CS 33 Final Study Guide

Eggert

Fall 2018

``````MIDTERM 1````````

INTRODUCTION

_____

Details:

- Big Endian vs Little Endian:
  - both representations have the most significant (biggest) bit at the left
    - Therefore no endian problems when looking at 1 byte
  - problem comes from the order of BYTES (not bits)
    - BIG = stores data big-end first - so the first byte (at the lowest address) is biggest
    - LITTLE = stores data little-end first - so the first byte is the smallest
  - conversions between 32 bit and 64 bit reps cause problems with big endian addressing, not little endian
  - x86-64 is LITTLE ENDIAN
- Hex
  - denoted by "0x"
  - binary representation is base 2, hex is base 16
  - each hex digit represents 4 binary digits
  - 1 to 9 = 1 - 9 and 10 to 16 = A - F

  ex. 0011 1101 = 0x3D

Functions / C Code Mentioned in Lecture:

- sizeof(Type) returns number of bytes required for the type
- typedef creates your own type

  i.e. typedef int* int_pointer;

  i.e. typedef struct student { ... };
- int / char / long all implied signed (Two's Complement)
- unsigned char / int / long are unsigned representations
- alignof(x) returns the alignment of variable x, which will depend on the type of x.

  i.e. alignof(struct s1) vs. alignof(struct s2)
- int x = (A)? B : C; is the same thing as:

```
int x;
if (A) {  x = B;  }
else {  x = C;  }
```

Linux Commands:

```
$ gcc -E foo.c > foo.i       (foo.c -> foo.i)  C to Sans Macro
$ gcc -S foo.i           (foo.i -> foo.s)  Sans Macro to Assembly
$ gcc -C foo.s           (foo.s -> foo.o)  Assembly to Obj Code
$ gcc foo.o              (foo.o -> a.out)  Obj Code -> executable
$ ./a.out                          runs executable file on Linux
```
```

$ gcc -C -O2 foo.c (foo.c -> foo.o) C to Assembly
$ objdump -d foo.o prints the assembly language of foo.c

$ gdb a.out runs executable on gdb debugger
$ (gdb) disas main disassembles the main function,
and allows you to trace through the
memory and assembly of entire program

INTEGERS

Binary Representations:

- Unsigned (no sign bit)
- Two's Complement (sign bit)
- Relationship: unsigned + | signed | = 2^w
  ex. for ints
  unsigned + | signed | = 2^32

Signed Integer Representations:

type	# bytes	# bits	# poss	range of possibilities
- char	1 byte	8 bits	2^8	[-2^7 to 2^7 - 1]
- short	2 bytes	16 bits	2^16	[-2^15 to 2^15 - 1]
- int	4 bytes	32 bits	2^32	[-2^31 to 2^31 - 1]
- long	8 bytes	64 bits	2^64	[-2^63 to 2^63 - 1]

Unsigned Integer Representations:

type	# bytes	# bits	# poss	range of poss.
- char	1 byte	8 bits	2^8	[0 to 2^8 - 1]
- short	2 bytes	16 bits	2^16	[0 to 2^16 - 1]
- int	4 bytes	32 bits	2^32	[0 to 2^32 - 1]
- long	8 bytes	64 bits	2^64	[0 to 2^64 - 1]

Integer Conversion:

- Type Casts
  - type casting from bigger types to smaller types
    truncates extra bits
    ex. int -> short, long -> int, etc.
  - from smaller types to bigger types, rest of the
    bytes are arithmetically - extended (see shifting)
    ex. int -> long, short -> int, etc.
  - bit level representation unchanged, just how we
    interpret bit rep does.
    ex. (int) type casts unsigned to signed

ex. (unsigned) type casts signed to unsigned

- when type casting size and sign

ex. unsigned x = 128282;
short u = (short) x;

compiler will change data type, then sign rep.

- Conversion formulas

T2U(x) = { x + 2^w , x < 0 | x , x >= 0 }

U2T(x) = { x , x <= Tmax | x - 2^w , x > Tmax }

Shifting:

- Left Shifts can only be logical!

i.e. machine instructions salq and shlq are the same

- Right Shifts can be logical or arithmetic

- sarq = arithmetic shrq = logical

- 2 types of shifting:

1. Logical

- zero extends

2. Arithmetic

- sign-extend if most significant bit is 1
- zero extend if most significant bit is 0

- Uses for shifting:

1. Multiplying:

x << 1 = x * 2

(x << 4) + (x << 3) = x * 24

2. Dividing:

x >> 3 = x / 8

(x>>3) + (x>>2) = (3x)/8

3. Isolating Bits:

unsigned long x;

((x << 20) >> 25) << 5

will zero out the top 20 and bottom 5 bits

Set Operations:

- integers can represent sets:

(bits 0 - 31 either included or not)

- set operations using on two integers (sets)

& and (intersection)

| or (union)

^ exclusive or (symmetric difference)
(only if they differ)

~ complement (opposite)

Overflow:

- In general, overflow occurs when the result to an arithmetic op.

"moves" in the wrong direction it's supposed to:

ex. adding two numbers = negative

ex. subtracting two numbers = number that is bigger
than operands

ex. multiplying two positive numbers = negative number,
or a number smaller than the operands

Integer Addition:

- Signed: x + y
  - Overflow occurs when two large positive numbers or two large negative numbers add together, and their resulting value can't be represented in the Two's Complement format.
  - Positive overflow
    bits keep adding up, so a "1" will replace the highest order sign bit and then make the number overflow to INT_MIN and work back up to zero
  - Negative overflow
    opposite of Positive
  - Formula:
    x + y = { x + y - 2^w | 2^(w-1) <= x + y //pos
    x + y | -2^(w-1) <= x + y <= 2^w - 1 //none
    x + y + 2^w | x + y < -2^(w-1) //neg }
- Unsigned: (unsigned) x + y
  - Overflow when the sum of two large positive numbers can't fit into the bit representation for that type
  - Overflow will reset the bits back to 0 and work back up
  - Formula
    x + y = { x + y | x + y < 2^w
    x + y - 2^w | x + y >= 2^w }

Integer Subtraction:

- big formula of integers: ~x + 1 = -x
- Signed: x - y
  - Overflows when two numbers subtract past INT_MIN or INT_MAX
- Unsigned: (unsigned) x - y
  - Overflows when y > x because it overflows past 0 and becomes a number close to INT_MAX.

Integer Multiplication: x * y

- Overflows quicker than any other arithmetic operation
- Unsigned overflow graph looks like a log graph
- Signed looks like logarithmic graph on every axis

Integer Division: x / y

- Overflow occurs nowhere, but errors do arise along each axis of the graph when division by 0 occurs

MACHINE INSTRUCTION PROGRAMMING

Machine Instruction Format:

- [instruction][suffix] [Source], [Destination]
- 4 types of suffixes
  - describe how much data you are moving

    b = byte (8 bits) chars

    w = word (16 bits) shorts

    l = longword (32 bits) ints

    q = quad (64 bits) longs
- 3 types of operands (Sources)
  1. $ immediate (value treated as a constant)
  2. Ra register (reference - the address in register)
  3. M[addr] memory reference (value in memory at addr)
- "Effective Address" formula

  Imm(ra, ri, s) = Imm + R[ra] + R[ri] * s
  - Imm can be used as an exact offset into ra

    i.e. commonly used in stack
  - ra is called the base register, and it's the reg that the effective address will dive into
  - ri is the indexing register, which can be any reg (rax, rdx, etc.) that holds an index to access values in the base register
  - s is the size of the data types in reg ra, as if you wanted to index into an array in ra, you would need to multiply the index by the size of the data type.
    - can only be 1, 2, 4, or 8
  - effective address used as an operand in machine instructions for insn.s like leaq and movq
- Destination can only be a register or a memory location, can't be a constant or an address.

Data Movement Instructions:

- mov instructions (see x86-64 Cheat Sheet for details) *******
  - mov (S), D moves memory at S to D
  - mov S, D moves address of S to D
- movsbl = sign extend the byte to a longword
- movzbl %al, %eax is a special instruction that zero extends a byte (b) to a long (l), same as zeroing out all of %eax EXCEPT for the byte in %al

  ex. %eax = 0x12345678 %edx = 0xAAAABBBB

  movb %dh, %al %eax = 0x123456BB

  movsbl %dh, %eax %eax = 0xFFFFFFBB

  movzbl %dh, %eax %eax = 0x000000BB
- memory references always give registers quadwords (8 bytes) of data, even if the operands or suffixes are less.

- Pointers in C = addresses, dereferenced pointers in C = accessing the memory at the address
  - 2 steps in machine instructions:
    1. Copying address (pointer) to a register
    2. Accessing the memory at that register

Load Effective Address Instructions:
- instructions that utilize the effective address form
- leaq S, D D = &S
  - moves the address made by the effective address (S) into reg D
  - used to make formulas (rdx holds x)
  - ex. leaq 7(%rdx , %rdx, 4), %rax = 5x + 7
  - leaq (, %rdx, 8) is faster than rdx << 3
- movq S, D D = S
  - moves the value at address S to reg D

Unary and Binary Operations:
- (see x86-64 Cheat Sheet for details) *******
- Destination serves as an operand and a destination for Binary operations
  - ex. add S, D places S+D into reg D

Special Arithmetic Operations:
- (see x86-64 Cheat Sheet for details) *******
- imulq, mulq, idivq, divq
- when the cheat sheet uses the notation:
  - result stored in %rdx:%rax

Multiplication:
- the compiler uses the two registers to hold up to a 128 bit result, with rax as the lower order bits and rdx as the higher order bits.

Division:
- rax used to store the quotient and rdx used to store the remainder.

Stack Operations:
- stack grows downward, moving from high addrs to low addrs
- %rsp points to the top element (lowest addr) on the stack, so pushq inherently performs the two machine instructions:
  - subq $8, %rsp //moves to next vacant stack element
  - movq $0x3030, (%rsp) //stores address in stack
- Pushq S
  - pushes the address in reg S onto the stack
  - always followed by a subq $0x18, %rsp to allocate more stack space (see Procedures)
  - essentially:

R[%rsp] = R[%rsp] - 8
M[R[%rsp]] = S
- used to push callee-save registers onto the stack frame before calling a function

- Popq D
  - pops the value from the top of the stack into D
  - always preceeded by an addq $0x18, %rsp to free the stack space (see Procedures)
  - essentially:
    R[D] = M[R[%rsp]]
    R[%rsp] = R[%rsp] + 8
  - used to pop callee-save registers off the stack frame at the return of a called function

CONTROL

Comparisons and Test Instructions:

- testq and cmpq
- won't modify any registers, only set the condition codes
- Condition Codes
  - organized by bit # in %CFLAGS register
  0. Carry Flag (set at unsigned overflow)
  6. Zero Flag (set if last operation is 0)
  7. Sign Flag (set if last operation is negative)
  11. Overflow Flag (set at signed overflow)
  - condition codes are set at EVERY arithmetic operation (Binary, Unary, Logical, shifting) and also the comparison instructions testq and cmpq.
- testq S, D
  - sets condition codes according to S&D
  - can set: zero flag (ZF) when S = D = 0
    sign flag (SF) when S&D < 0
  - typically operand is repeated or one operand is a mask indicating which bits to be tested
- cmpq S, D
  - sets condition codes according to D - S
  - can set: zero flag (ZF) when equal
    carry flag (CF)
    sign flag (SF) when (D-S) < 0
    overflow flag (OF)

