Introduction to Computer Systems

Lecture 1: Introduction

Example 1: is $x^2 > 0$ ->
- floats: ✓
- ints: ?

Example 2: is $(x+y)+z = x+(y+z)$?
- unsigned & signed ints: ✓
- floats: ?

Assembly: x86 assembly is the choice of the course

Memory Reference Error: C & C++ do not provide any memory protection

Optimize program:
- algorithm
- data representation
- procedure
- loop
- ...

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

4.3ms

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

81.8ms

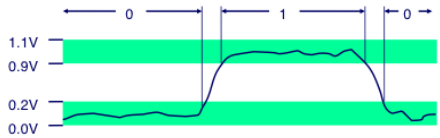
2.0 GHz Intel Core i7 Haswell

Lecture 2: Bits, Bytes and Integers — Part 1

everything is bits!

each bits is 0 or 1

bits -> electronic representation:



bits for base 2 representation

Byte = 8 bits

15213 : 0011 1011 0110 1101

3 B 6 D

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

example data representation ⇒

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	4	8	8

Bit level manipulation : And (&), Or (|), Not (~), Xor (^)

```
  0 1 1 0 1 0 0 1
& 0 1 0 1 0 1 0 1   ...
-----------------
  0 1 0 0 0 0 0 1
```

Bit level operation available in C, apply to any "integral" data type

Logic Operation in C : &&, ||, !

- 0 → False, nonzero → True
- Early termination : exp1 && exp2 && ...
  - ↳ If False, then return False

Useful pattern : p is a pointer

p && p* → avoids null pointer access

Shift Operation

- Left shift x<<y : Fill with 0's on the right
- Right shift x>>y
  - Logical shift
  - Arithmetic shift

Undefined Behavior : Shift amount < 0 or ≥ word size

Integers

Encoding Integers

Unsigned: $B2U(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i$

Two's Complement: $B2T(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$

x_{w-1}: signed bit

	-16	8	4	2	1	
10 =	0	1	0	1	0	8 + 2 = 10
-10 =	1	0	1	1	0	-16 + 4 + 2 = -10

Ranges

Unsigned Values:

$UMin = 0 \quad (0 \cdots 0)$

$UMax = 2^w - 1 \quad (11 \cdots 1)$

Two's Complement Values:

$TMin = -2^{w-1} \quad (10 \cdots 0)$

$TMax = 2^{w-1} - 1 \quad (011 \cdots 1)$

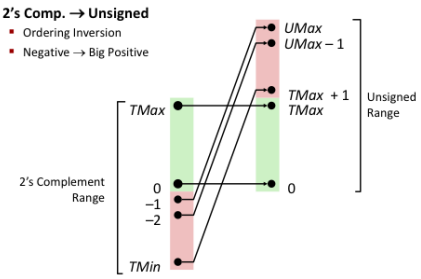
$-1 \quad (11 \cdots 1)$

$|TMin| = TMax + 1$

$UMax = 2 \cdot TMax + 1$

Mapping between Signed & Unsigned: Keep bit representations and reinterpret

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4	=	4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15



Casting: If there is a mix of unsigned and signed in single expression, signed value implicitly cast to unsigned

Sign Extension

- w-bit signed integer x
- Convert it to w+k bit integer with same value
  - → Make k copies of sign bit

Sign Truncation

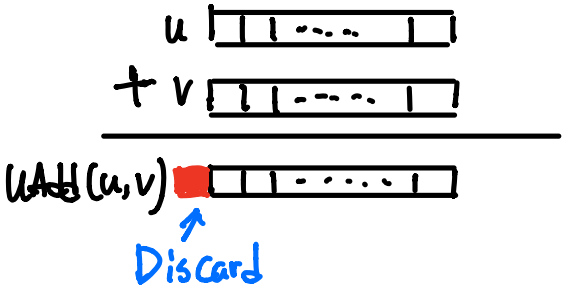
w+k bit integer to w bit

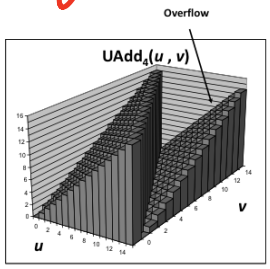
→ drop top k bit (there might be sign change)

Lecture 3: Bits, Bytes and Integers — Part 2

Addition:

Unsigned Addition : Discard carry bit





Two's Complement Addition : Discard carry bit

TAdd and UAdd have identical bit level behavior

Two's Complement Addition Overflow:

$$\begin{cases} \text{if } sum \geq 2^{w-1} & \text{becomes negative, occur at most once} \\ \text{if } sum < -2^{w-1} & \text{becomes positive, occur at most once} \end{cases}$$

$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \text{ (Negative Overflow)} \\ u+v & TMin_w \leq u+v \leq TMax_w \\ u+v-2^w & TMax_w < u+v \text{ (Positive Overflow)} \end{cases}$$

Multiplication

Unsigned Multiplication in C:

ignore high order w bits

$UMult_w(u, v) = (u \cdot v) \bmod 2^w$

Signed Multiplication in C:

ignore high order w bits

Some of which are different for signed vs unsigned multiplication

Lower bits are the same

$x << k \Rightarrow u \cdot 2^k$

$x >> k \Rightarrow \lfloor x/2^k \rfloor$ (unsigned use logic shift, signed use arithmetic shift)

If want $\lceil x/2^k \rceil$ (Round Toward 0) for negative number

$\Rightarrow$ Compute as $\lfloor (x+2^k-1)/2^k \rfloor$

$\Rightarrow$ In C: `(x + (1<<k) -1) >> k`

$\sim x + x == 11\cdots1 == -1$

therefore, $-x == \sim x + 1$

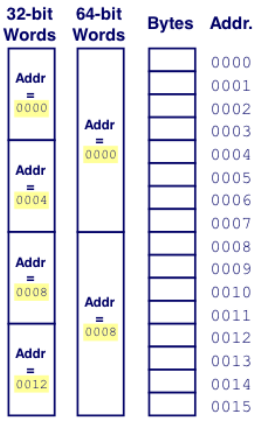
Machine Words

"Word Size": Normal size of integer-valued data, and of addresses

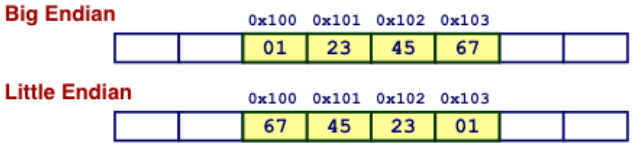
Addresses Specify Byte Location

- Address of first byte in word

Byte Ordering { Big Endian : Internet, Sun
Little Endian : x86



Example : 4-byte value 0x01234567
Address given by &x is 0x100



Lecture 4 : Floating Point

Representation : $b_i\, b_{i-1} \cdots b_0 \cdot b_{-1}\, b_{-2} \cdots b_{-j}$

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

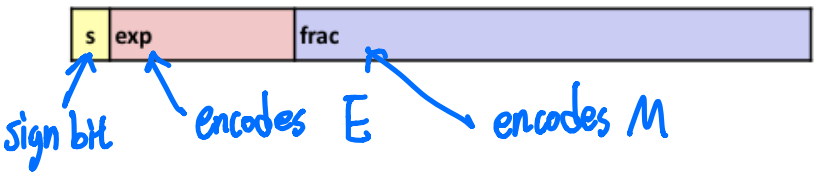
$[0.1111\cdots]_2 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \rightarrow 1.0$ $\quad 1.0 - \varepsilon$

IEEE Floating Point : IEEE Standard 754

Floating Point Representation :

$(-1)^s M\, 2^E$ ← Exponent $\quad M \in [1.0, 2.0)$

Sign bit ← Significand



single precision : 32 bits (exp: 8 bits, frac : 23 bits)
Double precision : 64 bits (exp: 11 bits, frac : 52 bits)

3 kinds of floating point number:

exp { 00...00 denormalized
    { ≠0 and ≠11...11 normalized
    { 11...11 special

Normalized Value:

$E = Exp - Bias$ $\quad Bias = 2^{k-1} - 1$ (k: number of exp bits)

single: 127
Double: 1023

$M = [1.xxxxx\cdots]_2$

bits of frac field

Denormalized Value:

$E = 1 - Bias$ (instead of $E = 0 - Bias$)

$M = [0.xxxxx\cdots]_2$

Special Values:

exp = 11...11 { frac = 00...0 → ∞ (infinity)
              { frac ≠ 00...0 → Not a Number (NaN)

Exercise: 0xC0A00000

1100 0000 1010 0000 0000 0000 0000 0000

$(-1)^1 \cdot 2^{129-127} \cdot (1.01_2)$

$= -1 \times 4 \times 1.25 = -5$

Floating Point Number Zero Same as Integer Zero: All bits = 0

Rounding Binary Numbers

"Even" when least significant bit is 0

"Half way": ...0 1 00... (all zero)

frac ↑ least significant bit

1. BBGRXXX

Guard bit: Least significant bit

Round bit

Sticky bit

Round = 1 Sticky = 1 → > 0.5

Guard = 1 Round = 1 Sticky = 0 → Round to even

Floating Point Number Multiplication

$(-1)^{S_1} M_1 2^{E_1} \times (-1)^{S_2} M_2 2^{E_2} \rightarrow (-1)^S M 2^E$

$S: S_1 \wedge S_2$

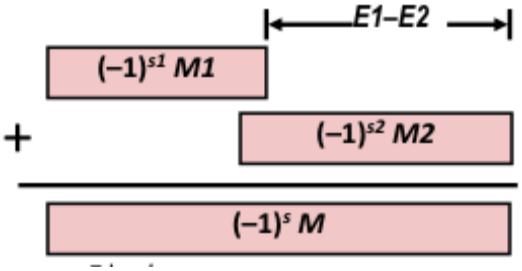
$M: M_1 \times M_2$

$E: E_1 + E_2$

- If $M \geq 2$, shift M right, increment E
- If E out of range, overflow
- Round M to fit `frac` precision

