15-213 Recitation: Final Exam Review

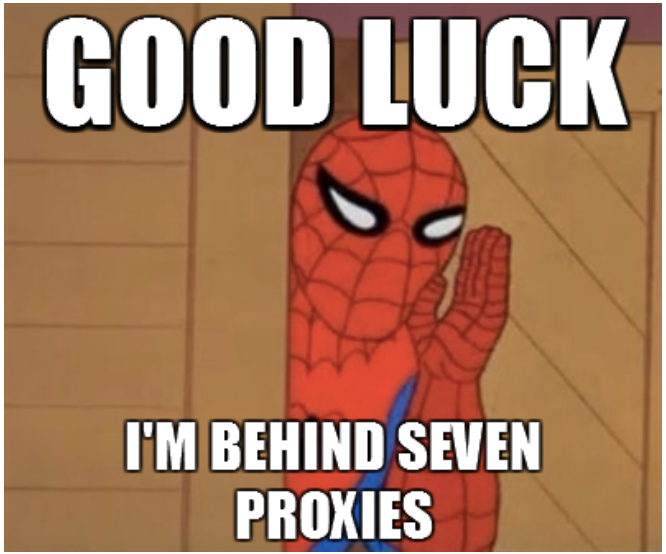
April 27, 2015

Arjun Hans, Felipe Vargas

Agenda

- Proxy Lab
- Final Exam Details
- Course Review
- Practice Problems

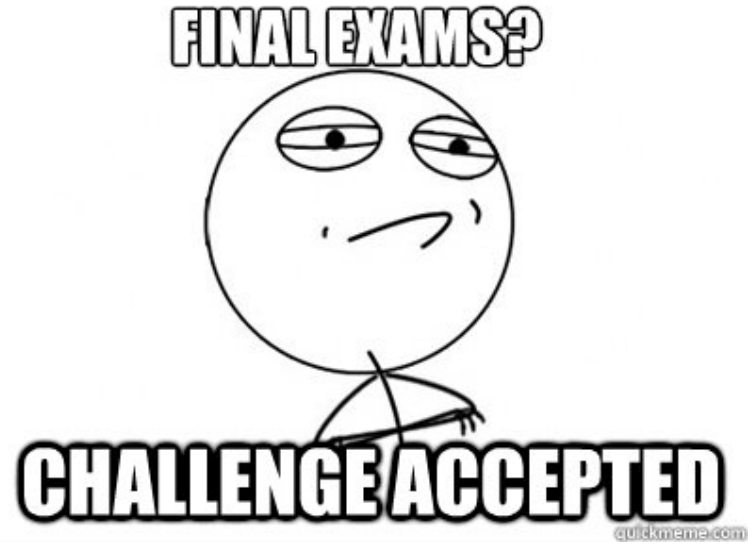
Proxy Lab



… is due tomorrow!

- Due Tuesday, April 28th
- Penalty late days allowed…
  - … but no grace days. Please finish on time!!!
- Reminder: we will test your proxy manually
  - http://www.cs.cmu.edu/~213/index.html
  - http://csapp.cs.cmu.edu
  - http://www.cmu.edu
  - http://www.amazon.com
- We will read your code
  - Correctness issues (race conditions, robustness, etc)
  - Style points: make your code shine! (write clean, well-documented, well-modularized code

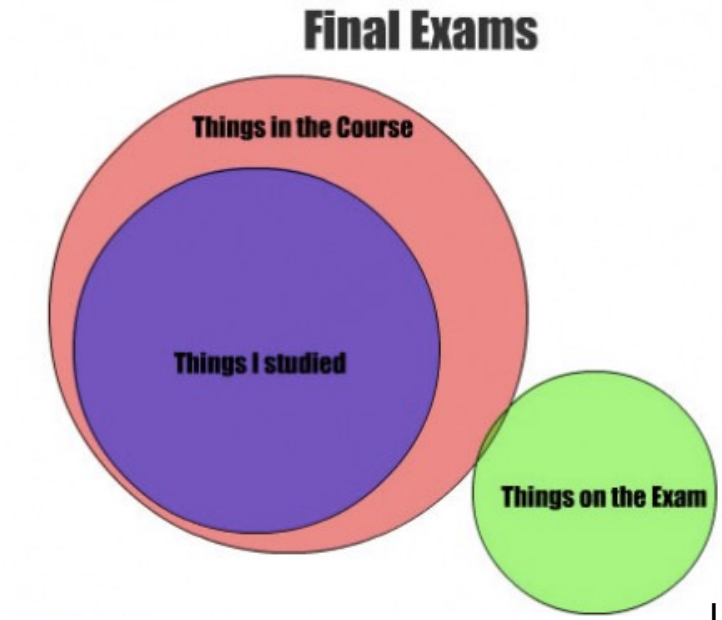
Final Exam Details



Final Exam Details

- **Mon May 4th to Fri May 8th**
  - Sign-ups will open soon!
- 10 problems, nominal time is 2-3 hours, but you get 6 hours!
- Review session to take place in the near future! Stay tuned…
- Cumulative: Chapters 1-3, 6-12
- 2 double-sided 8 1/2 x 11 sheets of notes
  - No pre-worked problems
  - Scratch paper will be provided

Course Review



Lol, not really

Course Review

- Integers/Floats
  - properties/arithmetic rules
- Assembly
  - basic operators/memory addressing
  - control flow
  - procedures/stacks
  - arrays/structs
  - x86 vs x86-64
- Memory Hierarchy
  - caches (address translation/implementation)
  - locality/cache friendly code
- Exceptional Control Flow
  - exceptions
  - processes (syscalls, properties)
  - signals (handlers, masks, synchronization)

Linking?!? … maybe

Course Review (con)

- Virtual Memory
  - uses (caching, memory management/protection)
  - implementation (page tables, TLB)
  - address translation
  - dynamic memory allocation
- File IO
  - syscalls (open, read/write, dup/dup2)
  - file-descriptor/file-entry tables
  - Rio package (buffered/unbuffered IO)
- Networking
  - sockets API
  - networking terminology (protocols, DNS, LAN's)
- Synchronization
  - pthreads API
  - thread safety
  - scheduling problems (starvation, readers-writers, producers-consumers)
  - concurrency problems (deadlock, livelock)

Course Review
In-Depth

A long time ago, in a galaxy far, far way…

In-depth Review

- Cover key-concepts from each chapter
  - Not in-depth; just things you should know/brush up on
- Describe common test-questions
  - Not a guarantee; just an indication of what to expect
- Outline tips/strategies to attack exam problems

Integers/Floats Concepts

- Integers:
  - Arithmetic/encodings for signed/unsigned integers
  - Translation between decimal/binary/hexadecimal
  - Bitwise operations
  - Casting rules (sign/zero extension)
- Floats:
  - Encoding rules (normalized/denormalized regions)
  - Calculations (bias, exponent/fractional values)
  - Special values (infinity, NaN)
  - Rounding (to even, generally)
- Miscellaneous:
  - Endian-ness (big: most significant byte stored at *lowest* address, little: most significant byte stored at *highest* address)

Integers/Floats Exam Problems/Tips

- Integers:
  - True/False for identities
  - Bit-representation of decimal values
  - Decimal value of expressions, given variable values
- Floats:
  - Provide binary representation/rounded decimal value given encoding formats (no. exponent/fractional bits)
- Tips:
  - For identities, try ‘extreme values’ (Int min/max, 0, -1, 1) to check for counter-examples
  - Write down values of min/max norm/denorm numbers given format parameters first (can then easily classify decimal values)
  - Know bit-patterns of key values (min/max norm/denorm values, infinity, NaN)

