**Assignment 2**

CS 5352 Advanced Operating Systems Design Spring 2016

**Due Date: 3/7, 9 a.m., soft copy via Blackboard.**

**No submissions accepted after 3/7, 9 a.m., as we will post solution for you to prepare for the midterm exam.**

**Q1. (Chapter 3, Problem 1)** In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in a cache in main memory. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?

In the single-threaded case, the cache hits take 15 msec and cache misses take 90 msec. The weighted average is 2/3 x 15 + 1/3 x 90. Thus the mean request takes 40 msec and the server can do 25 per second. For a multi-threaded server, all the waiting for the disk is overlapped, so every request takes 15 msec, and the server can handle 66 2/3 requests per second

**Q2.** What is the difference between a process virtual machine and hypervisor-based virtual machine?

A hypervisor, also called a virtual machine manager, is a program that allows multiple operating systems to share a single hardware host. Each operating system appears to have the host's processor, memory, and other resources all to itself.

A virtual machine (VM) is an operating system OS or application environment that is installed on software which imitates dedicated hardware.

The hypervisor is the device or software which runs the virtual machine. It's typically responsible for allocating the resources, providing the interface between the virtual machine (the "guest") and the host system as well as any management software.

**Q3. (Chapter 3, Modified Problem 15)** (a) Please list two reasons why code migration is desired in distributed systems; (b) What is strong mobility? (c) Strong mobility in UNIX systems could be supported by allowing a process to fork a child on a remote machine. Explain how this would work.

Two reasons code migration is desired are:

1. It helps in achieving scalability.
2. It helps to dynamically configure clients and servers.

Strong Mobility means that not only is it possible to transfer the code segments and initialization data but also the execution segment, and running processes can be stopped and subsequently moved to another machine then resume its execution where it left off.

Forking in UNIX means that a complete image of the parent is copied to the child, meaning that the child continues just after the call to fork. A similar approach could be used for remote cloning, provided the target platform is exactly the same as where the parent is executing. The first step is to have the target operating system reserve resources and create the appropriate process and memory map for the new child process. After this is done, the parent’s image (in memory) can be copied, and the child can be activated.

**Q4. (Chapter 4, Problem 5).** C has a construction called a union, in which a field of a record (called a struct in C) can hold any one of several alternatives. At run time, there is no sure-fire way to tell which one is in there. Does this feature of C have any implications for remote procedure call? Explain your answer.

If the runtime system cannot tell what type value is in the field, it cannot marshal it correctly. Thus unions cannot be tolerated in an RPC system unless there is a tag field that unambiguously tells what the variant field holds. The tag field must not be under user control.

**Q5. (Chapter 4, Problem 7)** Assume a client calls an asynchronous RPC to a server, and subsequently waits until the server returns a result using another asynchronous RPC. Is this approach the same as letting the client execute a normal RPC?

No, this is not the same. An asynchronous RPC returns an acknowledgement to the caller, meaning that after the first call by the client, an additional message is sent across the network. Likewise, the server is acknowledged that its response has been delivered to the client. Two asynchronous RPCs may be the same, provided reliable communication is guaranteed. This is generally not the same case.

**Q6. (Chapter 4, Problem 12)** Suppose that you could make use of only transient asynchronous communication primitives, including only an asynchronous receive primitive. How would you implement primitives for transient synchronous communication?

Consider a synchronous send primitive. A simple implementation is to send a message to the server using asynchronous communication, and subsequently let the caller continuously poll for an incoming acknowledgement or response from the server. If we assume that the local operating system stores incoming messages into a local buffer, then an alternative implementation is to block the caller until it receives a signal from the operating system that a message has arrived, after which the caller does an asynchronous receive.

**Q7.** (a) What does Quality of Service (QoS) mean in stream-oriented communication? (b) Please show a sample technique that can help enforce QoS.

Timing (and other nonfunctional) requirements are generally expressed as Quality of Service (QoS) requirements. These requirements describe what is needed from the underlying distributed system and network to ensure that, for example, the temporal relationships in a stream can be preserved. QoS for continuous data streams mainly concerns timeliness, volume, and reliability.

Assured Forwarding and Expedited Forwarding allow us to enforce QoS.

**Q8. (Chapter 5, Problem 5)** Outline an efficient implementation of globally unique identifiers.

Such identifiers can be generated locally in the following way. Take the network address of the machine where the identifier is generated, append the local time to that address, along with a generated pseudo-random number. Although, in theory, it is possible that another machine in the world can generate the same number, chances that this happens are negligible.

**Q9. (Chapter 5, Problem 6)** Consider the Chord system as shown in Fig. 5-4 and assume that node 7 has just joined the network. What would its finger table be and would there be any changes to other finger tables?

Let us first consider the finger table for node 7. Using the same method as we introduced when discussing Chord, it can be seen that this table will be [9, 9, 11, 18, 28]. For example, entry #2 is computed as succ(7 + 21) = succ(9) = 9. More tables will need to change, however, in particular those of node 4 (which becomes [7,7,9,14,28]), node 21 particular those of node 4 (which ([28,28,28,1,7]) and node 1 ([4,4,7,9,18]).

**Q10. (Chapter 5, Problem 18)** High-level name servers in DNS, that is, name servers implementing nodes in the DNS name space that are close to the root, generally do not support recursive name resolution. Can we expect much performance improvement if they did?

Probably not: because the high-level name servers constitute the global layer of the DNS name space, it can be expected that changes to that part of the name space do not occur often. Consequently, caching will be highly effec- tive, and much long-haul communication will be avoided anyway. Note that recursive name resolution for low-level name servers is important, because in that case, name resolution can be kept local at the lower-level domain in which the resolution is taking place.