3 Ways to Use Condition Codes:

1. Using Conditional Set Operations
   - sets lowest byte to 0 or 1 depending on codes
   ex. setne %rax sets %al if ZF = 0 (!ZF)
   - (see x86-64 Cheat Sheet for details) *******

2. Conditional Jumps
    - %rip jumps to another part of assembly
      depending on the condition codes
    ex. je L_B03 jumps to L_B03 if (!ZF)
                  otherwise continues on.
    - (see x86-64 Cheat Sheet for details) *******
    - how if, while, and for loops are implemented
      in assembly code
3. Conditional Moves
    - movs data in registers depending on codes
    ex. cmovns %rdx, %rax moves %rdx to
                          %rax if !SF (pos)
    - (see x86-64 Cheat Sheet for details) *******
    - No suffixes, but S and D must have the
      same sizes
    - cmovc %edx, %eax (midterm 2)
      will move M[%eax] to %edx if CF or OF = 1,
      same as if %edx overflowed (re-initializes
      %edx if previously corrupted)

Jumps:
- (see x86-64 Cheat Sheet for details) *******
- Conditional Jumps (above)
    - used for small local jumps (close jumps
      to implement if, while, for, switch loops)
- Absolute Jumps:
    - used for big jumps (across large portions of
      the overall program)
    jmp .L38 absolute jumps to .L38
    jmp *%rax indexed jump (rip = %rax)
        - hackers commonly attack indexed jumps
          in order to run their malware or crash
          your program (see Dynamic Linking)
    jmp *(%rax) indirect jump (rip = M[%rax])
- Switch statements with jumps:
    - uses an array of jump sections to decide which
      case the absolute jump will take %rip to.
        - called a JUMP TABLE
- Jump Tables
    - way for the compiler to look up the target branch
      in switch statements
    - implemented by an array of assembly labels
      (like .L2, .L8, etc.) called jtab[]
    - jtab[0] located at .L57
    - essentially:
        target = jtab[op];
        goto *target;

- in assembly:
  jmp *.L57(, %eax, 4)
  where %eax holds "op" the index of jtab
- jump tables don't appear in disassembly, but can be inspected using GDB.
- if there's a huge number of switch cases (100) the compiler will make a Binary Search Tree of the jump table - which is what makes switch statements so fast

Loop formats:

- written using "GoTo" shorthand
  - halfway between C and assembly, makes it easier to translate between them
- If() / Else() conditionals
  - conditional move
  Test / Cmpq
  Conditional Move
  Body

```
- conditional jump
if ( Test )
    goto else:
Body
return
else:
Body
return
```

- Do-While() loops

```
loop:
    Body
    if ( Test )
        goto loop:
    return
```

- While() loops and For(, ,) loops

```
    if ( !Test )
        goto done:
loop:
    Body
    Update
    if ( Test )
        goto loop:
done:
    return
```

PROCEDURES

Instruction Pointer:

- register %rip holds the address of the next machine insn to be executed.
  - in this way, we can think of the %rip register as a data structure that holds the next instruction

Function Calls: (step by step)

1. callq 4e<fooabs + 0xe> inherently does the following:
    1. pushes the return address onto stack frame
        pushq %rip (which inherently performs 2 other machine instructions)
    2. Jumps to label 4e<fooabs + 0xe>
        - compiler doesn't actually know where this is
        - 4e<fooabs + 0xe> is a placeholder (or a "hole" that the linker will fill) (see Linking)
    2. Creates a new stack frame for the function on top of the old stack frame (since stack grows (up/down?))

(steps 2 and 3 called function prologue)

2. push Callee - Save registers onto stack frame that will be used in the function

```
pushq %rbx
pushq %rbp      //stack base pointer
...             //keeps track of stack frames
```

- Callee - Save registers
    - registers that are saved across function calls to be restored before ret (FILO!!)
    - %rbx, %rbp, %r12 - %r15
- Caller - Save registers
    - registers that are not saved across function calls, and may be trashed within the function
    - %rax, %rcx, %rdx, %rdi, %rsi, %rsp, %r8 - %r11

3. allocate stack space
    - after pushing all the callee-save registers onto the new stack frame, we must allocate the amount of space we pushed:

```
subq $0x18, %rsp        24 bytes of space = 3 regs
```

    - stack allocation = alloca(...) and heap allocation = malloc(...) (see malloc vs. alloca on midterm 2)
4. machine code of function
    - if the function has a return value, it will be stored in %rax
    - if theres another function call within the function, the process starts over!
        - leaf function = function that doesn't call another function
    - when the nested function call returns, %rip picks up at step 5 (below) where it left off
5. free stack space
    - after the assembly of the main function, we must free the space to restore the registers we allocated when pushing them on the stack

```
addq $0x18, %rsp        24 bytes ( same amount )
```

6. pop Callee - Save registers

- remember stack rules, First In Last Out (FILO)
    ... pop %rbp
        pop %rbx

7. return pops %rip and the program continues where it left off

Calling Conventions for x86-64:

- differ machine to machine
- detailed descriptions of how arguments and function calls are handled internally
- return values are stored in %rax
- first 6 integer arguments are passed into:
    (1) %rdi (2) %rsi (3) %rdx (4) %rcx (5) %r8 (6) %r9
- first 8 floating point arguments:
    (1) %xmm0 (2) %xmm1 (3) %xmm2 ... (8) %xmm7
- arguments that don't fit into registers or integer arguments 7+ are pushed onto the stack where the last parameter in the func is furthest from %rsp.
    - to access, we have to use movq -C(%rsp), %r12 where C is equal to the number of bytes into the stack the parameter lies

Stack Frames in Procedures:

- Stack basics in x86-64:
    - stack grows downward, moving from high to low addrs
        - therefore to grow the stack, subq $0x18, %rsp
        - and to shrink the stack, addq $0x18, %rsp
    - at the beginning of the function, %rsp = %rbp (base ptr) which points to the base of the current stack frame.
        - mem[%rbp] = %rbp of the next older stack
        - %rbp is an optional convention used internally to keep track of the stack frames
- %rsp points to the top element on the stack (lowest addr)
- A typical stack frame follows the following order: (inverted)

```
[ "red zone" above stack (128 bytes) ]
[ local variables from function ]   <---      %rsp
[ Callee - Save regs ]
[ old %rbp ]                        <---   %rbp
[ old %rip "return address" ]
[ variable 7 ]
[ variable 8 ]
[ ... ]                                    func1( ) frame
-------------------------------------------------
[ local variables from function ]  foo( ) frame
[ ... ]
```

- consider the machine code:

```
<main>:
        lea    0x4(%esp),%ecx     loads the address of the object
                                  saved 4 bytes into deep in stack
        and    $0xfffffff0,%esp   makes sure stack is aligned to 16
                                  doesn't mess with any data bc by
                                  &ing, the stack can only grow
        pushl  -0x4(%ecx)                 works back and pushes old return
                                  address onto stack so main can
                                  return normally
        push   %ebp               pushes old rbp
        mov    %esp,%ebp                  sets new rbp = rsp
        push   %ecx               saves caller-save temp
                                  variable in rcx
        sub    $0x4c,%esp                 allocates total space on stack
                                  that will be used ( pushes and
                                  using the stack in the func )
    ------------------------------------------------------------  (end of function prologue)
        movl   $0xc,-0x8(%ebp)    moves 12 into memory 8 bytes
                                  below the base pointer ( temp
                                  variable used in the function )
        movl   $0x14,-0xc(%ebp)           moves 20 into memory 12 bytes
                                  below the base pointer ( another
                                  temp var)
        mov    -0x8(%ebp),%eax    moves 12 into %eax
        add    $0x7b,%eax                 123 + 12 = 135 into %eax
        mov    %eax,-0xc(%ebp)    moves 135 into 2nd temp var,
                                  overwriting 14.
        mov    $0x0,%eax                  sets rax = 0
    ------------------------------------------------------------  (start of function epilogue)
        add    $0x10,%esp                 frees the memory used in frame
        pop    %ecx               pops caller-saved temp var
        pop    %ebp               pops the old base pointer
        lea    -0x4(%ecx),%esp    loads current rbp into rsp
```

- this machine code was from the function:

```
int main() {
   int a, b;
   a = 12;
   b = 20;
   b = a + 123;
   return 0;
}
```

- consider the machine code:
    <fooabs>:
        44: test %rdi, %rdi
        47: js 0x60
        49: callq 4e<fooabs + 0xe6ff>
    *** 4e: add $0x8, %rsp
        52: add $0x2, %rax
        56: ret
    - the machine instruction located at offset 4e is an insn meant to align the instructions within the stack to dec multiples of 16.
        - insn.s will run faster if aligned in memory (see alignment below)

DATA STRUCTURES

Arrays:

- index into registers by adding scaled indexes to address in register than points to the first element
    ex. lea 0x4(%rdi),%rcx
  loads the address of the object 4 bytes into rdi (rdi is a pointer to an array or just an array);
    ex. long array ln = 0x8(%rdi) = ln[1]
        int array i = 0x4(%rdi) = i[1]
        char array c = 0x1(%rdi) = c[1]

Data Structures will run faster in C if their data is Aligned!

Struct Alignment:

- The alignment of a data type is the "offset" that the member variable should be a multiple of within a struct
    - char 1 byte
    - short 2 bytes
    - int 4 bytes
    - long 8 bytes
  therefore, within a struct, each character can be placed at any offset since all bytes are multiples of one, but integers must begin at any offset multiple of 4.
    - the first byte in memory is at offset 0
- The Struct itself also must align in memory, therefore we define the alignment of the struct = alignment of the largest data mem
    - the overall size of the struct must be a multiple of its alignment, therefore we can "pad" junk bytes at the end of the struct to meet this size requirement
- Consider the following structure:
    struct s1{ char c; int x; short s; int j; };

char c will be at offset 0, but in order to align int x, we must
fill the structure with 3 bytes of junk in between c and x

[c]	[junk]	[x]	[s]	[junk]	[j]	total
0	1-3	4-7	8-9	10-11	12-15	16

- rearranging this order changes how the memory will be aligned, and the overall size of the structure in memory:

```
struct s1{ int x; int j; short s; char c; };
```

translates to

[x]	[j]	[s]	[c]	[junk]	total
0-3	4-7	8-9	10	11-12	12

Unions

- unions are a special data structure that can only represent one of its data members at a time, and it is up to the user to keep track of what it represents.
- consider:

```
Union U{ long l1; char c; int i; };
```

in memory:

[[[c]	i	]	l1	]	total
0 1	3		7 8		

- the size of the union is the size of its largest data member, and all of the data members begin their representation at byte 0. If the user calls the character, the union will only return the 1st byte, but if the user calls the integer, the union will return the first 4.
  - same bit representation, that's why the user must know what data type the union was previously used.
- the alignment and size of a union acts the same as alignof(...) and sizeof(...) the unions biggest data member

Structures with Arrays:

- when structures are defined with varying size arrays, the comp. doesn't know how much space to allocate for the array.
- consider:

```
struct s1{ char c;  short s;  int j[ ]; };
```

in memory:

[c]	[junk]	[s]	[j]
0	1	2-3	4 ---> ?

- the array could go on forever (called a flexible array member)
- therefore WE CAN ONLY USE THIS struct when we allocate enough space to fill the whole union and space for our array:

```
struct s1 *p = malloc( sizeof(Struct s1) + 37 );
```

where 37 represents the extra bytes for the array.

