CS550 Homework #2

Vaishnavi Manjunath
Kalpana Pratapaneni
Kavya Ravella

3.1 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?

Case 1: Single threaded Server

If the needed data is in the cache, it takes 15 msec. Cache misses take 15 + 75 = **90 msec**.

The weighted average is (2/3 * 15) + (1/3 * 90) = **40 msec**.

The mean request takes **40 msec** and the server can handle **25 requests/sec**.

Case 2: Multithreaded Server

There will a cooperative (non-preemptive) scheduler. The waiting for the disk will be overlapped.

So, every request takes **15 msec**.

We are assuming that there are 6 threads.

The server can handle **66 2/3 requests/second**.

3.2 Would it make sense to limit the number of threads in a server process? Explain your answer.

It makes sense to limit the threads in a server process. Below are the few points to justify for limiting the threads:

1. The threads have their own stack and hence they require memory.
   Having multiple threads will consume a lot of memory and will certainly affect other threads and the viability of the process itself.
2. Having too many threads can be chaotic. It will generally delay the processing of requests as you need to wait for the thread to become available. The data will have to be exchanged between threads and they have overheads like context switches.
3. In a virtual memory system, it may be difficult to build a relatively stable working set, and hence there will be page faults. This might lead to performance degradation and hence page thrashing

3.10 Constructing a concurrent server by spawning a process has some advantages and disadvantages compared to multithreaded servers. Mention a few.

The server can be iterative i.e., it repeats through each client and serves one request at a time. Alternatively, a server can handle numerous clients at the same time in analogous, and this type of a server is called concurrent server.

Having separate processes has its advantages. They are protected against each other which may prove to be necessary like in the case of a supersaver handling completely independent services. The process spawning is a costly operation which can be saved by using multithreaded servers instead. If the processes need to communicate, then using threads will be cheaper and in many cases and we can avoid having the kernel implement the communication.

3.13 Is a server that maintains a TCP/IP connection to a client stateful or stateless? Justify your answer.

A stateless protocol is a communication protocol in which no session information is retained by the server. Relevant session data is sent to the receiver by the client in such a way that every packet of information transferred can be understood in isolation, without context information from previous packets in the session. In contrast, stateful connection keeps the internal state of the client.

Generally, the transport layer at the server maintains state on the client, but not the server. Server does not concern what the local operating system is keeping track of. Assuming the server maintains no other information on that client, server could be considered as stateless.

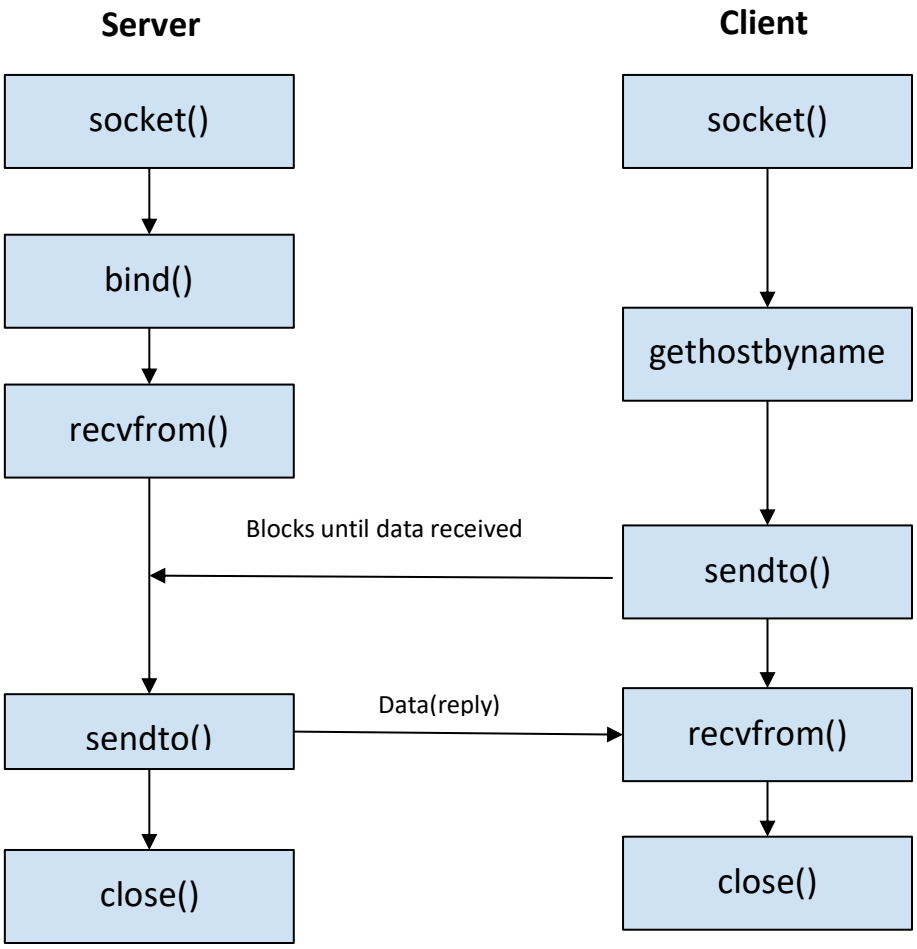
4.10 Describe how connectionless communication between a client and a server proceeds when using sockets.

<u>Connection-less Communication between Client and Server:</u>

Connectionless sockets do not establish a connection over which data is transferred. Instead, the server application specifies its name where a client can send requests. A server runs on some specific system and must have a socket that is bound with a port number. Generally, both the client and the server create a socket, but only the server binds the socket to a local endpoint. The server just waits for a client to make a request that can be generated by the sockets. The server can then subsequently do a blocking read call in which it waits for incoming data from any client. Likewise, after creating the socket, the client simply does a blocking call to write data to the server.

Under this model, communicating processes need not set up a connection before they exchange messages. Instead, the sender specifies a destination address in each message. There is no guarantee that the recipient will be ready to receive the message and there is no error returned if the message cannot be delivered. Connectionless sockets use User Datagram Protocol (UDP) instead of TCP/IP. Messages are sent and received using the system calls sendto and recvfrom.

The following figure illustrates the client/server relationship of the socket APIs for a connectionless socket design.



Connectionless server socket flow of events

The following sequence of the socket calls provides a description of the relationship between the server and client application in a connectionless design. Each set of flows contains links to usage notes on specific APIs. The first example of a connectionless server uses the following sequence of API calls:

1. The socket() API returns a socket descriptor, which represents an endpoint. The statement also identifies that the Internet Protocol version 6 address family (AF_INET6) with the UDP transport (SOCK_DGRAM) is used for this socket.
2. After the socket descriptor is created, a bind() API gets a unique name for the socket. For example, if the user sets the s6_addr to zero, which means that the UDP port of 3555 is bound to all IPv4 and IPv6 addresses on the system.
3. The server uses the recvfrom() API to receive that data. The recvfrom() API waits indefinitely for data to arrive.
4. The sendto() API echoes the data back to the client.
5. The close() API ends any open socket descriptors.

Connectionless server socket flow of events:

The connectionless client uses the following sequence of API calls.

1. The socket() API returns a socket descriptor, which represents an endpoint. The statement also identifies that the Internet Protocol version 6 address family (AF_INET6) with the UDP transport (SOCK_DGRAM) is used for this socket.

2. If the server string that was passed into the inet_pton() API was not a valid IPv6 address string, then it is assumed to be the host name of the server. In that case, use the getaddrinfo() API to retrieve the IP address of the server.
3. Use the sendto() API to send the data to the server.
4. Use the recvfrom() API to receive the data from the server.
5. The close() API ends any open socket descriptors.

4.14 Does it make sense to implement persistent asynchronous communication by means of RPCs?

Yes, but can only be implemented on hop-to-hop basis, in which a process managing a queue passes a message to a next queue manager by means of an RPC. Usually, the service offered by a queue manager to another queue manager is message storage. The calling queue manager is offered a proxy implementation of the interface to the remote queue, possibly receiving a status indicating the success or failure of each operation. In this way, even queue managers see only queues and no further communication.

4.17 With persistent communication, a receiver generally has its own local buffer where messages can be stored when the receiver is not executing. To create such a buffer, we may need to specify its size. Give an argument why this is preferable, as well as one against specification of the size.

Formal definition of persistent communication is a type of communication, where messages are stored by communication middleware as long as it takes to transport it at the destination application or until the destination application accepts it. The advantage of persistent communication is that it is not necessary for sending application to continue execution after submitting the message. Likewise, receiving application need not be running while message is being submitted. To make implementation easier it is better to specify the size of the buffer by the user prior itself.

***The size specification is preferred because the system creates a buffer of the specified size and with this buffer management becomes easy*.** Buffer Management – Incoming and Outgoing messages are buffered in memory and buffer management system ensures that there is enough memory available for the messages.

Still, if the buffer is full, messages may be lost. To avoid this to happen we should have the communication system which can manage buffer size. Creating a buffer with some default size, but then as needed buffer size should be increased or decreased. This way losing of message for lack of storage spaces will be reduced, but this method requires much more work of the system.

4.25 When searching for files in an unstructured peer-to-peer system, it may help to restrict the search to nodes that have files similar to yours. Explain how gossiping can help to find those nodes.

Based on how the nodes are linked to each other in overlay network, the peer-to-peer networks is categorized into structured and unstructured. Unstructured peer-to-peer networks do not impose a particular structure on the overlay network by design, but rather are formed by nodes that randomly form connections to one another.

With lack of structure in unstructured networks, when a node wants to find a file in the network, the search query must be flooded to all the other nodes which share the data across the network. Flooding causes very high amount of signal traffic, more CPU/memory usage and also does not ensure that search queries will be resolved and desired file will be searched or not.

So, instead of flooding method gossip protocol approach is used while finding the files in the nodes. ***Distributed system can use peer-to-peer gossip to guarantee that data is scattered to all nodes of overlay-network. The idea of using gossip protocol is very simple: if, during gossiping, nodes exchange information, every node will eventually get to know about all other nodes in the system*.** If node P has just been updated for data x, it contacts the adjacent node Q and tries to push the update to Q. Now node Q contacts adjacent another node R and tries to push the update to R. However, it is possible that R was already updated by another node. In that case, Q lose interest in spreading further and the data item x is removed. ***In this process each time when node discovers a new node, it can be evaluated with respect to its semantic proximity*** (It is calculation of semantic "distance" between two nodes within a network), for example, by counting number of files in common. The semantically nearest nodes are then selected for submitting a search query.