Floating Point Number Addition

$(-1)^{S_1} M_1 2^{E_1} + (-1)^{S_2} M_2 2^{E_2}$ (Assume $E_1 > E_2$)



Lecture 5: Machine Level Programming I: Basics

Intel x86 Processors — introduced in 1978

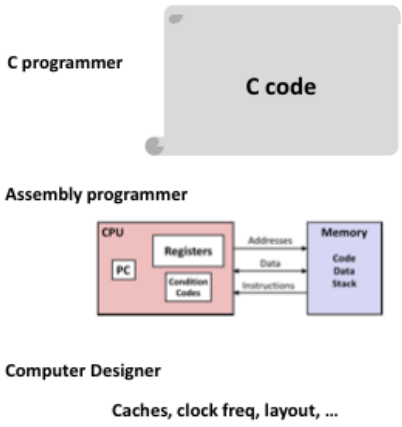
Complex instruction set computer (CISC)

Hard to much performance of Reduced instruction set computer (RISC)

x86 Clones: Advanced Micro Devices (AMD)

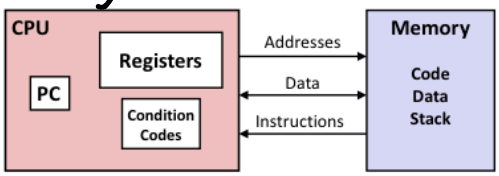
{ IA 32
  x86-64 ✓ Course only cover x86-64

Levels of Abstraction :



Architecture : ISA instruction set architecture
Microarchitecture : Implementation of the architector
Code Forms { Machine Code : Byte-level
Assembly Code : Text representation of machine code

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

Assembly Data Types { Integer data : 1, 2, 4, 8 Bytes
Floating point data : 4, 8, 10 Bytes

x86-64 Integer Registers: 16 totally
IA32 Registers: 8 totally (6 general purpose + %esp + %ebp)

Assembly Operations { Transfer data
Perform arithmetic
Transfer control

how many bytes
q for 8 bytes

Moving Data: movq Source, Dest

Operand Types: { Immediate: $0x400, $-533
Register: %rsp reserved for special use
Memory: given by register: (%rax)

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes:
{ Normal: (R) Mem[Reg[R]] (%rcx)
Displacement: D(R) Mem[Reg[R]+D] 8(%rbp)

Complete Memory Addressing Modes:
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 register
Ri: Index register: any, except for %rsp
S: Scale 1, 2, 4 or 8

example:

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Address Computation Instruction

leaq Src , Dst

address mode expression — Set Dst to address denoted by expression

example : leaq (%rdi, %rdi, 2), %rax

Some Arithmetic Operations

Two Operand Instruction

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	
andq	Src,Dest	Dest = Dest & Src	
orq	Src,Dest	Dest = Dest | Src	

One Operand Instruction

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

Turning C into Object Code

C program → Assembly program (.s) → object program (.o) → Executable program

Compiling into assembly : gcc -Og -S sum.c → sum.s

basic optimizations

Object Code : translate .s into .o

- binary encoding of each instruction
- nearly complete
- missing linkages between code in different files

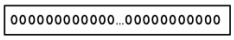
Lecture 6: Machine-Level Programming II: Control

Condition Codes (implicit Setting) (single bit register)

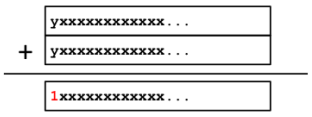
- CF Carry Flag (for unsigned)
- SF Sign Flag (for signed)
- ZF Zero Flag
- OF Overflow Flag (for signed)

leaq do not set leaq instruction!

ZF set when

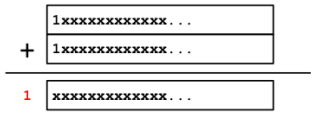


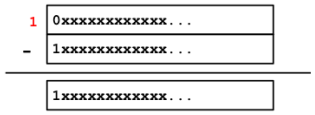
SF set when



For signed arithmetic, this reports when result is a negative number

CF set when

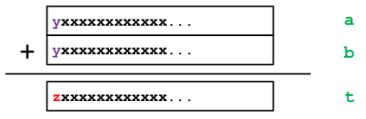




For unsigned arithmetic, this reports overflow

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

OF set when



z = ~y

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

For signed arithmetic, this reports overflow

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

Condition Codes can also be explicit set

Compare instruction: cmpq b, a → like a-b without set dst

Test instruction: testq b, a → like a&b without set dst

Set instruction: setX Dest: set low-order byte of destination Dest to 0 or 1 based on combinations of condition code

Conditional branches

Jumping — jX instructions

jump to different part of code depending on condition codes

implicit reading of 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)

Conditional Moves : val = Test ? Then_Expr : Else_Expr ;

Loops

"Do-While" Translation

C Code :

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

Goto Version:

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

"While" Translation

C Code :

```
while (Test)
    Body
```

Goto Version:

```
    goto test;
loop:
    Body
test:
    if (Test)
        go to loop;
```

"While" Translation

translate to "Do-Version" first

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

"For" Loop

translate to while loop

```
for (init; Test; Update)  →   init
      Body                    while (Test) {
                                    Body
                                    update;
                              }
```

"Switch" Statement : jump table

indirect jump: jmp *.L4 (, %rdi, 8)

Start of jump table : .L4

must scale by factor of 8 (addresses are 8 bytes)

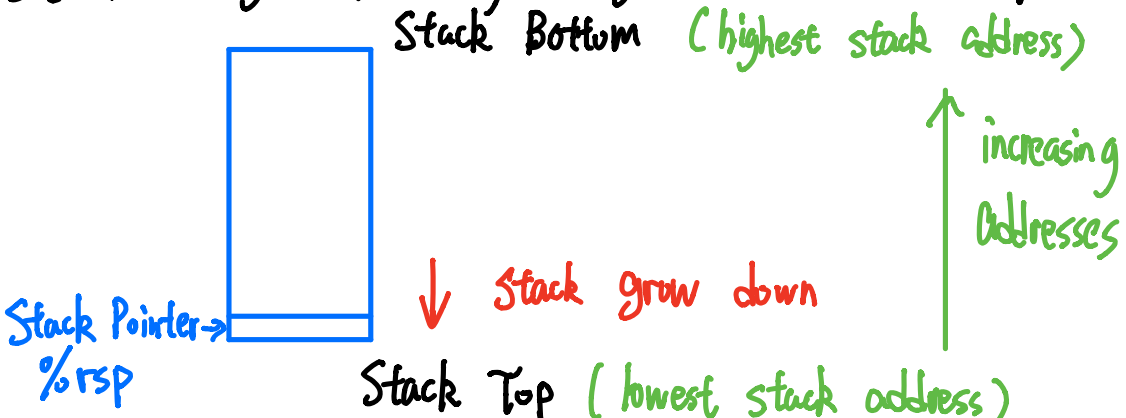
fetch target from effective address .L4 + x*8

Lecture 7: Machine-Level Programming III : Procedures

Mechanisms in Procedures:

- Passing Control
- Passing Data
- Memory Management
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

Stack – region of memory managed with stack discipline



increase the stack by decrease stack pointer

Push : pushq Src decrement %rsp by 8

Pop : popq Dest increment %rsp by 8 (but memory doesn't change)

Procedure Control Flow

passing control

- **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

passing data

first 6 arguments : %rdi, %rsi, %rdx, %rcx, %r8, %r9

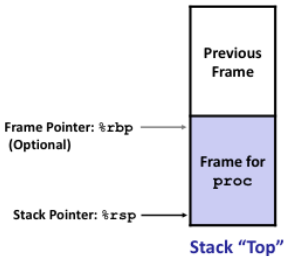
allocate stack space when needed:

return value : %rax

Stack-Based Languages

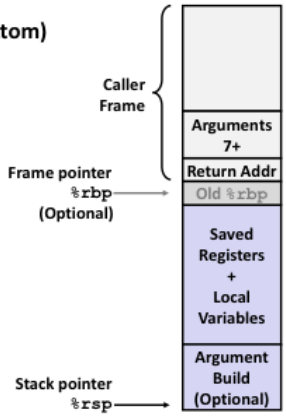
Stack allocated in Frames

Frame Pointer : %rbp



x86-64 / Linux Stack Frame

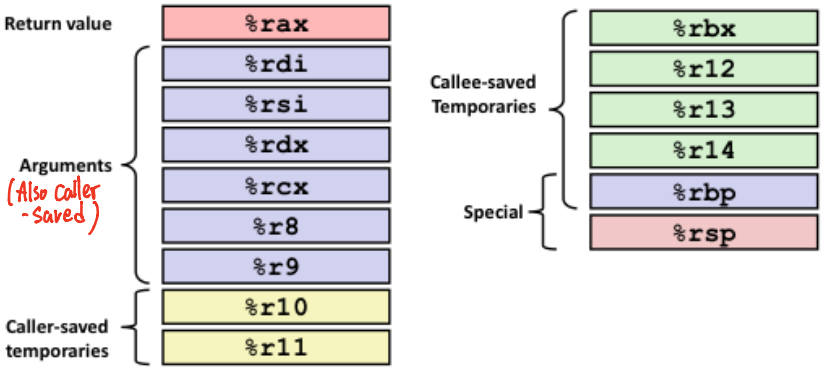
- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:" Parameters for function about to call
  - Local variables If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments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



