CS61 - Lecture 23 - 11/27/2012

--- Scribe Notes A by Hoang and Katzenelson

Agenda

1. Network programming
2. Serve programming
    a. File Descriptors
    b. Forked Servers
    c. Threaded Servers
    d. Event-driven Servers (if we have time)

Overview

- Today is about utilization.
- We will see lots of system calls as we learn different ways to structure network servers
- Explain system calls as we go along

*Code in L23 Directory

Last Time

- getservbyname(servicename, protoname): function on unix machines that implements a portion of the network database,
    - ex. servicename: “discard”
    - ex. protoname: “tcp”
    - returns a structure that represents the numeric information , ex. 9
    - Is slow for repeated use because it doesn’t store the database
- Implemented a cache (store the database) that allowed us to look up multiple services very quickly
- If all machines needed to look up services very quickly, then they would all have their own copies of these caches (multiple copies of databases with the exact same information)
    - What we will try to do is allow many processes on this computer (and any other computer) to utilize the same cache

This Time

- Will turn this function call into a remote call
- **Goal for the lecture**: implement a server cache to reduce redundancy

Look at the code!

- serviceparser.c
- void handle_connection(FILE * fin, FILE *fout):
    - loops over the lines of the input file
    - removes trailing whitespace
    - pass results to getservbyname function

Serviceparser program:

Why isn't this sufficient for implementing a cache that can be used by more than one program?

- Only one input/output location which are not (easily) sharable
- Connect input with cat and output with sort

- Or connect both input and output with cat

What is different about cat and sort ?

- Cat fills a 4096 char buffer before it outputs
- Cat does stuff incrementally as it reads in the input
- Sort needs to accept everything, sort has to read the entire input before it can produce anything because it has to know what order to output them in
- Sort needs to see an end of file before outputting, in this case: ctrl+D
- cat: example of a **streaming program**: generates output incrementally

yes | ./serviceparser | sort

- nothing happens because sort is waiting for the end of file on the input file, never sees one

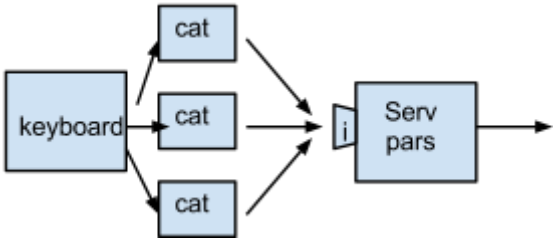
yes | ./serviceparser | cat

- does not need to see end of file to execute

This explains why "cat | ./serviceparser| sort" does not return any output unless we press ctrl + d to signify end of file

Ex. (cat; cat; cat) | ./serviceparser

- We have three cats that are hooked up to the same service parser (using the same input file)



- ";" means do one until it's done, done means ctrl+D
- Will execute first cat first, then the second one, then the third
- File closes after "ctrl+D" has been entered 3 times, one for each cat
- All sharing the same std input, the keyboard; also all sharing the same std output

Ex. (cat& cat& cat&) | ./serviceparser

- "&" means run command in background, the specific command keeps executing but the rest of the command line continues
    - sleep 1: wait for one second
    - sleep 2; echo p; sleep 3; echo x: wait for a total of 5 seconds (wait 2, print p, then wait 3, then print x)
    - sleep 2& sleep3& : returns after 0 seconds
        - programs are being run in parallel with the shell
        - both sleep 2 and sleep 3 are running in parallel
    - sleep 2& sleep 3: returns after 3 seconds
    - sleep 2; sleep 3 &: returns after 2 seconds
- One point: we can share file descriptors across multiple programs; makes it seem like we can use normal file descriptors to do this server cache
- Input is also shared, such that each line gets executed by one "cat"
- However, we can't distinguish which "cat" is being used when running the program (can't distinguish the connections) because they all have the same FD
- Also can't have each input read only its own output

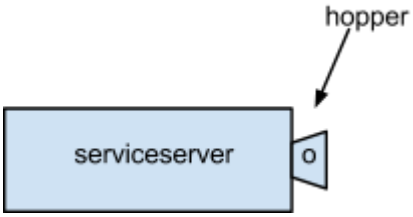
socket: A file descriptor used for networks; supports listening

listening socket: waits for another program to connect, creates a FD for it

sockets, like pipes, are not seekable

serviceserver00.c

- int fd = make_listen(port) // Creates/prepares a listening socket
    - Lots of stuff involved to make the listening socket (don't need to understand these details)
    - Many other system calls required
- Socket interface not as standardized as a regular FD - have to use lower level calls
- FILE *f = fdopen(fd, "a+")
    - takes a file descriptor and turns it into a file (gives you all the same awesomeness of buffer io)
- handle_connection(f, f);
    - socket is combining reading and writing stream for the same file descriptor
    - same basic process as before
- Sockets have a read stream and a write stream on a single FD

