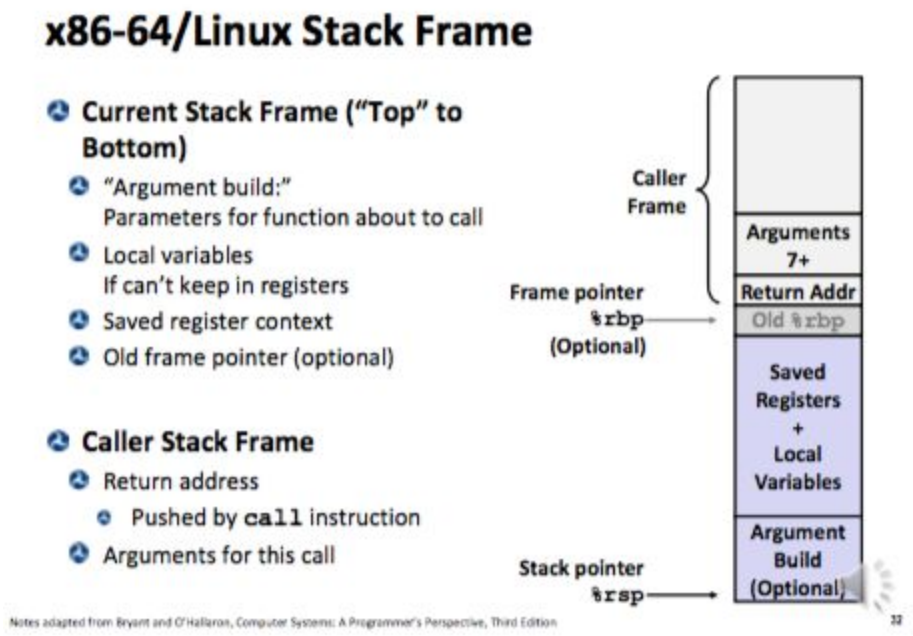
Only allocate stack space when needed

Return value: %rax



- Stack-based languages
  -- Languages that support recursion
    --- C, Pascal, Java
    --- Code must be "reentrant"
        ---- multiple simultaneous instantiations of single procedures
    --- Need some place to store state of each information
        ---- args, local variables, return pointer
  -- Stack Discipline
    --- State for given procedure needed for limited time
        ---- From when called to when return
    --- Callee returns before caller does
  -- Stack allocated in Frames
    --- State for single procedure instantiation
    --- Contents
        ---- Return info
        ---- Local storage
        ---- Temp space
    --- Management
        ---- Space allocated when enter procedure
            ----- Setup code
            ----- Includes push by call instruction
        ---- Deallocated when return
            ----- Finish "code"
            ----- Includes pop by ret function



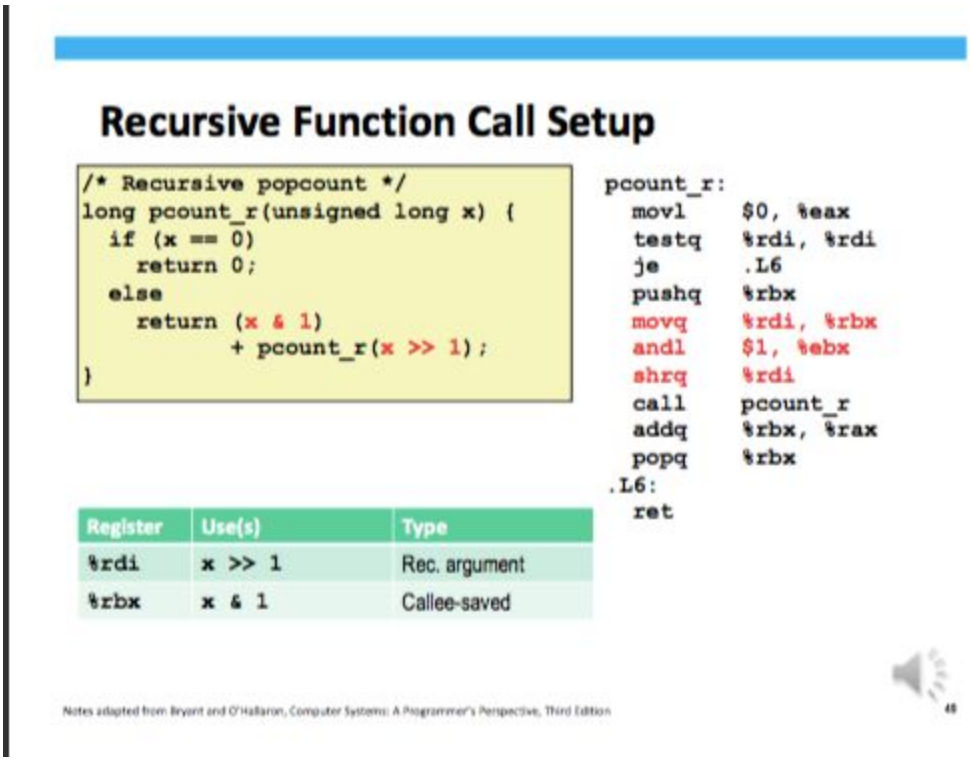


- Register Saving Conventions
  -- When procedure yoo calls who
    --- yoo is the caller
    --- who is the callee
  -- Can register be used for temp storage
    --- Contents of register %rdx may be overwritten by who
    --- Hence contents must be saved (caller saved / callee saved)
    --- Conventions
      ---- Caller Saved - saves temp values in its frame before the call
      ---- Callee saves temp values in its frame before using
      ---- Restores the before returning to caller

x86-64 Linux Register Usage
- %rax
  return value

  also caller-saved
  can be modified by procedure
- %rdi %rsi %rdx %rcx %r8 %r9
  args
  also caller saved
  can be modified by procedure
- %r10, %r11
  caller-saved
  can be mod. by procedure
- %rbx, %r12, %r13, %r14
  callee saved
  callee must save and restore
- %rbp
  callee saved
  callee must save & restore
  may be used a frame pointer
  can mix & match
- %rsp
  special form of callee save
  restored to original value upon exit from procedure

- Recursion (procedure call illustration)



- Summary
-- Stack is the right data structure for procedure call/return because if P calls Q, Q returns before P
-- Recursion
   --- result return in %rax
   --- put func args at top of stack
   --- can safely store vals in local stack frame and in callee-saved registers
-- Pointers are addresses of values

Machine Data

- Arrays
  -- Elements packed into contiguous region of memory
  -- Use index arithmetic to locate individual elements
- Structures
  -- Elements packed into single region of memory
  -- Access using offsets determined by compiler
  -- Possible require internal and external padding to ensure alignment
- Combinations - can nest structures and array code arbitrarily

Arrays
T A[L]

- Arrays Allocation
  -- Basic Principle
    --- Array of data type T and length L
    --- Contiguously allocated reigion of L * sizeOf(T) bytes in memory

E.g. char string[12] x x+1 x+2 ..... x+12
    int val[5] x x+4 x+8 ..... x+20
    char *p[3] x 0   x+8   1 x+16 2   x+24

- Array Access
  -- Basic Principle
    --- Identifier A can be used as a pointer to array element 0: Type T*
    --- Array is contiguous and hence offsetting by an amount gives successive elements
        A+1 --> points to element at position 1 in the array

Example
#define ZLEN 5
typedef int zip_dig[ZLEN] --> zip_dig is now a type of its own (it refers to an integer array of length 5)
zip_dig cmu = {1, 5, 2, 1, 3}
zip_dig mit = {0, 2, 1, 3, 9}
zip_dig ucb = {2, 2, 2, 2, 2}
-- arrays may or may not be contiguous relative to each other

```
int get_digit (zip_dig z, int digit)
{
        return z[digit];
}
```

x86-64
%rdi = z
%rsi = digit

movl (%rdi, %rsi, 4), %eax #z[digit]
Using memory reference (style of arithemetic)
%rdi + offset i.e. %rsi * 4 --> 4 because int has size = 4 bytes

Array Loop Example

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax          #   i = 0
  jmp     .L3               #   goto middle
.L4:                        # loop:
  addl    $1, (%rdi,%rax,4) #   z[i]++
  addq    $1, %rax          #   i++
.L3:                        # middle
  cmpq    $4, %rax          #   i:4
  jbe     .L4               #   if <=, goto loop
  ret
```

- Multidimensional (Nested) Arrays
  -- Declaration
    - T A[R][C]
    - 2D array of data type T
    - R rows, C cols
    Let type T require K bytes
  -- Array Size = R*C*K bytes
  -- Arrangement
    --- Row Major Ordering
    int A[R][C] ---->
    A[0][0] ... A[0][C-1] A[1][0] ... A[1][C-1] ... A[R-1][0] ... A[R-1][C-1]
    ---------- 4 * R * C bytes ----------
    Rows are contiguous, and successive rows are contiguos relative to each other

- Nested Row Access
  -- Row vectors
    -- A[i] os an array of C elements
    -- Each element of type T requires K bytes
    -- Starting address of A[i] = A + i * (C * K)
  -- Nested Array Row Access Code

```
int* get_pgh_zip (int index)
{
        return pgh[index]
}

# %rdi = index
# leaq (%rdi, %rdi, 4), %rax #5 * index
# pgh(, %rax, 4), %rax       # pgh (20 * index) --> stored in %rax
```

- Nested Array Element Access

-- Array Elements
-- A[i][j] is the element of type T which requires K bytees
-- Address = A + i * (C * K) + j * K = A + (i * c + j) * k

```
int  get_pgh_digit (int index, int dig)
{
      return pgh[index][dig];
}
```

%rdi = index
%rsi = dig
leaq (%rdi, %rdi, 4), %rax --> %rax = index * 5 --> 5 = num of cols
addl %rax, %rsi --> %rsi = dig + index * 5
movl pgh(, %rsi, 4), %eax--> %eax = pgh + %rsi * 4 = pgh + 20 * index + 4 * dig

- Multi-level Array

The first level is an array of pointers to the first elements of arrays
int *univ[UCOUNT] = {mit, cmu, ucb}
Since each element is a pointer: 8 bytes
univ: 160 [36] 168 [16] 176 [56]
cmu: 16 [3] 20 [5] 24 [9] ...
mit: 36 [0] 40 [1] 44 [4] ...
ucb: 56 [7] 60 [0] 64 [2] ...

Two memory lookups to get one particular element v/s nested array's one memory lookup

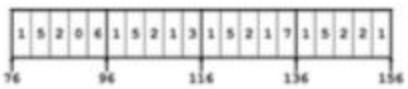
```
int get_univ_digit (size_t index, size_t digit)
{
        %rdi = index
        %rsi = digit
        salq $2, %rsi                   # 4 * digit
        addq univ(, %rdi,8), %rsi       # p = univ[index] + 4 * digit
        movl (%rsi), %eax               # %eax = *p
}
```

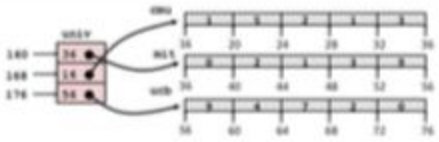
Nested array

```
int get_pgh_digit
  (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```

Multi-level array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```





Accesses looks similar in C, but address computations very different:

```
Mem[pgh+20*index+4*digit]   Mem[Mem[univ+8*index]+4*digit]
```

- N*N Matrix Code

-- Static (Fixed Dimensions)

-- Know value of N at compile time
--- Difficult or maybe impossible to use shifts instead of multiplication for compiler

```
#define N 16
typedef int fix_matrix [N][N]
int fix_ele(fix_matrix a, size_t i, size_t j)
{
    return a[i][j];
}
```

-- Variable dimensions

```
#define IDX(n, i, j) ((i) * (n) + j))
int vec_ele (size_t n, int* a, size_t i, size_t j)
{
    return a[IDX(n, i, j)];
}
```

-- Variable dimensions, implicit indexing (now supported by gcc)

```
int var_ele (size_t n, int a[n][n], size_t i, size_t j)
{
    return a[i][j];
}
```

-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x

Structures

Way of allocating different discreet types in contiguous memory

- Structures Representation

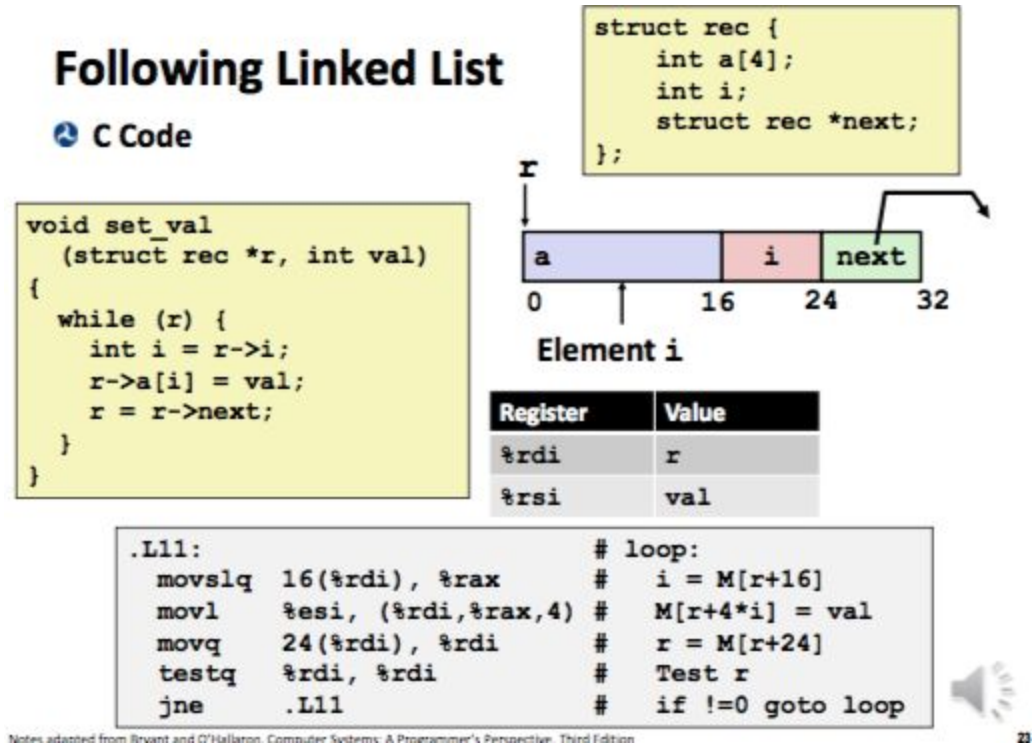
```
struct rec
{
        int a[4];
        size_t i;
        struct rec *next;
}
```

-- Structure represented as a block of memory
-- Atleast big enough to hold all fields (sometimes bigger due to alignment)
-- Fields ordered according to declaration
-- Regardless of optimality
-- Compiler determines overall size + positions of fields
-- Assembly doesn't understand structs

- Generating pointer to Structure Member

r 0[a]   16[i]   24[next]32

-- Generating Pointer to Array Element
-- Offset of each structure member determined at compiler time
-- compute as r + 4 * index



movslq --> mov sign extension 32 bit quantity to 64 bit quantity
M - memory (treating memory as an array)
testq %rdi, %rdi tests if %rdi is 0
jne .L11 jumps if not equal to 0

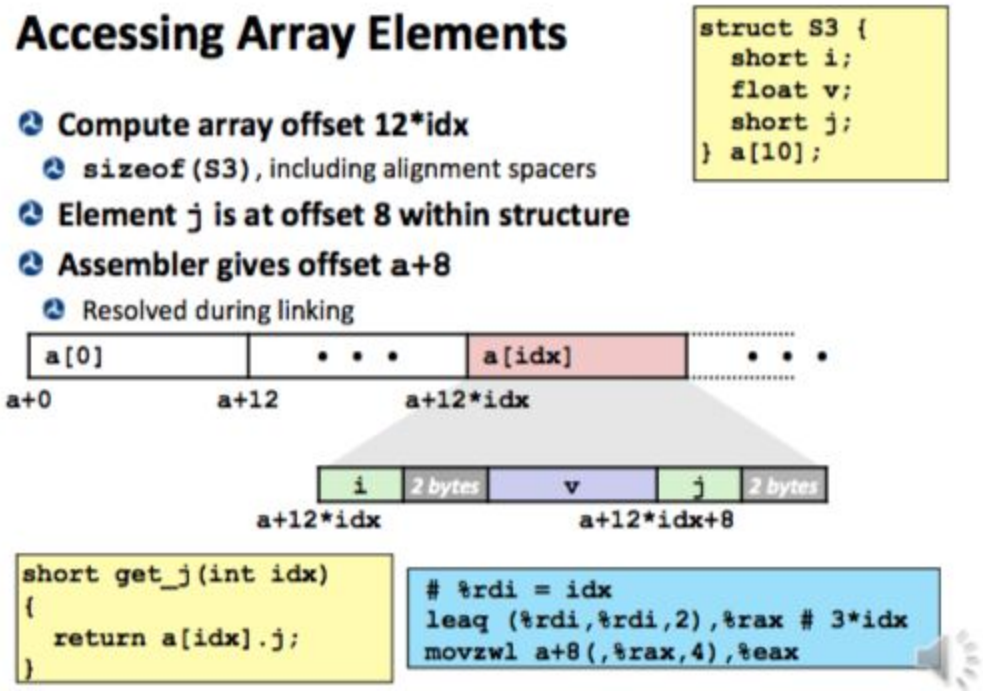
- Structures and Alignment
-- Alignment Principles
  -- Primitve Data Type requires K bytes
  -- Address must be multiple of K
  char - placed anywhere
  short - 2 bytes - must be placed in even numbered address - hence in binary last digit must be 0
  int - 4 bytes - must be placed in multiple of 4 address - hence in binary last 2 digits must be 00
  long - 8 bytes - must be placed in multiple of 8 address - hence in binary last 3 digits must be 000
  double, char * etc. --- same as above ---
  long double - 16 bytes - must be placed in multiple of 16 address - hence in binary last 4 digits must be 0000
  -- Required on some machines, recommended on x86-64
-- Motivation for Aligning Data
  -- Memory accessed by (algined) chunks of 4 or 8 bytes (system dependent)
    -- Inefficient to load or store datum that spans quad word boundaries
    -- Virtual memory trickier when datum spans 2 'pages'
  -- Compiler
    -- Inserts gaps in structure to ensure correct alignment of fields
- Satisfying Alignment with Structures
 -- Within structure - must satisfy eeach elements' reqs
 -- Overall structure placement - each struct has alignment requirement K = Largest alignment of any element
   -- Initial address & structure length must be multiples of K

```
struct S1
{
        char c;
```

```
        int i[2];
        double v;
} *p;
```

paste unaligned Slide 24
paste aligned Slide 24
p+0 --> must be multiple of 8 (overall strucutre alignment)

- Arrays of Structures
  -- Overall structure length - multiple of K
  -- Satisfy alignment requirement for every element



-- Accessing array elements
  -- Compute array offset 12*idx
    -- sizeOf(S3) --> including alignment spacers
  -- Element j is at offset 8 within structure
  -- Assembler gives offfset a+8
    -- Resolved during linking

- Saving Space lost due to alignment
  -- Put large data types first

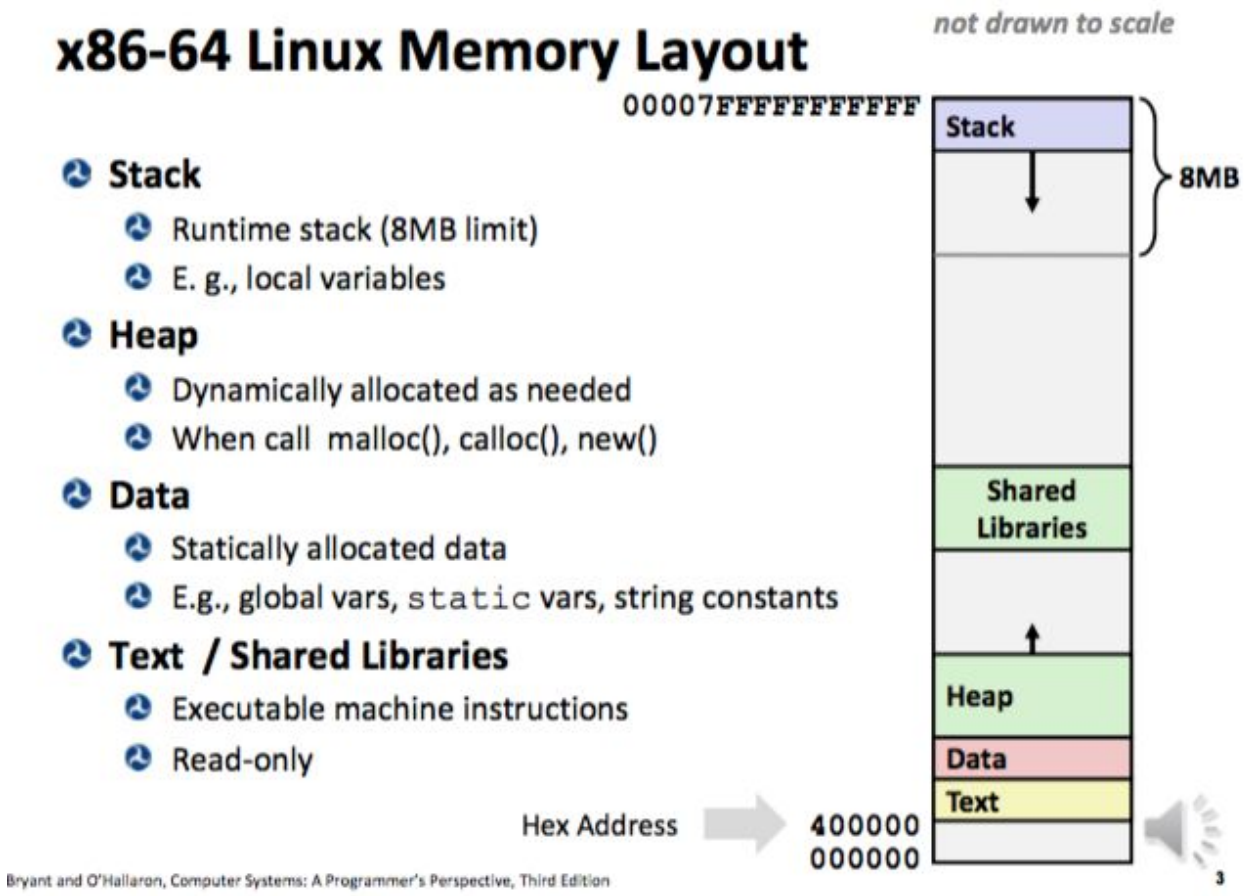
```
   struct s4
   {
        char c;
        int i;
        char d;
   }
```

c 3 bytes padding i d 3 bytes padding

```
   ...
   int i;    char c;  char d;    | c d 2 bytes padding
```

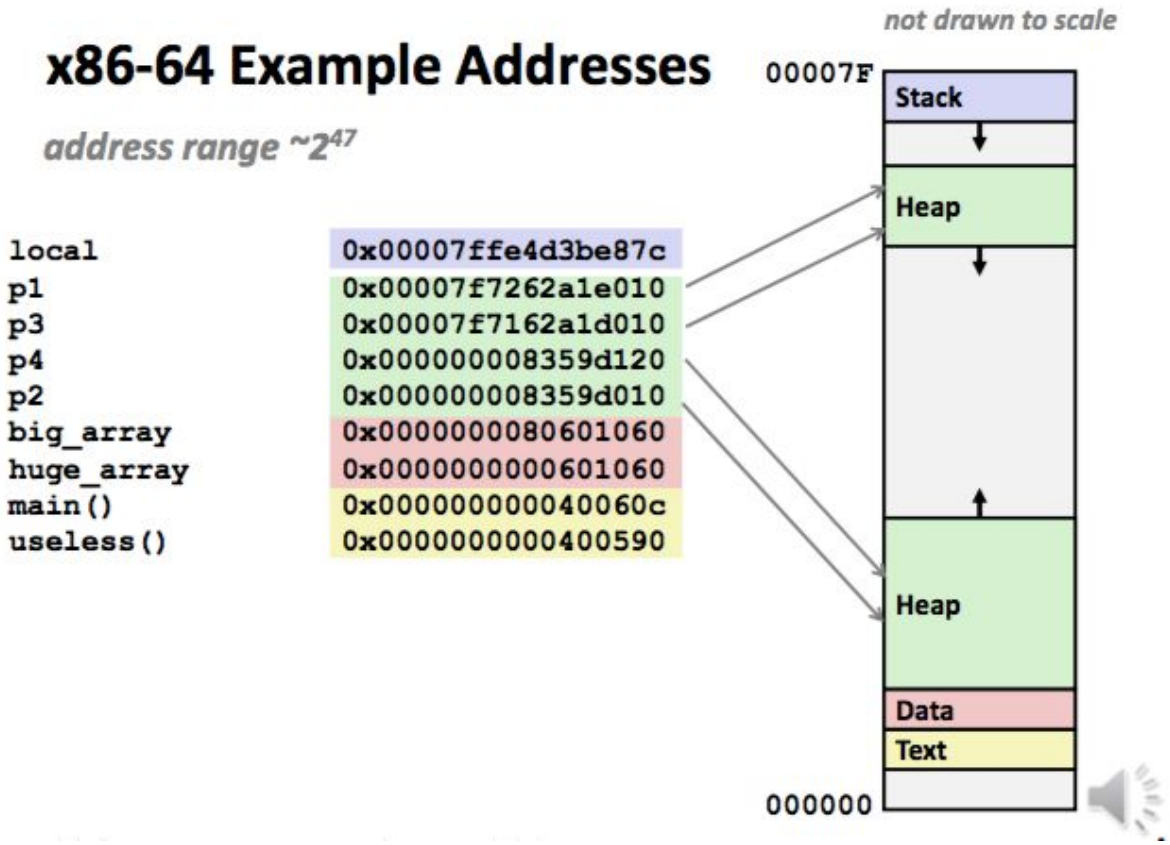
Machine Level Programming: Advanced

x86-64 Linux Memory Layout



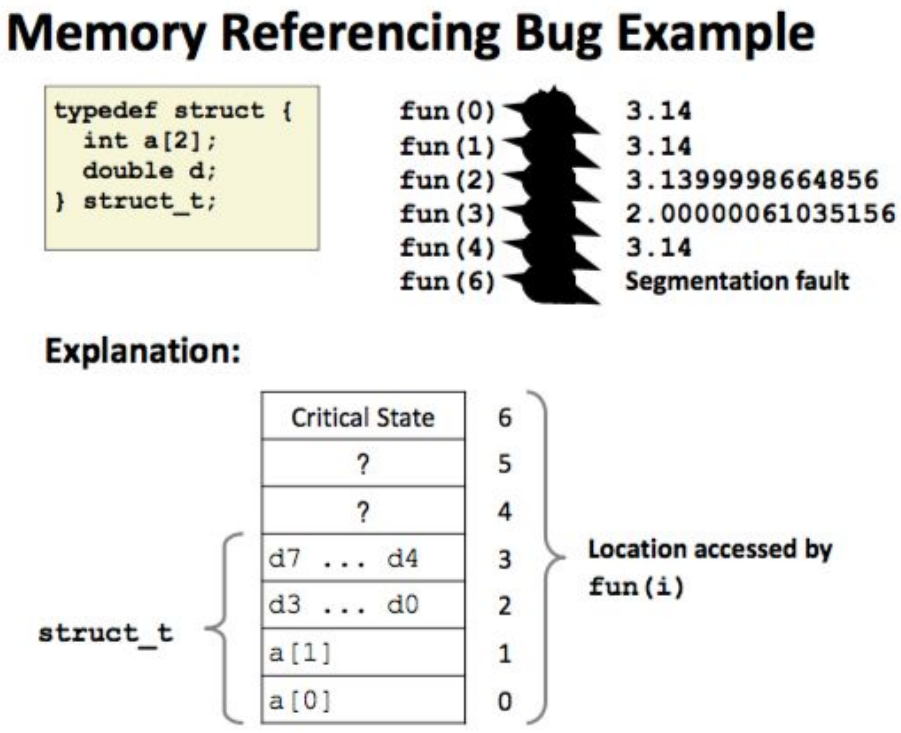
- Memory Allocation Example

```
char big_array[1L<<24]; /* 16MB Explain Syntax --> 1L is 1 in type long*/
char huge_array[1L<<31]; /* 2GB */

int global = 0;

int useless {return 0;}

int main()
{
        void *p1, *p2, *p3, *p4;
        int local = 0;
        p1 = malloc(1L << 28); /* 256 MB */
        p2 = malloc(1L << 8); /* 256 B */
        p3 = malloc (1L << 32); /* 4 GB */
        p4 = malloc(1L << 8) /* 256 B */
        /* Some print statements */
}
```



DOUBT: why does some heap data go to top and other to bottom?