Instruction Alignment:

- NOP maching instruction
  - "no operation" maching instruction

- does nothing, just increments %rip by x bytes of memory determined by the suffix. recall:
  b = 1B, w = 2B, l = 4B, q = 8B
  - b can pad anywhere from 1-15 bits,
    w can pad anywhere from 16 - 255, etc.
- used for alignment purposes
  (increases performance)

- We want to align code to 16 in x86-64 because that is the cache line size of the machine

POINTERS

Pointer Caution:

- pointers are POWERFUL but DANGEROUS
- if you misuse pointers, you can modify parts of your program unwillingly or trash valuable data.

Pointer Type Casting:

- numeric casts:
  1) attempts (often modified) to preserve the value
  2) requires overhead instructions by CPU
  ex. int x = 50; long g = (long) x;
- Pointer casts:
  1) preserves the bit level representation
     - pointing to an object of different size may lead to an access overflow, wrong result
  2) generates NO machine code / instructions
  ex. int x = 5050; long *p = (long*) &x;

Invalid Pointers:

- null pointers (points to null)
- dangling pointers (delete the object but not the pointer to it)
- uninitialized pointers (int*p;)
- subscript error (try to access array[i] where i is out of bounds)
- incorrect casts ((char *) 1036 - characters can only be ASCII values)

Runtime type checking:

- called Interpreters
- within the 8 bits inherently in each pointer, the last 4 bits are always 0 for alignment purposes (aligned to 16).
- We use these last 4 bits to hold the pointer type so the comp. doesn't have to inspect memory (slow):
  long get_type (void *p) { return (long) p & 15; }

Void* Pointer:

- when you use void *, you tell the compiler to suspend all runtime type checking (more danger, more freedom).

Aliasing:

- given T *p and U *c, aliasing can occur when:
  1. T = U (same types)
  2. T = char or unsigned char, U = int or unsigned int
  - characters act as integers based on their ascii value, so pointers can point to the same location in memory even if an int*

``````MIDTERM 2````````

INTRODUCTION

_____

Linux Commands:

$ gcc -O2	compiled with optimization level 2
$ gcc -O2 - fwrapv	wraparound for signed overflow
$ gcc -O2 - fsanitize = undefined	program crashes at signed overflow
$ gcc -O2 -ffast -math	assumes all math is simple, allows for opt.
$ size a.out	returns size of program a.out
$ text size	size of read-only instructions/constants
$ data size	size of read-write initialized variables
$ bss size	size of read-write variables uninitialized
$ gcc -pg ... foo.c	generates hotspot profile
$ ./a.out	runs the program and creates a txt file (gmon.txt)
$ gcc hello.c -o hello -lpthread	compiling multithreaded prog. in Linux using pthread directives

Post Midterm 1

_____

malloc vs. alloca:

- memory allocated using malloc() will allocate space on the heap
- it will remain there until it is given back using free()
- malloc used for big memory allocation

- memory allocated using alloca() will allocate space on the current function's stack frame.
- alloca's memory will be returned when the current function returns
- alloca limited to small memory allocation
``````

Q: can a pointer ever point into another thread's stack?

- called Cross-thread Stack access
- even if carefully coordinated and without race conditions, cross thread stack access can make your program run unexpected behavior.
- example:

(preparation)

```
int *p;                 //global pointer
CreateThread( ..., thread #1, ... );
CreateThread( ..., thread #2, ... );
```

(Thread #1)

```
int q[1024];        //allocated on thread 1's stack
p = q;              //set global pointer equal to
q[0] = 1;                   address on thread 1's stack
```

(Thread #2)

```
*p = 2;             //Thread 1's stack accessed
                    //sets q[1024] = 2
```

- ties into keyword "volatile" for global static variables
- see "Threads" below

Type Based Optimization:

```
C:                              MC:
int f ( int *p , int n ){       movl    (%rdi), %ebx
    int j = *p + 1;             addq    $1, %ebx
    g( *p, n );                         addq    %ebx, %esi
    return *p + 2;              movl    %esi, %rdi
}                               call    g
                                leal    1(%rbx), %rax
                                ret
```

- notice optimization:
  1. rdi holds &p, (rdi) holds *p
  2. compiler avoids computing *p twice by storing *p in ebx
- notice problems:
  3. machine code doesn't garuntee a correct result because *p could be changed in the call to g.

slightly different function:

```
C:
int f ( int *p, double *d ) {
        int j = *p + 1;
        *d = j;
        return *p + 2;
}
```

- notice no more problems, and optimization
  1. by making the second type a pointer to a different type, the compiler can optimize machine code since it knows

aliasing is impossible. (knows this through runtime type checking)
- also adding keyword "restrict" before variable name
will tell the compiler that there will be no aliasing.
2. if the second argument was int *d, then the second line
of source code could be problematic if an alias for *p.
- to handle this, the compiler would have to access
the memory of p and d to ensure it's doing it right
(very slow - compiling stepi by stepi)

Breakpoints:

Used by GDB to debug and start/stop programs

Software Breakpoints

(gdb) b foo

- changes the first two bytes of a call to "foo" in the program machine code to a random number, so when the compiler tries to call 'random number', the program crashes and stops at that line.

Hardware Breakpoint Registers

%DR0 - %DR3	watch registers (only 4!)
%DR6	status register
%DR7	control register

(gdb) watch px

- puts the address &px into one of DR0 - DR3 and watches for the memory at that address to change.
- after 5th watch variable, compiler will stepi by stepi your code.

FLOATING POINT

- x86-64 uses IEEE-754 format for floating point
- Floating point problems differ from integer arithmetic problems:
1. overflow still exists, less common due to range floats
- what is the range of floats ?
2. limited precision (floats will round)
- how much and when ?
ex. repeating decimals (like 1/3)
0.010101.... (base 2) will continue forever so it must round to an inexact value (like calculators)
3. division by 0
4. truncation of bits no longer a problem since floating point can represent decimals too
5. underflow (opposite of overflow) is when exponent becomes so small that the float will round to 0
6. not associative, order of operations will change the output
i.e. a + (b + c) does not equal (a + b) + c

- Floats:
    - bias = 127
    - bits represents total 32 bits
    1 (ms) sign
    8 exponent [-126, 127]
    23 (ls) fraction
- Doubles:
    - bias = 1023
    - bits represents total 64 bits
    1 (ms) sign
    11 exponent [-1022, 1023]
    52 (ls) fraction
- Deciphering bit representation of floats
    Categories of floats:
    1. 0 < exp < 255 normalized
        +/- 2^(exp - 127) * 1.f (base 2)
    2. exp = 0 tiny numbers (denormalized)
        +/- 2^(-126) * 0.f (base 2)
    3. exp = 255 (all 1's), frac = 0 infinities
    4. exp = 255 (all 1's), frac = nonzero NaN
        - NaN = not a number
        - remember, NaN's are infectious with all operations except ^ (exclusive or)
        - the sign bit of the NaN can help us figure out whether it overflowed or underflowed.

- Denormalized
    - normalized = leading bit 1.f
      denormalized = leading bit 0.f
    - we need denormalized numbers because there's cases in normalized representation that will return true for the following expression:
        (a - b = 0) && (a != b)
    - basically, theres a case where a - b = 0 but a and b are not equal.
        - happens in underflow (gets too close to 0 so it rounds to 0)
    - so we use the denormalized form for cases where underflow occurs:
    - form:
        +/- 2^(-126) * 0.f (base 2) 'tiny numbers'
- Rounding
    1. cieling operation (towards positive infinity)
    2. floor operation (towards negative infinity)
    3. round to even (round to nearest even number)
    4. trunc operation (round towards 0)

- Constructing a bias:
    2 ^ (#bits - 1) – 1

- Using a bias:
    the base 2 scientific notation that floating point is
    based off follows the following format:
        V = 2^E * frac where E = exp - bias
- converting between different floating point exponents:
    1. express the two formats in scientific notation
    2. E = exp(A) - bias(A) = exp(B) - bias(B)
    therefore exp(B) = E + bias(B)
    3. frac(A) = frac(B)
    if the #bits shrink, then round up
    if the #bits grow, then extend with 0's

Comparing Two Floats:
- each float falls into 1 of 8 categories of type and sign
    Types: normalized, tiny, infinity, NaN
    Sign: positive, negative
- most float comparing can be done with this information
    ex. tiny < normalized
        infinity > tiny
        NaN ? normalized
        - normalized < + normalized
- if both are of same sign and type, compare the exponent bits and
fractional bits together - it will work the same as comparing both individually.
- 4 possible results of float comparison (can be found in
condition flags after a comparison)
    < (less than)
    > (greater than)
    = (equal)
    ? (unsure) used for NaN's

Addition of Floats:
- check if the exponents are the same, and then add.
- if they aren't equivalent, we must shift the bits of the smaller number's
frac to the right by |exp1 - exp2|
- if there are "lost bits" after shifting to the right and adding, use them
to decide how to round:
    1. if < half of the next decimal value, floor operation
    2. if > half, ceiling operation
    3. if = half, round to even

Casting Floats:
- float -> double no trouble
- double -> float
    - there will only be trouble if the double value is within float range
    - if it overflows, the float will be +00
- float -> int
    - there will only be trouble if the double value is within float range
    - if overflows, int set to INT_MAX or INT_MIN
    - truncates when decimals

Floating Point Registers in x86-64:

- SSE2 128-bit registers %xmm0 - %xmm15
- one 128-bit register = 2*64bit registers together accessed as arrays of:
  - bytes 16 elements
  - ints 4 elements
  - longs 2 elements
  - floats 4 elements
  - double 2 elements

or simply: 128 bit-value1 element

PROGRAM OPTIMIZATION

Big Picture:

We optimize to save:

time, memory, power (energy) <-- competing goals

At a cost of complexity.

Measuring Performance

1. measuring size:
   - Statically - how much data is used by in program

   $ size a.out //returns size of program a.out

   $ text size //size of read-only instructions/constants

   $ data size //size of read-write initialized variables

   $ bss size //size of read-write variables uninitialized

   - Dynamically - how much data is used while running the program
2. measuring time:
   - clock time:

     "real time", or input/output total time with user
   - CPU time:

     time the CPU spends doing tasks in program

     $ time ./a.out //your program's CPU
3. I/O count: number of inputs and outputs used
4. Network Latency
   - i.e. DNS Latency

     a. the bottleneck that maps a "google.com" search to an IP address
   - more important in today's world as we try to improve machine learning and A.I.
5. measuring power/energy

2 Kinds of CPU Performance:

1. throughput (operations / second)
   - we want HIGH throughput
2. latency (seconds)
   - amount of time in between request and response (servers)
   - we want LOW latency

- high throughput and low latency are conflicting goals
    - we want to perform the most operations a second, but the more operations we send a computer/server the longer it will take to process each one.