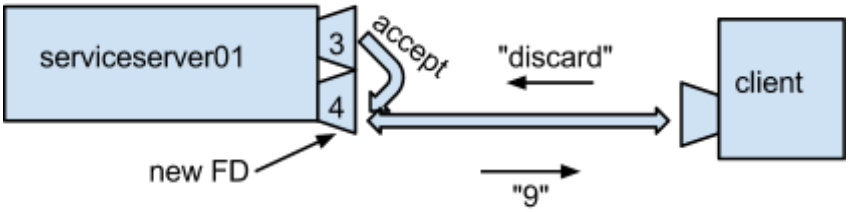


Initially this doesn't work. Why?

- Listening socket is not connected to anything. It's simply a stake in the ground. It only acts a place holder, has no other purposes. Therefore, when we try to read/write this place holder we receive an error
- Its only purpose is to start new connections. We need another system call.

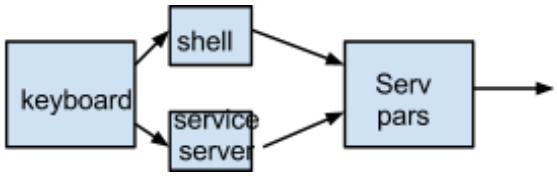
serviceserver01.c

- int cfd = accept(fd, NULL, NULL);
    - waits for a file connection from the listening socket, accepts, and then makes a new file descriptor
    - at the same time, the listening file descriptor stays where it is
- if (cfd < 0) {
  perror("accept");
  exit(1);
  }
  close(fd);
- Then can call handle_connection



run: "telnet localhost 6168"

- telnet allows us to make connection attempts to servers we do not understand
- on input "discard", we receive "discard, 9" - it works!
- When we exit and try to reconnect, it does not work - why not?
    - We closed the file descriptor as soon as we accepted one connection; therefore it cannot accept any other connections
- We also see "discard" at the command line when we exit serviceserver01 - why?
    - serviceserver can't accept the input because of the closed connection. The shell takes keyboard input also. Input just goes there instead

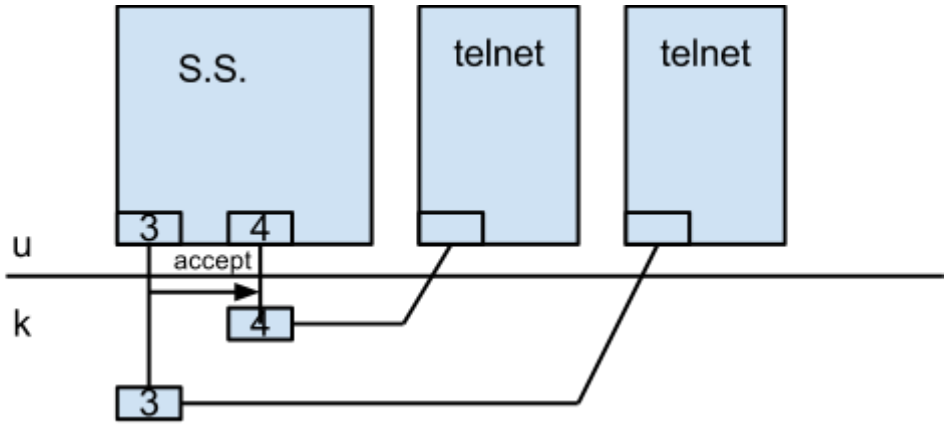


serviceserver02.c

- fixes the problem of exiting prematurely by introducing a loop
- while (1) {
  int fd = make_listen(port);...
  …
  fclose(f);
  }
- How many listening sockets will we have over the course of the program?
  - At least one per connection because we close it each time
- can now make multiple connections, but we still can't make simultaneous connections because there's no listening socket
  - It's like trying to open a file that doesn't exist
- kernel says to connection attempts go away when a listening socket isn't there

serviceserver03.c

- int fd = make_listen(port);...
  while (1) {
  ...
  }
- different from serviceserver02 because no longer creates a listening socket inside the loop
- Instead makes one at the beginning and accepts multiple connections on it
- While one connection works, we are able to make other connections. But, those other connections are dormant (nothing happens)
  - Only when we exit the active connection do the other connections start working
  - Why?
    - Although there is only one listening socket, the listening socket is fine.
    - The issue is with the connected socket