- Buffer Overflow



Segmentation fault - accessing a memory region that you're not supposed to be accessing

- Problems are a Big DEAL
  -- Called Buffer Overflow
  -- #1 technical cause of security vulnerabilities (#1 caue is social engineering/ user ignorance)

-- Most common form
  -- Unchecked lengths of string inputs
  -- Particularly for bounded character arrays on the stack (a.k.a. stack smashing)

String Library Code

- **Implementation of Unix function `gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - No way to specify limit on number of characters to read
- **Similar problems with other library functions**
  - **`strcpy`, `strcat`**: Copy strings of arbitrary length
  - **`scanf`, `fscanf`, `sscanf`,** when given **`%s`** conversion specification

st and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 10

- Vulnerable Buffer Code

```
void echo
{
        char buf[4];    /* Too small */
        gets[buf];
        puts[buf];
}

void call_echo()
{
        echo;
}
```

for some strings larger it may work, for others it will give segmentation fault or other errors

Buffer Overflow Disassembly

echo:

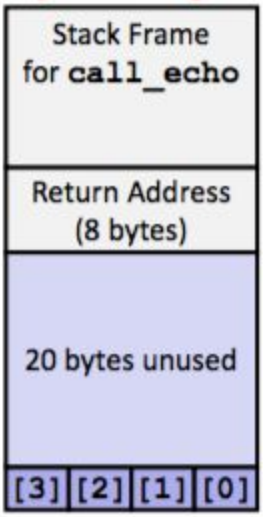
```
00000000004006cf <echo>:
  4006cf:   48 83 ec 18           sub    $0x18,%rsp
  4006d3:   48 89 e7              mov    %rsp,%rdi
  4006d6:   e8 a5 ff ff ff        callq  400680 <gets>
  4006db:   48 89 e7              mov    %rsp,%rdi
  4006de:   e8 3d fe ff ff        callq  400520 <puts@plt>
  4006e3:   48 83 c4 18           add    $0x18,%rsp
  4006e7:   c3                    retq
```

call_echo:

```
  4006e8:   48 83 ec 08           sub    $0x8,%rsp
  4006ec:   b8 00 00 00 00        mov    $0x0,%eax
  4006f1:   e8 d9 ff ff ff        callq  4006cf <echo>
  4006f6:   48 83 c4 08           add    $0x8,%rsp
  4006fa:   c3                    retq
```

Buffer Overflow Stack

Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
   subq  $24, %rsp
   movq  %rsp, %rdi
   call  gets
   . . .
```

Buffer Overflow Stack Example #1

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 2
0123456789012345678901 2
```

Overflowed buffer, but did not corrupt state

-- if overflow is more than though, it will courrupt return pointer
-- this may or may not lead to an error (Segmentation Fault possibly but program may also work)
-- Program works - when it returns to another address which is unrelated code but those garbage operations luckily don't
modify the critical state, eventually executes retq to get back to main

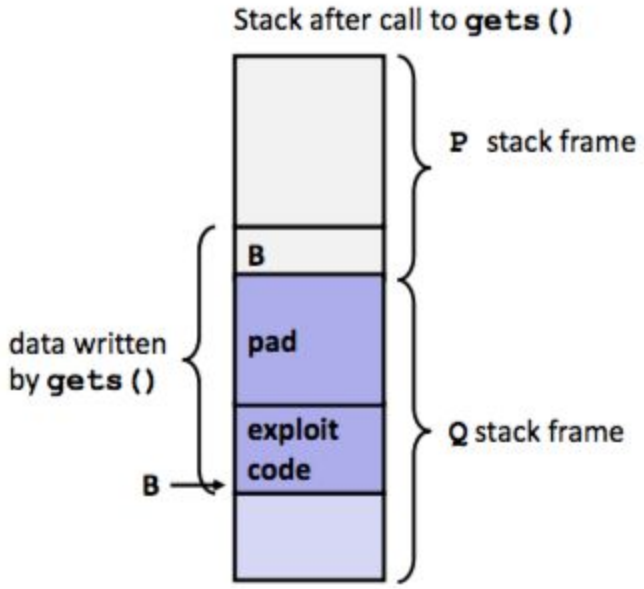
- Code Injection Attacks
-- exploit code - byte sequence of assembly code
-- Input string contains ^ exploit code
-- Overwhite return address A with address of buffer B
- When Q executes ret, will jump to exploit code

Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- **Input string contains byte representation of executable code**
- **Overwrite return address A with address of buffer B**
- **When Q executes `ret`, will jump to exploit code**

Return address stored at end of where buffer is stored, hence when q returns, it looks for return address where address of B is given

- Exploits based on Buffer Overflows
  -- Buffer Overflow bugs can allow remote machines to execute arbitrary code on victim machienes

E.g. The original internet worm (1988)

- Exploted a few vulnerabilities to spread
  -- finger droh@cs.cmu edu --> finger "exploit-code padding new-return address"
  -- Exploit code: executed a root shell on vitim machine with a direct TCP connection to the attacker

- How to handle buffer ovreflow attacks
  -- Avoid overflow vulnerabilities
  use library routines that liit string lengths (fgets instead of gets, strncopy instead of strcopy)
- System Level Protectoins can help
  -- Randomized stack offsets
    -- At start allocate random of space on stack
    -- Shifts stack addresses for entire program
    -- Makes it difficult for hacker to predict beginning of inserted code
    E.g. Stack reposition every execution
  -- Non-executable code segments
    -- In traditional x86 can mark region of space as read only or writeable
  -- Stack Canary
    -- Place special value "canary" on stack just beyong buffer
    -- Check for corruption before exiting function
    -- GCC implementation
      -- fstack-protector

Protected Buffer Disassembly

echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

-- first move moves canary from segment register (offset by 0x28) to rax
-- second move moves that to stack
-- third move moves it back to rax
-- xOr comparison checks if the values are same
-- if not jump not made, and callq to stack_chk fail will return error stack smashing detected

- Return Oriented Programming Attacks
  -- Challenge (for hackers)
    -- Stack randomization makes it hard to predict buffer location
    -- Marking stack non-executable makes it hard to insert binary code

  -- Alternate Strategy
    -- Use existing code (library code from stdlib)
    -- String together fragments to achieve overall desired outcome
    -- Doesn't overcome stack canaries

  -- Construct program from gadgets (this type of code)
    -- Sequence of intructions ending in ret
      -- Code pos fixed from run to run
      -- Code is executable

E.g.
Gadget Example 1
Use tail end of existing functions

```
long ab_plus_c (long a, long b, long c)
{
        return a*b + c;
}
```

0x4004d0 <ab_plus_c>
        0x4004d0: 48 0f af fe imilt %rsi %rdi

0x4004d4: 48 8d 04 17 lea(%rdi, %rdx, 1), %rax
0x4004d8: c3 retq
rax <-- rdi + rdx
gadget address = 0x4004d4

Gadget Example 2
Repurpose bite code

```
void setval (unsigned *p)
{
        *p = 3347663060u
}
```

<setval>
0x4004d9 c7 07 d4 |48 89 c7 movl $0xc78948d4
0x4004df c3| retq
48 89 c7 c3 encodes movq %rax, %rdi (rdi <-- rax_
Gadget Address = 0x4004dc

-- ROP Execution
- Trigger with ret instruction
-- will start executing gadget 1
-- Final ret ine ach gadget will start new one
on the stack they'll write pointers to the gadgets

- Union allocation
-- Allocate according to largest element
-- Can only use of field at any time
-- Different from struct where each field would be allocated memory

Floating Point

- Background: Fractional Binary Numbers
  -- What is 1011.101 = 11.625
    -- Bits to right of binary point represent fractional powers of 2
    -- Represents rational number
  -- Limitations:
    -- Can only exactly represent numbers of the form x/2^k
    -- Just one setting of binary point within the w bits
      -- Limited range of numbers (very small values? very large?)

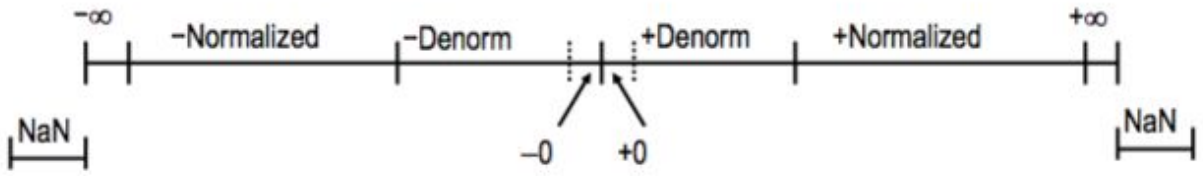
- IEEE floating point standard: Definition
  -- No real limit range
  -- IEEE - org that deals with defining computing arithmetic standards
  -- Floating Point Representation
    -- Scientific numbers sign significant portion and exponent: (-1)^S * M * 2^E
    -- Encoding:
      -- MSB S is sign bit s
      -- Exp field encodes E (not equal to E)
      -- frac field encords M (not equal to M)
      - | s | exp - not E but encoding of E | frac - not M but encoding of M |
  -- Precision Options
    -- Single precision 32 bits
      -- s | exp -- 8 bits | frac -- 23 bits
    -- Double precision 64 bits
      -- s | exp - 11 bits | frac - 12 bits
  -- Normalized Values
    -- When exp != 000.. or 111..
    -- Exponent codes as biased value: E = Exp - Bias
      -- Exp: unsigned value of exp field
      -- Bias: 2^(k-1) - 1 where k is the number of exponent bits
        -- single precision = 127
        -- double precision = 1023
        -- Large value for positive, small value for negative
    -- Significant codes with implied leading 1 = 1.xxx ... x2
      -- xxx ... xxx - bits of frac field
      -- minimum when frac = 000 (M = 1.0)
      -- Maximum when fract = 111 (2.0 - Epsilon)
      -- get extra leading bit for free
    -- Normalized Encoding Example
      -- Value: 15123.0 = 11101101101101 = 11101101101101 * 2 ^ 13
      -- Significand: M = 1.1101101101101 --> frac = 1101101101101
      -- Exponent E = 13, Bias = 127, Exp = 140 (10001100)
  -- Denormalized Value (Exponent case exp = 000....0)
    -- Exponent Value = 1 - Bias (instead of E = 0 - Bias)
    -- Significant coded with implied leading -: M = 0.xxxx
      -- xxxxx - fracts of bits
    -- Cases
      -- exp = 000...0, frac = 000...0
        -- represents zero value

-- note distinct values +0 and -0
-- exp = 000..000, frac != 000..0
-- Numbers closest to 0
-- Equispaced
-- allows for representation of very small values, while maintaining distinction between them

-- Special values
-- Case exp = 1111, fract = 0000 --> represents infinity
-- both positive and negative
-- instead of wrapping around like ints if overflow is to occur, it saturates with 111s
E.g. 1/0/0.0 = 1.0/-0.0infinity 1.0/-0.0 = -infinity

-- Case exp = 111..11, frac != 000
-- NOT A NUMBER (NAN)
-- Represents case wehn no numeric value can be determined
-- E.g. sqrt(-1) etc.

Visualization: Floating Point Encodings



- Tiny Floating point example
-- 8 bit floating point representation (1bit s | exp 4 bits | frac 3 bits |)
-- bias = 7
-- Same general form as IEEE format

Dynamic Range (Positive Only)

$v = (-1)^s M 2^E$

n: E = Exp – Bias

d: E = 1 – Bias

	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
Normalized numbers	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

- Distribution of Values
  -- 6 bit IEEE-like format
    -- e = 3 exponent bits
    -- f = 2 fraction bits
    -- bias = 3
  -- Density gets more as you're closer to 0

- Special Properties of the IEEE Encoding
  -- FP +ve Zero same as integer 0 -- all bits = 0
  -- Can almost use unsigned int comparison
    -- Must first compare sign bits
    -- Must consider -0 = 0
    -- NaN will be problematic as always greater than any other values
    -- Otherwise OK
      -- Denormalized vs Normalized - works denormalized < normalized
      -- Normalized vs Infinity - works normalized < infinity

- Rounding, addition, multiplication
  -- Don't really have to worry about overflow - because it overflows correctly till infinity
    -- But worry about loss of precision
    -- x + y = Round (x + y) or x * y = Round (x * y)
    -- Basic Idea
      -- compute result
      -- make it fit into desired precision
      -- possibly overflow if exponent too large, possibly round to fit into frac

- Rounding
  - Towards Zero

- Round down (- infinity)
- Round up (+ infinity)
- Nearest Even (default) - Regular rounding for numbers not at halfway mark, only round to nearest even at halfway point
  -- Default rounding mode
    -- All others are statistically biased
      -- Sum of set of positive numbers will consistently by over or underestimated
    -- Hard to get any other kind without dropping into assembly???
  -- Applying to other decimal places / bit positions
    -- When exactly halfway, round so that least sig digit is even

Rounding Binary Numbers

- **Binary Fractional Numbers**
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100\ldots_2$

- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

- Floating Point Multiplication
  -- (-1)^s1 * M1 * 2^(e1) * (-1)^s2 * m2 * 2^(e2)
  -- exact result: (-1)^s * M * 2^E
    -- sign s: s1 ^ s2
    -- significand M: m1 * m2
    -- exponent E: e1 + e2
  -- Fixing
    -- if M >= 2 shift M right increment E
    -- if E out of range, overflow
    -- Round M to frac position
  -- Implementation: multiplying significands is most expensive
- Floating point addition
  -- (-1)^s1 * M1 * 2^(e1) + (-1)^s2 * m2 * 2^(e2)
   -- Assume E1 > E2
  -- get binary points lined up
  paste part of slide 28 here

  -- Fixing

-- If M >= 2, shift M right, increment E
-- if M < 1, shift M left k positions, decrement E by k
-- overflow if E out of range
-- Round M to fit frac precision

- Floating Point in C
  -- C guarantees two levels
    -- float - single precision (32 bits)
    -- double -- double precision (64 bits)
  -- Casting between int, float and double changes bit representation
      -- double/float --> int
        -- truncates fractional part
          -- like rounding toward zero
          -- not defined when out of range or NaN (Generally set to Tmin)
      -- Int -> double - exact conversion as long as int has <= 53 bit word size
      -- int -> float - will round according to rounding mode (Default is round to even)
  -- Floating Point puzzles
    -- x == (int) (float) x -> not always true as sometimes loss of precision occurs from int to float
    -- x == (int)(double) x --> always true
    -- f == (float) (double) f --> always true as double is simply more precise that float
    -- d == (double)(float) d --> not always true loss of precision and smaller range, converting from float to double
    -- f == -(-f) --> always true as sign bit controls sign
    -- 2/3 == 2/3.0 --> never true as R.H.S. casts to floating point division and hence gives a fractional value while int division rounds to 0
    -- d < 0.0 implies (d*2) < 0.0 --> always true as even if overflow it saturates to -ve infinity
    -- d > f implies -f > d --> always true as no asymmetry like for ints
    -- d * d >= 0.0 - no overflow hence always true
    -- (d+f) - d == f --> not always true, if f is really small and d is really big, then f may be rounded away in the d + f step and hence when d is subtracted value would not account for f's contribution

    -- Floating point arithmetic violates associativity/distributivity

Optimization

- Overview
  - -- Performance Realities
    - -- Constant factors matter to:
      - -- Easily see 10:1 performance range depending on how code is written
      - -- Must optimize at multiple levels:
        - -- Algorithm, data representations, procedures and loops
    - -- Must understand system to optimize performance
      - -- How programs are compiled and executed
      - -- How modern processors + memory systems operate
      - -- How to measure program performance and identify bottlenecks
      - -- How to improve performance without destroying code modularity and generality
  - -- Optimizing Compilers
    - -- Provide efficient mapping of program to machine
      - -- register allocation
      - -- code selection and ordering (scheduling)
      - -- dead code elimination
      - -- eliminating minor inefficiencies
    - -- Don't (usually) improve asymptotic efficiency
      - -- up to programmer to select best overall algorithm
      - -- big-O savings are (often) more important constant factors
        - -- but constant factors matter too
    - -- Have difficulty overcoming "optimization blockers"
      - -- potential memory aliasing
      - -- potential procedures side effects
  - -- Limitations of Optimizing Compilers
    - -- Operate under fundamental constraint
      - -- Must not cause any change in program behavior
        - -- except, possibly when program making use of non-standard language features
      - -- often prevents it from making optimization that would only affect behavior,
        under pathological conditions (i.e. super weird conditions)
    - -- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
      - -- Data ranges can be more limited than variable types suggest
    - -- Most analysis is performed only within procedures
      - -- whole program analysis is too expensive in most cases
      - -- newer versions of GCC do interprocedural analysis within individual files
        - -- but not between code in different files
    - -- Most analysis is based only on static information
      - -- Compiler has difficult anticipating run-time inputs

    WHEN IN DOUBT, COMPILER HAS TO BE CONSERVATIVE

- Generally Useful Optimizations (should be done regardless of processor/compiler)
  - -- code motion (compiler can usually do)
    - -- reduce frequency with which computation is performed if result is always same
    - -- esp in loops (do a computation in loop that could be done outside)
  - -- reduction in strength
    - -- replace costly operation with simpler one
    - -- shift, add instead of multiply or divide
    - -- 16 * x --> x << 4

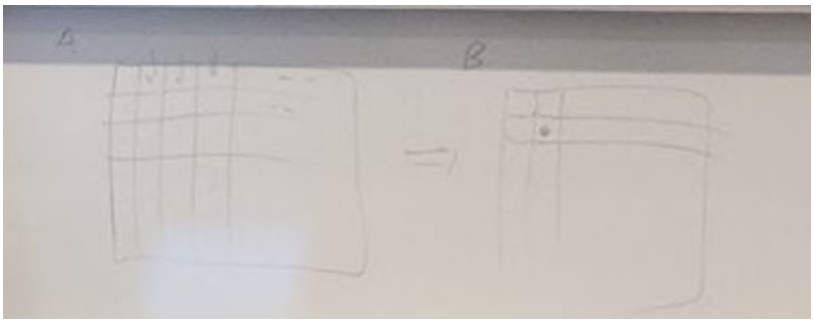
-- machine dependent (intel nehalem uses 3 CPU cycles: what is this?)
-- recognize sequence of sequence of products

```
for (i = 0; i < n; i++) {
   int ni = n*i;
   for (j = 0; j < n; j++)
     a[ni + j] = b[j];
}
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

-- sharing common subexpressions
-- stencil computation:



-- reuse portions of expressions
-- gcc will do this with O(1)

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
leaq    1(%rsi), %rax  # i+1
leaq    -1(%rsi), %r8  # i-1
imulq   %rcx, %rsi     # i*n
imulq   %rcx, %rax     # (i+1)*n
imulq   %rcx, %r8      # (i-1)*n
addq    %rdx, %rsi     # i*n+j
addq    %rdx, %rax     # (i+1)*n+j
addq    %rdx, %r8      # (i-1)*n+j
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

- Optimization Blockers
-- Procedure Calls

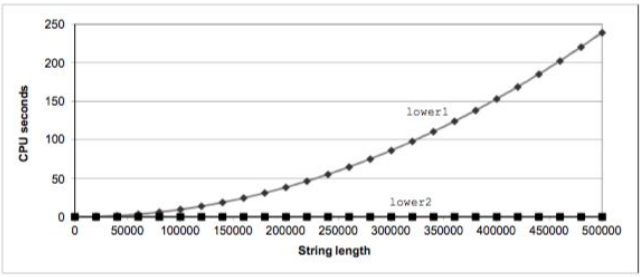
```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

-- strlen called every iteration, which itself does is a linear search

-- N calls to strlen hence, O(N^2))
-- Can be fixed by setting a var len to the string's length outside the loo
-- Example of code motion -- changes to O(N)

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



-- Compiler couldn't do this because (esp when not in same file)
-- Procedure may have side effects
-- Alters global state each time called (EVEN IF IT IS CONST? yes sometimes because it may still alter global state or it may actually get fixed)
-- String may change
-- Function may not return same value for given arguments
-- depends on other parts of global state
-- procedure lower could interact with strlen

--* Compiler treats procedure call as a black box, weak optimization around it *-
-- Remedies
-- Use of inline functions
-- copy pastes code when function call made at compile time
-- weak optimization near them
-- inline -- increases pressure on instruction cache
-- Do code motion

-- Memory Matters (what exactly is expensive? the loading and storing done by CPU)

Memory Matters

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0        # FP load
        addsd   (%rdi), %xmm0               # FP add
        movsd   %xmm0, (%rsi,%rax,8)        # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

-- Code looks up and updates b[i] on every iteration
-- Memory access and updates are extremely expensive
-- Compiler can't do this because it doesn't know if a and b overlap
-- Memory Aliasing (when the arrays etc. may overlap)

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

init: [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 22, 16]
i = 2: [3, 22, 224]

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

-- in this case b[i]'s value changes constantly -> causing a's values to change and hence aliasing occurs

-- doesn't happen with registers because they are explicitly referred to %rsi and %rdi will have no overlap

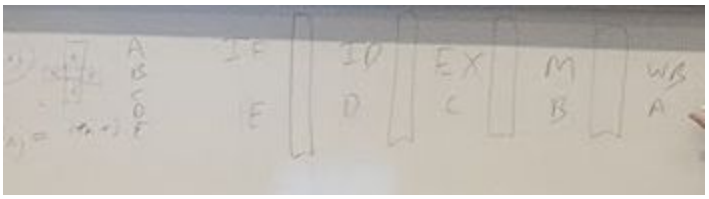
-- Remove aliasing

-- Initialize a temp variable val (that stores the sum in a register until the end)

-- possible as there is no need to store intermediate results

- Exploiting Instruction-Level Parallelism

Pipeline



IF - Instruction Fetch - get next instruction, increment %rsi

ID - Instruction Decode - figure out what to do, get values from named register, Simple Instruction format means we know which instructions we may need before the instruction is fully decoded

EX - Execute

M - Memory

WB - WriteBack - place results in appropriate registers

(here higher throughput- measure of the rate of flow of the system)

depends on the number of functional units you have

-- Latency - how many cycles it takes to complete an instruction

(many processor can execute more than a single instruction in a pipeline)

-- Need general understanding of modern processor design

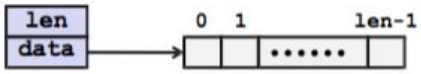
-- Hardware can execute multiple instructions in parallel

-- Performance limited by data dependencies
  -- x set to something and then something uses x --> marks a data dependency
-- Simple Transformation can yield dramatic performance improvement
  -- compilers can often not make these transformations
  -- lack of associativity and distributivity in floating point arithmetic

Hence, need to find instructions that are independent

Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```



- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

```
/* retrieve vector element
   and store at val */
int get_vec_element
  (*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Benchmark Computation

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

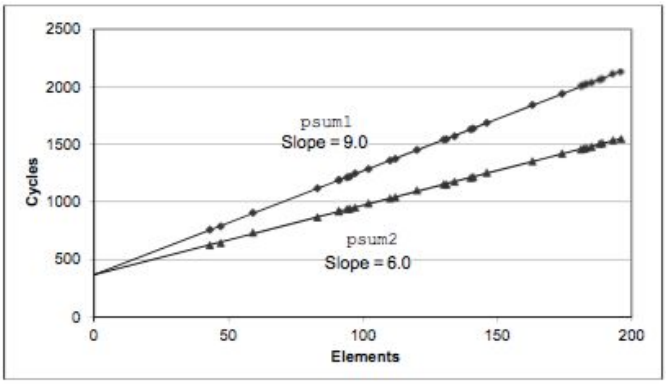
Compute sum or product of vector elements

- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`
- **Operations**
  - Use different definitions of `OP` and `IDENT`
  - `+` / `0`
  - `*` / `1`

- possible redundancies:
  -- function call inside loop condition
  -- function call inside loop
  -- possible memory aliasing issue?
-- Post Basic Optimizations (mov vec_length out of loop, accumulate in temp)

Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE is slope of line



Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

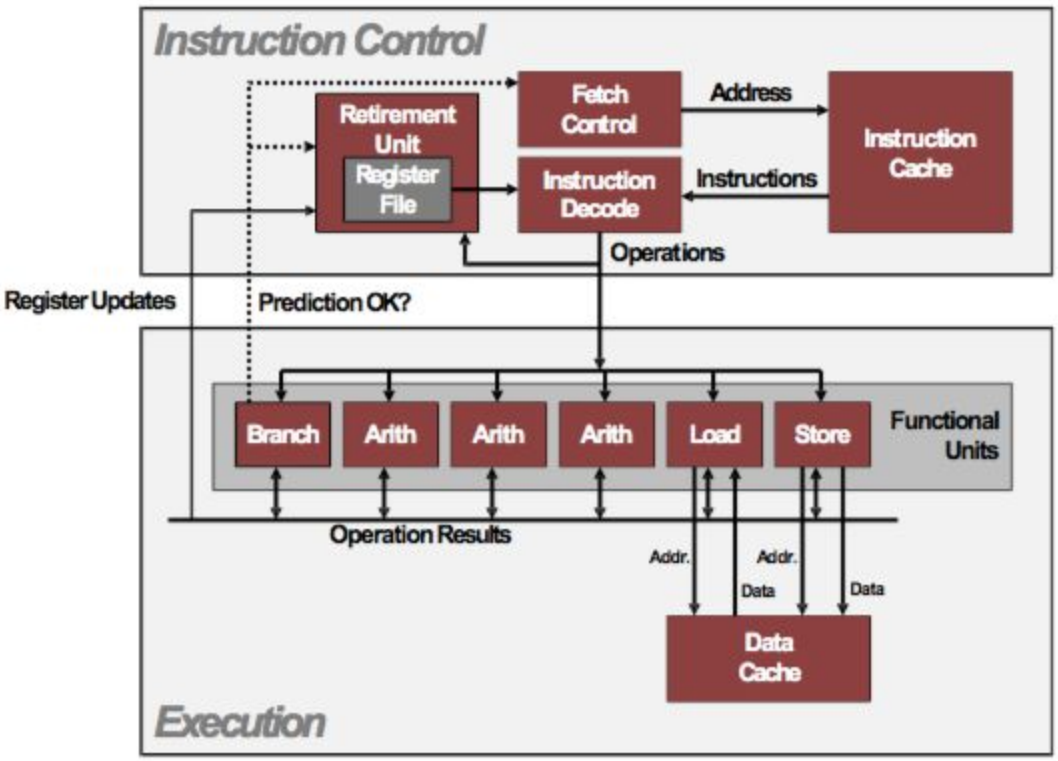
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	10.14

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Modern CPU Design



-- Execution is where addition, and or etc, mem access occurs
-- Instruction control
Fetch (in order) --> Execution (out of order) --> Retirement (out of order)
  -- Instruction cache contains the actual instructions that will be executed
  -- Fetch control - what location they
  -- Register file is where register values are stored
  -- Retirement Unit - makes sure that it looks like the instructions were executed in the same order
    -- ensures that the global state is as it should be when exceptions are thrown
-- Execution
  -- Functional units are where the addition etc, happens,
  -- Load Store - handles memory

Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

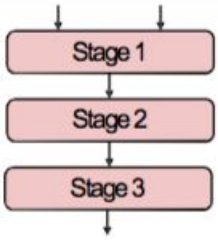
- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

-- Pipelined Functional Units

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

-- Divides the computation into stages
-- Pass partial computations from stage to stage
-- Stage i can start on new computation once values passed to i+1

-- If functional units didn't have pipelines then first a*b would have to be done, then a*c

Why can't a*b a*c enter stage 1 at the same time? as they're completely independent?

Haswell CPU

- 8 Total Functional Units
- **Multiple instructions can execute in parallel**
  - 2 load, with address computation
  - 1 store, with address computation
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide
- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

- What exactly does Cycles/Issue column mean? If it is pipelined would take fewer cycles (**give example with load)**

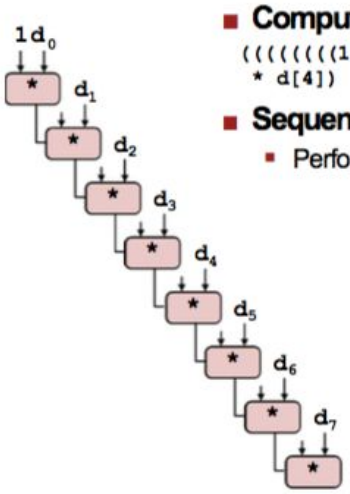
x86-64 Compilation of Combine4

- Inner Loop (Case: Integer Multiply)

```
.L519:                          # Loop:
  imull  (%rax,%rdx,4), %ecx    # t = t * d[i]
  addq   $1, %rdx               # i++
  cmpq   %rdx, %rbp             # Compare length:i
  jg     .L519                  # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Combine4 = Serial Computation (OP = *)



- Computation (length=8)

```
(((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- Sequential dependence
  - Performance: determined by latency of OP

-- Loop unrolling

-- Instead of doing one iteration at a time, do more than one at a time

Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

why would this improve anything because isn't it dependent?

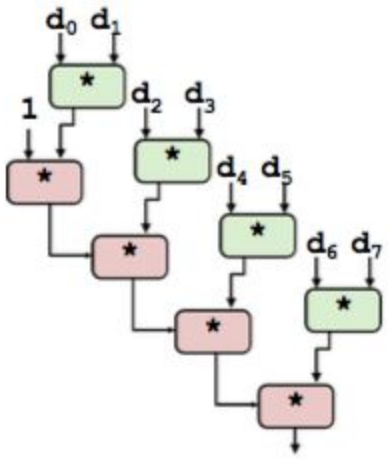
Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

- **Helps integer add**
  - Achieves latency bound
- **Others don't improve.** ***Why?***
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

- Breaks sequential dependency
  -- Ops in the next iteration can be started early (no dependency)
  -- as opposed to earlier - data dependency as operation is done on x directly
- Overall performance
  -- N elements, D cycles latency/op
  -- (N/2+1) * D cycles: CPE = D/2



Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

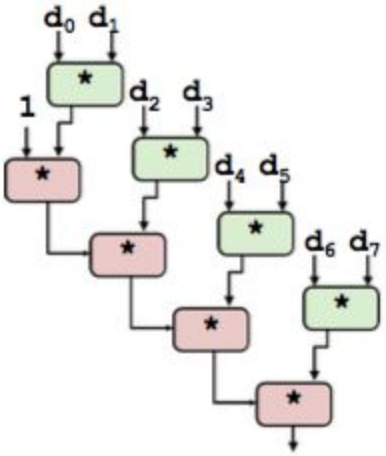
- Can this change the result of the computation?
- Yes, for FP. *Why?*

-- Improving performance even further

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation



Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- **Nearly 2x speedup for Int *, FP +, FP***
  - Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

  - Why is that? (next slide)

2 func. units for FP*
2 func. units for load

4 func. units for int +
2 func. units for load

-- Int + makes use of 2 load units
  -- not enough resources for others (if we added more functional units would this be fixed?)
-- Unrolling & Accumulation
  -- can unroll to any degree L
  -- can accumulate K results in parallel
  -- L must be multiple of K
-- Limitations
  -- Diminishing returns

Unrolling & Accumulating: Double *

- **Case**
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50

FP*	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Accumulators

-- cannot go beyond lim of exec units

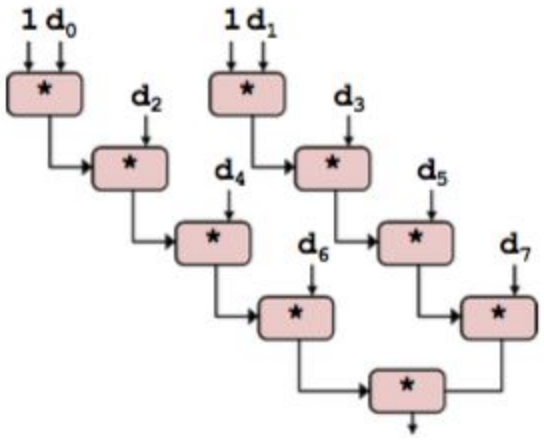
-- Large overhead for short lengths

-- finish off iterations sequentially

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- **Different form of reassociation**



Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

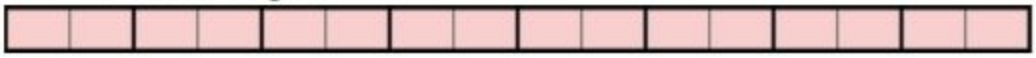
have registers that hold more than a single value
-- example data level parallelism - can occur by packing the data into a single register

Programming with AVX2

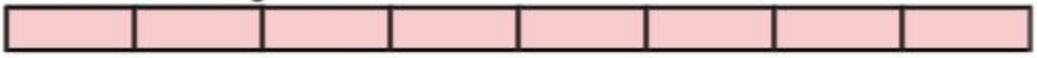
YMM Registers

- 16 total, each 32 bytes
- 32 single-byte integers

- 16 16-bit integers

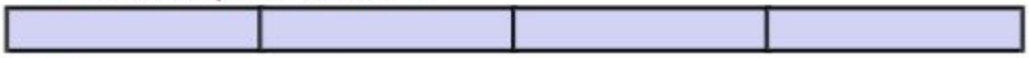


- 8 32-bit integers



- 8 single-precision floats

- 4 double-precision floats



- 1 single-precision float

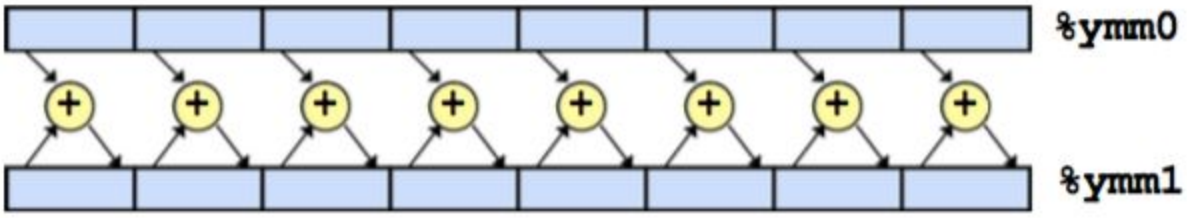
- 1 double-precision float

-- Allows the use of multiple register content values simultaneously as long as you have multiple func units for those values

SIMD Operations

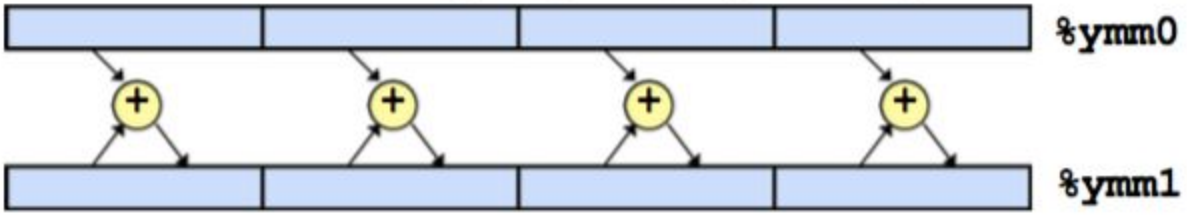
- SIMD Operations: Single Precision