GCC optimizations:
- gcc -O2 is gcc compiler with optimization level 2
    - compile time longer since it optimizes, but clock time shorter
- gcc is only allowed to make safe optimizations
    - safe optimizations are optimizations that still ensure the original meaning of the source code
        ex. no optimization on varibales allowed on functions where there could be aliasing (pointers)
- as humans, we can make optimizations even faster with unsafe optimizations that we know work
- gcc -O2 -ffast -math is optimization where the compiler takes risks and assumes all the math computation in the program will be simple arithmetic
- Why safe optimization is hard:
    1. debugging
    2. aliasing
    3. side effects in functions

Aliasing:
- as mentioned before, for the function foo, the compiler can't optimize the code since int *p and int *q could point to the same memory address
- if we add keyword "restrict" to the arguments in the function foo we tell the compiler that within the scope of our program, the two will never be aliases and will allow the compiler to optimize the code.

```
int foo ( int *p, int *q ){          int foo ( int restrict *p, int restrict *q ) {
    *p = 3;                              *p = 3;
    *q = 3;                              *q = 3;
}                                    }
*no optimization*                        *optimization*
```

Amdahl's Law:
- overall speedup = 1 / [(1-a) + (a/k)]
- where a = fraction of time in an optimization blocker (bottleneck) and k = ratio of how many times faster program runs

Statistical Profiling:
- helps to find the "hotspots" in your program
- randomly interrupts the program and creates a histogram of the %rip
    - gives us a "profile"

```
gcc -pg ... foo.c  //"generate profile"
./a.out  //runs the program and creates a txt file
    // gmon.txt
```

Coverage:

- gives you exact hotspots, unlike statistical profiling which is a guess
- more machine code, allocation, etc.
  "expensive but accurate" instead of "cheap but simple"

Common Bottlenecks:

- double for loops are almost always the most common bottleneck
  - runs the same code over and over again, making it one of the biggest hotspots in your code.

ex.

```
int i = 8;
for (i = 0; i < n; i++) {
        for ( j = i; j < n; j++){}
}
```

- Kernel == biggest/most key bottleneck in your program

Kernel Performance Improvement: ******

- all of the below will inherently be done by most compilers when they are safe to do so. If unsafe, us humans can help the compiler out

1. "hoist" code out of the for loop

ex.

turn `for ( i = 0; i < slen( j ); i++) { ... }`

into:

```
int k = slen( j );
for ( i = 0; i < k; i++ ) { ... }
```

- if the function slen() has 'n' instructions and the for loop runs k times, you will decrease the overall number of machine instructions from m*k to just m+k.
-also works for subscript calculations within the for loop

ex. turn

```
for (i = 0; i < k; i++) {
        if ( a[i] < a[i + 1] ) { return 1; }
        else if ( a[i] > a[i + 1] ) { return -1; }
        return 0;
}
```

into :

```
for (i = 0; i < k; i++) {
        int x = a[i];
        int y = a[i + 1];
        if ( x < y ) { return 1; }
        else if ( x > y ) { return -1; }
        return 0;
}
```

- this won't improve the kernel as much as the above optimization because if accessing an element in the array a takes 'm' instructions and the for loop will take 'k' instructions, then the above changes will change the overall number of instructions from 4*m*k to 2*m*k, which is still "O(n^2)" (not actually, but of the same order)

2. inline functions (avoid function calls)

- instead of calling a function, write the function in the for loop. when you call a function, the compiler must push %rip onto the stack, save all callee-save registers and create a new stack frame which trashes the current cache's!

3. cache accumulating

- machine code optimization where the variable that is constantly updated in the for loop should be stored in a register, and then placed back into memory at the end instead of constantly accessing RAM.

4. loop unrolling

change:
```
for (i = 0; i<n; i++) {
    v += a[i];
}
```
to:
```
for (i=0; i<n; i++) {
    v += a[i];
    v += a[i + 1];
}
```

- in the first for loop, there are 4 machine instructions for 1 'i'.

```
cmpq    i < n
addq    i++
addq    v += a[i]
movq    v += a[i]
```

but in the second for loop, there are 6 machine instructions for 2 'i'

```
cmpq            i < n
addq            i++
2 x addq        v += a[i], a[i+1]
2x movq         v += a[i], a[i+1]
```

- optimization is minute but after 1000 iterations, will make a difference

5. code selection

- shift << 4 instead of imulq *16
- imulq is much slower than sign extending 4 bits of a register

6. code ordering

- machine code optimization

ex.

change:
```
movq
addq
movq
addq
```
to:
```
movq
movq
addq
addq
```

7. identifying common expressions

ex. `for ( i = 0; i < n; i++ ) {  x *= a[i] ^ a[i+1];  }`

- in this for loop, i+1 is computed twice, once in a[i+1] and again in the next for (i = 0; i < n; i++) when it computes i++.
- to optimize, only do this computation once and store the result in a register to be used for both expressions.

8. dead code eliminating (meaningless code)

THE MEMORY HEIRARCHY AND CACHES

The Memory Heirarchy:

- pyramid format
  - at the top: fastest access, smallest amount of memory
    - Registers (closest to CPU)
    - L1 Cache
      - Instruction and Data
    - L2 Cache
    - L3 Cache
    - DRAM (Dynamic RAM - memory with capacitors)
    - Disk Drive
    - Github (furthest from CPU)
  - at the bottom: slowest access, largest amount of memory

Represents the BIG DELIMMA of Computer Organization:

- We can either have fast access, or large amount of space
  - big and slow (RAM) or small and fast (Registers)
- But we want both!
  - Answer? Caches.

Caches

Overview:

- Every memory access goes through the cache
- basically, caches will hold the most frequently used addresses and their memory, in order to improve program performance
- accessing memory in the cache ~ 1 cycle
  accessing memory from memory ~ 60 cycles

Locality:

- the concept that memory accessed by a program will access data local to the access in the future.
- Temporal Locality (time)
  - values accessed in memory will be accessed again in the future
    - temporal locality is why we have caches
- Spatial Locality (space)
  - values to the left and right of the last accessed in memory will also be accessed in the future
    - think indexing through an array
    - spatial locality is why we cache in blocks and not single entries

Cache Organization:

- Cache Blocks
  - Memory is stored in Caches in Blocks, not single entries
  - basic unit for cache storage
    - Block size varies
  - contains multiple bytes / words / addresses of data
    - this is to promote the concept of locality
  - called Cache Lines when more than one block can reside in the Cache set
- Cache Set
  - a "row" in the cache
  - a block's row "index" determined by the addresses in block
    - use the address in a hashing function
    - address breakdown:
      - high order 48 bits ID "Tag"
      - middle 10 bits Hashing value (index)
      - low order 6 bits offset value (in block)
    - we use the middle bits because we want to maximize spatial locality
    - if each block has B bytes, then log2(B) bits are used for the offset.
      - ex. each Block = 32 bytes (4x 64bit addr)
        - 32 = 2^5 = first 5 bits
    - if the cache has S sets, then log2(S) bits are used to hash the index
      - ex. 8 sets = 3 bits after offset
        - 4 sets = 2 bits after offset
    - tag bits are whatever is left after the offset/index
  - number of blocks per row depends on the layout
    - this also dictates how each handles collisions

    1) direct-mapped - multiple sets, one entry per set
       - no collisions allowed - must evict
       - least searching through cache
    2) set associative - "n" sets, a few entries per set
       - "n" collisions allowed
       - more searching than direct, less than fully associative
       - most commonly used in real world
    3) fully associative - only 1 set, multiple entries
       - no collisions allowed - must evict
       - more searching through cache
- Tag
  - bits taken from each address in a Cache block to differentiate between addresses within the cache block
  - used to search for specific addresses and entries

- Valid Bit
    - single bit used to determine whether the entry is valid (1) or not (0)
    - In multi processor systems, since there is more than one cache and each cache can contain copy of same data, if it is updated by one processor in one cache then all the others need to be updated
        - valid bit will be set to 0 in all other caches until the block is written to and reset
- Dirty Bit
    - single bit used to determine whether the cache block matches RAM (1) or doesn't (0)
    - used in Write-Back policy (see Hits and Misses below)

Midterm #2 Question 1a
- we want to have 8 collisions since using an 8 way associative cache
- all of the data must be accessed repeatedly so it causes most evictions in the 4-way associative cache:
    a[x] , a[x + a1], a[x + a1 + a2], a[x + a3], ...etc.

Accessing Data in Cache:
1) hash the requested address to get the set index
2) search through the cache set for the matching tag
    - if there's no match, read miss
3) look at valid bit to check if data is valid
    - if not valid (0) , read miss
4) use block offset to index into cache block for the data
5) copy to register / modify, etc.

Levels of Caches:
- L1 Caches
    - Smallest amount of memory, fastest access
    - 2 seperate Caches for Instructions and Data so the CPU can fetch the instruction and data in parallel while running
    1) Instruction Cache
        - cache that holds the meaning of machine instructions so the CPU knows what to do when the assembly code says "movl"
    2) Data Cache
        - cache that holds the data
- L2 and L3 Cache
    - as levels increase, increased amount of memory, and decreased speed of access.
    - Unified Caches (instructions and data both)
- Optional Graphics Caches
    - cache common graphics used in images / video games

Hits and Misses:

- CPU issues the address to the Cache. If the cache holds the memory address, it's called a HIT.
    - Hits are fast and return the memory fast.
- If the cache doesn't hold the memory address, it's called a MISS.
    - Misses are slow, since the cache then must fetch the block from memory and send it back to the CPU.
    - If there's a miss, the cache will then store the block in the cache for future use.
    - Cache must decide another entry to "evict" from the cache if the cache is full
- Read Hits
    - copy the value to local register
- Read Misses
    - check the next level caches down until main memory
    - copies the address and value into original cache then reruns the faulting operation (see Exception handling)
- Write Hits CPU wants to write to the memory
    - Write Hit Policies:
    1. Write Through
        - every time the Cache is changed, RAM and lower caches updated
        - pros: data synchronization between caches & RAM
            : if cache fails, all changes saved
        - cons: MUCH slower than Write Back, lose efficiency
    2. Write Back (lazy)
        - update cache and attach a "dirty bit" to the cache block to indicate if differs from RAM, wait as long as possible (until evicted) then store the changed memory in RAM.
        - Pro: improves performance
        - Con: "dirty bit" needed, and if cache fails then changes are not saved
- Write Miss Policies
    1. no write allocate policy
        - block is written to in main memory, and never fetched back into cache.
        - Write Through and No-Write Allocate usu. paired

        ex. initializing huuge arrays (you won't need the entire array in the near future)
    2. write allocate policy
        - block is fetched from main memory and placed in the cache, then written to.
        - Write Back and Write Allocate usually paired

        ex. initializing few local variables (you will be using these frequently and in the future)

INSTRUCTION LEVEL PARALLELISM

Goal: reduced time, increased efficiency

General solutions:

1. buy lots of computers, network them
2. multiple processors (Central Processing Unit)
3. multiple cores in 1 chip (quad processor = 4 cores in 1 chip)
4. combination of 1-3
   - MIMD (using hardware for parallelism)

****** 5. multiple operations active at one time in one core
   - SIMD, ILP (using coding for parallelism) (below)

Coding Parallelism:

- multiple operations active at one time in one core
- 2 methods
  1. SIMD - single instruction multiple data
     - uses the 128-bit floating point registers %xmm0 - %xmm15
     - 128-bit registers serve as an array of 4 * 32-bit integers
     - not super useful, also a lot of work
       a. this is why SIMD still hasn't 'taken off' today

     example of a SIMD instruction:
     dest[31:0] <-- (src[127:0] >> (order[1:0]*32))[31:0]
  2. ILP - instruction level parallelism
     - instructions are *mostly* unchanged, unline SIMD
     - to do ILP, we need to know:
       1. how to tune
       2. parts of microarchitecture and CPU
       3. understand micro-operations
          - how instructions are implemented in chips
          - micro-ops are secret (for competitive reasons)
     - best ILP technique:
       Pipelining micro-operations

Pipelining:

- staggers the execution of instructions to increase throughput of instructions
- Instruction execution stages:
  - IF instruction fetch
  - ID instruction decode
  - EX instruction execution
  - MEM memory access
  - WB write back to register
  - each of these stages to executing instructions is done by a different EU (execution unit) hardware in the chip, therefore we can start the second instruction without finishing it totally.

