1. First, write a simple server that can accept and serve TCP connections. You'll have to poke around the Internet a bit if you don't already know how to do this. Build this to serve exactly one request at a time; have each request be very simple, e.g., to get the current time of day.

TCP란 Transmission Control Protocol의 축약어로 컴퓨터가 다른 컴퓨터와 데이터 통신을 하기 위한 규칙이다. 만약 회선을 하나만 지정해준다면 통신을 중계해주던 곳이 망가지거나 중간에 선 하나가 잘못되면 통신이 끊어지게 된다. 따라서, 데이터를 안전하게 통신하기 위해 네트워크에서는 TCP를 사용한다.

사용법은 다음과 같다.

```
// 소켓을 만든다
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
    printf("socket creation failed...\n");
    exit(0);
}
else
    printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));
```

제일 먼저, 소켓을 만든다. 소켓이란 네트워크를 통해 데이터를 주고받기 위해 여는 창구이다.

데이터를 전송하기 위해서는 보내는 쪽과 받는 쪽 모두 소켓을 열어야 한다. 보내는 쪽이 소켓이라는 창구를 열고 데이터를 보내면 받는 쪽의 소켓을 통해 해당 프로세스에 데이터를 전달한다. 즉, 데이터를 주고받기 위한 문이라고 볼 수 있다.

```
// IP에 접근한다.
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

// IP와 소켓을 결합한다
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
    printf("socket bind failed...\n");
    exit(0);
}
else
    printf("Socket successfully binded..\n");
```

소켓을 만든 후에는 갈 곳의 주소를 적는다. 그 주소가 바로 Port(IP주소)이다. 그 후 Port와 소켓을 결합한다. 소켓이 사용하는 포트 주소가 다른 포트 주소와 중복된다면 해당 포트 주소로 데이터가 왔을 때 어느 소켓에서 그것을 처리해야 하는지 알기 어렵다. 따라서 포트주소가 중복되지 않도록 해당 소켓이 특정한 포트를 사용할 것이라고 알려주는 것이다.

```
// 소켓을 보낼 곳과 소통한다
if ((listen(sockfd, 5)) != 0) {
    printf("Listen failed...\n");
    exit(0);
}
else
    printf("Server listening..\n");
len = sizeof(cli);
```

그 후 데이터를 받을 곳에서 데이터를 보내라는 연락이 오기를 기다린다.

```
// 데이터를 보낸다
connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
    printf("server acccept failed...\n");
    exit(0);
}
else
    printf("server acccept the client...\n");
```

데이터를 보내라는 연락이 오면 데이터를 보낸다.

2. Now, add the select() interface. Build a main program that can accept multiple connections, and an event loop that checks which file descriptors have data on them, and then read and process those requests. Make sure to carefully test that you are using select() correctly.

그러나 이렇게 만든 소켓은 데이터를 적은 사람이 이용할 때에는 문제가 없겠지만 많은 사람이 이용하는 경우 순서가 밀려 데이터 처리에 오랜 시간이 걸린다는 단점이 있다. Client 수 만큼 스레드를 만들어 해결할 수 있겠지만 그 수가 엄청난 경우 스레드를 만드는 데에도 한계가 있다. 따라서 select 함수를 사용하여 여러 입출력 대상들을 검사하며 입출력이 생길 때마다 작업을 수행한다. 리눅스에서의 소켓은 파일과 동일하게 간주되기 때문에 파일 입출력 함수를 네트워크 상에서의 데이터 송수신에 사용할 수 있다.

따라서 파일 디스크립터를 통해 어떤 소켓이 데이터를 받았는지 확인할 수 있다.

fd_set* readfds

fdnum	0	1	2	3	4	5	6	7	8	9
bit	0	0	0	0	0	0	0	0	0	0

각각의 소켓(fdnum)은 파일처럼 관리되며 fdnum을 부여받는다. 만약 소켓이 데이터를 받는 등 변화가 생긴다면 bit을 1로 바꿔 변화가 있었음을 표시한다.

만약 3, 5번 소켓에 변화가 있었다면 fd_set은 다음과 같다.

fd_set* readfds

fdnum	0	1	2	3	4	5	6	7	8	9
bit	0	0	0	1	0	1	0	0	0	0

변화가 생겼음을 bit로 표시했으니 해당 어레이는 [0][0][1][0][1][0][0][0][0][0] 가 된다. 이 어레이를 for문을 돌며 살피면 3번과 5번 소켓에서 변화가 생겼음을 알 수 있다.

3. Next, let's make the requests a little more interesting, to mimic a simple web or file server. Each request should be to read the contents of a file (named in the request), and the server should respond by reading the file into a buffer, and then returning the contents to the client. Use the standard open(), read(), close() system calls to implement this feature. Be a little careful here: if you leave this running for a long time, someone may figure out how to use it to read all the files on your computer!

각각의 리퀘스트 과정은 파일을 buffer에 넣어서 open, read, close하고 client에게 보내는 것으로 이루어져 있다. 여러 client의 리퀘스트를 처리하기 위해 select()를 사용하여 응답이 오기 전까지 client가 다른 행동을 하는 것을 막은 채로 fd_set을 관찰하며 소켓에 변화가 생길 때까지 (응답이 발생할 때까지) client가 기다리도록 하는 것이다. 이러한 select()의 가장 큰 문제점은 여러 client 들이 동시에 접속하여 리퀘스트를 보냈을 경우, 다른 client의 리퀘스트들이 완료될 때까지 아무 것도 하지 못한 채 기다려야 한다는 것이다.

4. Now, instead of using standard I/O system calls, use the asynchronous I/O interfaces as described in the chapter. How hard was it to incorporate asynchronous interfaces into your program?

이러한 select의 단점은 client의 수가 커지면 커질수록 치명적이다. client들이 아무런 행동을 하지 못한 채 응답이 도착할 때까지 대기하고 있어야 하기 때문에 시간과 자원의 낭비가 발생한다. 이러한 문제를 해결하기 위해 asynchronous I/O를 사용한다. asynchronous I/O란 I/O 응답이 도착할 때까지 다른 CPU 작업을 할 수 있는 방법이다. I/O 작업이 이루어지는 동안 각 client는 대기하는 것이 아니라 다른 활동을 하며 응답이 오기를 기다릴 수 있다. 다른 활동을 하다가 I/O 작

업이 완료되면 callback을 호출 받아 응답이 도착했음을 알 수 있다. 이러한 방식으로 I/O 작업동안 다른 CPU 작업을 동시에 할 수 있다. 즉, client는 서버에게 리퀘스트를 보내고, 그 리퀘스트 응답이 오기 전까진 다른 일을 할 수 있다.

5. For fun, add some signal handling to your code. One common use of signals is to poke a server to reload some kind of configuration file, or take some other kind of administrative action. Perhaps one natural way to play around with this is to add a user-level file cache to your server, which stores recently accessed files. Implement a signal handler that clears the cache when the signal is sent to the server process.

6. Finally, we have the hard part: how can you tell if the effort to build an asynchronous, event-based approach are worth it? Can you create an experiment to show the benefits? Ho