

Lecture 2 : Bits, Bytes and Integers – Part 1

1 Byte = 8 bits

Hex Decimal Binary

		0000
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

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

Encoding Integers

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

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

Numeric Ranges

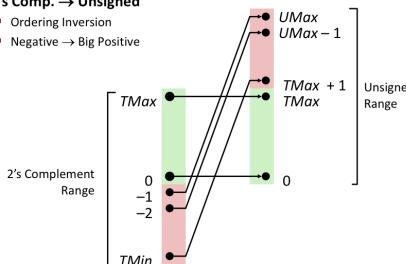
$$\text{Unsigned Values : } \begin{cases} U_{\min} = 0 \\ U_{\max} = 2^w - 1 \end{cases}$$

$$\text{Two's Complement Values: } \begin{cases} T_{\min} = -2^{w-1} \\ T_{\min} = T_{\max} + 1 \\ T_{\max} = 2^w - 1 \\ T_{\max} = 2 * T_{\max} + 1 \end{cases}$$

Conversion Visualized

- 2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



- If there is a mix of unsigned and signed in single expression, signed values implicitly Cast to unsigned

Sign Extension

- Given w-bit signed integer x
- Convert it to $(w+k)$ bit integer with same value
⇒ Make k copies of sign bit

Truncation

- Given $w+k$ bit signed or unsigned integer X
- Convert it to w -bit integer x' (with small value for "Small enough" X)
⇒ Drop top k bits

Lecture 3 : Bits, Bytes and Integers - Part 2

$$\text{Unsigned Addition: } S = UAdd_w(u, v) \\ (\text{Discard Carry}) \quad = (u + v) \bmod 2^w$$

Two's Complement Addition:

If sum $\geq 2^{w-1}$: Becomes Negative
At most once

If sum $< -2^{w-1}$: Becomes Positive
At most once

Power-of-2 Multiply with Shift
 $u < k$ gives $u < 2^k$

Unsigned Power-of-2 Divide with Shift
 $u > k$ gives $Lu / 2^{k-j}$ (logical shift)

Signed Power-of-2 Divide with Shift

$x >> k$ gives $Lx / 2^{k-j}$ (arithmetic shift)

$$(x + (1 << k) - 1) >> k \Rightarrow \lfloor (x + 2^{k-1}) / 2^k \rfloor$$

$$\Rightarrow \lceil x / 2^k \rceil$$

Complement & Increment : $\sim x + 1 = -x$

Size-t: Unsigned value with length = word size

Byte Ordering :

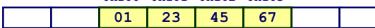
- Big Endian: Sun (Oracle SPARC), PPC Mac, Internet

- Least significant byte has highest address

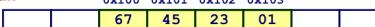
- Little Endian: x86, ARM processors running Android, iOS, and Linux
- Least significant byte has lowest address

Example: 4-byte value: 0x01234567

Big Endian $0x100 \ 0x101 \ 0x102 \ 0x103$



Little Endian $0x100 \ 0x101 \ 0x102 \ 0x103$



Lecture 4: Floating Point

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
- exp field encodes E (but is not equal to E)
- frac field encodes M (but is not equal to M)

s	exp	frac
1	8-bits	23-bits

▪ Single precision: 32 bits

≈ 7 decimal digits, $10^{±38}$

s	exp	frac
1	8-bits	23-bits

▪ Double precision: 64 bits

≈ 16 decimal digits, $10^{±308}$

s	exp	frac
1	11-bits	52-bits

Normalized Value:

$$\exp \neq 00\cdots 0 \ \& \ \exp \neq 11\cdots 1$$

$$E = \text{Exp} - \text{Bias}$$

$$\text{Bias} = 2^{k-1} - 1 \quad \begin{array}{l} \text{single precision: 127} \\ \text{Double precision: 1023} \end{array}$$

$$M = 1. \underline{xxx\cdots x} \quad \begin{array}{l} \text{frac field} \\ \text{frac} \end{array}$$