- Diagram:

time ---->

[IF] [ID] [EX] [MEM] [WB] (instruction 1)

[IF] [ID] [EX] [MEM] [WB] (instruction 2)

[IF] [ID] [EX] [MEM] [WB] (instruction 3)

- in this diagram, the scaling will only fetch, decode, execute, etc. 1 instruction per cycle.
- downfalls of pipelining:
    1. If the instruction being executed is a jump instruction, the next 4 instructions that have already started executing are now invalid
    2. stages of instruction execution are not equal in time, therefore the hardware doing the fast stages will be doing nothing for periods of time. (throughput bound)
- some solutions:
    1. unconditional jumps quickly decode and begin unpackaging the next instruction at the new address line because they are unconditional
    2. for conditional jumps, the first pass will be slow and instructions will be invalidated. BUT, the cache will remember what happened the last time the unconditional move happened, so it will use this to guess whether to start decoding instructions at the next address or the new address.
        - see "Branch Prediction" below
        - a point professor brought up in class: if for some reason you write code that will alternate whether the conditional move is taken or not, that will make your ILP always slow.
- Super Scalar Pipelining: (modern processor)

[ ] [ ] [ ] [ ] [ ] (instruction 1)

[ ] [ ] [ ] [ ] [ ] (instruction 2)

[ ] [ ] [ ] [ ] [ ] (instruction 3)

[ ] [ ] [ ] [ ] [ ] (instruction 4)

- super scalar processors will fetch, decode, execute, etc. more than one instruction per cycle.
- downfalls of super scaling:
    1. here when the second instruction is dependent on the conclusion of the first instruction, then instructions will begin to stack up regardless off the pipelining. (latency bound)
    2. these are called dependencies

Optimization through finding dependencies:

- plot a dependency diagram to find the critical path (takes most time/memory)
- optimize this critical path if you can to speed up your code

ex.

C:
```
for ( i = 0; i != len; i++) {
    acc *= data[i];
}
```

```
MC:   .L25
        mulsd (%rdx), %xmm0, %xmm0
        addq $8, %rdx
        cmpq %rax, %rdx
        jne .L25
```

dependency diagram = Inputs/Outputs = registers
Nodes = micro-operations

Out-of-Order Execution

- form of parallelism where the compiler will run micro-operations out of order if the user and the source code doesn't know in order to keep all hardware units busy in the CPU
- the compiler will run independent instructions while the dependent instructions are stacked up, therefore keeping the compiler busy

Branch Prediction

- when the machine code encounters a branch due to a conditional jump, the compiler will predict which route the code will take called Branch Prediction based on the last pass through the conditional jump.
- the first pass will be slow and instructions might be invalidated. But the cache will remember what happened the last time the unconditional move happened, so it will use this to guess whether to start decoding instructions at the next address or the new address.

Conditional Moves instead of Branch Prediction:

- branch predictions can cause your program to be very slow when they are wrong so when you can, use conditional moves.
- downside to conditional moves:
    1. in order for a conditional move to be used by a compiler, the following 3 cases must already be computed internally
        1. 1st return value 2. second return value 3. condition
    - this causes a latency bound, creating less chance for parallelism
    - this is a tradeoff for avoiding branch prediction

Getting Data from A --> B in chips:

1. run a wire
2. use a register dropoff point
3. register renaming:
    chip does this so whenever the registers are changed in micro-ops the value wont be copied back to memory until the complete machine instruction is executed.

Spilling:

- suppose you have 30 variables, but only 16 registers!
- compiler will keep your 10 important local variables in registers and the 20 local variables will then share 6 registers
- these variables will "spill" onto the stack

THREAD LEVEL PARALLELISM

(from above)
Goal: reduced time, increased efficiency
General solutions:

1. buy lots of computers, network them
2. multiple processors (Central Processing Unit)
3. multiple cores in 1 chip (quad processor = 4 cores in 1 chip)

****** 4. combination of 1-3

- MIMD (using hardware for parallelism) (below)

5. multiple operations active at one time in one core

- SIMD, ILP (using coding for parallelism)

Using Hardware for Parallelism:

- MIMD multiple instruction, multiple data
  - utilizes thread-based and process-based parallelism
  - fewer program changes, and these changes are more SCALABLE than ILP, giving us more speedup.
- Additional Pros of MIMD
  1. lets us use slow devices at a good pace
  2. defers work while busy
     ex. garbage collecting
  3. fair (ish) access by multiple clients to a single server
  - these 3 are essentially "managing a to-do list"
    a. most important first, least important last
  - concurrency lets us do this with just one core
  - parallelism does this with multiple cores

Concurrency vs. Parallelism:

- Concurrency
  a. idea that an application is partitioned into "flows" and you can run these "flows" concurrently
- Parallelism
  a. actually running these in parallel

Scaling:

1. Strong Scaling:
   - fixed size problem, add CPU's
   - used for real time applications
   - latency
2. Weak Scaling:
   - growing problem, growing CPU's
   - used for server throughput

3 Concurrency Approaches:

1. Multiple Processors (CPU's)
   - computations run on multiple "machines" that are independent of each other
   PROS:
   1. communication is slower (only through system calls) but also safer.
   2. easiest from programmers perspective
   3. scales better (> 1 servers)
2. Multiple Threads
   - thread = a program trying to run on a core in a CPU
   - computation has multiple CPU's, but only 1 shared memory
   - popular in BIG computations
   - PROS:
   1. communication easier and faster since they can directly access the same memory
   2. Speedup is scaled
   3. but this allows for Race Conditions (BAD)
      a. Race Conditions = when multiple threads access the same memory location, and at least 1 thread writes to that memory address (changes value in memory)
      b. sometimes called "threads stepping on each others toes".
3. Multiplexing / Event- Driven
   - one CPU shared amongst multiple threads
   - popular in IoT devices (Internet of Things)
   - PROS:
   1. communication very fast since same memory same CPU
   2. Race conditions rare since same CPU
   3. simplifies code

Threads

Definition:

- Thread = a program trying to run on a core in a CPU
- Core = hardware in a CPU that executes threads

Properties of Threads:

1. Fast IPC (inter process communication)
2. each Thread has its own:
   - %rip, %rax, %rflags, %xmm0
   - Stack
3. But NOT its own memory
   - therefore if the memory location in one thread's stack is accessed / written to by another thread, it will create undefined behavior.
   - this can happen whenever a global pointer or global static variable is used (see below)

(from "pointers" above)
Q: can a pointer ever point into another thread's stack?
- called Cross-thread Stack access
- even if carefully coordinated and without race conditions, cross thread stack access can make your program run unexpected behavior.
- example:

(preparation)

```
int *p;              //global pointer
CreateThread( ..., thread #1, ... );
CreateThread( ..., thread #2, ... );
```

(Thread #1)

```
int q[1024];      //allocated on thread 1's stack
p = q;            //set global pointer equal to
q[0] = 1;                 address on thread 1's stack
```

(Thread #2)

```
*p = 2;           //Thread 1's stack accessed
                  //sets q[1024] = 2
```

- keyword "volatile" for global static variables tells the compiler that the global variable might be changed at any time in the code, so it fixes any potential undefined behavior

Context Switch:
- given 'T' threads and 'C' cores, if T > C, which threads should run when?
  - the operating system decides
  - referred to in class as a "scheduler"
- if a thread has run on a core for way too long (10 milliseconds) the compiler will perform a context switch
- Definition:
  - when a core turns its attention from one thread to another
- Steps in Context Switching:
  1. must save %rip, %rax, %rflags, %xmm0, stack, etc.
  2. must restore the above registers from the next thread
  - this destroys the cache, therefore we want to do context switching as little as possible.
    a. this is because to store the registers in memory, data is transferred from registers --> cache --> RAM and will change the caches when doing so.

Threads in Software:
- pthread_t is a data type
  - integer used to identify the thread in the system
  - must declare this before creating a thread

ex. `pthread_t thread_id;`

- pthread_create( )
  - creates a real life, living thread

- takes 4 arguments:
    1. pointer to thread_id
    2. NULL creates the thread to "default values"
    3. function that new thread will execute
    4. pointer / data we can pass to the thread and it can return to us at the end of the thread if we want.
- after pthread_create() returns, program now has 2 threads

ex. pthread_create(&thread_id, NULL, PrintHello, (void*) t);

- pthread_exit
    - causes current thread to exit and free any thread-specific resources it is using.
    - we must use pthread_exit at the end of the main program, otherwise all the current threads will be killed without returning.
    - the only argument in pthread_exit( ) is a pointer to a void * that is available to any successful join with the terminating thread.

ex. pthread_exit (void* value_to_pass_on)

- pthread_join()
    - thread equivalent to waitpid in processes
    - calling pthread_join blocks the calling thread until the thread specified in the first argument terminates.
        - used for synchronization

    ex. pthread_join(pthread_t tid, void * return_value);

    - if we don't care about a return value, use NULL for 2nd arg.
- pthread_detach( )
    - at any point, a thread is either joinable or detached
    - default is joinable
    - Joinable threads must be killed by other threads to free memory resources
        - using pthread_join(&thread_tid to be killed, ...) for synchronization
    - Detached threads can't be killed and resources are automatically freed at termination
        - if not used for synchronization, better to call pthread_detach(&pthread_self) to ensure it finishes and resources will be freed at termination

-pthread_self( )
- returns the process ID of the current thread

- Compiling a multi-threaded C program:
    - must link it with the pthread library
        : $ gcc hello.c -o hello -lpthread

``````FINAL EXAM```````

INTRODUCTION

_____

Linux Commands:

$ gcc -c foo.c compiles a C file

$ readelf -a foo.o accesses the symbol table of a file

$ gcc -wl, --wrap, malloc resolves undefined uses of malloc with
_ _wrap_malloc (write your own _ _wrap_malloc)

$ export LD_PRELOAD = /home/eggert/mylib.so loads "mylib" first when linking

SYNCHRONIZATION

_____

Goal:

- To eliminate multiprocess and multithreaded applications from stepping on each others' toes and creating problems
  - "aligning" the processes and threads to have them flow smoothly without conflict
- Software Concept

Data Synchronization:

Keeping multiple sets of records of data in order to verify and maintain data integrity.

- Thread and Process synchronization commonly used to implement data synchronization

Simple Synchronization:

- MultiProcessing Functions

fork()

- duplicates a virtual machine
- completely identical virtual memory and registers, but %rax = 0
- each process has it's own process id
- called a "clone" or a "child"
- creates a process tree

waitpid(childpid, ...)

- pauses parent process until children "die" or finish
- this is how we synchronize

exit()

- the return directive at the end of a process
- called by the child
- waitpid will not continue until the process dies, or the child process calls exit().
``````

- MultiThreading Functions (same as before in Threads)
    - Thread Synchronization is the mechanism which ensures that two or more concurrent processes / threads don't cause race conditions
        - if the compiler context switches in the middle of a process or a thread, then the data may affect the program
        - therefore when accessing these critical sections we must make pthread_join(...) to force the compiler to wait until the thread has finished
    - threads belong to processes and cannot access memory outside of their process
    - (see Threads Midterm 2)

pthread_create(...)
- creates a new thread to run in parallel with others
- analogous to fork()

pthread_join(...)
- calling pthread_join blocks the calling thread until the thread specified in the first argument terminates.
- analogous to waitpid(...)

pthread_exit(...)
- causes current thread to exit and free any thread-specific resources it is using.
- analogous to exit()

pthread_cancel(...)
- asks another thread to kill itself

pthread_detach(...)
- causes thread to be non-joinable
- therefore pthread_join(...) will not work on this thread, and the parent won't wait for the thread to finish.

