Concurrent Network Servers

Setup: Server

```
listenfd = socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servadr.sin_family = AF_INET;
servadr.sin_addr.s_addr = htonl(INADDR_ANY);
servadr.sin_port = htons(9876);
bind(listenfd, (SA *)&servadr, sizeof(servadr));
listen(listenfd, LISTENQ_SZ);
for (;;) {
  clen = sizeof(cliadr);
  connfd = accept(listenfd, (SA *)&cliadr, &clen);
  if ((childid = fork()) == 0) { /* child proc */
    close(listenfd);
    process_request(connfd);
    exit(0);
  }
  close(connfd); /* parent closes conn sock */
}
```

Setup: Client

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
bzero(&servadr, sizeof(servadr));
servadr.sin_familiy = AF_INET;
servadr.sin_port = htons(SERV_PORT);
inet_pton(AF_INET, "localhost", &servadr.sin_addr);

connect(sockfd, (SA*)&servaddr, sizeof(servadr);
send_request(sockfd);
exit(0);
```

Issue #1: Server child zombie after client termination

Here's what happens client terminates.

1. CLIENT: calls `exit(0)`.
2. CLIENT: as part of client process termination, all open descriptors are closed, including **client socket**; the closing of client sock initiates TCP connection termination:
   (a) CLIENT: sends FIN to the server (as a result of client socket close)
   (b) SERVER: responds with ACK
   (c) first half of TCP connection termination finished
   (d) SERVER: now, server socket is in CLOSE_WAIT state
   (e) CLIENT: now, client socket is FIN_WAIT_2 state
3. SERVER: when it receives FIN, server child is blocked in a call to `readline`, and this returns 0
4. SERVER: server child terminates by calling `exit`
5. SERVER: all open file descriptors in *server child* closes; this causes final two segements of TCP connection termination to take place:
   (a) SERVER: child server sends FIN to client
   (b) CLIENT: client sends an ACK
   (c) now, TCP connection is fully terminated
   (d) CLIENT: client socket enters TIME_WAIT state
6. SERVER: **SIGCHILD** signal is sent to the parent when the server child terminates

To handle this, we add a **signal handler for SIGCHILD**

```
vid sig_child(int signo) {
  pid_t pid;
  int stat;
  pid = wait(&stat);
  printf("child %d terminated\", pid);
  return;
}
```

Issue #2: Multiple client connections terminated Let 10 clients are connected to 10 server children, respectively. When all 10 clients terminate, then 10 SIGCHILD signals are sent to their parent. However, one one signal is caught and signal handler is executed only once. **This is because UNIX signals are not queued.** So, we now have 8 or 9 zombies, depending on the timing.

The solution is to use `waitpid` instead of `wait`.

```
vid sig_child(int signo) {
  pid_t pid;
  int stat;
  while ((pid = waitpid(-1, &stat, WNOHANG)) > 0);
    printf("child %d terminated\", pid);
  return;
}
```

Issue #3: Connection aborts before accept returns

Server concurrency There are several options for achieving concurrency in servers (also for clients).

- **polling**
- **threads**
- **Unix select**

Server state Concurrent server needs to remember server-wide and per-state information such as:

- the number of clients,
- the state of connection for each client, and
- the state of each transaction/protocol for each client

For example, a transaction requires reading a file and put it into a buffer to send it to the client. Then, each such buffer is part of the per-client state. So, a large number of clients implies a large state. However, if a transaction can take indefinite amount of time to finish, we cannot bound the size of server state.

Stateless servers By maintaining a stateless server (i.e. no per-client state), we can avoid the potential problem of state explosion. One possibility is to make