Assembly Concepts

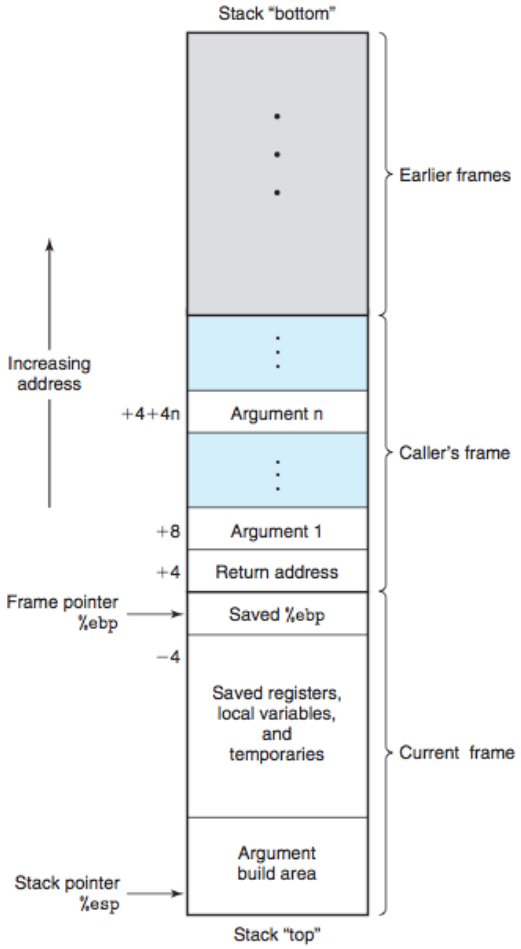
- Basics
  - Registers (%rax, %eax, %ax, %al; 64, 32, 16, 8 bits)
  - Arithmetic operations (op <dest> <src>, generally)
  - Memory addressing (immediates, registers;
  - eg $Imm(E_b,E_i,s) = M[Imm+R[E_b]+R[E_i]\cdot s]$ with mov, $Imm+R[E_b]+R[E_i]\cdot s$ with leal)
  - Suffix indicates data-type (l: long, b: byte, s: short, etc)
- Control Flow
  - cmp S1, S2 => S2 - S1, test S1, S2 => S1 & S2)
  - jumps: direct, indirect (switch statements), conditional (je, jne, etc)
  - identify if/else (comparison, goto)
  - identify loop constructs (translate into do-while loop, with init value/check outside loop, then update/check inside loop)
  - condition codes (zero, overflow, carry, signed flags)
- Pointer Arithmetic
  - given *T* a*, *a + i = a + i * sizeof(T)* address
  - given *T* a*, **a* reads/writes *sizeof(T)* bytes

Assembly Concepts (Arrays/Structs/Unions)

- Arrays:
  - Contiguous array of bytes
  - *T A[n]*: allocate array of *n * sizeof(T)* bytes; *a[i]* is at address *a + T * i*
  - Nested arrays: T a[M][N]: M arrays of N elements each (M rows, N columns)
- Structs:
  - Combination of heterogenous elements, occupying disjoint spaces in memory
  - Alignment: address multiple an element can be located on
  - Alignment rules of types (char: 1 byte, short: 2 bytes, ints: 4 bytes, etc)
  - Machine-dependent (Windows vs Linux, IA32 vs x86-64, etc)
  - Entire struct aligned to *maximum alignment* (for usage in arrays)
- Unions:
  - Single object can be referred using multiple types
  - All elements share space in memory

Assembly Concepts (Procedures/Stacks)

- Key Instructions:
  - **push S**: R[%esp] <- R[%esp] - 4, S <- M[R[%esp]]
  - **pop S**: S <- M[R[%esp]], R[%esp] < R[%esp] + 4
  - **call <proc>**: push ret. addr, jump to proc instruction
  - **leave**: mov %esp, %ebp, pop %ebp
  - **ret**: pop %eip
- Key Registers
  - %esp: stack-pointer (push/pop here)
  - %ebp: base-pointer (base of stack frame for procedure)
  - %eip: instruction pointer (address of next instruction)
- Miscellaneous
  - Arguments pushed in reverse order, located in caller frame, above return address/saved %ebp
  - Caller vs Callee saved registers
  - Routines generally start with saving %ebp of calling function/end with restoring %ebp of calling function



Assembly Concepts (x86 vs x86-64, Miscellaneous)

- Size Comparisons:
  - x86: 32-bits, x86-64: 64-bits (q: quad, 64-bits)
- x86-64 Procedures
  - arguments passed via registers (order: %rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - stack-frames usually have fixed size (move %rsp to required location at start of function)
  - base-pointer generally not needed (as stack-pointer is now fixed; can be used as reference point)
  - procedures generally don't need stack-frame to store arguments (only when more arguments needed are they spilled onto the stack)
- Miscellaneous:
  - special arithmetic operations (imull: R[%edx]:R[%eax] ← S × R[%eax], idivl: R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S)
  - conditional moves: move value into register, if appropriate flags set
  - buffer-overflow attacks: concept and defenses (stack-randomization/nop-sleds, canaries)

Assembly Problems/Tips

- Assembly Translation
  - Annotations (how register values change, where jumps lead to) help
  - Know conditional/loop translation (determine the condition, identity of the iterator, etc)
  - Know where arguments are located (0x8(%ebp) onwards for x86, special registers for x86-64)
  - Identify 'patterns':
    set to zero: (xor %eax, %eax, shr $0x1f, %eax)
    check if zero: (test %esi, %esi)
    array-indexing (eg: (%edi, %ebx, 4): %edi + 4 * %ebx for accessing elem in integer array)
- Stacks:
  - Know the diagram (location of args, base pointer, return address)
  - Go over buflab (locating the return-address/stack-pointer, writing content to the stack, etc)
  - Otherwise, relatively simple code-tracing
- Struct Alignment:
  - Double-check the alignment of special types (usually provided in question; add them to your cheatsheet)
  - Minimize padding: place elements with maximum alignment constraints at start of struct (may still have to pad the struct itself, though)
  - Mapping assembly fragments to code-snippets for struct field access requires drawing the struct diagram, then determining offset of accessed field.

Assembly Problems/Tips

- ‘M & N’ Array Dimensions:
  - Derive expressions for array elements a[i][j] given dimensions
  - eg given array int arr[M][N],a[i][j] is at address (arr + 4 * (N * i + j))
  - Follow assembly trace to determine coefficients M/N
- Switch Statements:
  - Look for this: *jmpq *jtaddr* (jump to address of jump table, plus some address offset, usually multiple of value we’re casing on).
  - Value at jump table address gives address of instruction to jump to
  - Determine which values map to the same set cases(‘fall-through’ behavior), default case

Jump Table

0x400598:	0x0000000000400488	0x0000000000400488
0x4005a8:	0x000000000040048b	0x0000000000400493
0x4005b8:	0x000000000040049a	0x0000000000400482
0x4005c8:	0x000000000040049a	0x0000000000400498

Jump Table Addresses

Instruction Addresses

Memory Hierarchy

- Principle:
  - Larger memories: slower, cheaper; Smaller memories: faster, more expensive
  - Smaller memories act as caches to larger memories
- Locality:
  - Temporal: reference same data in the near future
  - Spatial: reference data around an accessed element
- Cache Implementation:
  - <Tag bits><Set bits><Block Offset bits> indexing of address
  - Tag: uniquely identify an address in a set
  - Set: determine which set the line should go in
  - Block-offset: determine which byte in line to access
  - Valid bit: set when line is inserted for first time
  - Eviction policies (LRU, LFU, etc)
- Cache Math:
  - m address bits (M = 2^m = size address space);
  - s sets bits (S = 2^s = no. sets)
  - b block-offset bits (B = 2^b = line size in bytes);
  - t = m - (s + b) tag bits
  - E: no lines per set;
  - Cache Size = S * E * B
- Miss Types
  - Cold: compulsory misses at start, cache is warming up
  - Capacity: not enough space to store full data set in cache
  - Conflict: access pattern leads to ‘thrashing’ of elements in cache (map to same cache lines)

Memory Hierarchy Problems/Tips

- Potential Questions:
  - Precisely analyze cache performance (no. hits/misses/evictions, access time penalty, etc)
  - Approximate cache performance (hit/miss rate)
  - Qualitatively analyze cache design principles (cache size, line size, set associativity, etc)
- Tips:
  - Compute key quantities first (line size, no. sets, etc) from provided parameters
  - Mapping each address to a set/line is helpful (look for trends in the hit/miss patterns). Generally will need to write address in binary to extract values.
  - Row-major access has better cache performance than column-major access
  - Remember: we access the cache only on a miss. All the data on the same line as the element that caused the miss is loaded into the cache

Processes Concepts

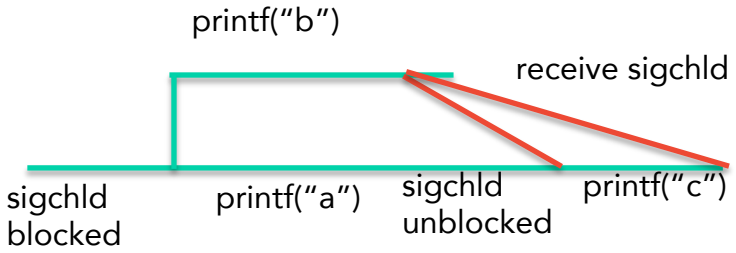
- Key Ideas:
  - Concurrent flow: execution is *concurrent* with other processes
  - Private address space: own local/global variables (own stack/heap)
  - *Inherit* values from parent (global values, file-descriptors, masks, handlers, etc).
  - Changes are then private to the process
  - Process *can* share state (eg file-table entries)
- fork:
  - Creates a new process once, Called once, returns twice
  - 0 returned to child, pid of child process returned to parent
- execve:
  - loads/runs new program in context of process.
  - Called, once, never returns (except in case of error)
- waitpid:
  - suspends calling process until process in *wait-set* terminates; reaps terminated child, returns its pid
  - +ve pid: wait set contains single child; -ve pid: wait set contains all children
  - options: return with 0 if none terminated, reap stopped processes, etc (see textbook)
  - status: can be examined with macros, return signal no. (see textbook)

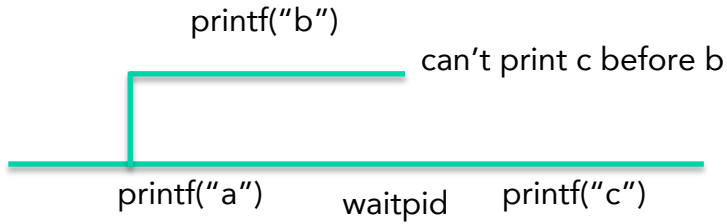
Signals Concepts

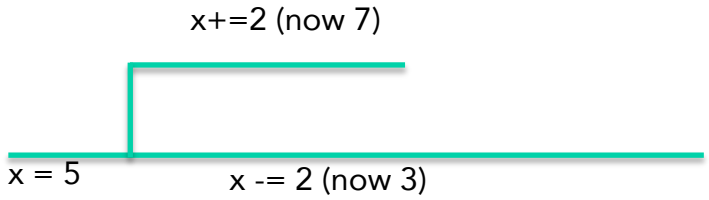
- Key Ideas:
  - message notifying process of some event; sent via the kernel
  - caught using a signal handler, else handled with default behavior (eg ignore, terminate program)
  - pending signals *not* queue'd (are *blocked*)
  - signal handlers *can be interrupted* by other signals
- Kill:
  - send signal to process/process group
  - +ve pid: send to single process, -ve pid: send to process group with paid = abs(pid)
- Masks:
  - bit-vector containing signals
  - manipulated using syscalls to empty/fill set, add/delete signals
  - set mask of calling process via *sigprocmask* (block, unblock, set); also can save old mask
  - sigsuspend: suspend process until signal *not* in provided mask received

ECF Problems/Tips

- Output Problems:
  - choose possible outputs/list all possible outputs
  - typically child is forked, multiple processes concurrently running
  - simple print statements, parent/child could modify variables, read/write to/from files, etc
- Output Problem Tips:
  - Draw timeline indicating events occurring along each process's trajectory, consider interleaving of execution
  - Note when a process is suspended, waiting for another process to terminate/send signal (using waitpid/sigsuspened)
  - Note when a process can receive a signal; consider what happens when signal handler is invoked at different points







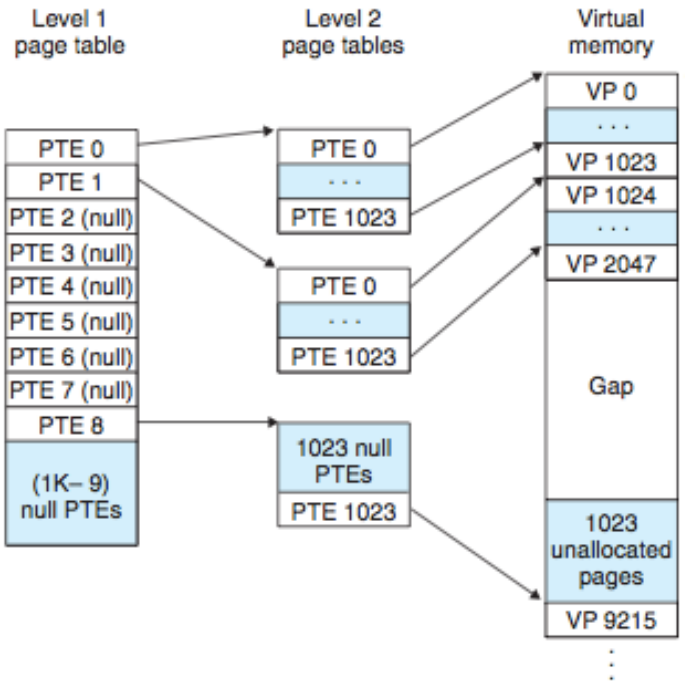
parent/child have own view of memory

Virtual Memory Concepts

- Virtual vs Physical Addresses:
  - Virtual addresses used by CPU, translated to physical addresses before sent to memory
- Implementation:
  - Page Table: maps virtual addresses to physical addresses (addresses resident on *pages*). Mappings known as page-table entries
  - TLB: MMU requests TLB to check if the page-table entry present. If so, return the physical address; else, search in page table.
  - Page-fault: choose victim page, page contents to disk, insert new page/ update PTE, reset the faulting instruction
- Address Translation:
  - VPN: <TLB tag><TLB set>, VPN maps to PPN in page table
  - VPO and PPO are same for physical/virtual addresses (offset into page)
  - Physical Address: <PPN><PPO>, Virtual Address: <VPN><VPO>

Virtual Memory Problems/Tips

- Potential Questions:
  - Given virtual address/size of virtual/physical address spaces, check if physical address is stored in TLB/page table.
  - Outline operations performed in accessing page
- Tips:
  - Understand the translation process (what happens in a page hit/fault)
  - Draw the binary-representation of the address first to extract the PN/PO and the TLB tag/set
  - Multi-level page tables: VPN is divided into VPN's (generally equally sized) for each page-table (<VPN 1><VPN 2> … <VPN k>).
  - For i < k, VPN i is base address of PT. VPN k is address of VP



Dynamic Memory Allocation Concepts

- **Evaluation:**
  - **Throughput**: no. requests/operations per unit time
  - **Utilization**: ratio of memory requested to memory allocated at peak
- **Fragmentation:**
  - **Internal:** due to overheads (headers/footers/padding/etc)
  - **External**: due to access pattern
- **Design-Spaces:**
  - **Implicit Free-List**: no pointers between free blocks, full heap traversal
  - **Explicit Free-List**: pointers between free-blocks, iterate over free blocks
  - **Segregated Free-Lists**: explicit-free lists of free-blocks based on size ranges
- **Search Heuristics:**
  - **First-Fit:** return the first block found
  - **Next-Fit**: first-fit, maintains rover to last block searched, search starts at rover
  - **Best-Fit**: returns the block that best fits the requested block size
- **Coalescing Heuristic:**
  - **Immediate**: coalesce whenever free'ing block/extending heap
  - **Deferred**: coalesce all free blocks only when free-block can't be found
- **Free-block Insertion Heuristics:**
  - **LIFO**: insert newly coalesced block at head of free-list

