CPS 210 Midterm Exam
Spring 2010

This is a wordy exam asking you to explain things rather than to work out problems. Answers are graded on content and substance, not style. Please do not waste any words. I am looking for understanding and insights, rather than details. Please give the most complete answer that you can in the space provided for each question. You may attach additional pages if necessary. Watch the time: you have 75 minutes. This is a closed-everything exam, except for your one page (two sides) of notes. The five questions are each worth 40 points: 200 points total.

This is basically a test of whether you can write for an hour about systems without garbling anything. In grading an exam like this I allow a pretty wide range of levels of detail in the responses. For each question I am looking for a few specific points. Any answer that hits those points gets full credit, even if it does little more than say the right keywords. But any detail that is related to the question gets credit, or at least can help me to be charitable with points if something else I was looking for was left out. In general: a blank answer gets zero points, an answer that is garbled in some way gets half credit or left, an answer that is vague but not obviously incorrect gets about half credit, a ramble with a few gaps gets about 75%, a good answer that leaves out some key points gets 90%.

The high score for this exam was a perfect 200. I graded 18 exams. The low was 135. Five students scored 175 or higher, and six more scored 160-175.

Below I summarize the key points I was looking for in each question.

Short Answers

Q1. Answer the following questions in the space provided. Since there isn't much space, you will have to gloss over some details, but that is OK.

(a) Data cached at multiple clients can become stale if it is updated. Many distributed systems have taken a position that stale data is acceptable if it is not stale for "too long" (e.g., HTTP, DNS, and early NFS). Outline a technique to control staleness in such systems.

Keyword: TTL or expiration date, freshness date. Server responses include a time-to-live, and the receiver timestamps them before entering them in the cache. A cache access checks the timestamp and invalidates the entry if it is older than its time-to-live. Variants of this scheme are used in HTTP, DNS, and NFS.

(b) What support is needed from the operating system *kernel* to implement heap allocation primitives? (E.g., support for *malloc* and *free*, or *new* and *delete*).

Keyword: sbrk. Heap allocation is implemented in a runtime library as part of the programming environment. It involves management by a process of the virtual memory assigned to it, so it does not need any specific support from the kernel. Different processes can run different heap managers internally. However, the kernel must provide a system call for the heap manager to acquire blocks of virtual memory to use for its heap. In classic Unix this system call is sbrk, or "set break", i.e., specify the end of the heap region.

Short Answers on File Systems

Q2. Answer the following questions in the space provided. Since there isn't much space, you will have to gloss over the details, but that is OK.

(a) In the Unix file system, data written to a file (i.e., in write system calls that complete with success) may be lost after recovery from a system crash. However, the original Network File System (NFS) requires a file server to commit each write before returning success. Why the discrepancy, i.e., why are weak assurances "good enough" in one case but not the other?

Keyword: fate sharing. If the system crashes, any process running on that system crashes along with it. Any writes the process would have written after the crash are lost, because the crash prevents the process from even issuing those writes. It is not "really" any worse if the system also discards writes written just before the crash, since the crash could have occurred earlier, and the process cannot know exactly when it occurred. Classic Unix systems assure a process that writes to a file are committed before the file is closed, and they also provide an fsync system call to force any uncommitted writes to commit. Uncommitted writes may be lost in a crash.

But in a network file system, the server may fail independently of the client. That is, the NFS server may fail while the process continues to run. It is necessary that the failure (and eventual recovery) of an NFS server does not cause the process to execute incorrectly, e.g., by corrupting data. Early versions of NFS required writes to complete synchronously at the server before returning from the write RPC call. In class we discussed extensions in NFSv3 to allow writes to execute asynchronously for better performance. They still ensure that a server crash does not lose any writes, unless the client also crashes (fate sharing again).

(b) A read system call may cause a process to block (wait or sleep) in the kernel. Why? In each case, what causes the process to wake up?

Keywords: lock, I/O, interrupt. It may block to acquire a lock to access an internal kernel data structure such as an inode or a buffer. In that case, some other process that holds the lock wakes it up by releasing the lock. It may block to wait for I/O to complete, e.g., a block read to fetch file contents from disk. In that case the incoming interrupt handler from the device, or some other code that processes the completion event, will awaken the waiting process (e.g., by signaling the event on a semaphore or condition variable).

(c) Why does Unix designate specific file descriptors as "standard" for standard input (*stdin*), standard output (*stdout*), and standard error (*stderr*)?

Keywords: pipes, composing programs, uniform I/O. In general, a Unix program does not need to know or care where its input comes from or where its output goes. The same program can be used as part of a pipeline, or to interact directly with the user, or to talk over a network, or to operate on files, without any changes to the program. The parent process (e.g., the shell) controls how the mapping of descriptors to the underlying I/O objects (pipes, files, sockets, tty). This feature of Unix has supported powerful scripting features to build complex functionality from collections of simple programs that can be combined in various ways.

Short Answers on Cryptosystems

Q3. Answer the following questions in the space provided. Since there isn't much space, you will have to gloss over the details, but that is OK.

(a) Are digital signatures on digital documents more or less secure than ink signatures on printed documents? Justify your answer.

Keyword: secure hash. A digital signature is bound to a secure hash value computed over the document contents. Any change to the document will change the hash value, and this allows the receiver to determine that the signature is invalid. An attacker cannot modify the signature for the new hash value unless it knows the signer's public key. In contrast, a signature on a printed document contains no information about the document. My signature looks almost the same on any document. If the document can be modified in a way that is not obvious to a reader, then the reader may still consider the signature to be valid.

(b) Certificates enable the bearer to authenticate to another entity without the involvement of a third party (e.g., a certifying authority) at the time of authentication. How does this work?

