Design and Engineering of Computer Systems

Practice Problems

1. Consider the following CPU instructions found in modern CPU architectures like x86. For each instruction, state if you expect the instruction to be privileged or unprivileged, and justify your answer. For example, if your answer is "privileged", give a one sentence example of what would go wrong if this instruction were to be executable in an unprivileged mode. If your answer is "unprivileged", give a one sentence explanation of why it is safe/necessary to execute the instruction in unprivileged mode.

   (a) Instruction to write into the interrupt descriptor table register.

   Answer: Privileged, because a user process may misuse this ability to redirect interrupts of other processes.

   (b) Instruction to write into a general purpose CPU register.

   Answer: Unprivileged, because this is a harmless operation that is done often by executing processes.

2. For each of the following variables declared in a C program, state which part of the memory image of the process will contain the memory allocated to the variable.

   (a) A global variable declared outside any function in the program.

   Answer: Code + compile-time data

   (b) A static variable declared inside a function definition.

   Answer: Code + compile-time data

   (c) An integer variable declared inside the main function.

   Answer: Stack

   (d) The variable "p" in the following line of code located inside the main function `int *p = malloc(sizeof(int));`

   Answer: Stack

3. Consider a process P which invokes the default wait system call. For each of the scenarios described below, state the expected behavior of the wait system call, i.e., whether the system call blocks P or if P returns immediately.

   (a) P has no children at all.

   Answer: Does not block

(b) P has one child that is still running.
Answer: Blocks

(c) P has one child that has terminated and is a zombie.
Answer: Does not block

(d) P has two children, one of which is running and the other is a terminated zombie.
Answer: Does not block

4. For each of the traps below, state whether the trap instruction (e.g., int n in x86) is invoked by user code, or by kernel code, or by a hardware entity.

(a) System call to read a few bytes of a file from disk.
Answer: User code

(b) System call to obtain the PID of a process.
Answer: User code

(c) Interrupt from the hard disk device controller to indicate completion of I/O operation.
Answer: Hardware

(d) Interrupt from MMU due to page fault of accessed virtual address.
Answer: Hardware

5. For each of the events below, state whether the execution context of a running process P will be saved on the user stack of P or on the kernel stack.

(a) P makes a function call in user mode.
Answer: user stack

(b) P makes a function call in kernel mode.
Answer: kernel stack

(c) P makes a system call and moves from user mode to kernel mode.
Answer: kernel stack

(d) P is switched out by the CPU scheduler.
Answer: kernel stack

6. Consider the following C program. How many child processes are created due to the execution of this program? Draw a process family tree to explain your answer. Assume there are no syntax errors and the program executes correctly. Assume the fork system calls succeed.

```
void main(argc, argv) {
  for(int i = 0; i < 4; i++) {
    int ret = fork();
  }
}
```

Answer: 15 processes, besides the parent are created. Each round of fork creates one child processes each, for all existing processes so far.

7. Suppose two threads in a multi-threaded program use the following incorrect algorithm that implements software-based locking without using hardware atomic instructions. In this algorithm, "self" denotes the identifier of the thread trying to acquire the lock, and "other" denotes the identifier of the other thread. Below is shown code for the functions called by a thread when it wishes to acquire or release the lock. The flags for both threads are initialized to false. Explain what will go wrong when both threads contend for a lock using this algorithm.

```
acquire():
    flag[self] = true;
    while(flag[other] == true); //busy wait

release():
    flag[self] = false;
```

Answer: First thread sets its flag to true, context switches out, next thread sets flag to true too. Both will now spin indefinitely on each other's true flags. Deadlock occurs.

8. Consider two threads T1 and T2 in a multi-threaded program, that use a userspace spinlock for mutual exclusion when accessing a critical section. T1 acquires a spinlock on core 1, runs for some time, and is context switched out, while still holding the lock. At the same time, T2 starts running, requests for the same spinlock held by T1, and starts busily spinning for it. In which of the following scenarios is there a possibility of a deadlock between the two threads, i.e., will both threads end up waiting for each other indefinitely without making progress? Justify your answer by explaining how a deadlock does or does not occur. You may assume that the OS uses a simple round robin scheduler to schedule the threads.

   (a) T2 starts running on core 1 itself, i.e., on the same core on which T1 previously ran.
   Answer: No deadlock occurs. T2 will be switched out by scheduler after some time, T1 runs again, will release lock, which T2 can use later.

   (b) T2 starts running on core 2, i.e., on a different core from where T1 previously ran.
   Answer: No deadlock occurs. T2 will spin for lock until T1 releases.

