Operating Systems Depth Exam

Instructions:
Answer any six of the following seven questions.

Question 1. File Servers:

(1a) The original design of the Andrew File System (AFS) required a client to poll the server each time the client opened a file, even if the file was cached. Why was this polling necessary?

(1b) The revised AFS design (and used in the commercial AFS systems) does not use polling. Describe the method used in this revised design.

(1c) Describe the advantages and disadvantages of the original and revised designs. Why was polling eliminated?

Question 2. Shadow Tables:

Operating systems often provide tables that shadow hardware-based tables; i.e., the OS table has one entry for each entry in the tabled defined by the hardware. These "shadow tables" are used to extend or help manage a resource provided by the hardware.

Both Multics (Bensoussan, Clingen, and Daley, *The Multics Virtual Memory: Concepts and Design*) and VAX Unix (Babaoglu and Joy, *Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits*) use shadow tables. For each system, describe

(2a) the information contained in the hardware-defined table,

(2b) the information contained in the shadow table, and

(2c) how the operating system combines this (i.e., why are they both needed).

Question 3. TCP Performance

(3a) Some TCP implementations retransmit all unacked data when their retransmission timer goes off. When might this be the wrong thing to do and why? Describe the method proposed by Van Jacobson for dealing with this problem.

(3b) Describe "slow start" as specified by Van Jacobson including both the mechanism used and the problem that it addresses. How is slow start used in conjunction with the method described in part (3a)?

(3c) Why is it important to be able to estimate round trip times for receipt of TCP acknowledgements? Explain some of the difficulties with obtaining such estimates.

Question 4. Threads:

Consider a user-level thread package that runs in a standard Unix process. Suppose the package is implemented entirely in the user's address space with no kernel modifications.

(4a) What are the advantages of using user-level threads over standard Unix processes?

(4b) How do blocking system calls (such as input from a terminal) complicate implementation of user-level threads?

(4c) How would you modify the standard library or operating system kernel to get around this problem?

Question 5. Transactions

(5a) Dijkstra cites critical sections as a "fundamental problem" in the design of concurrent systems. What are critical sections, and why are they important? Explain their semantics in terms of the concept of *serializability*.

(5b) Serializability is sometimes called "synchronization atomicity." If processes that modify persistent data can fail, a related concept, called "failure atomicity" comes into play. Define this concept and describe two techniques for achieving it.

(5c) A process that makes changes at more that one site can guarantee atomicity with an algorithm called "two-phase commit." Describe this algorithm.

Question 6. Storage Allocation Consider the problem of allocating contiguous, variable-sized chunks of physical memory.

(6a) Briefly describe first-fit, best-fit, and worst-fit policies and compare their expected performance.

(6b) A variation of first-fit, sometimes called "progressive first-fit" or "next fit," starts each search of the free list from the point where the last search left off, rather the beginning. In what way would you expect this version

of the algorithm to perform better?

(6c) Suppose the problem is to allocate virtual memory (for example, to implement the `malloc()` and `free()` functions in the Standard C Library). What new considerations arise in this context? How do they change your choice of algorithm?

(6d) Suppose the problem is to allocate physical disk memory. What new problems arise in this context?

Question 7. Distributed Allocation In a "traditional" distributed file system, a file page requested by a process may be found in any of three places:

- In the file on the server's disk.
- In the server's disk-page cache in the server's RAM.
- In the client workstation's page-cache in RAM.

(In some systems, file pages are also cached on workstations' local disks. For the purposes of this question, we will assume client workstations are diskless.) With local-area networks becoming faster than disks[1], it makes sense to look in a fourth place: the disk-block caches of other workstations on the network. Managing the RAM disk caches of workstations as a shared resource has been called "cooperative caching" in a recent paper.

(7a) What are some of the advantages of cooperative caching? Consider the kinds of workloads you would expect on workstations and the correlation between their access patterns.

(7b) Devise an algorithm for cooperative caching. To simplify the problem, you may assume that all access is read-only and all pages are on a single server machine, which coordinates all requests for file pages. When a workstation has a cache miss (i.e., it wants a page that is not in its local cache), it sends a request to the server, which may forward the request to another workstation. What data structures does the server need to maintain? What page replacement policies should workstations use? Should workstations ever cache pages that have not been requested by local processes?

[1]For rough quantitative estimates, assume it takes 15msec to read a block from disk, 1,500μsec to fetch it from the cache of the server or of another workstation, and 250μsec to read it from local memory.