Carnegie Mellon
Computer Science Department.

15-712 Fall 2007
Midterm 1

Name: _____

Andrew ID: _____

INSTRUCTIONS:

There are 13 pages (numbered at the bottom). Make sure you have all of them.

Please write your name on this cover and put your initials at the top of each page **except the last.**

If you find a question ambiguous, be sure to write down any assumptions you make.

It is better to partially answer a question than to not attempt it at all.

Be clear and concise.
Closed book. 80 minutes.
We will grade the first 10 questions answered. All questions have the same weight.

Do you see what I RPC

1. How do the semantics of Birrell's remote procedure call differ from local procedure call? (List at least two ways in which they differ.) What could you do to minimize these differences?

Solution: Call by value rather than call by reference. Don't pass elements of big structures. Enable call by reference by building a very fast, low interference remote read/write interface like RDMA.

Marshaling by type. With local procedure calls and call by reference, only the formal arguments need to be explicitly typed. But with call by value and complex structures, all of the types of variables in the structures must be known to the marshaling code. This is usually the responsibility of the programmer; using an interface definition language, all the types that must be marshaled are disclosed to the stub generating code.

Rendezvous differences. Local procedure call will always succeed to find the right code because it is linked into the application and is inherently waiting to execute (because local procedure call "sends a thread with the invocation", while RPC has to determine where the remote procedure is on a network and what port it should be listening to and how to get its attention, if it is not running or is doing something else.

Failure semantics. Local procedure caller and callee failure or survive together. Remote procedure callees and callers fail separately. Servers, the callee, usually try to be stateless, so caller failures don't matter to them, and in some cases provide reply caches to repeat a return result if the caller restarts and retries a call that is not idempotent. Callers often poll callees, usually by retrying the whole operation, unless the operation is idempotent, in which case they may try to poll for "alive and making progress" before invoking a recovery operation (which might be as severe as failing the caller to simulate the "fails together" semantics).

We expect at least two of these to get full marks.

2. Pendulous Threads

Deferring work by forking a worker thread is a classic strategy for improving a) responsiveness and b) throughput. Justify each of these improvements and describe a scenario where deferring work can cause problems due to excessive dependence on the operating system scheduler.

Solution: Improves responsiveness because server replies to caller before all work is done. Improves throughput by batching, coalescing, canceling deferred work when subsequent work enables such action. Also if the machine is a multiprocessor, multiple threads can be executed at the same time, which if the concurrency was not already high, will increase throughput by not wasting the other processors.

Deferred work needs to be scheduled at an appropriate time; leave it too long, perhaps by assigning it too low a priority, and resources backup; run it too soon and some of the throughput benefit is not achieved. If deferred work is not preempted by incoming new work, or execution of deferred work does not yield frequently enough, response time of subsequent operations can be increased, nullifying the response time benefit.

We are looking for at least one reason for each of responsiveness and throughput and at least one dependence on the scheduler.

Time, time, time, see what's become of my state machine

3. Lamport says that the most important contribution of the "Lamport Clocks" paper is that it provided the first framework for creating replicated state machines. Why does Lamport's ordering allow a set of independent processes to act as a replicated state machine? What must these processes do in order to act like a replicated state machine? Briefly explain why you might want to create such a replicated system, and give one example of a service to which this technique might apply.

Solution: The causal ordering provides a way for all of the replicas to observe a consistent order of operations. As a result, each can then independently apply the operations (in the right order), ensuring that the replicas are consistent.

An example of this is creating a robust file server. Write operations can be replicated to all replica nodes, and reads can be satisfied by any replicas. In a system that is read-dominated, this replication can provide very high performance while ensuring that the system is robust to the failure of one or more nodes. (Note that without a voting scheme, such as that provided by the byzantine fault tolerant systems, this scheme is not robust to malice or errors, only "fail-stop" failures.)

Eraser

4. Sketch the reasoning why a "happens before" analysis of a program might be used to detect data races and explain two limitations of this approach.

Solution: Happens before is a Lamport logical clocks analysis. It operates by observing/tracing all interactions between threads and shared variables where lock acquiring replaces messaging as the "connecting" operation; then analyzing these traces to detect shared variable accesses that are concurrent in the Lamport sense, th at is, there is no happens before ordering by lock free/acquire between them. This has at least two limitations: first, it only discovers races that actually happened, rather than races that might happen because no lock is held; and, second it is exceedingly expensive (in slowdown) to capture all the ordering and shared variable accesses from a running program.

Slipped Disk

5. The original RAID paper claimed that the failure probability of a RAID system was astronomically low. Explain two reasons that this analysis is not true in today's RAID systems.

