Network Programming Laboratory
Winter 2018/2019

Session 4:
Advanced I/O Concepts

Janne Riihijärvi

RWTH Aachen University - Institute for Networked Systems

Contents

1 Overview **1**

2 Alternatives to Sequential Blocking I/O **2**
2.1 Non-blocking I/O 2
2.2 Using select() 2
2.3 Signals and signal handlers 3

3 Programming Tasks **4**

1 Overview

In this session we will cover alternative approaches to I/O issues with sockets. Until now the clients and servers developed in the laboratory sessions have accessed sockets in a sequential fashion. Usually a loop was written which does `recv()`, `recvfrom()` or `read()` from a socket, blocking until data arrives at the socket, followed by processing of the data, possibly sending back a reply, and starting the loop from the beginning. Using `fork()` it is possible to do this for multiple sockets at the same time, starting a new process per socket, each operating in a sequential manner.

While the above approach is sufficient for a large number of applications, there are alternatives that are important to know. This lab session will be dedicated towards exploring those alternatives, especially using `select()` to simultaneously monitor the state of a number of sockets (or, more generally, file descriptors) at the same time within a single thread.

2 Alternatives to Sequential Blocking I/O

In this section we describe briefly the main alternatives to sequential blocking I/O with sockets.

2.1 Non-blocking I/O

Conceptually simplest alternative to blocking I/O is to make the sockets used *non-blocking*. This causes calls like `recv()`, `recvfrom()` or `read()` to return immediately, whether there was data to be received or not. In case data is not available to read, the call returns -1, and the corresponding error number stored in the global variable `errno` is set to `EWOULDBLOCK`. The latter can be used to distinguish between a read from non-blocking file descriptor returning due to no data being available, or due to an actual error condition.

A socket (or other file descriptor type) is set to non-blocking state using the general purpose function call `fcntl()` (see `man 2 fcntl` for details), which has the following prototype:

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

Here `fd` is the file descriptor to be manipulated, with the rest of the arguments used to define the operation to be carried out. Setting `fd` to a non-blocking mode can be done by

```
fcntl(fd, F_SETFL, O_NONBLOCK);
```

It should be noted that using non-blocking sockets is not a particularly common technique, with `select()` offering much more general alternative as discussed below. However, it is occasionally used in practice, so the foundations of non-blocking operation are still useful to know.

2.2 Using select()

The idea behind `select()` is to offer a way to wait for multiple sockets simultaneously. For example, a particular server might have a large number of sockets opened, each corresponding to a connection with a client, together with the original socket listening for new connections. Using `select()` the server could block waiting for *any* of the sockets to have data or a new connection arriving, and then handle that before returning to wait for new data again.

The basic concept in using `select()` is that of *file descriptor sets*. These define the collections of file descriptors `select()` should wait for. The prototype of `select()` is

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Each of the arguments of type `fd_set` corresponds to such a file descriptor set, for which `select()` should wait until they become available for reading, writing or exceptions occur, respectively. Finally, one can give a timeout value after which the call should return even if nothing happens regarding the defined file

descriptor sets.

File descriptor sets are manipulated using the macros

```
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

described in more details on `man 2 select`. When the call returns, the file descriptor sets are modified to indicate which file descriptors actually changed status. Note that this usually implies that you have to restore the structure of the file descriptor sets before issuing `select()` again after processing the changes in the involved sockets.

The manual page of `select()` has a good example how it is used, and we recommend everyone to spend some time in the lab studying it.

2.3 Signals and signal handlers

The third I/O model we consider here is based on the concept of *signals*. These form one of the simplest form of interprocess communication (IPC) in UNIX-like systems, and are very similar to the concept of interrupt handlers ubiquitous on especially embedded systems. A process or the operating system kernel can send to another process a *signal* which simply is an asynchronous notification that an event has occurred. The type of the signal can be used to encode the type of event. A process can associate to a given signal type a *signal handler*, which is simply a function that gets called upon reception of the signal. An overview of signals in Linux can be found from `man 7 signal`.

Signals can be used in conjunction with sockets by using the `fcntl` function to set the socket into an asynchronous I/O model with command `O_ASYNC`. After this the signal `SIGIO` will be delivered to the process as soon as data becomes available for the socket.

For a given signal type the handler is set using either the old (and now deprecated) `signal()` or the more modern `sigaction()` call. Both of these can take as arguments a pointer to the signal handler function.

3 Programming Tasks

1. Using `select()` make a version of your webserver that is capable of handling multiple connections at a time. Your server should both support waiting for new connections as well as waiting for new requests arriving from sockets corresponding to already established connections. Finally, add also a support for getting commands from the keyboard. For example, if user presses `q` the server should quit, and pressing `s` should stop the server from accepting new connections.

2. Make a version of the UDP echo server using signals.