This exam has six parts, each of which contains five short questions. All questions are equally weighted at 10 points each (300 points total). Please write a short answer to each question. Each question has a full-credit answer that consists of a few key words or phrases drawn from the “lingo” of this class, or at most a sentence or two [shown in bold in this answer set]. Drawings help. But please keep your answers short and focused and within the box provided for each question: I will be grading thousands of those boxes.

These questions all apply to the idealized/simplified view of the systems presented in class. There are lots of variations across systems, e.g., from one version or “flavor” (Unix, Android) to another. Feel free to add some of those details, but it is not required. Some of the questions are specific to one or more of the systems we talked about.

For some of the questions, I am interested in crucial variations or alternatives that exist within a given system, e.g., different scenarios in which the question could apply. Please note those where you see them.

Understanding the questions. This whole course is about the structure of systems with multiple elements (e.g., machine, kernel, process, component, procedure, thread, device/driver). The elements store state in various ways and interact or communicate in various ways (e.g., files, messages, pipes, sockets, stack, heap, global variables, registers).

• When I ask “how” an element “knows” something or “learns” something, I am asking what element specifies or tells it, and in what form (e.g., a message, a file, passing it in memory or in a register) and under what conditions (e.g., on request, on a fault, on commit, when called).

• When I ask “what” the something is that the element knows or learns, I am asking about the content of the information (e.g., the faulting address, the process ID, the name of the file).

• When I ask “why” a particular feature or property exists, I am asking what purpose it serves: what system goal or property is met or supported by that design choice? What are the tradeoffs of that design choice? When I ask “why” some condition might occur, I am asking you to describe the scenario(s) in which it occurs.

If you don’t know the answer to a question, then just say something else relevant for partial credit.

Please keep your answers short!
Part 1. Control

As a thread executes on a core, various instructions and handlers may change the virtual address in the PC (IP) register to transfer control to some other piece of software. For each example below, indicate how the new target PC address is determined or obtained, i.e., how the thread “knows” the instruction address to transfer control to.

(a) Return from procedure

Retrieved from stack frame or register
Stored there by instruction sequence for procedure call

(b) First context switch into a newly created thread

Stored in the thread register context in memory (TCB) by thread initialization code: set to starting procedure entry point for thread (e.g., main()). Context switch code retrieves register values from the saved TCB.

(c) Context switch into a thread that previously switched out

Retrieved from thread register context saved in memory (TCB). Code for context switch saves register values of outgoing thread into its TCB, and loads registers from of the TCB of the incoming thread.

(d) Return from fault

Fault PC address saved in fault context object on fault (e.g., on thread kernel stack). Fault handler may modify the context, e.g., to retry the faulting instruction, or redirect control to some other selected PC (e.g., for signal delivery), or it might not return at all (if kill/exit).

(e) Return from system call (trap)

Trap PC address saved in trap context object on trap (e.g., on thread kernel stack). Trap handler code may modify the context, e.g., to redirect control some other selected PC (e.g., for exec or signal delivery), or it might not return at all (if exit).
### Part 2. The machine

An operating system kernel controls a machine, and relies on machine functions that support protected kernels. These questions pertain to the interaction of the machine and the operating system software.

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) LRU replacement generally makes good eviction choices for caching, but operating systems don't use LRU for virtual memory page caching. Why not?</td>
<td>Most memory references are handled by the hardware (MMU/TLB), and the OS never sees them. The OS sees only a subset of references. It can control the sampling (e.g., by arranging faults or setting/clearing reference bits) to approximate LRU, but full LRU would be far too costly.</td>
</tr>
<tr>
<td>(b) How does the kernel “know” when to free the memory it uses for page tables?</td>
<td>Address space teardown, e.g., process exit. (Also, page tables pages may themselves be page out of memory on most systems.)</td>
</tr>
<tr>
<td>(c) Why are system call stubs useful?</td>
<td>Stubs contain assembly language instructions to invoke specific system calls, using special trap instructions and register transfers, according to the system’s Application Binary Interface (ABI). Stubs hide these details from application programs, and allow those programs to access system calls from high-level programming languages. Note: stubs are <em>not</em> trusted code.</td>
</tr>
<tr>
<td>(d) Why might a page fault occur on a page that is already resident in memory?</td>
<td>It could be a protection fault, e.g., a program error or a choice by the OS to disable protection on a valid page in order to receive a fault and know that the page was referenced (see (a)). More commonly, a shared page was brought into memory by some other process. A common example is page sharing between the parent and child after a fork().</td>
</tr>
<tr>
<td>(e) What happens if the machine raises a fault on a core that is already executing a fault handler?</td>
<td>The core is already in kernel mode, so this means that the kernel code incurred a fault. The new fault handler is entered immediately with a stack frame pushed on the same kernel stack. The new fault handler may decide to crash the system, or it might resolve the fault and resume the previous handler (e.g., if it is a page fault on some pageable part of the kernel).</td>
</tr>
</tbody>
</table>
Part 3. Threads

These questions pertain to threads: how the kernel supports them, how they use the machine, and how they interact.

