

EXAM 2 Solutions

CS 4210 Advanced Operating Systems

Fall 1999 • Georgia Tech/Computer Science • Hutto

Bershad User-level IPC

1. [10 points] Bershad et al. say "Shared memory message channels do not increase the 'abusability factor' of client-server interactions." Explain.

Traditionally the kernel can provide two types of "protection" for mediated data. The kernel can ensure integrity meaning that it verifies that the data is in some sort of acceptable format. You can think of this as type checking. Perhaps the most important thing the kernel does is to try and avoid "buffer overflow" situations. It can also provide privacy, keeping data from being seen by those that should not see it.

Bershad et al. make a strong argument that cross-address-space calls can perform the same types of integrity checks that we expect when we call a procedure in the same address space. Data type and size can be verified as part of the procedure call "prologue". There can be no buffer overflow issues because the client is simply pointing at a region of its address space and asking the "server" to map these pages. The server gets to decide WHERE.

As for privacy, the general memory mapping mechanism provides privacy. The data could also be encrypted if additional privacy was desired.

2. [10 points] The authors had previously developed a highly optimized "traditional" kernel-level IPC system they called LRPC. What observations about that system lead them to the development of a user-level IPC mechanism?

They had optimized the heck out kernel-level IPC. Cost breakdowns showed that a significant portion of the cost was trapping to the kernel and copying data into and out of the kernel. This lead them to conclude that kernel participation was, itself, the bottleneck. Hence a "user-level" IPC mechanism.

SG Chs 15, 16 Distributed Systems

3. [10 points] What is process migration? Why might you want to migrate a process?

Process migration is the process of moving an executing process from one machine to another in a distributed or, perhaps, multiprocessor system. There are many technical difficulties involved. Obviously you can't easily migrate a process between machines with different architectures. Migration is often suggested as a mechanism for "load balancing" systems to share or equalize the load. Periodically determining the "current load" of all machines in a distributed environment presents all sorts of technical difficulties as well. You also must avoid "thrashing" behaviors where a process keeps being moved back and forth between systems.

4. [10 points] How might the kernel of a distributed operating system differ from the kernel of a centralized operating system?

This is an open research question. Lot's of possible answers. An extreme example would be to distribute all kernel state (infeasible) so that cooperating kernels "share" all those complex data structures (page tables, process tables, file allocation tables, process blocking queues, etc.). Since all this data is rather sensitive, you must ensure the integrity and privacy of data as it is transmitted between systems. Also you must be very careful to avoid "lost data" during transfers. At a higher level, you need to extend the kernel "name spaces" to span multiple machines. One kernel must know about other machines and their resources and make periodic decisions: is this local or remote? Ideally this is hidden from the user but as we have seen you must have mechanisms for dealing with latency and failure.

Edwards paper (Jini book chapter)

5. [10 points] How are the issues of partial failure and consistency maintenance related in a distributed system?

Well, whenever a system fails it can be in an "inconsistent" state. Remember all those mutex locks? We add them in order to maintain some sort of logical "invariant" on our data structures. For example, updating a linked list involves reconnecting all the pointers properly after you have added or deleted an element. Interrupting the operation may leave "dangling pointers". It is this same problem on a larger scale. This problem is addressed by transaction systems that provide an "all or nothing" semantics to operations.

Now think about the complications when your linked-list is actually distributed across a set of machines. Now partial failures are possible. The other systems must determine and agree on where the failure is and what "inconsistent state" the linked-list is in if they have any hope of cleaning up the mess. This is very difficult in a distributed system. For example, different nodes may believe that different machines have failed because of link delays. Also new failures are possible during the cleanup process itself! Yick...

In short consistency maintenance is exacerbated (made worse) in the presence of partial failure.

6. [10 points] Keith begins the section entitled "New Failure Modes" with the sentence:

"Networked systems can fail in ways which stand-alone systems cannot."

Explain why this is a problem for a system that tries to "paper over the network." Give an example.

He gave an example of a stand-alone program that had been "bullet-proofed". The programmer checks all error codes from all system calls and responds appropriately. Assume we are accessing the filesystem. Now the program runs on top of a distributed file system but it has no knowledge of this. (That's the point of "papering over the network" – abstract away the existence of the network.) Now the remote system is down and a read command eventually returns some generic error code. We can't have a specific error code for this situation because that violates the "paper-over" principle. We don't want the program to know about the network. But there really isn't a corresponding situation in a stand-alone system. It is unusual for a process to be able to open and access a file and then have some subsequent read "time out". The best the application can do is assume that the file was deleted but that's not even the standard behavior under stand-alone UNIX semantics. Yucko...

Spring Nucleus (Doors)

7. [10 points] Assume that a cross-domain call-chain spanning several machines is active in Spring. What happens if the machine in the middle crashes?