Register Saving Conventions

Caller saved : Caller saves temporary values in its frame before the call

Callee saved: Callee saves temporary values in its frame before using, callee restores them before return



Recursion Function Handled Without Special Consideration

Lecture 8: Machine-Level Programming IV: Data

Array Allocation

T A[L]

T: type, L: length

Contiguously allocated region of L * sizeof(T) bytes in memory

example: int val[5];

val[4] — int

val — int* — assume value is x

val+1 — int* — x+4

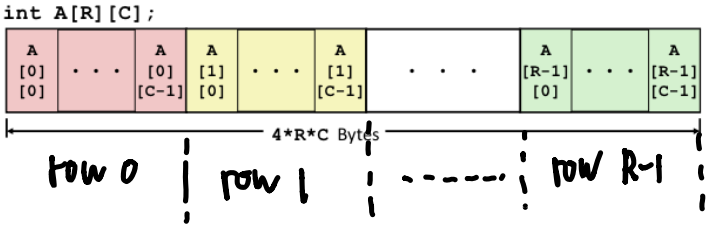
val+i — int* — x+4*i

Multidimensional (Nested) Arrays

T A[R][C]

size = R * C * sizeof(T)

arrangement: row-major ordering

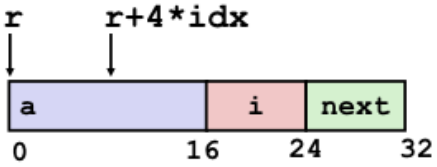


address of $A[i] = A + i * (C * \text{sizeof}(T))$

$A[i][j] = A + i * (C * \text{sizeof}(T)) + j * \text{sizeof}(T)$

Structure Representation

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



Aligned Data:

Primitive data type requires K bytes

Address must be multiple of K

Required on some machines; advised on x86-64

- 1 byte: char ...
- 2 bytes: short ...
- 4 bytes: int, float ...
- 8 bytes: double, long, char*, ...

Overall Alignment Requirement:

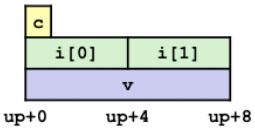
For largest alignment requirement K

Overall structure must be multiple of K

Unions

- Overlay declaration
- Way to circumvent type system
- Allocate according to largest element
- Can only use one field at a time

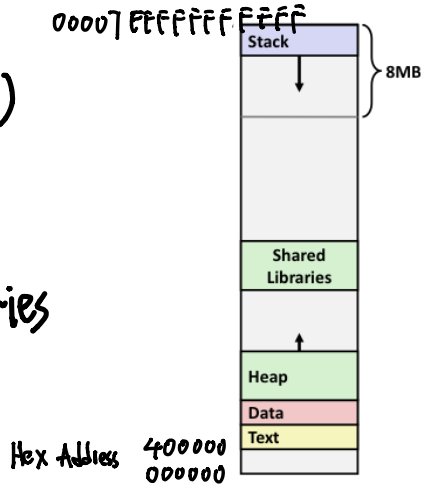
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



Lecture 9: Machine-Level Programming V: Advanced Topics

Memory Layout

- Stack (8MB limit)
- Heap
- Data
- Text / Shared Libraries



Buffer Overflow Stack Attack

① Stack Smashing Attacks:

- Overwrite normal return address A with address of some other code S
- When Q executes ret, will jump to other code

② Code injection Attacks:

- Input string contains byte representation of executable

code.

- Overwrite return address A with address of buffer B
- When Q executes ret, will jump to exploit code

Avoid Buffer Overflow Stack Attack

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use "stack canaries"

③ Return-Oriented Programming Attacks

Floating Points

XMM Registers

16 total, each 16 bytes

Basics:

Arguments passed in %xmm0, %xmm1, ...

Results returned in %xmm0

All XMM registers Caller-saved

Lecture 10: Code Optimization

Generally Useful Optimizations

Code Motion: Reduce frequency with which computation performed (especially moving code out of loop)

Reduction in Strength: Replace costly operation with simpler one Shift, Add instead of multiply or divide

Share Common Subexpressions: Reuse portions of expressions

Optimization Blocks

① Procedure Calls

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

calculated every loop, should be move out of the loop

Warning: Compiler may treat procedure call as a black box
Weak optimizations near them

Remedies: Use of inline functions
Do your own code motion

② Memory Aliasing
Two different memory references specify single location
Get in habit of introducing local variables
- Your way of telling compiler not to check for aliasing

Exploiting Instruction-Level Parallelism
Superscalar Processor:
Definition: A superscalar processor can issue and execute multiple instructions in one cycle
The processor fetches instructions sequentially and executes out of order (OoO) via dynamic scheduling
Benifit: without programming effort, OoO superscalar execution

can take advantage of the instruction level parallelism (ILP) that most program have.

Pipelined Functional Units

	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

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

- 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

Loop Unrolling

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

v.s.

```
x = x OP (d[i] OP d[i+1])   ✓
```

(might lead to different result for floating points, over-flow, loss precision ....)

v.s.

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

Branch Prediction

Default behavior:

- Backwards branches are often "loops" so predict taken
- Forwards branches are often "if" so predict not taken

Lecture 11 : The Memory Hierarchy

Random-Access Memory (RAM)

packaged as a chip

basic storage unit is normally a cell (one bit per cell)

multiple RAM chips form a memory

RAM { SRAM (Static RAM)
DRAM (Dynamic RAM)

	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

*EDC: Error detection and correction

DRAM: SDRAM, DDR SDRAM, DDR, DDR2, DDR3, DDR4

Both DRAM and SRAM are volatile memories

Nonvolatile memories : ROM, PROM, EPROM, EEPROM

Disk

Capacity: maximum number of bits that can be stored

Capacity determined by { Recording density
Track density
Areal density : Recording density × Track density

Capacity = (# bytes / sector) × (avg # sectors / track)
× (# tracks / surface) × (# surfaces / platter)
× (# platters / disk)

Access Time

$T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$

Seek Time: 3-9 ms

Rotational latency: $\frac{1}{2} \times \frac{1}{RPMs} \times \frac{60\ sec}{1\ min}$

Transfer time: $\frac{1}{RPMs} \times \frac{1}{(avg\ \#\ sectors/track)} \times \frac{60\ sec}{1\ min}$

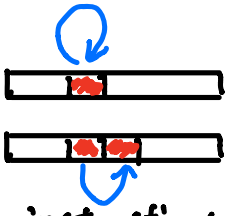
Solid State Disks (SSDs)

advantages: faster, less power, more rugged

disadvantages: potential to wear out, more expensive

Locality of reference

locality { Temporal locality
Spatial locality



programs tend to use data and instructions with addresses near or equal to those they have used recently

Memory Hierarchy

Cache ↑ smaller, faster, more expensive

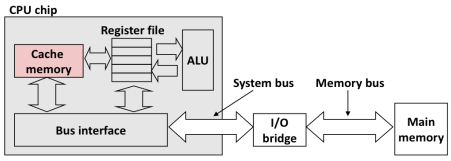
Memory ↓ larger, slower, cheaper

Cache misses { Cold miss
Capacity miss
Conflict miss

Lecture 12: Cache Memories

Cache memories are small, fast SRAM-based memories managed automatically in hardware.

CPU looks first for data in cache



General Cache Organization (S, E, B)

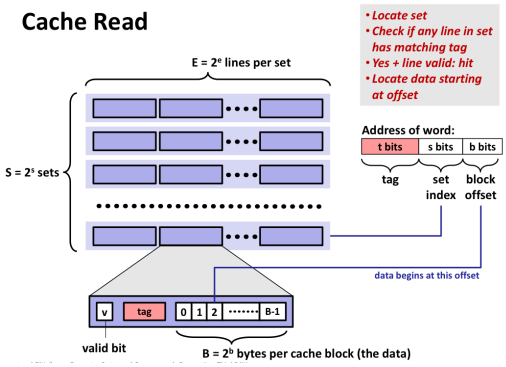
$S = 2^s$ sets

$E = 2^e$ lines (cache block) per set

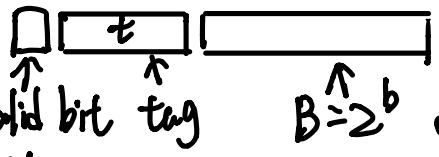
$B = 2^b$ bytes per cache block

Cache Size = S × E × B data bytes

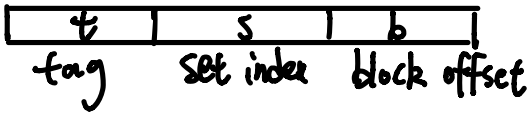
Cache Read:



Cache block:



Address:



Directed Mapped Cache (E = 1)

E-way Set Associative Cache

Cache Writes
- write-hit
  - write-through
  - write-back — need a dirty bit
- write-miss
  - write-allocate
  - no-write-allocate

Typical:
- Write-through + no-write-allocate
- Write-back + write-allocate

Why index set using middle bits?

