Questions on the problem set:

1. Command groups
    a. One or more pipelines connected by && or ||. May contain multiple commands in it.
    b. Never will have ; or &.
    c. Nothing in a command group is explicitly in the background, since & is applied to the group. Some commands in a command group should not be waited for; in a pipeline, you don't wait for the first elements in a pipeline. Only the last element in a pipeline's status matters.
2. How does a command group end at interrupts? Ctrl-Z puts the whole command group in the background, and Ctrl-C also applies to the entire command group.
    a. You may think it's easy to have a list of commands in a command group and then kill them one at a time, but some commands in that list may have already exited or forked
    b. To address this, Unix implements the following system: every process has an ID and every process has a group ID, which defaults to the group ID of its parent. Setpgid(p, g) sets p's group to g. You can change the group of your children and your own group, but you can't change the groups of your parents. Also, Kill(-g, sig); sends sig to *all* processes in group g. This allows us to handle the entire group at once.
        i. Fork a new process "command parent" and set its group to a unique group, then fork off all the processes in one command group from it.
        ii. Main shell waits for command parent (if not background), command parent waits for children and worries about status.
    c. You want to kill just the currently running command group, not any other command groups
    d. In our shell, we're not managing sets of foreground and background processes
    e. Ctrl-C sets a global flag in the process that says that Ctrl-C has been pressed, and at some point later, that global flag is acted on. We want the signal handler to do as little as possible. We rely on the fact that the signal handler blocks whatever system call is currently running, and after the global is set, we then act on it.

UNIT 6: CONCURRENCY

IPC: Interprocess communication. One big example: the file system.
We'll motivate concurrency through network services.
We're going to talk about moving from interprocess communication to remote procedure call (RPC).
RPC: calling a function that's implemented in another process, possibly on another machine.

Service-lookup: Goal: repeatedly looking up service cache entries (protocols and ports).

- In the default implementation of the function that looks up things in this directory, finding things is quite slow especially if you want to look for many things.
- We implemented a cache for that function, which involved canonicalizing protocols, etc.
    - Example input/output:
    - ./servicelookup discard http
    - discard,9
    - http,80
- Every time a process starts up, it will have to start from a cold cache, and it will have to read a whole file and cache it. If every process has their own copy of this cache, there are many copies of this cache floating around in memory, and this memory would be better used for some other purpose. Wouldn't it be great if there were a shared cache between these processes? Improve speed / utilization.

Goal: implement my_getservbyname as a remote procedure call. Run this as a service server and then have our processes query it.
Some options…

- Signals work for very simple things, and don't have the ability to carry back complex responses.
- Pipes seem like they would be a good idea to communicate between files, but pipes need to be set up before the process started with the same parent. We can't ensure that Firefox, sh, service client, etc inherit a pipe from this service server, as we can't ensure they're started from the same parent.
- A file system would work, but it would be a bit ugly (you could have serviceserver writing responses in a directory).

We need communication between processes that don't have a common parent.
Name of this abstraction: a socket.

A **socket** is a Unix communication channel for bidirectional streaming communication, that's initialized separately by the client end and the server end. This is like two pipes, one in each direction. This is designed for scenarios in which we don't have both endpoints of the communication at once.

The **server** is on the passive end of the socket (listening for connections), and the **client** is actively making connections to the server.

1. Server creates a LISTENING file descriptor with a port.
2. Client creates an ACTIVE socket file descriptor, which connects to another listening file descriptor.
3. The operating system connects endpoints.
4. The server ACCEPTS the new connection, which creates a new socket file descriptor. Now the server has two sockets that are connected to the same port. The ACCEPT socket is connected to a specific client, and the LISTEN socket is still waiting for more connections.

Client side: Use socket and connect system calls to create fd. Once we have the file descriptor from connect, we can use it for writing and reading by opening with RDWR mode (a+).
Server end: more complicated, but in brief, sets up sockets, reads from the socket, and writes back to the socket

We can understand this through **message sequence diagrams**, which show messages between processes as they make RPC. These are shown as diagonal lines with time moving down, with responses and replies.

Code examples:

Serviceserver-00.c:

- Make the listening socket
- In the server, we read a request and then write a reply.
- We read a response using fgets and we write a response using fprintf
- But in this code, we're giving handle_connection the listening socket, not the client's socket. We haven't connected the client's socket to anything. We're missing the step when we accept a new connection from a client.

Serviceserver-01.c:

- We make a listening socket, accept a connection on listening socket, and pass that socket to handle_connection.
- When run, it's blocked on accept. When you make a connection from a client, we get an answer.
- There is no loop to handle additional connections.

Serviceserver-02.c:

- Forever, we accept a new connection and handle that connection.
- Now we can make multiple requests.
- We can run "telnet localhost 6168" to connect to the server, and now we can't make additional requests. This happens because this server can only handle one connection at a time. If someone starts a connection to the server and does nothing, they are using the server's entire CPU, preventing other programs from using that service. This is a **denial of service (DOS)** attack.

Serviceserver-03.c:

- Fork a new server connection
- Where's the right place to fork? In the new process, we want a connection to be in its helper process. Thus, we fork after accept, and then exit after handling a single connection (remember to exit from the child!)
- New type of connection: while true; do ./serviceclient discard. This will stop at some point. We're using a number of resources in our forks, so this can't go on forever.
- Problem: the parent isn't closing the connection. Eventually we'll have too many zombie processes. To solve this, we need to close the connection and use waitpid.

Serviceserver-04.c:

- We're automatically reaping our children, and we're closing all of the child file descriptors appropriately.
- Serviceblaster: opens a huge number of connections to the server and doesn't do anything with the connections. We are putting the client in charge of the server in the following sense: we're in charge of the number of child processes on the server.
- Processes are pretty heavy objects. They have their own virtual memory, etc. Since these processes do simple things, we don't really something heavyweight.
- We can use threads instead of processes here. A process can start another thread, and the second thread will act like another CPU running in the same process's memory space. Threads are like processes that share the same heap, but have different stacks.

Serviceserver-05.c:

- Instead of calling fork when a new request made, call pthread_create to make a new thread.
- Much better memory utilization, since threads don't have isolated memory spaces.
- Connection_thread: open the listener, handle the connection, and close it.
- Ran serviceblaster: We ran out of memory <500 requests, because we have created too many threads, each of which has its own stack so that the stack reached the heap.
- We need a way to control the number of threads we try to create per process. We can count the number of threads in a global variable, and we don't accept more connections if there are more than 100 threads (or some other number). This is an advantage of using threads, since with processes it's hard to manipulate globals with separate memory spaces.
- But serviceserver is constantly checking if any of the threads has exited so its using a ton of CPU. Fix will come next class…