Dynamic Memory Allocation Problems/Tips

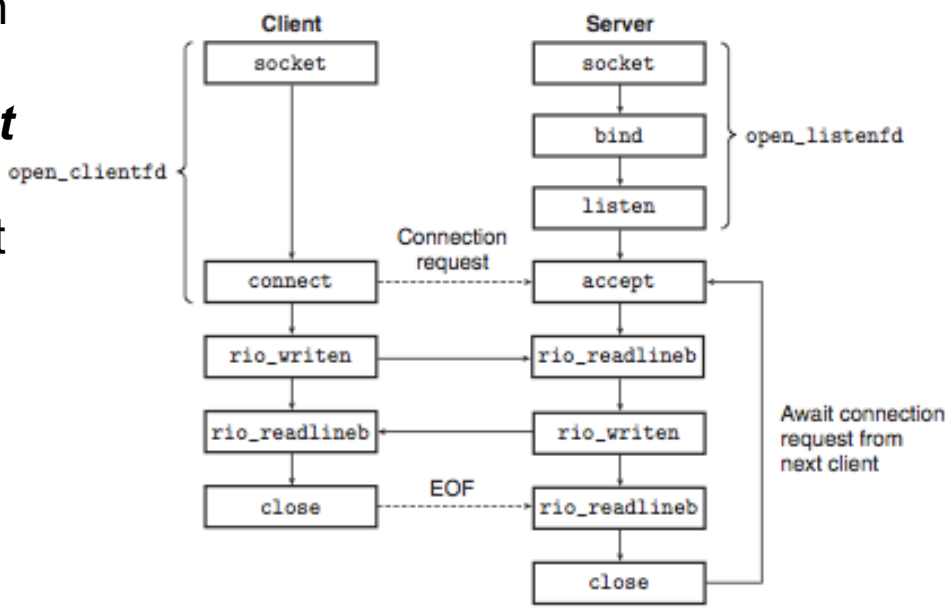
- Potential Questions:
  - Simulate an allocator, given a set of heuristics (fill in the heap with values corresponding to header/footer/pointers)
  - Provide correct macro definitions to read/write parts of an allocated/ free block, given specifications
  - Analyze an allocator qualitatively (impact of design decisions on performance) and quantitatively (compute util ratio, internal fragmentation, etc)
- Tips:
  - Go over your malloc code/textbook code; understand your macros (casting, modularization, etc)
  - Go over block-size calculations (alignment/minimum block size)
  - Practice simple request pattern simulations with different heuristics

IO Concepts

- Overview:
  - file-descriptor: keeps track of information about open file
  - Standard streams: stdin: 0, stdout: 1, stderr: 2
  - File-table is *shared* among *all* processes (each process has its own *descriptor table*)
- Important Syscalls:
  - open(filename, flags, mode): open file using flags/mode, return fd
  - write(fd, buf, size): write up to ‘size’ characters in ‘buf’ to file
  - read(fd, buf, size): read up to ‘size’ characters from file into ‘buf'
  - dup(fd): return file-descriptor pointing to same location as fd
  - dup2(fd1, fd2): fd2 points to location of fd1
  - close(fd): close a file, restores descriptor to pool
- Flags:
  - O_RDONLY: read-only, O_WRONLY: write-only
  - O_RDWR: read/write, O_CREAT: create empty file, if not present
  - O_TRUNC: truncate file, if present, O_APPEND: append to file

Networking Concepts

- Sockets API:
  - socket: end-point for communication (modeled like a file in Unix)
  - **Client**: create ***socket***, then ***connect*** to server (***open_clientfd***)
  - **Server**: create ***socket***, ***bind*** socket to server address, ***listen*** for connection requests and return *listening file-descriptor* (***open_listenfd***),
  - Server ***accepts*** connection, return *connected file-descriptor*
  - Client/Server communicate via reading/writing from file-descriptors at end-points of channel



Networking Concepts (con)

- Terminology
  - **Networking Components:** router, hub, bridge, LAN's, etc
  - **HTTP**: HyperText Transfer Protocol, used to transfer hypertext
  - **HyperText**: text with references (hyperlinks) to other immediately accessible text
  - **TCP**: Transmission Control Protocol, provides reliable delivery of data packets (ordering, error-checking, etc)
  - **IP**: Internet Protocol, naming scheme (IPv4: 32 bits, IPv6: 128 bits)
  - **DNS**: Domain Naming System (domain names; use gethostbyaddr/ gethostbyname/etc to retrieve DNS host entries)
  - **URL:** name/reference of a resource (eg **http://www.example.org/wiki/Main_Page**)
  - **CGI:** Common Gateway Interface (standard method to generate dynamic content on web-pages; interface between server/scripting programs)
  - **Dynamic Content:** construction is controlled by server-side scripts (using client inputs
  - **Static Content:** delivered to user exactly as stored

Synchronization Concepts

- Threads:
  - logical flow in context of a process (can be multiple threads per process). Scheduled by kernel, identified by tid
  - own stack, share heap/globals
- Pthreads API
  - pthread_create: run routine in context of new thread with args
  - pthread_join: block thread until specified thread terminates, reaps resources held by terminated thread
  - pthread_self: get tid of calling thread
  - pthread_detach: detaches joinable thread, thread can reap itself
- Thread Un-safe Functions:
  - Class 1: do not protect shared variables
  - Class 2: preserve state across multiple invocations (str_tok, rand)
  - Class 3: return pointer to static variable (gethostbyname)
  - Class 4: call other thread-unsafe functions

Synchronization Concepts (con.)

- Locking Primitives:
  - **Semaphores**: counter used to control access to resource (init with sem_init(sem_t *sem, 0, unsigned int value)
  - P(s): decrements s if s is > 0, else blocks thread (sem_wait)
  - V(s): increments s, chooses arbitrary thread to proceed if multiple threads blocked at P (sem_post)
  - **Mutexes**: binary semaphores (0/1 value)
- Concurrency Problems:
  - **Deadlock**: no threads can make progress (circular dependency, resources can't be released). e.g. thread 1 has A, waiting for B; thread 2 has B, waiting for A)
  - **Livelock:** thread constantly change wrt each other, but neither makes progress (eg both threads detect deadlock, actions mirror each other exactly)

Synchronization Concepts (con.)

- Starvation:
  - some threads never get serviced (eg unfair scheduling, priority inversion, etc). Consequence of readers-writers solution
- Consumers-Producers:
  - shared a bounded buffer with n slots
  - producer adds items to buffer, if empty spaces present
  - consumer removes items from buffer, if items present
  - implementation: semaphores to keep track of no. slots available/no. items in buffer, mutex to add/remove item from buffer
- Readers-Writers:
  - multiple readers can access resource at a time
  - only one writer can access resource at a time
  - policies: favor readers/favor writers. Either may lead to starvation
  - implementation (favor readers): keep track of no. readers; first reader prevents writers from proceeding, last reader allows writers to proceed

Synchronization Problems/Tips

- Potential Questions:
  - Is there a race on a value?
  - Add locking primitives to solve variants of readers/writers, producers/consumers problems
  - Identify/fix thread-unsafe function
- Tips:
  - Draw diagrams to determine thread execution ordering (similar to processes); determine shared resources
  - Copying values to malloc'd memory blocks usually solves problems of resource sharing
  - Sharing globals/addresses of stack-allocated variables among threads requires synchronization (pthread_join, locking)
  - Acquire/release locks in reverse order (acquire A,B; release B, A); ensure that locks are released at end of all execution paths
  - Understand textbook solutions to readers/writers, producers/consumers problems (need/ordering of locks, usage of counters)

Practice Problems

Should NOT be you on the final!

Floating Point

Floating point encoding. In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different 6-bit formats:

Format A:

- There is one sign bit s.
- There are $k = 3$ exponent bits. The bias is $2^{k-1} - 1 = 3$.
- There are $n = 2$ fraction bits.

Format B:

- There is one sign bit s.
- There are $k = 2$ exponent bits. The bias is $2^{k-1} - 1 = 1$.
- There are $n = 3$ fraction bits.

For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 - \text{bias}$. For normalized numbers, $E = e - \text{bias}$.