Middle bit indexing: adjacent memory blocks map to different sets —— makes good use of spatial locality

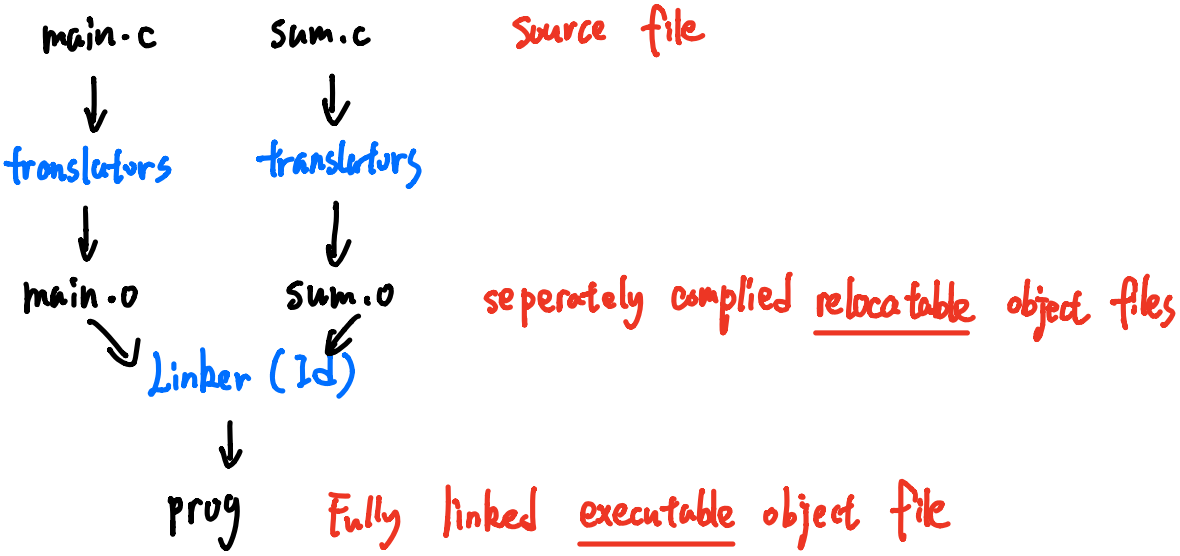
High bit indexing: adjacent blocks map to same set, program with high spatial locality would generate lots of conflicts

Cache performance Metrics

- Miss Rate
- Hit Time
- Miss Penalty

Lecture 13 : Linking

Programs are translated and linked using a compiler driver



{ static linking
dynamic linking

What do linkers do

Step 1: Symbol Resolution

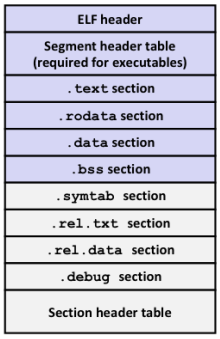
Step 2: Relocation

three kinds of object files { Relocatable object file (.o file)
Executable object file (a.out file)
Shared object file (.so file)

Executable and Linkable Format (ELF)

- Standard binary format for object file
- One unified format for : .o , a.out , .so
- Generic name : ELF binaries

- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **`.text` section**
  - Code
- **`.rodata` section**
  - Read only data: jump tables, string constants, ...
- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space



Linker Symbols { Global symbols
External symbols
Local Symbols (not local program variables, they are not on stack)

```c
int sum(int *a, int n);
int array[2] = {1, 2};
int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```
main.c

external symbol
global symbol
global symbol

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

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

global symbol

"Static" keyword:

- symbol only visible in enclosing scope (file or function)
- Stored in either .bss or .data or .text (Not on Stack)

(static function)

```c
int f() {
    static int x = 17;
    int y=0;

