CS 61 Lecture 11/21/13
Scribe Notes – Ana-Maria Constantin & Andrew Malek

**Questions from HW (Pset 5):**

**Command Groups** – pipelines connected by **&&** and **||**. They are treated as a whole, implementing them as a linked list is not necessarily the easiest solution.

How is UNIX dealing with Command Groups:

Every process has an ID.

Every process has a Group ID.

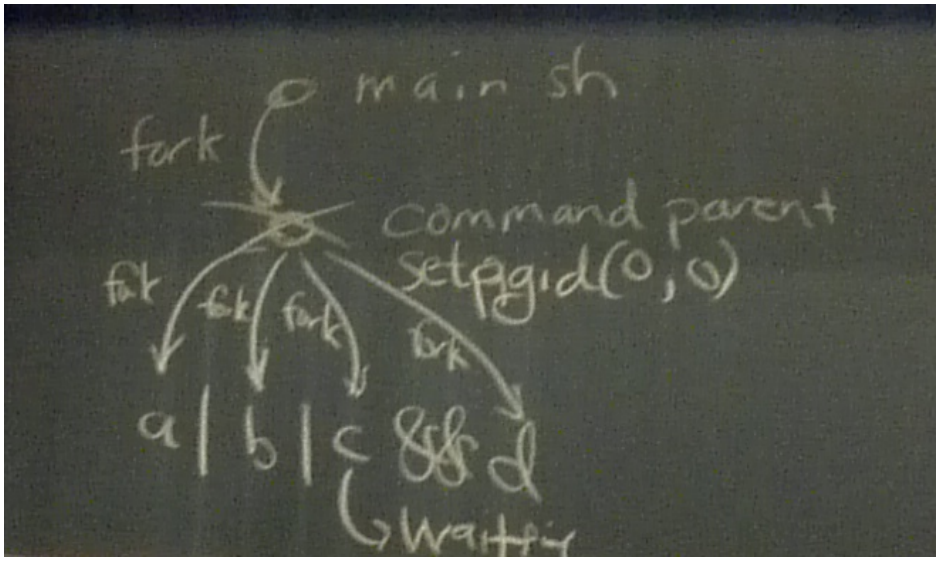
The command **setpgid(p,g)** : sets **p**'s group to **g**.

When to call the system call? You can't change the groups of your parents. Command groups are of the shell for its children, and of the children for itself.

Kill(-g, sig) : sends sig to all processes in Group g.

Kill is an atomic operation, like all system calls.

We have to make sure that all the processes in a command group have the same process group, as summarized in the picture below:



CS 61 Lecture 11/21/13
Scribe Notes – Ana-Maria Constantin & Andrew Malek

**Concurrency:**

IPC: Inter- Process Communication (signals, pipes, files)

We are moving from IPC to RPC (Remote Procedure Call). RPC makes the Web Possible.

RPC:

- calling a function that is implemented in another process, possibly on another machine.

Network services -> concurrency. We're moving from interprocess communication to an abstraction called remote procedure call. The latter is what makes the web possible. RPC refers to calling a function that's implemented in another process possibly on another machine. Why is that useful?

./servicelookup discard http
discard, 9
http, 80

→ **servicelookup** tells us the port where we find the given input (discard, http)

The files we are looking through are very big, which is why we are implementing a cache to go through the file.

It would be great if we could have a shared cache.

→ Therefore, what we do:

Aim to implement a service client that looks up processes and communicates (probably by other kernel) to a service server.

**IPC**: -*signals* don't work

-*Pipes* work!

But in order to make pipes work, we need to keep in mind that the processes connected by pipe have the same parent. This is hard to implement.

-Let's now look into *files*.

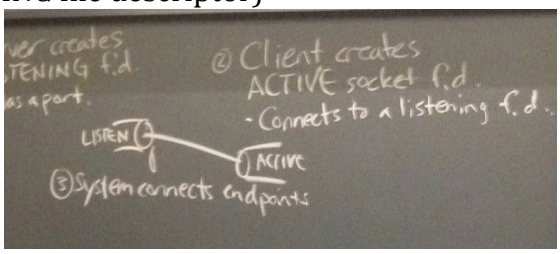
CS 61 Lecture 11/21/13
Scribe Notes – Ana-Maria Constantin & Andrew Malek

**Sockets!**

Unix communication channel (in other words a file descriptor) for bidirectional streaming communication, that is initialized separately by what we call *the client end* and *the server end*.

Pattern looks like this:

1. Server created LISTENING end point (LISTENING file descriptor)
   This LISTENING fd has a port.
2. Client creates ACTIVE socket file descriptor, and this connects to a LISTENING fd.
3. The system then makes the connection.



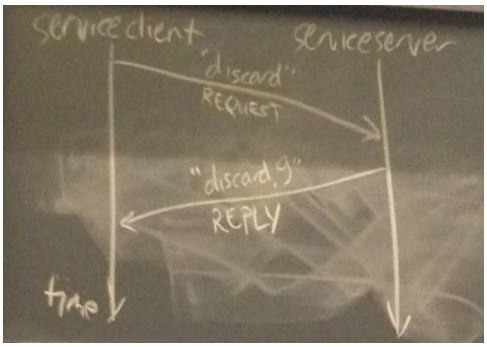

4. The server ACCEPTS the new connection, and that creates a new socket file descriptor.
   Then, the Operating System (Kernel) does something interesting. Server processes are created to speak with more than one connection at once.
   Both Server and Client are Processes. Server is a process that is listening for connections. The Client is a process that is actively making connections to Server.
   In the web, your browser acts like a client.

A socket basically gives us two pipes
→ A pipe to write to the socket.
→ A pipe to read from the socket.
The two pipes have the same file descriptors.

**Message Sequence Diagrams:**

How we expect the server to be used?

Any real server has a loop that accepts more than one connection. Then the server has to accept a new connection, then handle that connection. It is important to keep in mind that the server can do more than one connection at a time – if not, we can control the server by keeping it busy from other incoming connections. This attack is called **DOS** – **Denial of Service Attack**.




<u>serviceclient.c:</u>
Socket system call, and connect system call. Once we have got the file descriptor from connect, we can ask questions and read answer.

<u>serviceserver-00.c</u>
We create a listening socket. In the service, we read a service and write a reply. fgets to read a request from socket and writes to the same file descriptor. missing the step where we accept new connection from client.

CS 61 Lecture 11/21/13
Scribe Notes – Ana-Maria Constantin & Andrew Malek

serviceserver-01.c
Blocked at accept. You need a loop to accept more connextion.

serviceserver-02.c
Has a loop seems to be working
Use telnet localhost 6168 connected to localhost then we type the name of the service name and we get information but then go ba to the client code requesting a service number and it doesn't seem to be working. Why? the server is setup to handle only one client at the time!! while loop is only for one client. We are blocking out the server. Denial of service attack. We want to fork.
We want to fork when we have a new connection. Only want the parent to handle new connection to we want the child to exit. Now it works. But if we run it infinitely it crashes, because we forgot to close the fd from the parents. Now it's better.

serviceserver-04.c
Reaping our children, not creating zombie processes.

serviceblaster.c
Creating a connexion and keeping it open. This means that the server will open as many files as it can then crash.
But do we need processes? A process if heavy. We therefore need threads since they're less heavy. A process can implement a thread. So Threads are like processes but they share the same key.

serviceserve-05.c
Uses threads. Instead of calling forks to create an new process every time a new connection is created, we use pthread_create to create a new thread. Less memory utilization. They don't have isolated memory spaces. A thread_function is a function that will be executed by the thread once the thread begins It's exactly what was going on in the child process. This design has nice advantages. We can connect to it easily, also using telnet.

We ran out of virtual memory of the process since now we're using threads.
If we want to limit our number of open connections, we just have a global variable that sets a limit of let's say 100. But then we're in a spinloop that uses 99% of the CPU.