Denormalized Values

$$\exp = 00\cdots 0$$

$$E = 1 - \text{Bias}$$

$$M = 0. \underline{xx\cdots x}$$

Special Values

$$\exp = 11\cdots 1$$

$$\text{frac} \neq 00\cdots 0 \Rightarrow \infty$$

$$\text{frac} \neq 11\cdots 1 \Rightarrow \text{NaN}$$

Tiny Floating Point Example

s	exp	frac
1	4-bits	3-bits

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

$$n: E = \text{Exp} - \text{Bias}$$

$$d: E = 1 - \text{Bias}$$

$$\text{Bias} = 7$$

$$(-1)^0 (0+1/4)*2^{-6}$$

Dynamic Range (Positive Only)

s	exp	frac	E	Value
0	0000 000	-6	0	0
0	0000 001	-6	1/8*1/64 = 1/512	1/512
0	0000 010	-6	2/8*1/64 = 2/512	2/512
...
0	0000 110	-6	6/8*1/64 = 6/512	6/512
0	0000 111	-6	7/8*1/64 = 7/512	7/512
0	0000 100	-6	8/8*1/64 = 8/512	8/512
0	0000 001	-6	9/8*1/64 = 9/512	9/512
...
0	0110 110	-1	14/8*1/2 = 14/16	14/16
0	0110 111	-1	15/8*1/2 = 15/16	15/16
0	0111 000	0	8/8*1 = 1	1
0	0111 001	0	9/8*1 = 9/8	9/8
0	0111 010	0	10/8*1 = 10/8	10/8
...
0	1110 110	7	14/8*128 = 224	224
0	1110 111	7	15/8*128 = 240	240
0	1111 000	n/a	inf	largest norm

Rounding : 1. BBGRXXX

Guard bit Round bit Sticky bit

(LSB of result)

Round = 1, Sticky ≠ 0 → > 0.5

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

- `x == (int)(float) x` X
- `x == (int)(double) x` ✓
- `f == (float)(double) f` ✓
- `d == (double)(float) d` X
- `f == -(-f);` ✓
- `2/3 == 2/3.0` X
- `d < 0.0` $\Rightarrow ((d*2) < 0.0)$ ✓
- `d > f` $\Rightarrow -f > -d$ ✓
- `d * d >= 0.0` ✓
- `(d+f)-d == f` X

Lecture 5: Machine-Level Programming I: Basic

x86-64 Integer Registers: %rax, %rbx,

%rcx, %rdx, %rsi, %rdi, %rsp, %rbp,

%r8, %r9, ..., %r15

$D(Rb, Ri, S) \Rightarrow \text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri]] + D]$

Instructions:

`movq Src, Dst`

`leaq Src, Dst`

$(\text{leaq}(\%rdi, \%rdi, 2), \%rcx)$

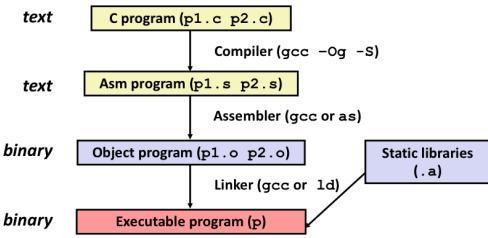
$\Rightarrow \text{rax} = 2 * \text{rdi} + \text{rdi}$

```

addq Src,Dest Dest = Dest + Src incq Dest Dest = Dest + 1
subq Src,Dest Dest = Dest - Src decq Dest Dest = Dest - 1
imulq Src,Dest Dest = Dest * Src negq Dest Dest = -Dest
salq Src,Dest Dest = Dest << Src notq Dest Dest = ~Dest
sraq Src,Dest Dest = Dest >> Src
shrq Src,Dest Dest = Dest ^ Src
xorg Src,Dest Dest = Dest & Src
andq Src,Dest Dest = Dest | Src

```

Turning C into Object Code



Lecture 6: Machine-Level Programming II: Control

%rsp: Current Stack Top

%rip: Instruction pointer

Condition Codes: CF, ZF, SF, OF

CF: Carry Flag (for unsigned)

SF: Sign Flag (for signed)

ZF: Zero Flag

OF: Overflow Flag (for signed)

Cmpq Src 2, Src 1:

Cmpq b, a like computing a-b without setting destination

testq Src 2, Src 1:

testq b, a like computing a&b without setting destination

Set X Dest:

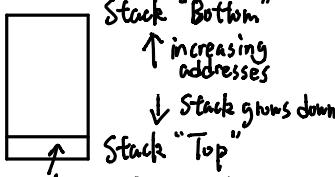
Set low-order byte of destination Dest to 0 or 1 based on combinations of condition codes.

jX Instructions:

Jump to different part of code depending on condition codes

Lecture 7: Machine-Level Programming III

x86-64 Stack



%rsp: Contains lowest stack address

push Src: ① Fetch operand at Src

② Decrement %rsp by 8

③ Write operand at address given by

popq Dest: ① Read value at address given by %rsp.

② Increment %rsp by 8

③ Store value at Dest (usually a register)

(The memory doesn't change, only %rsp)

Procedure call: call label

① Push return address on stack

② Jump to label

Procedure return: ret

① Pop address from stack

② Jump to address

Passing data

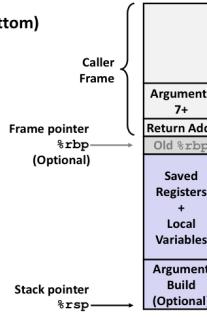
First 6 arguments: %rdi, %rsi, %rdx
%rcx, %r8, %r9

Return value: %rax

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

Caller-saved: %r10, %r11

Callee-Saved: %rbx, %r12, %r13, %r14

Lecture 8: Machine-Level Programming IV: Data

T A[L];

• Array of data type T and Length L

int (*A3)[3]: A3 is a pointer to a array with 3 integers (A3 → int[3])

T A[i][L];

• A[i] address: A + i * (C * sizeof(T))

• A[i][j] address: A + i * (C * k) + j * k
= A + (i * C + j) * K

Structure Alignment:

① Primitive data type requires K bytes

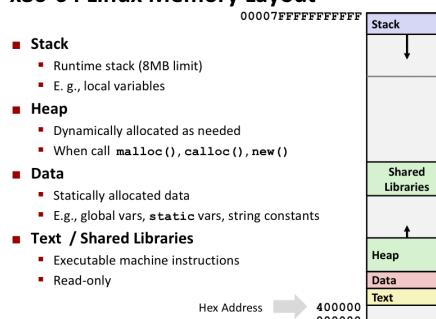
② Address must be multiple of K

Unions: ① Allocated according to largest element

② Can only use one field at a time

Lecture 9: Machine Level Programming : Advanced

x86-64 Linux Memory Layout



Buffer Overflow

① Stack Smashing Attacks: Overwrite normal return address with address of some other code

② Code Injection Attacks: Input string containing byte representation of executable code; Overwrite return address A with address of buffer B

Protection: ① Avoid Overflow Vulnerability in code

② System-Level Protection: Random stack offsets; Nonexecutable code segments

③ Stack Canaries: Place special value on stack just beyond buffer

Return-Oriented Programming Attacks

Construct program from gadgets

① Sequence of instructions ending in ret

② Code positions fixed from run to run (0xc3)

③ Code is executable

Lecture 10 : Code Optimization

$b \times X \rightarrow X \ll 4$

Move call to strlen outside of loop

CPE = cycles per OP

Loop Unrolling

Branch Prediction Numbers

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

Lecture 11 : The Memory Hierarchy

RAM { SRAM : fast, expensive

DRAM : slow, cheap

Disk: Capacity = (# bytes/sector) x (avg. # sectors/track) x (# tracks/surface) x (# surfaces/platter) x (# platters/disk)

Disk Access Time

① Seek time (Tang seek)

② Rotational latency (Tang rotation)

$$\text{Tang rotation} = \frac{1}{2} \times \frac{1}{\text{RPMs}} \times \frac{60 \text{ sec}}{1 \text{ min}}$$

③ Transfer time (Tang transfer)

$$\text{Tang transfer} = \frac{1}{\text{RPMs}} \times \frac{1}{(\text{avg. # sectors/track})} \times \frac{60 \text{ sec}}{1 \text{ min}}$$