    x++;
    y++;
    printf("%d %d\n", x, y);
}

Void
main() {
    for (int i=0; i<10; i++) {
       f();
    }
}
```

result: 18 1
19 1
20 1
⋮

x is in .data and only initialized once, y is on stack

Duplicate Symbol Definitions

Program symbols are either strong or weak

Strong - procedures and initialized globals

weak - uninitialized globals

Very weak - uninitialized globals declared with "extern"

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc -fno-common`
- **Rule 4: Never pick a "very weak" symbol**

Linking with Static Libraries V.S. Shared Libraries

Shared Libraries:

- Dynamic linking can occur when executable is first loaded and run (load-time linking)
- Dynamic can also occur after program has begun (run-time linking)
- Shared library routines can be shared by multiple processes.

Dynamic libraries required .interp section & .dynamic section

Case study: Library Interpositioning

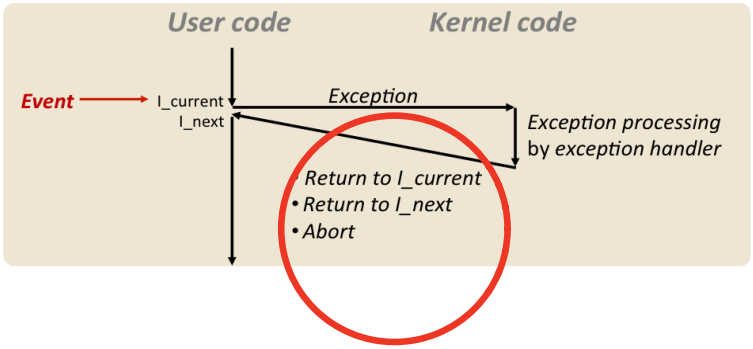
Lecture 14: Exceptional Control Flow: Exceptions and Processes

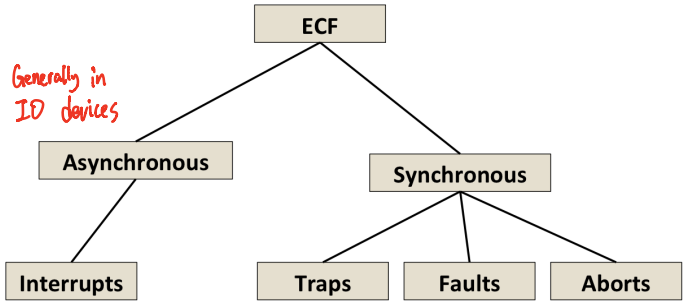
Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  1. Exceptions
- Higher level mechanisms

2. Process context switch
3. Signals
4. Nonlocal jumps

Exceptions: a transfer of control to the OS kernel in response to some event





Asynchronous Exceptions (Interrupts)
- Caused by events external to the processor

Synchronous Exceptions
- Caused by events that occur as a result of executing an instruction

System Calls: read file, write file, open file, ...

Fault: Page Fault, Invalid Memory Reference, ...

Processes

A process is an instance of a running program

two key abstraction { Logical control flow
Private address space

Concurrent Processes

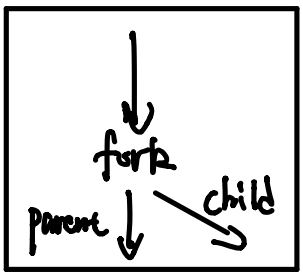
- Each process is a logical control flow
- Two processes run concurrently (are concurrent) if their flows overlap in time
- Otherwise, they are sequential

Processes are managed by a shared chunk of memory-resident OS code called the kernel

Process states { Running
Stopped
Terminated

Creating Processes: int fork(void)

Called once but returns twice



return 0 to child process, return child's PID to parent process

< pid_t getpid(void): get current PID
pid_t getppid(void): get parent PID

Reaping Child Processes: Performed by parent on terminated child (zombies), using wait or waitpid

execve: Loading and running programs

int execve(char *filename, char* argv[], char* envp[])

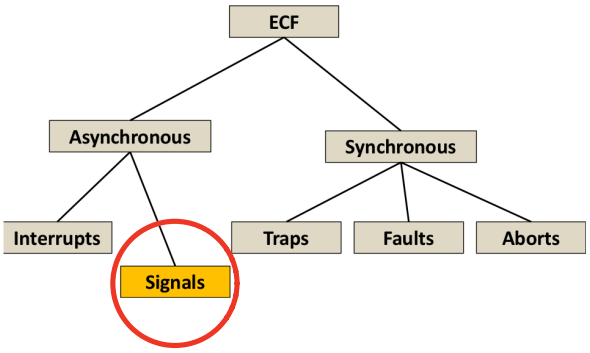
filename — argument list — environment variable list

overwrites code, data, and stack

Called once and never returns

Lecture 15: Exceptional Control Flow: Signals and Nonlocal Jumps

Taxonomy



Shell Programs

A shell is an application program that runs programs on behalf of the user.

example: > /bin/sleep 10 & (Run Program in background)

Signals

A signal is a small message that notifies a process that an event of some type has occurred in the system.

- Sent from the kernel to a process
- Identified by small integer ID's (1–30)

A destination process **receives** a signal when it is forced by the kernel to react in some way to delivery of the signal.

- **ignore** the signal
- **Terminate** the process
- **Catch** the signal by executing a user-level function called **Signal handler**

Pending: A signal is **pending** if sent but not yet received

Block: A process can **block** the receipt of certain signals

Receiving Signals:

Receiving Signals

- **Suppose kernel is returning from an exception handler and is ready to pass control to process *p***
- **Kernel computes `pnb = pending & ~blocked`**
  - The set of pending nonblocked signals for process *p*
- **If (`pnb == 0`)**
  - Pass control to next instruction in the logical flow for *p*
- **Else**
  - Choose least nonzero bit *k* in **`pnb`** and force process *p* to ***receive*** signal *k*
  - The receipt of the signal triggers some ***action*** by *p*
  - Repeat for all nonzero *k* in **`pnb`**
  - Pass control to next instruction in logical flow for *p*

Actions → { Default Actions
Installing Signal Handlers

Blocking and Unblocking Signals

- Implicit blocking mechanism
  kernel blocks any pending signals of type currently being handled.
- Explicit blocking and unblocking mechanism

"sigprocmask" function

Safe Signal Handling Guidelines:

- **G0: Keep your handlers as simple as possible**
  - e.g., Set a global flag and return
- **G1: Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
- **G2: Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
- **G3: Protect accesses to shared data structures by temporarily blocking all signals.**
  - To prevent possible corruption
- **G4: Declare global variables as `volatile`**
  - To prevent compiler from storing them in a register
- **G5: Declare global flags as `volatile sig_atomic_t`**
  - *flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - Flag declared this way does not need to be protected like other globals

Do not use signals to count events! such as children terminating.

Lecture 16: System-Level I/O

Two sets: system-level and C-level

Unix I/O:

A Linux file is a sequence of m bytes

All I/O devices are represented as files

Even the kernel is represented as a file

open(), close(), read(), write(), lseek()

file types: { regular file
directory
socket
others

regular files { text files (only ASCII or Unicode characters)
binary files
directories : an array of links

Opening files: fd = open(PATH, FLAG)
returns a small identifying integer file descriptor
fd == -1 ⇒ an error occured

file descriptor { 0 : Standard input (stdin)
1 : Standard output (stdout)
2 : Standard error (stderr)

Standard I/O
fopen, fclose, fread, fwrite, fgets, fputs, fscanf, fprintf....
Standard I/O functions use buffered I/O
Buffer flushed to output fd on "\n", called to fflush or exit, or return from main.

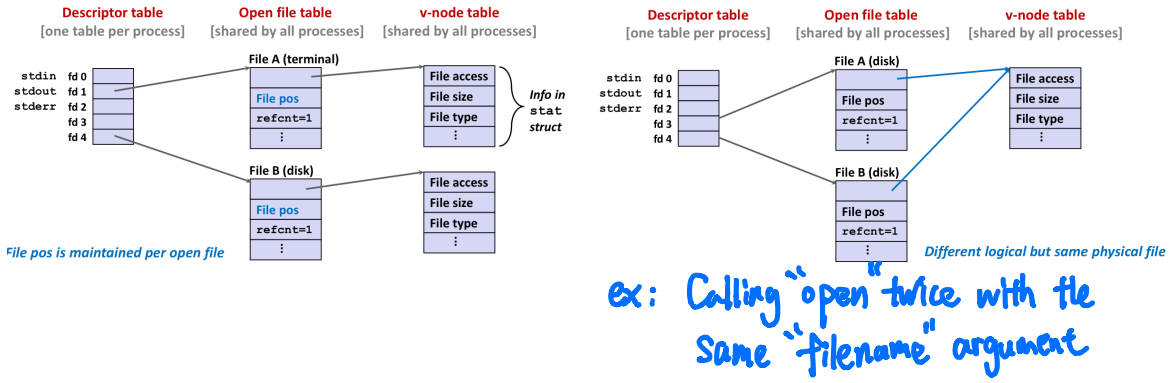
Choosing I/O Functions
General rule: use the highest-level functions you can

- **When to use standard I/O**
  - When working with disk or terminal files
- **When to use raw Unix I/O**
  - *Inside signal handlers, because Unix I/O is async-signal-safe*
  - In rare cases when you need absolute highest performance
- **When to use RIO**
  - *When you are reading and writing network sockets*
  - Avoid using standard I/O on sockets

Functions that should never use on binary files:
- Text-oriented I/O
- String functions

File Metadata: Metadata is data about data.

How the Unix kernel Represents Open Files

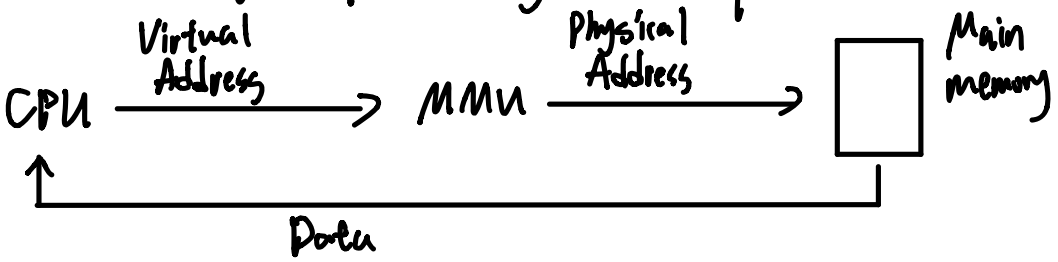


ex: Calling "open" twice with the same "filename" argument

Fork: Child gets a copy of its parent's descriptor table

I/O Redirection: dup2(oldfd, newfd)
Copies descriptor table entry oldfd to entry newfd

Lecture 17: Virtual Memory: Concepts



Address Spaces { Linear address space $0, 1, 2, 3, \ldots$
Virtual address space $N = 2^n$
Physical address space $M = 2^m$

Conceptually, virtual memory is an array of N contiguous bytes stored on disk
The contents of the array on disk are cached in physical memory (DRAM cache)

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

Page Table: - per process
- Locate in DRAM

{ Page Hit: reference to VM word that in physical memory (DRAM cache hit)
Page Fault: reference to VM word that not in physical memory (DRAM cache miss)

Handling Page Fault: Waiting until the miss to copy the page to DRAM (demand paging)

Working Set: At any point of time, programs tend to access a set of active virtual pages
If working set < main memory size: Good Performance
else: Thrashing!

VM as a Tool for Memory Management:
- Each process has its own virtual address space
- Simplifying memory allocation
- Sharing code and data among processes
- Linking
- Loading

VM as a Tool for Memory Protection
- Extend PTEs with permission bits (SUP, READ, WRITE, EXEC)
- MMU checks these bits on each access

Address Translation

Virtual Address Space $V = \{0, 1, \cdots N-1\}$

Physical Address Space $P = \{0, 1, \cdots M-1\}$

Translation: $V \rightarrow P \cup \{\phi\}$

$N = 2^n$ virtual address space

$M = 2^m$ physical address space

$P = 2^p$ Page size

VPO: Virtual page offset

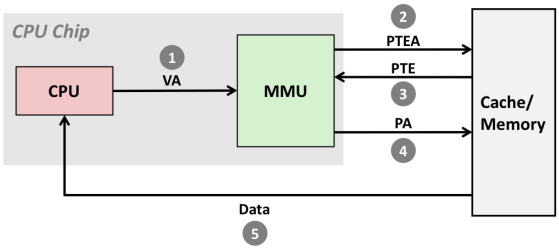
VPN: Virtual page number

PPO: Physical page offset

PPN: Physical page number

Page table base register (PTBR): Physical page table address for the current process

Page Hit:



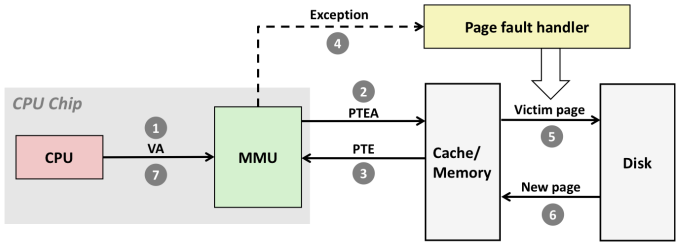
1) Processor sends virtual address to MMU

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

Page Fault:



1) Processor sends virtual address to MMU

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

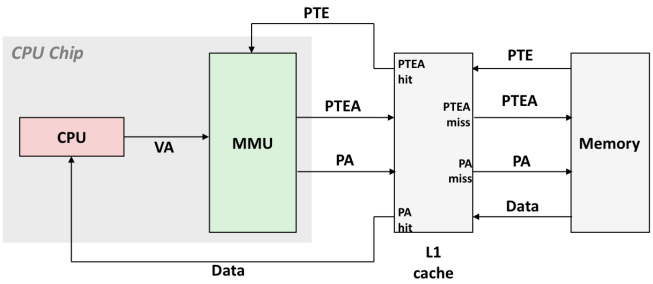
4) Valid bit is zero, so MMU triggers page fault exception

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

With Cache:



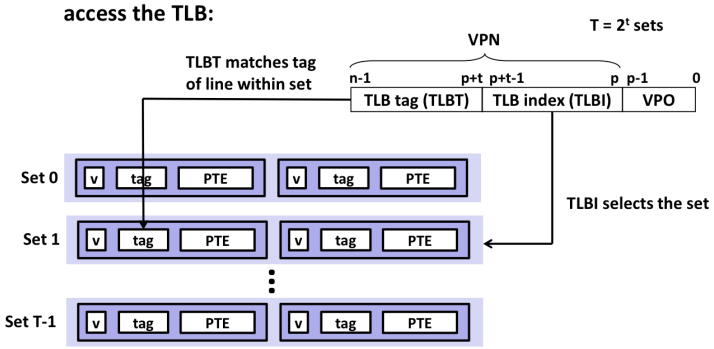
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

TLB (Translation Lookaside Buffer) - Speed up translation
- 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

Virtual address
+ TLBI: TLB index
+ TLBT: TLB tag

Accessing the TLB

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



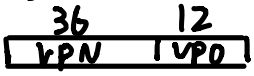
TLB misses are rare (locality & small working set)

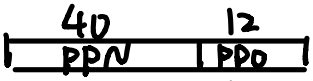
Multi-level Page Tables
Example: 2-level page table
level 1: each PTE points to a page table
level 2: each PTE points to a page

Lecture 18 : Virtual Memory : Systems

A TLB hit eliminates the k memory accesses required to do a page table lookup.

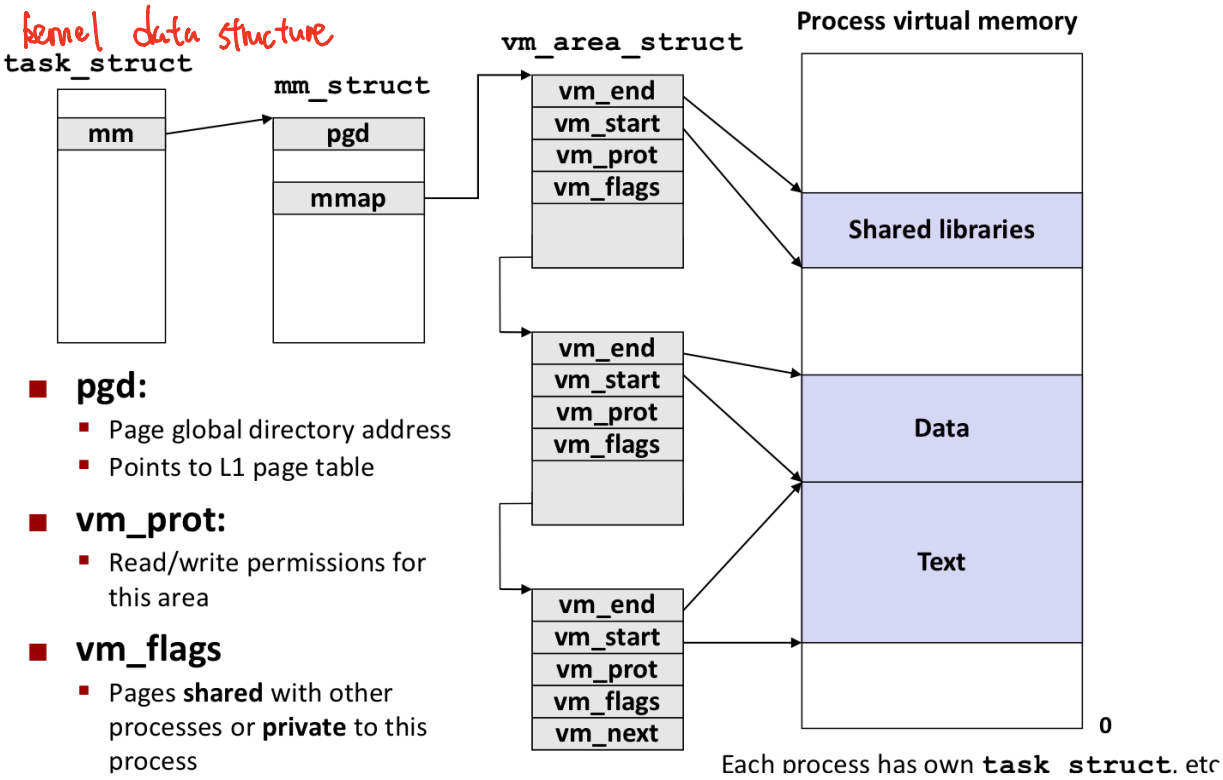
Core i7 / Linux memory system

- Each entry references a 4k child page table
- Virtual address : 

- Physical address : 

- Bits that determine Cache Index are identical in virtual and physical address, can index into cache while address translation taking place, "Virtually indexed, physical tagged"

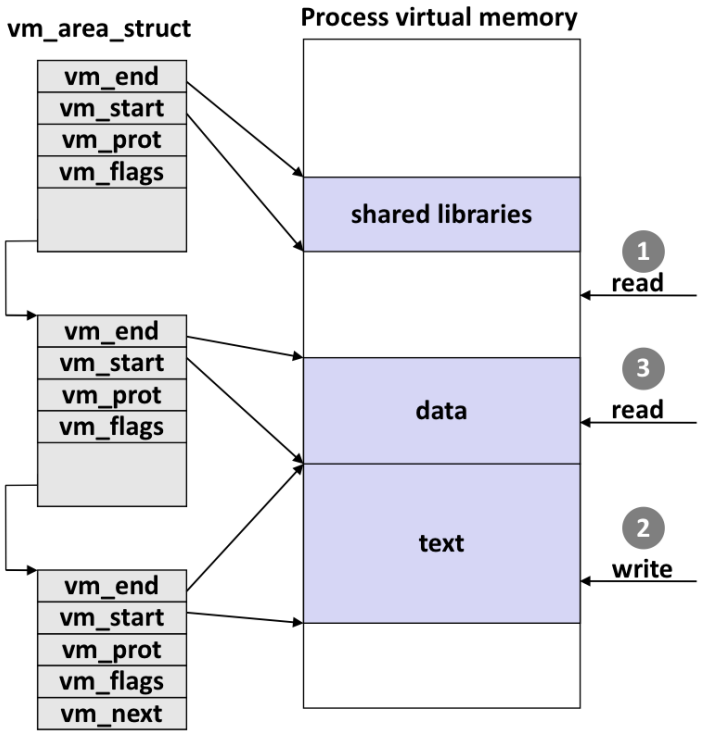
Linux Organizes VM as Collection of Areas



- **pgd:**
  - Page global directory address
  - Points to L1 page table
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Pages **shared** with other processes or **private** to this process

Each process has own `task struct`, etc

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

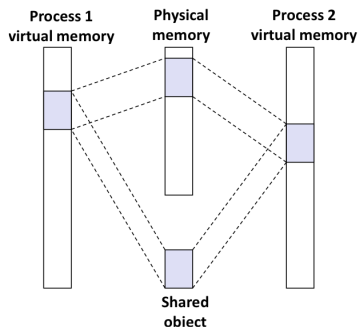
Memory Mapping

memory mapping: VM areas initialized by associating them with disk objects

Areas can be backed by { Regular file / Anonymous file

Shared Objects

example: libc



- Process 2 maps the same shared object.
- Notice how the virtual addresses can be different.
- But, difference must be multiple of page size

private copy-on-write (COW) object
- Instruction writing to private page triggers protection fault
- Handler creates new R/W page
- Instruction restarts upon handler return
- Copying deferred as long as possible

fork Function gets revisited

execve Function gets revisited

User-Level Memory Mapping: void *mmap(...)

Lecture 19: Dynamic Memory Allocation - Basic Concepts

Basic Concepts:

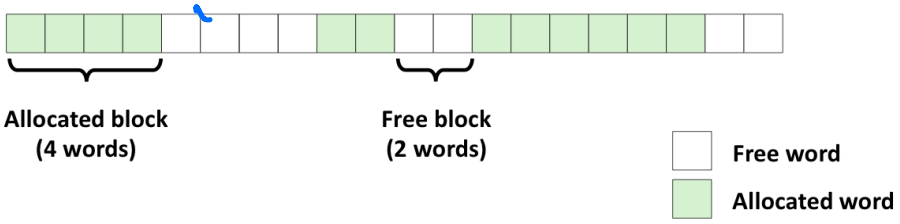
- Programmers use dynamic memory allocators (malloc) to acquire virtual memory (VM) at run time
- Dynamic memory allocators manage an area of process VM known as the heap
- Allocator maintains heap as collection of variable sized blocks, which are either allocated or free
- { Explicit allocator: malloc and free in C
  { Implicit allocator: new and garbage collection in Java

The malloc Package

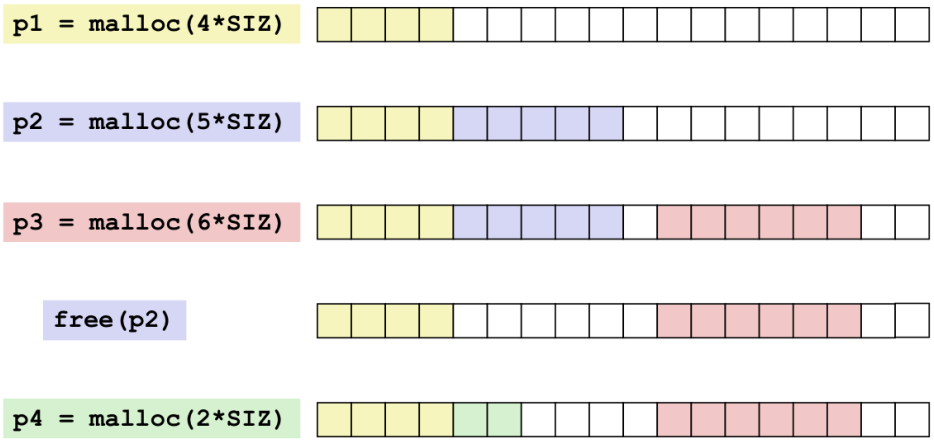
malloc, free, calloc, realloc, sbrk

Some simple assumptions in this lecture:

- Memory is word addressed
- Words are int-sized
- Allocations are double-word aligned




