15-213 Recitation: Final Exam Review

December 1, 2014
Arjun Hans
Agenda

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

GOOD LUCK

I'M BEHIND SEVEN PROXIES
... is due soon!

- Thursday, 4th December
- Late days allowed!
  - … but aim to be done by the deadline
- Reminder: we will test your proxy manually
  - http://csapp.cs.cmu.edu
  - http://www.cmu.edu
  - http://www.amazon.com
- We will read your code
  - Correctness issues (race conditions, robustness, etc)
  - 10 points for style: make them count! (write clean, well-documented, well-modularized code)
Final Exam Details
Final Exam Details

- **10am to 10pm, Mon Dec 8 to Thu Dec 11**
  - Sign up *now*, if you haven’t already
- **10 problems, nominal time is 2-3 hours, but you get 6 hours!**
- Will probably be a review session this coming Sunday; stay tuned!
- **Cumulative: Chapters 1-3, 6-12**
- **2 double-sided 8 1/2 x 11 sheets of notes**
  - No pre-worked problems
  - Can bring scratch paper too (will provide at exam room too)
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)
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)
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 \texttt{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**
  - \texttt{cmp S1, S2 \rightarrow S2 - S1}, test S1, S2 \rightarrow 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 \(T* a, a + i = a + i \cdot \text{sizeof}(T)\) address
  - given \(T* a, *a\) reads/writes \text{sizeof}(T)\) bytes
Assembly Concepts (Arrays/Structs/Unions)

- **Arrays:**
  - Contiguous array of bytes
  - $T A[n]$: allocate array of $n \times \text{sizeof}(T)$ bytes; $a[i]$ is at address $a + T \times i$
  - 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

<table>
<thead>
<tr>
<th>Jump Table Addresses</th>
<th>Instruction Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400598:</td>
<td>0x000000000400488</td>
</tr>
<tr>
<td>0x4005a8:</td>
<td>0x00000000040048b</td>
</tr>
<tr>
<td>0x4005b8:</td>
<td>0x00000000040049a</td>
</tr>
<tr>
<td>0x4005c8:</td>
<td>0x000000000400498</td>
</tr>
</tbody>
</table>
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); $t = m - (s + b)$ tag bits
  - $s$ sets bits ($S = 2^s$ = no. sets); $E$: no lines per set;
  - $b$ block-offset bits ($B = 2^b$ = line size in bytes); 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 (e.g., 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 syccalls 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/sigsuspended)
- Note when a process can receive a signal; consider what happens when signal handler is invoked at different points
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
  - **Address-Ordering:** free-blocks are connected by ascending addresses
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
  - `pthreadDetach`: 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 (e.g., 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

Studying for finals

When did you become an expert in thermonuclear astrophysics?

Last night.

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, $E = e - \text{bias}$.

<table>
<thead>
<tr>
<th>Value</th>
<th>Format A Bits</th>
<th>Format B Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>One</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\frac{1}{2}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\frac{11}{8}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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;
    }
}

mystery_sort:
    jmp     loop1_check

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

loop2:
    mov     ($rdi, $rcx, 0), $rax
    cmp     $rax, ($rdi, $rdx, 0)
    jg      loop2_check
    mov     $rcx, $rdx

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

    dec     $rsi
    mov     ($rdi, $rsi, 0), $rax
    mov     ($rdi, $rdx, 0), $rcx
    mov     $rcx, ($rdi, $rsi, 0)
    mov     $rax, ($rdi, $rdx, 0)

loop1_check:
    test    $rsi, $rsi
    jnz     loop1

ret
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:  0x000000000004004d1  0x000000000004004c8
0x400610:  0x000000000004004c8  0x000000000004004be
0x400620:  0x000000000004004c1  0x000000000004004d7
0x400630:  0x000000000004004c8  0x000000000004004be
```

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
```

```c
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>:
    push    %ebp
    mov     %esp,%ebp
    sub     $0xc,%esp
    mov     0x8(%ebp),%ecx
    mov     $0x1,%eax
    test    %ecx,%ecx
    je      de <doSomething+0x2f>
    mov     %ecx,%edx
    shr     $0x1f,%edx
    lea     (%ecx,%edx,1),%edx
    sar     %edx
    mov     0x10(%ebp),%eax
    mov     %eax,0x8(%esp)
    mov     %ecx,0x4(%esp)
    mov     %edx,(%esp)
    call    da <doSomething+0x2b>
    leave
    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.