9. Consider two threads T1 and T2 in a multi-threaded program. Both threads increment a variable `counter` multiple times during the course of their concurrent execution. In which of the following cases do the threads necessarily need to hold a lock when accessing this variable, in order to not have any race conditions?

   (a) `counter` is a global variable. T1 and T2 are running on a multicore system.

   (b) `counter` is a global variable. T1 and T2 are running on a single core system.

   (c) `counter` is a local variable inside the common start function of T1 and T2, which are running on a multicore system.

(d) `counter` is a local variable inside the common start function of T1 and T2, which are running on a single core system.

Answer: Cases (a) and (b) above need lock. The other two do not.

10. Consider the various CPU scheduling mechanisms and techniques used in modern schedulers.

    (a) Describe one technique by which a scheduler can give higher priority to I/O-bound processes with short CPU bursts over CPU-bound processes with longer CPU bursts, without the user explicitly having to specify the priorities or CPU burst durations to the scheduler.

    Answer: Reducing priority of a process that uses up its time slice fully (indicating it is CPU bound)

    (b) Describe one technique by which a scheduler can ensure that high priority processes do not starve low priority processes indefinitely.

    Answer: Resetting priority of processes periodically, or round robin.

11. Consider an OS running on a 56-bit CPU, which uses hierarchical paging to manage its virtual memory. The page size in the system is 8KB and each page table entry is 4 bytes in size. You may assume 1K = 2^{10}. Show all calculations below, and not just the final answer.

    (a) Calculate the number of levels in the page table of a process in the system.

    Answer: Total pages = 2^{43}. PTEs per page = 2^{11}. Pages in innermost level = 2^{32}, next level has 2^{21}, next has 2^{10}, and the final fourth level has 1 page. So number of levels = 4.

    (b) What is the maximum number of pages that may be required to store all levels of the page table in such a system?

    Answer: $2^{32} + 2^{21} + 2^{10} + 1$

12. Continuing with the previous question, you are now told that the OS uses demand paging, and the pages at all levels of the hierarchical page table are allocated on demand, i.e., when there is at least one valid PTE within that page. A process in this system has accessed memory locations in 4K unique pages so far. You may assume that none of these 4K pages has been swapped out yet.

    (a) Compute the minimum possible size of the page table of this process after all accesses have completed.

    Answer: 2 + 1 + 1 + 1 = 5

    (b) Compute the maximum possible size of the page table of this process after all accesses have completed.

    Answer: $2^{12} + 2^{12} + 2^{10} + 1$

13. Consider a system with 16-bit virtual addresses and 64 byte pages. The page table of a process in this system maps virtual page number i to physical frame number $i + 2$ for even values of i, and does not contain any valid page table entries for odd values of i. Using this page table, translate the following virtual addresses (given in decimal) to physical addresses (to be

computed in decimal), or state that the translation cannot be performed due to a missing page table entry. Show calculations in addition to the final answer.

(a) 1092

Answer: Page number 17, does not have a valid page table entry, so page fault.

(b) 1029

Answer: Page number 16, maps to frame 18. Physical address = 1157

14. Consider a process P that memory-maps a file into its virtual address space starting at virtual address v. The OS implements demand-paging.

(a) When (or on what event) is the page table entry corresponding to address v marked as valid?

Answer: During mmap system call

(b) When is the page table entry corresponding to address v expected to be marked as present?

Answer: When file data is accessed

(c) When is the file data expected to be read into the system memory?

Answer: When file data is accessed

(d) Assuming no other process is accessing this same file, how many copies of the file data do we expect to be present in memory, across user and kernel memory?

Answer: Only one

15. The page table design we studied in class is a regular page table, where the OS maintains one page table for every active process. The regular page table contains one page table entry for every virtual page of a process, which maps the page number to a physical frame number. Consider an alternate design of the page table called the inverted page table. In this design, there is only one page table for the entire system, and there is one page table entry for each physical frame in the system that maps the physical frame number to the logical page number (and PID of process to which the page belongs) stored in that frame.

(a) Between the two page table designs, which is expected to have a lesser memory consumption overhead? Justify your answer.

Answer: Inverted page table, because there is only one table for the entire system, unlike one regular page table per process.

(b) Between the two designs, which is expected to have a faster lookup when translating a virtual address to a physical address? Justify your answer. You may assume that the inverted page table also has a hierarchical structures and both page tables have the same number of levels.

Answer: Regular page table, because it is indexed by page number. One has to go over all entries in inverted page table to see which one matches.

16. A CPU accesses an instruction or data at a particular virtual memory address X. For each of the statements below, indicate whether the statement is true or false.