```
vaddsd %ymm0, %ymm1, %ymm1
```



- SIMD Operations: Double Precision

```
vaddpd %ymm0, %ymm1, %ymm1
```



Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec Throughput Bound	0.06	0.12	0.25	0.12

- **Make use of AVX Instructions**
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

- Dealing with Conditionals

What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

  . . .

404685:  repz retq
```

Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

-- Issue is that there are two possible sets of instructions that can be pre-loaded with conditionals, the branch target or the next instruction

-- Branch Prediction
  -- guess which way branch will go
  -- begin executing instructions at predicted position
    -- don't actually modify register or memory data

Branch Misprediction Invalidation

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

Assume vector length = 100

i = 98

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 100

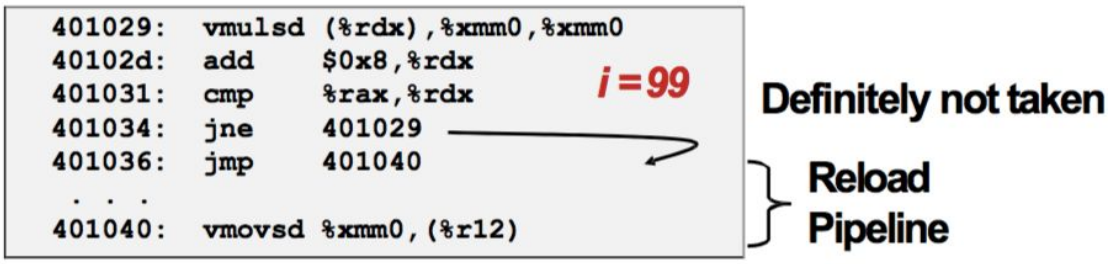
```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Invalidate

-- %xmm0 - read invalid location
-- i = 100 - incorrect executed
-- i = 101 - incorrect fetching

Branch Misprediction Recovery



- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

- Summary
  -- Good compiler (that can do aggressive optimization) and flags
  -- Avoid:
    -- hidden algorithmic inefficiencies
    -- compiler-unfriendly code
      -- optimization blockers -- procedure calls -- memory references
    -- look carefully at innermost loops (where most work is done)
  -- Tune code for machine
    -- exploit instruction-level parallelism
    -- avoid unpredictable branches
    -- make code cache friendly (later)

The Memory Hierarchy

- Storage technologies and trends
  -- RAM
    -- Key features
      -- Traditionally a chip
      -- basic unit is a cell, one bit per cell
      -- multiple ram chips form memory
  -- Two varieties:

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

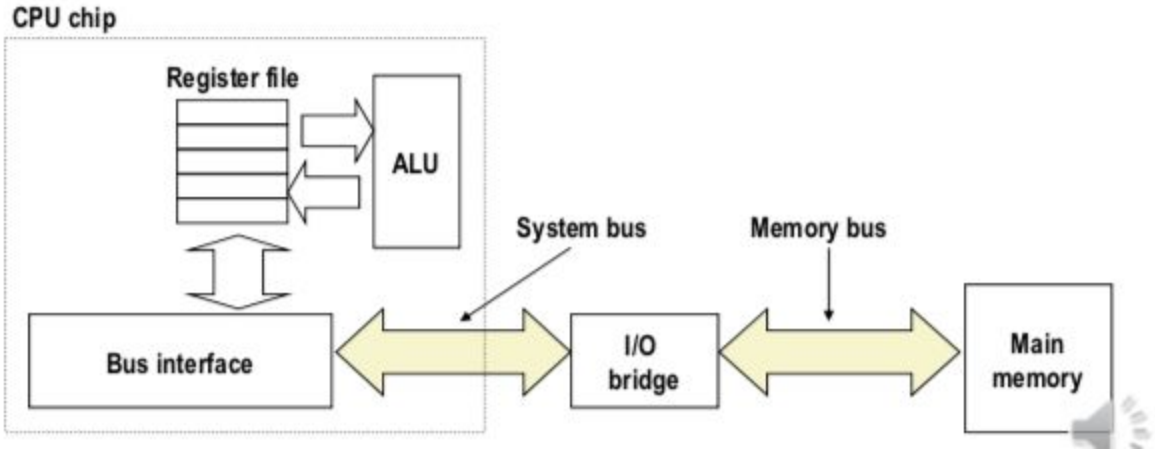
    -- Transistors per bit:
    -- Need Refresh (UNCLEAR):contents of the SRAM don't need to be constantly renewed or refreshed to ensure data's preservation, does need power however hence is volatile

    -- Needs EDC: What does this mean?

Nonvolatile Memories

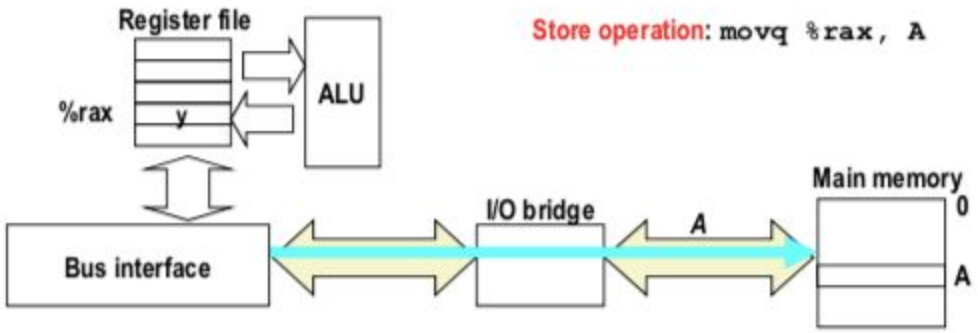
- **DRAM and SRAM are volatile memories**
  - Lose information if powered off.
- **Nonvolatile memories retain value even if powered off**
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically eraseable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 erasings
- **Uses for Nonvolatile Memories**
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,...)
  - Disk caches

  -- Traditional Bus Structure Connecting CPU and Memory
    -- A bus is a connection of parallel wires that carry address, data and control signals
      -- shared by multiple devices



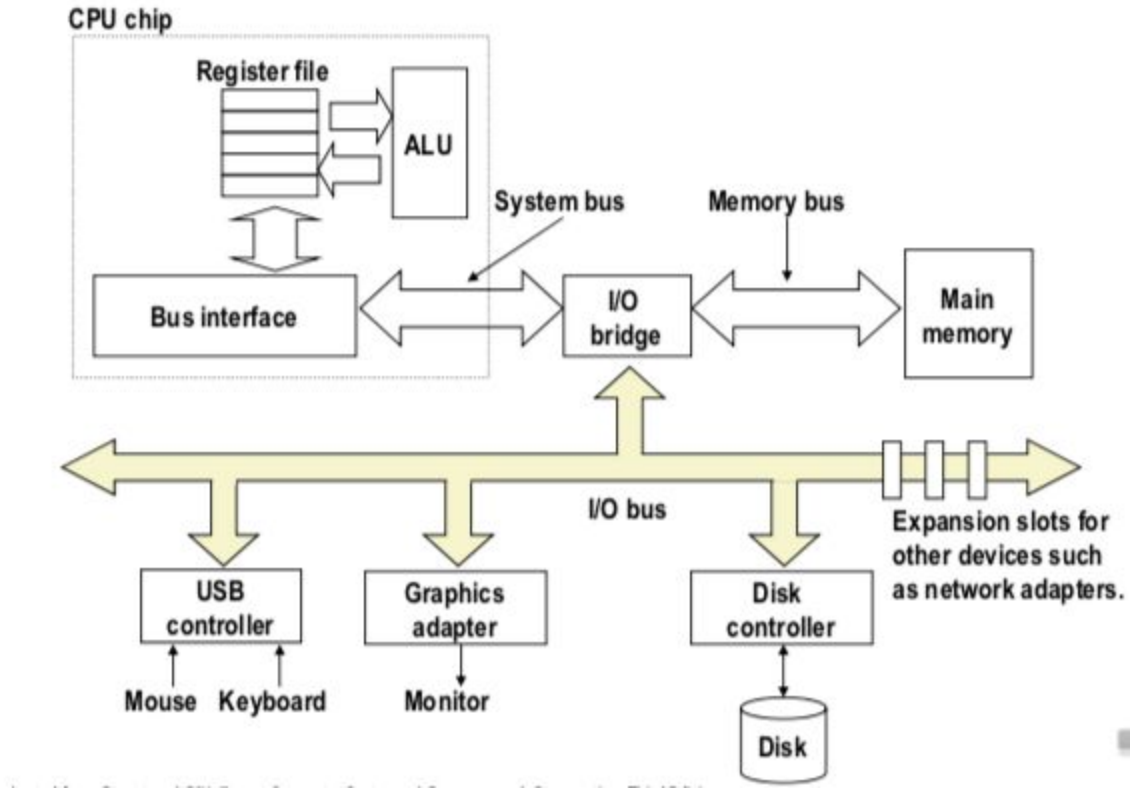
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



-- Memory Read Transaction (Load op: movq A, %rax)
  -1- CPU places address A on the memory bus
  -2- Main memory reads A from the memory bus, retrieves word x and places it on the bus
  -3- CPU read word x from the bus and copies it into register %rax
-- Memory Write Transaction (Store op: movq %rax, A)
  -1- CPU places address A on bus, main memory reads it and waits for correposining data word to arrive
  -2- CPU places data word y on the bus
  -3- Main memory reads data word y from the bus and stores it address A
-- I/O Bus

I/O Bus



-- Reading a Disk Sector

-1- CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller

-2- Disk controller reads the sector and performs a direct memory access (DMA) and transfer into main memory

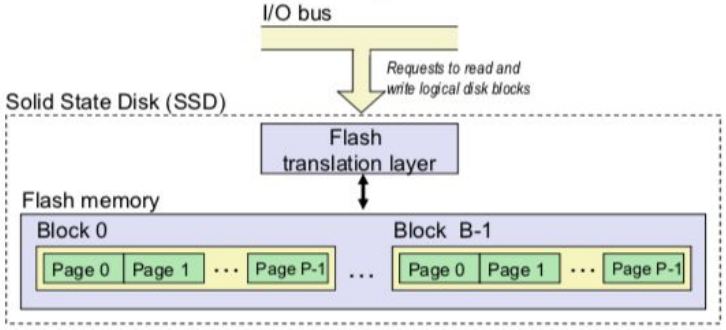
(non deterministic latency)

-3- when the DMA transfer completes, the disk controller notifies the CPU with an interrupt

(i.e. asserts a special "interrupt" pin on the CPU")

-- Solid State Disks (non volatile)

Solid State Disks (SSDs)



- **Pages: 512KB to 4KB, Blocks: 32 to 128 pages**
- **Data read/written in units of pages.**
- **Page can be written only after its block has been erased**
- **A block wears out after about 100,000 repeated writes.**

-- Advantages:
-- No moving parts hence latency is hugely improved (is it deterministic?)
-- Less chances for hardware issues (impact doesn't affect as much)
-- Disadvantages:
-- Cannot store as much memory as rotational disks
-- Limited number of writes before the memory becames unwriteable
-- Most SSDs use a special sophisticated wearout spreading alg. that allows the # of writes to one block to be spread out (by moving data around)
-- more expensive

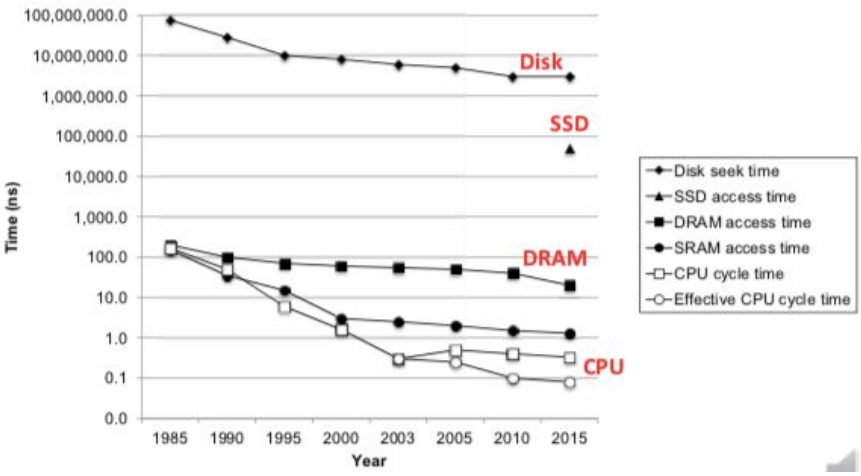
-- SSD Performance Characteristics
(sequential access is faster than random access contiguous blocks of memory are faster to access than randomly organized memory: data that is closer will have faster access time)

Sequential read tput	550 MB/s	Sequential write tput	470 MB/s
Random read tput	365 MB/s	Random write tput	303 MB/s
Avg seq read time	50 us	Avg seq write time	60 us

-- Random writes are somewhat slower
-- erasing a block takes a long time (~1ms)
-- Modifying a block page requires all other pages to be copied to a new block
-- in earlier SSDs, the read/write gap was much larger

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



-- CPU gotten way faster (transistors), but wires didn't scale as well (hence memory latency didn't improve: except for SRAM that scaled more than most but still less than CPU)
-- going to disk is extremely expensive, SSD less so but still, DRAM still so but not as much, SRAM a bit too
-- hence programming must be done to minimize such memory accesses

- Locality of Reference: the key to bridge the CPU-Memory gap
-- Principle of Locality: programs tend to use data and instructions with addresses near or equal to those they have used recently
-- Temporal Locality - recently referenced items are likely to be referenced again in the near future
-- Spatial locality - items with nearby addresses tend to be referenced close together in time

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern). — **Spatial locality**
  - Reference variable sum each iteration. — **Temporal locality**
- **Instruction references**
  - Reference instructions in sequence. — **Spatial locality**
  - Cycle through loop repeatedly. — **Temporal locality**

Example:

-- Qualitative Estimates of Locality

-- Learn to look at code and get a qualitative sense of its locality

Qualitative Estimates of Locality

- **Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.**
- **Question: Does this function have good locality with respect to array a?**

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

-- m rows and n columns

-- the loops change row number and col number

-- good temporal locality - as instructions are repeated inside for loop

-- spatial locality demonstrated by array a here as j increments to give successive elements in memory as the cols are contiguous

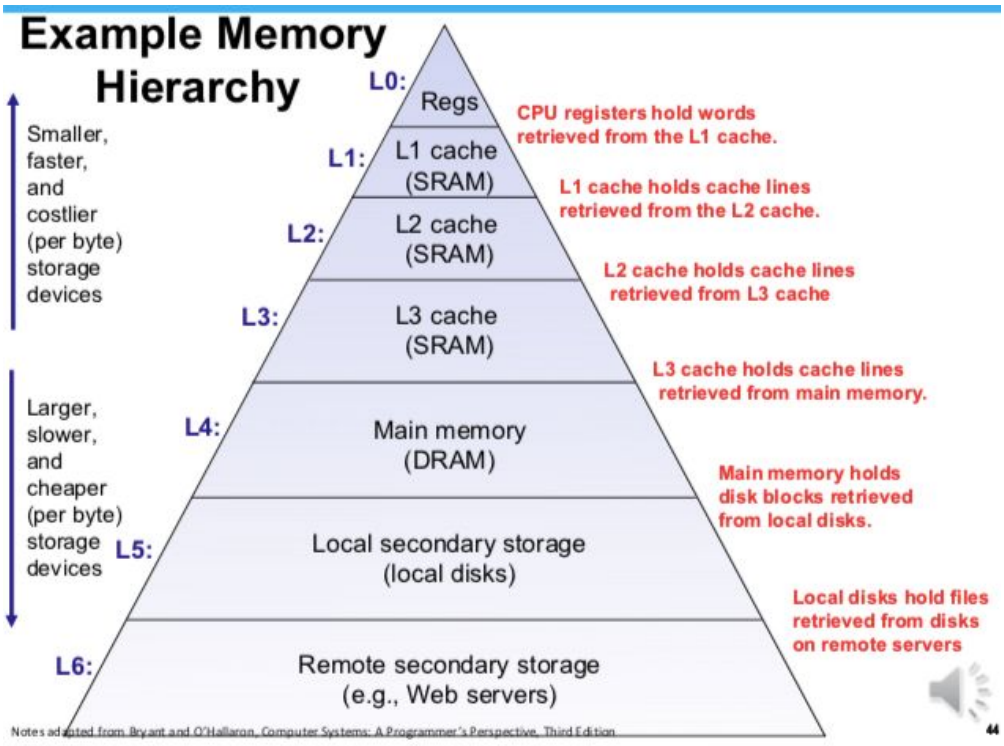
in contrast to here

where the first element of successive rows are referenced and hence are not contiguous

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

-- Memory Hierarchies
  -- Some fundamental & enduring properties of hardware & software
    -- Fast storage tech cost more per byte, have less capacity and require more power-->heat up
    -- gap between CPU and main memory speed is widening
    -- well written programs tend to exhibit good locality
  -- suggests an approach for organizing memory and storage systems known as a memory hierarchy

- Caching in the memory hierarchy

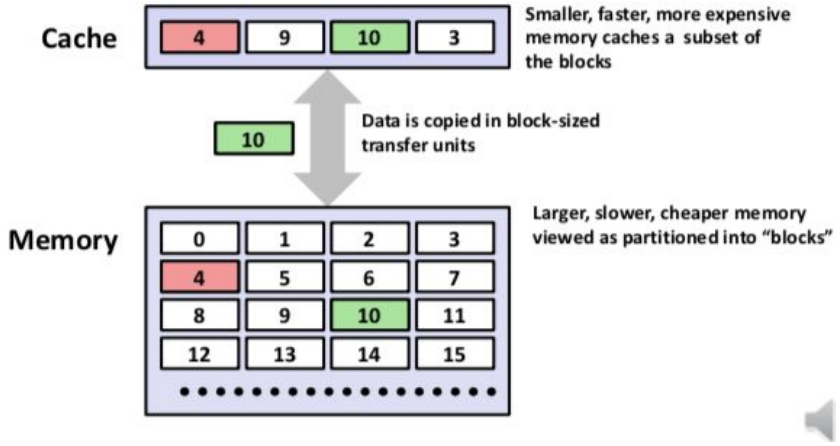


  -- Aim is to keep most memory accesses to be limited to the upper sections as much as possible
  -- Can be done by pulling a datum from a lower structure into an upper structure for use and keeping it there until its locality of reference has been exhausted

-- Caches
  -- Caches: A smaller, faster storage device that acts as a staging area for s subset of the data in a larger, slower device
  -- Fundamental idea of a memory hierarchy:
    -- for each k, the faster smaller device at level k serves as a cache for the larger,slower device at level k+1
    -- the memory hierarchy creates a large pool of storage that costs as much as the cheap storage

near the bottom, but that serves data to programs at the rate of the fast storage near the top

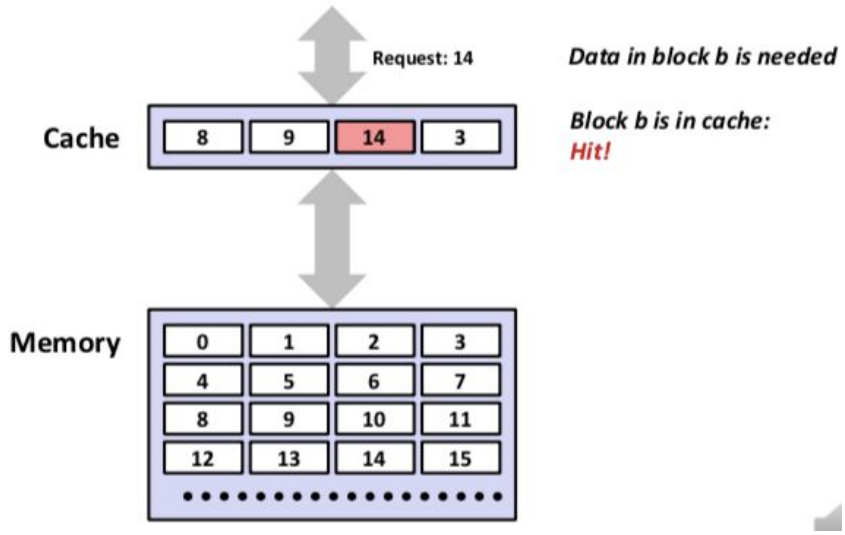
General Cache Concepts



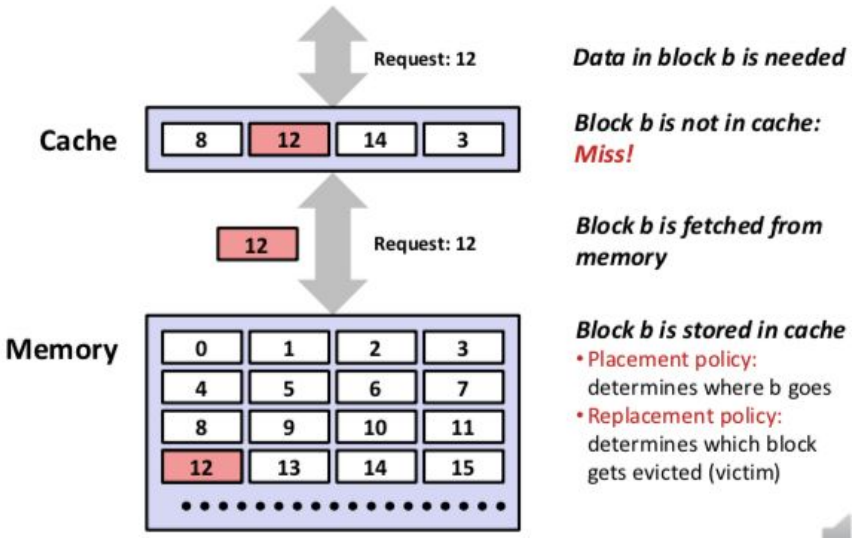
-- Cache is hardware managed (typically), registers are software managed

-- General Cache Concepts:

General Cache Concepts: Hit



General Cache Concepts: Miss



-- Examples of Caching in the Memory Hierarchy

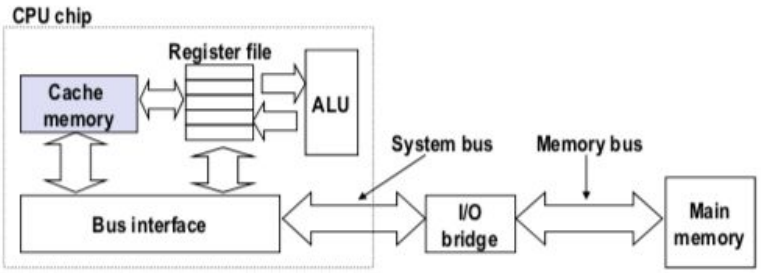
Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

-- Cache Memories

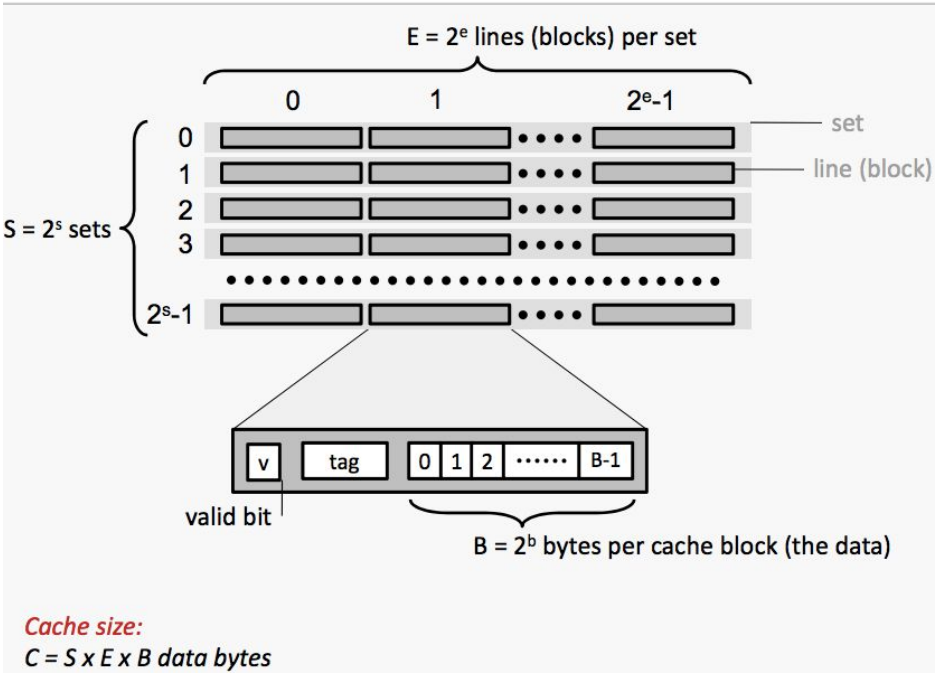
-- C.M. are small, fast SRAM based memory

Cache Memories

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in cache**
- **Typical system structure:**



-- General Cache Organizations (S, E, B)

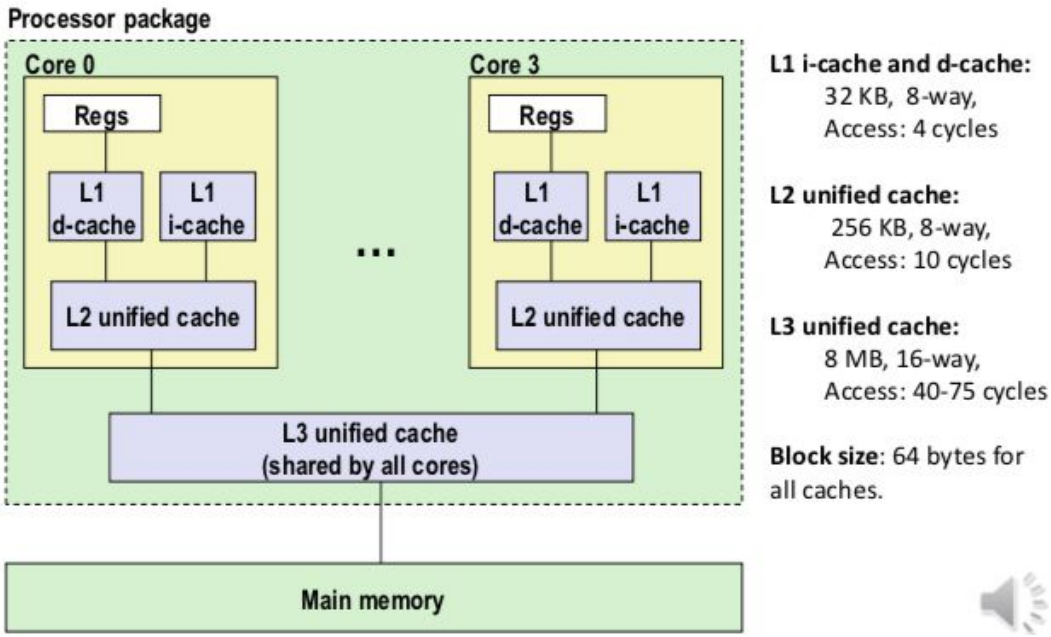


Cache size:
C = S x E x B data bytes

-- Works similar to hash table (many entries, many sets)
-- Number of sets where a block could do - a cache block will map to a certain set depending on its address
-- Number of ways/lines === the buckets in the set
-- valid bit: whether data is valid, tag bit:
-- C = S * E * B bytes of data

-- INTEL Core i7 Cache Hierarchy

Intel Core i7 Cache Hierarchy



-- Processor pipeline

-- Cache Performance Metrics

-- Miss Rate (fraction of not found of mem refs not found in cache)

-- typical numbers: 3-10% for L1, can be < 1% for L2 (Depending on size)

-- Hit Time

-- Time to deliver a line in cache to processor (including time to check whether a line is in the cache

-- Typical numbers: 4 clock cycles for L1, 10 clock cycle for L2

-- Miss Penalty:

-- Additional time required due to miss: 50-200 cycles for main memory (increases as you have to go further down)

Let's think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory
- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**
- **This is why "miss rate" is used instead of "hit rate"**

-- Writing Cache Friendly Code (qualitative understanding of locality quantified by understanding of cache memories)

-- Make the common case go fast

-- Focus on the inner loops of the core functions - called most times

-- Minimize misses in inner loops

-- Repeated refs to variables are good (temporal locality)

-- stride-1 reference patterns are good - accessing data that is not too far apart (spatial)

-- The Memory Mountain

-- Read Throughput (Read bandwidth)

(Number of bytes read from Memory per Second (MB/s)

-- Memory Mountain: Measured read throughout as a function of spatial and temporal locality to characterize memory system performance

Memory Mountain Test Function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *      array "data" with stride of "stride", using
 *      using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

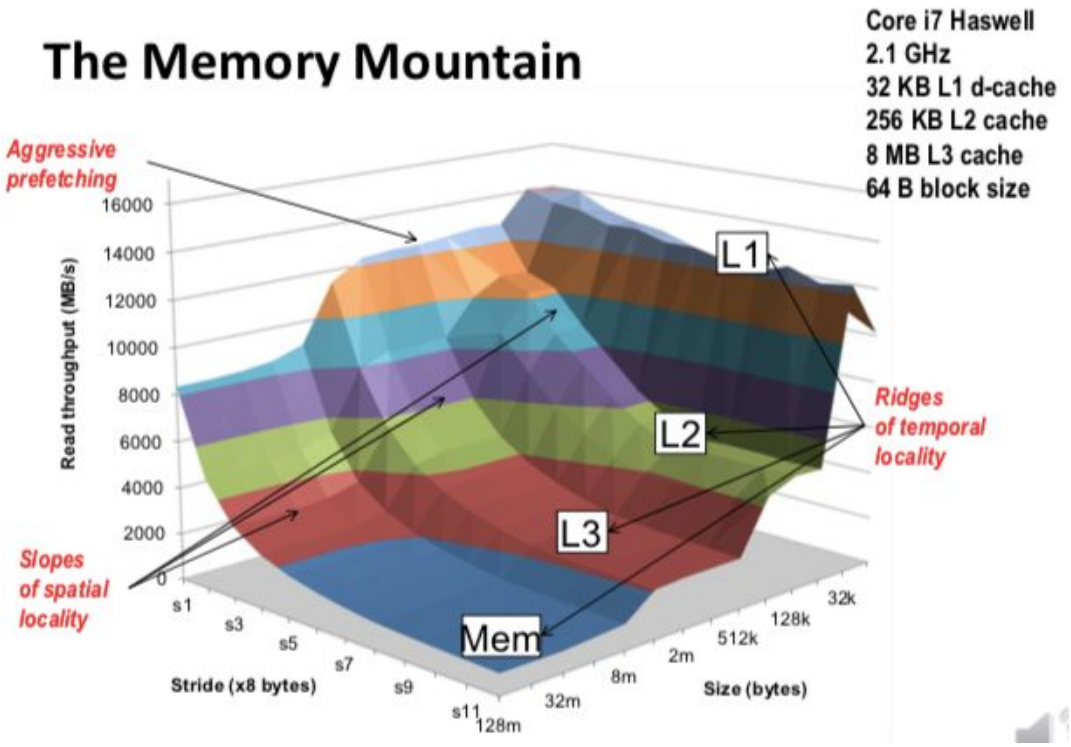
mountain/mountain.c

Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

2. Call `test()` again and measure the read throughput (MB/s)



-- Prefetching - guessing spatial locality by pulling values that are close by

Matrix Multiplication Example

- **Description:**
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

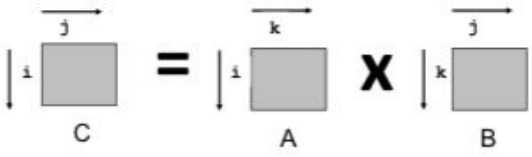
```
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable sum held in register

matmult/mm.c

Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- **Analysis Method:**
  - Look at access pattern of inner loop



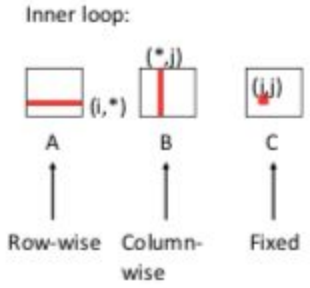
Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations
- **Stepping through columns in one row:**
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- **Stepping through rows in one column:**
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

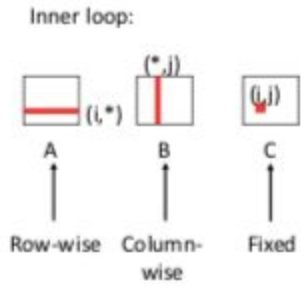


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```
matmult/mm.c

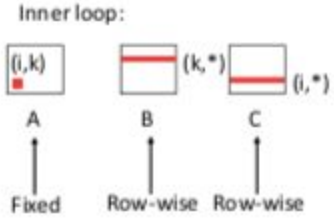


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```
matmult/mm.c

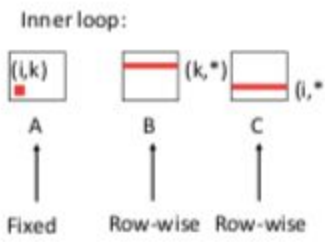


Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```
matmult/mm.c



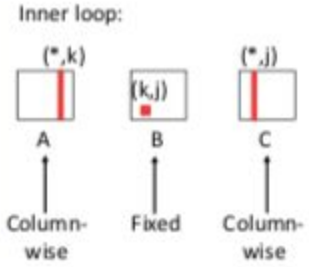
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```
matmult/mm.c



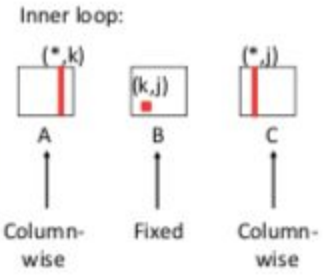
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

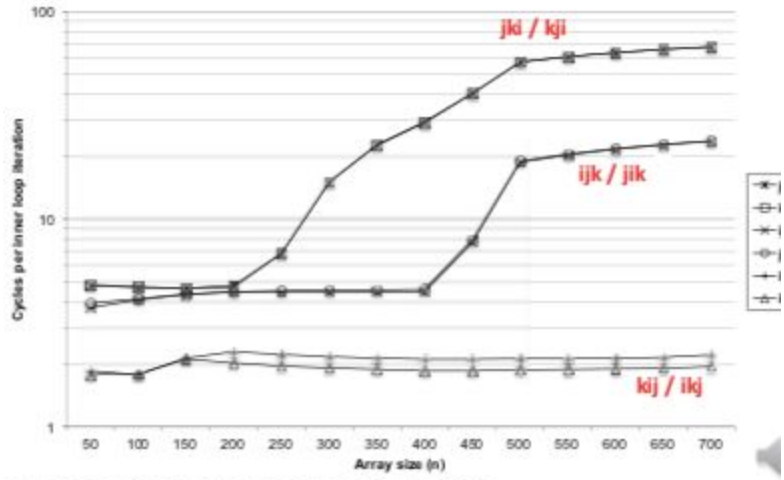
```
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

Core i7 Matrix Multiply Performance



adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

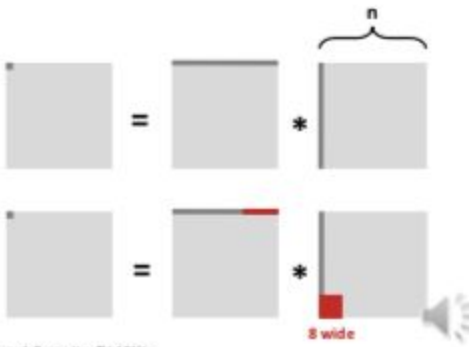
Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

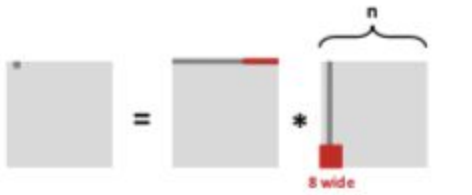
- **First iteration:**
  - n/8 + n = 9n/8 misses
  - Afterwards in cache: (schematic)



Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **Second iteration:**
  - Again: n/8 + n = 9n/8 misses

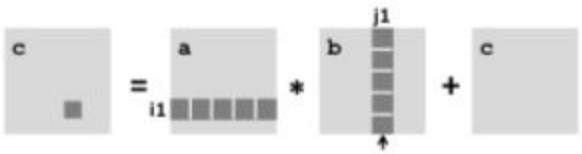


- **Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

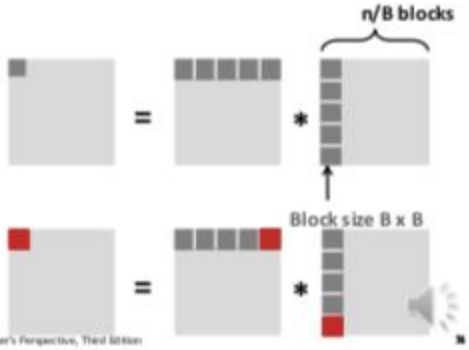
matmult/bmm.c



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Progr

Cache Miss Analysis

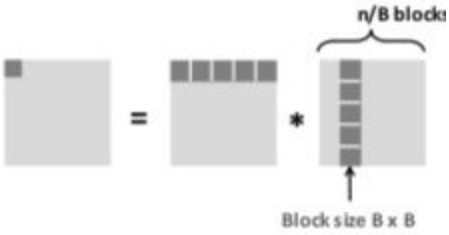
- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$



- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

Blocking Summary

- **No blocking: (9/8) * n^3**
- **Blocking: 1/(4B) * n^3**
- **Suggest largest possible block size B, but limit $3B^2 < C$!**
- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

Summary

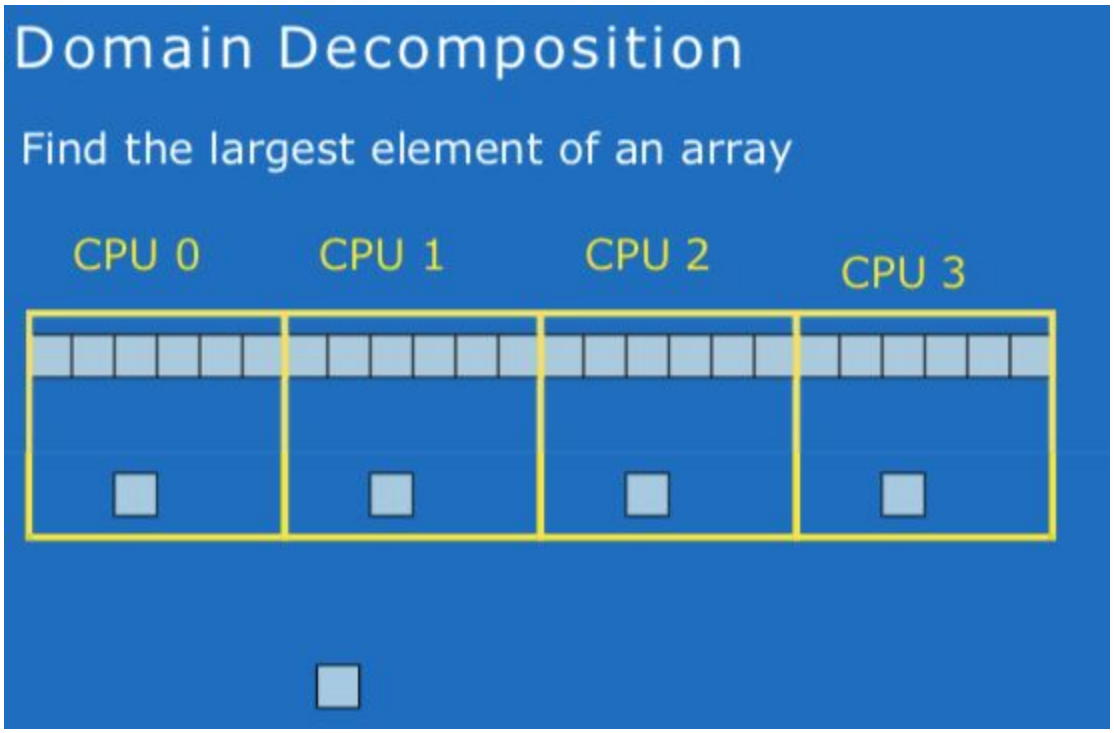
- **The speed gap between CPU, memory and mass storage continues to widen.**
- **Well-written programs exhibit a property called *locality*.**
- **Memory hierarchies based on *caching* close the gap by exploiting locality.**
  - Cache memories can have significant performance impact
- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

Parallelism

Objective:
Attempt to speed solution of a particular task by:
1. Dividing task into sub-tasks
2. Executing sub-tasks simultaneously on multiple processors

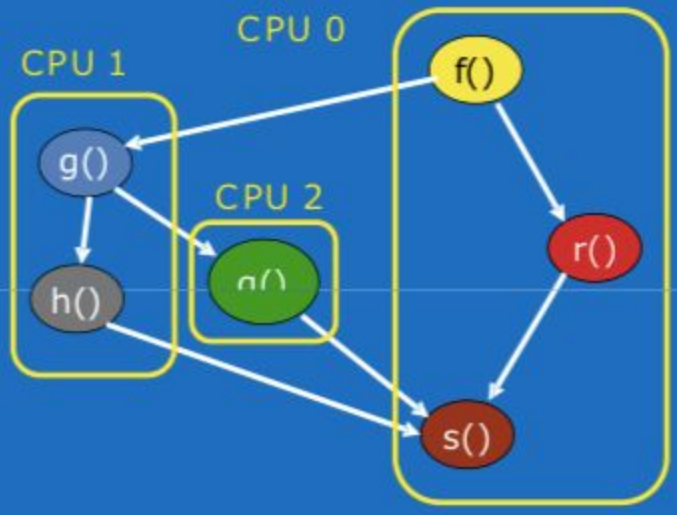
- Methodology
  -- Study problem, sequential program or code segment
  -- Look for opportunities for parallelism - few dependencies
  -- Try to keep all processors busy

Ways of Exploiting Parallelism
- Domain Decomposition
  -- Decide how data elements should be divided among processors
  -- Decide which tasks each processor should be doing
  E.g. Vector addition, Find largest element in array:



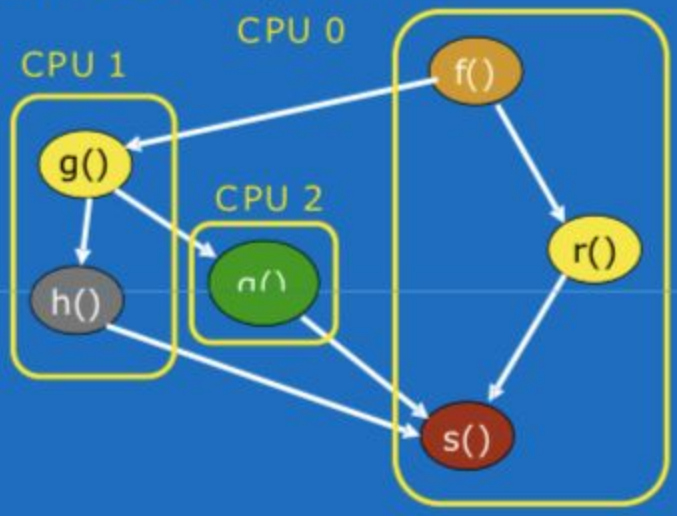
- Task (Functional) Decomposition
-- Divide tasks among processors
-- Decide which data elements are going to be accessed read and/or written by processors
E.g. event handler for GUI

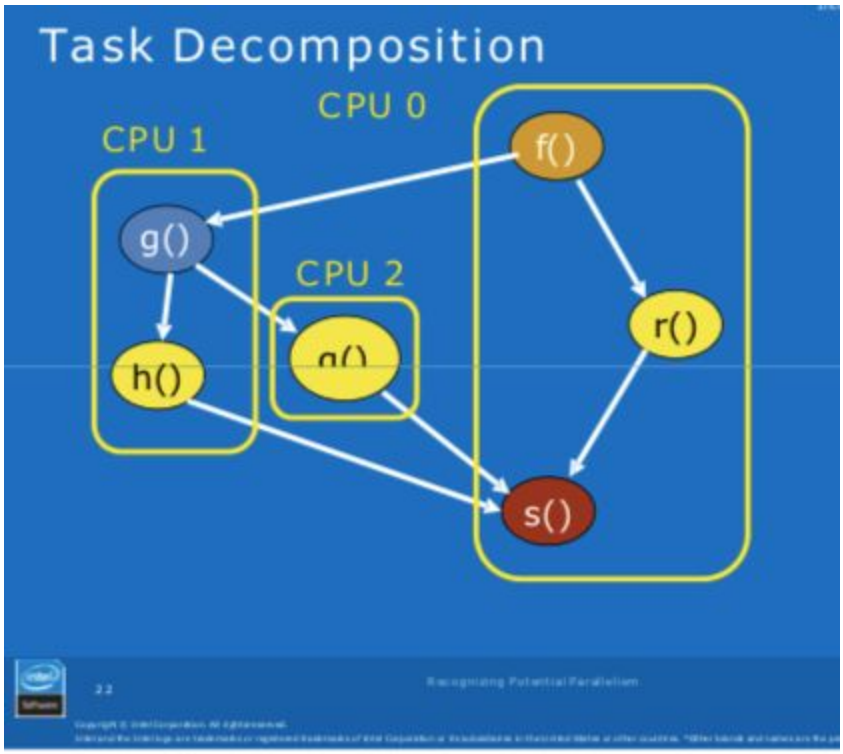
Task Decomposition

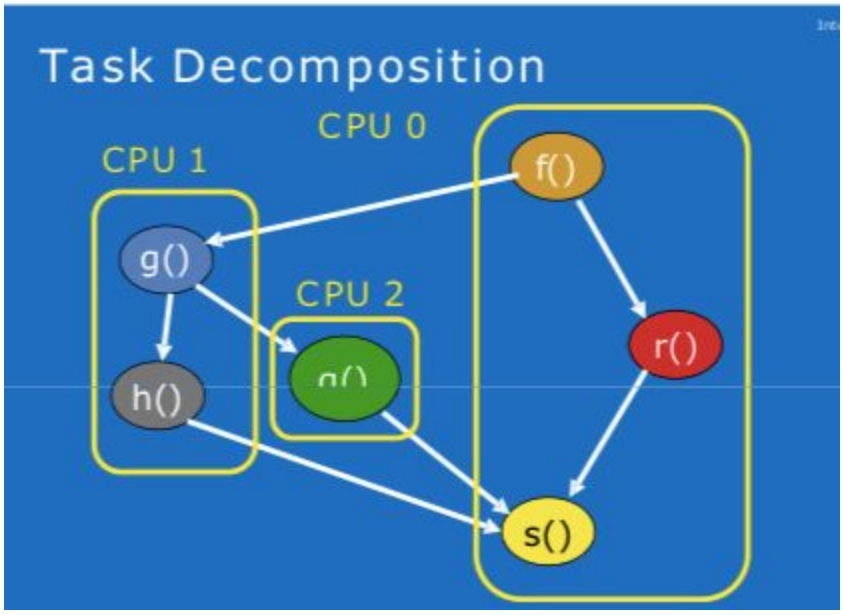


Intel® Software College

Task Decomposition



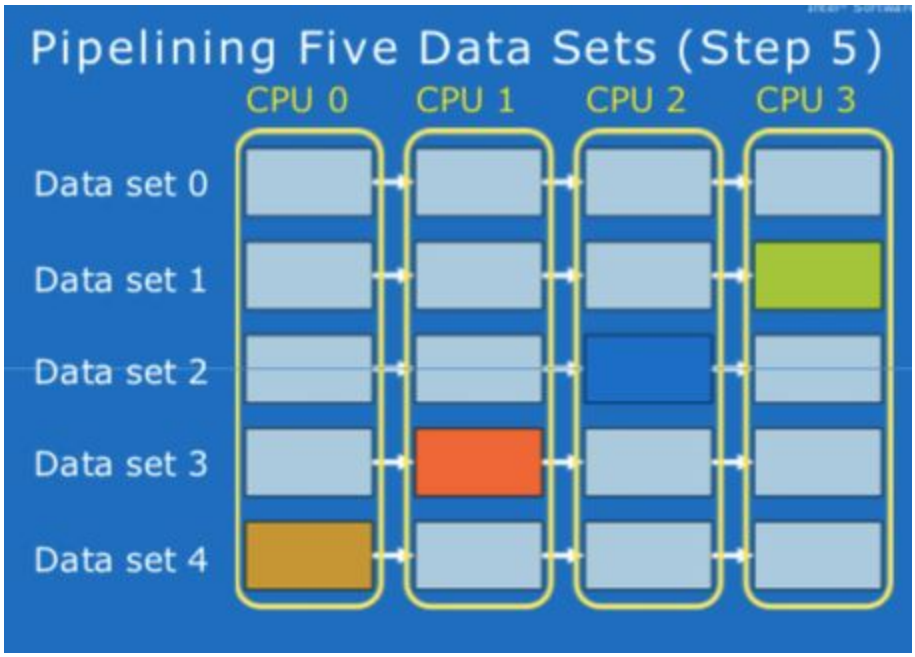




-- Communication chains (represented by edges) inter and intra CPU
-- Try and minimize this so that the communication overhead is not worse than the gain from parallelism
-- Not as optimal as domain decomposition in this case

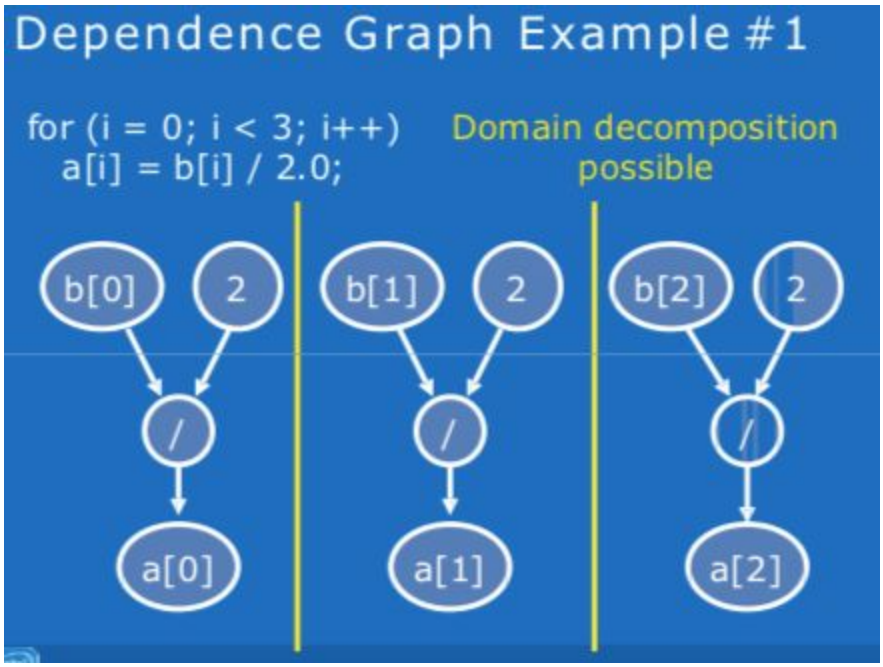
- Pipelining
  -- Special kind of task decomposition
  -- Assembly line parallelism

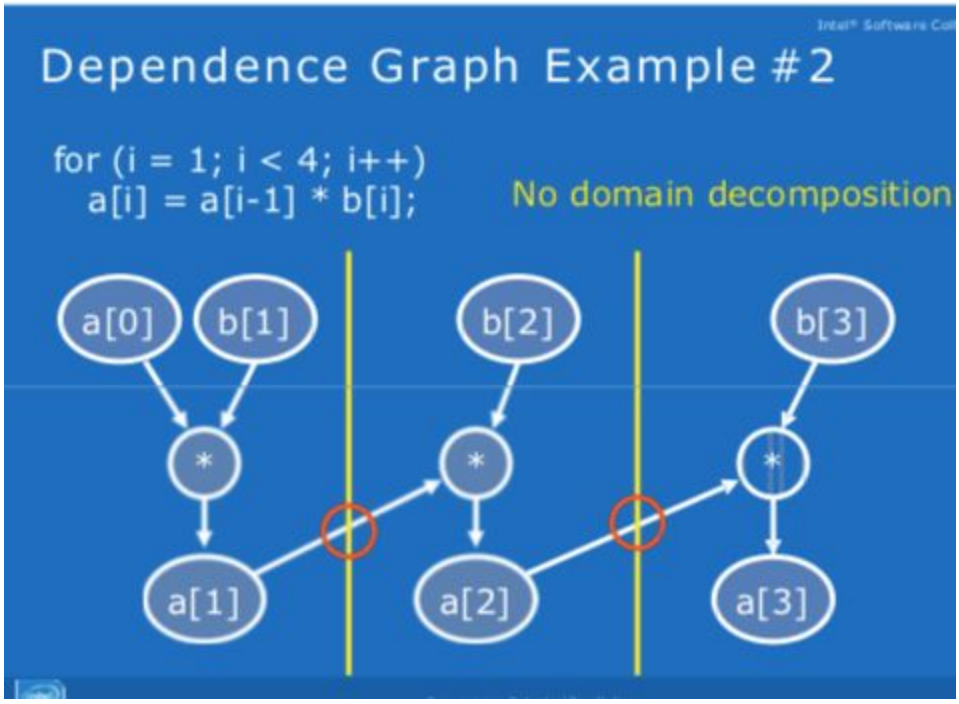
E.g. 3D rendering in computer graphics
The pipeline processes 1 data set in 4 steps

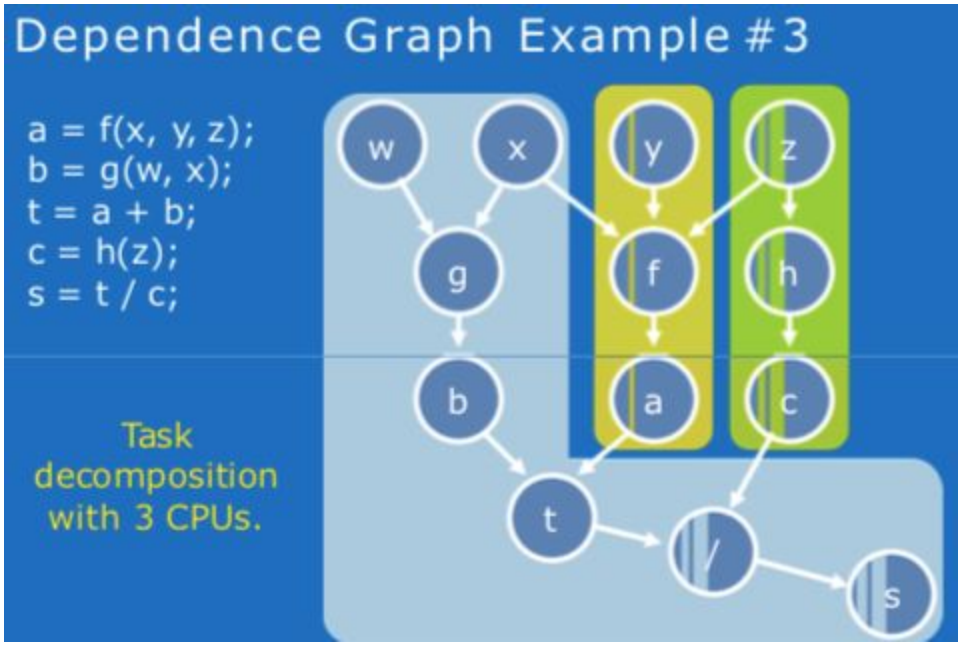


- Dependence graph
-- Node for each - var assignment, constant, operator or func call
-- Edges indicate use of variables and constants (data flow &/or control flow)

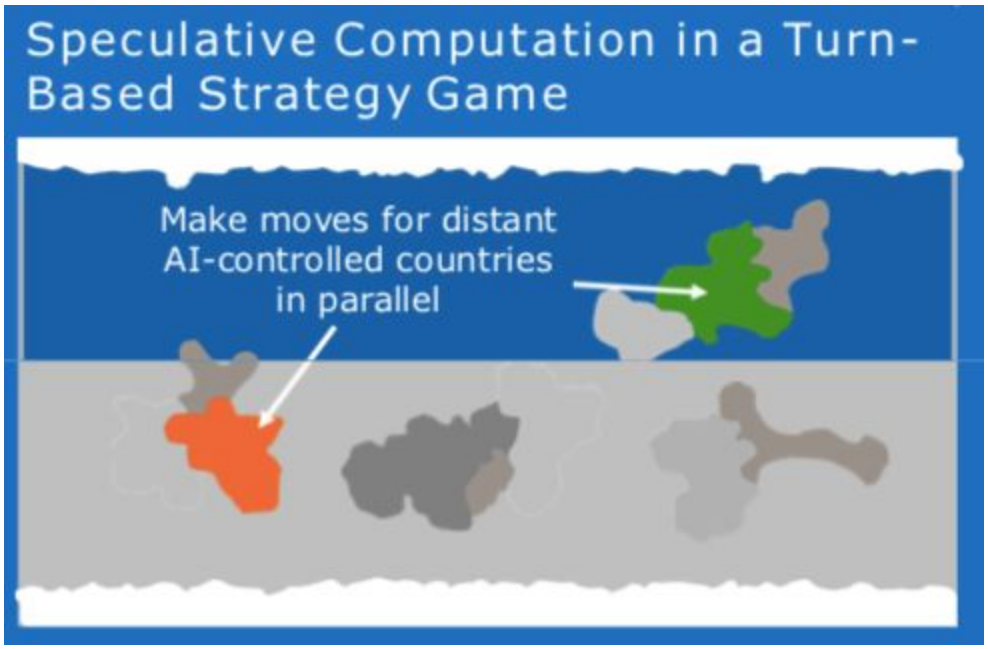
Examples:



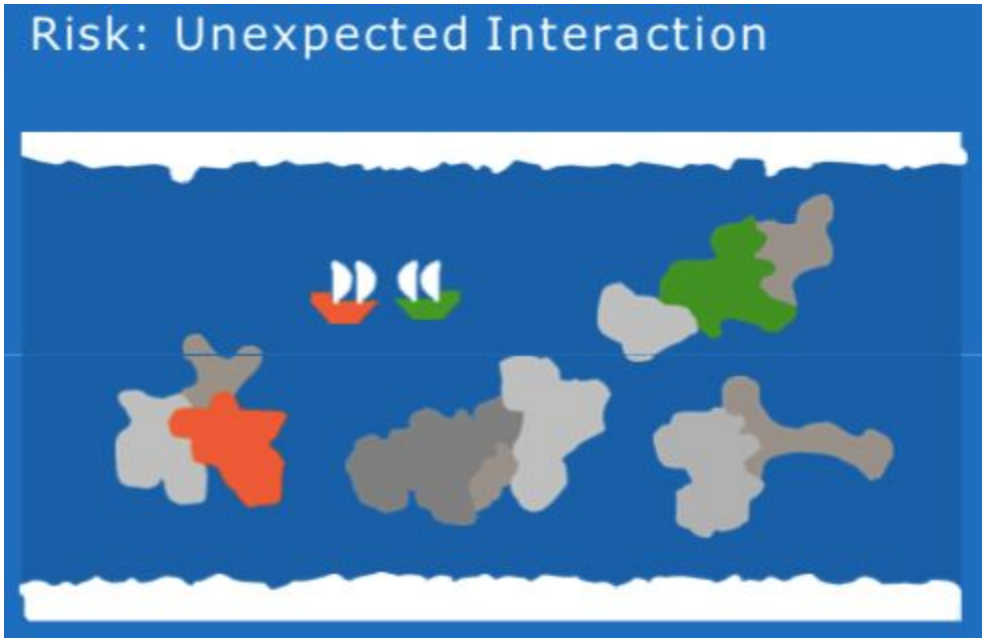




- Speculative Computation in a turn-based strategy game



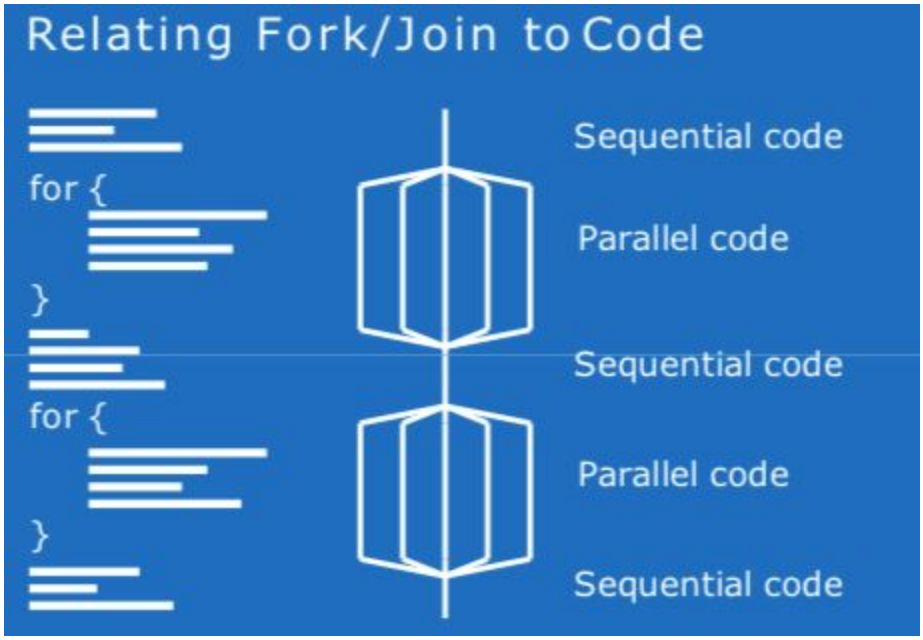
distant enough that they are independent - plan in advance separately



Risk: Unexpected interaction?

-- Might have to undo or rollback actions
-- Solution: Reverse time: must be able to undo an erroneous, speculative computation
like incorrect branch prediction
usually don't have a big pay off in parallelization

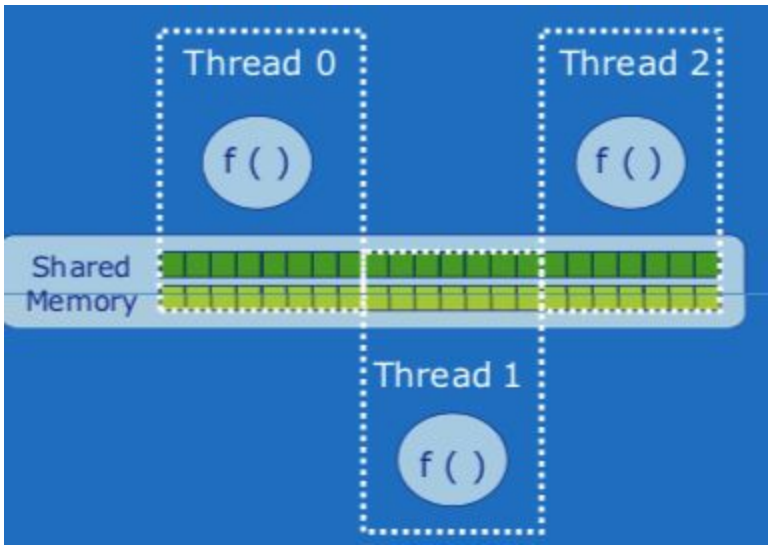
- Fork/Join Programming model:
  -- When program begins execution, only master thread active
  -- master thread executes sequential portions of program
  -- for parallel proportions of program, master thread forks (creates/awakens) additional threads
  -- At join (end of parallel section of code), extra threads are suspended or die

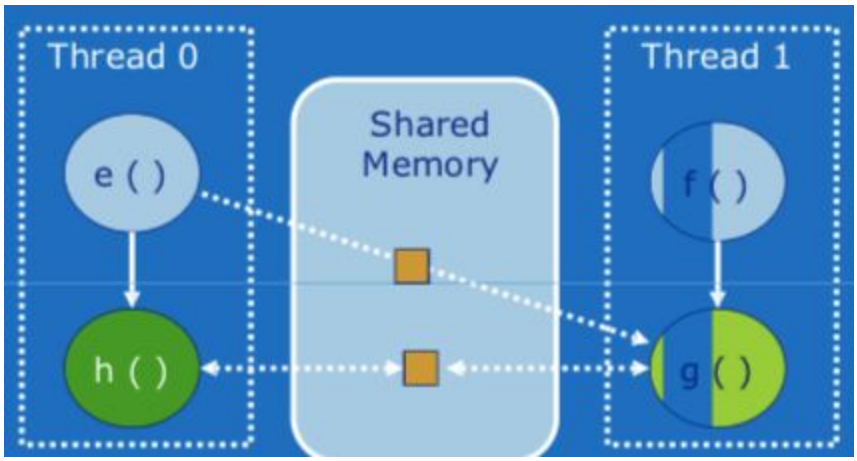


- Incremental Parallelization:
  -- Sequential program a special case of threaded program
  -- Programmers can add parallelism incrementally
  -- Profile program execution
  -- Repeat:
    -- choose best opportunity for parallelization
    -- transform sequential code into parallel code

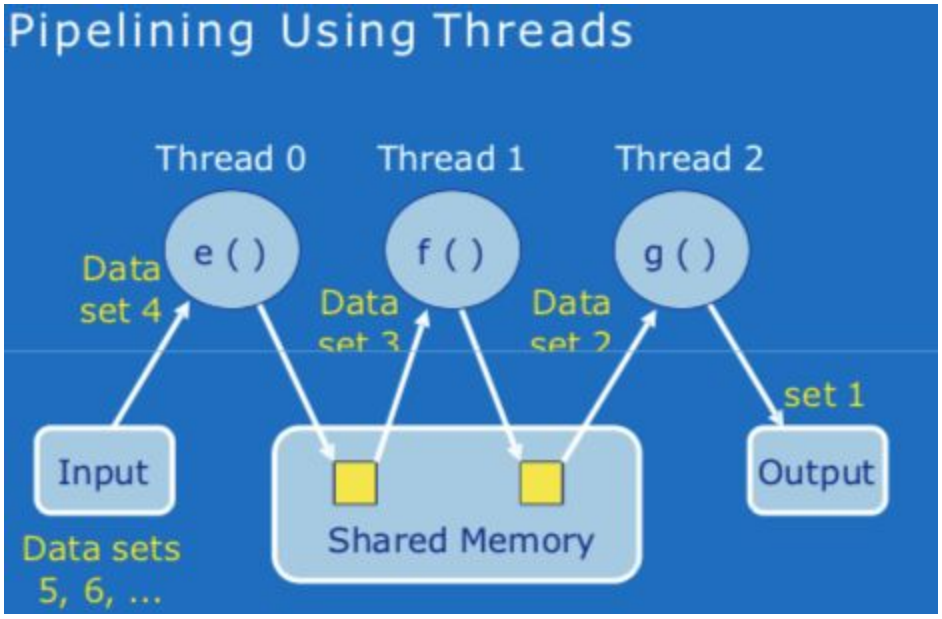
-- Until no further improvement

- Utility of Threads





-- Need for synchronization to be able to use shared memory



- Threads may have private and shared variables (take up less space cause only one instance)
not an issue if read only

-- Race condition when two threads need to read & write or write & write to the same shared mem

- OpenMP
  -- API for parallel programming (can use with C/C++/Fortran) - extension for existing langs
  -- Provides a set of pragmas that is converted to actual code from the libraries

- Strengths & Weaknesses of OpenMP
  -- Strengths
    -- Well-suited for domain decompositions
    -- Available on Unix and Windows NT
  -- Weaknesses
    -- Compiler cannot check for deadlocks or race conditions
    -- Not well tailored for functional decompositions

- Syntax of Compiler Directives
  -- A C/C++ compiler directive is called a pragma
  -- Pragmas handles by the preprocessor
  -- All OpenMP pragmas have the syntax
    #pragma omp <rest of pragma>
    -- at compile time, libraries insert actual code that does the parallelization
    -- hence without changing code, can be further parallelized as actual parallelization done by preprocessor by looking at pragmas (essentially just highlight avenues for parallelization
  -- Pragmas appear immediately before relevant construct

- Pragma: parallel for
  #pragma omp parallel for:
  tells the compiler to fork, also tells that the for loop that immediately follows can be done parallelly
  -- number of loop iterations must be computable at run time before loop executes
  -- loop must not contain a break return or exit
  -- no goto for a label outside of the loop

Example

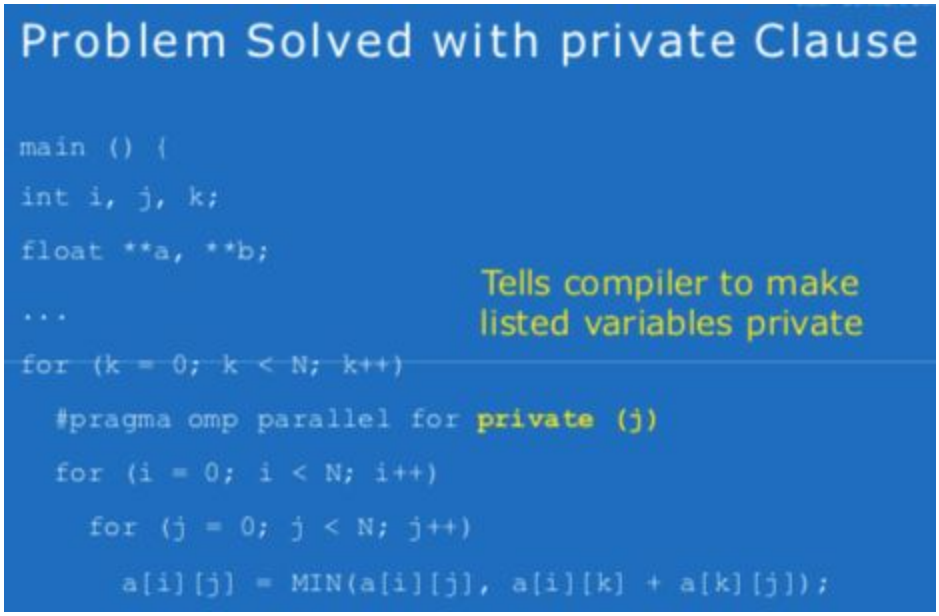
```
int first, *marked, prime, size;
...
#pragma omp parallel for
for (i = first; i < size; i += prime)
     marked[i] = 1;
```

- Matching Threads with CPUs
  - Function int omp_get_num_procs(void) returns the number of physical processors available to the parallel program

- Function void omp_set_num_threads (int t) allows you to set the number of threads that should be active in parallel sections of code
  - Can be called with different arguments at different points in the program
  - Inner loops just check every element with the condition
  - Outer loop depends on previous command (is it cause a[i][j] is changing)

- Grain size
  -- since fork/join is a source of overhead, we want to maximise the amount of work done for each fork/join i.e. the grain size
  -- Therefore middle loop made parallel
  -- The work done inside the inner loop will be work shared

- Private Variables
  #pragma omp parallel for shares the work of the loop



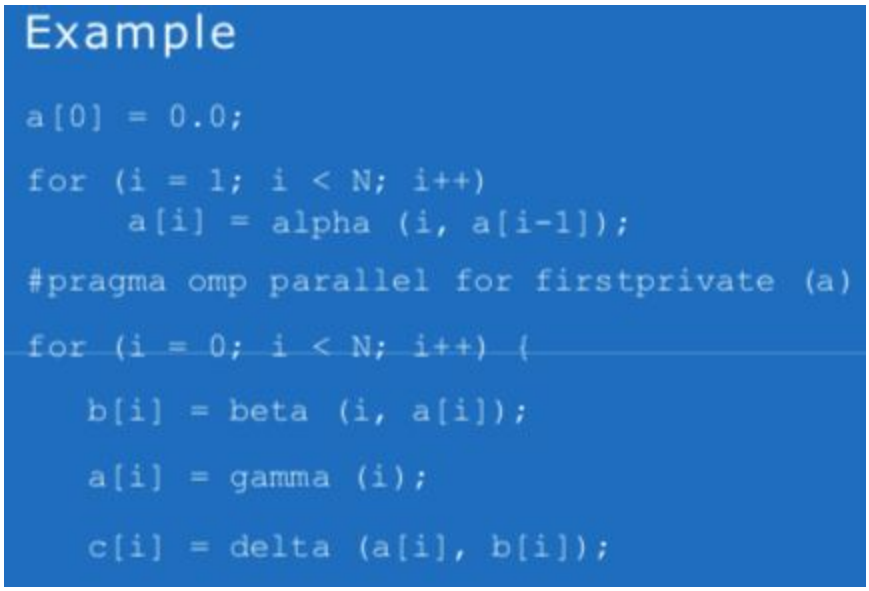
  -- each thread has their own i - for loop iterator auto assumed to be private
  but j is assumed to be shared if not explicitly specified - j being shared prevents program for executing as desired WHY THO? - in this case if they are all at the same on the same j it doesn't matter - because some may finish earlier

  E.g. 2
  int i;
  float *a, *b, *c, temp;
  ...
  #pragma omp parallel for private (tmp)
  for (i = 0; i < N; i++){
      tmp = a[i]/b[i];
      c[i] = tmp * tmp;
} -- fixes issues that would

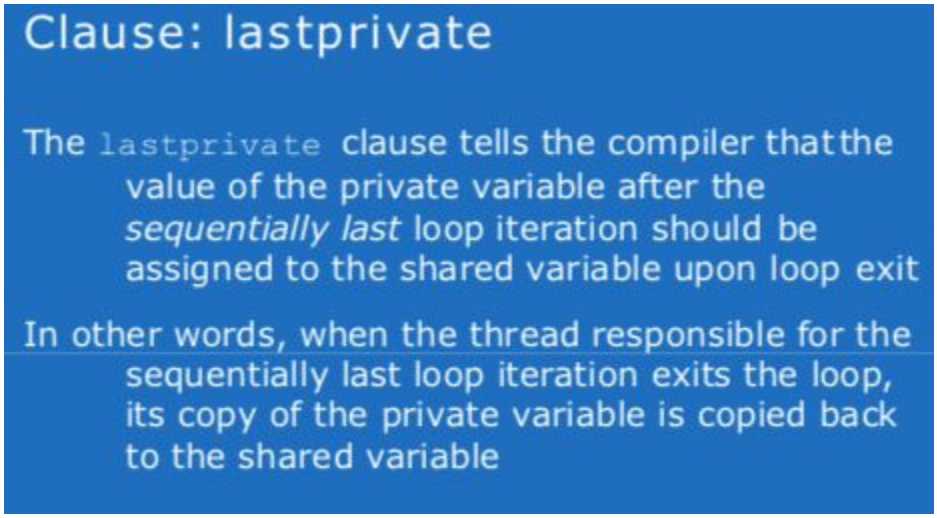
- More about pvt variables:
  -- Each thread has its own copy of the private variables
  -- if j is declared pvt, then inside the for loop no thread can access the "other" j (the j in shared memory)
  -- No thread can assign a new value to the shared j

-- pvt vars undefined at loop entry/exit unless otherwise defined (reduces execution time)
WHY DOES THIS HELP AT LOOP EXIT?

- Clause:firstprivate:
  -- pvt var should inherit the value of shared var upon loop entry
  -- value is assigned once per thread, not once per iteration



- Clause:lastprivate
  -- tells the compiler that the value of the pvt variable after the sequentially last loop (not the last loop to finish but the nth iteration according to the og loop_, should be assigned to the shared var upon loop exit
  -- basically at end of parallelization, last iteration value copied back into shared var



- Pragma: parallel
  -- creates a fork, used when a
  -- how many threads created?

- Pragma: for (WorkSharing construct - WHAT EXACTLY DOES THIS MEAN?)
  -- used inside a block of code already marked with parallel pragma
  -- indicates a for loop whose iterations should be divided among the active threads
  -- barrier synchronization of the threads at the end of the for loop
    -- explicitly handled - method to synchronize the for loop threads

- Pragma: single
  -- single pragma is used inside a parallel block of code
  -- tells compiler that only a single thread should execute the statement/block of code following

- Clause: nowait
  -- nowait clause tells compiler --> no need for a barrier synchronization at end of a parallel for loop or single block of code (sometimes you want the threads to wait for the other threads, other times you want them to 'nowait')

Solution: parallel, for, single Pragma