Recall "BIG PROBLEM" with Threads:
- Threads share the same memory, so they can "stomp" on each others feet and memory.
    - happens in global variables accessed by pointers
        - causes Race Conditions

Solutions to BIG PROBLEM:

1) Don't share state among threads
    - basically don't use global varibales or same pointer variable in multiple threads

    i.e. localtime( ) vs. localtime_r( )

    struct tm * localtime(time_t * const);

    struct tm * localtime_r(time_t * const, struct(m*));
        - the _r in localtime_r refers to the function being "re-entrant" vs localtime() which isn't

- since we pass the local temp struct(m*) into localtime_r(...), no shared data is accessed that could be the same between different threads.
    - must have a "re-entrancy" variable for every shared state by threads

- Re-Entrancy
    - Thread-safe functions
    - definition: re-entrant functions do not call any shared data
    - i.e. recursive functions
    - i.e. signal handling
    - i.e. multiple threads (above)

2) Semaphores (Locks)

- shares the state (shared data), but locks the data. If a thread wants to write, the thread needs to gain access to the lock
- each semaphore has an associated integer value with it, which represents the number of spots left for access to the shared state.
- too few locks = race conditions / bugs
  too many locks = bottlenecks / deadlock
    - Deadlock
        - program stops working totally
        - Thread 1 owns resource 1, waits for resource 2
          Thread 2 owns resource 2, waits for resource 1
- initialized by:
    sem_t s;
    sem_init(&s, 0, int (slots available))
        - the 0 indicates that the semaphore can't be shared by processes
- Mutexes = semaphores with only 1 slot
- uses two functions:
    1) P(s) = sem_wait(sem_t *s);
        - checks to see if the semaphore has a vacancy (if it is > 0), and if it does, decrements s then it returns.
        - if it is = 0, then there are no spots left and the thread is then suspended until another thread vacates the lock by a V(s) operation, then the suspended thread is restarted, decreased, and returned.
    2) V(s) = sem_post(sem_t *s);
        - frees a spot on the semaphore and increments s. if there's a suspended thread waiting, it automatically restarts the thread.

ex. using a semaphore to ensure that no element can be removed from an empty buffer (say if both threads were deleting elements at the same time, then one would delete the only element and the other would cause an error)

```
int remove(void) {
        sem_t m, s;
        sem_init( &m, 0, 1)     ;   //mutex
        sem_wait(&m); // locks buf
        int i = buf[front ++ % 128];  //operation
        sem_post(&m); //frees buf
        return i;
}
```

Finding Race Conditions:

- Look for reading / writing to memory where the program could potentially be reading / writing to the SAME location.

Web Servers:

- Background
  - Uses HTTP (hyperlinks act as pointers)
  - Web Client (Browser) opens internet connection to a server and requests some content. The server responds with the content and closes the connection. The browser reads the content and displays it on the screen.
    - content = bytes with associated MIME type
- Web Server format in general:

```
void process ( int fd ) {
        read( fd, ... );
        think( );
        write( fd, ... );
        close( fd );
}
```

- MultiPlexed
  - single CPU, so one request at a time.
  - slowest
- Multiple Processes
  - uses waitpid() and fork()
  - fork()s at every request and creates a new process
- Multiple Threads
  - best approach
  - avoids fork(), uses parallelism
    - fork() will clone the entire process, and therefore create a bottleneck when accepting file connections
    - parallelism and threads acts the same way but w/o all the copying and cloning.

- 2 Approaches
    1) use 1 thread per core in machine
        - best approach
        - too many threads = scheduling slower
        - too few threads = extra CPU's aren't working
    2) call pthread_create at each request and pthread_exit when done
        - slower
        - pthread_create and pthread_exit have overhead, which would be scaled to the amount of requests made to the server.
        - a constant n threads (approach A) is faster because it eliminates the scaled overhead cost.

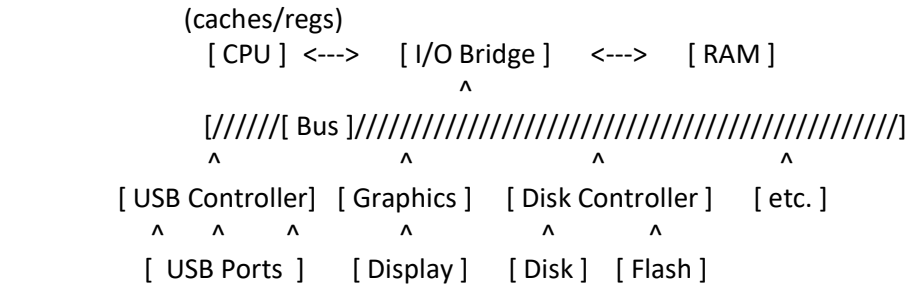
I/O

Introduction:

- I/O stands for input / output and is the section of this course related to computer interaction with the outside world
    i.e. user input, keyboard, mouse, USB ports, camera
- Programmers are responsible for moving data between disk and memory through file I/O, while the hardware is responsible for moving data between memory and caches.
- I/O is the biggest bottleneck to most programs today
    - CPU is extremely fast, bit sending / recieving data to and from the CPU is the real time cruncher
- Big Picture:
    - Multiple Levels of Abstraction in I/O
        - abstraction = the principle that higher levels don't have to worry about the gritty details of the low levels, just the essential ideas of the task being completed
            - user types on keyboard, will only see the characters appear on the screen
            - requires intuition and measurements

    { user } complication of lower levelsunseen at
    { application / program } higher levels of abstraction
    { Operating Systerm }
    { ... }
    { hardware } complicated at low level

Visual:



Disk Controller:

- an interface between the disk drive and the bus connecting it to the rest of the computer system.
- disk controller enables the CPU to communicate with a hard disk, floppy disk, or other kind of disk drive.
    - converts the signals and data read from the disk drive into data able to be transmitted across the bus and read by the CPU
- Disk Drives
    - disks of memory that must be read like a record player
    - to decrease latency of reading from disk drives:
        1) faster read arm
        2) faster rotation
        3) denser recording on disk
        4) larger caches
        5) faster bus
        6) more disk drives in parallel
    - AFR = annual fail rate
        - statistic of disk drives specific to each kind
        - tallies device failure rate
            - hard errors (non-recoverable)

The Bus:

- Strip of wires in hardware
- connects all hardware devices to each other
    - how data, addresses, signals, etc. are passed around
- 1 way system
- Wires in Bus:
    1) Control Lines
        - links CPU to memory and to the bus
        - many signals used on control lines
        1) Clock
            - generates pulses at a constant rate to synchronize CPU and bus operations
        2) Interrupt
            - used to signal an interrupt, which suspends the current CPU process

3) Read
- signal from CPUI instructs device to retrieve data from a specific memory location

4) Write
- signal from CPU instructs device to send data to a specific memory location

2) Data Lines
- carries the data to be transferred from A -> B
- larger data line = more data can be sent in 1 clock cycle
  - better performance
  - better throughput

3) Address Lines
- holds locations of B, where the data in the data line should be sent
- larger address line = more addresses can be accessed by the CPU

- Who can use the Bus?
  - Bus masters
    - devices that can initiate data transfer or directions on the bus
    - control of the bus switches after bus operations finish
  - Slave devices
    - all other I/O devices connected to the bus
    - only respond to requests from bus master
- Using the Bus:
  - program wants to write data to disk
    1) CPU gains control of the bus
    2) CPU sends a message to disk controller via the bus
       "please copy data from memory location M to disk location D"

    ----------CPU free - can now work in parallel----------

    3) Disk controller gains control of the bus
    4) Disk controller sends message via the bus to DRAM
       "please tell me what's at memory location M"
    5) DRAM gains control of the bus
    6) DRAM sends data at memory location M to disk
    7) DRAM recieves the memory, copies it into disk location D
  - How do devices send commands to a controller?
    - Control Lines
      read, write lines
    - special instructions
      outb, inb (x86-64 specific)
    - memory-mapped
      issuing mov and leaq instuctions with a special address that points to the bus, not RAM
- How to tell when Bus operation is done?

A) Polling
- CPU asks disk controller if it's done yet

- sends message to controller asking for status
- quick response time (low latency) BUT the more you poll, the more CPU time is wasted.

B) Interrupts
- Disk controller sends an interrupt through the control line to signal to CPU that the bus is done.
- When the signal reaches the CPU, %rip is updated to an address pointing into the interrupt service vector (list of tags distinguishing different interrupts)
- analogy = tornado warning on tv, stops the current process

- Bus Performance Issues
- Contention for the Bus
- When more than one process needs to send data between pieces of hardware

1) detect the collision
2) stop transmitting
3) wait a bit (time chosen randomly)
4) retry

- analogy = when two people call each other at the exact same time and get the busy signal

- Size of data sent on the Bus is hardware device specific

ex. optical disks = 2048B, traditional disks = 512B, etc.

- if data sent is too small, Bus time is wasted
- takes up bus time to preform a small task when it could be used for better things
- if data sent is too large, Bus time is also wasted
- backs up the bus with large operation

- Cache Coherency with Busses
- Disk controller modifies RAM through the bus using a DMA
- DMA = Direct Memory Access (no CPU involved)
- if location was cached, the cache is now wrong!
- called a Cache Coherency Problem
- Solutions:

1) don't cache I/O data that could be changed
2) invalidate cache by "snooping"
- x86-64 technique
- hardware included to "snoop" on the bus and watch for writes into RAM, then invalidate that block in cache
- can be done in parallel by CPU

3) flush the cache block to RAM before allowing I/O to write to RAM itself
- flushing = evicting

Metadata:
- Data that provides information about other data
- data describing files/programs on computer

- makes finding and using the data easier in system calls
- we can access a file's metadata by calling stat and fstat:
  int stat(const char *filename, struct stat *buf);
    - takes a filename
  int fstat(int fd, struct stat *buf);
    - takes a file descriptor

System Calls:
- The operating system is the first program that gets executed when turning on your computer, and your program is simply a program running inside the larger OS program.
- System calls allow programs to talk to the operating system
  - looks like a procedure call, but it is really a request for the OS to perform a function / procedure.
  ex. open, close, read, write, etc.
- System calls are EXPENSIVE
  1) computer saves its state
  2) let the OS take control of the CPU and perform some function
  3) OS saves its state
  4) CPU takes control back from OS and continues
- Basic system calls:
  - open
    - requests OS to use a file signified by a file descriptor (int fd)
  - close
    - tells the operating system you are done with the file fd
  - read
    - tells the operating system to read 'x' bytes from the opened file fd
    - returns the number of bytes actually read
  - write
    - same as read, but instead returns the read bytes
- Procedures vs. System Calls
  - The stdio (POSIX) library functions are procedures that implement the basic system calls above.
    ex. fopen() calls the system call open
  - Procedures serve as a level of abstraction, so the user / programmer does not neccesarily have to understand the details of the system call, but could call a simpler version of it.

I/O Synchronization and Timing:
- Goal:
  - to synchronize CPU and bus operations when dealing with I/O
- Synchronous Busses:
  - uses the clock signal on the control line in order to synchronize CPU and bus operations
  - high throughput, since operations are synchronized, but have small delays since operations must wait microseconds for specific CPU clock cycles.