(a) If the entry to translate X is present in the TLB, then X will not be present in any level of the CPU caches. **Answer:** False

(b) If the page table entry corresponding to X has both valid and present bits set, then accessing X will never result in a trap to the OS. **Answer:** False

(c) If the entry to translate X is not present in the TLB, then accessing X will always result in a page fault. **Answer:** False

(d) If the MMU gets a TLB hit when translating X, then it can avoid performing the page table walk to translate X. **Answer:** True

17. Consider a simple filesystem design as discussed in class. For each of the scenarios below, can an abrupt power failure in the middle of a system call execution cause inconsistencies in the on-disk data structures of a filesystem? Answer yes/no and justify your answer briefly.

    (a) The disk buffer cache is a write-through cache, and changes to a block in cache are flushed to disk immediately.

    Answer: Yes, inconsistencies can occur, if some changes have been written through, but some changes of a system call have not been made yet.

    (b) In every system call implementation, all changes to data blocks are flushed to the disk before making any changes to metadata blocks.

    Answer: Yes, inconsistencies can occur. For example, there could be multiple changes to metadata blocks and only some could have made it to disk.

18. Consider a process that opens a file `/home/a/b.txt`, which does not already exist but will be created during the open system call using a suitable flag argument to the system call. The parent directory "a" does not have any more space within its data blocks to add a new directory entry. There is enough space on the disk to accommodate the new file. List down four changes that happen to on-disk data structures of the filesystem (not in-memory data structures) during the successful execution of this open system call.

    Answer:

    (a) Allocate new inode for b.txt, and update bitmap

    (b) Allocate new data block for directory a, and update bitmap

    (c) Add pointer to new data block in inode of a

    (d) Add directory entry mapping filename b to its inode number in data block of a

19. Consider a process that has a file open. The process now reads 64 bytes from the file (all located within the same data block on disk) using the read system call, providing the character array `buffer` as an argument to the system call. The data block containing the requested data is not cached in memory. The disk storing the file data is DMA-capable and its device driver is interrupt-driven. List any 4 actions performed by the filesystem during the execution of the read system call.

    Answer:

(a) Using file desciptor table and open file table, access inode of file

(b) Issue command to read data block from disk

(c) Disk DMA data into buffer cache, raises interrupt

(d) OS handles interrupt and marks process as ready to run

(e) When process runs again, copy data from buffer cache to userspace buffer

(f) Offset updated in open file table

20. Consider a multi-threaded webserver running on a multicore system with N cores. State whether the following statements are true/false, with suitable explanation.

    (a) Suppose the server has a one-thread-per-connection architecture, with M threads currently serving M connected clients. Then we always require $N \geq M$ in order to ensure that all clients get prompt service.

    Answer: It is not always required since M threads can run concurrently on fewer than M cores by time sharing.

    (b) Suppose the server has an event-driven architecture, with a single-threaded process handling M connected clients. We require that the server process must not perform any blocking operation in order to ensure that all clients get prompt service.

    Answer: Yes, we require server not to block in performing disk I/O and other such blocking system calls in order to promptly respond to events.

21. Suppose a system administrator finds that a long running application (e.g., a webserver that is not rebooted often) is consuming a disproportionately large amount of memory even while doing very little work. The system still has enough free memory in spite of this excess memory consumption. State two possible reasons/bugs which could be causing this behavior in the application, and suggest corresponding solutions to fix them.

    (a) First possible problem + solution:

    Answer: Allocating memory on heap and not freeing it up. Solution is to free up memory not in use, or to use a automatic garbage collector.

    (b) Second possible problem + solution:

    Answer: Not reaping zombie children. Solution is to call wait properly.

22. State whether the following statements are true or false.

    (a) The connect system call at the client returns a separate connected socket file descriptor after connecting to the remote server.

    Answer: False

    (b) The accept system call at the server returns a separate per-client connected socket file descriptor after connecting to the remote client.

    Answer: True

(c) A single TCP listen socket can be used to receive connection requests from multiple clients.

Answer: True

(d) All per-client connected TCP sockets at a server receive traffic on the same TCP port number.

Answer: We have given marks to both true and false, since the question does not mention if there is only one or more than one listen sockets.

23. Suppose a system administrator finds that the applications in the system are consuming a large amount of memory, leaving very little free memory in the system. The administrator also notices that the applications are very slow/unresponsive, and there is excessive disk activity.

(a) What is the most likely explanation for this observed behavior? What is the term used to describe this behavior?

Answer: Thrashing, where processes spend more time servicing page faults rather than doing useful work.

(b) Which of the following changes to the system (done independently) is/are likely to fix the above problem? Underline all that apply: terminating some processes, adding more DRAM, increasing page size, increasing disk space.