Solution:

- Failure rates are roughly 3x higher in deployed systems than the manufacturer's specifications suggestion.
- Disk failures appear correlated. As a result, the chance of a second failure during reconstruction is much higher than the naive independent analysis suggests.
- Disks are much larger than they were in the original RAID paper, but disk transfer speeds have not increased as quickly. As a result, the time needed to reconstruct a RAID after a failed disk has gotten longer, increasing the window of vulnerability to a second failure.

As a result of this higher failure rate, some RAID implementors are turning to RAID 6 (two-dimensional parity). Explain the effect in read and write performance of moving from RAID 0 (pure striping, no redundancy) to RAID 4 (single, 1D parity disk) to RAID 6 (2D parity with fixed stripes).

Solution: RAID 4 (or 5) maintains high throughput relative to the number of disks for all accesses except small writes. Small writes still require a read+write on two disks, reducing the throughput to about 25% of the raw disk capacity.

For small writes, RAID 6's penalty is similar - it must read+write the data disk and two parity disks, reducing (in theory) maximum capacity to $\frac{1}{6}$ of what it could be. RAID 6 does, however, introduce overhead for large writes as well—while the large write will occupy all spindles in the row, the disks must still be read in order to update the column values. As a result, a large write requires a read+write of twice as many disks as it would in a RAID-5 system, which one might expect would reduce maximum system capacity to 25%.

There's a log-structured filesystem on the hole in the bottom of the disk

6. A log structured file system writes everything at the end of the log, implementing the log as a wrap-around data structure in a physical disk (or RAID). Even (and especially) without cleaning the disk, the process of recovering after a CPU power failure and restart depends on finding the end of the log. What could define the end of the log, how would it be found, and what can be lost during this recovery process?

Solution: The root of a traditional UNIX file system on disk is the superblock. It has pointers to freeblock bitmaps/structures and to the root directory's inode, which points to the data in the root directory, which point to the inodes of the children files and directories under the root directory, etc. In case a superblock is damaged, multiple are written in different parts of the disk.

In Rosenblum's log-structured filesystem, he replaces much of the superblock with another fixed location data structure, the checkpoint. This serves the same purpose, in that it is in multiple fixed locations (with each timestamped so the latest can always be identified) and it points to the latest addresses of the pieces of the inode table (where a particular inode is the root of the filesystem) and the ordered regions defining the log. A checkpoint is written when all data structures are consistent in the log; that is, all pending updates to files, directories, and inodes are written to disk.

But between checkpoints data and metadata changes are written to the log. In order not to lose these changes on powerfail, LFS processes the piece of the log beyond the checkpoint according to the list of regions in the log listed in the checkpoint. This "processing" is called roll-forward. First, the end of the log is always contained in one "region" of the disk, that region is always empty before the log is written into it. So the end of the log is in that region, covering the beginning to some point in the log. LFS always writes as much as possible in each time it triggers the disk to write at all. Each of the "writes" starts with a descriptor of the rest of the contents of the write. On roll-forward, provided that the last write before crashing was completed, the end of the log is found by reading the region linearly looking at the first blocks to find the map, computing the end of the disk write, then looking to see if there is another descriptor there (and repeat). In order to detect errors in a single disk write, LFS uses predictability in some of the structure in the log, like inodes, directories and descriptors. Hence cleaning a region needs to write zeros into the freshly empty region.

Full marks comes from recognizing the need for a fixed location point to the log and rules for reading forward through the log for a bounded distance (with quite a lot self-describing information allowing it to be easy to determine when a block is no longer part of a valid log; i.e., zeros).

NASD GFSes

7. Suggest a workload that is better suited to the Google File System than a NASD system, and vice versa. Explain your answer, and explain the parameters of the workloads that are most important in explaining why that workload is better or worse on each system.

Solution:

GFS is optimized for and contains special semantics for concurrent appends (e.g., for producer-consumer type problems), whereas a NASD client would need more coordination to avoid overwriting another client's write.

GFS writes replicas through a sequence of chunkservers. Hence, a NASD system would exhibit lower latency for a particular write.

You Down With RDMA?

8. Remote Direct Memory Access (RDMA) is a communications protocol enabling hardware support for simple read-memory and write-memory messaging. Explain how this could be especially beneficial for stock market trade servers that are not allowed to lose any completed transactions if the servers fail?

Solution: Stock market trading is highly concurrent and every transaction needs to happen quickly, in order that the price paid does not (much) change during the trade as perceived by the buyer and seller. And trades that complete cannot be lost. So during the trade, a record of it must be written to at least two different servers. That means that writing a log record into the memory of another server is a critical path in the latency of each transaction. RDMA allows this write to occur without interrupting the second server, which means no need to wait until the log write server can be scheduled to run. In fact the entire remote log write occurs at hardware speeds – a round trip message and a memory write from the second server's NIC. This will make trades faster and reduce the work that each server does as a secondary server, increasing throughput.

We are looking for the idea that the log write does not require attention from the second server's CPU.

