Chapter 11

Concurrent Connection-Oriented Server

Used when: After making a connection, the client has a long, continuous interaction with the server.
(telnet, rlogin, ftp)

For each client, the master server creates a separate process (slave server).

If you have 5 clients you will have 6 servers,
one master (parent) and 5 slaves (children).

Issue: don't forget to shutdown obsolete slave servers (exit the processes).

Note: run-away client and slave processes are the most common error in this type of server.

A run-away may consume substantial CPU!

Detailed Comments

see Section 11.5 for code. (`TCPechod.c`)

The base server:

```
int main(int argc, char* argv[]){
  char *service = "echo";
  struct sockaddr_in fsin;
  int alen;
  int msock;
  int ssock;
  switch (argc) {
   ...
  }
  msock=passiveTCP(service,QLEN);
  (void)signal(SIGCHLD,reaper);
```

This is very similar to the iterative server:
`"echo"`: default service
`fsin`: client address (`accept`)
`alen`: for accept (`accept`)
`msock`: master socket
`ssock`: slave socket
`switch`: same as in all servers
`passiveTCP`: get a master socket (same as Chap 10).
`signal`: see reaper discussion (new).

```
while (1) {
  alen=sizeof(fsin);
  ssock=accept(msock,
    (struct sockaddr *)&fsin,&alen);
  if (ssock<0){
    if (errno==EINTR) continue;
    errexit(...);
  }
```

`while (1)`: Handle one client request per iteration

`accept`: Accept a client,
get slave socket connected to that client.

`ssock<0`: this happens if there is an accept error.

It also happens if the reaper runs while we are waiting at the accept. In this case the accept is interrupted by the reaper. The error will be `EINTR`. If it was an interrupt, the `continue` will take us back around to the while and we will reschedule an accept.

```
  switch (fork()) {
   case 0: /*child*/
     close(msock);
     exit(TCPechod(ssock));
   default: /*parent*/
     close(ssock);
     break;
   case -1: errexit(...);
  }
}
```

`fork`: generate a slave server process

Parent process:
`close(ssock)`: the master server doesn't talk to the client, so it closes the slave socket and loops back to accept.

Child Process:
`close(msock)`: the master socket is used only to accept clients, the slave server doesn't need it, so it closes it.

`exit(TCPechod...)`: The child calls the procedure containing the slave server.
When the slave procedure is done, the slave process is done, and the exit terminates the slave process
You never return from the exit call!

The Service Procedure

```
int TCPechod(int fd){
 char buf[BUFSIZ];
 int cc;
 while (cc=read(fd,buf,sizeof(buf))) {
   if (cc<0) errexit(...);
   if (write(fd,buf,cc)<0) errexit(...);
 }
 return 0;
}
```

`while (cc=read...`: Read a "block" of information from the client. On end of file, read returns a 0; which causes us to exit the loop and return from the procedure. On a read error, generate a message and exit.

`write`: Write exactly what we read back to the client.

If anything goes wrong `errexit`.

Note: segment boundaries are not preserved.

Example: Client may write 3, 2000 byte buffers and server may get 2, 1000 byte reads and a 4000 byte read.

Again, a 0 byte only occurs if the socket at the other end of the connection has been closed. (Otherwise the I/O blocks until something is available.)

Termination:

Rule: a process delivering a signal must make the delivery before it can exit.

Notation: a child exiting delivers a signal.

Parent: the parent must receive delivery before the child can exit.
if the parent dies,
New Unix: `init` (process 1) receives delivery of any signals and discards them.

In the main program we saw:
`signal(SIGCHLD,reaper);`
When a `SIGCHLD` (child has exited signal) occurs, interrupt the main program and call the `reaper`, when the reaper is done, the return becomes a return from interrupt.
If the main program is waiting at the accept, the accept will get an `EINTR` error.

```
void reaper(int sig){
 int status;
 while (wait3(&status,WNOHANG,NULL)>0);
 (void) signal(SIGCHLD,reaper);
};
```

reaper–`while`: consume all pending `SIGCHLD` interrupts.

`wait3` is the call waits for a child to exit.
It returns the process ID of the process that exited.
Since we are here because a child exited we should get at least one process id returned.
The `WNOHANG` says to return the value 0 immediately if there are no exit signals waiting.

It also returns information about the interrupt in the `status` and variable. This information is not used by this program.

Comer code in the book has an error (>= in the while).
If no `SIGCHLD` interrupts are waiting,
`wait3` returns 0 and we want to leave the reaper.
The >= will cause the program to hang until a child has exited, which could be forever.
(This has been fixed in our version of the Comer code.)

Notice the `while` loop is empty, it does nothing.

`signal`: the signal must be reset (POSIX)

Reaper

What happens with no reaper.
No one receives the children exit signals
children don't exit
You run out of processes
Message: cannot fork, out of processes

Problem: You can't do a "ps" to see this (you're out of processes and ps is a process).

Problem: When you kill the parent, `init` handles all the children exit signals, so they go away (before you get a chance to do a ps).

Data Boundaries

Since TCP does not preserve boundaries it is up to the programmer to do so. There are several methods.

1) Make a new connection for each data block. You can read until end of file. In the echo the code was:
`while (cc=read(fd,buf,sizeof(buf)))`

2) Know the size and use a reassembly loop. Code from chapter 7 was:
`for (inchars=0; inchars<outchars; inchars+=n)`
where `outchars` was the expected size of the echo, or
`for (inchars=0; inchars<4; inchars+=n)`
where `4` was the expected size of the integer

3) Send the size, then send the data. A for loop to get the integer containing the size. A `ntohl` on that integer (since it needs to be shipped in network order). Put that integer into `outchars` and loop.

4) An end of data block flag. Common flags are `\n` when the data block is a text line and `\0` when the data block is a string.