Answer: Terminating some processes, adding more DRAM

24. Consider a networking application running on a Linux system that is receiving large amounts of network traffic. Because the amount of traffic exceeds the capacity of the system, some packets will be dropped inevitably. For each scenario below, describe the specific point of the receive packet pipeline (e.g., which queue at which stage) at which the packet drops will likely occur.

(a) The bottom half interrupt handler of the OS does not get enough CPU time to run and perform the TCP/IP processing.

Answer: At the RX ring in the device driver

(b) The application does not get enough CPU time to read and process the received packets.

Answer: At the socket RX queue

25. Consider a multi-threaded web server, implemented over the socket API in Linux, running on a multicore system. The system administrator finds that the CPU core that is assigned to receive interrupts from the network card is at 100% utilization, and is unable to handle all incoming interrupts. Suggest any two solutions that can be implemented to mitigate this specific problem. You must clearly describe how the solution solves the problem, not merely state the name of the technique.

Answer:

(a) Receive side scaling to distribute interrupts to multiple cores

(b) Reduce number of interrupts via NAPI or poll mode drivers like DPDK

26. An engineer is trying to optimize the performance of a database server application and improve its throughput. The application stores large database tables on persistent storage, and also caches popular tables in memory for quick processing. The engineer performs a load test and profiling exercise to determine the performance bottleneck in the system. For each scenario given below, describe (not just name) in 1-2 sentences one technique that the engineer can use to optimize performance.

    (a) The engineer finds that all cores of the server are fully utilized due to processing compute-intensive user queries. Memory usage (fraction of DRAM that is used) is fairly low, and cache hit rates are reasonably high.

    Answer: Add more CPUs (vertical scaling) or optimize the compute-intensive parts of the code to be more efficient, e.g., via better data structures or libraries.

    (b) The engineer finds that all cores of the server are fully utilized, and the CPU spends most of its time stalling for memory access. The memory usage is fairly low, but the the memory bandwidth utilization (i.e., utilization of the memory bus that is used to fetch data from DRAM) is very high.

    Answer: Optimize code to have better cache locality and higher cache hit rates, so that the need to fetch from DRAM is reduced.

    (c) The engineer finds that the disk I/O bandwidth at the server is fully utilized. The memory usage is also very high. The CPU utilization is fairly low.

    Answer: Optimize code to reduce the working set size in memory, or add more DRAM.

27. Consider a multi-tier server system. The frontend server has several worker threads running on a single CPU core. Upon receiving a request, the server identifies a free worker thread and assigns the request to the worker for all further processing. Processing a request requires 0.1 milliseconds of initial computation in the worker thread, followed by at least 5 milliseconds (may be more if there is queueing) of wait time to perform blocking I/O operations to a back-end database. The OS scheduler takes care of context switching out the worker thread during its blocking period, so that CPU cycles are not consumed while waiting for a reply from the backend database. You may ignore the CPU time taken for all other operations, e.g., reading and writing over the network, context switching between threads, and so on.

    (a) Assume that the server uses a fixed pool of worker threads. What is the minimum number of worker threads in the pool that will ensure that the frontend server's CPU is fully utilized? What is the throughput of the frontend server (in units of requests/second) when it is fully saturated?

    (b) From now on, assume that the server does not use a fixed pool of threads. Instead, it spawns a new worker thread every time a request arrives, and the thread exits once the request processing completes. Assume that requests arrive at the frontend server at the rate of 1000 requests/sec, and the average time to process the request and return a response by frontend server is measured to be 10 milliseconds (including computation and wait time at the database). In this scenario, what is the average number of worker threads in the system at any point of time?

(c) In the previous part, compute the throughput and utilization of the frontend server.

(d) Consider the scenario in part (b) once again. Assume that the database has been upgraded to reduce the blocking wait time of a request from 5 milliseconds to 1 millisecond. What metrics (among throughput, utilization, response time) of the server do you expect will change, and how?

Answer:

(a) Number of threads = 5.1/0.1 = 51. Throughput at saturation is 1/0.1ms = 10,000 requests/sec.

(b) By Little's law, average number of threads = 1000 * 10 ms = 10

(c) Throughput = 1000 req/s, utilization = 1000 / 10000 = 0.1

(d) Throughput and utilization do not change, but response time will reduce.

28. Consider a multi-tier web server running on an 8-core CPU. The server has a master-worker thread pool architecture with a fixed number of worker threads. Upon receiving a client request, the master identifies a free worker thread and assigns the request to the worker for all further processing. Processing a request requires at least 10 milliseconds of wait time to perform blocking disk I/O operations, followed by 1 millisecond of CPU work in the worker thread. Identifying a suitable worker thread and dispatching a received request to it requires 0.1 milliseconds of CPU work per request in the master thread. The master thread is pinned to only execute on core 0 always, and the worker threads are pinned to run on cores 1 through 7. The OS scheduler takes care of context switching out the worker thread during its blocking period, so that CPU cycles are not consumed while waiting for a reply from the backend database. You may ignore the time taken for all other operations, e.g., network I/O, context switching between threads, etc.

(a) Calculate the maximum achievable throughput of the server in terms of requests/second, showing all steps in your calculation.

Answer: We can consider the master and worker threads as a multi-tier system. The master thread can process 10,000 req/s, and each of the worker threads can process 1000 req/s per core. The processing capacity of 7 worker threads is less than that of the master thread. So the bottleneck is at the worker threads. Maximum capacity is thus the processing capacity of the bottleneck which is 7000 req/s.

(b) Calculate the minimum number of worker threads that will allow the server to achieve the maximum throughput computed above.

Answer: Minimum worker threads needed per core = turnaround time / service demand = approximately 11 milliseconds / 1 millisecond = 11. Total number of worker threads needed across 7 cores = 77.

(c) Suppose the worker thread code has been optimized to reduce the per-request CPU work from 1 millisecond to 0.5 milliseconds. Calculate the maximum achievable throughput of the server once again in this new scenario, assuming everything else remains the same.

Answer: Now the processing capacity of the master thread (10,000 req/s) is lower than the combined processing capacity of all worker threads on the remaining 7 cores (2000 X 7 = 14,000 req/s). So the master thread becomes the bottleneck. The maximum throughput is 10,000 req/s.

29. Consider an implementation of a distributed key-value store, where multiple clients issue requests to get/put key-value pairs. Shown below are multiple requests issued by a client, along with the time the request was issued and the time that the completion response was received.

    - `put(k, v1)` at time $t = 1$, completed at $t = 1.5$.
    - `put(k, v2)` at time $t = 2$, completed at $t = 2.5$.
    - `put(k, v3)` at time $t = 3$.

    You are not told when the third put request was completed and response received. Now, the client issues `get(k)` to the key-value store at $t = 3.5$.

    (a) If the key-value store provides atomic (strong) consistency, what is the set of possible values that can be received by the client? You must list all feasible values under all scenarios, and there is no need to explain your answer.

    (b) Now, you are told that the client has issued the `get(k)` request at $t = 3.5$ to two different replicas simultaneously, and obtained a value of `v1` from one of the replicas. What is the set of possible values that can be received from the other replica, if the key-value store provides eventual consistency.

    Answer:

    (a) v2, v3.

    (b) v1, v2, v3.

30. Consider a set of 5 replicas N1–N5 maintaining a replicated log by running the RAFT consensus protocol. Shown below are the logs at each of the 5 nodes. Entries a, b, c are from term 1, entries p, q, r are from term 2, and entries x, y, z are from term 3. The nodes are about to elect a leader for their fourth term.

N1	a	b	p	q	r		
N2	a	b	c				
N3	a	b	p	q	x	y	z
N4	a	b	p	q			
N5	a						

(a) Which of the five nodes can be elected as leader in term 4? For each node that you think can be elected leader, list the set of followers who will vote for it, to demonstrate how the node can achieve a majority vote.

(b) Can entry c in the log be committed in term 4 (without the client retrying the request again)? Answer yes/no and justify your answer. If yes, describe the scenario under which this can happen. If you answer no, explain why not.

(c) Repeat part (b) for entry r.

(d) Repeat part (b) for entry z.

Answer:

(a) N1 (votes from N1, N2, N4, N5), N3 (votes from all), N4 (votes from N2, N4, N5).

(b) No, because for c to be committed, N2 must become the leader, and that is not possible.

(c) Yes, if N1 becomes leader.

(d) Yes, if N3 becomes leader.

31. Consider 5 replicas N1, N2, N3, N4, and N5 running the RAFT protocol to maintain a consistently replicated log for some application. The nodes have received the following entries: a, b, c in term 1, p, q, r s in term 2, and x, y, z in term 3. The content of the logs of nodes N1 to N4 are as follows. N1: (a, b, p, q, x, y), N2: (a, b, p, q, x, y, z), N3: (a, b, p, q, r, s), N4: (a, b, c). The log at N5 is not provided to you.

(a) Which of the nodes was likely not a leader in any of the three terms? Justify your answer.

Answer: N1, because all other nodes have some extra uncommitted entries in their logs, likely from the term when they were leaders.

(b) You are told that N5 put itself up as the candidate in term 4, got positive votes from N1, N3, and N4 (but not N2), and was eventually elected leader in the new term. Given this information, what are the probable contents of N5's log? You must list only those entries that will definitely appear in N5's log.

Answer: N5 will have all committed entries from all past terms, and fewer entries than N2. So the answer is: (a, b, p, q, x, y)

(c) After N5 becomes the leader in term 4, list all the uncommitted entries in the logs of nodes that will be rolled back.

Answer: c, r, s, z

32. Consider the design of a strongly consistent replicated key-value store application, which supports the following two operations: put (key, value) and get (key). All key-value pairs are replicated at 5 servers, using RAFT for consensus. A put request received at the leader is replicated at a majority of replicas before returning a response to the client. Put requests which are successfully replicated at a majority are considered as committed. Consider the following different variants of the implementation of the get request received at the leader. For each variant, state whether the implementation is correct or not, with a suitable justification or example. You may assume there are no network partitions.

(a) Get (key) will return the value in the latest committed put request for that key at the leader.

Answer: This is the correct option, because a strongly consistent key value store is expected to return the latest committed value.

(b) Get (key) will return the value in the latest put request received for that key at the leader, across both committed and uncommitted entries.

Answer: This is incorrect, because it may return uncommitted values which may be rolled back in the future, violating strong consistency.

(c) Get (key) request is sent to all replicas by the leader. Each replica returns the value in the latest put request received for that key. The leader randomly picks one of the responses and returns it.

Answer: This is incorrect. Some replicas may return uncommitted values which may be inconsistent.

33. Consider three nodes A, B, C in a distributed system. The nodes wish to run a distributed transaction using the two-phase commit protocol. Node C acts as the transaction coordinator, and starts a new transaction. Describe what happens in the following failure scenarios.

(a) The nodes finish the first phase of the transaction. Node A replied `yes` to the `prepare` message in the first phase. Subsequently, A does not receive the `commit` or `abort` message of the next phase from the coordinator, and times out while waiting for this message. What does A do in this situation, as per the 2 phase commit protocol studied in class?

(b) The coordinator C sends a `prepare` message in the first phase to nodes A and B. A replies `yes` to C, but B fails before it can respond. C times out while waiting for a reply from B. What does C do in this situation?

Answer:

(a) A must block while waiting for the next message from C.

(b) C can abort the transaction and send a message to abort to A.

34. Consider an e-commerce application, consisting of a HTTP frontend and several application servers to handle specific types of requests. One of the application servers is the shopping cart server that maintains the shopping cart for users. When the user adds (removes) an item from the shopping cart, the shopping cart server makes the change in the user's shopping cart stored locally, and also contacts another application server called the product inventory server (which maintains the currently available count for each product) to suitably decrement (increment) the count of the product added to the cart. This is done to reserve the product for the user. When the user updates the shopping cart, we require that these two operations of changing the user's shopping cart and updating the product inventory happen in an all-or-nothing atomic manner. To achieve atomicity, the shopping cart server uses write-ahead redo logging, where it logs the updated shopping cart and the operation to be invoked at the product inventory server in a persistent log, before actually executing the changes to the shopping cart (locally) and at the

product inventory server (via the API provided by the server). The product inventory server maintains a database of products (identified by a unique product ID) and the corresponding count. Several other application servers, e.g., server handling supplier restocking, also concurrently update the product inventory information via the APIs exposed by the product inventory server. Given below are several choices for what API must be exposed by the product inventory server to the shopping cart server, for it to correctly reserve products added to the shopping cart. For each design choice below, comment whether the API choice is good enough to achieve the goal of atomicity, with suitable justification or example for your answer.

(a) reserveItem(product ID, deltaQty), where deltaQty is the change to be made to the quantity of the specified product, in order to reserve it. The value of deltaQty is positive when adding items to the cart and reserving from the inventory, and negative when removing items from cart and returning items to the inventory.

Answer: This is not good enough. When the log is being replayed after a crash, some items may be reserved multiple times incorrectly due to replaying this API call, and there will be no easy way to identify such duplicate requests.

(b) reserveItem(transaction ID, product ID, deltaQty) where transaction ID is a unique identifier generated at the frontend for the user's action of adding/removing an item to/from the shopping cart, and deltaQty is as above.

Answer: This API is good enough. When an item is being reserved a second time due to replaying log after crash, the transaction ID can be used to check if the item has been reserved or not in the past, and the product inventory server can correctly reserve only once per transaction.

(c) reserveItem(transaction ID, product ID, finalQty) where finalQty is the updated value that the product quantity must be set to. The shopping cart server calculates finalQty by getting the original quantity of the product via another API call getQty(productID) to the product inventory server, and then subtracting deltaQty from the original quantity value.

Answer: This API is not good enough. Between the shopping cart server getting the quantity and updating it with final value, other components in the system may have updated the quantity in the product inventory server. By overwriting those changes with a final quantity via this API, some changes may be lost.

35. Consider the design of a distributed online banking server. The system consists of several banking server replicas and a load balancer that distributes incoming traffic across the replicas. Users send HTTP requests over TCP for the banking transactions they wish to perform on their accounts, which are redirected to a suitable banking server replica by the load balancer. Information that identifies a user account, e.g., account number, is available only inside the HTTP request, and cannot be inferred from any other fields in the TCP/IP packet headers.

(a) Suppose the system uses a partitioned stateful design, where the user accounts are partitioned across the replicas, and stored locally at each replica. Describe one efficient way in which the load balancer can correctly redirect incoming traffic to a suitable replica. You must explain the technique and not simply state its name.

Answer: The load balancer can accept connections, read the request, and use the account number in the request to redirect the request packets to a suitable replica that has the account information.

(b) Suppose the system uses a completely stateless design, where all replicas store/retrieve account information from a common remote database when handling requests pertaining to a user account. Describe one efficient way in which the load balancer can correctly split incoming traffic across replicas in this case. You must explain the technique and not simply state its name.

Answer: The load balancer can hash the TCP 4-tuple and redirect the request to any replica.

(c) Suppose the load balancer is receiving traffic at the rate of 10,000 HTTP requests per second (which is below the capacity of the replicated server system), and the average HTTP response latency of the system is calculated to be 20 milliseconds. Calculate the average number of active connections at the load balancer.

Answer: By Little's law, average number of active connections = 10,000 X 0.020 = 200

36. Consider the design of a distributed ticket reservation server, consisting of a load balancer which distributes incoming traffic across five server replicas for horizontal scaling. The system implements a completely stateless design, and all server replicas store and retrieve application state from a backend relational database. The backend database itself consists of two stateful replicas working in an active-active configuration for fault tolerance. The maximum throughput of a single instance/replica of the load balancer, server and the database components is 10,000 req/s, 1000 req/s, and 200 req/s respectively. The corresponding application-layer processing times at these nodes are 100 microseconds, 1 millisecond, and 5 milliseconds respectively. Besides these application-layer processing times, the end-to-end network roundtrip time through the system is 5 milliseconds on average. The performance bottleneck in the system is the backend database.

(a) Calculate the maximum possible throughput (in requests/second) and minimum possible response time of the system.

Answer: Since the backend database is the bottleneck, maximum throughput of the system is the throughput of the backend, which is 200 req/s. Note that even though the backend has two replicas, they are both processing the same set of requests in active-active mode, so the throughput is still 200 req/s. Minimum response time = 0.1 + 1 + 5 + 5 = 11.1 milliseconds.

(b) Suppose the system is receiving traffic of 100 requests/second, and we find that there are on average 10 requests under processing in the system. Calculate the total average waiting time for requests in the system across various components. Note that the waiting time is the time spent by a request waiting for service in various application queues, and does not include the application-layer processing times or the network round trip time.

Answer: By Little's law, the average time in the system = 10/100 = 0.1 seconds = 100 milliseconds. Out of this, about 11.1 milliseconds is spent in application processing and

in the network. So the time spent waiting for service is approximately 100-11.1 = 88.9 milliseconds.

37. Consider a multi-threaded application that processes incoming requests in batches, with a batch size of N. There are two kinds of threads in this application: N request threads, each handling one application request, and one batch processor thread that coordinates the processing of N requests at a time in a batch. When a request arrives, it is assigned to one of the request threads. The request thread must wait until all N requests in a batch arrive, then process the request and finish. The batch processing thread must wait until N requests arrive, then start the batch by signaling the N request threads. You can assume that only N requests arrive in the system and no more. Shown below is the code for the request thread. Write the corresponding code for the batch processor thread. The solution uses two condition variables (`cv_request` and `cv_batch_processor`), one integer `count` (initialized to 0), one boolean variable `batch_started` (initialized to false), and a mutex/lock. Your solution must not use any other variables.

```
//Request thread
lock(mutex)
count++
if(count == N)
    signal(cv_batch_processor)
while(not batch_started)
    wait(cv_request, mutex)
unlock(mutex)
//proceed to handle request
```

Answer:

```
lock(mutex)
while(count < N) wait(cv_batch_processor, mutex)
batch_started = true
signal_broadcast(cv_request)
unlock(mutex)
```

38. Consider a key-value store server using a primary-backup design for fault tolerance. The client sends put (key, value) and get (key) requests to the primary over TCP connections. The same TCP connection can be used to send multiple requests from a client one after the other. When the primary receives the request, it logs it to a persistent log, processes the request locally, and returns a response back to the client over the same TCP connection. When the primary fails, the backup becomes the primary, replays the entire log, and then proceeds to handle client requests again. Suppose a client has opened a TCP connection to primary server S1, and has sent the following requests put (k, a), put (k, b), put (k, c). The server has successfully sent a response for the first two. However, before the server could finish logging/processing the third request,

it faced a power failure and crashed. Subsequently, the backup server S2 took over as primary, replayed the log, and started processing client requests again. Assume that the IP address of the primary server S1 has been now assigned to the backup S2, and that both the primary and backup servers run on a Linux system with the standard TCP/IP stack.

(a) When this failover to S2 happens, does the TCP connection opened by the client to the key-value store break, and should the client reestablish the connection again? Or, can the client continue to send requests over the same TCP connection/socket as before? Justify your answer.

Answer: TCP connection breaks when any endpoint goes down. Even though S2 has the same IP address, it does not have the TCP-level state in the OS data structures to continue the TCP connection. So the connection has to be reestablished with S2.

(b) Suppose the first request that the client issues after the failure is get(k). What value is returned by the new primary S2, and why?

Answer: It will return b, since the value c is not yet logged or acknowledged to the client.

39. Consider a distributed key-value store where a load balancer distributes traffic across multiple server replicas. Clients perform get(key) and put(key, value) operations which are executed at the server replicas. The servers use a consensus protocol to maintain the replicated key-value database in a consistent state. Clients open a separate TCP connection for each get/put request, and the client port numbers are chosen automatically by their local operating systems. The load balancer distributes incoming packets to different server replicas based on the hash of the TCP 4-tuple (source and destination IP address and port number), i.e., all packets with the same hash go to the same server replica. Each server replica is designed using the master-worker multi-threaded architecture. One master thread running on one core accepts new TCP connections, and assigns the corresponding connected socket file descriptor to one of the worker threads based on a hash of the TCP 4-tuple, i.e., all sockets with the same hash go to the same worker thread. A worker thread handles the get/put request on the TCP connection suitably, and closes the connection. Consider two requests put (k, v1) and put (k, v2), for the same key, made by the same client over two different TCP connections at two different points in time. The put requests contain large values which may be sent across multiple packets of the TCP connection.

(a) Will all packets of one put request be entirely handled at one one worker thread at one replica, or can different packets in the put request be handled by multiple threads and/or replicas? Answer with suitable justification.

Answer: All packets will have the same hash of TCP 4-tuple and hence will be handled by the same thread at the same replica.

(b) Will both put requests be handled by the same worker thread in the same server replica? Answer yes/no and provide a suitable justification.

Answer: Both put requests may have different TCP 4-tuples and hence may be handled by different replicas and/or worker threads.

40. Consider a web server that uses a key-value store as a look-aside cache, in order to store results of recent queries to a backend database. Whenever the web server needs to look up a key "k"

in the database, it checks the cache first, and fetches the value "v" from the backend only if there is a cache miss. In this case of a cache miss, the web server also updates the cache with this fetched value by performing put (k, v) at the cache, for future queries. When the web server updates the database by performing put (k,v), it also updates the value in the cache by performing the same put at the cache also. Consider the following scenario: client C1 performs get (k) in the cache, and realizes that it is a cache miss. It then fetches the value v1 from the database. Before C1 can populate the cache by performing put (k, v1) in the cache, another client C2 performs put (k, v2) at the database and suitably updates the cache with value v2. Client C1's machine and network being very slow, its put operation to the cache reaches after C2's put completes, and C1 finally writes put (k, v1) into the cache.

(a) Is the data maintained in the lookaside cache consistent with the master copy in the database for key k? Answer yes/no with suitable explanation.

Answer: Cache is not consistent with the main database because it has an old, stale value v1 while the database has more updated value v2.

(b) Suppose we wish to design the get/put procedures in the system in such a way that the cache is never updated with stale or incorrect values due to delays or reordering of network packets. Suggest one possible design of the system by which you will ensure this property. Describe your idea briefly, and illustrate with an example. Your design must not change the basic operation of the lookaside cache described above, but you can maintain additional information or perform additional checks during the various procedures. You cannot assume that all clocks are synchronized accurately across nodes.

Answer: There are many possible ways to solve this problem. One could assign a monotonically increasing sequence number to every key-value pair when it is updated in the database, and this sequence number can be returned as response to the put request. This sequence number can then be presented to the cache when doing a put at the cache. The cache will accept puts only if the sequence number is more recent than the one it has already stored. Another valid answer is to prevent any concurrent updates to a key-value pair. That is, until C1 finishes its put at the cache, any further puts to the key at the database will be disallowed. Another valid answer is for the cache to check with the database every time it receives a put request, to make sure it's value is the latest. These last two solutions are somewhat inefficient though.