UCB CS 162 - Operating Systems and Systems Programming "Lecture 5 - Abstractions 3- IPC, Sockets"

Lecture 5: Abstractions 3: IPC, Sockets

IPC

IPC stands for inter-process communication. The process model we have seen so far is specifically designed so that one process is not able to affect another in any way. This is by design, but it does make inter-process communication more difficult.

Therefore, we have to have a specialized way for two processes to communicate with each other, if both processes agree. One initial idea may be to communicate through reading and writing to files that are shared by processes. We have seen that forking a process creates file descriptors that are shared by both the parent and the child, so maybe one process can write to a shared file descriptor, and another process could read from it. The drawback to this approach is, as you might imagine, efficiency. Communicating with disk is really slow, and this approach becomes completely impractical as the number of processes increases.

The approach we use is to have an in-memory queue maintained by the kernel, which provides the same POSIX interface for reading and writing as reading and writing to a file. This is called a "pipe". You can create a pipe using the `pipe(int filedes[2])` syscall.

Sockets

When we extend the principle of IPC to two processes that are not running on the same machine, we get communication across the network. This involves a "network connection", which is essentially a pair of sockets connected to each other (for a bidirectional connection).

Actually transmitting data over the network is a whole another task, and this course does not deal with that. We assume that the messages we send over the network are correctly transmitted and reach the other device across the network.

Aside from the fact that these two sockets are on different physical devices, they work in the exact same way that a local socket (or pipe) works.

The client has a socket that is connected to a socket on the server. The client opens the socket (which may establish a TCP connection using a 3-way handshake), and can then read from it or write to it. Data written to the socket is transmitted over the network, arrives at the server, which reads it, and sends a response. The server then closes the socket once the client's request has been served.

In order to serve multiple requests at once, the server may create either a new process or a new thread for each request. Creating a new process for each request gives us better isolation, but is more heavyweight. Creating a new thread is faster, but doesn't offer the same isolation.

If a server creates a different thread for each request, then there is a chance that a spike in traffic can create so many threads that the throughput actually decreases. To avoid that, we create a group of threads called the "thread pool". The thread pool has a certain number of threads waiting to serve requests. We maintain a queue of requests that are waiting to be served, and when a thread from the thread pool becomes free, it dequeues the next request from the request queue. This way, there is an upper limit on how many threads can be created, and requests are still served concurrently.