```
#define SIZ sizeof(int)
```



Def: Aggregate payload P_k

- malloc(p) results in block with a payload of p bytes
- After requests R_k has completed, the aggregate payload P_k is the sum of currently allocated payloads.

Def: Current heap size H_k

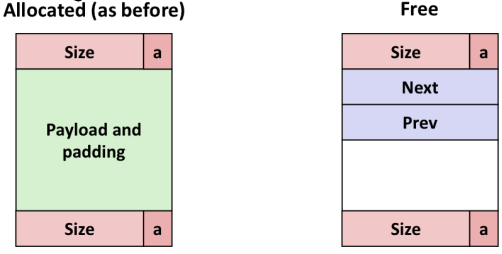
- Assume H_k is monotonically nondecreasing
  - i.e. heap only grows when allocator uses sbrk

Def: Peak memory utilization after k+1 requests
- $U_k = (\max_{i \le k} P_i) / H_k$

Framentation: Poor memory utilization caused by fragmentation
- internal: Payload is smaller than block size
- external: No single free block is large enough

Keep Track of Free Blocks
Method 1: Implicit Free List
For each block we need both size and allocation status
Could store this two information in one word
Finding a free block:
- First fit
- Next fit
- Best fit

Lecture 20: Dynamic Memory Allocation: Advanced Concepts
Keep Track of Free Blocks
Method 2: Explicit free list
Only maintain list(s) of free blocks, not all blocks



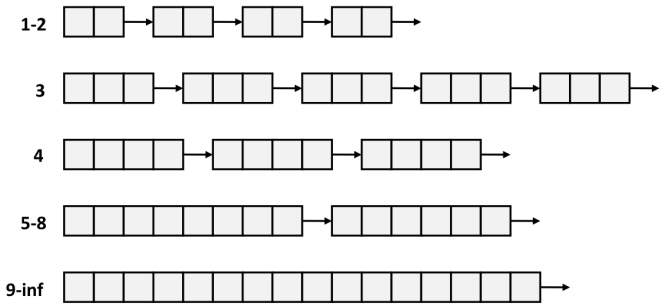
Freeing with explicit free lists

Insertion policy: Where in the free list do you put a newly freed block?

- Unordered:
  - LIFO (last-in-first-out)
  - FIFO (first-in-first-out)
- Address-ordered policy

Method 3: Segregated List (Seglist) Allocator

- Each size class of blocks has its own free list
- Often have separate classes for each small size
- For larger sizes: One class for each size $[2^{i}+1, 2^{i+1}]$



Advantages:

1. Higher throughput
2. Better memory utilization

Seglist Allocator

- **Given an array of free lists, each one for some size class**
- **To allocate a block of size *n*:**
  - Search appropriate free list for block of size $m > n$ (i.e., first fit)
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found
- **If no block is found:**
  - Request additional heap memory from OS (using `sbrk()`)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.

Garbage Collection

Garbage collection: automatic reclamation of heap-allocated storage —— application never has to explicitly free memory

Classical GC Algorithms:

1. Mark-and-sweep collection
2. Reference counting
3. Copying collection
4. Generational Collectors

View memory as a directed graph

- Each block is a node in the graph
- Each pointer is an edge in the graph
- Locations not in the heap that contain pointers into the heap are called **root** nodes

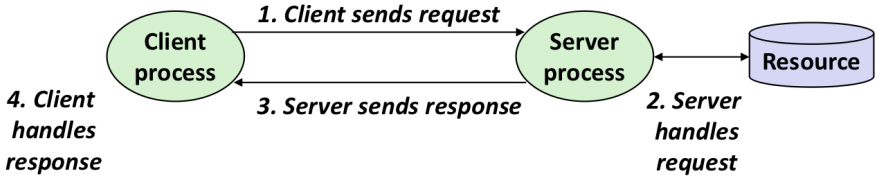
Mark and Sweep Collecting

- Can build on top of malloc / free package
  - Allocate using malloc until you "run out of space"
- When out of space
  - Use extra **mark bit** in the head of each block
  - **Mark:** Start at roots and set mark bit on each reachable block
  - **Sweep:** Scan all blocks and free blocks that are not marked

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Lecture 21 : Network Programming : Part 1



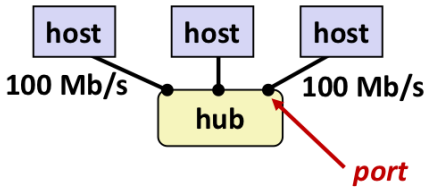
A network is a hierarchical system of boxes and wires organized by geographical proximity

- SAN: System Area Network
- LAN: Local Area Network
- WAN: Wide Area Network

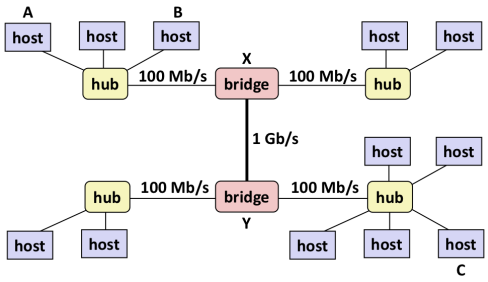
An internetwork (internet) is an interconnected set of networks

- The Global IP Internet (uppercase "i") is the most famous example of an internet (lowercase "i")

Lowest Level: Ethernet Segment

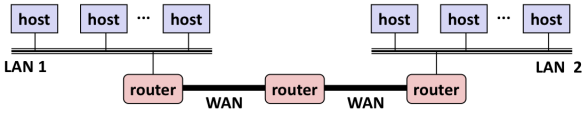


Next Level: Bridged Ethernet Segment



Conceptual View of LANs

Next Level: internets



Protocol: a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network

What Does an internet Protocol Do?

- **Provides a *naming scheme***
  - An internet protocol defines a uniform format for ***host addresses***
  - Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it

- **Provides a *delivery mechanism***
  - An internet protocol defines a standard transfer unit (***packet***)
  - Packet consists of ***header*** and ***payload***
    - Header: contains info such as packet size, source and destination addresses
    - Payload: contains data bits sent from source host

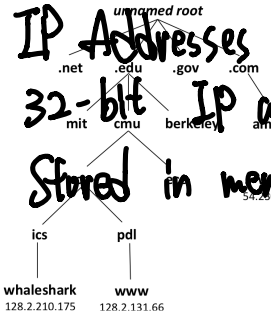
Global IP Internet (upper case)

- Most famous example of an internet
- Based on the TCP/IP protocal family
  - IP: Provides basic naming scheme and unreliable delivery capability of packets (datagrams) from host-to-host
  - UDP: Uses IP to provide unreliable datagram delivery from process-to-process
  - TCP: Uses IP to provide reliable byte streams from process-to-process over connections
- Accessed via a mix of Unix file I/O and functions from the sockets interface

A Programmer's View of the Internet

1. Hosts are mapped to a set of 32-bit IP addresses
2. The set of IP addresses is mapped to a set of identifiers called Internet domain names
3. A process on one Internet host can communicate with a process on another Internet host over a connection

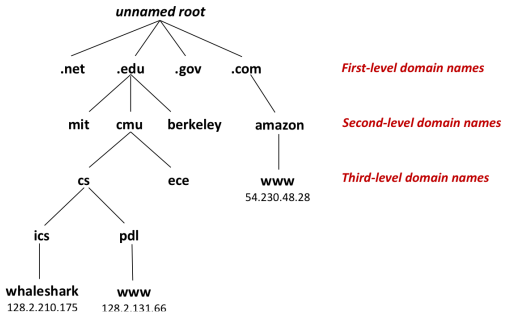
IP Addresses



32-bit IP addresses are stored in an IP address struct

Stored in memory in network byte order (big-endian byte order)

Internet Domain Names



Domain Naming System (DNS)

The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called DNS

localhost —— 127.0.0.1

whaleshark.ics.cs.cmu.edu —— 128.2.210.175

cs.mit.edu
eecs.mit.edu
> 18.62.1.6

www.twitter.com ——
- 104.244.42.129
- 104.244.42.65
- 104.244.42.193
- 104.244.42.1

ics.cs.cmu.edu —— don't map to any IP address

Internet Connections

Connection : point-to-point

Socket : An endpoint of a connection

port : a 16-bit integer that identifies a process

Anatomy of a Connection

A connection is uniquely identified by the socket addresses of its endpoints (socket pair)

Lecture 22: Network Programming : Part II

Generic socket address