Get Down, Make MapReduce

9. What kind of workloads does the MapReduce workload work well for? What kinds of workloads would perform poorly? (Explain your answer, and give a concrete example of each type of workload.)

Solution: MapReduce works well for "embarassingly parallel" workloads in which there is little, if any, sharing between computations on different chunks of data, and the chunks of data to be operated on are large. An example is processing logfiles to identify all web pages accessed by a particular client. Each line in the logfiles can be processed independently, the "map" step (presumably) reduces the data volume to a very small amount compared to the input, and multiple lines or entire logfiles can be assigned as the work units so that the data chunks are large.

In contrast, MapReduce is less suited to tightly coupled computations that require communication between all nodes analyzing data. Many scientific computations match this model, such as the earthquake finite mesh example from class, in which computation proceeds in timesteps and each node must communicate its state to its neighbors after each (relatively small) timestep.

Explain and justify two important performance optimizations in MapReduce. Why are these optimizations important (e.g., how do they save whatever it is they're optimizing, and why is it important to save that resource?).

Solution:

1. **Combiner functions**: The combiner function is simply the same as the "reduce" function, but it runs locally on each mapper machine before sending the data to the reduce machine. This optimization can substantially aggregate information before it is (expensively) sent over the network for some workloads, saving bandwidth and reducing load on the reducer.
2. **Backup tasks**: MapReduce re-issues tasks to other nodes if that task is taking a long time and delaying the end of computation. This optimization saves a lot of time by avoiding failed or slow machines at the small cost of potentially repeating work. In some of the example workloads, disabling backup tasks added 44% to the time required to complete a computation.
3. **Assigning map tasks to nodes with data present in GFS:** MapReduce attempts to have nodes operate on data that is already physically present on their disk, to reduce the amount of network bandwidth required. This optimization substantially reduces traffic across the network and means that machines don't have to act (as much) both as a data server and a data processor.

You SPIN me Right Round, Exokernel

10. The web server in the Exokernel paper takes advantage of operating system extensibility to achieve good performance. Name two ways in which the web server takes advantage of this extensibility to improve its performance.

Solution: Precomputed checksums. Rather than compute a TCP checksum for each packet sent, which may duplicate work if a particular web page is accessed several times, Cheetah (the web server in the Exokernel paper) can precompute checksums and store them along with data. (Modern network cards often have offload hardware that performs this task for the CPU.)

Zero copy. A request in a typical web server may be copied from disk to kernel memory then kernel memory to application memory (to read from the disk), and then from application back to kernel memory (to send to the network). Cheetah can read from the disk to the file cache and then send data straight from the file cache, without ever copying the data. (The "sendfile" system call serves a similar purpose.)

Packet merging. An incoming TCP packet generates an ACK response from the receiver. Rather than sending a packet just for this ACK, Cheetah can delay the ACK and send it with the response because it knows a response will be sent shortly. (TCP's delayed ACK serves a similar purpose.)

HTML-based file caching. Cheetah allocates related files on adjacent blocks on the disk when possible. This improves performance when the file cache does not hold the requested web page.

What is one advantage and one disadvantage of the Exokernel approach vs. the RDMA model argued for by Thekkath and the DAFS paper?

Solution: Both approaches lower CPU utilization by avoiding copying data. The Exokernel approach also allows further optimization, such as HTML-based file grouping and precomputation of file checksums. The Exokernel approach also does not change the client protocol. But, the Exokernel approach also requires using a nonstandard operating system, and unlike RDMA, the CPU must be involved in each request.

Don't worry, Be Optimistic

11. Kung presents an alternative to lock-based concurrency control. Under which workloads is optimistic concurrency control likely to perform best? Under which is it worst?

> **Solution:** A workload that has mostly reads or non-conflicting writes can take full advantage of optimistic concurrency control without needing to restart many transactions.
>
> In contrast, a workload that forces many restarts would perform poorly. In the extreme case, a workload with constant contention would force each transaction to remain in the critical section, which is similar to grabbing a single global lock and processing each transaction sequentially.

An optimistic concurrency system is exactly that—optimistic. Sometimes this optimism is unjustified. For what kind of operations is this optimism likely to be incorrect, and what happens to those operations? How does Kung suggest solving this problem?

> **Solution:** Workloads with many conflicts do not meet Kung and Robinson's optimistic assumptions. For example, if two transactions try to write the same object, both transactions may fail the validation phase and restart. This restarting may occur many times. To prevent transactions from continuously failing the validation phase, Kung and Robinson suggest that such transactions hold the semaphore for the critical section (essentially write-locking the entire database) after restarting often enough, which prevents other transactions from completing and thus guaranteeing that the current transaction will complete.

5 free points for tearing off page: Anonymous Feedback

List one thing you like about the *class* and would like to see more of or see continued (any topic - lectures, homework, projects, discussion site, topics covered or not covered, etc., etc.):

List one thing you would like to have changed or have improved about the class: