1. Consider a socket program that has exactly one socket open for communication. The logic of the application is as follows: it calls `read` once on the socket by providing a buffer of size 1KB, prints out whatever has been read (if any), and repeats the read again in an infinite loop. It is known that the other remote end point of the socket rarely sends data. The designer of the application has the option of using a blocking or non-blocking socket. While of two choices will lead to a lower usage of CPU cycles by the application, and why?

2. Consider a system running a web server application on a standard Linux system. The system has \( N \) processor cores. The web server has \( N \) worker threads, each pinned to execute on a core. All threads simultaneously run `accept` on the same server socket and accept new TCP connections in parallel. Each server thread manages several clients that it has accepted, and communicates with the remote clients via `read` and `write` system calls to service their requests. Answer the following design questions.

   (a) Suggest one I/O mechanism by which a single worker thread on one core can handle potentially blocking I/O operations (such as network reads) across multiple client sockets simultaneously. (Note that you are not allowed to spawn new threads.)

   (b) When the number of incoming connections is large, it is found that the threads spend a long time waiting in the execution of the `accept` system call. Suggest one explanation for why this may be happening, and suggest one kernel modification to address this issue.

   (c) When a packet \( p \) of a TCP connection arrives at the network card, it is possible that the interrupt handling and TCP/IP processing for \( p \) runs on one core C1 and the final application layer processing is done by the server worker thread running on a different core C2. State one disadvantage of handling the same packet on two different cores. And state one mechanism by which this problem can be mitigated.

3. 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.)

4. 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.
5. 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.

6. 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 studied in class 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.

   (d) Opening a new socket requires a lock on the per-process file descriptor table and other unnecessary file-system related locking overheads.

7. 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.