(a) Why is it important not to write code that blocks the UI thread? When the UI thread blocks, what does it wait for?

UI is User Interface. On Android and other OS the UI thread is the main thread. Normally it blocks only when idle to wait for a new event, including a UI event, but also (in Android) incoming intents. If the UI thread blocks when it is not idle, then the app’s UI “freezes” until the UI thread wakes up.

(b) Suppose thread T1 stores value V to location X. Then T2 loads from X on another core. The machine might not return value V to T2’s load unless T2 acquires a lock from T1 after T1’s store and before T2’s load. How does the machine “know” if T2 acquired a lock?

Any lock acquire executes an atomic instruction “under the hood”, such as test-and-set-lock or compare-and-swap. These instructions drive propagation of memory updates among caches that are attached to different cores.

(c) Why might an interrupt handler wake up a thread?

I/O complete or alarm based on passage of time marked by a clock tick interrupt. Note that a clock interrupt does not wake up the next ready thread: a ready thread is already awake.

(d) Why is recursion dangerous with threads?

Each thread has its own stack. When there are multiple thread stacks within a virtual address space, each stack has a bounded size. The risk of stack overflow is higher when recursion is used.

(e) How does the kernel “know” when to reuse the memory it consumes for thread stacks?

Thread exit
Part 4. Services

These questions pertain to network services. Consider an elastic service that is running in the cloud, with multiple servers. If the question requires it you may make simplifying assumptions, like: each request takes mean service demand \(D\) on a single server, and any state needed to serve a request is replicated “for free” on all of the servers.

(a) When a client uses `connect` to open a socket to a server, how does it specify which server it wants to talk to? How does the client name the server?

**IP address and port number.** The IP address could be obtained by a DNS lookup on a DNS name (e.g., www.cs.duke.edu).

(b) Often a server returns a certificate to the client after a connection is established. Why?

**To authenticate the server** if the connection is an SSL/TLS connection. The certificate is a statement asserting the server’s DNS name and a public key. The certificate is digitally signed by a Certifying Authority. If the client knows and trusts the CA, then it believes that the server’s public key is valid for the DNS name: it believes that the server is legitimate for that DNS name.

(c) If a service has multiple servers, how does it balance the request load across them? What would go wrong if the load is not balanced?

A front-end or **dispatcher node** intercepts requests and forwards them to selected back-end servers, or use DNS Round Robin. Select the most lightly loaded back-end server, or choose randomly, or by round robin, or based on specific data requested. If the load is not balanced then there is a bottleneck and some response times go up.

(d) How does one determine the “right” number of servers needed to meet a given response time target (say, mean response time of \(R\))? 

Use the **inverse idle time law** to identify the maximum arrival rate corresponding to the mean response time ceiling. Then **divide** the total current arrival rate by the maximum per-node arrival rate to get the number of nodes (round up).

(e) Why do Web servers return Web pages with an expiration time? Why do DNS servers return DNS entries with an expiration time? (The expiration time is also called a TTL.)

Web pages and DNS responses may be cached by the receiver, or by an intermediate cache, such as a Web proxy or client DNS server. The caching sites use the expiration time to **detect stale cache entries.** An expired entry is discarded and the data is fetched again on the next request.
### Part 5. Cryptosystems and certificates

These questions pertain to basic crypto techniques.

