Midterm practice questions
Fall 2000 15-712

The midterm is open book, open notes, and will be held in class. We suggest that you prepare for the exam by answering these review questions, reading your classmates' paper reviews posted to the course webpage, studying the lecture notes and any of your own notes that you have taken, and skimming the assigned papers.

In the actual exam, you should write brief, precise answers. As opposed to what we said earlier, we are not looking for essays. Rambling brain-dump answers may be punished twice — you run out of time writing them and we run out of patience reading them. :–)

1. Soft Updates and Log-structured file systems both try to maximize the value of write-back caching. Give an example of what circumstances would cause each approach to outperform the other (assume soft updates is used with BSD FFS as described in the paper we read).

   Examples of when a Log-structured file system can outperform Soft Updates: (1) Soft Updates needs to undo / redo metadata blocks to ensure that on-disk data structures are consistent. Some workloads can cause an excessive amount of undo / redo activity. (2) In some situations (say you are deleting a large directory tree), the data structures used by Soft Updates can grow so large that the system must synchronously commit meta data updates to reclaim memory space. (3) Since the question concerns Soft Updates atop FFS, the writes to metadata blocks for some workloads may be scattered across the disk, while writes in LFS are sequential.

   Examples of when Soft Updates can outperform a Log-structured file system: (1) The cleaning overhead in a Log-structured file system may be a significant overhead that isn't present in Soft Updates over FFS. (2) By writing data and metadata sequentially, Log-structured file systems hope that read patterns will be similar to write patterns so that reads that don't hit in the buffer cache will be sequential. However, some workloads may have read patterns that do not match write patterns, and in these cases, the FFS layout policies may be superior.

2. Delaying changes to a set of files is a classic performance improvement scheme for distributed file systems. Describe the callback-based scheme for ensuring cache consistency in distributed file systems.

When a client caches some data, the server registers a callback, which is a promise to inform the client when the data is modified. Upon modification, the server "breaks the callback," meaning that it informs the client that its cached data is stale. The client invalidates its cached data so that when it is accessed again, it will miss in the local cache and a fresh copy will be fetched from the server, installing a new callback.

This is a performance and correctness improvement over having the client periodically ask the server if its cached data is current.

Another alternative is for the server to broadcast changes to all clients caching the data. But this approach is wasteful when the server isn't positive that the clients will actually need the new data in the future.

3. NASD devices export an "object"-level interface (instead of block-level). What are the advantages and disadvantages to this approach?

   Advantages: (1) It is easier to write a file system on top of an object-level interface versus a block-level interface because the file system implementer need not concern himself with writing code to allocate and manage individual disk blocks. We note, though, that an object-level interface could be provided as a library below the file system on a standard server machine with traditional, non-NASD disks. (2) An object-level interface at the device gives higher-level access information that the device could exploit for various optimizations. A NASD device that is aware of operations made on whole objects has the opportunity to do better object layout (both within and among objects) and request scheduling.

   Disadvantage: Some highly-tuned applications (such as databases) may have enough knowledge of their I/O workload so that they can achieve the best performance by controlling block layout themselves. An object-level interface prevents such application-specific optimizations.

4. Explain what NASD and xFS mean by "scalability" and how does each achieve it (in general terms).

   NASD: scalability results from not having a central file server that services data accesses. There is, however, a central file manager *that answers requests for object capabilities that are later used by clients to access data directly from NASD devices.*

   xFS: scalability results from no central server for any file system operation. Data service, metadata service, and cache service are three parts of the file system that can be shared by some or all of the participating machines.

5. Consider two types of callback semantics: (1) When a cached block is overwritten, the block's callbacks are broken before the writer is allowed to commit its changes; (2) the writer is allowed to commit its changes immediately and callback breaking can occur asynchronously. For these two alternative semantics, is it more natural to implement callback breaking

with message passing or RPCs? For this question, assume machines won't fail and there are no network partitions.

For the semantics in alternative #1, RPC is the more natural communication model. In this case, we need to know whether the client has invalidated its cached before we can proceed. This is easy to ensure with an RPC by having the RPC actually do the invalidation. Thus, when the RPC returns to the server, the server knows the data has been marked invalid.