- What happens when we have 3 connections?
    - There is a queue of waiting connections; other telnets will wait their turn in the kernel
    - serviceserver does not call accept on the waiting connections because it is waiting for end of file for the first one
    - The data telnet sends is ignored until accept is called

./serviceblaster 1000

- attempts 1000 connection attempts. It will work. but does serviceserver have high utilization?
- utilization of this server: Prof. Kohler doesn't think it is very good. Multiple connection attempts, multiple clients trying to ask questions; however, it can only handle one thing at a time
    - rookie mistake: server has given control to other people on the network. never give control to other people
- how to improve utilization/make this system work?
    - what if we call interrupt? - Event driven server
    - Can't call accept before there's a connection because the machine will get stuck until it's handled, which it may never be
    - what if we handled separate connections in separate processes!

Calling fork to improve serviceserver03.c

- // handle connection
  pid_t p = fork();
    - copy-on-write fork means that cache memory is shared
    - fork returns twice since it creates a copy. to the original, it returns the new processe's id. to the child it returns 0.
    - want to handle connection in new process and let parent go back to listening
    - know it's the child process if p == 0
- if (p == 0) {
  FILE * f = fdopen (Cfd, "a+");
  handle_connection(f,f);

```
        fclose(f);
    }
```

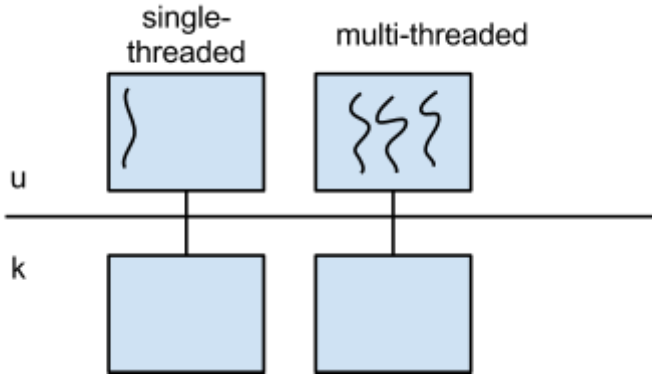
- starts a new process but does not close the parent process

- Parallel telnet connections work!
- Inside terminal there is a shell. Inside the shell we ran serviceserver03 once. serviceserver03 created two children, one with each telnet call. In total, there are three serviceserver03 processes running
- All processes are listening on the same accepted socket (that is shared by all of them)
- After we exit the child process, the system continues to remember that the process has existed because the parent process still is tracking the child process's pid returned by fork
- System is waiting for the parent to check up on the child (see whether it has exited). We can use the "wait" system call to do this check
  - this is how ";" works
- // Adding in the following line of code will allow us to end with only one serviceserver signal(SIGCHILD, SIG_IGN); // ignore your children, process level version of interrupts
- New problem - serviceserver03 gives the internet control over how many processes are being run
- This can cause the server to create an arbitrarily large number of processes and fail

<u>serviceblaster.c</u>

- If we keep running this, we open more and more connections and start more and more processes, theoretically infinite
- Original run capped at 1020 processes because the new connection was still present in both the parent and the child - need to close it in the parent
- Once we fixed that, we created a state where the machine was full of processes and couldn't fit any more
  - This is a problem because we should always be able to create a shell process to shut down the other processes

What if there were a way to have multiple logical threads within a single process?
Process: abstract computer (abstraction of the cpu, primary memory, file descriptors)
Thread: abstract CPU, much more light-weight (share fd, fd tables, memory, etc)



serviceserver07.c

- calls pthread_create to create a new thread
- instead of creating new processes, we are creating new threads - 2 threads within a process
- calls connection_thread: handles connection for thread and then exits

Preventing attacks

- serviceblaster fails at 482 connections for serviceserver07.c because the server ran out of memory
- Need to limit number of incoming connections
- One way: put fork in its own loop, limit it to n=20 times
- This spawns 20 processes by preforking, all share a socket