Machine-Level Programming I: Basics

Today: Machine Programming I: Basics

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

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.

Today: Machine Programming I: Basics

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

Definitions

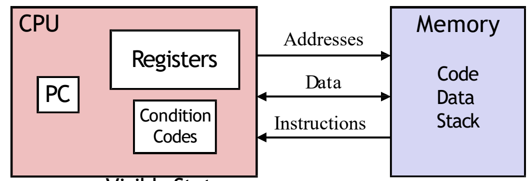
- Instruction Set Architecture (ISA): The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Code Forms:
  - 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

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

5

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

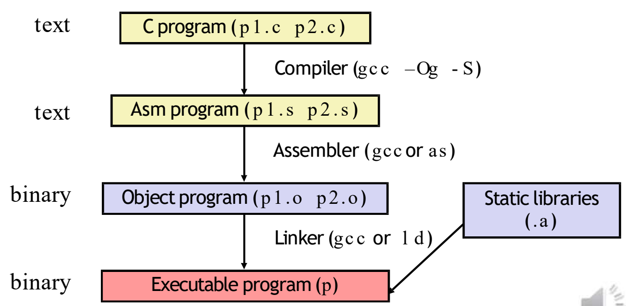
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

6

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



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

7

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

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

8

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 9

9

Object Code

Code for sumstore

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

- Total of 14bytes
- Each instruction 1, 3, or 5bytes
- 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

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 10

10

Machine Instruction Example

```
*dest = t;
```

- C Code
  - Store value t where designated by dest

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

- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64parlance
  - Operands:
    - t: Register %rax
    - dest: Register %rbx
    - *dest: MemoryM[%rbx]

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

- Object Code
  - 3-byte instruction
  - Stored at address 0x40059e

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 11

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 ofinstructions
  - Produces approximate rendition of assemblycode
  - Can be run on either a.out (complete executable) or .o file

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 12

12

Today: Machine Programming I: Basics

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

13

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)

14

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

15

movq Operand Combinations

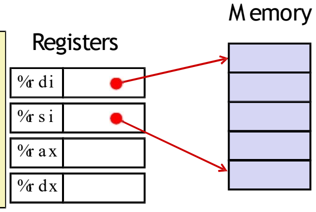
	Source	Dest	Src,Dest	C Analog
movq	Imm	Reg	movq $0x4,%rax	temp = 0x4;
		Mem	movq $-147,(%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax,(%rdx)	*p = temp;
	Mem	Reg	movq (%rax),%rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

16

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
```


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
```


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]]

Today: Machine Programming I: Basics

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

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
```


21

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	

orq

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

22

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

23

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

24

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)

Registers

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

Current stack top

%rip Instruction pointer

CF ZF SF OF Condition codes

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
  - testq Src2, Src1
    - testq 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
```

Register	Use(s)
%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;
}
```

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

13

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

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

14

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;
```

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

15

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
```

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

16

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;
}
```

Register	Use(s)
%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 Version

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

Goto Version

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

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

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
}
```

Jump Table

```
jtab: Targ0
      Targ1
      Targ2
        •
        •
        •
      Targn-1
```

Jump Targets

```
Targ0:   Code Block 0
Targ1:   Code Block 1
Targ2:   Code Block 2
   •
   •
   •
Targn-1: Code Block n-1
```

Translation (Extended C)

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

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

32

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
```

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

35

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;
}
```

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

36

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
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

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

39

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 jumptables

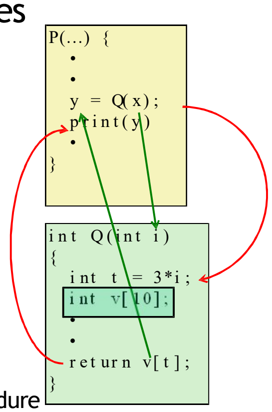
53

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 implementedwith machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



55

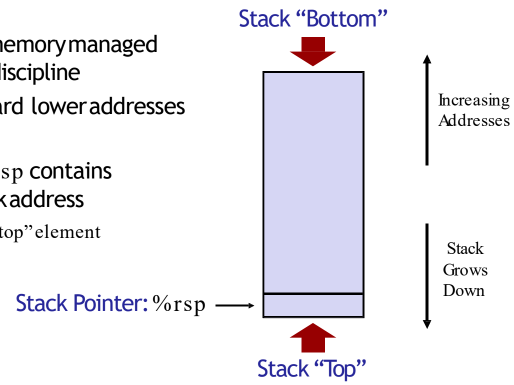
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing localdata
  - 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

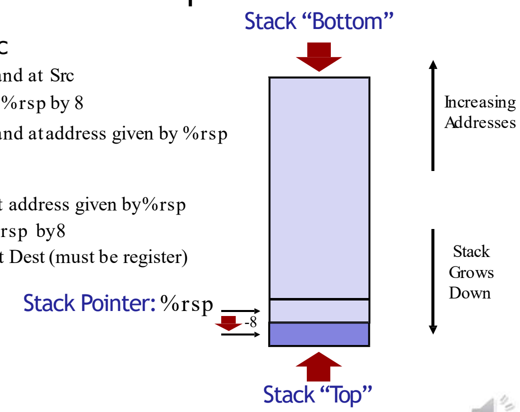


Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 4

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)



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 5

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
```

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 8

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 9

60

Procedure Data Flow

Registers

- First 6 arguments

%rdi
%rsi
%rdx
%rcx
%r8
%r9

- Return value

%rax

Stack

• • •
Arg n
• • •
Arg 8
Arg 7

- Only allocate stack space when needed

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 15

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;
}
```

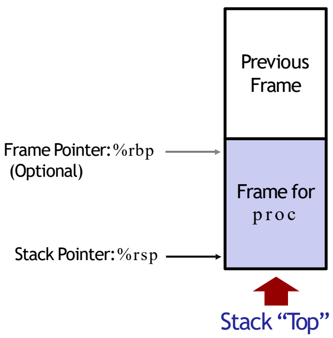
```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov   %rdi,%rax     # a
  400553:  imul  %rsi,%rax     # a * b
  # s in %rax
  400557: retq                 # Return
```

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 16

62

Stack Frames

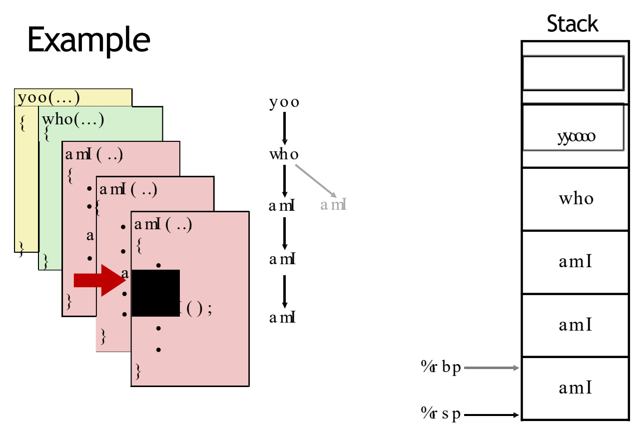
- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by call instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by ret instruction



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 20

63

Example

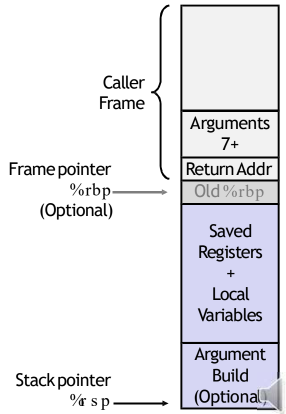


Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 25

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



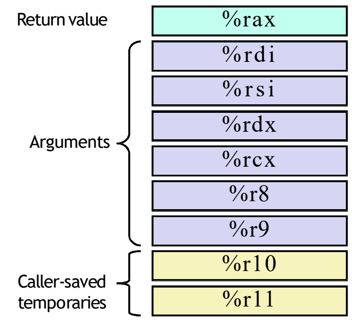
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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”

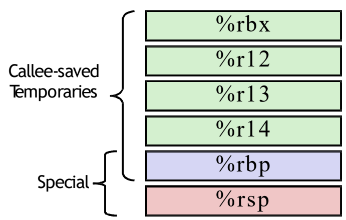
Caller saves temporary values in its frame before the call

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

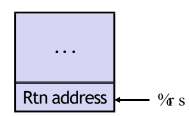
67

Callee-Saved Example #1

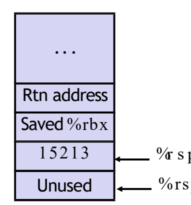
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

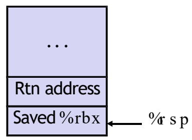
68

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

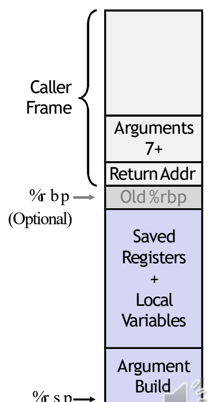


Observations About Recursion

- 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

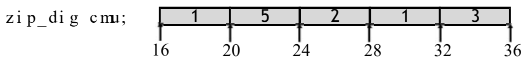
x86-64 Procedure Summary

- 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



Machine-Level Programming IV: Data

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)

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 6

73

Array Loop Example

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

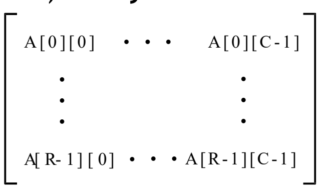
```
  # %rdi = z
  movl    $0, %eax          #   i = 0
  jmp     .L3               #   goto middle
.L4:                        # loop:
  addl    $1, (%rdi,%rax,4) #   z[i]++
  addq    $1, %rax          #   i++
.L3:                        # middle
  cmpq    $4, %rax          #   i:4
  jbe     .L4    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