- Asynchronous Busses:
    - No clock used
    - lower throughput since CPU and bus operations can stack up, but there is no delay from a CPU instruction and the bus operation
- Synchronous System Calls
    ex. int i = read(fd, buf, x)
    - returns only after the OS reads 'x' bytes from the file fd
    - this causes the program to 'hang' for some time before the read is complete
- Asynchronous System Calls
    ex. aio_read(...), aio_write(...)
    - initiates the OS request and then returns immediately
    - no hang time, but you now must check the status of you system call:
        aio_error(...) returns the status of an enqueued request
        aio_return(...) returns the status of a completed request
      or use other commands:
        aio_suspend(...) suspends program until request completes
        aio_return(...) attempts to cancel an enqueued request
- Line (wire) number delimma:
    - increasing number of wires in hardware increases the cost that each processor / computer will be
    - increasing number of wires also increases throughput and speed

VIRTUAL MEMORY

Introduction:
- |Virtual Memory| > |Physical Memory|
    - coders don't have to worry about memory restrictions
- In machine code, the CPU sends and receives virtual addresses that are too big for RAM:
    %rdx = 0x7fff0f0f0813af3
- CPU thinks it's accessing the virtual address in RAM, but it's really accessing a different address in physical RAM
- How does this work?
    - Physical memory partitioned into equal sized page frames which map to pages in each process' page tables.
- How does RAM hold all the process' page frames?
    - Not all pages are mapped at all times
    - Operating System can decide that one process' needs are less important than the current ones, and will free up some space from their frame to use in the current one.
        - to the CPU, both of the these programs have distinct virtual memory spaces and their addresses are different, but they use the same hardware.

- this is done by storing the page back to flash (or disk) until a page fault brings it back into the TLB / caches.
    - how does the RAM change its mapping to another virtual address?
        mmap function (below)
- Allows us to run mutliple processes on the same hardware but that access different virtual memory.

Mapping Virtual Addresses to RAM

- Each process uses its own virtual memory space
- the MMU (memory management unit) is in charge of translating the virtual addresses to physical addresses
    - this is all blind to the CPU, that just works with virtual memory addresses and registers, sending them to the MMU
    - one level of abstraction
- Page Table
    - analogous to an index in a book (RAM) that tells you what page in the book (RAM) corresponds to what topic (virtual address)
    - data structure stored in RAM
        - maps the high order bits of the virtual address to the high order bits of the physical address
            - page index = bits above 12 bit offset
        - each process has it's own page table since each process has its own RAM
    - Multi-Page Tables
        - page table entries point to more page tables (like a tree)
        - indexed by the higher order bits of virtual addr
        [unused highest 12 bits]
        [top level page index 9 bits]
        [2nd level page index 9 bits]
        [3rd level page index 9 bits]
        [4th level page index 9 bits]
        [12 bit offset]
        - enables smaller page tables and faster searching for a page table entry
- 12 Bit Offset
    - each bit in the 12 bit offset describes the memory in RAM

bit	description
0	p bit - (1) if page is in RAM (0) if page fault
1	r/w bit - (1) for read/write (0) for read only

3	wt bit - (1) write through (0) write back
5	a bit - (1) any access (0) not
6	dirty bit - (1) has been written to (0) has not been
8	global bit - (1) keep if context switch (0) don't
12 - 51	used to find page index
52 - 62	unused
63	XD bit - (1) execution disabled (0) execution enabled

- if bit 0 is 0 (p bit), the binary representation that follows is replaced with the flash address for the MMU to dig it back into the TLB and RAM

- Cache for Page Tables:
  - TLB (translation lookaside buffer)
    - associative cache of recent page table entries from virtual to physcial addresses
  - we need the TLB to keep virtual memory efficient
    - otherwise we would need to access 4 page tables, and potential page faults to access each physical address
- Mapping flow
  1) MMU searches TLB
     - any point after TLB miss, if there is a hit, the p.t. entry is cached into the TLB and the faulting instruction is rerun.
  2) MMU searches L1-L3 caches
     - just like any other memory access, search down the caches until RAM
  3) MMU searches Page Table (in RAM)
  4) If there is a Page Fault (Page Table miss) the page was relocated to another process and must be put back into RAM (see Translation Failures)
- Translation Failures
  - lookup will fail in the following 2 cases:
    1) Segmentation Fault
       - virtual address is invalid
       i.e. user error
    2) Page Fault
       - page is not currently mapped to a location in RAM, must be drug out of flash / disk back into RAM
       - takes thousands of times longer than accessing RAM

Starting a Program:

- starting a program is kind of like running a smaller program within your larger source code
- computer iterates down the pages in your programs' page table (seperate from the operating systems' since it's a seperate process)
  - recall bit 63: execution disabled bit
  - at the beginning, only the pages to set up your program will be marked "enabled", and therefore the computer will first initialize:
    - [text] pages read-only (code/data/constants)
    - [data] initialized data
    - [bss] initialized to 0
    - ...
    - [stack] stack

Virtual Space in Processes:

- within each process space, there's 2 sides
  1) Virtual Space
     - virtual addresses that map to page frames in physical memory (RAM)
     - contains text, stack, data, bss, heap, etc.
       - normal parts of program/process
     - 0x0 (null pointer) to 0x7fffffffff
       - since the most significant bit is 0 (hence 7), these addresses are positive.
     - contain forbidden zones
       - page frames inaccessible by the current process
       - pages that the least significant bit (p bit) = 0
  2) "Negative Address" Kernel Space
     - visible to the kernel only
       - kernel = Operating System module in charge of memory management
     - contains page tables
       - wait, I thought page tables are stored in RAM?
         - they ARE stored in RAM but they are only visible to the kernel (OS), so they appear in the process' "negative address"

```
                [/////////////]     : null pointer ( 0 )
                [     text    ]     : read only data ( code/constants )
                [     data    ]     : initialized data
                [      bss    ]     : initialized to 0
                                     : ( heap space )
                [    heap     ]     : heap ( space to be used by malloc )
                [/////////////]
                [/////////////]     : forbidden zone    ( page frames
                [/////////////]                           used elsewhere )

                [/////////////]     : forbidden zone for stack overflow
        0x7ffffff[    stack    ]    : stack
          ------------------------------------------------- ( below visible only to OS )
        0xffffff0[   kernel    ]    : Operating System and its data
                [    Pages    ]     : Page Tables ( multi layer )
                [ Page Table ]      : points to Page Table ( par of RAM )
                [/////////////]     : forbidden zone for stack overflow
                [ kernel stack ]    : kernel stack
```

Implementation of Virtual Address Mapping:

- mmap function
  - how the operating system re-maps RAM space to different processes and virtual memory spaces
    - recall: same hardware used for multiple processes, although at different times.
    - to the CPU, all virtual memory has a RAM counterpart
  - mmap(void* start, size_t len, int prot, int flags, int fd, off_t offset);
    - start stores virtual address of the page whose mapping will be changed
    - len is the number of bytes affected
    - prot stands for "protections", which signals the 12 offset and execution diabled bits to be set
      - none = forbidden zone
    - flags represent how the page will be used among other processes
      1) MAP_ANON
         - new, zeroes pages
      2) MAP_PRIVATE
         - private copy of a part of a file on flash
      3) MAP_SHARED
         - shared among processes
    - fd is the file from flash to map into RAM
    - offset is the byte offset into file fd

- Uses of mmap:
    1) Virtual Memory
    2) Crashing programs on purpose
        - if you remap a page to a forbidden zone and the compiler tries to access the memory, the program will crash
    3) Sending messages between processes w/o copying
        - some operating systems used MAP_SHARED to pass messages from one process to another without having to use a signal or copy it into a buffer
            - by remapping to the location of the data in RAM to be sent with MAP_SHARED, the data already exists in the recieving process' RAM
    4) Malloc
        - recall: malloc is the "new" equivalent in C
        - remaps a forbidden zone to an "allowed zone" on the heap.
            - zeroes out the amount of memory in the malloc call in RAM and remaps this page to that process
        - for every malloc(p), there is a free(p) that will give the RAM space back up for grabs
            - if another call to malloc(x) is made in the same process, the page frame from malloc(p) will not be zeroed
                - optimized by operating sys.
    5) Linking
        - piecing together all the files in a program
        - (see Linking)

LINKING

Introduction:

- Linking pieces together all the files in a program
    - recall CS31 & CS31 using #include "student.h" header files and libraries
- when you call a function, the linker "cuts in" the code from the header file or the library in order to make the code flow sequentially for the compiler
- all the Tables, text, read only data, data, bss, debugging info, etc. resides in an object file (hello.o)
- Why Link?
    1) Quicker compiling after bug fixes
        - when you edit a cetain file the compiler will only have to recompile the file you changed, not the entire program will all the libraries and header files inlined

String Table:

- A table that holds all the constant strings defined in a file
- Strings are referenced by an offset into the String Table
    - ex. int name; in the symbol table entry is the byte offset into the string table - a reference to the name of the function.

Static Linking:

- Linking occurs at compile time
- Uses Symbol Table (what) and Relocation Table (where)
- Linker "inlines" the missing symbols from libraries into the C file
    - does this by referencing their Symbol and Relocation Tables

Pros:

- faster runtime
- doesn't require a linked library where it's being run
    - more portable

Cons:

- longer compile - time
- requires more disk space and memory
- lots and lots of copying
    - creates a bottleneck in files that use libraries alot
- must link all missing symbols, even if they are never called by user

Symbol Tables:

- Data structure attached to each file that is accessed by the compiler during program execution to find locations of symbols in the file
    - stores information about the occurence of various objects referenced (when linking header files or libraries)
- When the linked combines lib.a and hello.o, it reads the symbol tables and fills in the missing values in hello.o with the definitions in lib.a
- Implementation of a Symbol Table:
    - Symbol tables can be implemented in 3 ways:
        1) Linear List
        2) Binary Search Tree (BST)
        3) Hash Table
- Symbol Table Entries:
    - Only list the unknown AND known procedure calls / variables, not how many times the calls are made or their missing locations in the assembly code.
        - This is kept track of in the Relocation Table (see below)
    - external global variables (allocated statically)
        ex. global variable from an included header file
    - external function addresses (library function)
        ex. printf( ) from stdio.h

- static functions and variables
    - functions defined only in current file that are marked "static" signifying they can't be accessed outside the file
    - linker creates funny names for these so they won't collide
- x86-64 Implementation of Entries:

```
struct Entry {
        int name;               //offset into the string table
        char type: 4    //data, text, function, etc.
              binding: 4  //static local or global
        char reserved;
        short section;  //section number
        long value;     //offset from section start
        long size;              //number of bytes
};
```

so the symbol table appears as:

VALUE	SIZE	TYPE	BIND	INDEX	NAME
0	2c	FUNC	GLOBAL	1	main( )
0	0	NOTYPE	GLOBAL	UNDEF	printf( )
...	...	...	...	...	...

- main( ) represents the main function in a c file, say hello.c
  the index is 1 because it is the first thing executed, type function, the entire function takes 2c bytes, and its value is 0 because it is offset 0 from the start of the section
- printf( ) represents a missing entry (one from a library)
  that must be retrieved from another file during linking.
  the value, size, type, and index are all null values because nothing is known about printf( ) from the hello.c file

- Strong vs. Weak symbols
    - By default, a symbol in an object file is Strong.
    - When two functions collide because they have the same function name, the compiler defines two definitions of the function:
      ex. socket( ) appears in both files, so compiler defines two definitions:
      _ _socket( ) strong definition
      socket( ) weak definition
    - weak symbols do not need definitions, but strong ones always do.
    - weak symbol definitions defer to stronger ones

Relocation Table:

- List of entries that hold pointers to each missing function / global variable reference in the file, so that the linker can fill in each of these when linking
  ex. 30: call printf
  relocation entry:
  printf T[30]

Static Linking Steps:

- given a C file linked to a library file <stdio.h>

1) Symbol Table of C file contains all the used symbols in C file and their data:
   static functions, missing functions, static and missing global variables
2) Relocation Table of C file contains all the offsets into the C file that reference missing variables or functions.
3) Symbol Table of library file contains all the library functions and their offsets
4) At compile time, when compiler encounters a missing symbol in the C file, Linker looks at C file Symbol Table and finds the function def. in the library file
5) Linker copies the function and inlines it at every instance of the symbol in C file's relocation table.

