Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Problem Set: Networking

1. Consider a high performance networking application running on a high end multicore server. It is found that, under high incoming packet load, the system spends a large fraction of its time handling interrupts and context switches, leading to very little productive work at the application layer. Suggest one mechanism by which this problem can be mitigated. (For this question and the next, you are required to provide a description of the mechanism, not just its name.)

   Ans: Polling or batching interrupts

2. The current Linux network stack copies packet buffers several times, from the device to kernel space to user space. Suggest one mechanism by which the overhead of packet copies can be minimized, in order to build a high performance network stack.

   Ans: Directly DMA into user space (DPDK), or mmap kernel packet buffers to userspace (netmap).

3. Consider a web server that uses non-blocking event-driven I/O for network communication, but uses blocking I/O to access the disk. The web server wishes to run as multiple processes, so that the server can be available even if some subset of the processes block on disk I/O. Further, the web server wishes to receive web requests only on port 80, and not at different ports in the different processes. Suggest one mechanism by which the multiple server processes can handle requests arriving on a single port on the system.

   Ans: The master process can open a socket on port 80, fork multiple processes, and all child processes can accept connections from the same socket on port 80 using locks for mutual exclusion. Or, the master process can alone listen to requests on port 80 and assigns all blocking disk I/O to worker processes via IPC mechanisms.

4. Below are several problems with the kernel network stack that arise in multicore systems desiring high-performance network I/O. For each problem below, describe one technique that attempts to solve the stated problem. You are required to provide a 1–2 sentence description of the mechanism and how it fixes the stated problem, and not just its name.

   (a) Buffers to store packets are dynamically allocated and deallocated in the kernel, leading to dynamic memory allocation overheads.

   (b) The payload of a packet is copied multiple times, once from the NIC to kernel memory, and once again from kernel memory to userspace memory.

   (c) Multiple threads of an application running on different cores all contend for a lock to accept connections on the shared listen socket.

   Ans:

(a) Preallocate a circular ring of packet buffers and expose them to userspace processes (netmap and DPDK).

(b) Memory map the NIC ring into user space (netmap) or provide a userspace packet buffer ring to the NIC via a userspace device driver (DPDK).

(c) Have per-core accept queues.

5. Consider a multicore system running a TCP-based multi-threaded key-value store application. The incoming traffic to the system consists of new TCP connection requests, and get/put requests over the established TCP connections. In order to distribute the interrupt load across all cores, the NIC partitions incoming packets into multiple hardware queues using the hash of the 4-tuple (source and destination IP address and port) of the packet. The interrupts from each hardware queue are delivered to a separate core. The interrupts are processed via the regular Linux network stack on the various cores thereafter. The key-value store application consists of multiple threads, all of which access a shared hashmap data structure containing the key-value pairs.

   (a) Are the interrupts generated for all packets of a certain TCP flow guaranteed to be delivered to the same core by the NIC? Answer yes/no and justify.

   (b) Are all `get` requests for a certain key guaranteed to be handled by the same core at the application layer? Answer yes/no and justify.

   Ans:

   (a) Yes, because all packets of a flow will have the same 4-tuple hash.

   (b) No, because a request of a key can be sent over different TCP connections with different 4 tuple hashes, and hence can be processed by different cores.

6. Consider a TCP server socket program written in an event-driven manner. The server receives requests from multiple concurrent clients. The main thread of the server monitors read events on the server listening socket as well as all client sockets using the `select` or `epoll` family of system calls. When not processing any events, the server always blocks in an event-driven wait loop, e.g., `epoll_wait`, waiting for notifications. To process a client request, the server must read some data from the disk and send a reply back to the client. In order to avoid blocking the main event-driven server thread during disk reads, the server uses worker threads to block on disk I/O. After reading a client request, the main server thread spawns a new worker thread and passes the client request to the thread. This worker initiates the disk read and blocks until the disk read completes, while the main server thread continues to process other events on the sockets. Once a worker thread completes the disk read, it places the data read from the disk in a shared datastructure that is accessible by the main server thread (using suitable locking), and terminates. Assume that the main server thread does not automatically get any notification from the OS when the worker thread terminates. Now, we require that the server send a response back to the client once the worker thread has completed the disk read operation. There are multiple mechanisms to accomplish this goal, and the rest of the question lets you explore the various design choices.

   (a) One way in which the server can send responses back to the clients is by monitoring the status of the shared datastructure every time an event occurs, i.e., when the main server thread returns from the `select` or `epoll_wait` system calls. At this time, if the server finds that some worker has placed a response in the shared buffer, it can fetch this response and send it back to the client on its socket. This solution is almost but not fully correct. Describe one scenario where this solution will fail.

(b) Describe how you can fix the server design to overcome the above failure scenario.

Ans: (a) if only one request and no other traffic comes in, server will never wake up from epoll loop (b) Have a unix socket from worker to main thread. When worker finishes, it will write to the unix socket, which will cause the server to wake up from the event loop.

7. This question will explore the design of a simple multi-process file server. The server is expected to serve multiple concurrent clients. Each client opens a TCP connection to the server, and sends requests for files on that connection. The web server must read the name of the requested file from the socket, read the file from disk, write it back into the socket, and wait for the client to request the next file. The server must serve the client in this manner until the client closes the connection.

   You are given a single-threaded file server running as a single process on a multicore system with a Linux-like OS. Because the server performs multiple blocking operations (accepting new connections, reading from sockets, reading from disk), a single server process can neither effectively serve multiple clients nor efficiently utilize the multiple CPU cores. In order to overcome these problems, you must modify the server to fork some children and delegate work to the child processes. Note that you are constrained to increase the parallelism of the server only by spawning processes and not threads. Further, you may not make any of the blocking operations (e.g., socket reads) non-blocking.

   The following sub-parts of the question will let you describe the design of your multiprocess file server. Note that multiple correct designs are possible; it is sufficient if you describe one of the possible correct designs.

   (a) When (i.e., at what point in the server's code) does the main server process spawn a child process? And when does it reap this process?

   (b) How is the work of handling a client (accepting a connection, reading from socket, reading from disk, writing to socket) divided between the parent and child processes? In other words, who does what part of the work?

   (c) How do the parent and child processes exchange information with each other? Note that the communication can be implicit via variables or file descriptors inherited at the time of fork, or explicit via IPC mechanisms You must describe all such exchange of information between the parent and child processes.

   (d) State any other aspect of your design not covered by the questions above, and any assumptions you have made in your solution.

   Ans: One possible design is spawn a child every time a new client connects and accept returns. The parent process periodically (say before calling accept) reaps any dead children. The child process does the network read, disk read, network write for the client, while the main process does the job of accepting new connections. The child gets the new file descriptor of the client to serve via a variable implicitly from the parent.

8. Which **one** of the following system calls initiates the three-way TCP handshake?

   (a) `socket`

(b) `listen`

(c) `connect`

(d) `accept`

Ans: c

9. A modern DMA-capable NIC has received a packet over the network, and has raised an interrupt. A modern Linux-like OS executes an interrupt handler to service this interrupt. Which **one** of the following operations is executed by the OS during the top-half of the interrupt handler?

   (a) Copy the packet buffer from NIC memory to kernel memory

   (b) Copy the packet buffer from kernel memory to user memory

   (c) Updates to the RX ring pointers

   (d) Generation of TCP acknowledgement for the received packet

   Ans: c

10. Consider a web server system consisting of N replicas. Clients send HTTP requests for web objects (HTML pages, images, etc.) to the server over TCP connections. All requests are sent to a single server IP address and port that is publicized to clients (say, via DNS). A load balancer placed before the replicas rewrites the destination IP address of the packets coming to the server to redistribute traffic to the various server replicas. For every packet arriving for the web server, the load balancer computes the hash h of the TCP/IP header 4-tuple (source/destination IP address/port), computes $i = h$ mod N, and redirects the packet to the i-th server replica. Ignore any changes to the set of replicas, or any failures.

    (a) Are all packets of a given TCP connection always sent to the same replica? Answer yes/no, and justify.

    (b) Are all requests for a given HTTP web object always sent to the same replica? Answer yes/no, and justify.

    Ans: (a) yes because all packets of a connection hash to same replica (b) no because requests of a web object can come over multiple connections, and can hash to different replicas