For the semantics in alternative #2, message passing is more natural because the server doesn't have to wait for the data to be invalidated by the clients. The server can simply send messages (possibly asynchronously) to all the clients with cached data and then proceed to committing the writer's new data.

6. River and Abacus both try to spread work across nodes to improve performance. Explain how their foci differ and the resulting effects on their designs.

   River tries to balance data-parallel work across multiple producers and consumers. River is concerned with moving data adaptively between the two types of nodes to compensate for performance imbalances. Their design introduces a high-performance centralized queue to balance work among consumers and graduated declustering to balance producer work. Their programming model lets the programmer replicate consumer code and producer code across a number of consumer and producer nodes, respectively. River also has monitoring components to determine to what degree the data should be declustered.

   Abacus is concerned with function location. Given a stack of objects that make up an application, Abacus tries to determine how the objects should be partitioned among multiple machines to achieve high performance. The Abacus design includes a programming model to make it easier to monitor the resources needed by objects. This information, along with cluster resource availability information is fed to a resource manager that makes placement decisions.

7. Emerald presents a call-by-reference interface to the programmer, but for performance reasons, calls may actually be translated into call-by-move (where the object and possibly objects that the object references, ..., are moved to the remote node). Discuss scenarios under which the Emerald system should do call-by-reference and call-by-move.

   Call-by-move: (1) if there are no other current invocations on the object; and (2) if there seems that there will be many invocations on the object by the remote method.

   Call-by-reference (essentially the opposite of the above): (1) if there are other current invocations on the object; and (2) if there seems that there will not be many invocations on the object by the remote object.

Keep in mind that the object size, and the relative costs of mobility, local and remote invocation factor into the above decisions. For instance, if the object is very big and the system believes that there will be many invocations on the object by the remote method, then the system should do call-by-move. But, if the object is small and there will be many invocations, then call-by-move may not be a significant win.

Note that the programmer can specify that two objects should move together with the attach *annotation. Also, the programmer can force an object to be call-by-move with another annotation.*

8. Abacus currently assumes that an object's behavior is independent of where it is running. Assuming that correctness is not impacted, what goes wrong if this assumption is false?

   The cost/benefit resource manager in Abacus will not function properly. Abacus assumes that recent past is a good predictor of the short-term future. If an object changes its behavior on migration, then the history has no relevance.

9. SPIN and Exokernel both promote particular organizations for providing safe and efficient OS extensibility. Explain why this can be useful. Discuss what is needed, beyond such structures, to achieve such benefits in practice.

   Extensible operating systems enable the fast deployment of new services. Also, by enabling low-level access to resources, applications can often obtain performance close to the maximum performance obtainable from the raw hardware.

   Such benefits will be realized when there are clean interfaces available for programmers to use. People desire high-level programming interfaces because programming close to the hardware is hard. One can imagine such an interface being provided by a LibOS in the Exokernel. However, there is the perennial tension that higher-level interfaces will squander the potential benefit from access to low-level resources.

10. The "virtual file system" (i.e., vnodes) abstraction defines a specific interface for new FSs to be added. In most systems, this is done by statically or dynamically linking object code into the OS kernel. Explain why this is dangerous and one low-overhead alternative.

    Linking object code into the OS kernel is dangerous because bugs can crash the kernel. Also, if the code isn't trusted, it may steal secrets from the kernel or add back doors.

    There are several potential solutions: (1) write the code in a type-safe language to provide protection in a SPIN-like manner; (2) interpret the code so that safety checks can be made by the interpreter; (3) sandbox the code using software fault-isolation (not discussed in class); (4) supply the *code with an unforgeable proof of its correctness as in the proof-carrying code work (not discussed in class).*

Another solution is to write a small trusted vnode operation wrapper in the kernel that bounces vfs operations to a user-level program that implements the operations. Since most of the code is in a user process, it cannot interfere with the kernel. Crossing the kernel boundary frequently can lead to bad performance. In reality, it probably isn't, since file system operations are usually an order of magnitude slower than crossing the kernel boundary.