There are two options: try to terminate the call or let it "complete". Spring does both when the machine in the middle fails. The caller returns immediately with a failure code. The "callee" machine will continue executing (to avoid inconsistency) and return normally. The OS kernel itself "receives" or "accepts" the return value and then just throws it away. Why do all this? It is very difficult to know what is happening "downstream". Cleanly terminating this computation is hard to do so we let it complete, maintaining some notion of consistency.

8. [10 points] Compare and contrast the Spring door mechanism to a traditional RPC system.

System V Shared Memory

9. [10 points] What's the difference between a shared memory id (shmid) and a shared memory key?

A key is the "name" of the shared memory segment. Think of it as a filename with the restriction that it has to be a number from a certain range. In fact the `ftok()` mechanism seems to have been added as an afterthought to allow an actual mapping of filenames to keys. The shmid is the kernel's internal identifier for the segment. It is actually an index into a table with a few complications.

10. [10 points] How does the kernel actually implement shared memory segments?

Stevens goes into some detail about this. Basically it does tricks with page tables. Page tables belonging to processes that share a page BOTH point to the frame containing the page. Page replacement, swapping, etc. are complicated. For example, if a shared page is selected for replacement then BOTH page tables must be updated. Note that processes can "map" the same page into different locations in their address spaces. Finally, shared memory is "kernel persistent" so the kernel maintains a table of "active" shared segments with attributes like owner, size, etc. Who "owns" this table? It doesn't belong to any particular process so it is a "system-wide" table. There are lots of other specific details. Actually there are two implementations. System V shared memory is implemented as described above. POSIX shared memory uses the "file mapping" mechanism that allows a process to "memory map" a file into its address space. Shared memory segments are associated with a temp file that is mapped by both processes. File mapping is ultimately implemented using page table tricks as described above.

Birrell and Nelson RPC

11. [10 points] Under what circumstances might an RPC-based program be less efficient than a carefully optimized message-based program?

Lot's of possible answers. If marshalling/unmarshalling is not implemented carefully then it is possible to go through a lot of extra encoding/decoding when you send data between two similar machines. Most commercial implementations like Sun RPC avoid this cost, however. A single-threaded RPC server (the default for rpcgen) can introduce delays that can be avoided with multi-threading. There is a small cost associated with the extra level of indirection introduced by the client and server stubs in an RPC-based implementation. This can be avoided in hand-crafted code.

12. [10 points] If the caller machine (client) in an RPC crashes while waiting for a reply, we say that the computation which was initiated on the callee machine is an "orphan". Properly terminating orphan computations is called the **orphan elimination problem**. What sort of issues makes orphan elimination difficult in RPC systems?

You got a taste of this problem when trying to interrupt an ongoing data transfer from the MediaServer in P2. Simply "killing" the thread or process that is executing the orphan computation is not sufficient because this may leave the system in an inconsistent state. The orphan computation may have itself called other procedures on remote systems. These must be also terminated cleanly. The details are VERY problem specific and it is difficult to provide operating system support that can guarantee proper termination for arbitrary computations. You'd basically have to have something like a transaction system that backs up and logs all activity so the effects of the computation can be "rolled back" or undone.

SG, Tannenbaum Distributed File Systems

13. [10 points] What are the advantages of a stateless distributed file server?

Not having to record and maintain client state means the server doesn't need any notion of "session". Clients don't need to "open/close" or "start/stop". This drastically simplifies the code. Statelessness also drastically simplifies consistency maintenance during "recovery" when the server fails and restarts. A stateful server must log all client activity to persistent storage. When it comes up, it must check its log and try to resume any incomplete activity. Of course the clients may have gone away so the server must go through a costly process of "checking in" with all its former clients. Statelessness even makes client failure easier to deal with. A stateful server maintains resources for active clients (allocates memory, etc.). If the client fails, the server must be sure to release these resources so the server is forced to monitor the status of all active clients.

14. [10 points] Caching is an important optimization technique. Describe the three potential locations for caching in a distributed file system. Briefly discuss the advantages and disadvantages of each location.

Tannenbaum does a superb job of explaining the various caching locations possible in a distributed file system. The client can cache recently requested file blocks in memory. If memory resources are tight, then the client can also cache blocks on the client disk (assuming it isn't diskless). This still saves the costly network communication to the server. Finally, the server itself can maintain a memory-based cache. Clearly, it is not useful to have a server-side disk cache! The whole point of a distributed file system is to retrieve disk blocks from a remote disk! It is possible to use both server-side and client-side caching at the same time.

client memory

- + fastest access
- relatively limited space
- + slightly simplifies cache maintenance (blocks don't need to be retrieve from disk)
- complex cache maintenance compared to server memory strategy

client disk

- + still pretty fast
- + lots of space
- doesn't work for diskless clients
- + slightly slower than client memory for cache hits
- complex cache maintenance compared to server memory strategy

server memory

- + no decentralized cache maintenance
- slower to access block than client-side caching