```
#pragma omp parallel
{
   #pragma omp for nowait
   for (i = 0; i < N; i++)
      a[i] = alpha(i);
   #pragma omp single nowait
   if (delta < 0.0) printf ("delta < 0.0\n");
   #pragma omp for
   for (i = 0; i < N; i++)
      b[i] = beta (i, delta);
}
```

- a going to be a shared var, i going to be pvt
(no wait means a thread that finishes early can continue ahead)
only one thread will execute the statement, all other threads can move ahead
since no nowait, hence there is a synchronization before end of clause reached

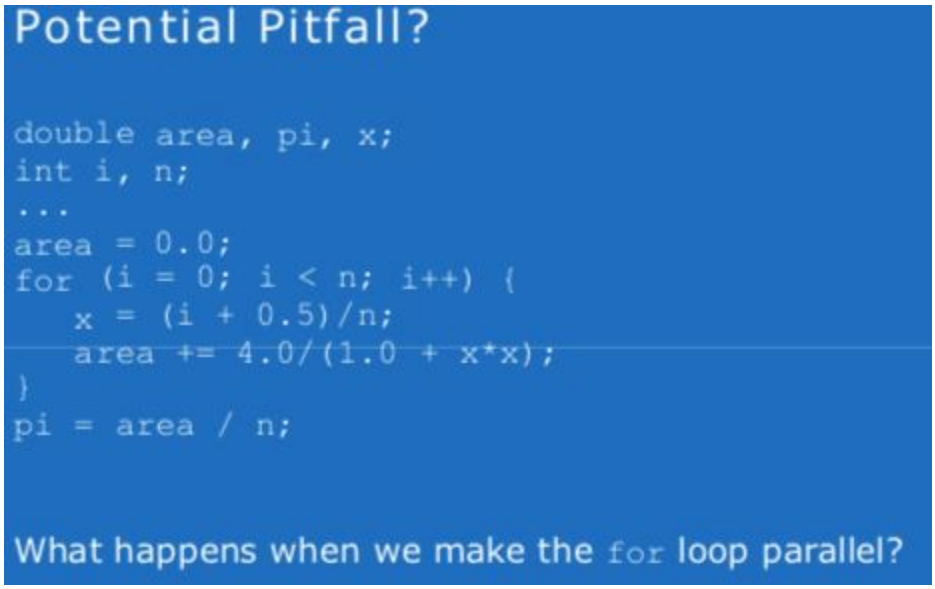
Extended Example

```
#pragma omp parallel private (i, j, low, high)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      #pragma omp single nowait
      printf ("Exiting during iteration %d\n", i);
      break;
   }
   #pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] += alpha (i, j);
}
```

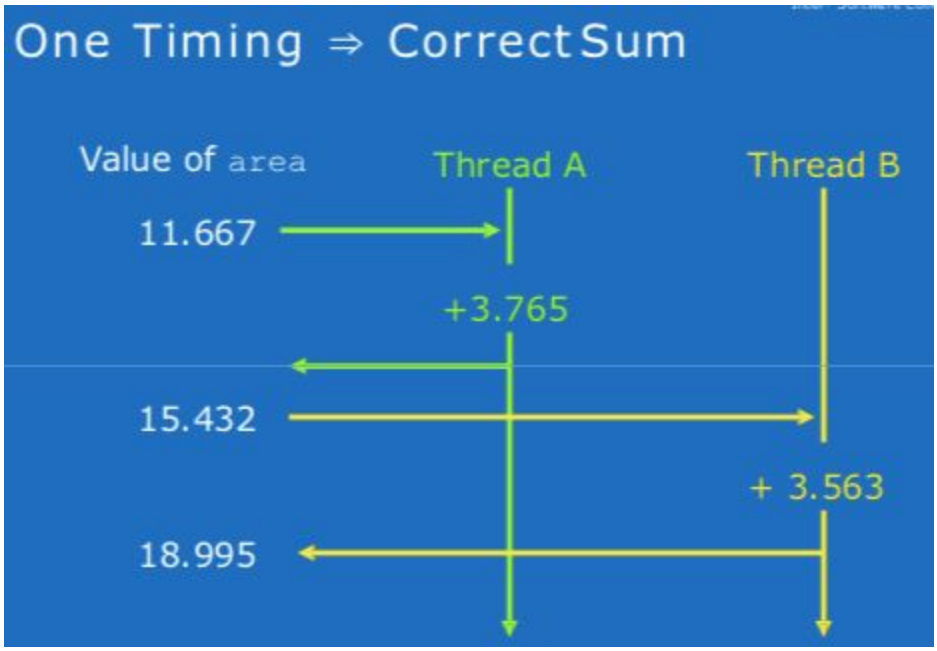
-- Threads created at start of block of code
-- All individually execute the for loop with i, j, low and high being pvt variables
-- The single no wait ensures that printf only done once but control flow remains the same
This is all unnecessary but the benefit is seen due to reduction of repeated creation of threads

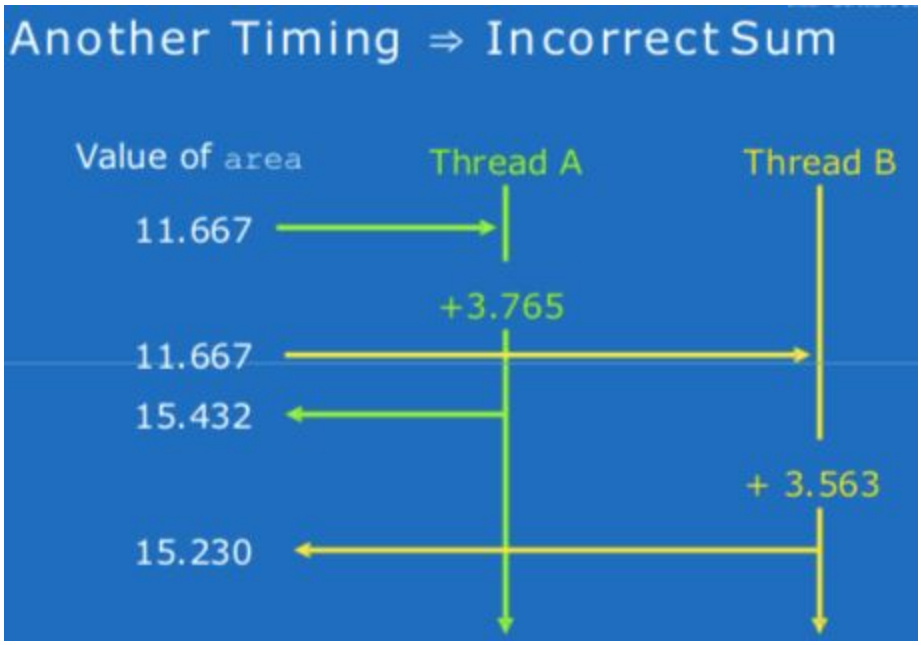
hence reduced overhead?
-- The threads will work share the last for loop

- Race Condition: Possible Issue

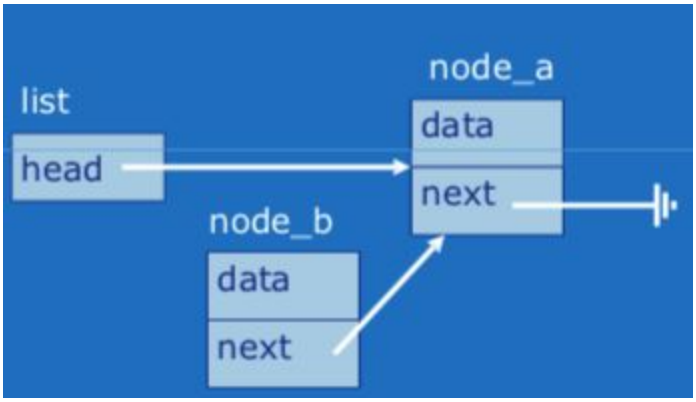


-- Race condition - non deterministic behaviour caused by times at which two or more threads access a shared variable
-- E.g. both thread A and thread B executing area += line



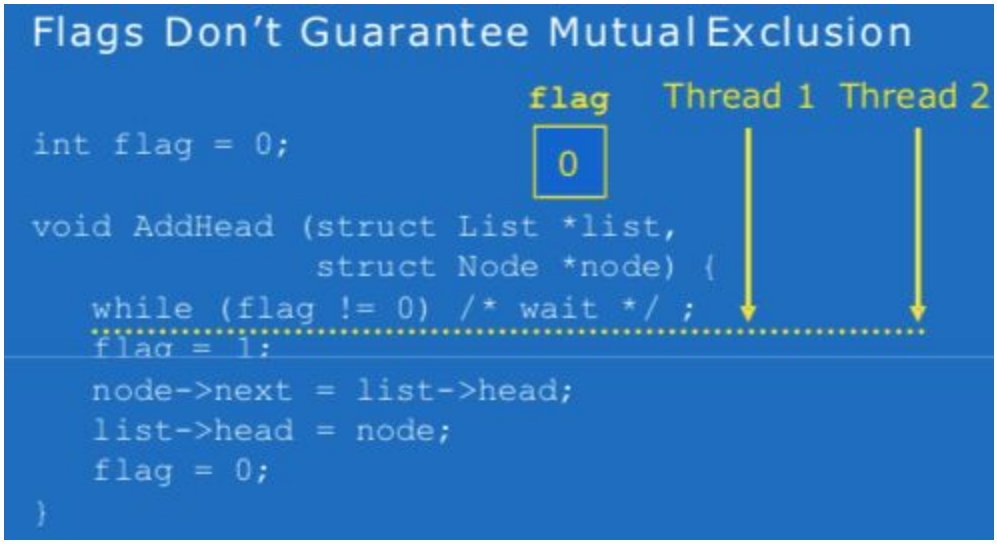


-- Another race condition example
-- Adding a node at the top of a linked list



-- Difficulty of debugging race conditions:
-- Exhibit non deterministic behavior
-- Sometimes give correct/incorrect
-- Usually work with trivial data sets, and small number of threads
-- Errors manifest with more threads or longer execution time

-- Mutual Exclusion - helps prevent race conditions
-- type of synchronization that allows only a single thread or process at a time to have access to a shared resource
-- implemented using some form of locking x



-- both threads could reach the code segment at the same time

-- Locking Mechanism
-- previous method failed because checking the value of flag and setting its value were two distinct operations
-- need some sort of atomic test-and-set - operation that cannot be broken further down
-- OS provides functions to do this
-- generic term "lock" used to control access to a shared resource

OpenMP
#pragma omp parallel
causes a forking into the number of threads specified

Work-sharing construct
For - threads will split up in the

single - forces only one thread to execute the command - the first one, the rest don't get to
critical - one at a time, but all eventually get access - e.g. adding to an accumulator - method of locking code (inefficient)
compared to locking data

sections - defines a region where task based parallelism will exist
within the the sections command you will have #pragma omp section
to mark section of code that one thread will execute

reduction (+: sum)
after parallelism is done, the private sum for each thread will be summed efficiently
(because + is specified as the operator to be applied)

Synchronization techniques:
- barrier -- wait for all the other threads to be done

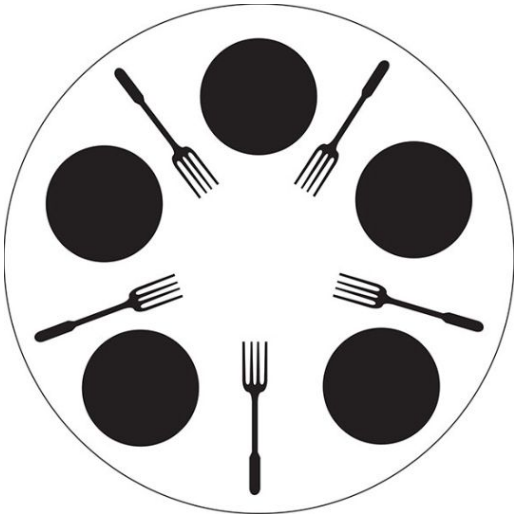
Clauses:
- schedule () - how the work is split up amongst the threads

Problem is load balancing
-- static - breaks up the thread as evenly as possible - in case of for loop as equal a division of the number of iterations as possible
-- dynamic - set a chunk size, and then threads keep picking up work of those chunks as they finish the work they were allocated
higher communication overhead as work is communicated multiple times
-- trade-off between small & big granularity
-- small granularity - better load balancing - worse communication overhead
-- large granularity - worse load balancing - better communication overhead
-- might be better:
-- when the iterations are heterogenous
-- heterogeneous cores - even identical iterations may take varying time spans on different cores
-- one of the cores may be doing OS work
-- guided
-- starts off with big chunk sizes and then starts breaking the work at a smaller granularity exponentially

Four conditions for deadlock:
-- cyclic dependency - easiest thing to break
-- resource locked until the action is completely
-- mutually exclusive access to a resource
-- all threads are identical - no one thread can take a resource away from another

Problems with locks:
-- every call to lock should be matched with unlock
-- may compile without this ^^^
-- programmer may forget unlock or incorrect arg for unlock
-- thread that never releases a shared resource creates a deadlock

Dining philosophers problem (deadlock)

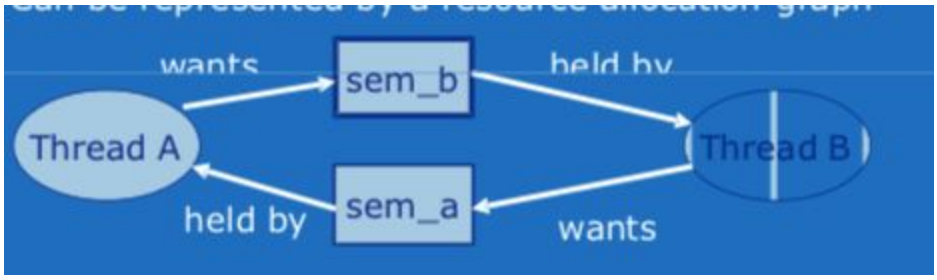


hierarchy of forks

Faulty Implementation

What happens if threads are at this point at the same time?

Thread A	Thread B
`lock (lock_a);`	`lock (lock_b);`
`a += 5;`	`b += 5;`
`lock (lock_b);`	`lock (lock_a);`
`b += 7;`	`a += 7;`
`a += b;`	`a += b;`
`unlock (lock_b);`	`unlock (lock_a);`
`a += 11;`	`b += 11;`
`unlock (lock_a);`	`unlock (lock_b);`

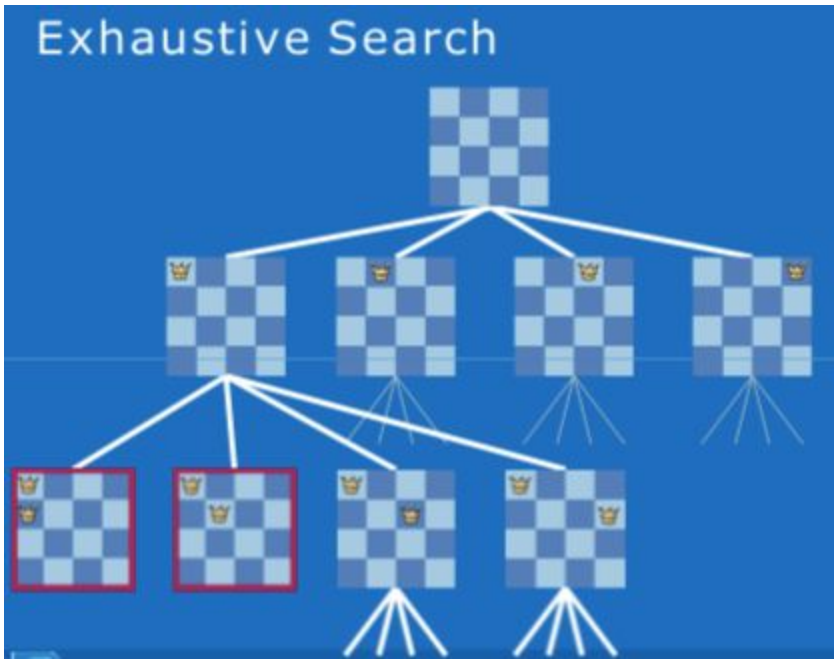


Correct Implementation

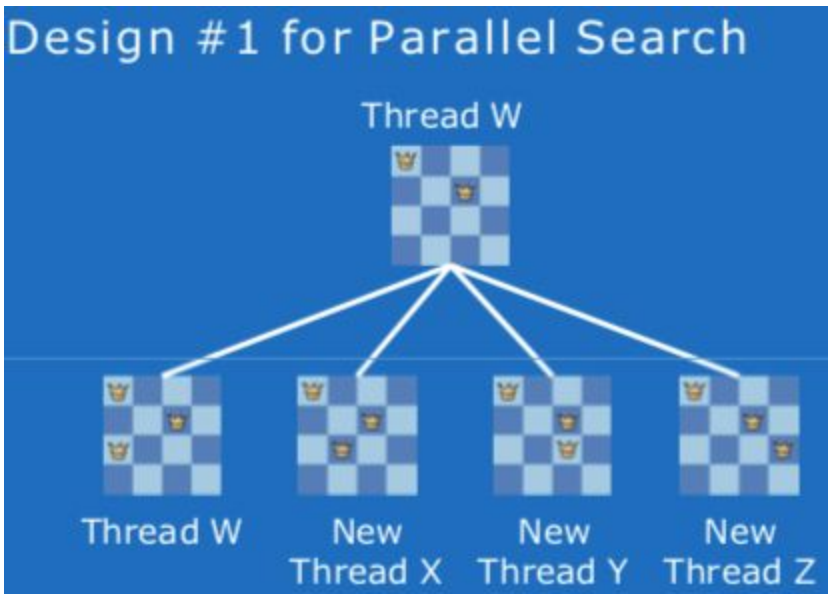
Threads must lock lock_a before lock_b

Thread A	Thread B
`lock (lock_a);`	`lock (lock_a);`
`a += 5;`	`lock (lock_b);`
`lock (lock_b);`	`b += 5;`
`b += 7;`	`a += 7;`
`a += b;`	`a += b;`
`unlock (lock_b);`	`unlock (lock_a);`
`a += 11;`	`b += 11;`
`unlock (lock_a);`	`unlock (lock_b);`

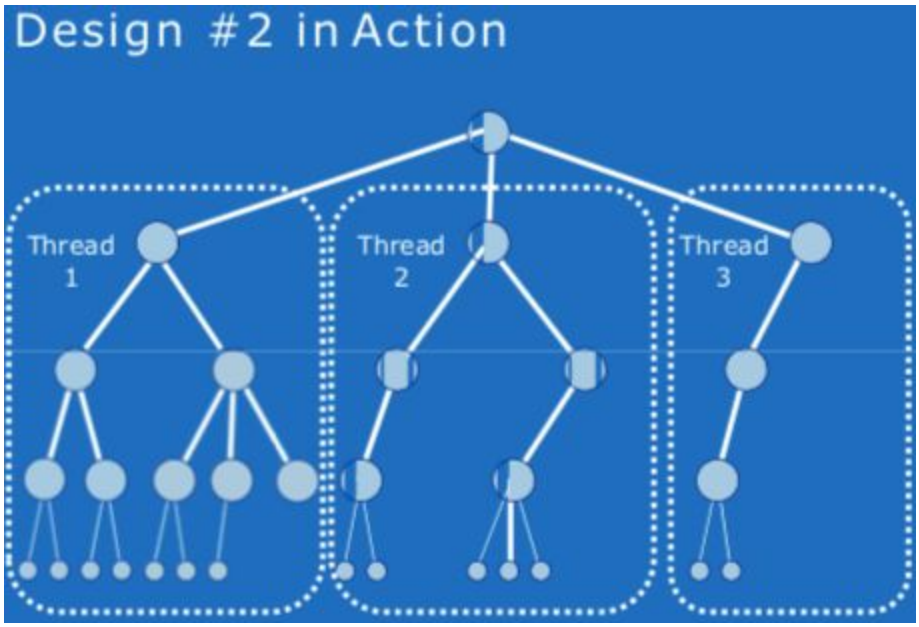
- N Queens Problem



Possible Parallelism Designs:

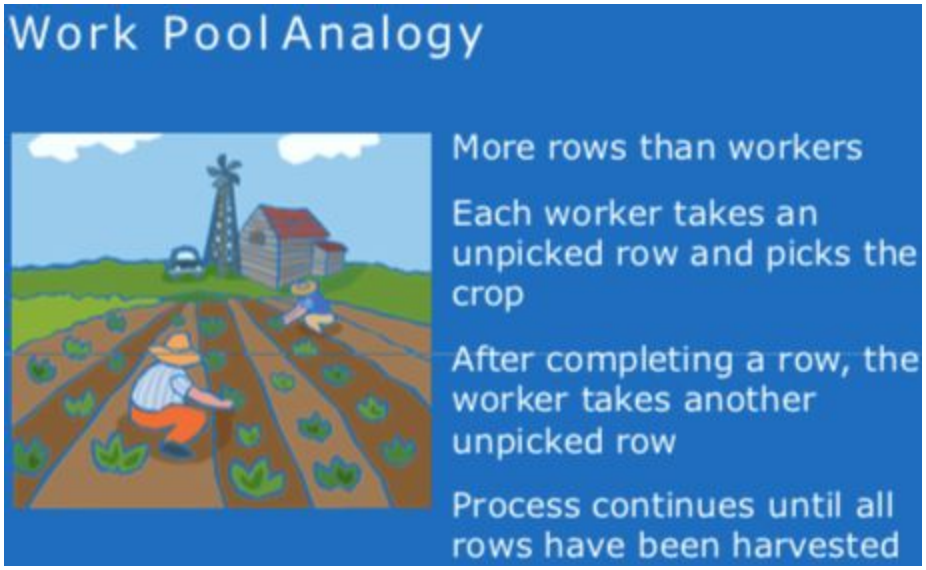


1. Create threads to explore different parts of the search tree simultaneously
   -- if node has children
      -- the og thread takes on child and creates new threads for the other children
   Pros:
      -- Easy to implement
      -- Good load balancing
   Cons:
      -- Too many threads created
      -- Too much overhead as a result



2. One thread for every subtree at a certain depth
   Each thread sequentially explores its subtree

   Pros:
   -- thread creation/termination minimized
   Cons:
   -- subtree sizes may vary
   -- some threads may finish before others
   -- imbalanced workload lowers efficiency



3. Main thread creates work pool - list of subtrees to explore
   Finite number of co-threads created
   Each thread finishes its subtree
   After its done, go to pool to get more work
   (Fewer workers than rows in a farm analogy)

   Pros:
   -- Thread creation/termination time minimized
   -- better workload balancing than strategy #2

Cons:

-- Thread need exclusive access to the data structure containing work to be done (a sequential component)

-- worse load balancing than #1

Hence a good compromise

Parallel Program Design:

-- One thread creates list of partially filled-in boards

-- Fork: Create one thread per CPU

-- Each thread repeatedly gets board from list, searches

-- for solutions, and adds to solution count, until

-- no more board on list

-- Join: Occurs when list is empty

-- One thread prints number of solutions found

Insertion of OpenMP Code

