Computational Systems Engineering

Christoph Kirsch
University of Salzburg

3 Unit Graduate Course, Winter 2004/2005
Chapter 4: Socket Programming
(see also Stevens, Fenner, Rudoff: UNIX Network Programming, Volume 1)

History

- Berkeley Sockets API originated from the 4.2 BSD system in 1983
- Few API changes in 1990
- All networking code, kernel support and applications (FTP, telnet), independent of Unix license requirements
- Linux sockets have been implemented from scratch

Berkeley Sockets

- The Berkeley Sockets API comprises a library for developing applications that access a computer network
  - A Berkeley Socket is an endpoint for communication
  - We distinguish server and client sockets
  - Sockets are identified by file descriptors (which are integer values)

Server and Client API

- `socket()`: create a socket
- `read()`: read from a socket
- `write()`: write to a socket
- `close()`: close a socket

Client API

- `connect()`: connects a socket to a remote socket identified by an IP address and a port

Server API

- `bind()`: bind a socket to a port
- `listen()`: converts a socket to a server socket
- `accept()`: connects a server socket to a remote socket that tries to `connect()` to the server socket

Web Server

```
socket = socket();
bind(socket, myserver.com:80);
listen(socket);

while (true) {
  connection = accept(socket);

  request = read(connection);

  file = parse(request);

  page = read(file);

  write(page, connection);

  close(connection);
}
```

I/O Models

- Blocking I/O
- Nonblocking I/O
- I/O Multiplexing
- Signal-driven I/O
- Asynchronous I/O

Example: Input

- Wait for data to be ready
- Copy ready data from kernel to user space

Blocking I/O

- Default!
- `recvfrom()` blocks in the kernel until data is ready and has been copied to user space

Nonblocking I/O

- Do not put process to sleep but return an error instead if data is not ready
- `recvfrom()` blocks in the kernel until data has been copied to user space if data is ready

I/O Multiplexing

- Use two system calls, one to wait for data and one to copy data
- `select()` blocks in the kernel until data is ready (can wait for more than one descriptor)
- `recvfrom()` blocks in the kernel until data is ready and has been copied to user space

Signal-Driven I/O

- Get notified by a signal when data is ready
- `sigaction()` installs a signal handler that is invoked when data is ready
- `recvfrom()` blocks in the kernel until data is ready and has been copied to user space

Asynchronous I/O

- Get notified by a signal when data is ready and has been copied to user space
- `aio_read()` returns immediately. We get notified that the operation is complete, e.g., by a signal
- Signal-driven I/O tells us when an I/O operation can be *initiated*, asynchronous I/O tells us when an I/O operation is *complete*

Synchronous vs. Asynchronous I/O

- Synchronous I/O causes the requesting process to be blocked until the I/O operation is complete
- Asynchronous I/O does not cause the requesting process to be blocked
- Blocking, nonblocking, I/O multiplexing, and signal-driven I/O are synchronous I/O

Appointments

- Calendars: Partial or total ordering?
- Clocks: Real time or CPU time?

Jobs

- System Administration: Peter
- Website: Harald
- Benchmarking: Max
- Webserver development: Claudiu
- Library development: Claudiu