Locality { Temporal locality

{ Spatial locality

3 Types of Cache Misses

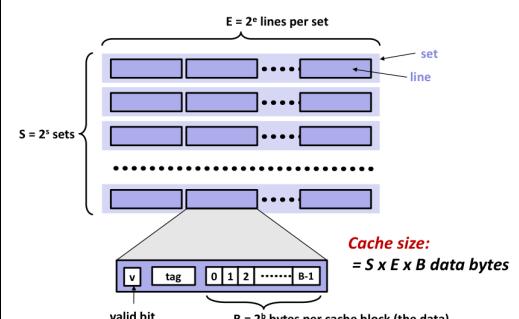
① Cold (compulsory) miss

② Capacity miss

③ Conflict miss

Lecture 12: Cache Memories

General Cache Organization (S, E, B)



Address of word: t bits s bits b bits

tag set index block offset

Cache size: $= S \times E \times B$ data bytes

Direct Mapped Cache: $E = 1$

E-way Set Associative Cache

Write-through { Write-back

Write-miss: s Write-allocate

{ No-write-allocate

Typical: Write-through + No-write-allocate

Write-back + Write-allocate

Lecture 13: Linking

What do linkers do?

① Step 1: Symbol resolution

• associates each symbol reference with exactly one symbol definition

② Step 2: Relocation

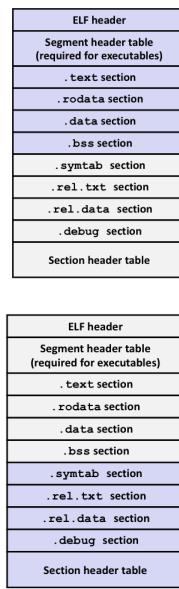
• merges separate code and data sections into single

- Relocates symbols from their relative locations in the files to their final absolute memory locations in the executable
- Updates all references to these symbols to reflect their new positions.

ELF Object File Format

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
.symtab section
▪ Symbol table
▪ Procedure and static variable names
▪ Section names and locations
.rel.text section
▪ Relocation info for .text section
▪ Addresses of instructions that will need to be modified in the executable
▪ Instructions for modifying.
.rel.data section
▪ Relocation info for .data section
▪ Addresses of pointer data that will need to be modified in the merged executable
.debug section
▪ Info for symbolic debugging (gcc -g)
Section header table
▪ Offsets and sizes of each section

Elf header
▪ Page size, virtual addresses memory segments (sections), segment sizes.
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.text section
.rel.data section
.debug section
Section header table



- Exit: Called once but never returns
- fork: Called once but return twice (return 0 to child, child's PID to parent)

Reaping Child Processes:

When process terminates, it still consumes system resources

- wait: Parent reaps a child by calling "wait"

execve: Loading and Running Programs

- Overwrites code, data and stack
- Retains PID, open files and signal context
- Called once and never returns

Lecture 15: Exceptional Control Flow:

Signals and Nonlocal Jumps

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

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

Kernel sends a signal to a destination process by updating some state in the context of the destination process.

Destination process receives a signal!

- ① Ignore the signal

- ② Terminate the Process

- ③ Catch the signal by executing a user-level function called signal handler

< pending: A signal is sent but not yet received
block: A process blocks the receipt of certain signals

- Kernel maintains pending and blocked bit vectors in the context of each process

- pending: represents the set of pending signals
 - Kernel sets bit k in pending when a signal of type k is delivered
 - Kernel clears bit k in pending when a signal of type k is received
- blocked: represents the set of blocked signals
 - Can be set and cleared by using the `sigprocmask` function
 - Also referred to as the `signal mask`.

Each signal type has a predefined default action

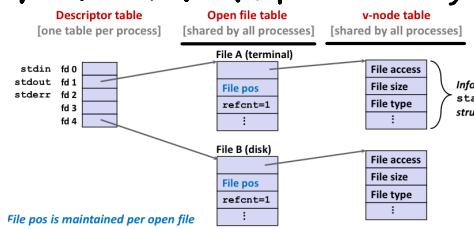
Lecture 16: System-Level I/O

A Linux file is a sequence of n bytes
open: returns a small identifying integer file descriptor.