```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.  */
};
```

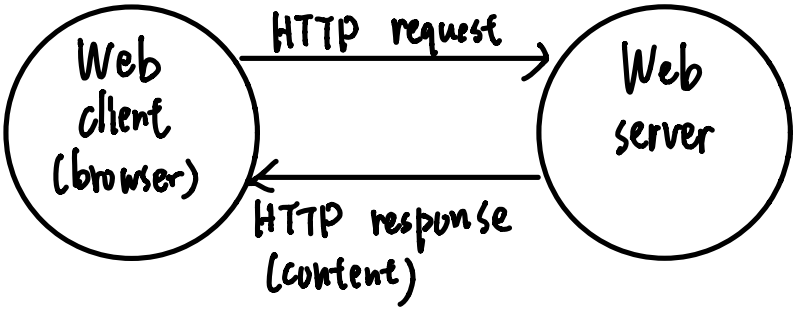
SA family: IPV4 or IPV6 or ...

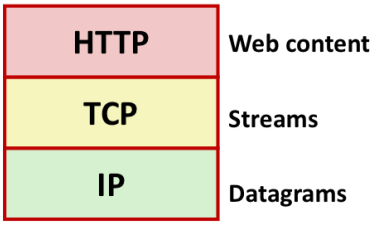
Connected vs Listening Descriptor

- Listening descriptor
  - End point for client connection requests
  - Created once and exists for lifetime of the server
- Connected descriptor
  - Endpoint of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

Web Server Basics

Client and servers communicate using the HyperText Transfer Protocol (HTTP)





Web servers return content to clients

- content: a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

The content returned in HTTP responses can be either static or dynamic

URL (Universal Resource Locator): http://www.cmu.edu:80/index.html

HTTP request is a request line, followed by zero or more request headers

- Request line: <method> <uri> <version>
- Request headers: <header name> : <header data>

HTTP response is a response line followed by zero or more response headers, possibly followed by content, with blank line ("\r\n") separating headers from content.

- Response line: <version> <status code> <status msg>
- Response headers: <header name> : <header data>

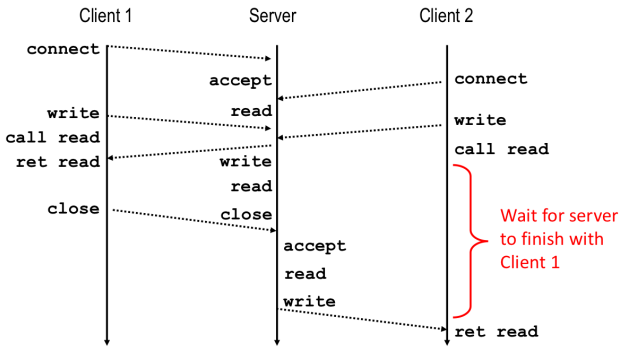
CGI: Common gateway interface

Lecture 23: Concurrent Programming

Concurrent Programming is hard

- Data Race
- Deadlock
- Livelock
- Starvation

Iterative Servers : Process one request at a time



3 Approaches for Writing Concurrent Servers

1. Process-based — private address space
2. Event-based — same address space
3. Thread-based — same address space

Process-based: client call connect

↓

server fork child

Event-based Servers: Server maintains set of active connections

Thread-based Servers: Similar to Process-based but using threads instead of processes

Concurrent Threads: Two threads are concurrent if their flows overlap in time

Lecture 24: Synchronization: Basics

Traditional View of a Process:

Process = process context + code, data, and stack

Alternative View of a Process:

Process = thread + (code, data, and kernel context)

- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)

The stacks of different threads from same process are in the same shared virtual memory space

Threads Memory Model

- Separation of data is not strictly enforced
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread

Mapping Variable Instances to Memory

- Global variables

- Def: Variable declared outside of a function
- Virtual memory contains exactly one instance of any global variable

– Local variables
- Def: Variable declared inside function without static attribute
- Each thread stack contains one instance of each local variable

– Local static variables
- Def: Variable declared inside function with the static attribute
- Virtual memory contains exactly one instance of any local static variable

Enforcing Mutual Exclusion

We must synchronize the execution of the threads so that they can never have an unsafe trajectory

Need to guarantee mutually exclusive access for each critical section

Semaphores

Semaphores: non-negative global integer synchronization variable

Manipulated by P and V operations

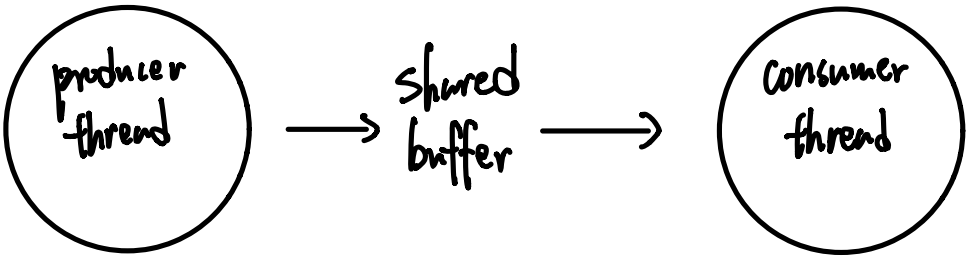
P(s) – is $s \neq 0$, decrement s by 1 and return immediately
– $s = 0$, suspend thread until s becomes non-zero and the thread is restarted by a V operation
– After restarting, the P operation decrements s and return control to the caller

V(s) – Increment s by 1
– If there are any threads blocked in a P operation waiting for S to become non-zero, then restart exactly one of those

thread, which then completes its P operation by decrementing s.

Lecture 25: Synchronization: Advanced

Producer – Consumer Problem



Common synchronization pattern

- Producer waits for empty slots, inserts them in buffer, and notifies consumer
- Consumer waits for item, removes it from buffer, and notifies producer

Producer - Consumer on 1-element Buffer

Producer Thread

