

CSIS0234B Computer and Communication Networks (Class B)

Reading for Tutorial 2

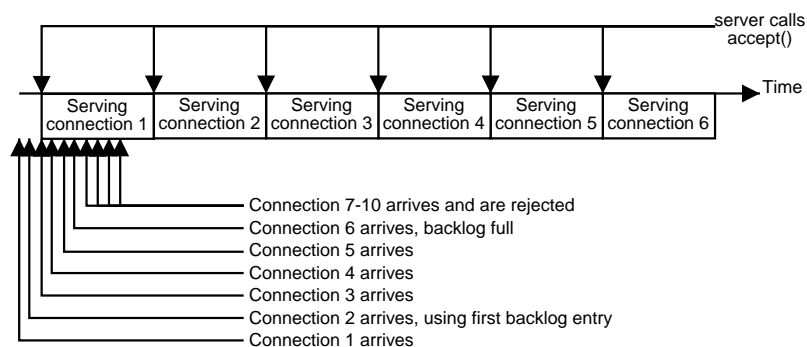
When Operating System Concepts do matter in Networks Programming

In the reading material for tutorial 1, we examined the API for C/C++ to make network connections, acting as servers and clients respectively. We showed some example code showing how iterative and concurrent servers differ. If you have not read that part of the tutorial reading, it is time to do so.

In this reading, we will examine the benefits and problems of different classes of servers, especially in cases where the servers are very busy. Apart from examining iterative and concurrent servers, we will also have a look at two other classes of servers: preforking servers, and super-servers.

1. Iterative servers

Iterative servers repeatedly do the following: accept a new connection, talk with the client using the connection until the client disconnects, and close the connection. With this simple approach, the server requires very **little resources**, but the client might not get the service it wants. Consider the situation if the server keeps a backlog of 5 connections, while 10 clients attempts to connect at nearly the same time:



As can be seen from above, connections are served in sequence, so clients must queue in the backlog and **wait** for its turn until other **clients** are done. Furthermore, some connections are simply **rejected** because the backlog is full. Typically, conversation within a connection involves a lot of waiting (e.g., waiting for the client to respond), so the CPU of the server can be under-utilized.

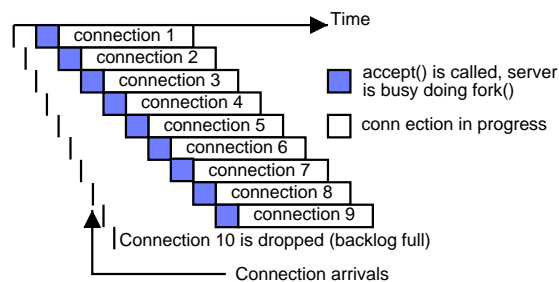
Why not increase the size of backlog instead of rejecting connections? Notice that connection 6 in the figure above will be served only after a long delay. Accepting further connections will further increase the delay. So the relatively small backlog (i.e., argument of `listen()`) can be seen as a way for the server to inform the client that “*to avoid forcing you to wait for an unreasonable amount of time, your connection is not made. Please do something else and connect again later*”.

2. Concurrent servers

In a concurrent server, a separate “slave” process is created for communicating with each client. The parent, or “master”, continue to wait for new connections. This has the benefit that **multiple connections** can be served **concurrently**. It also has the benefit that many processors can be used at the same time, with each slave process running in a separate processor.

The technique is especially important when connections involve more waiting time than computation time. Forking allows the server computer to switch to another slave process and begin to handle the next connection while a slave process is waiting for the client in another connection.