- 0: standard input (`stdin`)
- 1: standard output (`stdout`)
- 2: standard error (`stderr`)

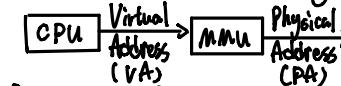
File Metadata: data about data.

How the Unix Kernel Represents Open Files



I/O Redirection: dup2(4,1)
cause fd=1 (`stdout`) to refer to disk file pointed at by `fd=4`

Lecture 17: Virtual Memory: Concepts

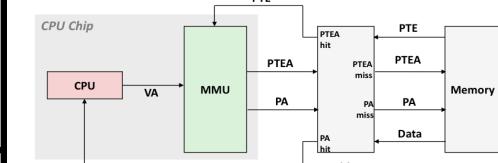


Page Table: A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages

Page Hit vs Page Fault

working set: At any point of time, programs tend to access a set of active virtual pages. Each process has its own virtual address space

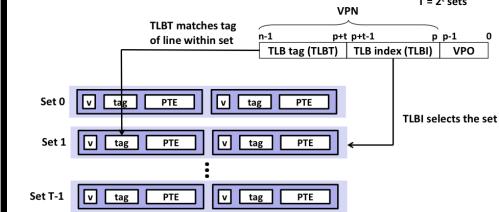
Address Translation Symbols: $N = 2^n$ (VA)
 $M = 2^m$ (PA), $P = 2^p$ (Page size), VPO: virtual page offset, VPN: virtual page number, PPD, PPN



TLB: Translation Lookaside Buffer

- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

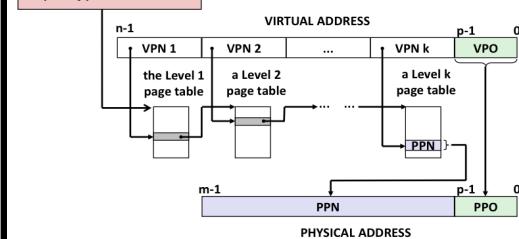
- MMU uses the VPN portion of the virtual address to access the TLB:



Lecture 18: Virtual Memory: Systems

K-Level Page Table

Page table base register (CR3) part of process context



Lecture 19: Dynamic Memory Allocation: Basic Concepts

- Def: Aggregate payload P_k
 - `malloc(p)` results in a block with a payload of p bytes
 - After request 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
 - $U_k = (\max_{i \leq k} P_i) / H_k$

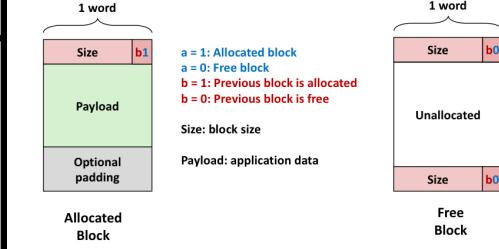
internal fragmentation: $\text{payload} < \text{block size}$

external fragmentation: there is enough aggregate heap memory, but no single free block is large enough

Method 1: Implicit list using length-links all blocks



need allocated/free tag for each block
first fit vs next fit vs best fit



Static

- Symbol only visible in enclosing scope
- Stored in either .bss or .data

Program symbols are either strong or weak

- Strong: procedures and initialized globals
- Weak: uninitialized globals
- Very weak: uninitialized globals declared with `extern`

Linker's Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
- Rule 2: Given a strong symbol and multiple weak symbols, choose the strong
- Rule 3: If there are multiple weak symbols, pick an arbitrary one
- Rule 4: Never pick a "very weak" symbol

Lecture 14: Exceptional Control Flow:

Exceptions and Processes

Exception: An exception is a transfer of control to the OS kernel in response to some event

ECF ← Asynchronous — Interrupts (I/O...)

