Topics

Network programming
Synchronization
Threads

Problem 1: Denial of service

Problem: Can't run two service clients at a time

The code from last time in serviceserver-02.c presents a fragile server. We can't run two service clients at the same time, which means that a malicious user can connect and prevent anyone else from connecting, or a denial of service attack.

Solution: Fork to another server

We will try to fix this by running GDB and back tracing to see where the server is stalled in handleconnection when reading the line.

The answer is to fork another server.

Problem 2: Malicious control of resources

Problem: Each fork takes up too many resources

Solving the problem of the denial of service attack where a single malicious user can connect and prevent others from connecting actually creates another problem where we have given the malicious user control over the resources. Every fork takes up process id space and memory!

Solution: Organize the different processes using threads

The way we solve this is by considering threads.

Since a process is a virtual computer, a thread can be thought of as a virtual processor. A single process contains one more more threads each of which shares memory but has different stacks, register sets, and instruction pointers.

In C we use pthreads (an abbreviation of posix threads) and use pthread_create to create a new thread that is running a new, specific function. We see this done in serviceserver-05.c.
Problem 3: Too many threads

Problem: Too many threads

A program that handles multiple threads and multiple connections still doesn't avoid being killed by evil people because the evil users can still generate more and more threads that never get closed and still take up resources.

Solution: Set a limit on number of connections

One approach is to create a policy that is enforced in the program, such as a limit on the number of connections the user can have. We create a global variable, n_connection_threads, since global variables are shared by all of the threads. We increase this when a new thread is created and decrease if there are existing threads. Don't start the new threads until there are less than 100 open at a time.

This approach is shown in serviceserver-06.c.

The rate of new connections is slowed down because additional connections are accepted into a long queue. The OS throws out old connections before letting a new one in.

Problem 4: Low utilization

Problem: Utilization is poor

We see that utilization is poor because the service server is just spinning in a loop, waiting for one of the threads to die. What we really want to do is to block until the number of service servers goes down below 100 using sched_yield().

Solution: Condition variable

We can create a condition variable, which waits until a condition becomes true and signals when a condition will be true. In this case, we are waiting for fewer than 100 connections (pthread_cond_wait) and we want to signal the condition might be true when the thread exits (pthread_cond_signal).

We can see this in serviceserver-08.c.
Problem: Synchronization

However, this is more complicated than we would have thought because of synchronization. **Synchronization** is the process of coordinating multiple threads or processes, based on the concept that the relative order of events in multiple threads is completely undetermined except with explicit communication.

In the synchronization diagram below (where time flows down and multiple threads of control of program are seen as different lines), we see the potential problems that may occur.

![Synchronization Diagram](image)

In the following examples of execution ordering, we see the potential problems:

(a) If the code exits before checking the main connection, the number of connections will be stored at 100, which means a new thread should be started. This is fine.

(b) If pthread connection waits but then signals, the main thread wakes up and processes the next condition. This is fine.

(c) However, if the signal happens before wait, we have a **sleep wakeup race**.
**Solution:** Use locks

The solution is to make the check to go to sleep and wake up **atomic**, or executed without interruptions.

A **synchronization object** is an object that helps with synchronization, such as locks. The property of **mutual exclusion** is the property that at most one thread is allowed in a region of code at a time, where those regions are known as **critical regions**.

In our example, we see that if we combine the operations directly associated with synchronization and can place a lock before and after the critical regions.