The certificate is a digitally signed document from the third party endorsing the bearer's public key. Any receiver can determine that the certificate is valid just by examining it, assuming the receiver knows the public key of the third party that issued the certificate. It does not need to contact the third party to validate the certificate. If the certificate is valid, the bearer can then authenticate to the receiver by proving that it holds the private key corresponding to the public key endorsed in the certificate. It can prove this by successfully using its private key to encrypt or decrypt content known to the receiver.

For example, the receiver might issue a challenge with a random number (nonce), and ask the bearer to encrypt the nonce and return it. The receiver can then decrypt the response with the public key in the certificate: if the nonce matches, then it knows that the sender is the entity named in the certificate, i.e., it possesses the corresponding private key.

(c) In secure DNS (DNSSEC), each zone server holds an asymmetric keypair. Why use an asymmetric cryptosystem rather than symmetric crypto, which is less costly?

Keyword: key distribution problem. In DNSSEC, each zone server is endorsed as authoritative by its parent in the DNS hierarchy. The parent does this by issuing a digitally signed endorsement of the zone server's key. With symmetric crypto, it is not possible for the parent or any third party to endorse the key without revealing it. Once the key is revealed, an attacker to masquerade as the authoritative zone server. This would defeat the purpose of DNSSEC.

With asymmetric crypto the parent can endorse the zone server's public key without revealing the private key. Asymmetric crypto is slow, but it enables secure third-party endorsements, which are the cornerstone of DNSSEC.

More File Systems and WAFL

Q4. To handle reads or writes on files, file systems must identify the physical disk locations where logical file blocks reside on disk. This involves lookups through layers of naming structures. WAFL uses similar data structures to classical Unix file systems. (Let's ignore WAFL's structures for aggregates and logical volumes or flexvols.) But WAFL makes some changes to support more flexible placement of blocks on disk.

(a) Outline the steps to map a logical block to a physical block in WAFL. You do not need to describe the data structures themselves in detail, just the sequence of mapping steps.

Keyword: ifile. WAFL interprets an inode number as offset into an inode file or ifile. The blocks of this file may reside anywhere on disk, just like any other file. WAFL finds the ifile inode from the root node, obtains the ifile block map from the ifile inode, maps the inode number to a physical block through the ifile block map, retrieves the file's inode, obtains the file's block map from the file inode, and maps the logical block number to a physical block number through the file's block map. Mapping the logical block number of a file may involve one or more layers of indirect blocks.

(b) What advantages does WAFL's flexible block placement offer? Consider the handling of both reads and writes.

Keywords: clustering, snapshots, cloning, locality, no-overwrite. "File system design is 99% block placement." If any block can be placed anywhere, many heuristics are possible to optimize the placement of blocks for performance. Blocks written together can be blasted to disk in large chunks, minimizing seek overhead. Blocks read together can be placed in adjacent locations and read in batch. Perhaps more importantly, a block can be written in a different location from its last version, without overwriting the last version. This property enables fast cloning and consistent snapshots of data objects. It also has certain atomicity benefits: a new version of a data object can be written to disk in its entirety and then committed with a single atomic disk write to the inode at its root.

(c) Outline the implementation choices that WAFL made to enable this flexible block placement, in terms of differences from the Unix file system implementation.

Keyword: floating inodes. Only the root inode is fixed. This requires some extra indirection on lookups, to store the inodes in a file and manage a dynamic mapping of those inodes to the disk. Once that is done, it becomes possible to relocate anything that the inodes point to: block maps, files, etc., and change the pointers to reference the new copy. That may require more updates to block maps or other metadata to update the points, but it is easy for WAFL to batch those writes.

In the classic Unix file system all of the inodes reside at fixed locations on disk. The locations are obtained from some offset arithmetic on the inode number.

Services and Binding

Q5. Any client/server system (e.g., an RPC system) must define some method to establish a binding between the client and a server. How does the client determine who and where the server is (e.g., its IP address, port number)? How does the client verify that it is talking to the correct server? How does the server authenticate the client and determine if the client is authorized to invoke the service or operate on specific objects? Summarize and contrast the approaches to binding and authorization for three systems we have discussed: NFS, DNS, and Web/HTTP. I am asking about "plain vanilla" versions of these systems as discussed in class (e.g., plain NFSv3, DNS and not DNSSEC, HTTP including HTTPS).

This was the most difficult problem to grade. The question covers a lot of ground and people were short on time. Most answers were of the form "the user gives a DNS name to get the IP address and port number, and there might be some certificates for authentication", followed by some rambles.

I was looking for how/where the server name (DNS or IP) is provided and when the binding occurs. For NFS, the binding is provided statically at mount time. For DNS the binding is dynamic at lookup time: DNS has well-known roots that provide bindings for the authoritative servers for various domains, which in turn may provide bindings for subdomains in a hierarchical fashion. In the Web the server's name is encoded in a URL provided by the user or in a hyperlink. This seems obvious now, but it was a crucial innovation.

To authenticate the server, many systems rely on the security of IP addresses. This approach is vulnerable to various ways to spoof networks that we have not discussed in this class. Relying on the network is a reasonable strategy for early NFS, which was designed to run in a single administrative domain that controls its entire network. DNS relies on the network and is vulnerable to spoofing. These systems can be augmented with stronger authentication using certificates, as with HTTPS and DNSSEC.

How does the server authenticate and authorize the client? For classic NFS a server administrator exports volumes to specific sets of hosts (DNS names or IP addresses), and these hosts are trusted to represent the identity of the user correctly. User access is checked with access control lists on files and directories. DNS lookups are open to anyone. Web servers can choose to authenticate their clients using passwords or certificates.