Synchronous Traps (intentional)
Faults (unintentional)

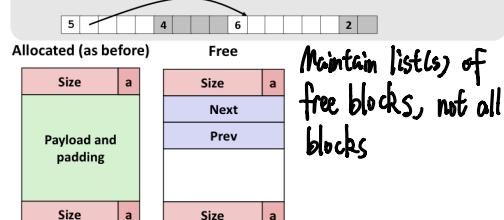
Faults: Page fault
Aborts: Segmentation fault

Process: an instance of a running program
Logical control flow & private address space

Processes: running, stopped, terminated

Lecture 20 : Dynamic Memory Allocation : Advanced Concepts

Method 2: **Explicit free list** among the free blocks using pointers



Segregated List (SegList) Allocators: Each size class of blocks has its own free list

Lecture 21: Network Programming : Part I

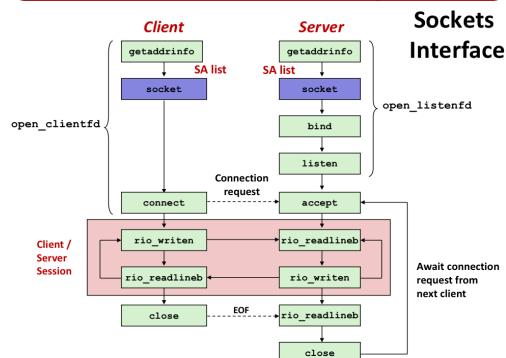
IP Addresses: 32-bit IP addresses are stored in an "IP address struct".

- IP addresses are always stored in memory in network byte order (big-endian byte order)

Clients and Servers communicate by sending streams of bytes over connections.

- A socket is an endpoint of a connection
- Socket address is an IP address : port pair
- A port is a 16-bit integer that identifies a process

Lecture 22: Network Programming : Part II



HTTP Requests

Request line: <method> <uri> <version>

- A URL is a type of URI

Request headers: <header name> : <header data>

HTTP Response

Response line: <version> <status code> <status msg>

Response headers: <header name> : <header data>

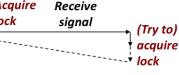
- Content-Type
- Content-Length

Lecture 23: Concurrent Programming

Deadlock:

```
void catch_child(int signo) {
    printf("Child exited!\n");
    if (kill(wpid, WNOHANG) > 0) continue; // reap all children
}
```

- Print code:
- Acquire lock
 - Do something
 - Release lock



Race :

- two or more threads in a single process access the same memory location concurrently, and at least one of the accesses is for writing, and the threads are not using any exclusive locks to control their accesses to that memory.

Starvation:

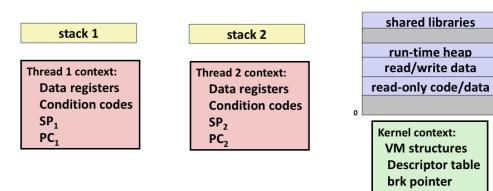
Process = process context + code, data, and stack

Process = thread + code, data, and kernel context

A Process With Multiple Threads

- 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)

Thread 1 (main thread) Thread 2 (peer thread)



Second Version

// Pseudo code for the second version of reader-writer problem:
int readcount, writecount; (initial value = 0)
semaphore mutex_1, mutex_2, mutex_3, w, r; (initial value = 1)

READER: <pre>while(true) { wait(mutex_3); wait(r); readcount++; if readcount == 1 then wait(w); signal(mutex_1); signal(r); signal(mutex_3);</pre>	WRITER: <pre>while(true) { wait(mutex_2); writecount := writecount + 1; if writecount == 1 then wait(w); signal(mutex_1); signal(w); wait(mutex_2); writecount := writecount - 1; if writecount == 0 then signal(w); signal(mutex_1);</pre>
---	--

Lecture 24: Synchronization: Basic

- 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.

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

Semaphore: non-negative global integer synchronization variable

P(S): If S is nonzero, then decrements S by 1 and return immediately

V(S): Increments S by 1

Lecture 25: Synchronization: Advanced

Producer-Consumer on 1-element Buffer

Initially: empty==1, full==0

Producer Thread <pre>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; }</pre>	Consumer Thread <pre>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; }</pre>
---	--

Variants of Readers-Writers

- 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 Problem

Readers: <pre>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); } }</pre>	Writers: <pre>void writer(void) { while (1) { P(w); /* Writing here */ V(w); } }</pre>
--	--