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 7

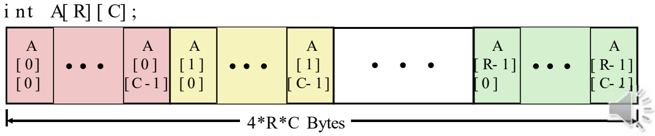
74

Multidimensional (Nested) Arrays

- Declaration
  - T A[R][C];
  - 2D array of data type T
  - R rows, C columns
  - Type T element requires K bytes
- Array Size
  - R * C * K bytes
- Arrangement
  - Row-Major Ordering



int A[R][C];



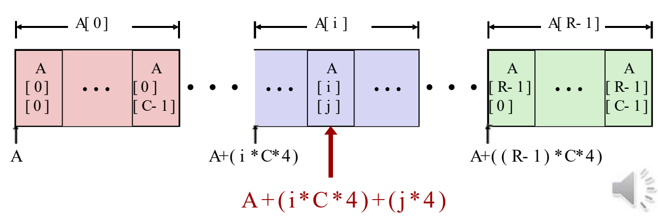
Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 8

75

Nested Array Element Access

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

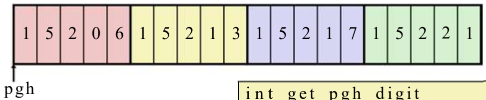
int A[R][C];



Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 12

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)

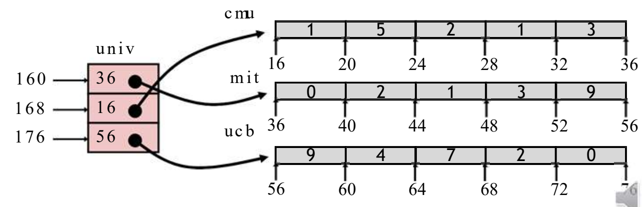
77

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};
```

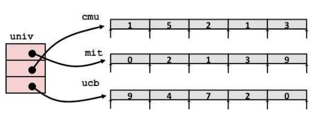
- Variable univ denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of int's



78

Element Access in Multi-Level Array

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



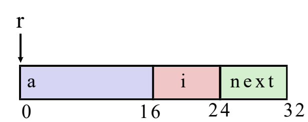

```
salq    $2, %rsi              # 4*digit
addq    univ(,%rdi,8), %rsi   # p = univ[index] + 4*digit
movl    (%rsi), %eax          # return *p
ret
```

- 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;
};
```

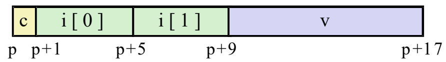


- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding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

80

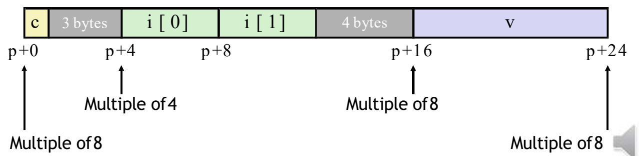
Structures & Alignment

- 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)

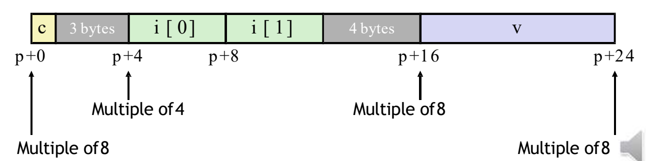
- 1 byte: `char`, …
  - no restrictions on address
- 2 bytes: `short`, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: `int`, `float`, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `long`, `char *`, …
  - lowest 3 bits of address must be 000_2
- 16 bytes: `long double` (GCC on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

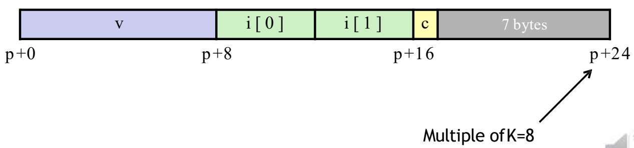


84

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

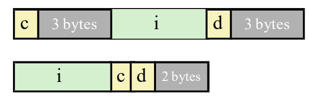
Saving Space

- 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

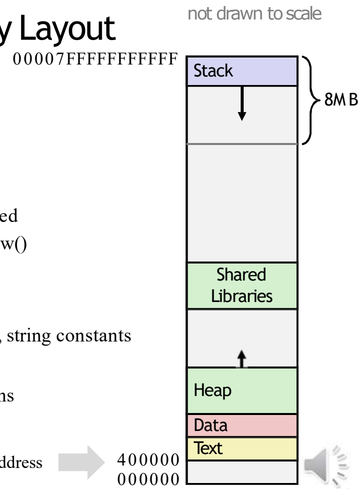
Today

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

88

x86-64 Linux Memory Layout

- 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, static vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



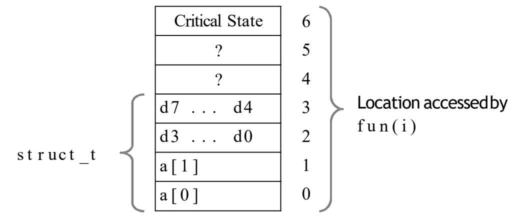
89

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
```

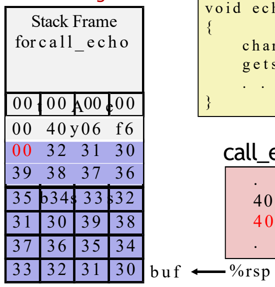
call_echo:

```
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
  . . .
```

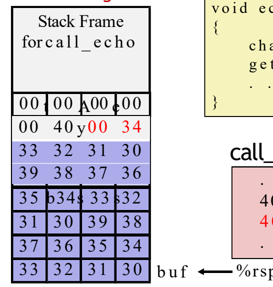
```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

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
  . . .
```

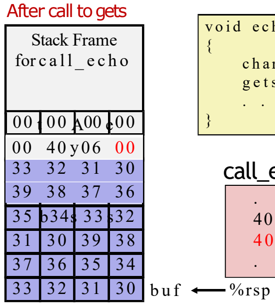
```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

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

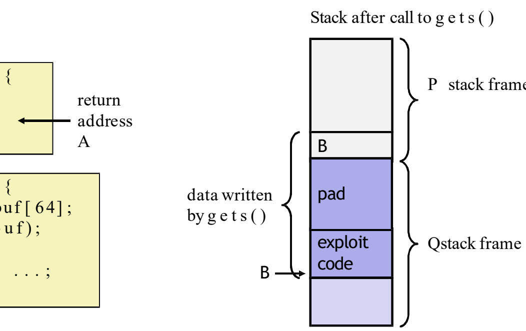
Lots of things happen, without modifying critical state Eventually executes `retq` back to `main`

95

Code Injection Attacks

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

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



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

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

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 27

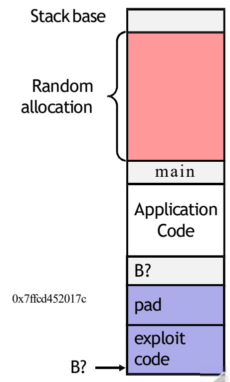
97

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

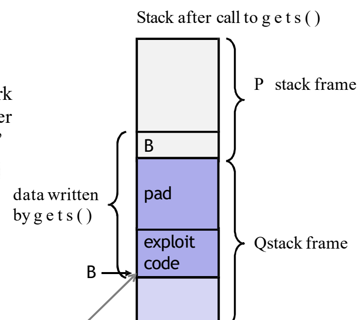


Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 28

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

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 29

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 ***
```

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 30

100

Protected Buffer Disassembly

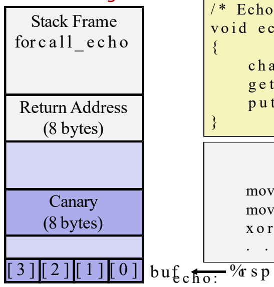
echo:

```
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




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

```
echo:
    . . .
    movq    %fs:40, %rax    # Get canary
    movq    %rax, 8(%rsp)   # Place on stack
    xorl    %eax, %eax      # Erase canary
    . . .
```

```
echo:
    . . .
    movq    8(%rsp), %rax   # Retrieve from stack
    xorq    %fs:40, %rax    # Compare to canary
    je      .L6             # If same, OK
    call    __stack_chk_fail  # FAIL
    . . .
```


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
```

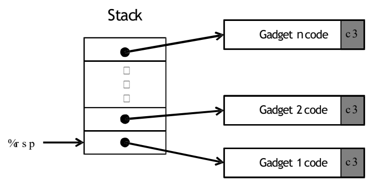
rdi ← rax

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

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

37

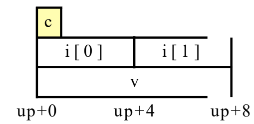
105

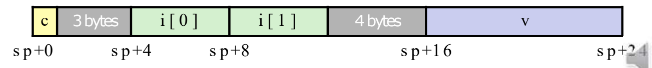
Union Allocation

- 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;
```





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

39

106

Floating Point

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

1

107

Today: Floating Point

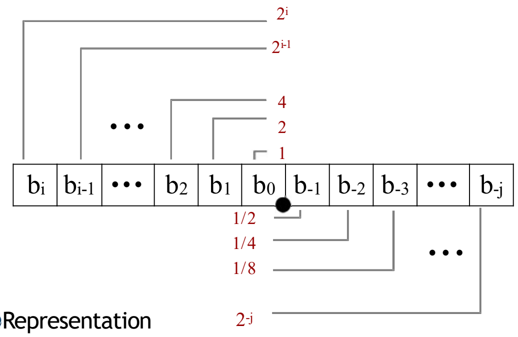
- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

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

2

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

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)

s	exp	frac

111

Precision options

- Single precision: 32 bits

s	exp	frac
1	8-bits	23-bits

- Double precision: 64 bits

s	exp	frac
1	1 -bits	52-bits

- Extended precision: 80 bits (Intel only)

s	exp	frac
1	15-bits	63 or 64-bits

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"

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 11

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 = $1.\underline{1101101101101}_2$
  frac = $\underline{1101101101101}0000000000_2$
- Exponent
  E = 13
  Bias = 127
  Exp = 140 = 10001100_2
- Result:

s	exp	frac
0	10001100	11011011011010000000000

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 12

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 13

115

Carnegie Mellon

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 14

116

Dynamic Range (Positive Only)

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

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 18

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 21

118

Floating Point Operations: Basic Idea

- x $+_f$ y = Round(x + y)
- x □$_f$ y = Round(x □ 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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 23

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 26

120

FP Multiplication

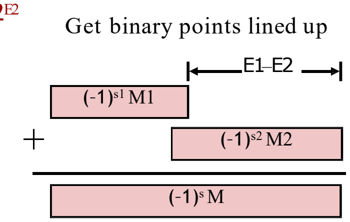
- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: s1 ^ s2
  - Significand M: M1 x M2
  - Exponent E: E1 + E2
- Fixing
  - If M ≥ 2, shift M right, increment E
  - If Eout of range, overflow
  - Round M to fit frac precision
- Implementation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

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 27

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 Eby k
  - Overflow if Eout of range
  - Round M to fit frac precision



Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 28

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 30

123

The Memory Hierarchy

Notes adapted from Bryant and O'Hallaron, Computer Systems: AProgrammer's Perspective, Third Edition 1

124

Today

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

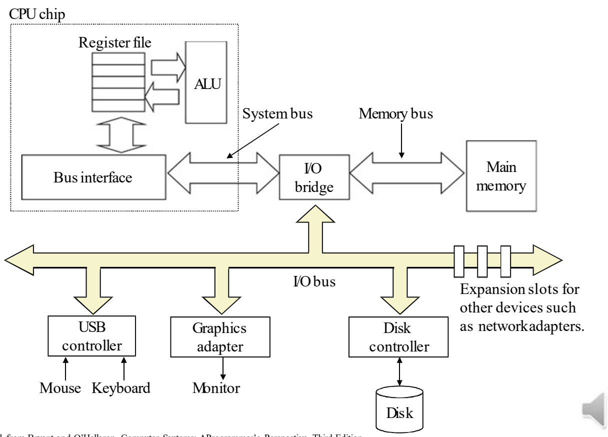
125

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



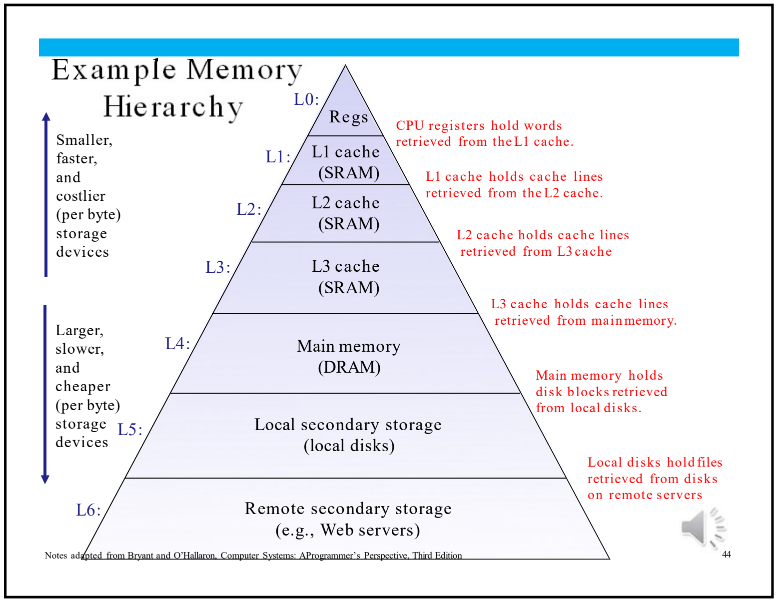
127

Locality Example

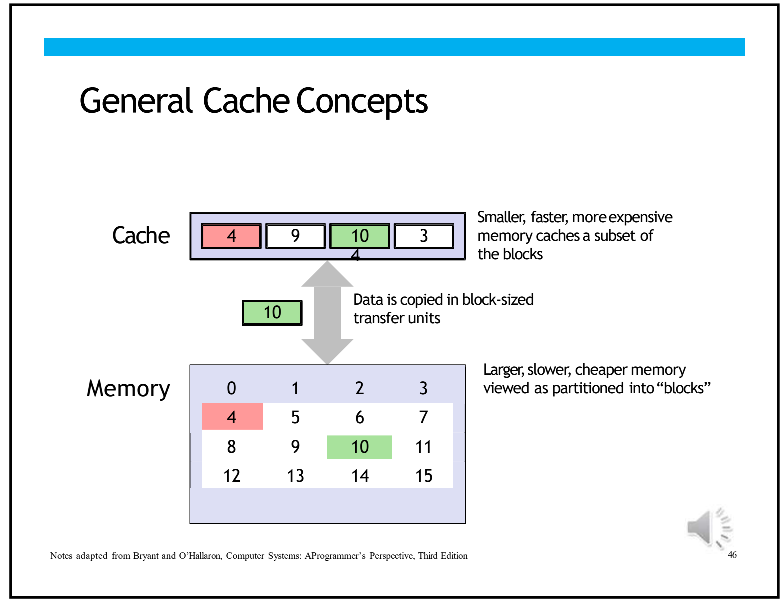
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- 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

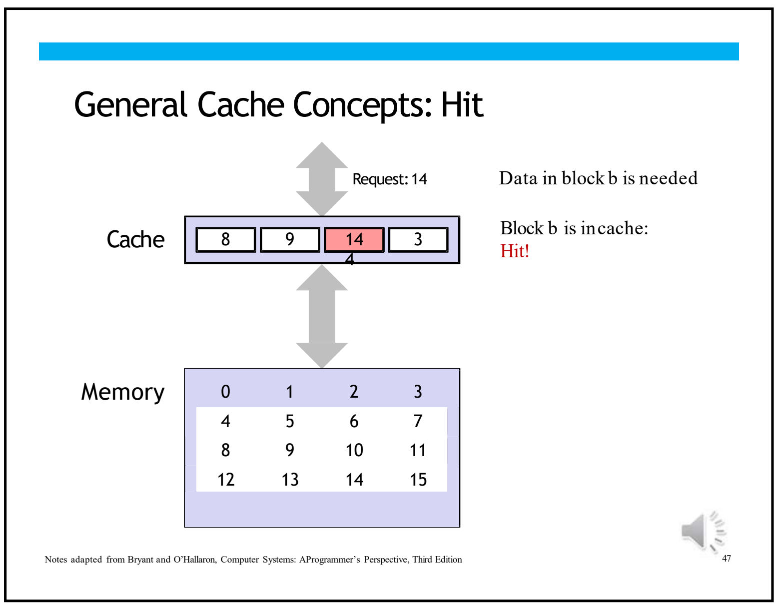
128



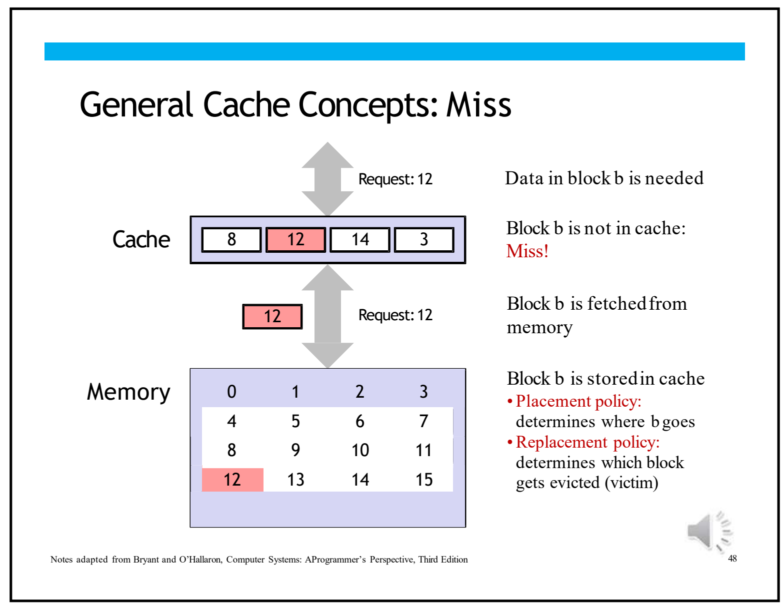
129



130

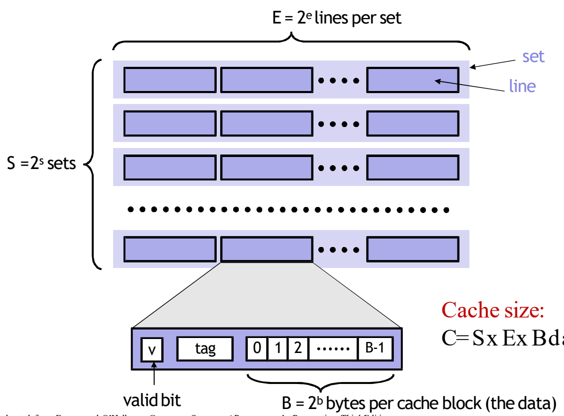


131



132

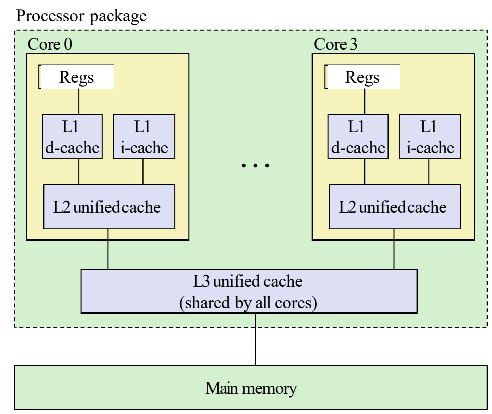
General Cache Organization (S, E, B)



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.

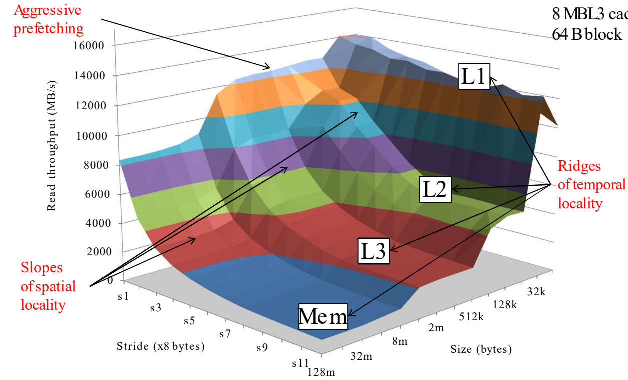
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

56

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



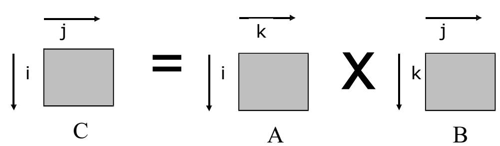
Read throughput (read bandwidth)
Number of bytes read from memory per second (MB/s)

58

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



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

60

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

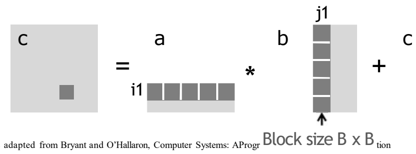
Notes adapted from Br... y ..., Third Edition

68

140

Blocked Matrix Multiplication

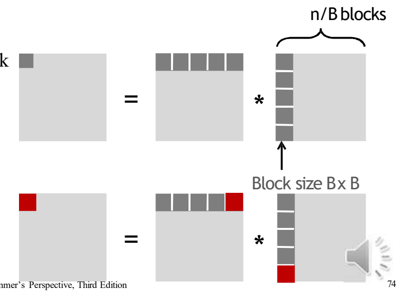
```
c =(double *) calloc(sizeof(double), n*n);

/* 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
```



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)



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

143

Program Optimization

144

Today

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals

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

145

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler
- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially 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: A Programmer's Perspective, Third Edition

146

Compiler-Generated Code Motion (-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: A Programmer's Perspective, Third Edition

147

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: A Programmer's Perspective, Third Edition

148

Share Common Subexpressions

- Reuseportions of expres ions
- GC 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;
```

```
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;
```

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

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
```

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

9

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
- Sinceresult does not change from one iteration to another
- Form of codemotion

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

14

150

Optimization Blocker: Procedure Cals

- Why couldn't compiler moves strlen out of inner loop?
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value forgiven arguments
    - Depends on other parts of globalstate
    - Procedure lowercould interactwith strlen
- Warning:
  - Compiler treats procedure call asa black box
  - Weak optimizations near them
- Remedies:
  - Use of inline functions
    - GC does this with –O1
      - – Within singlefile
  - Doyour own codemotion

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

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

16

151

Removing Aliasing

```
/* Sum rows is of n Xn 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 localvariables
  - Accumulating within loops
  - Your way of telling compiler not to check foraliasing

- No need to store intermediateresults

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

19

152

Exploiting Instruction-Level Parallelism

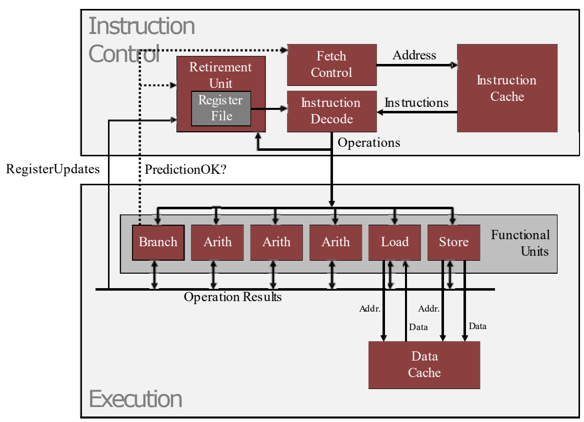
- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity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

Basic Optimizations

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

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

Modern CPU Design

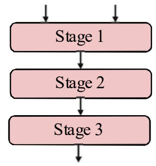


Superscalar Processor

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

Pipelined Functional Units

```
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

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

30

157

Haswell CPU

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

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

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

31

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?