```
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* Produce item */
    item = i;
    printf("produced %d\n",
            item);

    /* Write item to buf */
    P(&shared.empty);
    shared.buf = item;
    V(&shared.full);
  }
  return NULL;
}
```

Consumer Thread

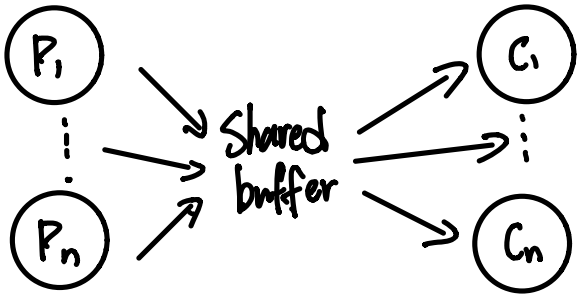
```
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* Read item from buf */
    P(&shared.full);
    item = shared.buf;
    V(&shared.empty);

    /* Consume item */
    printf("consumed %d\n", item);
  }
  return NULL;
}
```

Why 2 Semaphores for 1-Entry Buffer

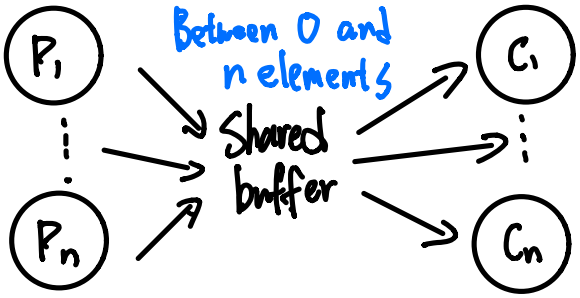
Consider multiple producers & multiple consumers



Producers will contend with each to get empty

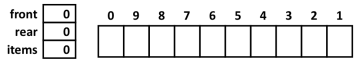
Consumers will contend with each other to get full

Producer - Consumer on an n-element Buffer



Circular Buffer (n = 10)

- Store elements in array of size n
- items: number of elements in buffer
- Empty buffer:
  - front = rear
- Nonempty buffer
  - rear: index of most recently inserted element
  - front: index of next element to remove – 1 (mod n)
- Initially:



Require a mutex and two counting semaphores:

- mutex: enforces mutually exclusive access to the buffer and counters
- slots: counts the available slots in the buffer
- items: counts the available items in the buffer

Sequential Circular Buffer Code

```
init(int v)
{
   items = front = rear = 0;
}
```

```
insert(int v)
{
   if (items >= n)
       error();
   if (++rear >= n) rear = 0;
   buf[rear] = v;
   items++;
}
```

```
int remove()
{
   if (items == 0)
       error();
   if (++front >= n) front = 0;
   int v = buf[front];
   items--;
   return v;
}
```

Insert:

Removing an item from a shared buffer:

```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);              /* Wait for available item */
    P(&sp->mutex);              /* Lock the buffer         */
    if (++sp->front >= sp->n)   /* Increment index (mod n) */
        sp->front = 0;
    item = sp->buf[sp->front];  /* Remove the item         */
    V(&sp->mutex);              /* Unlock the buffer       */
    V(&sp->slots);              /* Announce available slot */
    return item;
}
```

sbuf.c

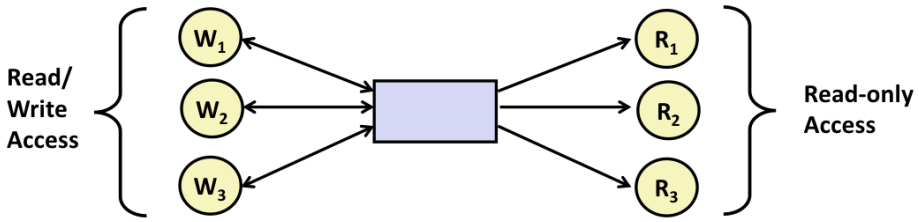
Remove:

Removing an item from a shared buffer:

```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);              /* Wait for available item */
    P(&sp->mutex);              /* Lock the buffer         */
    if (++sp->front >= sp->n)   /* Increment index (mod n) */
        sp->front = 0;
    item = sp->buf[sp->front];  /* Remove the item         */
    V(&sp->mutex);              /* Unlock the buffer       */
    V(&sp->slots);              /* Announce available slot */
    return item;
}
```

sbuf.c

Readers – Writers Problem



- Reader threads only read the object
- Writer threads modify the object (read/write access)
- Writers must have exclusive access to the object
- Unlimited number of readers can access the object

- First readers - writers problem (favors readers)
  - No reader should be kept waiting unless a writer has already been granted permission to use the object
  - A reader that arrives after a waiting writer gets priority over the writer.
- Second readers – writers problem (favors writers)
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting

Solution to First Readers-Writers Program:

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

Writers:

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Some concurrency issues

Races: A race occurs when correctness of the program depends on one thread reaching point x before another threads reaches point y

Deadlock: A process is deadlocked if it is waiting for a condition that will never be true

Typical Scenario:

Process 1 acquires A, waits for B

Process 2 acquires B, waits for A

Thread Safety:

Functions called from a function must be thread safe

Thread-Unsafe Functions

1. Failing to protect shared variables
2. Relying on persistent state across multiple function invocations
3. Returning a pointer to a static variable
4. Calling thread-unsafe functions

Reentrant Functions: A function is reentrant if it accesses no shared variables when called by multiple threads.