```
struct board *stack;
...
stack = NULL;
for (i = 0; i < n; i++) {
  initial=(struct board *)malloc(sizeof(struct board));
  initial->pieces = 1;
  initial->places[0] = i;
  initial->next = stack;
  stack = initial
}
num_solutions = 0;
omp_set_num_threads (omp_get_num_procs());
#pragma omp parallel
search_for_solutions (n, stack, &num_solutions);
printf ("The %d-queens puzzle has %d solutions\n", n,
          num_solutions);
```

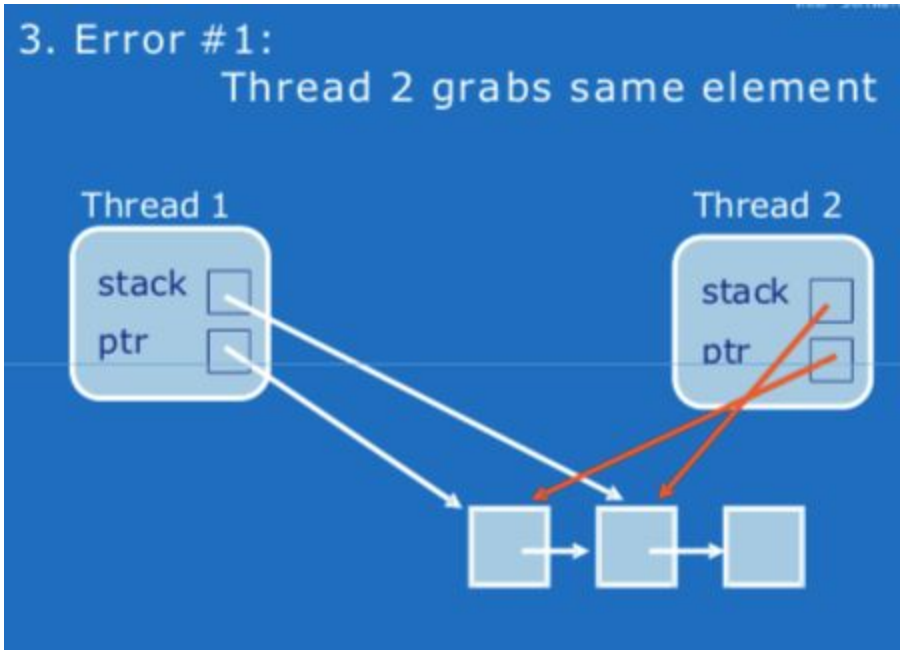
C/OpenMP Function to Get Work

```
void search_for_solutions (int n,
   struct board *stack, int *num_solutions)
{
   struct board *ptr;
   void search (int, struct board *, int *);

   while (stack != NULL) {
     #pragma omp critical
     { ptr = stack; stack = stack->next; }
     search (n, ptr, num_solutions);
     free (ptr);
   }
}
```

C/OpenMP Search Function

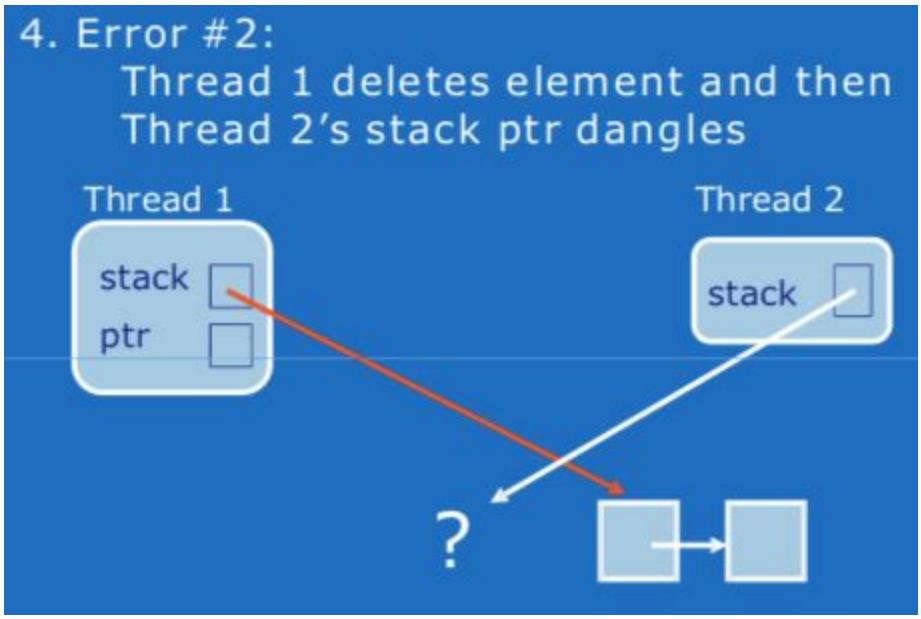
```
void search (int n, struct board *ptr,
             int *num_solutions)
{
   int i;
   int no_threats (struct board *);

   if (ptr->pieces == n) {
      #pragma omp critical
      (*num_solutions)++;
   } else {
      ptr->pieces++;
      for (i = 0; i < n; i++) {
         ptr->places[ptr->pieces-1] = i;
         if (no_threats(ptr))
            search (n, ptr, num_solutions);
      }
      ptr->pieces--;
   }
}
```

3. Error #1:
Thread 2 grabs same element



(Diagram unclear)

4. Error #2:
Thread 1 deletes element and then Thread 2's stack ptr dangles



Solutions:

1. Make the stack static
2. Use indirection a pointer to the stack pointer

Corrected main Function

```
struct board *stack;
...
stack = NULL;
for (i = 0; i < n; i++) {
  initial=(struct board *)malloc(sizeof(struct board));
  initial->pieces = 1;
  initial->places[0] = i;
  initial->next = stack;
  stack = initial
}
num_solutions = 0;
omp_set_num_threads (omp_get_num_procs());
#pragma omp parallel
search_for_solutions (n, &stack, &num_solutions);
printf ("The %d-queens puzzle has %d solutions\n", n,
          num_solutions);
```

Corrected Stack Access Function

```
void search_for_solutions (int n,
   struct board **stack, int *num_solutions)
{
   struct board *ptr;
   void search (int, struct board *, int *);

   while (*stack != NULL) {
     #pragma omp critical
     { ptr = *stack;
       *stack = (*stack)->next; }
     search (n, ptr, num_solutions);
     free (ptr);
   }
}
```

Fancy Web Page Example:

Pseudocode, Option A

-- Implementing Task Decompositions
-- Retrieve page
-- Identify links
-- Enter parallel region
  -- Thread gets ID number (id)
  -- If id = 0 draw page
  -- else fetch page & build snapshot image (id-1)
-- Exit parallel region

C/OpenMP Code, Option A

```
page = retrieve_page (url);
find_links (page, &num_links, &link_url);
for (i = 0; i < num_links; i++)
   snapshots[i].image = NULL;
omp_set_num_threads (num_links + 1);
#pragma omp parallel private (id)
{
id = omp_get_thread_num();
if (id == 0) display_page (page);
else generate_preview (&snapshots[id-1]);
}
```

Pseudocode, Option B

-- A Retrieve Page
-- Identify links
-- Two activities happen in parallel
  -- Draw page
  -- For all links do in parallel
    -- Fetch page and build snapshot image

Parallel Sections

```
#pragma omp parallel sections
{
<code block A>
#pragma omp section
<code block B>
#pragma omp section
<code block C>
}
```

Meaning: The following block contains sub-blocks that may execute in parallel

Each block executed by one thread

Dividers between sections

- Nested Parallelism

-- Parallel sections can be used to define the 2 concurrent activities
-- for loop to create multiple snapshots - trying to make this parallel -- hence nested parallelism

OpenMP + Parallelism Class Notes

#pragma omp parallel
causes a forking into the number of threads specified

Work-sharing construct
For - threads will split up in the

single - forces only one thread to execute the command - the first one, the rest don't get to
critical - one at a time, but all eventually get access - e.g. adding to an accumulator

sections - defines a region where task based parallelism will exist
within the the sections command you will have #pragma omp section
to mark section of code that one thread will execute

reduction (+: sum)
after parallelism is done, the private sum for each thread will be summed efficiently
(because + is specified as the operator to be applied)

Synchronization techniques:
- barrier -- wait for all the other threads to be done

Clauses:
- schedule () - how the work is split up amongst the threads
Problem is load balancing
-- static - breaks up the thread as evenly as possible - in case of for loop as equal a division of the number of iterations as possible
-- dynamic - set a chunk size, and then threads keep picking up work of those chunks as they finish the work they were allocated
higher communication overhead as work is communicated multiple times
-- trade-off between small & big granularity
-- small granularity - better load balancing - worse communication overhead
-- large granularity - worse load balancing - better communication overhead
-- might be better:
-- when the iterations are heterogenous
-- heterogeneous cores - even identical iterations may take varying time spans on different cores
-- one of the cores may be doing OS work
-- guided
-- starts off with big chunk sizes and then starts breaking the work at a smaller granularity exponentially

Four conditions for deadlock:
-- cyclic dependency - easiest thing to break
-- resource locked until the action is completely
-- mutually exclusive access to a resource
-- all threads are identical - no one thread can take a resource away from another

ECF

- Control Flow
  -- Processors only read and execute a sequence of instructions
  <startup>
  inst1
  inst2
  inst3
  .....
  instn
  <shutdown>

  -- as time progresses you move from startup to shutdown
  -- branches, procedures calls, other programs can alter it

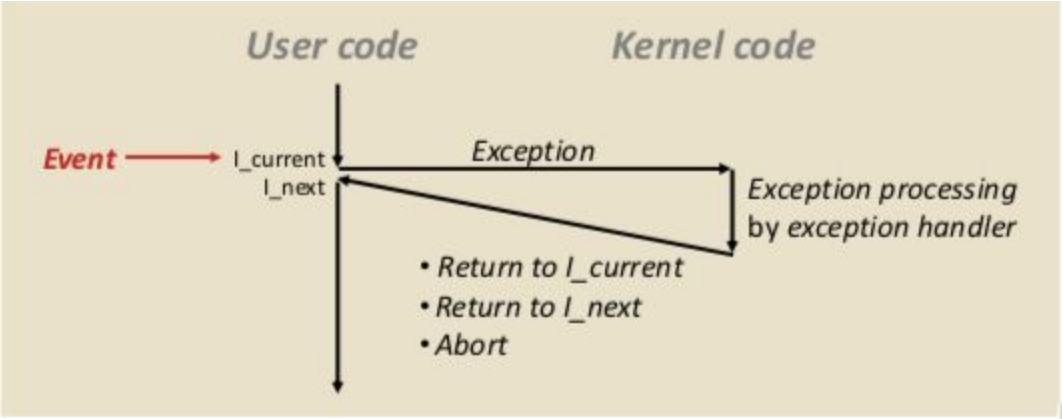
- Altering the Control Flow
  -- React to changes in program state
    -- Jumps and branches
    -- Call and return

  -- Insufficient for a useful system
    (Difficult to react to changes in system state)
    -- Data arrives from a disk or a network adapter
    -- Instruction divides by 0
    -- User hits Ctrl-C at the keyboard
    -- System timer expires

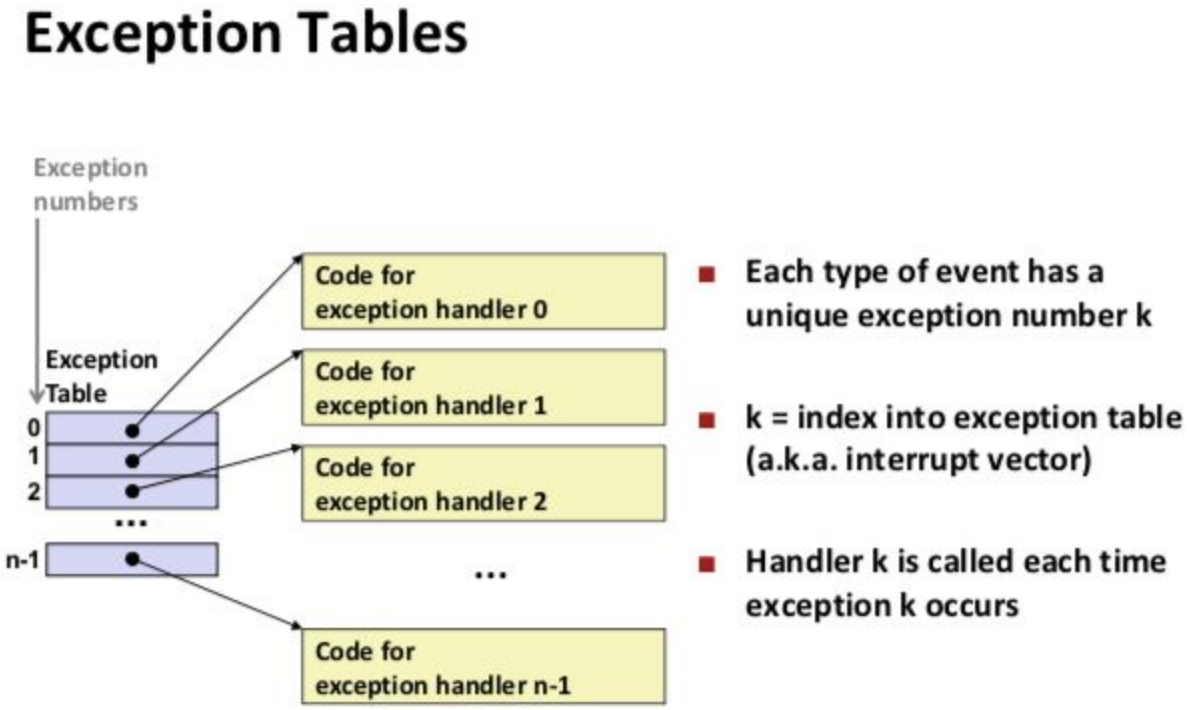
- Exceptional Control Flow
  -- Low level mechanisms
    -- Exceptions
      -- Change in control flow in response to a system event
      -- Implemented using combination of hardware and OS

  -- Higher Level Mechanisms
    -- Process context switch: impl. by OS and hardware timer
    -- Signals: impl. by OS
    -- Non-local jumps: setjmp(), longjmp() - impl by C runtime library

- Exceptions
  -- An exception is a transfer of control to the OS kernel in response to some event
  (change in processor state)
    -- Kernel is the memory-resident part of the OS - has higher privilege than regular code
    -- E.g. divisions by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C



- Exception Tables (much like a jump table)



- Asynchronous Exceptions (Interrupts)
  -- Caused by event external to the processor
    -- Indicated by setting the processor's interrupt pin
    -- Handler returns to the "next" instruction
  -- Examples:
    -- Timer interrupt:
      -- Every few ms, an external timer chip triggers and interrupt
      -- used by kernel to take back control from user programs
    -- I/O interrupt from external device
      -- hitting Ctrl-C at the keyboard
      -- Arrival of a packet from a network
      -- arrival of data from a disk

- Synchronous Exceptions
  -- Caused by events that occur as a result of executing an instruction
    -- Traps - intention - need help from OS typically cause you don't have permission to what you want -- returns to 'next' instruction

-- Examples: system calls, breakpoint traps, special instructions
-- Faults
-- Unintentional but possibly recoverable
-- Examples: Protection Fault: tried to access memory we weren't supposed to,
Segmentation Fault - going to code that you're not supposed to go to in arrays,
Page Faults, Floating Point Exceptions
-- Either re-executes or aborts
-- Aborts
-- Unintentional and unrecoverable - ABORTS CURRENT PROGRAM
-- E.g.: illegal instruction, parity error, machine check

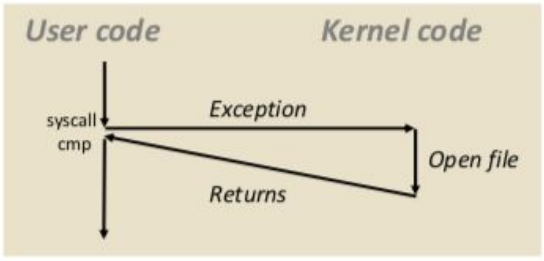
- System Calls - each call has a Unique ID

Number	Name	Description
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

System Call Example: Opening File

- User calls: **`open(filename, options)`**
- Calls **`__open`** function, which invokes system call instruction **`syscall`**

```
00000000000e5d70 <__open>:
...
e5d79:  b8 02 00 00 00     mov  $0x2,%eax  # open is syscall #2
e5d7e:  0f 05              syscall         # Return value in %rax
e5d80:  48 3d 01 f0 ff ff  cmp  $0xfffffffffffff001,%rax
...
e5dfa:  c3                 retq
```



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

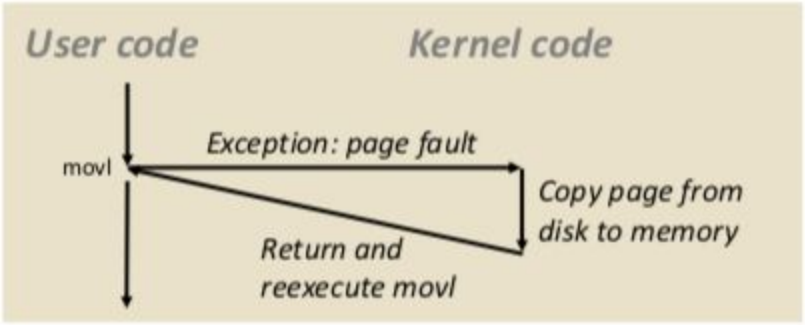
- Page Fault Example
-- If data is not in main memory, but in disk -- is a page fault

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

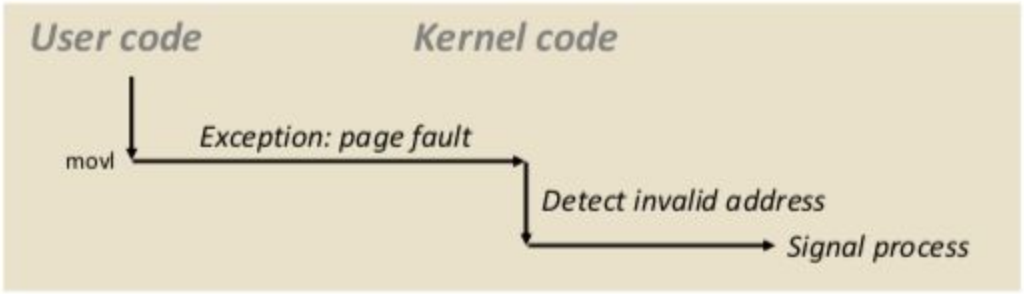
```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



- Sends **SIGSEGV** signal to user process
- User process exits with "segmentation fault"

Linking

-- Example C Program

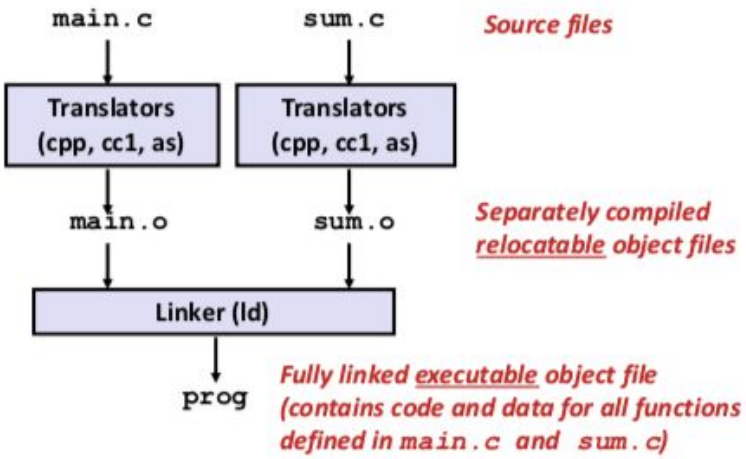
main.c - a global array, a main function, a func prototype for sum

sum.c - implementation of sum

-- Static Linking

Static Linking

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



The compilation is entirely separate

-- Why Linkers?

-- Modularity

-- Programs can be written as a collection of smaller source files

-- can build libraries of common functions e.g. math, std c library

amortize cost of construction of those libraries

-- Efficiency

-- Time: Separate compilation

-- Change on source file, compile and relink

-- no need to recompile other source files

-- Space: Libraries:

-- Common functions can be aggregated into a single file

-- yet executables and running memory images (?) contain only code for the functions they use

-- What Do Linkers Do?