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

36

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

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

39

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

Assume vector length = 100

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029          i = 98
```

Predict Taken (OK)

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029          i = 99
```

Predict Taken (Oops)

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029          i = 100
```

Read invalid location

Executed

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029          i = 101
```

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

Domain Decomposition

First, decide how data elements should be divided among processors

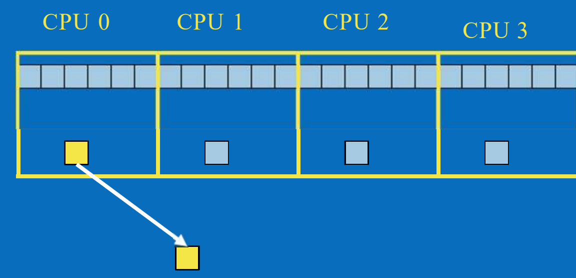
Second, decide which tasks each processor should be doing

Example: Vector addition

165

Domain Decomposition

Find the largest element of an array



166

Task (Functional) Decomposition

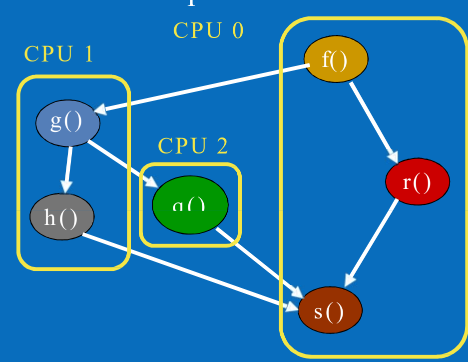
First, divide tasks among processors

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

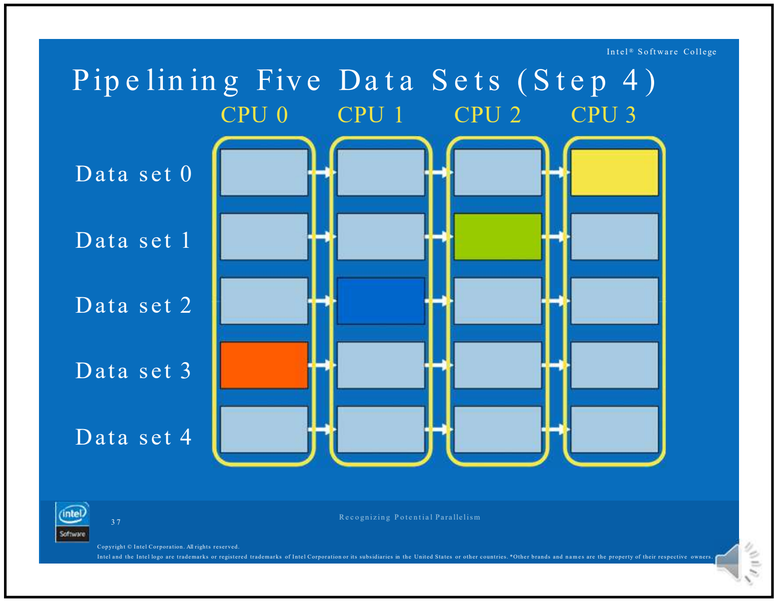
Example: Event-handler for GUI

167

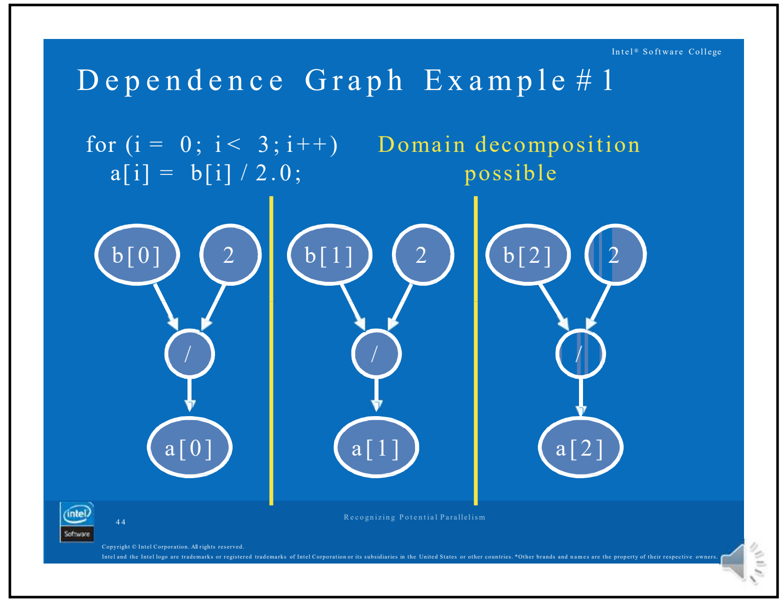
Task Decomposition



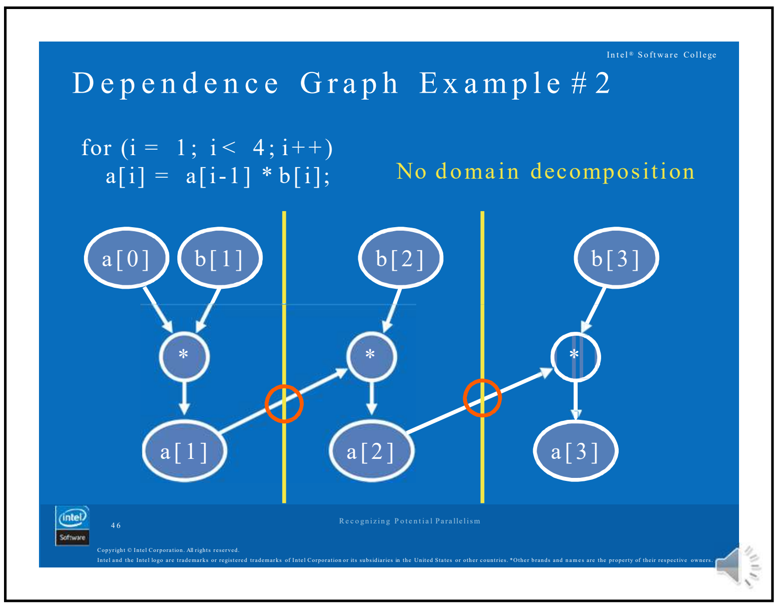
168



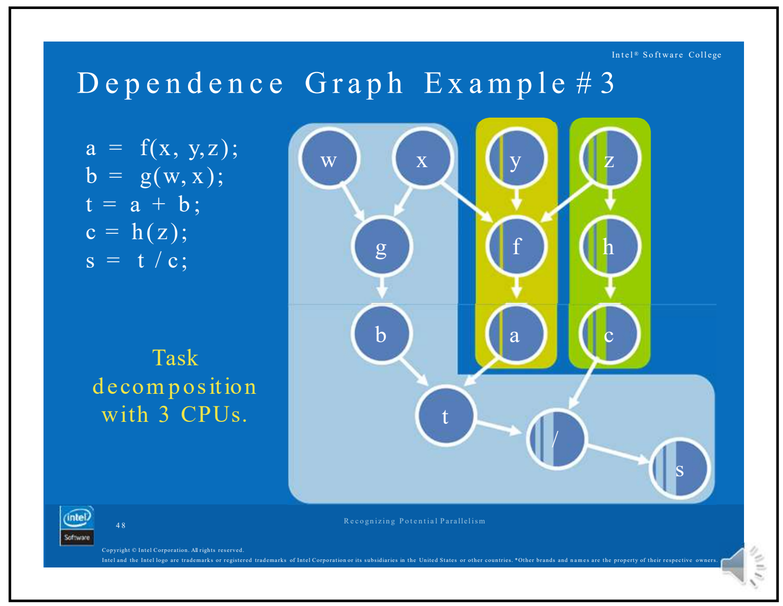
169



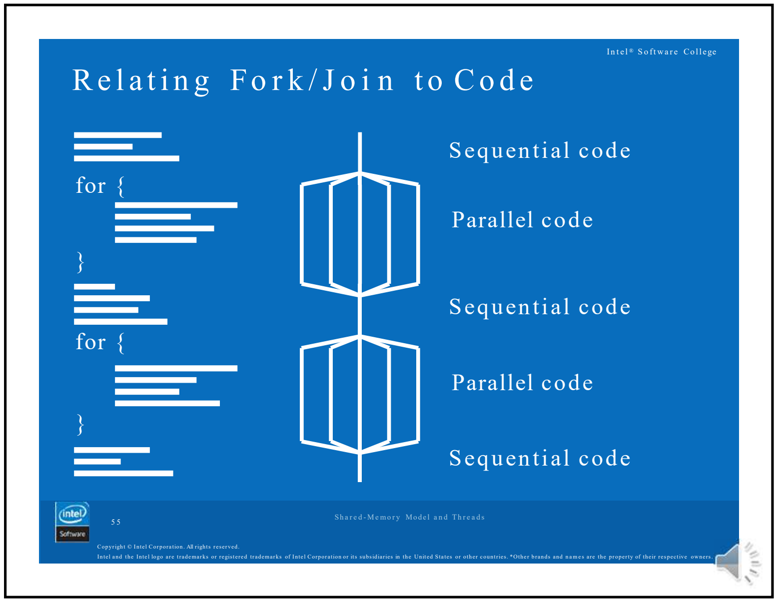
170



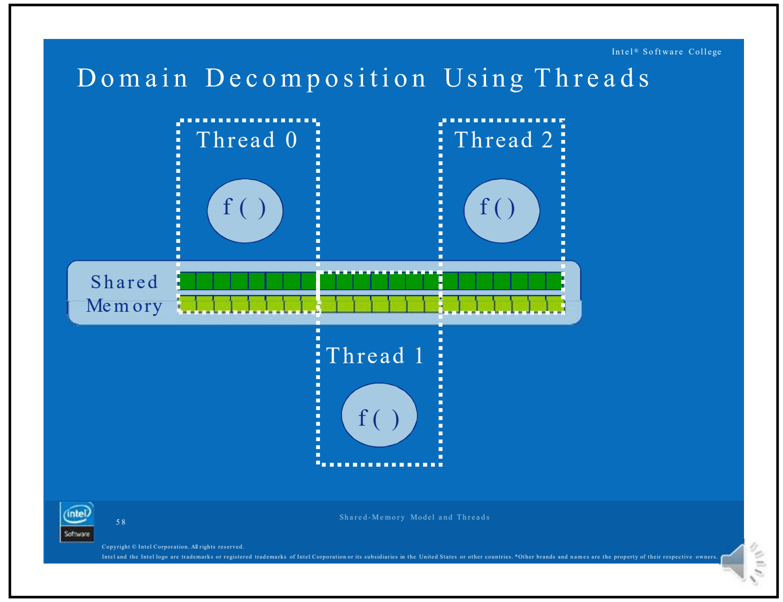
171



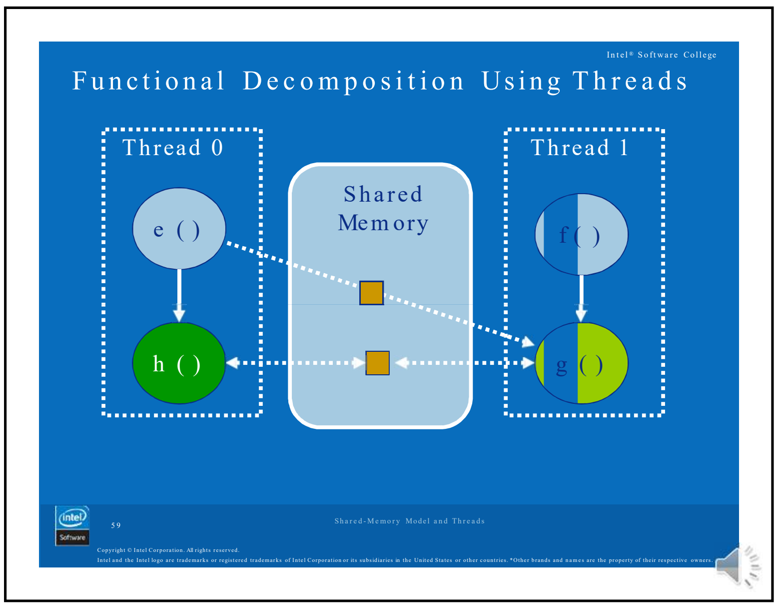
172



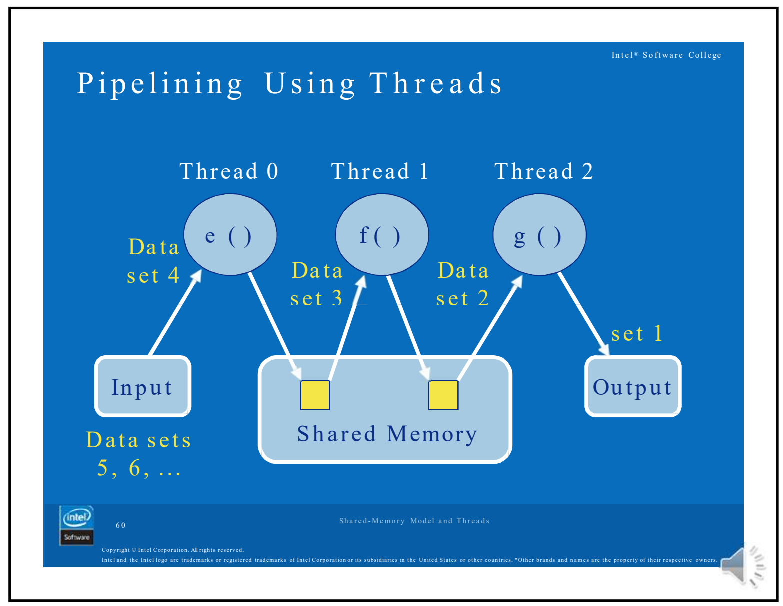
173



174



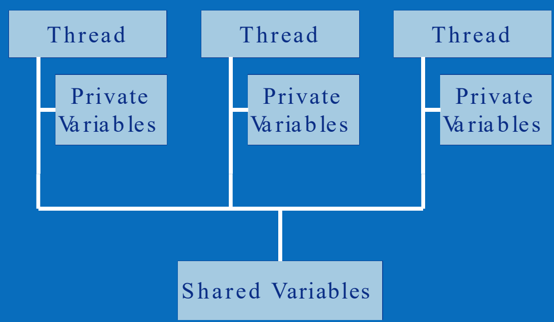
175



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

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

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

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

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

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

Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

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

Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

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

Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

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 of code

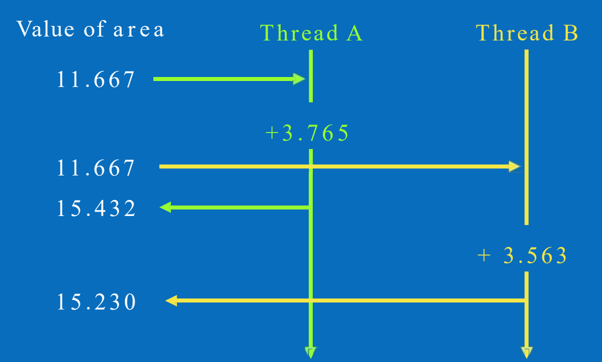
189

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

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

Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

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

Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

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;
```


Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

196

Important: Lock Data, Not Code

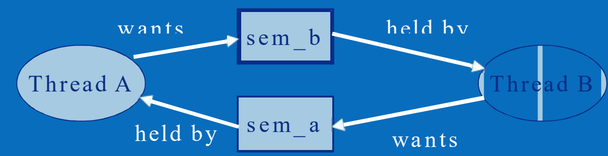
- Locks should be associated with data objects
- Different data objects should have different locks
- Suppose lock associated with critical section of code instead of data object
  - Mutual exclusion can be lost if same object manipulated by two different functions
  - Performance can be lost if two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

Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

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

Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

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.

Copyright © Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.

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

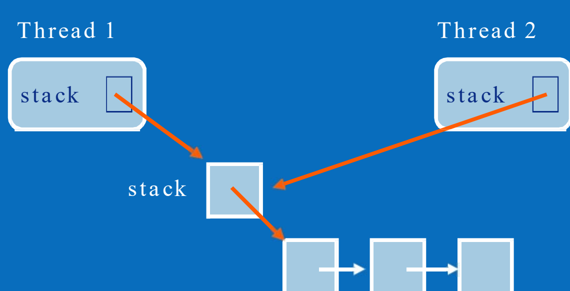
201

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

#pragma omp parallel sections

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

{

<code block A>

Each block executed by one thread

#pragma omp section

<code block B>

Dividers between sections

#pragma omp section

<code block C>

}

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 1

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 3

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 5

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

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
```


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
```


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

Notes adapted from various sources 11

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
```