On the other hand, slave processes can be **time-consuming to create**. Depend on the amount of memory regions allocated to the master process, there may be significant delay introduced for the master to fork out a new slave process. Also, the master process cannot go back to accepting new connection until the `fork()` completes, which limits the frequency of connection acceptance. It also makes the server slightly **more complex**, e.g., need to deal with dying processes. More slave processes running concurrently can also slow down the communication. These behaviours are illustrated in the following figure:



3. Preforking servers

While concurrent servers above increases the CPU utilization and avoid the need for clients to wait for other clients, it can add significant CPU overhead and delay for process creation. In really busy servers, such overhead and delay can reduce the efficiency of a server. There are two problems here. Firstly, instead of creating a server process to wait for a connection, we ask a connection to wait for a slave process to be created for it. Secondly, instead of creating a server process to work for many connections, we create a slave process to work only on one connection. The former problem increases the delay, the latter increases the overhead.

Both problems can be dealt with by creating slave processes **before** they are actually needed (“**preforking**”). To use this technique, the master process can create N slaves after the `listen()` call. All children then call `accept()` to wait for connections. When a new connection comes, one of the child processes begins execution and handles the connection, while the other continue to wait. When the connection is over, the slave process will not terminate, but instead wait for the next connection. The following shows a simple way to achieve the effect:

```
pid_t pid[N];
for (i=0; i<N; i++) {
    pid[i] = fork();
    if (pid[i] == 0) { /* child */
        for (;;) {
            int newfd = accept(sockfd, ...);
            /* read and write using newfd */
            close(newfd);
        }
    }
}
/* the parent continues here. It might exit, or monitor the children. */
```

How to determine N? A simple solution is to use a fixed N. However, in a dynamic environment this creates too many idle processes during the time when the server is not in high load, and thus consume too much memory during those times. If most of these memory ends up in the swap space, heavy delay can result. Another solution is for the slave processes to report the number of connections that are being handled by the clients. The master will add them up to find the current “load” of the server. When the load falls below a certain threshold, some slaves can be shut down, thus reducing the memory usage. New slaves will be created when the load increases again.

When using preforking, it can be expected that backlog will be served nearly N times faster than when using an iterative server. As a result, it is usually a good idea to increase the backlog to allow more clients to be waiting.

One final note: in some OS (like Linux 2.2), when a connection arrives, **all** processes running `accept()` on the connection will be woken up. All except one will be put to sleep again. This is quite inefficient, and shows up boldly on some benchmark tests on Apache during the days of Linux 2.2. The problem is described as a **herd of the thunder**, and is fixed since then.

4. Use of multi-threading

One OS technique to reduce the cost of creating many processes is to use many threads in a single process instead. This also applies to networking. Both the concurrent method and preforking method can be used with multi-threading. For the former case, we say a **worker thread** is created for each connection. For the latter case, we say a **thread pool** is created to wait for connections. Both methods have better performance than a method employing many processes, with the thread pool being more efficient in resources usage. Apache is now moving towards multi-threading methods from a preforking method. However, since most students of this course do not have hands on experience with multi-threading, we will leave this option to the really adventurous few.

5. Super-servers

After spending a large amount of time looking at the techniques for dealing with very busy servers, let's spend a few moments looking at those servers that are idle most of the time. This is an important area of optimization, since many computers run many seldomly used services. If all these services are executed in a separate process executing a different program, the processes will use up a lot of memory, although most of the time they sit there doing nothing. The situation can be improved by having **one server**, called a “**super-server**”, to open and listen to **many sockets** at the same time, using `select()`. When the service is actually used, a new process is created, and a new **program** is executed, to deal with the request. In this way, most of the time the computer runs only one process to listen to all the connections. The downside is that to receive a connection now involve both a complete fork-exec, so it is much more costly. But for seldomly used service we probably would be more careful about how much memory is used.

For most Unix computer, the super-server used is called `inetd`. It is typically executed during boot time, and looks at the configuration file `/etc/inetd.conf` to find what ports should be listened, what program to execute when a connection is made at that port, what user id the process should be executed under, etc. You are advised to have a quick look at that file, and the man page of `inetd`, to clear up the concept. There are also improvements to `inetd`, most notably `xinetd`, which allows further configurations like during what time a service should be made available, etc.