```c
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'.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the miss rate of this code sample?

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'.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the miss rate of this code sample?
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.

<table>
<thead>
<tr>
<th>Output Line Number</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st line of output</td>
<td></td>
</tr>
<tr>
<td>2nd line of output</td>
<td></td>
</tr>
<tr>
<td>3rd line of output</td>
<td></td>
</tr>
<tr>
<td>4th line of output</td>
<td></td>
</tr>
<tr>
<td>5th line of output</td>
<td></td>
</tr>
<tr>
<td>6th line of output</td>
<td></td>
</tr>
<tr>
<td>7th line of output</td>
<td></td>
</tr>
<tr>
<td>8th line of output</td>
<td></td>
</tr>
<tr>
<td>9th line of output</td>
<td></td>
</tr>
</tbody>
</table>

```c
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:

```c
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");
    }
}
```

<table>
<thead>
<tr>
<th>Snippet 1 Outcome</th>
<th>Possible? (Y/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing is printed.</td>
<td></td>
</tr>
<tr>
<td>&quot;a&quot; is printed.</td>
<td></td>
</tr>
<tr>
<td>&quot;b&quot; is printed.</td>
<td></td>
</tr>
<tr>
<td>&quot;ab&quot; is printed.</td>
<td></td>
</tr>
<tr>
<td>&quot;ba&quot; is printed.</td>
<td></td>
</tr>
<tr>
<td>A process does not terminate.</td>
<td></td>
</tr>
</tbody>
</table>

Code Snippet 2:

```c
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");
    }
}
```

<table>
<thead>
<tr>
<th>Snippet 2 Outcome</th>
<th>Possible? (Y/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing is printed.</td>
<td></td>
</tr>
<tr>
<td>&quot;ba&quot; is printed.</td>
<td></td>
</tr>
<tr>
<td>&quot;abc&quot; is printed.</td>
<td></td>
</tr>
<tr>
<td>&quot;bac&quot; is printed.</td>
<td></td>
</tr>
<tr>
<td>&quot;bca&quot; is printed.</td>
<td></td>
</tr>
<tr>
<td>A process does not terminate.</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>02</td>
</tr>
<tr>
<td>03</td>
</tr>
<tr>
<td>04</td>
</tr>
<tr>
<td>05</td>
</tr>
<tr>
<td>06</td>
</tr>
<tr>
<td>07</td>
</tr>
<tr>
<td>08</td>
</tr>
<tr>
<td>09</td>
</tr>
<tr>
<td>0A</td>
</tr>
<tr>
<td>0B</td>
</tr>
<tr>
<td>0C</td>
</tr>
<tr>
<td>0D</td>
</tr>
<tr>
<td>0E</td>
</tr>
<tr>
<td>0F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

(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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>0x</td>
<td>TLB Hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>TLB Index</td>
<td>0x</td>
<td>Page Fault? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>TLB Tag</td>
<td>0x</td>
<td>Physical Address</td>
<td>0x</td>
</tr>
</tbody>
</table>

**Virtual address: 0x14213**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>0x</td>
<td>TLB Hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>TLB Index</td>
<td>0x</td>
<td>Page Fault? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>TLB Tag</td>
<td>0x</td>
<td>Physical Address</td>
<td>0x</td>
</tr>
</tbody>
</table>
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?

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

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

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

```c
add_item(item);
/* 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?

```c
struct big_number *ts_next_prime(struct big_number current_prime) {
    static struct big_number next;

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

    return &next;
}
```

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

```c
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.

C. Fill in the blanks below to fix ts_next_prime.