-- Step 1: Symbol resolution

-- Programs define and reference symbols (global variables and functions):

void swap() {...}, swap() // define and reference symbol swap

int *xp = &x // define symbol xp, reference x

-- Symbol defs stored in obj file (by assembler) in symbol table:

-- symbol table is an array of structs

-- each entry includes name, size and location of symbol

-- During symbol resolution, the linker associates each symbol reference with exactly one symbol definition

-- Step 2: Relocation

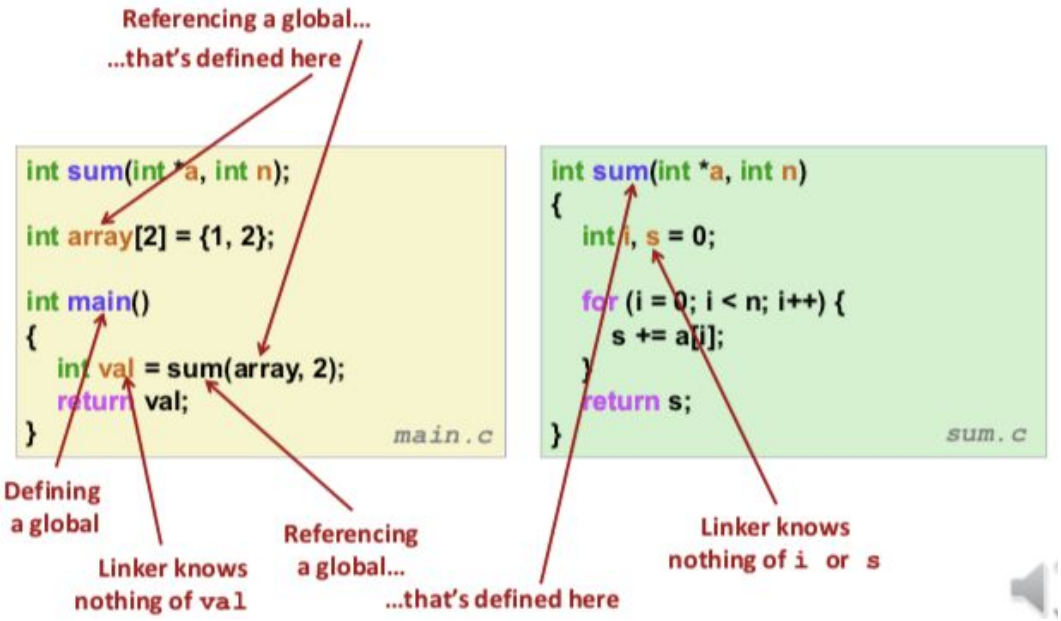
-- Merge separate code and data sections into single sections
-- Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable
-- Updates all references to symbols to reflect their new position

-- Types of modules (object files)
-- Relocatable object file (.o file)
-- Contains code and data in a form that can be combined with other relocatable object files to for executable object file
- Each .o file is produced from exactly one source (.c) file
-- Executable object file (a.out file)
-- Contains code and data in a form that can be copied directly into memory and then executed
-- Shared object file (.so file)
-- Special type of reloc. obj file - can be loaded into memory and linked dynamically
-- at load time or run-time
-- Benefits: by delaying linking into memory, allow for modification to occur till much later without having to recompile all code, update any code without original source (partial updates)
-- Called DLL (Dynamic Linked Libraries) by Windows

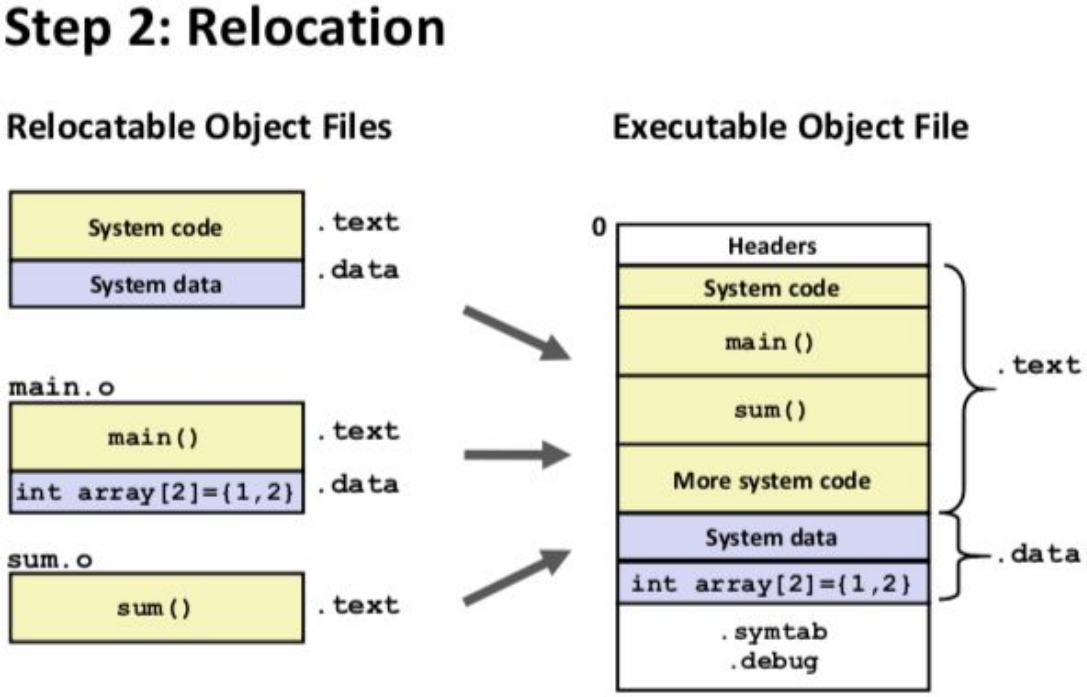
-- Executable and Linkable Format (ELF)
-- Std. binary format for obj files
-- One unified format for all modules
-- Generic name: ELF binaries

-- Linker Symbols:
-- Global Symbols
-- Symbols defined by module m that can be referenced by other modules
-- e.g. non-static c functions and non-static global vars
-- External symbols:
-- Global symbols that are referenced by module m but defined by some other module
-- Local symbols -- symbols defined and referenced exclusively by module m
-- Eg. C functions and global variables defined with the static attribute

-- Example
-- Step 1: Symbol Resolution



-- Step 2: Relocation



-- internal functions declared, initialized and used on stack directly not in global memory

-- Packaging Commonly Used Functions

-- How to package funcs commonly used

-- Awkward, given current linker:

-1- Put all useful funcs into a single source file

-- Programmers link big obj files into their programs

-- Space and time inefficient

-2- Put each func in a separate source file

-- Programmers explicitly link appropriate binaries into their programs

-- More efficient but burdensome on the programmer

-- Old solution: Static Libraries

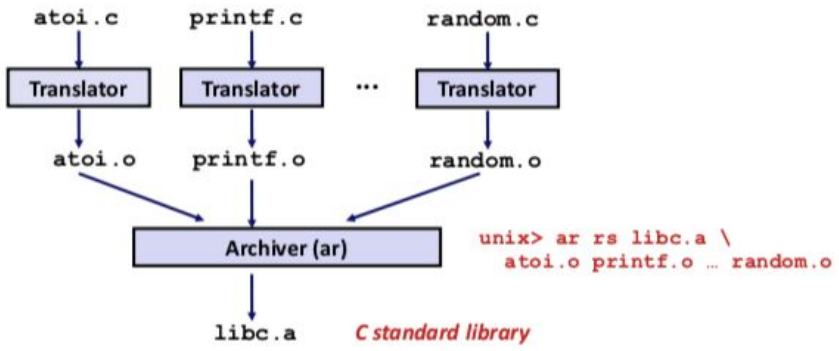
-- Static libraries (.a archive files)

-- Concatenate related reloc obj files into a single with an index (called an archive)

-- Enhance linker so it tries to resolve unresolved external references by searching symbols in or more archives

-- if archive member file resolves reference, link it into the executable

Creating Static Libraries



- **Archiver allows incremental updates**
- **Recompile function that changes and replace .o file in archive.**

Commonly Used Libraries

libc.a (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

libm.a (the C math library)

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, ...)

```
% ar -t libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Example (P.T.O.)

Linking with Static Libraries

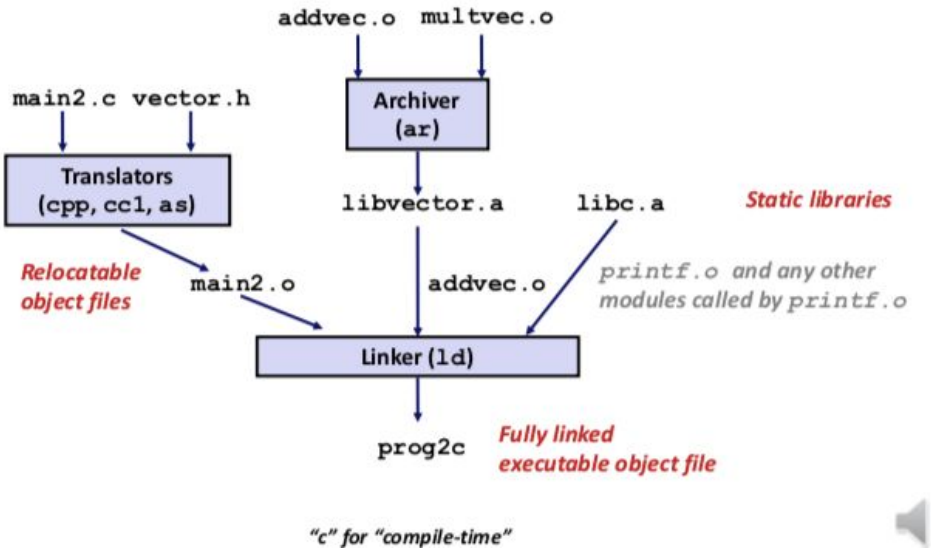
libvector.a

```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
   addvec(x, y, z, 2);
   printf("z = [%d %d]\n",
        z[0], z[1]);
   return 0;
}
```
main2.c

```
void addvec(int *x, int *y,
          int *z, int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}
```
addvec.c

```
void multvec(int *x, int *y,
          int *z, int n)
{
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] * y[i];
}
```
multvec.c

Linking with Static Libraries



-- Using Static Libraries
  -- Linker's algorithm for resolving external references
  Scan .o files for resolving external references
  During scan, keep a list of the current unresolved references
  As each new .o or .a file, obj is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj
  If any entries in the unresolved list at end of scan, then error

  -- Problem occurs if libraries are put first, because when the unresolved symbols are found the program can't check for resolution by remembering what symbols the library had
  -- ALWAYS PUT LIBS AT END for STATIC LINKING

-- Modern Solution: Shared Libraries
  -- Static libraries have the following disadvantages
    -- Duplication in the stored exes (every function needs libc)
    -- Duplication in the running executables
    -- Minor bug fixes of system libs, require each application to explicitly relink
  -- DLLS - Dynamically Linked Libraries - Shared Libs

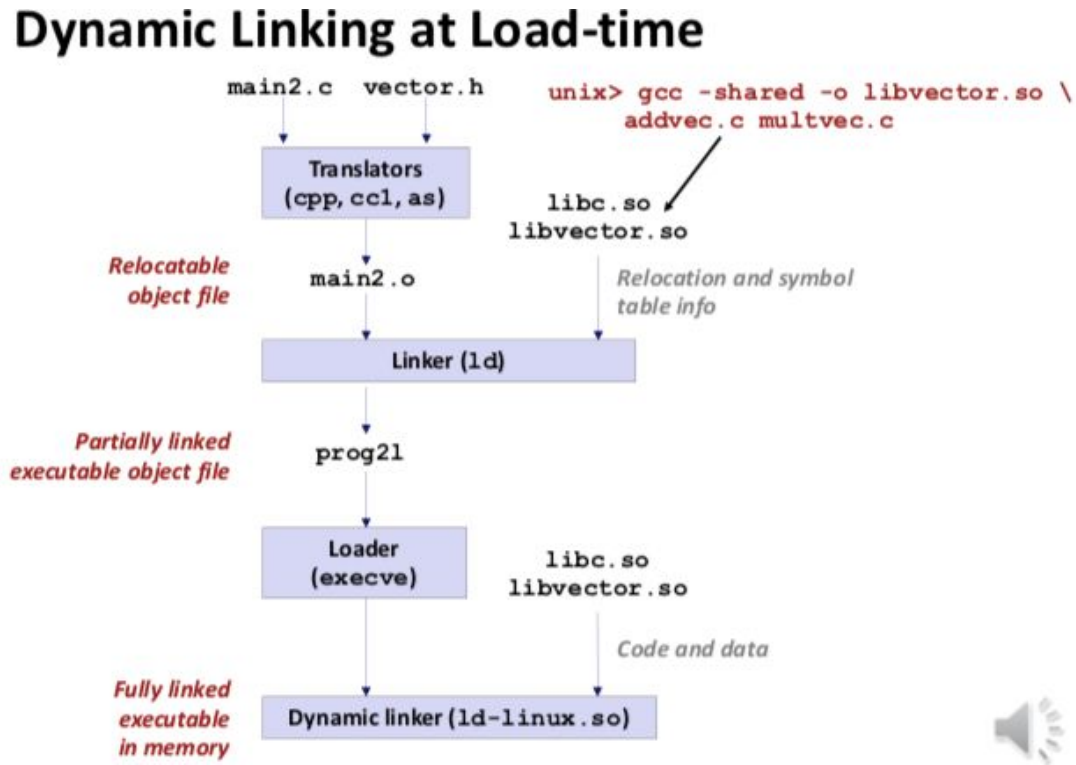
-- Object files that contain code and data are loaded and linked into an application dynamically at either load or run time

-- Dynamic Linking can occur when executable is first loaded and run (load-time linking)

-- Common case for linux, handled automatically by the dynamic linker (ld-linux.so) 00 std libc.so are usually dynamically linked

-- Shared Library routines can be shared by multiple processes

Example of DL at Load-Time



-- Drawback that the system your running on must have the required .so files

-- Linking Summary:

-- Linking is a technique that allows programs to be constructed from multiple obj files

-- Linking can happen at different times in a program's lifetime:

-- compile time

-- load time - when it is loaded into memory

-- run time

-- Case study: Library Interpositioning

-- Library interpositioning: powerful linking technique that allows programmers to intercept call to arbitrary functions -- unclear?

-- Interpositioning - compile time, link time, load/run time

-- Some interpositioning Applications:

-- Security

-- Confinement (Sandboxing) - if you want to prevent actual execution, could interposition and link to stubs

-- Behind the scenes encryption - write wrappers around standard code to encrypt & decrypt

-- Debugging: Facebook example

-- Monitoring and Profiling

-- Count num of calls to functions

-- Characterize call sites and arguments to functions

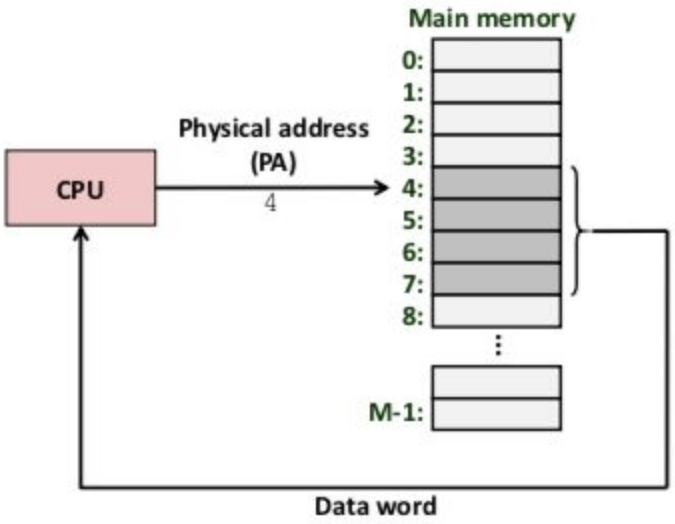
-- Malloc tracing -- (Detecting memory leaks and generating address traces)

-- Instrumentation - interposition functional code to model cache hit & miss behaviour

- snoop all addresses coming out of the program - interpositioning to a cache simulator

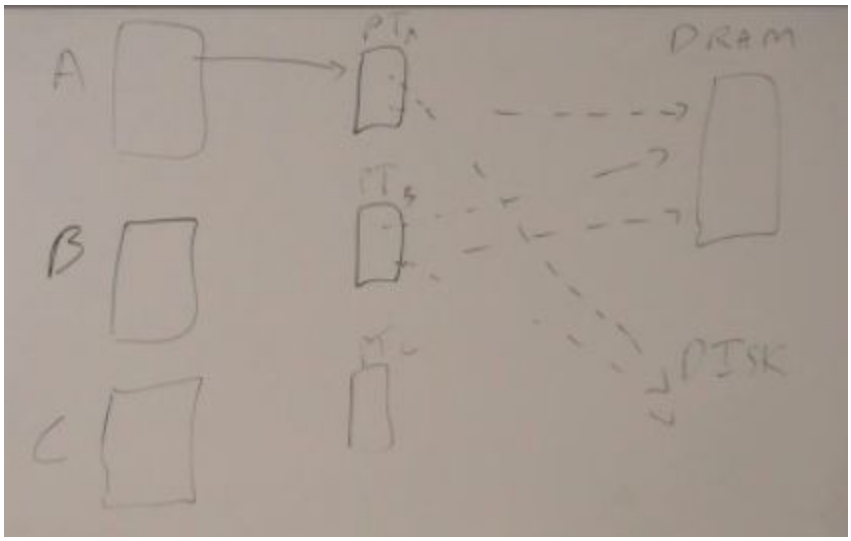
Virtual Memory

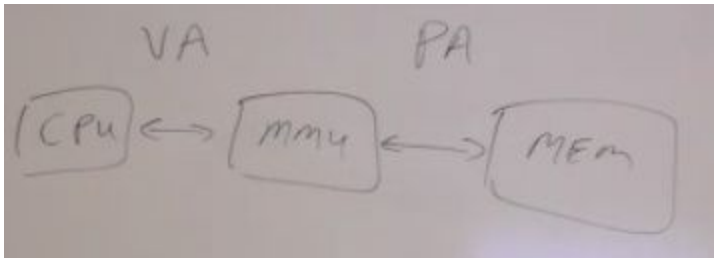
A System Using Physical Addressing



- **Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames**

- Why use virtual memory?
  -- Use main memory efficiently - e.g. static allocation v/s dynamic allocation
  instead of dividing amongst 3 programs the same or even just dividing it,
  lot of memory goes may go unused and more memory intensive
  uses DRAM as cache for parts
  -- Simplified memory management - each process get the same uniform linear address space
  - compiler doesn't need to know where exactly there is free space and it can put, it can just allocate assuming it has all the space
  -- Isolates address space - one process can't interfere with another memory
  - user program cannot access privileged kernel information
  - assuming both prog A and B want to access 0x1024, they won't interfere if they are not supposed to to - and A's 1024 will in reality map to something different in virtual memory (compared to B's 1024), however if C is enabled w data sharing w B, than B & C's 1024 could point to the same real physical memory address e.g. 0x400023

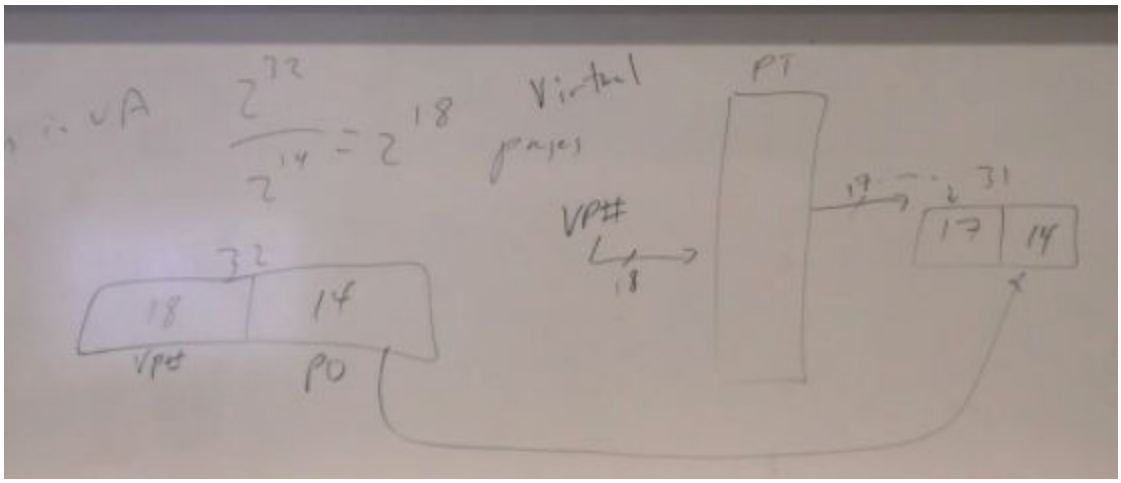




-- Virtual Memory is contiguous - the page table maps it to another contiguous set of memory in physical memory

-- Benefit is that regardless of the DRAM size, the address spaces remain the same from the program's point of view

- Pages Tables:



-- 2^32 = 4GB -- Virtual Address Space

-- if Pages are of size 2^14 B, then 2 ^ 18 pages exist in virtual address space

While the virtual memory is only 1/2 the size of the original memory,
the advantage that it gives is seen when more than one application can use the same 2GB of DRAM - to feel like it has access to the whole memory -- allows for illusion of more space

-- 32 bit address = 18 bit virtual page number + 14 bit page offset goes in to page table

-- page table maps the 18 bit virtual page number to a 17 physical bit page number (not all pages are mapped and valid bit indicates if a virtual page is mapped or not)

-- Page Table lives in the memory

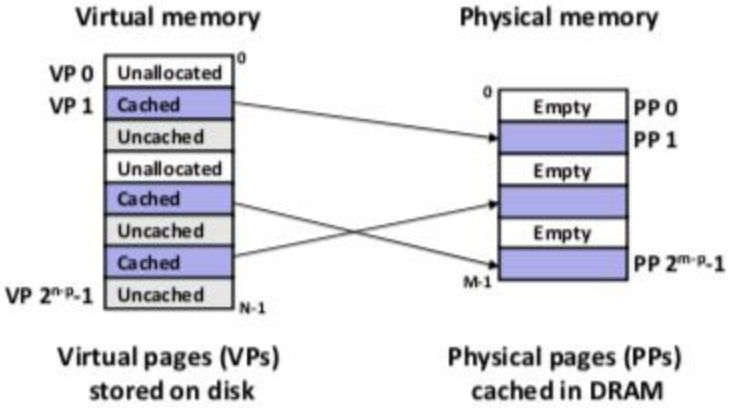
-- Overhead of the Page Table - Quite big but not as big as DRAM Still cause it's 2^18*2^5 which

leaves a lot of space still in the DRAM which is 2^31 in this example

- VM as a tool for Caching

VM as a Tool for Caching

- **Conceptually, *virtual memory* is an array of N contiguous bytes stored on disk.**
- **The contents of the array on disk are cached in *physical memory* (*DRAM cache*)**
  - These cache blocks are called *pages* (size is P = 2^p bytes)



-- virtual memory is an array of N contiguous bytes stored on disk

- DRAM Cache Organization

DRAM Cache Organization

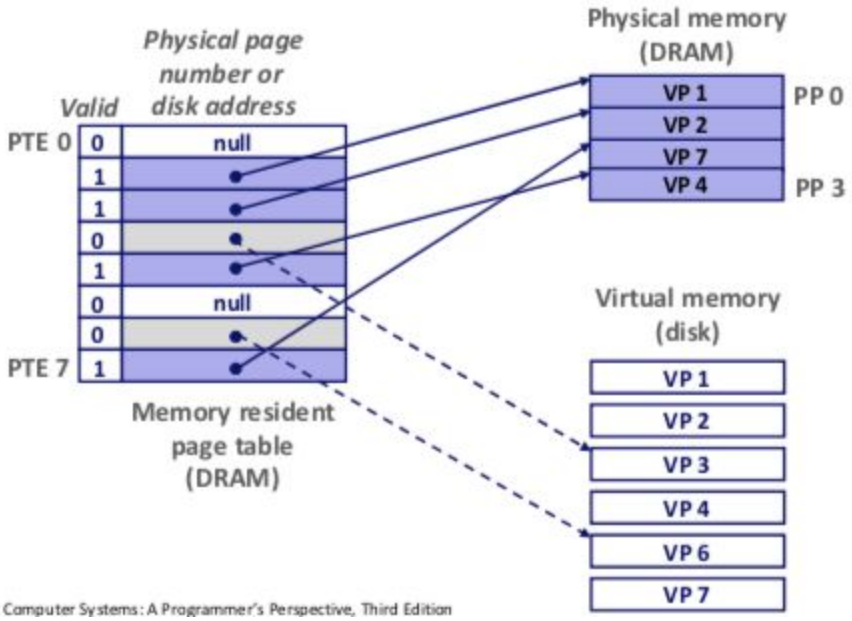
- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM

- **Consequences**
  - Large page (block) size: typically 4 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a "large" mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

- Enabling Data Structure: Page Table
-- A page table is an array of page tables entries that maps virtual pages to physical pages
-- per process kernel data structure

Enabling Data Structure: Page Table

- **A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages.**
  - Per-process kernel data structure in DRAM



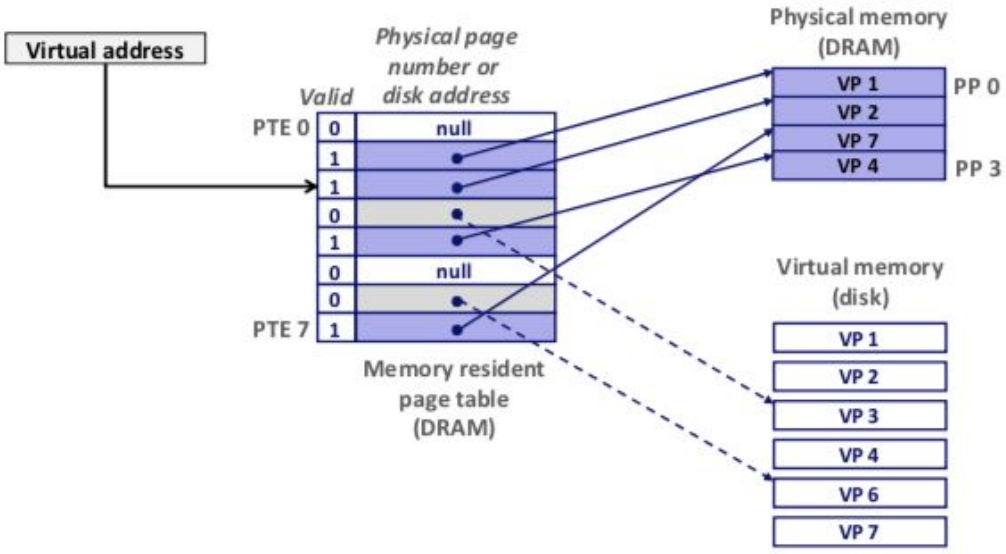
t and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

- Cost of accessing the page table?

-- Page hit - in DRAM cache

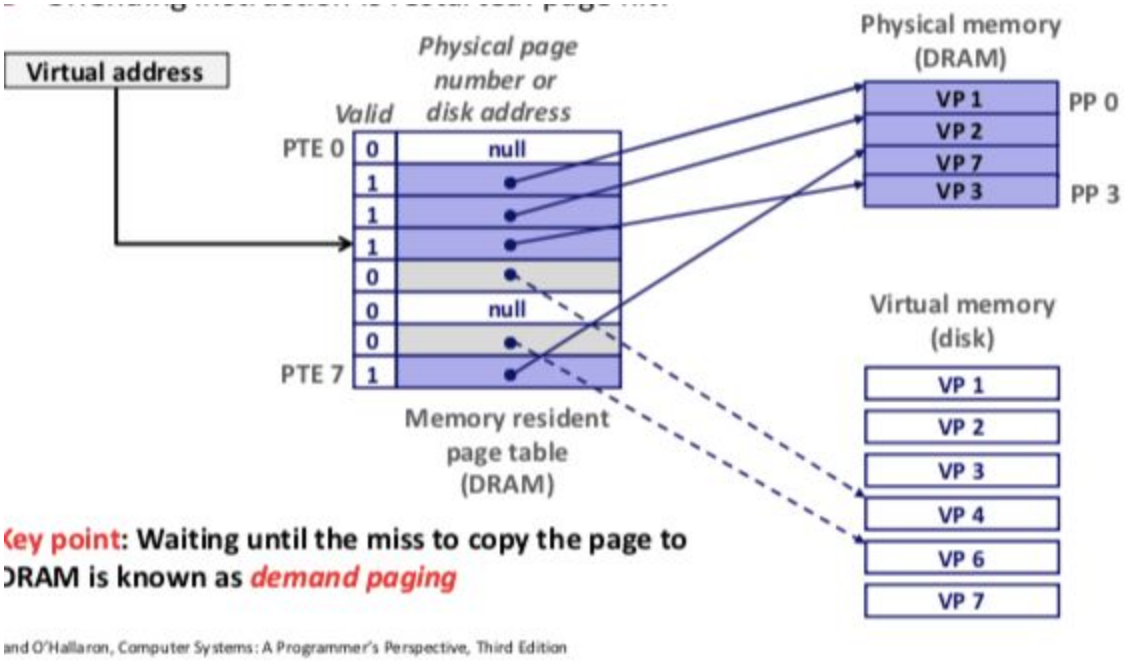
Page Hit

- ***Page hit:* reference to VM word that is in physical memory (DRAM cache hit)**



-- Page fault - not in DRAM cache miss (depends on valid bit)

-- Handling:
  -- Page miss causes page fault (an exception)
  -- page fault handler selects a victim to be evicted and bring in the desired virtual page
  -- go back to the instruction that caused an issue and re run it and now results in a hit
-- Demand paging - waiting until the miss to copy the page to DRAM is a form of this

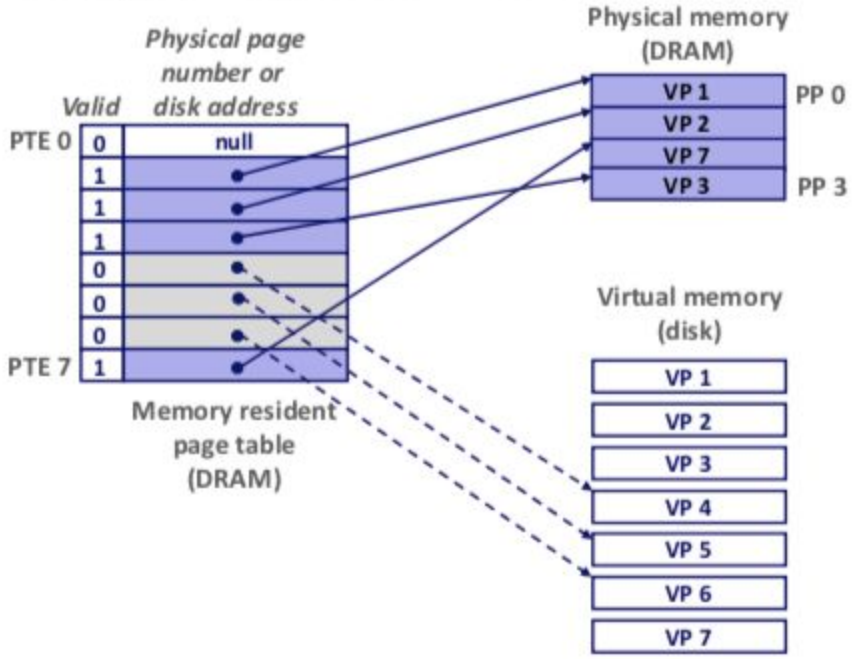


Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



-- Locality

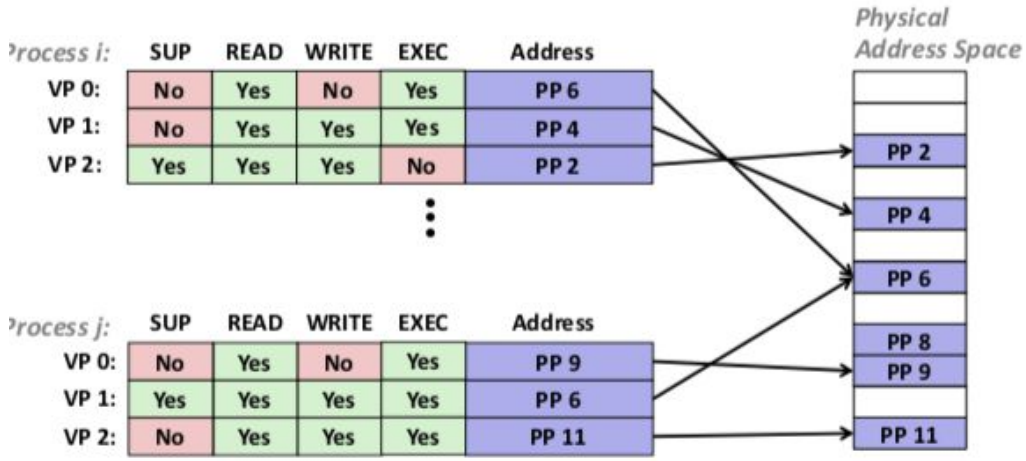
- **Virtual memory seems terribly inefficient, but it works because of locality.**
- **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - Programs with better temporal locality will have smaller working sets
- **If (working set size < main memory size)**
  - Good performance for one process after compulsory misses
- **If (SUM(working set sizes) > main memory size)**
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

- VM as a Tool for Memory Management - physical memory is like a cache of the virtual memory
  -- key idea: each process has its own virtual address space
  -- sees memory as a simple linear array
  -- mapping scatters addresses through physical memory, well chosen mappings improve locality
  -- Simplifying memory allocation - each virtual page can be mapped to any physical page
  one virtual page can be stored in different physical pages at different times
  -- Sharing code and data amongst processes - map virtual pages to same physical page

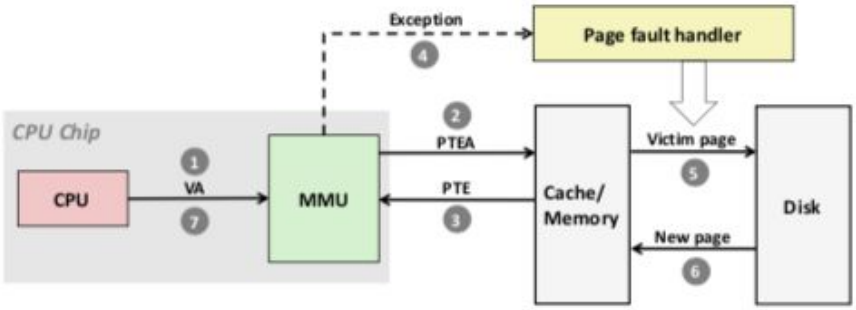
- VM as a tool for Memory Protection:
  -- extend PTEs with permission bits (like execution, reading, writing bits)
  -- MMU checks these bits on each access - helps prevent stack smashing

VM as a Tool for Memory Protection

- **Extend PTEs with permission bits**
- **MMU checks these bits on each access**



Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

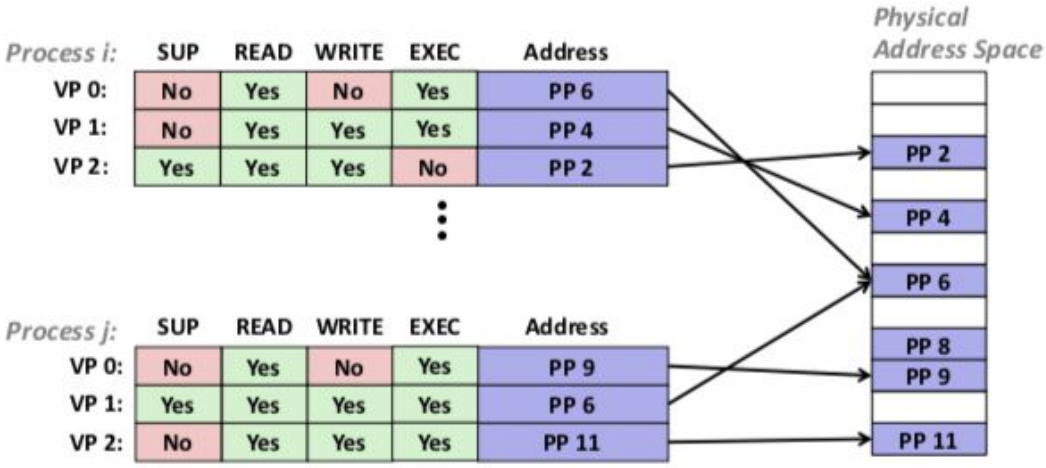
4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim (and, if dirty, pages it out to disk)

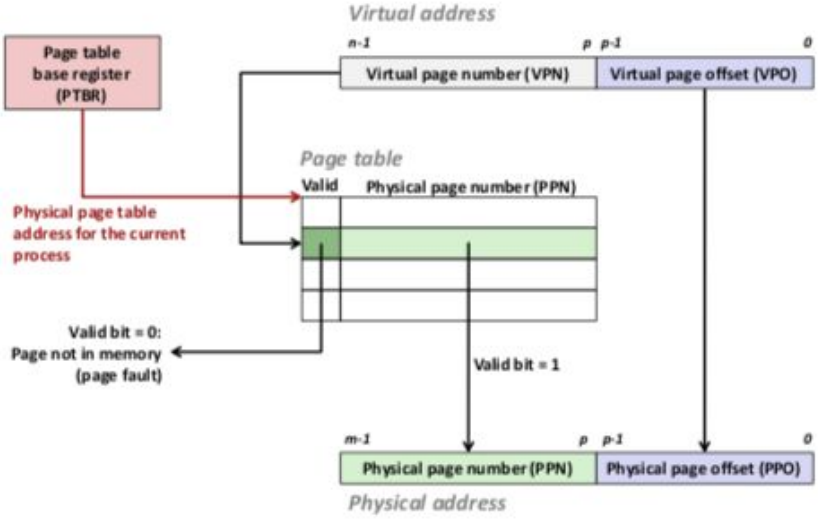
6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

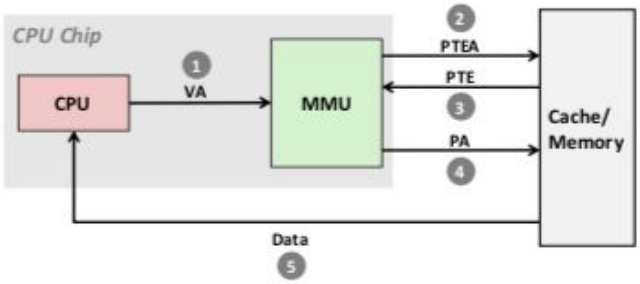
- **Extend PTEs with permission bits**
- **MMU checks these bits on each access**



Address Translation With a Page Table



Address Translation: Page Hit



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

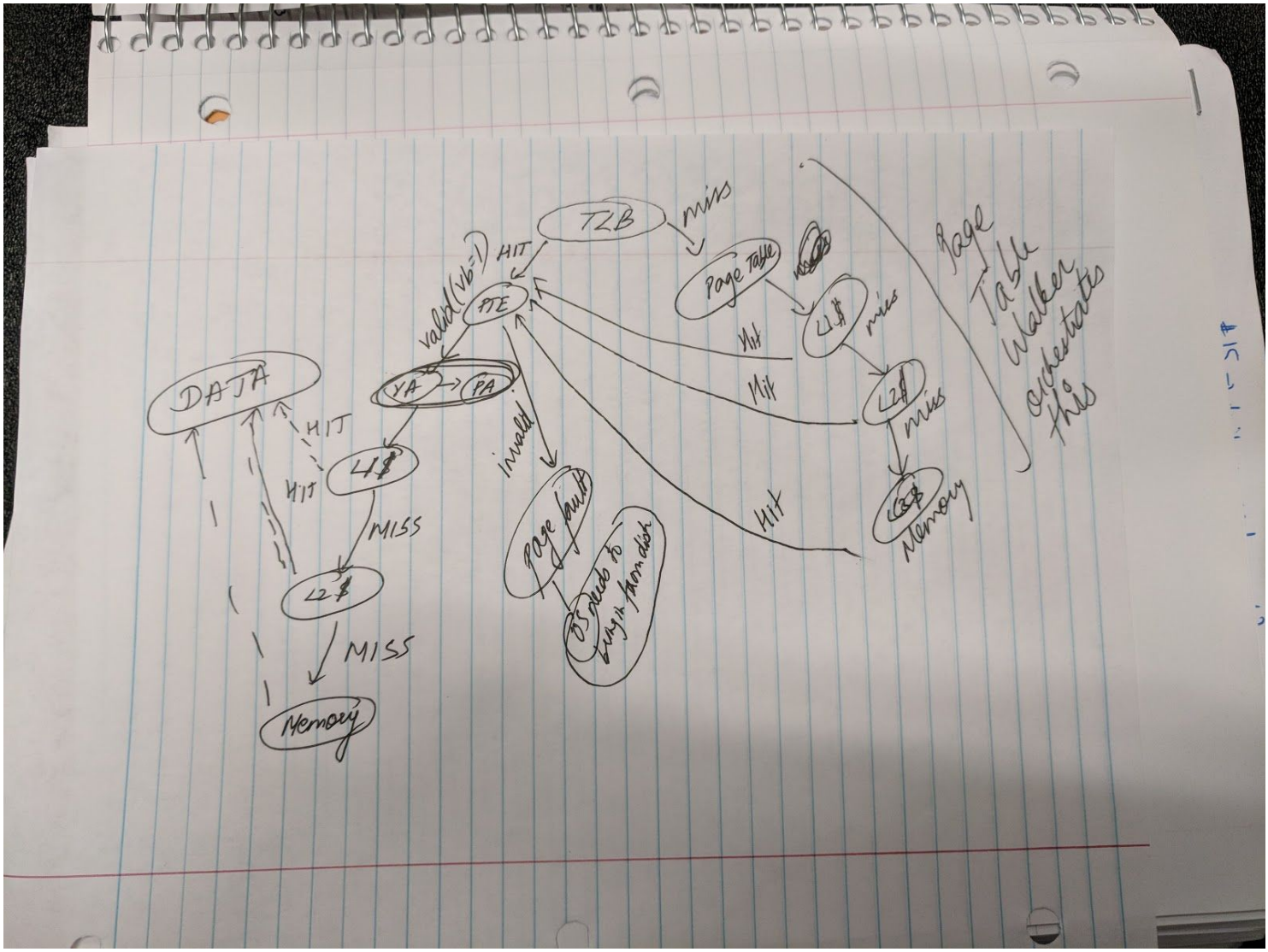
5) Cache/memory sends data word to processor

- Integrating VM and Cache
  -- Page table and contents of that page table could be in the L1 Cache
  -- Problem - start polluting cache with page table entries rather than exploiting spatial locality of program
  -- Hence, TLB - a cache dedicated for the Page Table - valid bits and tag bits and the payload - i.e. the valid bit and the page mapping

Summary of Address Translation Symbols

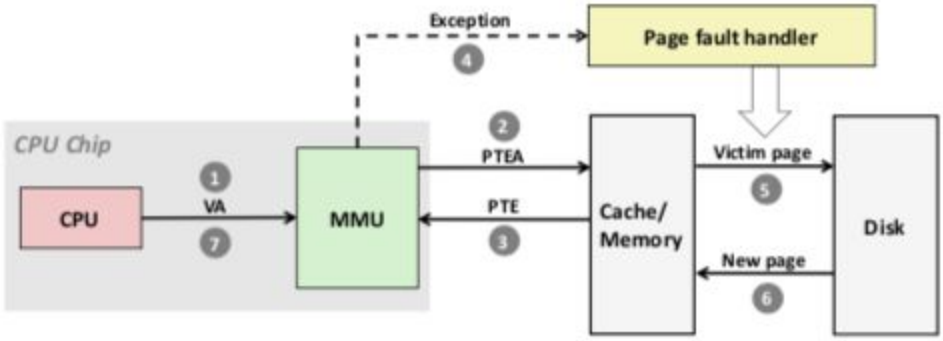
- **Basic Parameters**
  - $N = 2^n$: Number of addresses in virtual address space
  - $M = 2^m$: Number of addresses in physical address space
  - $P = 2^p$: Page size (bytes)
- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number

PTE - Page Table Entry, VA - Virtual Address Space, PA - Physical ..., TLB - Translation LookAside Buffer



- Multi-Level Page Tables:
  -- break up the page table into page size chunks and swap working set into memory

Address Translation: Page Fault

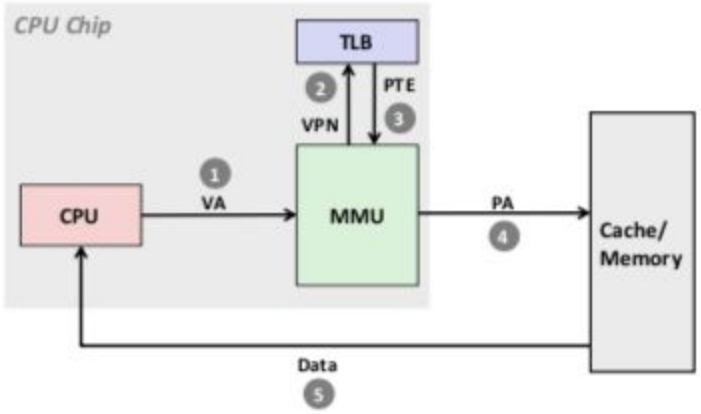


1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

Summary of Address Translation Symbols

- **Basic Parameters**
  - $N = 2^n$: Number of addresses in virtual address space
  - $M = 2^m$: Number of addresses in physical address space
  - $P = 2^p$: Page size (bytes)
- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number

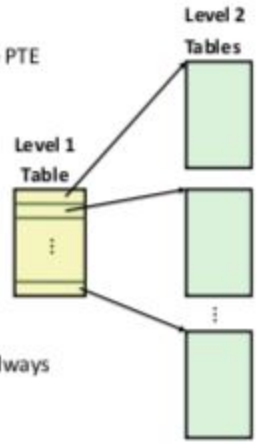
TLB Hit



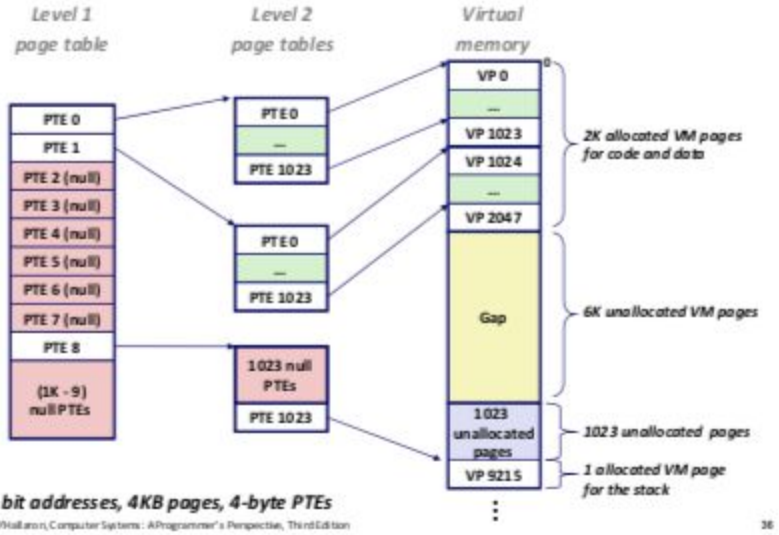
- worst case - could add more memory pressure if you have to go to kth level

Multi-Level Page Tables

- **Suppose:**
  - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- **Problem:**
  - Would need a 512 GB page table!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
- **Common solution: Multi-level page table**
- **Example: 2-level page table**
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)



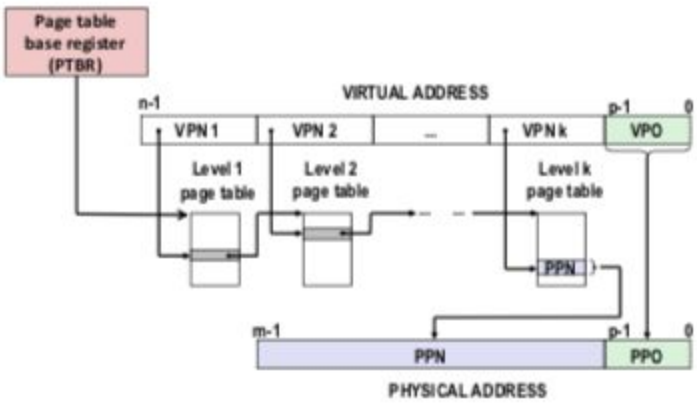
A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs

Usually just make the TLB better - make it a 2 level TLB

Translating with a k-level Page Table

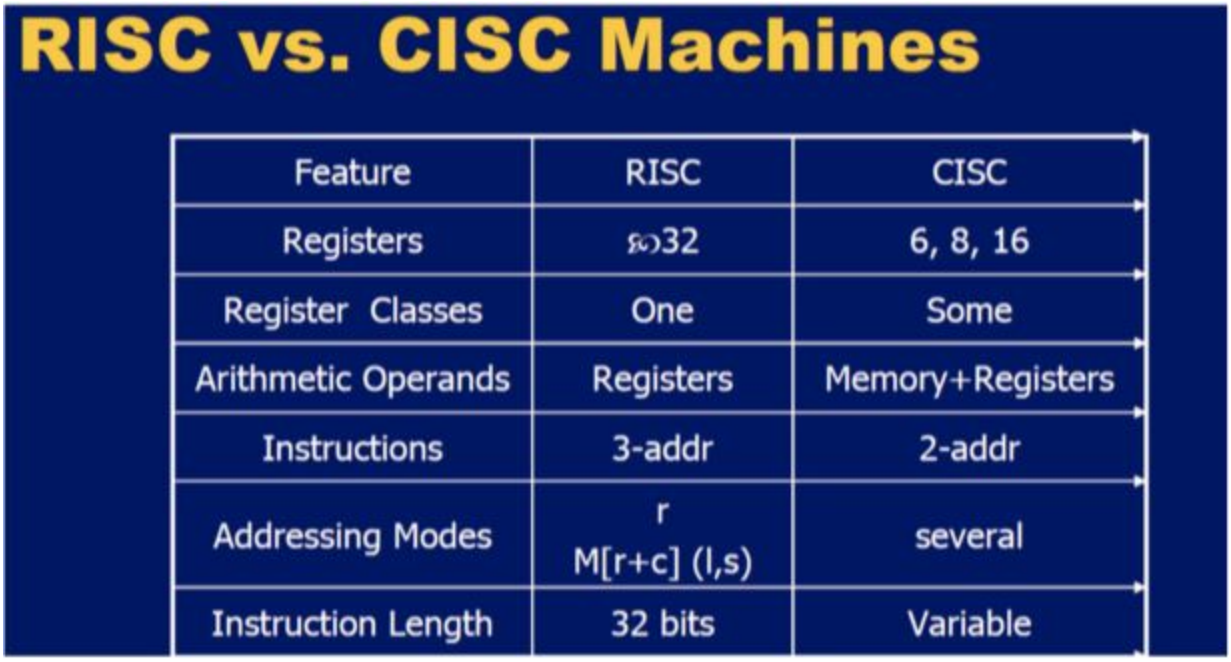


Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Summary

- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes
- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions

MIPS



- Arithmetic Operands: ?
- 3 address code x = y + z (3 addr can be specified in one instruction)
- Short instruction length - better for parallelism and pipelining

- MIPS ia a RISC = Reduced Instruction Set Computer
  -- All arithmetic operations are of form Rd <- Rs op Rt
  -- Importation restriction: MIPS is a load store architecture, ALU can only operate on registers
    -- Different from x86 - memory register architecture, can add something from memory to register in one command
    -- explicit instructions to load from or store in memory

  -- Basic Operations
    -- Arithmetic Operations
    -- Logical Operations
    -- Comparison
    -- Control (branches, jumps etc.)
    -- Memory access (load and store)

    -- All instructions are 32 bit long, hence %rip must always be incremented by 4 (4 bytes long)

- MIPS is a Load-Store Architecture
  -- Every operand must be in a register (with exceptions)
  -- variables must be loaded into registers
  -- Results have to be stored into memory

- Example C fragment...