Dynamic Linking:

- Linking occurs at run time
  - "Lazy Linking"
    - links the function at the first call, not before like static
- Uses GOT (real addresses) and PLT (addresses of instructions to call correct GOT entry)
  - also called Jump Tables since the compiler will also use the addresses in the GOT and PLT to jump to the virtual address of the shared library function which was mmapped into RAM by dynamic linker
- Instead of inlining the missing function, dynamic linking inlines a "dynamic linking module" address
  - ex. callq 0x4005c0 <getChar( )>
  - when compiler comes across this address, %rip goes through a series of jumps, which essentially:
    1) instructions go in a circle the first time dynamically linked
       - causes steps 2 and 3 to run
    2) push relocation address onto the stack
    3) calls GOT entry that holds the dynamic linker
       - changes the function GOT entry to the remapped getChar( ) address
    4) runs the GOT entry with the correct remapped getChar( ) address
    5) pops the relocation address and continues

  the first time the function is dynamically linked, and every time after that:
    1) jumps to PLT entry instruction which runs the correct GOT address that points to remapped function in virtual memory
    2) pops the relocation address and continues

Pros:

- faster compile time
- multiple files can share one copy of a library
- uses mmap / jump, not copying
- only links the missing symbols that are called

Cons:

- slower run time

Global Offset Table (GOT)

- separate GOT for each process
- stored in read-only section of object file
- All addresses of symbols logged in the GOT
  - used to find addresses of missing global variables and functions unknown at compile time
  - GOT entries are the addresses to remapped functions in VM after dynamically linked the fist time
    - Lazy Linking
    - first time, GOT entry is an address that will push the relocation address onto the stack and then call the GOT entry that points to the dynamic linker, which will update the GOT address to be the remapped function in VM.
  - This is why the GOT is the target of hackers, so they can put the address to their "spy" function instead of a library function or something else (see ASLR below)

Procedure Linkage Table (PLT)

- Table of addresses to instructions that call the correct GOT entry for a specific function.
- Lazy Linking with PLT
  - The first time a function is dynamically linked, the PLT entry can't be directly called because the GOT entry associated with the PLT call won't hold the remapped address for the function.
    - this "wrong" address stored will be the address that pushes the relocation address and eventually calls the dynamic linker
  - after the dynamic linker runs, the GOT entry will now point to the correct remapped function address, and therefore the PLT entry will now point to the instruction that calls the correct GOT entry.

Dynamic Linking Walkthrough:

Start:

```
callq  0x4005c0 <getChar( )>                    (1)
```

Instructions:

```
0x4005a0        pushq       *GOT[1]                         (6)
0x4005a6        jmpq *GOT[2]                        (7)
0x4005c0        jmpq *GOT[4]                        (2)  (9)
0x4005c6        pushq       $1                      (4)
0x4005cb    jmpq        *PLT[0]                     (5)
```

Global Offset Table:

```
GOT[1] =  &(relocation)
GOT[2] =  &(dynamic linker)                         (8)
GOT[4] =  0x4005c6 ( before dynamic linker )        (3)
          0x709a36c4 ( after dynamic linker )       (10)
```

Procedure Linkage Table:

```
PLT[0] =  0x4005a0
```

Steps:

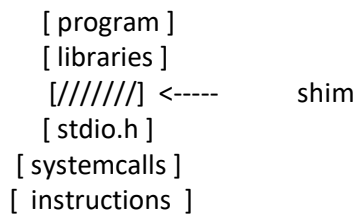
1) callq 0x4005c0 <getChar( )>
   - 0x4005c0 address of PLT[0]
2) %rip = 0x4005c0 = jmpq *GOT[4]
   - the address stored at 0x4005c0 is a jump to GOT[4]
3) %rip = GOT[4] = jmpq 0x4005c6
4) %rip = 0x4005c6 = push $1
5) %rip = 0x4005cb = jmpq *PLT[0]
6) %rip = PLT[0] = 0x4005a0 = pushq *GOT[1]
   - GOT[1] = address of the "relocation" module
   - PLT[0] pushes the address of the "relocation" onto stack so when the remapped getChar( ) finishes running, it can pop the relocation and continue, linking getChar( ) into the right spot.
7) %rip = 0x4005a6 = jmpq *GOT[2]
8) %rip = GOT[2] = address of dynamic linker
   - at this point, the dynamic linker runs since %rip now points to the first instruction of the dynamic linker. Dynamic linker goes back to GOT[4], and changes the address 0x4005c6 to 0x709a36c4, the real address of the getChar( ) function remapped into virtual memory
9) %rip = 4005c0 = jmpq *GOT[4]
   - notice this is the same instruction as step 2), except now the GOT[4] entry points to the remapped getChar( ) instead of the instructions that lead to PLT[0]
10) %rip = 0x709a36c4 = getChar( ) remapped into virtual memory

Address Space Layout Randomization (ASLR)

- When the dynamic linker re-maps library functions into virtual memory or creates a GOT for a process, the linker picks a virtual address at random to store them at.
- Why?
  - the GOT holds the addresses of the functions that actually run in your programs
  - This is why the GOT is the target of hackers, so they can put the address to their "spy" function instead of a library function or something else
- Attackers don't know where the remapped functions or the GOT lies in memory
  - the more randomization that occurs, the safer your program.

Library Interpositionings and "Shims":

- Shims
  - shims are a way to debug, gather data, or spy on programs using the interpositioning of libraries
  - visual:



- small cases: write
  #define malloc my_malloc
  above source code, then define my_malloc in the source code:
  void* my_malloc(sizeof_t t) { ... };
- Static Linking Shims
  $ gcc -wl, --wrap, malloc
  - resolves undefined uses of malloc( ) with _ _wrap_malloc( )
  - #define your _ _wrap_malloc( ) above your program to do what you want to do
    - total up how much space you allocate throughout an entire program, or gather other data regarding malloc
    - don't forget to include the actual call to malloc( ) in your new function, so your program will still work.
- Dynamic Linking Shims
  $ export LD_PRELOAD = /home/eggert/mylib.so
  - loads "mylib.so" first before any other libraries
    - modifies the symbol binding order
    - this tells the dynamic loader "mylib.so" should be the first library to search when looking up symbols
      - modifies the symbol binding order
      - foo( ) declared in mylib.so will take precedence over foo( ) declared in a later library
  - presents a security issue
    ex. login program shim that saves username and password to a site

EXCEPTIONS AND ERRORS

Exceptions:

- Exceptions are rare events that are triggered by the hardware and force the processor to execute an exception handler
- floating point values are exceptional values
  - recall tiny numbers, infinities, NaNs
- getChar( ) which returns a byte 0 - 255 or EOF(-1)
  - end of file EOF(-1) is an exceptional value
- After running the above two, we must run a conditional in order to find out if the exceptional case was used or not.
- in CS31 and CS32, our version of exceptional control flow was try ... catch
  - how do we implement this in C, so that we don't have to check a condition after every calculation for getChar( ) and floating point?

Exceptional Control Flow (Software)

- We "try" and "catch" in C by using the following functions in our code:
- int setjmp(jmp_buf j);
  - initializes j so we can "throw" to it later
  - returns the value in %rax
  - can return multiple times

- this is because longjmp restores the %rip and the stack from the initial setjmp call that pushes these values onto the stack

- void longjmp(jmp_buf j, int v);
  - restores %rip and the stack from the setjmp initialization and stores int v into %rax
    - therefore will re-run the code at setjmp( ) and call the exceptional code
  - caller's locals might be ruined by calling longjmp if it jumps over any register modifying code
    - use keyword "volatile" on the variables that might be changed in exceptional code
- since setjmp can return multiple times, it is the caller's responsibility to call longjmp only before setjmp returns

ex.
```
if ( setjmp( j ) == 0 ) {
        ordinary code;   //"try"
        foo( );
} else {
        exceptional code;          //"catch"
}
```
- if foo( ) contains a longjmp( ), it will change rax = v, and when setjmp( ) is re-run, the exceptional code will be executed.

True Exception Handling (Hardware)
- Exceptional cases = rare cases, and rare cases = slow cases
- Exceptional transfer is slow because it works poorly with caching
  - ex. Context switch (exceptional case)
    - slow b/c caches are blown
- General Idea: (bad instruction)

```
[ ] [ ] [ ] [ ] [ ] [ X ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]       //instruction trace
                ^
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]  //fixup instructions run, then bad
                                                instruction can be rerun or skipped
```
- After a faulting instruction, you can:
  1) retry a bad instruction
  2) skip a bad instruction
  3) fail (shutdown / crash)
  4) context switch to another thread
- Things that can cause Hardware Exceptions:
  - I/O device interrupt
  - Traps
    - i.e. division by 0, breakpoint, etc.
    - traps are addresses that point to interrupt handlers
    - implemented by Interrupt Service Vector
      - Data structure that associates a list of interrupt handlers with a list of interrupt requests
  - Faulting instructions
    - ex. movq $0, 4(%rbx)
    - also called "Page Faults"
  - Aborting instructions
    - happens during hardware malfunctions, infinite loops, etc.

- System Calls
    - invoked by a trap instruction or a syscall
    - %rdi, %rsi, %rdx, %r10, %r8, %r9, %rax inherently used as arguments
    - destroys %rcx and %r11, loads return value in %rax
- 2 types of exceptions:
    - Asynchronous Exceptions
        - can occur any time between any pair of instructions
        - usually caused by devices external to the processor / memory
            i.e. I/O Interrupt
        - asynchronous because can be handled after the current instruction instruction finishes
            - makes them easier to handle since they don't have to be handled in a distinct order
        examples:
            I/O Interrrupt, hardware malfunction, power failure
    - Synchronous Exceptions
        - Exceptions that are triggered by a machine instruction / program
            - occur within the instruction
            - harder to handle than asynchronous because the current instruction must be stopped and then restarted
        - examples:
            systemCalls, breakpoints, page faults, integer overflow, etc.

Signals:

- Software generated interrrupt sent to a process from another process or the OS
    - 1 level of abstraction above hardware interrupts
    - limited form of inter-process communication
- Signals can temporarily delayed, and some can even be ignored by recieving process'
- Signal calls are NOT queued
- Signals can even interrrupt System Calls
- Sending Signals:
    - unsigned alarm(unsigned sec);
        - delayed kill( ) after the number of seconds in paramenter
    - int kill(pid_t p, int signal);
        - pid_t is the process ID of the recieving process
        - signal is the signal sent to the process
            - SIGKILL (kill signal)
                - can't be handled by recieving process
                - terminates immediatley
                - potential loss of data
            - SIGINT(interrupt signal)
                - can be ignored or delayed by recieving process
                - "gracefully terminating"
            - SIGTERM (terminate signal)
                - cannot be ingored, but can be delayed
                - terminates after files closed, memory re-allocated

- Recieving Signals:
    - Processes recieve signals through signal handlers:
    - sighandler_t sighand(int signal, signhandler_t handler);
        - signal is the signal recieved by a process
        - handler is a function that will handle the signal
        ex. signal(SIGINT, foo( ));
            - runs foo(int x) with the signal passed to it
    - Application of Signal Handlers:
        - when writing your own program, you can write your signal handlers so when you send signals across processes, you get to decide how the processes will respond to the signals.