```c
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;
}
```
Answers

Apologies for my handwriting. :(  

![Image](DIYLOL.COM)
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.

\[ A : \frac{11}{8} = \left(1 + \frac{3}{8}\right) \cdot 2^{3-3} \]
\[ \text{frac} = \frac{3}{8} \rightarrow 011 \rightarrow 10 \text{ (round even)} \]

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.

\[ B : \frac{11}{8} = \left(1 + \frac{3}{8}\right) \cdot 2^{1-1} \]
\[ \text{frac} = \frac{3}{8} \rightarrow 011 \rightarrow 01011 \]

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, $E = e - \text{bias}$.

<table>
<thead>
<tr>
<th>Value</th>
<th>Format A Bits</th>
<th>Format B Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0 000 00</td>
<td>0 000 00</td>
</tr>
<tr>
<td>One</td>
<td>0 011 00</td>
<td>0 0100</td>
</tr>
<tr>
<td>1/2</td>
<td>0 00100</td>
<td>0 00100</td>
</tr>
<tr>
<td>1/4</td>
<td>0 01000</td>
<td>0 01000</td>
</tr>
<tr>
<td>1/8</td>
<td>0 00110</td>
<td>0 00110</td>
</tr>
</tbody>
</table>

- \( A : 2^{-3} \times (1+0)\) Denorm.
- \( B : 2^{0} \times (0) \times 000 \) Denorm.

- \( \frac{1}{2} \):
  - Format A Bias: 0 000 00
  - Format B Bias: 0 000 00

- \( A : \text{Norm.} , 2^{-3} \times (100) \):
  - Exp: 2^{-3}
  - Frac: 0, 0 010 00

- \( B : \text{Denorm.}, 4 \cdot \frac{1}{8} : 0 00 100 \)
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:

```
0x400600: 0x00000000000004004d1 0x00000000000004004dc8
0x400610: 0x00000000000004004c8 0x00000000000004004be
0x400620: 0x00000000000004004c1 0x00000000000004004d7
0x400630: 0x00000000000004004c8 0x00000000000004004be
```

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

```
# on entry: %rdi = x, %rsi = y, %rdx = z
0x4004b0:   cmp  $0x7,%edx    z >= 7?
0x4004b3:   ja  0x4004c8
0x4004b5:   mov  %edx,%eax
0x4004b7:   jmpq *0x400600(,%rax,8)
0x4004be:   retq
0x4004c0:   mov  $0x3,%eax result = 3
0x4004c1:   jmp  0x4004da
0x4004c8:   mov  %esi,%eax
0x4004ca:   nopw  0x(%rax,%rax,1)
0x4004d0:   retq
0x4004d1:   mov  %edi,%eax result = x & 0x25.
0x4004d3:   add  $0x19,%eax
0x4004d6:   retq
0x4004d7:   lea  (%rdi,%rdi,1),%eax
0x4004da:   add  %esi,%eax result += 2 * y.
0x4004dc:   retq
```

```
int test(int x, int y, int z)
{
    int result = 3;
    switch(z)
    {
        case 0:
            result = x & 0x25
            break;
        case 3:
        case 7:
            result = x
            break;
        case 5:
            result = 2 * x;
            break;
        case 4:
            result = result++;
            break;
        default:
            result = y;
    }
    return result;
}
Stacks

```c
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 0x80.

```
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!)

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’

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 M</td>
<td>0 M</td>
<td>0 M</td>
<td>1 M</td>
<td>2 M</td>
<td>3 M</td>
<td>4 M</td>
<td>5 M</td>
</tr>
</tbody>
</table>

What is the miss rate of this code sample?

Miss rate: [ ]

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’

What is the miss rate of this code sample?

Miss rate: [ ]

- Cache line: stores 4 elements
- Cache line: stores 5 elements
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.

<table>
<thead>
<tr>
<th>Output Line Number</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st line of output</td>
<td>u</td>
</tr>
<tr>
<td>2nd line of output</td>
<td>s</td>
</tr>
<tr>
<td>3rd line of output</td>
<td>r</td>
</tr>
<tr>
<td>4th line of output</td>
<td>u</td>
</tr>
<tr>
<td>5th line of output</td>
<td>c</td>
</tr>
<tr>
<td>6th line of output</td>
<td>e</td>
</tr>
<tr>
<td>7th line of output</td>
<td></td>
</tr>
<tr>
<td>8th line of output</td>
<td></td>
</tr>
<tr>
<td>9th line of output</td>
<td></td>
</tr>
</tbody>
</table>

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

```c
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);
    printf("2 = %c\n", c);
    printf("3 = %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);
        printf("Move file2 pointer to file1\n");
        read(file1, &c, 1);
        printf("5 = %c\n", c);
        read(file2, &c, 1);
        printf("6 = %c\n", c);
    }
    return 0;
}
```
Signals

Code Snippet 1:
```c
int main() {
    int pid = fork();
    if (pid > 0) {
        kill(pid, SIGKILL);
        printf("a\n");
    } else {
        /* getppid() returns the pid of the parent process */
        kill(getppid(), SIGKILL);
        printf("b\n");
    }
}
```