Value	Format A Bits	Format B Bits
Zero	0 000 00	0 00 000
One		
1/2		
11/8		

Assembly Translation

```
mystery_sort:
        jmp     loop1_check

loop1:
        xor     %rdx, %rdx
        mov     %rsi, %rcx
        jmp     loop2_check

loop2:
        mov     (%rdi, %rcx, 8), %rax
        cmp     %rax, (%rdi, %rdx, 8)
        jg      loop2_check
        mov     %rcx, %rdx

loop2_check:
        dec     %rcx
        test    %rcx, %rcx
        jnz     loop2

        dec     %rsi
        mov     (%rdi, %rsi, 8), %rax
        mov     (%rdi, %rdx, 8), %rcx
        mov     %rcx, (%rdi, %rsi, 8)
        mov     %rax, (%rdi, %rdx, 8)

loop1_check:
        test    %rsi, %rsi
        jnz     loop1

        ret
```

```
void mystery_sort (long* array, long len)
{
  long a, b, tmp;

  while (_____ > _____)
  {

    a = _____;

    for (b = _____; b > _____; b--)
    {

      if (array[_____] > array{_____])
      {

        _____ = _____;
      }
    }

    len--;

    tmp = array[_____];

    array[_____] = array[_____];

    array[_____] = tmp;
  }
}
```

Switch Statement

Switch statements. The problem concerns code generated by GCC for a function involving a switch statement. The code uses a jump to index into the jump table:

```
  0x4004b7:       jmpq   *0x400600(,%rax,8)
```

Using GDB, we extract the 8-entry jump table:

```
  0x400600: 0x00000000004004d1 0x00000000004004c8
  0x400610: 0x00000000004004c8 0x00000000004004be
  0x400620: 0x00000000004004c1 0x00000000004004d7
  0x400630: 0x00000000004004c8 0x00000000004004be
```

Here is the block of disassembled code implementing the switch statement:

```
# on entry: %rdi = x, %rsi = y, %rdx = z
  0x4004b0:       cmp    $0x7,%edx
  0x4004b3:       ja     0x4004c8
  0x4004b5:       mov    %edx,%eax
  0x4004b7:       jmpq   *0x400600(,%rax,8)
  0x4004be:       mov    %edi,%eax
  0x4004c0:       retq
  0x4004c1:       mov    $0x3,%eax
  0x4004c6:       jmp    0x4004da
  0x4004c8:       mov    %esi,%eax
  0x4004ca:       nopw   0x(%rax,%rax,1)
  0x4004d0:       retq
  0x4004d1:       mov    %edi,%eax
  0x4004d3:       and    $0x19,%eax
  0x4004d6:       retq
  0x4004d7:       lea    (%rdi,%rdi,1),%eax
  0x4004da:       add    %esi,%eax
  0x4004dc:       retq
```

```
int test(int x, int y, int z)
{
  int result = 3;
  switch(z)
  {
    case ___:

      _____________;

    case ___:

    case ___:

      result = _____________;

      break;

    case ___:

      result = _____________;

    case ___:

      result = _____________;

      break;

    default:

      result = _____________;
  }
  return result;
}
```

Stacks

```
000000af <doSomething>:
af:   push   %ebp
b0:   mov    %esp,%ebp
b2:   sub    $0xc,%esp
b5:   mov    0x8(%ebp),%ecx
b8:   mov    $0x1,%eax
bd:   test   %ecx,%ecx
bf:   je     de <doSomething+0x2f>
c1:   mov    %ecx,%edx
c3:   shr    $0x1f,%edx
c6:   lea    (%ecx,%edx,1),%edx
c9:   sar    %edx
cb:   mov    0x10(%ebp),%eax
ce:   mov    %eax,0x8(%esp)
d2:   mov    %ecx,0x4(%esp)
d6:   mov    %edx,(%esp)
d9:   call   da <doSomething+0x2b>
de:   leave
df:   ret
```

```
int doSomething(int a, int b, int c){
    int d;
    if (a == 0){ return 1;}
    d = a/2;
    c = doSomething(d,a,c);
    return c;
}
```

Please draw a detailed stack diagram for this function in Figure 1 on the next page, starting with a function that calls this function and continuing for 2 recursive calls of this function. (That is, at least two stack frames that belong to this function). Please label everything you can.

Caches

The Hit or Miss Question

Given a 32-bit Linux system that has a 2-way associative cache of size 128 bytes with 32 bytes per block. Long longs are 8 bytes. For all parts, assume that `table` starts at address 0x0.

```
int i;
int j;
long long table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

A. This problem refers to code sample 1. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0								
1								
2								
3								

What is the miss rate of this code sample?

```
int i;
int j;
int table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

B. This problem refers to code sample above. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0								
1								
2								
3								

What is the miss rate of this code sample?

Processes/IO

Assume that the disk file `buffer.txt` contains the string of bytes `source`. Also assume that all system calls succeed. What will be output when this code is compiled and run? You may not need all the lines in the table given below.

Output Line Number	Output
1st line of output	
2nd line of output	
3rd line of output	
4th line of output	
5th line of output	
6th line of output	
7th line of output	
8th line of output	
9th line of output	

```
int main() {
  char c;
  int file1 = open("buffer.txt", O_RDONLY);
  int file2;

  read(file1, &c, 1);
  file2 = dup(file1);
  read(file1, &c, 1);
  read(file2, &c, 1);
  printf("1 = %c\n", c);

  int pid = fork();
  if (pid == 0) {
    close(file1);
    file1 = open("buffer.txt", O_RDONLY);

    read(file1, &c, 1);
    printf("2 = %c\n", c);
    read(file2, &c, 1);
    printf("3 = %c\n", c);

    exit(0);
  } else {
    waitpid(pid, NULL, 0);

    printf("4 = %c\n", c);

    close(file2);
    dup2(file1, file2);

    read(file1, &c, 1);
    printf("5 = %c\n", c);
    read(file2, &c, 1);
    printf("6 = %c\n", c);
  }

  return 0;
}
```

Signals

Code Snippet 1:

```
int main() {
    int pid = fork();
    if(pid > 0){
        kill(pid, SIGKILL);
        printf("a");
    }else{
        /* getppid() returns the pid
           of the parent process */
        kill(getppid(), SIGKILL);
        printf("b");
    }
}
```

Snippet 1 Outcome	Possible? (Y/N)
Nothing is printed.	
"a" is printed.	
"b" is printed.	
"ab" is printed.	
"ba" is printed.	
A process does not terminate.	

Code Snippet 2:

```
int a = 1;

void handler(int sig){
    a = 0;
}

void emptyhandler(int sig){
}

int main() {
    signal(SIGINT, handler);
    signal(SIGCONT, emptyhandler);

    int pid = fork();
    if(pid == 0){
        while(a == 1)
            pause();
        printf("a");
    }else{
        kill(pid, SIGCONT);
        printf("b");
        kill(pid, SIGINT);
        printf("c");
    }
}
```

Snippet 2 Outcome	Possible? (Y/N)
Nothing is printed.	
"ba" is printed.	
"abc" is printed.	
"bac" is printed.	
"bca" is printed.	
A process does not terminate.	

Virtual Memory

- The system has 1MB of virtual memory
- The system has 256KB of physical memory
- The page size is 4KB
- The TLB is 2-way set associative with 8 total entries.

The contents of the TLB and the first 32 entries of the page table are given below. **All numbers are in hexadecimal.**

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	1B	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

(a) How many bits are needed to represent the virtual address space? _____

(b) How many bits are needed to represent the physical address space? _____

(c) How many bits are needed to represent a page table offset? _____

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

Virtual address: `0x1F213`

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	Physical Address	0x

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

Virtual address: `0x14213`

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	Physical Address	0x

Synchronization

Synchronization. This problem is about using semaphores to synchronize access to a shared bounded FIFO queue in a producer/consumer system with an arbitrary number of producers and consumers.

- The queue is initially empty and has a capacity of 10 data items.
- Producer threads call the `insert` function to insert an item onto the rear of the queue.
- Consumer threads call the `remove` function to remove an item from the front of the queue.
- The system uses three semaphores: `mutex`, `items`, and `slots`.