Notes adapted from various sources 12

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`

Notes adapted from various sources 13

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

Notes adapted from various sources 14

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"
```

217

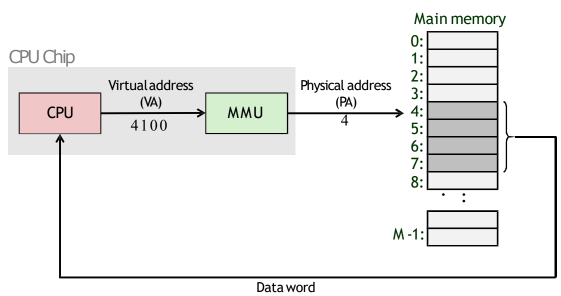
Virtual Memory

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

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 1

218

A System Using Virtual Addressing



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

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 3

219

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
  $\{0, 1, 2, 3, \ldots, M-1\}$

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 4

220

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

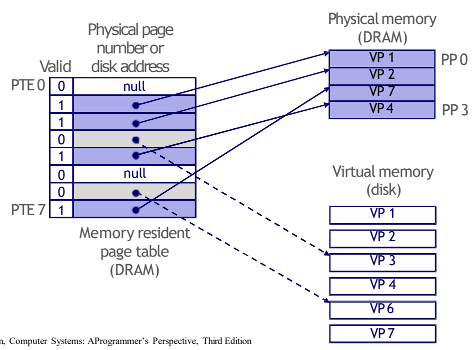
DRAM Cache Organization

- 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

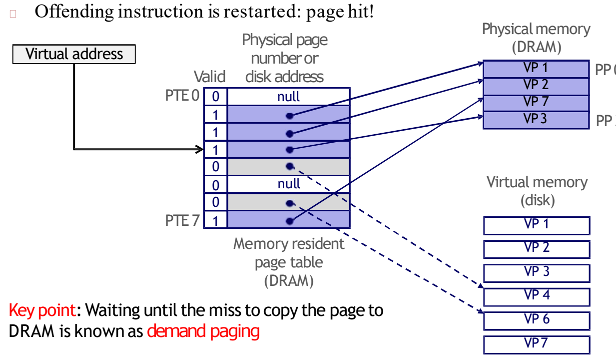
- A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM



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!

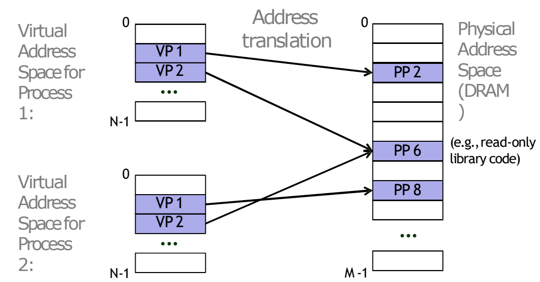
- Virtual memory seems terribly inefficient, but it works because of locality.
- At any point in time, programs tend to access a set of active virtual pages called the working set
  - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
  - 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

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 17

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

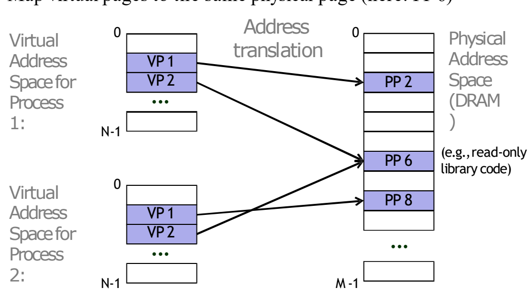


Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 19

226

VM as a Tool for Memory Management

- Simplifying memory allocation
  - Each virtual page can be mapped to any physical page
  - A virtual 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)

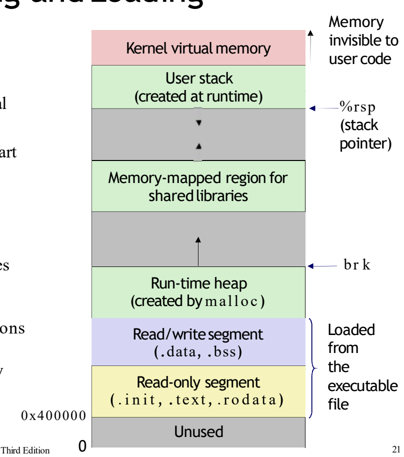


Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 20

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

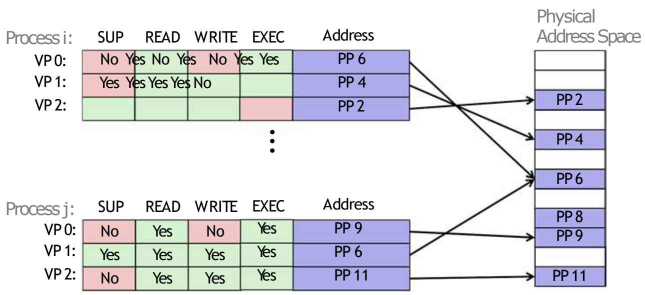


Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 21

228

VM as a Tool for Memory Protection

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



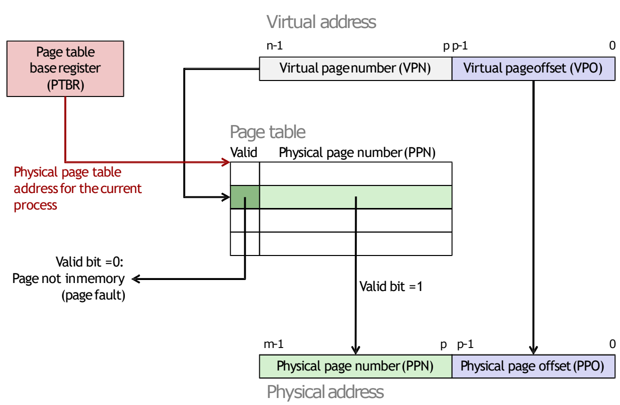
229

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

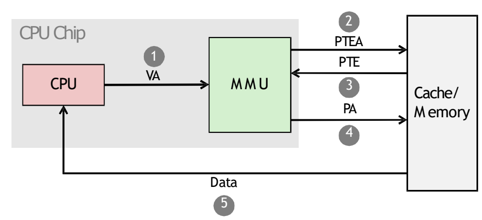
230

Address Translation With a Page Table



231

Address Translation: Page Hit



1) Processor sends virtual address to MMU

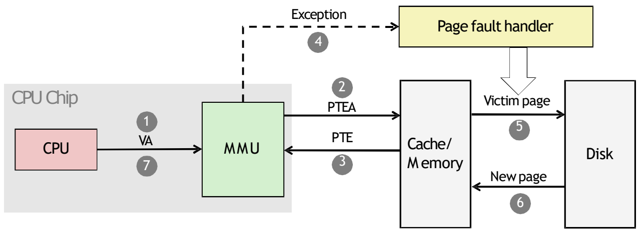
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

232

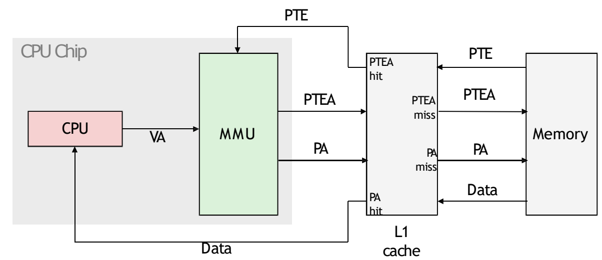
Address Translation: Page Fault



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

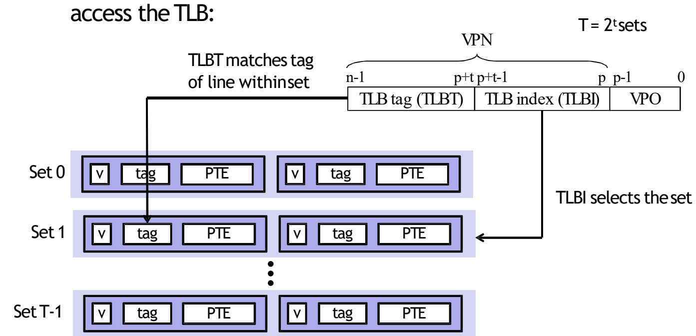
Speeding up Translation with a TLB

- 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

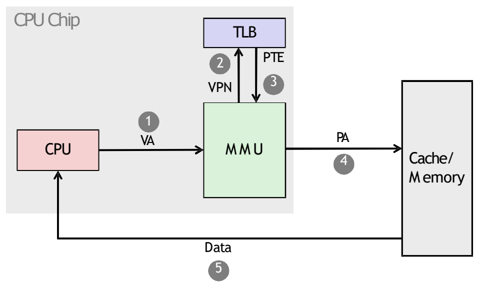
Accessing the TLB

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



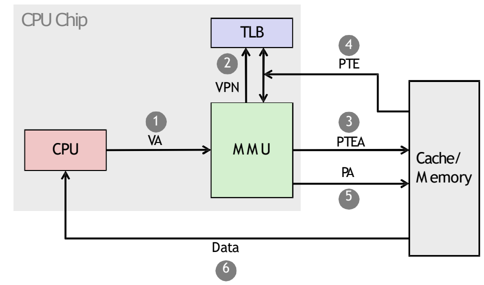
236

TLB Hit



A TLB hit eliminates a memory access

TLB Miss

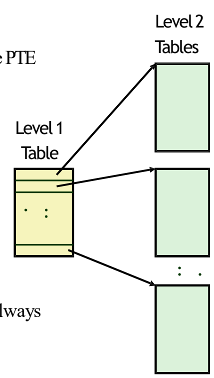


A TLB miss incurs an additional memory access (the PTE)

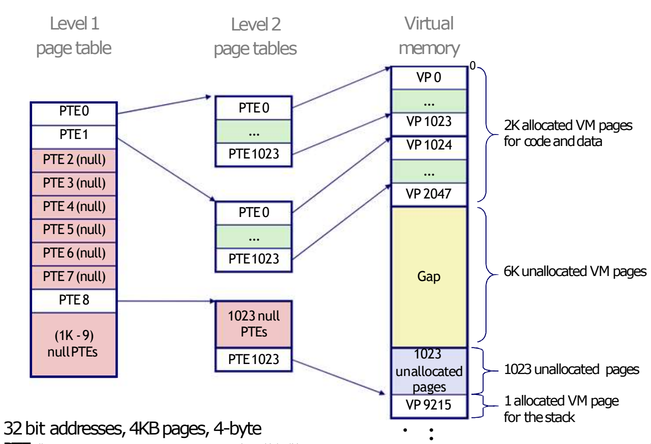
Fortunately, TLBmisses are rare. Why?

Multi-Level Page Tables

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



A Two-Level Page Table Hierarchy



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

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

241

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"

242

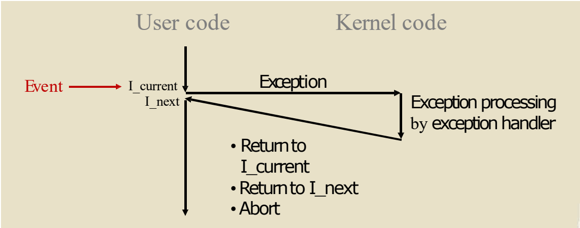
Exceptional Control Flow

- 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 C runtime library

243

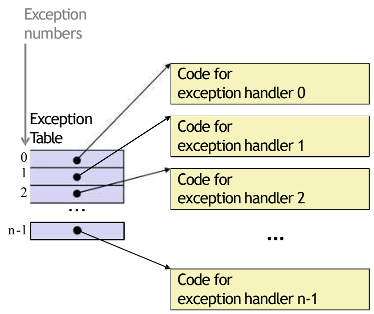
Exceptions

- 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



244

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:

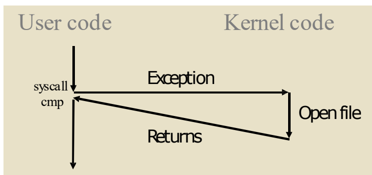
Number	Name	Description
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

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

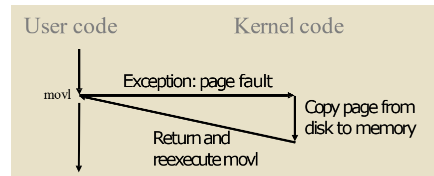
249

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
```