- Parent:
  - Could receive either sigkill at any time
  - One of the two threads that does not die
- Child:
  - Will be killed after child
  - Will not be able to print the strings

<table>
<thead>
<tr>
<th>Snippet 1 Outcome</th>
<th>Possible? (Y/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing is printed.</td>
<td>Y</td>
</tr>
<tr>
<td>&quot;a&quot; is printed.</td>
<td>Y</td>
</tr>
<tr>
<td>&quot;b&quot; is printed.</td>
<td>Y</td>
</tr>
<tr>
<td>&quot;ab&quot; is printed.</td>
<td>Y</td>
</tr>
<tr>
<td>&quot;ba&quot; is printed.</td>
<td>Y</td>
</tr>
<tr>
<td>A process does not terminate.</td>
<td>Y</td>
</tr>
</tbody>
</table>

Code Snippet 2:
```c
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\n");
    } else {
        printf("b\n");
        kill(pid, SIGCONT);
        printf("c\n");
    }
}
```

<table>
<thead>
<tr>
<th>Snippet 2 Outcome</th>
<th>Possible? (Y/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing is printed.</td>
<td>N</td>
</tr>
<tr>
<td>&quot;ba&quot; is printed.</td>
<td>N</td>
</tr>
<tr>
<td>&quot;abc&quot; is printed.</td>
<td>Y</td>
</tr>
<tr>
<td>&quot;bca&quot; is printed.</td>
<td>Y</td>
</tr>
<tr>
<td>A process does not terminate.</td>
<td>Y</td>
</tr>
</tbody>
</table>

- Child misses wakeups
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.

```
<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>05</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>3F</td>
<td></td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0F</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1F</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1F</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>2B</td>
<td>1</td>
</tr>
<tr>
<td>1D</td>
<td>23</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>0B</td>
<td>0</td>
</tr>
<tr>
<td>04</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>06</td>
<td>0F</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>08</td>
<td>1C</td>
<td>0</td>
</tr>
<tr>
<td>09</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>0B</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>0C</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>0E</td>
<td>0C</td>
<td>0</td>
</tr>
<tr>
<td>0F</td>
<td>2B</td>
<td>1</td>
</tr>
</tbody>
</table>
```

(a) How many bits are needed to represent the virtual address space? \(2^{20}\)
(b) How many bits are needed to represent the physical address space? \(2^{18}\)
(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.

**Virtual address:** 0xF213

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>0x1F</td>
</tr>
<tr>
<td>TLB Hit?</td>
<td>Y</td>
</tr>
<tr>
<td>Page Fault?</td>
<td>N</td>
</tr>
<tr>
<td>TLB Tag</td>
<td>0x07</td>
</tr>
<tr>
<td>Physical Address</td>
<td>0x15213</td>
</tr>
</tbody>
</table>

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

**Virtual address:** 0x14213

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>0x14</td>
</tr>
<tr>
<td>TLB Hit?</td>
<td>Y</td>
</tr>
<tr>
<td>Page Fault?</td>
<td>N</td>
</tr>
<tr>
<td>TLB Tag</td>
<td>0x05</td>
</tr>
<tr>
<td>Physical Address</td>
<td>0x13213</td>
</tr>
</tbody>
</table>
Thread Safety

A. Why is the function ts_next_prime thread unsafe?

```c
struct big_number *ts_next_prime(struct big_number current_prime) {
    struct big_number *next_prime = current_prime;
    static struct big_number next;
    int isNotPrime = 1;
    while (isNotPrime) {
        next_prime = addOne(next_prime);
    }
    return next_prime;
}
```

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

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

C. Fill in the blanks below to fix ts_next_prime.

```c
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;
}
```

- References static data, returns ptr to that block.
- value_ptr is different address for each invocation.
- Thread 1: sem_wait.
  - next_prime
  - sem_post
  - value_ptr
  - return value_ptr
- Thread 2: sem_wait.
  - next_prime
  - sem_post
  - sem_wait
  - next_prime
  - sem_post
  - value_ptr
  - return value_ptr

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

\[
\begin{align*}
\text{mutex} &= 1 \\
\text{items} &= 0 \\
\text{slots} &= 10
\end{align*}
\]

See producers—consumers solution from textbook.

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

```c
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!