A. What is the initial value of each semaphore?

```
mutex = _______

items = _______

slots = _______
```

B. Add the appropriate P and V operations to the psuedo-code for the `insert` and `remove` functions:

```
void insert(int item)
{
    /* Insert sem ops here */

    add_item(item);
    /* Insert sem ops here */

}
```

```
int remove()
{
    /* Insert sem ops here */

    item = remove_item();
    /* Insert sem ops here */

    return item;
}
```

Thread Safety

A. Why is the function `ts_next_prime` thread-unsafe?

```
struct big_number *next_prime(struct big_number current_prime) {
  static struct big_number next;

  next = current_prime;
  addOne(next);
  while (isNotPrime(next)) {
    addOne(next);
  }

  return &next;
}

struct big_number *ts_next_prime(struct big_number current_prime)
  return next_prime(current_prime);
}
```

C. Fill in the blanks below to fix `ts_next_prime`.

```
struct big_number *ts_next_prime(struct big_number current_prime) {
  struct big_number *value_ptr;

  struct big_number *ret_ptr = _____________________________________;
  sem_wait(&mutex);
  value_ptr = next_prime(current_prime);
  _____________________________________;
  sem_post(&mutex);

  return ret_ptr;
}
```

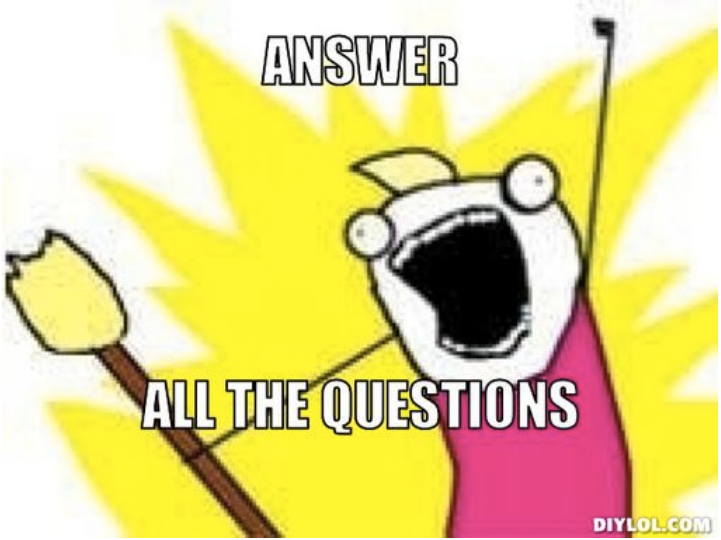
B. Assume the mutex guarding the call to `next_prime` is initialized correctly in the following code.

```
struct big_number *ts_next_prime(struct big_number current_prime) {
  struct big_number *value_ptr;

  sem_wait(&mutex);
  value_ptr = next_prime(current_prime);
  sem_post(&mutex);

  return value_ptr;
}
```

The following modification to the function is still not thread safe. Explain why, and show an example execution with two threads showing the problem?

Thread 1	Thread 2

Answers



Apologies for Arjun's handwriting. :(

Floating Point

Floating point encoding. In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different 6-bit formats:

Format A:

- There is one sign bit s.
- There are $k = 3$ exponent bits. The bias is $2^{k-1} - 1 = 3$.
- There are $n = 2$ fraction bits.

Format B:

- There is one sign bit s.
- There are $k = 2$ exponent bits. The bias is $2^{k-1} - 1 = 1$.
- There are $n = 3$ fraction bits.

For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 - \text{bias}$. For normalized numbers, $E = e - \text{bias}$.

Value	Format A Bits	Format B Bits
Zero	0 000 00	0 00 000
One	0 011 00	0 01 000
1/2	0 010 00	0 00 100
11/8	0 011 10	0 01 011

• Format A:
• smallest denorm: 0 000 01 (exp = 1−3 = −2, frac = 1/4, val = $2^{-2} \cdot 1/4 = 1/16$.
• largest denorm: 0 000 11, val = 3/16
• smallest norm: 0 001 00, val = 1/4
• largest norm: 0 110 11. exp = 6−3 = 3, frac = 1 + 3/4; val = 14

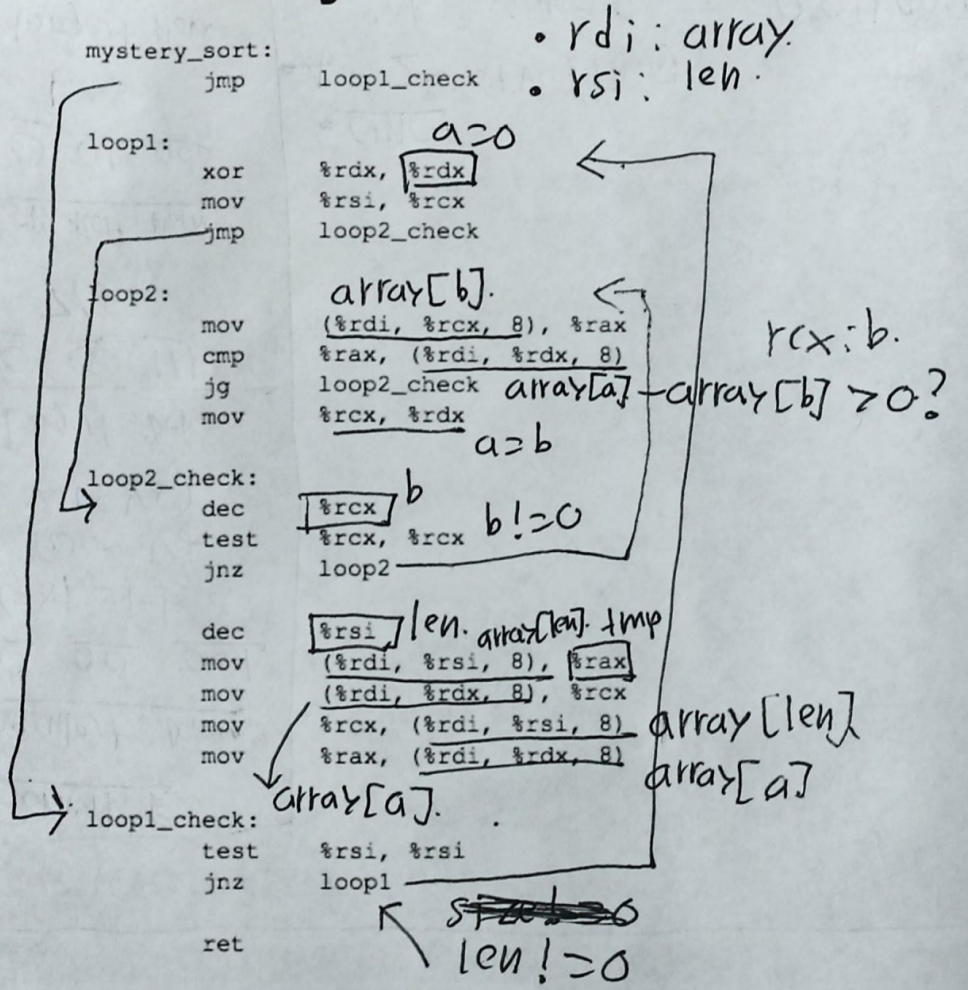
• Format B:
• smallest denorm: 0 00 001, (exp = 1−1 = 0, frac = 1/8, val = 1/8)
• largest denorm: 0 00 111, 7/8
• smallest norm: 0 01 000, 1.
• largest norm: 0 10 111, exp = 2−1 = 1, frac = 1 + 7/8, val = 15/4.

• 11/8:
* A: $\frac{11}{8} = (1 + \frac{3}{8}) \cdot 2^{3-3}$.
· 3/8: 0 11 → 10 (round even)
· 0 011 10 (3/2)
* B: $\frac{11}{8} = (1 + \frac{3}{8}) \cdot 2^{1-1}$.
· 3/8: 0 11, ⇒ 0 01 011.

• One:
• A: $2^{3-3} \cdot (1+0)$ [norm], exp = 3, frac = 0, 0 011 00.
• B: 0 01 000 (smallest norm)

• 1/2
• A: norm, $2^{-1} \cdot (1+0)$.
· exp = 2−3
· frac = 0, 0 010 00
• B: denorm, $4 \cdot \frac{1}{8}$; 0 00 100.

Assembly Translation




```
mystery_sort:
        jmp     loop1_check

loop1:
        xor     %rdx, %rdx
        mov     %rsi, %rcx
        jmp     loop2_check

loop2:
        mov     (%rdi, %rcx, 8), %rax
        cmp     %rax, (%rdi, %rdx, 8)
        jg      loop2_check
        mov     %rcx, %rdx

loop2_check:
        dec     %rcx
        test    %rcx, %rcx
        jnz     loop2

        dec     %rsi
        mov     (%rdi, %rsi, 8), %rax
        mov     (%rdi, %rdx, 8), %rcx
        mov     %rcx, (%rdi, %rsi, 8)
        mov     %rax, (%rdi, %rdx, 8)

loop1_check:
        test    %rsi, %rsi
        jnz     loop1

        ret
```

```
void mystery_sort (long* array, long len)
{
  long a, b, tmp;

  while (len > 0)
  {

    a = 0;

    for (b = len-1; b > 0; b--)
    {

      if (array[b] > array[a])
      {

        a = b;

      }
    }

    len--;

    tmp = array[len];

    array[len] = array[a];

    array[a] = tmp;
  }
}
```

Switch Statement

Switch statements. The problem concerns code generated by GCC for a function involving a switch statement. The code uses a jump to index into the jump table:

```
0x4004b7:       jmpq    *0x400600(,%rax,8)
```

Using GDB, we extract the 8-entry jump table:

```
                  0                 1
0x400600: 0x00000000004004d1 0x00000000004004c8
0x400610: 0x00000000004004c8 0x00000000004004be
0x400620: 0x00000000004004c1 0x00000000004004d7
0x400630: 0x00000000004004c8 0x00000000004004be
```

Here is the block of disassembled code implementing the switch statement:

```
# on entry: %rdi = x, %rsi = y, %rdx = z
  0x4004b0:       cmp     $0x7,%edx
  0x4004b3:       ja      0x4004c8
  0x4004b5:       mov     %edx,%eax
  0x4004b7:       jmpq    *0x400600(,%rax,8)
  0x4004be:       mov     %edi,%eax
  0x4004c0:       retq
  0x4004c1:       mov     $0x3,%eax
  0x4004c6:       jmp     0x4004da
  0x4004c8:       mov     %esi,%eax
  0x4004ca:       nopw    0x(%rax,%rax,1)
  0x4004d0:       retq
  0x4004d1:       mov     %edi,%eax
  0x4004d3:       and     $0x19,%eax
  0x4004d6:       retq
  0x4004d7:       lea     (%rdi,%rdi,1),%eax
  0x4004da:       add     %esi,%eax
  0x4004dc:       retq
```

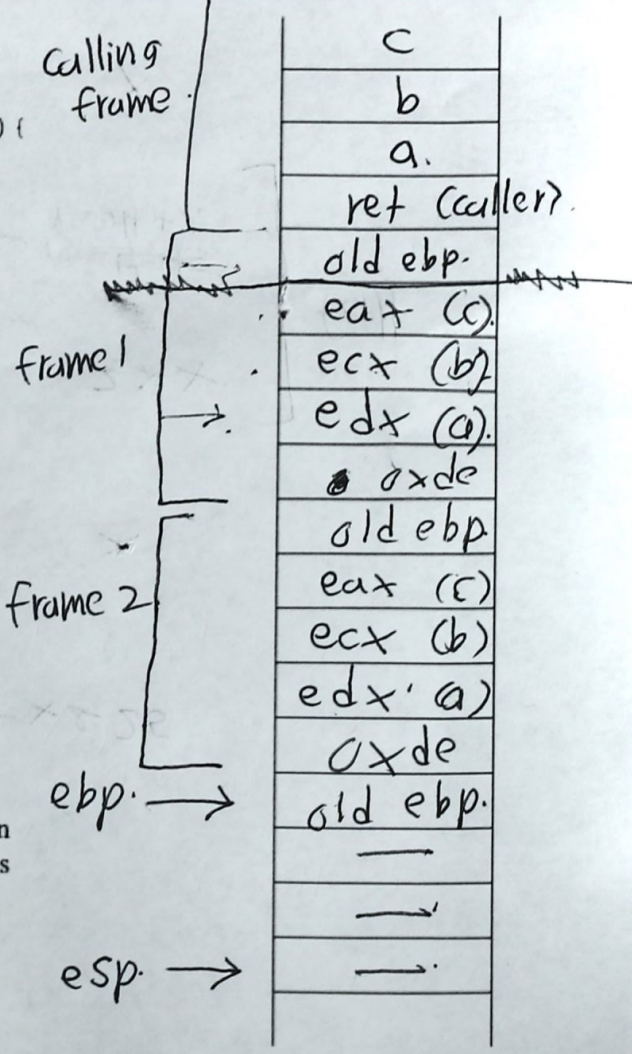
- 1,2,6 → 4004c8: (default case)
- 3,7 → 4004be
- 0 → 4004d1
- 4 → 4004c1
- 5 → 4004d7

```
int test(int x, int y, int z)
{
  int result = 3;
  switch(z)
  {
    case 0:
      x = x & 25;
    case 3:
    case 7:
      result = x;
      break;
    case 5:
      result = 2*x;
    case 4:
      result = result + y;
      break;
    default:
      result = y;
  }
  return result;
}
```

Stacks

```
000000af <doSomething>:
af:   push   %ebp
b0:   mov    %esp,%ebp
b2:   sub    $0xc,%esp
b5:   mov    0x8(%ebp),%ecx
b8:   mov    $0x1,%eax
bd:   test   %ecx,%ecx
bf:   je     de <doSomething+0x2f>
c1:   mov    %ecx,%edx
c3:   shr    $0x1f,%edx
c6:   lea    (%ecx,%edx,1),%edx
c9:   sar    %edx
cb:   mov    0x10(%ebp),%eax
ce:   mov    %eax,0x8(%esp)
d2:   mov    %ecx,0x4(%esp)
d6:   mov    %edx,(%esp)
d9:   call   da <doSomething+0x2b>
de:   leave
df:   ret
```

```
int doSomething(int a, int b, int c){
    int d;
    if (a == 0){ return 1;}
    d = a/2;
    c = doSomething(d,a,c);
    return c;
}
```



Please draw a detailed stack diagram for this function in Figure 1 on the next page, starting with a function that calls this function and continuing for 2 recursive calls of this function. (That is, at least two stack frames that belong to this function). Please label everything you can.

Caches

The Hit or Miss Question

2 lines per set - 5 block-offset bits.

Given a 32-bit Linux system that has a 2-way associative cache of size 128 bytes with 32 bytes per block.
Long longs are 8 bytes. For all parts, assume that `table` starts at address 0x0.

- $\frac{128}{32} = 4$ lines
- 2 sets. (1 set bit)

```
int i;
int j;
long long table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

iterate down columns (cache-unfriendly!)

```
int i;
int j;
int table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

same problem as before.

A. This problem refers to code sample 1. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0	M: 1	M	M	M	M	M	M	M
1	M: 2	M	M	M	M	M	M	M
2	M: 3	M	M	M	M	M	M	M
3	M: 4	M	M	M	M	M	M	M

What is the miss rate of this code sample?

miss rate: 1

- Cache line: stores 4 elements

- line 0 set 0,
- &a[0][0]: 0 (0 00000)
- &a[1][0]: 64. (0 00000) set 0, line 1

Conflict Misses.

- &a[2][0]: 128 (0 00000) set 0, line 0
- &a[3][0]: 192. (0 00000) set 0, line 0

- one accessing a[1][0], we find that line that contained data (loaded by accessing a[0][0]) has been evicted.
- pattern continues.

B. This problem refers to code sample above. In the table below write down in each space whether th element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0	M	H	H	H	H	H	H	H
1	M	H	H	H	H	H	H	H
2	M	H	H	H	H	H	H	H
3	M	H	H	H	H	H	H	H

What is the miss rate of this code sample?

miss rate: 1/8

- Cache line: stores 8 elements
  set 0, line 0

&a[0][0]: 0 (0 00000)
&a[1][0]: 32 (1 00000) set 0, line 0
&a[2][0]: 64 (0 00000) set 0, line 1
&a[3][0]: 96 (1 00000) set 1, line 1.