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Digitally signed messages (or documents) are “tamper-proof”. How does the receiver detect if an attacker has modified a signed message while it was in transit?</td>
<td>Decrypt the signature to obtain the sender’s hash of the message. Then, <strong>run the hash function over the received message</strong>. If the message was modified then the <strong>result does not match the sender's hash in the signature</strong>. Since the sender’s hash is encrypted in the signature, no attacker can forge it without the sender’s private key.</td>
</tr>
<tr>
<td>(b) Why does a certificate contain an expiration time?</td>
<td>The certificate attests that a given named principal uses a keypair with a given public key. The longer a keypair is used, the more likely the <strong>keypair may be compromised by theft or crack</strong>. The expiration time <strong>bounds the time that a client trusts the keypair</strong>. After it expires, the CA must re-endorse the keypair with a fresh certificate.</td>
</tr>
<tr>
<td>(c) Why does a certificate contain a hash? What is the data that is hashed?</td>
<td>A certificate is a digitally signed document, i.e., a “message” of 5(a) above. It is signed so that the receiver can validate that it was issued by a particular Certifying Authority. As with any signed message, the <strong>hash covers the contents of the message</strong>. The contents of a cert includes expiration time, subject’s public key, and subject’s name, such as a DNS name.</td>
</tr>
<tr>
<td>(d) How does an SSL server “know” who its client is, i.e., how does it authenticate a client?</td>
<td>SSL/TLS supports exchange of certs in both directions, but most clients in the Web don’t have certs. Once an SSL connection is established, the client can authenticate by <strong>logging in with a username and password</strong>, or log in via a third-party Single-Sign On (SSO) provider. Note that the client must provide an actual password, and not just a password hash.</td>
</tr>
<tr>
<td>(e) Why does the length of cryptographic keys matter? What are the tradeoffs in choosing the key length to use?</td>
<td>Everyone got this. <strong>Longer keys are harder to crack, but more expensive to store and use</strong>. Always be sure that you choose your key lengths with intention: thoughtfully, not thoughtlessly. “Computationally infeasible” might not mean what you think it means: keys that were “long enough” in 2004 are vulnerable in 2014.</td>
</tr>
</tbody>
</table>
Part 6. File/storage systems

These questions pertain to the nexus of file systems and virtual memory.

(a) Why are file block maps skewed?

Some answers described inode maps and said (in essence) they are skewed because they are. The question is **why**: what **purpose** does it serve? **Small files are cheap and large files are possible.** Small files are cheap: no indirect blocks, just a few block pointers in the inode. Extending the map with indirect blocks allows very large files at some cost. But most files are small.

(b) Sometimes a read system call on a file returns zero-filled data rather than reading the data from disk. How does the system “know” to return zero-filled data for a fetched block?

This is called a “hole”: a portion of the file that was never written to. A logical block L is a hole if the offset of the first byte of L is less than the file’s size, and the value of the block map entry for the logical block L is zero. The value for a block map entry for L is zero if the entry exists and contains a zero, or if the map entry for an indirect block that covers L exists and contains a zero.

(c) Sometimes a page fault handler installs a zero-filled page rather than reading the page from disk. How does the system “know” to return zero-filled data for a missing page?

Answer: the virtual page P is a valid page in the address space, and has never been written to. Generally P is a page of the stack segment, heap segment, or uninitialized global data (BSS) segment that has never been written to. These segments are called **anonymous**: they are not bound to a named file. Else return a zero-filled page if the corresponding block of the file is a hole.

(d) Why does mirroring (RAID-1) improve throughput for random block reads but not for writes?

Every logical block of the volume is stored on every disk: **any read can be served by any disk, but a write must be applied at all disks.** Random reads may be evenly distributed across the disks in the array, but a stream of writes executes no faster on the array than on a single disk.

(e) Why does striping-with-parity (RAID-5) have lower throughput for random block writes than pure striping (RAID-0) does?

In RAID-0 and RAID-5 the logical blocks of the volume are striped across the N disks. **With striping, random writes distribute evenly across the array** (throughput increase by factor of N). But in RAID-5, each random block write (sub-stripe) must also modify the stripe’s parity block on a different disk. So a logical block write requires two writes: throughput increase N/2.
Extra credit

What is a virtual address space? I am looking for an explanation that is better than the one I gave in class. Feel free to illustrate.

- Sandbox / containment
- Lockbox / isolation
- Uniform name space: gives process the illusion of access to the machine
- A “window” on files and anonymous segments used by the program
- Level of indirection: allows OS to manage and vary the amount of memory allocated to each process
- A hardware abstraction that determines how the hardware translates addresses issued by executing instruction streams (threads). The OS loads a virtual address space and/or its map into a core register context, in a protected register: any instructions executing on that core are understood and translated within that address space.

As always what I “really wanted” was a cartoon that conveyed insight into the concept and also made me laugh. Perhaps it is an imposition to coerce you into producing creative artwork for me before you can walk out the door free as the summer heat, but the good ones are so wonderful that I keep doing it.

Have a great summer!