```
a = b + c;
d = a + b;
```

- ... would be "translated" into something like:

```
Load b into register Rx
Load c into register Ry
Rz <- Rx + Ry
Store Rz into a
Rz <- Rz + Rx
Store Rz into d
```

- MIPS Registers
  -- 32 Bit Registers, $0, $1, -- $31 (aliases exist)
    -- integer arithmetic
    -- address calculations
    -- special-purpose functions defined by convention
    -- temporaries
  -- A 32-bit program counter (PC)
  -- 2 32 bit registers HI & LO specifically used for multiplication and division
    -- increase in precision from operating on 32 bit vals, is fit into HI & LO together
  -- 2 32-bit registers, $f0, $f1 --> used for point arithmetic
  -- Special registers

Register	Name	Function	Comment
$0	zero	Always 0	No-op on write
$1	$at	reserved for assembler	don't use it!
$2-3	$v0-v1	expression eval./function return	
$4-7	$a0-a3	proc/funct call parameters	
$8-15	$t0-t7	volatile temporaries	not saved on call
$16-23	$s0-s7	temporaries (saved across calls)	saved on call
$24-25	$t8-t9	volatile temporaries	not saved on call
$26-27	$k0-k1	reserved kernel/OS	don't use them
$28	$gp	pointer to global data area	
$29	$sp	stack pointer	
$30	$fp	frame pointer	
$31	$ra	proc/funct return address	

  -- Zero register useful - instead of storing 16 bit literal, store 5 bit register value when storing 0 (shorthand way of storing 0)
  -- don't use register once
  -- temporaries? vs volatile temporaries?
  -- pointer to global data area - heap pointer
  -- stack pointer
  -- frame of pointer - stack - top of the stack, bottom of frame (between these 2 = current frame of execution)
  -- $31 vs $1 = address to return to and value to be returned

- MIPS Instruction Types
Rtype - can't access memory
R (Register type) - first 6 bits OPCODE (specifies the type), (Rs, Rd Rt) 5 x 3 bits - Registers, Shift amount - how much shifting to be performed on the destination register, Func - 6bits (specifies what function is to be

used)
I (Instruction type) - first 6 bits OP code (specified the type), (Rs, Rt, Imm) 5 x 2 bits - registers, 16 bits for the immediate
J (Jump type) - first 6 bits OPCode, Imm - 26 bits instructions
-- rd (destination register), rs - source register, rt (source/destination register), immed: 16 bit: value (literal embedded into the actual instruction)

R[ ] -- addresses register file contents, M[ ] - address memory contents
Load word:
R[Rt] = M[R[Rs] + Sign Extended [Immediate]]
- Load and Store Instructions
-- Address is specified as part of instruction in base + displacement
-- Each load or store must specify the memory address to be written or written
-- Because MIPS is always 32 bits long, address must be specified in more compact way
-- Base register used to address memory, instruction specified the register number and a 16-bit signed offset
-- A single base register can be used to 2^32

Load and Store Examples

- Load a word from memory:

```
lw  rt, offset(base)  # rt <- memory[base+offset]
```

- Store a word into memory:

```
sw rt, offset(base)  # memory[base+offset] <- rt
```

- For smaller units (bytes, half-words) only the lower bits of a register are accessible. Also, for loads, you need to specify whether to sign or zero extend the data.

```
lb  rt, offset(base) # rt <- sign-extended byte
lbu rt, offset(base) # rt <- zero-extended byte
sb  rt, offset(base) # store low order byte of rt
```

-- base would be looked up and then the value from there would be used

Arithmetic Instructions

```
Opcode  Operands        Comments

ADD     rd, rs, rt      # rd <- rs + rt
ADDI    rt, rs, immed   # rt <- rs + immed
SUB     rd, rs, rt      # rd <- rs - rt

Examples:

ADD     $8, $8, $10     # r8 <- r9 + r10
ADD     $t0, $t1, $t2   # t0 <- t1 + t2
SUB     $s0, $s0, $s1   # s0 <- s0 - s1
ADDI    $t3, $t4, 5     # t3 <- t4 + 5
```

add/sub dest source1 source2

- Conditional Branches

Flow of Control: Conditional Branches

```
BEQ     rs, rt, target # branch if rs == rt
BNE     rs, rt, target # branch if rs != rt
```

Comparison Between Registers

- What if you want to branch if R6 is greater than R7?
- We can use the SLT instruction:

```
SLT     rd, rs, rt      # if rs<rt then rd <- 1
                        #   else rd <- 0
SLTU    rd, rs, rt      # same, but rs,rt unsigned
```

- Example: Branch to L1 if $5 > $6

```
SLT     $7, $6, $5    # $7 = 1, if $6 < $5
BNE     $7, $0, L1
```

Conditional Operator Rs Rt Target(pc relative address, add 4 to pc + target)
BEQ rs, rt, target #branch is rs == rt
BNE same as above #branch if not equal
SLT rd, rd, rt # if rs < rt then rd set to -1 # else rd set to 0
SLTU rd, rs, rt # same but treats rs and rt as unsigned integers
- Example: branch to L1, if $5 > $6
SLT $7, $6, $5 --> $7 = 1, if $6 < $5
BNE $7, $0, L1 --> branch is $7 not equal to 0 -- equal to 1
(Clarify)

- Jump Instructions

-- J target - #go to specified target (not offset) - 26 bit target, upper bits of PC + 4 concatenated w lower bits
-- JR rs - #jump to address stored in rs
-- Jump and link used for procedure calls
JAL target -- #jump to target, $31 <- PC + 4
JALR rs, rd -- #jump to addr in rs
#rd <- PC + 4
-- When calling a procedure, use JAL; to return JR $31

- Logic Instructions

Logic Instructions

- Used to manipulate bits within words, set up masks, etc.

```
Opcode  Operands        Comments

AND     rd, rs, rt      # rd <- AND(rs, rt)
ANDI    rt, rs, immed   # rt <- AND(rs, immed)
OR      rd, rs, rt
ORI     rt, rs, immed
XOR     rd, rs, rt
XORI    rt, rs, immed
```

- The immediate constant is limited to 16 bits
- To load a constant in the 16 upper bits of a register we use LUI:

```
Opcode  Operands        Comments

LUI     rt, immed       # rt<31,16> <- immed
                        # rt<15,0>  <- 0
```

-- immediate limited to 16 bits
-- LUI rt, immed # Load upper immediate loads the immediate into the upper 16 bits of the register
rest are set to 0

- Pseudoinstructions (available to compiler writer, or assembler etc.)
Not real machine code, makes assembly easier to program

Data moves

Name	Assembly syntax	Expansion	Operation in C
move	`move $t, $s`	`addiu $t, $s, 0`	`t = s`
clear	`clear $t`	`addu $t, $zero, $zero`	`t = 0`
load 16-bit immediate	`li $t, C`	`addiu $t, $zero, C_lo`	`t = C`
load 32-bit immediate	`li $t, C`	`lui $t, C_hi` `ori $t, $t, C_lo`	`t = C`
load label address	`la $t, A`	`lui $t, A_hi` `ori $t, $t, A_lo`	`t = A`

- System Calls - Traps - go to kernel

System Calls

Service	Code	Arguments	Result
print integer	1	$a0=integer	Console print
print string	4	$a0=string address	Console print
read integer	5		$a0=result
read string	8	$a0=string address $a1=length limit	Console read
exit	10		end of program

-- v0 - used to determine which syscall

Hello World

```
        .text           # text segment
        .global __start
__start:                # execution starts here
        la $a0,str      # put string address into a0
        li $v0,4        #
        syscall         # print
        li v0, 10       #
        syscall         # au revoir...
        .data           # data segment
str:    .asciiz  "hello world\n"
```

-- .global__start -- global variable start
.text #text segment heading - indicates instructions
.data #data segment heading - indicates data

ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

ASCII 01

SOH

alt + 1
(Start of Header)

frequently-used (spanish language)

ñ	alt + 164
Ñ	alt + 165
@	alt + 64
¿	alt + 168
?	alt + 63
¡	alt + 173
!	alt + 33
:	alt + 58
/	alt + 47
\	alt + 92

vowels acute accent (spanish language)

á	alt + 160
é	alt + 130
í	alt + 161
ó	alt + 162
ú	alt + 163
Á	alt + 181
É	alt + 144
Í	alt + 214
Ó	alt + 224
Ú	alt + 233

vowels with diaresis

ä	alt + 132
ë	alt + 137
ï	alt + 139
ö	alt + 148
ü	alt + 129
Ä	alt + 142
Ë	alt + 211
Ï	alt + 216
Ö	alt + 153
Ü	alt + 154

mathematical symbols

½	alt + 171
¼	alt + 172
¾	alt + 243
¹	alt + 251
³	alt + 252
²	alt + 253
ƒ	alt + 159
±	alt + 241
×	alt + 158
÷	alt + 246

commercial / trade symbols

$	alt + 36
£	alt + 156
¥	alt + 190
¢	alt + 189
¤	alt + 207
®	alt + 169
©	alt + 184
ª	alt + 166
º	alt + 167
°	alt + 248

quotes and parenthesis

"	alt + 34
'	alt + 39
(	alt + 40
)	alt + 41
[	alt + 91
]	alt + 93
{	alt + 123
}	alt + 125
«	alt + 174
»	alt + 175

x86-64 Reference Sheet (GNU assembler format)

Instructions

Data movement

`movq` Src, Dest	Dest = Src
`movsbq` Src,Dest	Dest (quad) = Src (byte), sign-extend
`movzbq` Src,Dest	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove` Src, Dest	Equal / zero
`cmovne` Src, Dest	Not equal / not zero
`cmovs` Src, Dest	Negative
`cmovns` Src, Dest	Nonnegative
`cmovg` Src, Dest	Greater (signed >)
`cmovge` Src, Dest	Greater or equal (signed $\geq$)
`cmovl` Src, Dest	Less (signed <)
`cmovle` Src, Dest	Less or equal (signed $\leq$)
`cmova` Src, Dest	Above (unsigned >)
`cmovae` Src, Dest	Above or equal (unsigned $\geq$)
`cmovb` Src, Dest	Below (unsigned <)
`cmovbe` Src, Dest	Below or equal (unsigned $\leq$)

Control transfer

`cmpq` Src2, Src1	Sets CCs Src1 Src2
`testq` Src2, Src1	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed $\geq$)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed $\leq$)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq` Src	%rsp = %rsp 8, Mem[%rsp] = Src
`popq` Dest	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

Arithmetic operations

`leaq` Src, Dest	Dest = address of Src
`incq` Dest	Dest = Dest + 1
`decq` Dest	Dest = Dest − 1
`addq` Src, Dest	Dest = Dest + Src
`subq` Src, Dest	Dest = Dest − Src
`imulq` Src, Dest	Dest = Dest $*$ Src
`xorq` Src, Dest	Dest = Dest ^ Src
`orq` Src, Dest	Dest = Dest | Src
`andq` Src, Dest	Dest = Dest & Src
`negq` Dest	Dest = − Dest
`notq` Dest	Dest = $\sim$ Dest
`salq` k, Dest	Dest = Dest $\ll k$
`sarq` k, Dest	Dest = Dest $\gg k$ (arithmetic)
`shrq` k, Dest	Dest = Dest $\gg k$ (logical)

Addressing modes

- **Immediate**
  $val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved