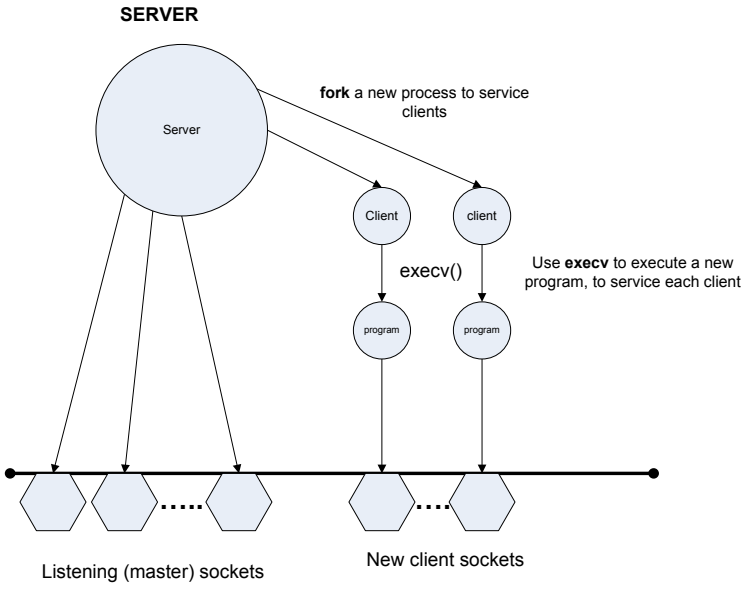
Client-Server Design Alternatives

- There are several alternatives when it comes to designing servers that will be servicing multiple clients.
- The simplest method is to have an **iterative** server that will service a pending client first and then move on to the next client. This is not a recommended design due to its inability to quickly service multiple clients.
- Another approach is to have a **concurrent** server that will create a new process or thread using fork to service each new client connection.
- Yet another approach is to use multiplexing calls such as ***select*** to handle multiple client requests
- One of the main disadvantages of most of the above designs approaches is their inflexibility: changing the code for any single service requires recompilation of the entire multi-service server.
- One solution to this drawback is to break down the multi-service server into independent components by using independently compiled programs to handle each service.
- This is best understood when applied to a concurrent, connection-oriented design. The following diagram illustrates how the design can be modified to break the large server into separate pieces.



- As the figure shows, the master server uses ***fork*** to create a new process to handle each connection.
- The new client process calls ***execve*** to replace the original code with a new program that handles all client communication.

- This simplifies the model since all that is required to upgrade the application is replacing the code that services the client requests. The server code does not have to be touched at all.

- The above describes a typical "Wait for a request - Spawn a new process - Give request to new process - Wait for next request concurrent design model.

- Server response times can be significantly improved by altering where the "Spawn a new process" step falls in the "Wait for request - Serve request" cycle.

- Deciding whether a server should be iterative or concurrent is a very important choice because it impacts the development and reliability of the entire program.

- Some factors a programmer should consider when making this decision are:

    - User demand
    - Processing speed
    - Communication capabilities

- The **level of concurrency** is the total number of processes a server has running at any given time.

- For most servers the current level of concurrency is not a major concern, the usual concern is the server's maximum level of concurrency.

- The "Wait for a request - Spawn a new process - Give request to new process - Wait for next request" model is **demand-driven** concurrency.

- Though this demand-driven model will suffice for most applications, there are system overhead costs associated with creating a new process for every request.

- Whenever a request arrives the server must ask the kernel to create a new process then wait for the process to be created before it can start servicing the request.

- If a concurrent server is providing a consistently simple or an interactive service (e.g. echo), then waiting for the system to create a new process will probably take longer than just serving the request directly.

- In this case a concurrent design should be abandoned in favor of an iterative one because the iterative server will have a faster response time and use fewer system resources.

Process Preallocation (Preforking)

- This model preallocates concurrent processes when the server starts to avoid the overhead of waiting for them to be created while a client request is pending.
- These processes do not exit when finishing a request, instead they wait for another request similar to a master server.
- Preallocation can be used on a multiprocessor system to match the amount of resources used to the amount of resources available, one server per processor, maximizing hardware utilization and minimizing server response time.

Preallocating Connection-Oriented Servers

- When the master server starts it calls ***socket*** and ***bind*** on the well known port then creates the desired number of "worker" processes and either exits or begins functioning as a worker itself.
- When a worker process is created it calls ***accept*** on the socket for the well known port inherited from the master server.
- In this way, all the workers are blocked waiting for a connection on the well known port. When a connection is made to the well known port the system unblocks one and only one of the blocked workers and creates a new socket for the connection and returns the new socket from the ***accept*** call.
- The worker handles the request, closes the new socket, and calls ***accept*** on the well known port again to wait for another connection.

Preallocating Connectionless Servers

- Preallocating connectionless servers is similar to a preallocating connection-oriented servers, each worker inherits the socket for the well known port from the master server.
- Each worker then calls ***recvfrom*** on the well known port, when a datagram arrives one of the workers is given the datagram and its return address.
- All of the workers can share the single socket for the well known port by using ***sendto*** to transmit replies without overlapping or interfering with each other.

Delayed Process Allocation

- Sometimes server performance can be improved by moving the process creation step after the processing of a request that has already started rather than before the request is even received.
- The reason for this is that even a blocked process uses system resources and there is a delay while the system's process manager changes context to the newly unblocked worker process.
- If the master server can handle a simple request (or reject an error) faster than the system can create a new process to handle it then time is lost by allocating a new process.
- When requests consistently take a long time to process, a concurrent server is preferred, and when requests are consistently short an iterative server is preferred.
- But when request processing times can vary widely the best approach is an iterative/ concurrent hybrid, delayed process allocation.
- A delayed process allocation server tries to handle every request it gets iteratively. Once it becomes apparent that a particular request is too long and/or complex for an iterative approach to be efficient the server creates a worker process to complete the request and resumes monitoring its well known port (s) for new requests.
- The most common method of deciding that a request is "too big" is to set a timer just before starting to process a request iteratively.
- If the request is finished before the timer runs out, the timer is canceled. If the timer goes off before the request is finished then the server creates a child process, goes back to monitoring its ports, and lets the child finish serving the request.
- An alternate method of deciding if a request will take too long is to check it for easily detectable errors and perform other "easy" parts of servicing the request.
- If the request has errors it is discarded and, if applicable, an error message is issued. If processing the request consists entirely of simple, quick activities then handle it iteratively, otherwise hand it off to a worker and resume monitoring ports.

Combining Process Preallocation and Delayed Processes

- Process preallocation and delayed processes allocation can be used together in the same server.
- One method is to have the master server use delayed allocation but have the worker servers remain after they are created as in preallocation.
- The tricky part to this method is managing the number of worker processes created. If too many are created they bog down the system, and too few means that the master server is doing a large portion of the work itself.
- One method of controlling the number of processes is to build an inactivity timer into the child processes, if a worker process has not handled a request for a predetermined length of time it terminates.
- Another method is for the master server to keep track of how many children it has by monitoring ***fork*** calls and **SIGCHILD** interrupts.
- Once a preset number of worker processes are active the master server stops responding to requests so it won't create any more processes and allows its workers to handle all requests themselves until the worker count drops.
- Other schemes are available on systems with different forms of Inter-Process communication.
- The main point to note is that most well-designed servers don't fall under a single design model, they are combinations of bits and pieces of different design models, customized to maximize the design strengths and minimize the weaknesses.

Concurrency In Clients

- **Servers** use concurrency for two main reasons:
    - Concurrency can improve the observed response time (overall throughput to clients).
    - Concurrency can eliminate potential deadlocks.
- It may seem that clients wouldn't benefit from concurrency, mainly because a client usually performs only one activity at a time. It sends a request to the server and cannot proceed until it receives a response.
- The issue of client efficiency and deadlock isn't as serious for clients either. However, concurrency does have advantages in **clients** as well:
    - Concurrent implementations can be easier to program.
    - Concurrent implementations can be easier to maintain and extend.
    - They can contact several servers at the same time, either to compare response times or to merge the results the servers return.
    - Concurrency can allow the user to change parameters, inquire about client status, or control processing dynamically.
- Most client programs simply wait until a response arrives from a server. If the server malfunctions, deadlock occurs and the client will block while attempting to read a response that will never arrive.
- Unfortunately, the user has no way of knowing if deadlock has occurred or if processing on the server side is just slow.
- Furthermore, the user can't know if the client has received any messages from the server.
- If users become impatient and decides that a particular response is taking too much time, they can only wait or abort the client program and try again later.
- Concurrency can help in situations like these by permitting the user to continue interaction with the client while the client waits for a response.
- The user can find out if any data has been received (status), choose to send a different request, or terminate the communication (abort).

Concurrent Contact with Multiple Servers

- Concurrency can allow a single client to contact several servers at the same time and report to the user as soon as it receives a response from any of them.

- Consider how concurrency can enhance a client that uses an ***echo*** application to measure **throughput**.

- Instead of measuring one connection at a time, it can access multiple destinations at the same time.

- Since measurements are concurrent execution is faster than a non-concurrent client. Another benefit is that measurements are all taken at the same time and are affected equally by the loads on the CPU and the local network.

Concurrent Clients Implementation

- Like concurrent servers, most concurrent client implementations follow one of two basic approaches:

    - The client divides into two or more processes, each handling one function.
    - The client consists of a single process that uses select to handle multiple input and output asynchronously.

- Modern operating systems support Inter-Process memory sharing (shared memory segments) and other Inter-Process Communications facilities such as Message Queues and Pipes.

- The multiple process approach can be used in such systems to support a connection-oriented application protocol that uses IPC mechanisms.

- Multiple processes allow the client to separate input and output processing. An input process reads from standard input, formulates requests, and sends them to the server over the TCP connection.

- An output process receives responses from the server and writes them to standard output. A third control process accepts commands from the user that control processing.

Single-Process Implementations

- Concurrent clients designed for UNIX systems usually implement concurrency with a single-process algorithm.

- A single-process client uses asynchronous I/O like a single-process server. The client creates socket descriptors for its TCP connections to multiple servers. It might also have an input descriptor.

- The body of a client program consists of a loop that uses ***select*** to wait for a descriptor to become ready.

- A single process concurrent client shares many advantages and disadvantages with a single-process server implementation:

    - Client reads input or responses from the server at the rate they are generated
    - Local processing will continue even if the server delays…thus the client will continue to read and honor control commands even if the server fails to respond
    - A single-process client can become deadlocked if it invokes a system function that blocks

Summary

- Concurrent execution works for clients as well as servers.

- Concurrent client implementations offer faster response time and can avoid deadlock problems.

- Concurrency can help designers separate control and status processing from normal input and output.