No conflict misses!

Processes/IO

Assume that the disk file `buffer.txt` contains the string of bytes `source`. Also assume that all system calls succeed. What will be output when this code is compiled and run? You may not need all the lines in the table given below.

Output Line Number	Output
1st line of output	u
2nd line of output	s
3rd line of output	r
4th line of output	u.
5th line of output	c
6th line of output	e.
7th line of output	
8th line of output	
9th line of output	

```
int main() {
  char c;
  int file1 = open("buffer.txt", O_RDONLY);
  int file2;

  read(file1, &c, 1);
  file2 = dup(file1);
  read(file1, &c, 1);
  read(file2, &c, 1);
  printf("1 = %c\n", c);

  int pid = fork();
  if (pid == 0) {
    close(file1);
    file1 = open("buffer.txt", O_RDONLY);

    read(file1, &c, 1);
    printf("2 = %c\n", c);
    read(file2, &c, 1);
    printf("3 = %c\n", c);

    exit(0);
  } else {
    waitpid(pid, NULL, 0);

    printf("4 = %c\n", c);

    close(file2);
    dup2(file1, file2);

    read(file1, &c, 1);
    printf("5 = %c\n", c);
    read(file2, &c, 1);
    printf("6 = %c\n", c);
  }

  return 0;
}
```

read 's' alias file1, file2.
read 'o'
read 'u'
print 'u'.
reset file1.
read 's'
print 's'
read 'r' (move parent's file1/file2)
print 'r'.
wait for child to terminate.
print 'u'
file2 points at file1
read 'c'.
print 'c'
read 'e'
print 'e'.

- Remember: processes share file-table entries. have private fd table

Signals

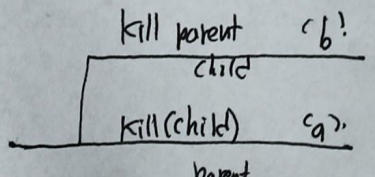
Code Snippet 1:

```
int main() {
    int pid = fork();
    if(pid > 0){
        kill(pid, SIGKILL);
        printf("a");
    }else{
        /* getppid() returns the pid
           of the parent process */
        kill(getppid(), SIGKILL);
        printf("b");
    }
}
```

parent: kill child.

child: kill parent.

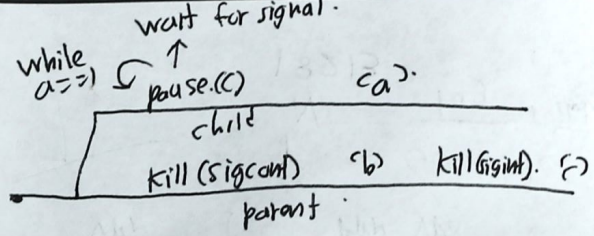
Snippet 1 Outcome	Possible? (Y/N)
Nothing is printed.	Y (both die).
"a" is printed.	Y
"b" is printed.	Y
"ab" is printed.	Y
"ba" is printed.	Y
A process does not terminate.	



- Could receive either sigkill at any time (both in parent/child)

Code Snippet 2:

```
int a = 1;

void handler(int sig){
    a = 0;
}

void emptyhandler(int sig){
}

int main() {
    signal(SIGINT, handler);
    signal(SIGCONT, emptyhandler);

    int pid = fork();
    if(pid == 0){
        while(a == 1)
            pause();
        printf("a");
    }else{
        kill(pid, SIGCONT);
        printf("b");
        kill(pid, SIGINT);
        printf("c");
    }
}
```



set a=0 (break while loop)

do nothing (awaken process from sleep)

Snippet 2 Outcome	Possible? (Y/N)
Nothing is printed.	N
"ba" is printed.	N
"abc" is printed.	N
"bac" is printed.	Y (parent prints b, c).
"bca" is printed.	Y (parent prints b first. before a)
A process does not terminate.	Y (child misses wakeups)

Virtual Memory

- The system has 1MB of virtual memory (2^{20})
- The system has 256KB of physical memory (2^{18}).
- The page size is 4KB ($2^{10} \cdot 2^2 = 2^{12}$).
- The TLB is 2-way set associative with 8 total entries. → 4 sets.

The contents of the TLB and the first 32 entries of the page table are given below. **All numbers are in hexadecimal.**

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	1B	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

(a) How many bits are needed to represent the virtual address space? 20

(b) How many bits are needed to represent the physical address space? 18

(c) How many bits are needed to represent a page table offset? 12.

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

0001 1111. 0010. 0001 0011.
VPN VPO

Virtual address: 0x1F213

Parameter	Value	Parameter	Value
VPN	0x 1F.	TLB Hit? (Y/N)	N.
TLB Index	0x 3.	Page Fault? (Y/N)	Y.
TLB Tag	0x 07.	Physical Address	0x 15213.

0001 1111
tag index

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

0001 0100 0010 0001 0011.
VPN PPN VPO

Virtual address: 0x14213

Parameter	Value	Parameter	Value
VPN	0x 14.	TLB Hit? (Y/N)	Y.
TLB Index	0x 0.	Page Fault? (Y/N)	N
TLB Tag	0x 05.	Physical Address	0x 18213

0001 0100
tag index

Thread Safety

A. Why is the function `ts_next_prime` thread-unsafe?

```
struct big_number *next_prime(struct big_number current_prime) {
  static struct big_number next;

  next = current_prime;
  addOne(next);
  while (isNotPrime(next)) {
    addOne(next);
  }

  return &next;
}

struct big_number *ts_next_prime(struct big_number current_prime)
  return next_prime(current_prime);
}
```

- references static data, returns ptr to that block.

B. Assume the mutex guarding the call to `next_prime` is initialized correctly in the following code

```
struct big_number *ts_next_prime(struct big_number current_prime)
  struct big_number *value_ptr;

  sem_wait(&mutex);
  value_ptr = next_prime(current_prime);
  sem_post(&mutex);

  return value_ptr;
}
```

- returns ptr to static data

The following modification to the function is still not thread safe. Explain why, and show an exam execution with two threads showing the problem?

C. Fill in the blanks below to fix `ts_next_prime`.

```
struct big_number *ts_next_prime(struct big_number current_prime) {
  struct big_number *value_ptr;

  struct big_number *ret_ptr = Mallac (sizeof (struct big_number));
  sem_wait(&mutex);
  value_ptr = next_prime(current_prime);
  memcpy(ret_ptr, value_ptr); ] same value as value_ptr's block.
  sem_post(&mutex);

  return ret_ptr;
}
```

different address for each invocation.

Thread 1	Thread 2
sem_wait.	
next_prime	
sem_post.	
⋮	sem_wait.
	next_prime.
	sem_post.
return value_ptr	return value_ptr

- value_ptr's block has been modified by thread 2.

Synchronization

Synchronization. This problem is about using semaphores to synchronize access to a shared bounded FIFO queue in a producer/consumer system with an arbitrary number of producers and consumers.

- The queue is initially empty and has a capacity of 10 data items.
- Producer threads call the `insert` function to insert an item onto the rear of the queue.
- Consumer threads call the `remove` function to remove an item from the front of the queue.
- The system uses three semaphores: `mutex`, `items`, and `slots`.

A. What is the initial value of each semaphore?

`mutex` = 1

`items` = 0

`slots` = 10

- see producers-consumers solution from textbook

B. Add the appropriate P and V operations to the psuedo-code for the `insert` and `remove` functions:

```
void insert(int item)
{
    /* Insert sem ops here */
    P(slots)
    P(mutex)

    add_item(item);
    /* Insert sem ops here */
    V(mutex)
    V(items)
}
```

```
int remove()
{
    /* Insert sem ops here */
    P(items)
    P(mutex)

    item = remove_item();
    /* Insert sem ops here */
    V(mutex)
    V(slots)

    return item;
}
```

Questions?
All the Best!