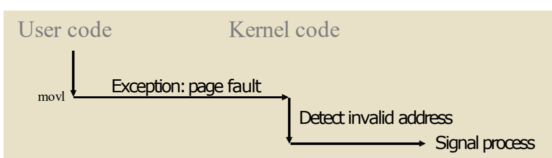


250

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

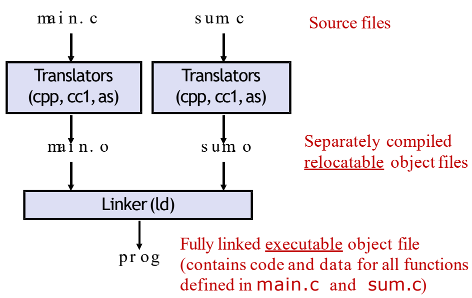
Linking

- 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 C library

- 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

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 8

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.

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 12

258

Step 1: Symbol Resolution

Referencing a global…
…that's defined here

```
int sum(int *a, int n);

int array[2] = {1,2};

int main()
{
  int val = sum(array, 2);
  return val;
}
```

main.c

```
int sum(int *a, int n)
{
  int i, s = 0;

  for (i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
```

sum.c

Defining a global

Linker knows nothing of `val`

Referencing a global…

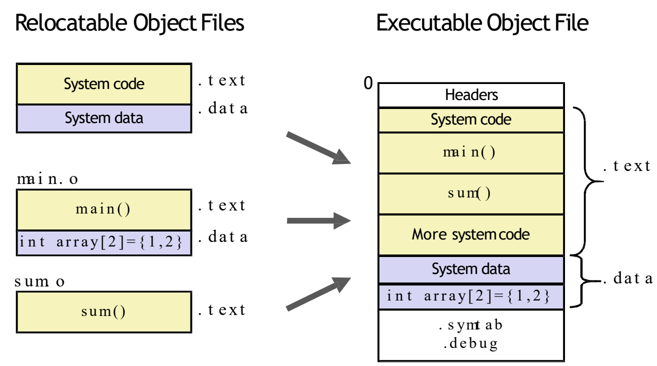
…that's defined here

Linker knows nothing of `i` or `s`

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 13

259

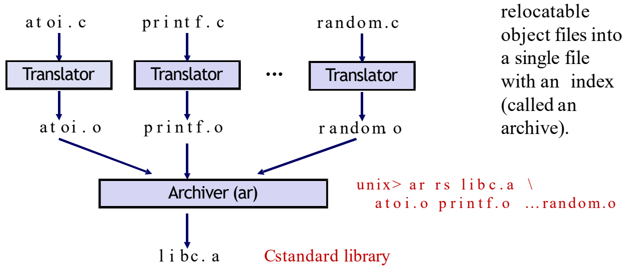
Step 2: Relocation



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 19

260

Creating Static Libraries

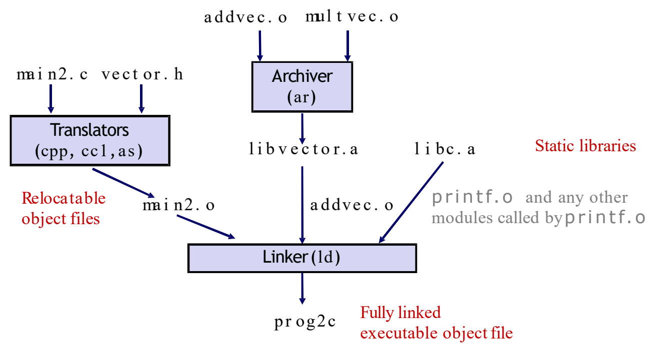


- Concatenate related relocatable object files into a single file with an index (called an archive).

- 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

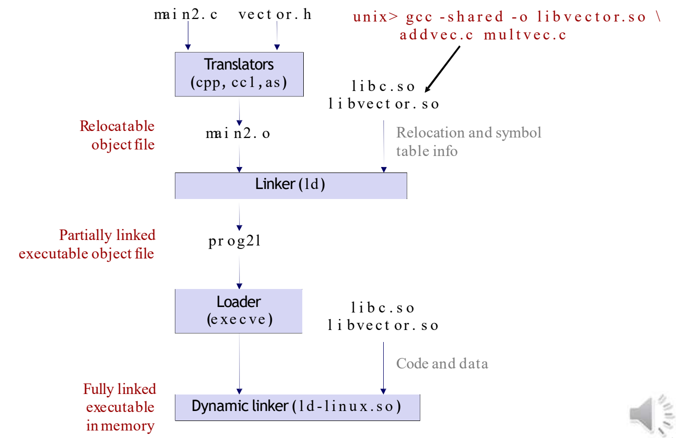
264

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 C library (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 object files.
- Linking can happen at different times in a program's lifetime:
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)
- Understanding linking can help you avoid nasty errors and make you a better programmer.

267