Advanced Socket Capabilities
Socket I/O

A buffer space issue that occurs in commercial programs that we haven't had to handled, because our homeworks have been student size.

Issue: TCP can run out of buffer space.
Only a limited amount of system/network buffer space is made allocated by the system; if you exceed that, the write may fail.

If the data transfers to the server are large, the server can become clogged and the client can stop at the `write`.

If the client is using blocking I/O, it can pause at the `write` forever.

Example

Client:

```
// send data to server
write(ssock,buf,sizeof(buf));
// read config
n = read(ssock,buf,sizeof(buf));
// ...
// read reply
n = read(ssock,buf,sizeof(buf));
```

Server:

```
// send config
write(ssock,buf,sizeof(buf));
// read client data
n = read(ssock,buf,sizeof(buf));
// ...
// read to client
n = write(ssock,buf,sizeof(buf));
```

Solution 0: avoid having both client and server do writes. May not be possible in context of some programs.

Solution 1: the system allocates the default size, you can change that. (`man 7 socket`)

```
int bufferSize = 8388608; //8M
// Also can do with SO_RCVBUF
int buffSizeResult = setsockopt(ssock, SOL_SOCKET,
  SO_SNDBUF, &bufferSize, sizeof(bufferSize));
int getsize;  // See what it gave us.
int getsizesize = sizeof(getsize);
int result = getsockopt(ssock, SOL_SOCKET,
   SO_SNDBUF, &getsize, &getsizesize);
```

The maximum amount that can be allocated is set by the `sysctl`s `rmem_max` and `wmem_max`.

Note: on a buffer back up, the data is either in 'your' send buffer or 'his' receive buffer. With a single process concurrent server (such as chat), increasing the send buffer distributes the data backup amoung the clients; increasing the receive buffer doesn't scale with the number of clients.

Note: You can still clog, it just takes bigger writes.

Solution 2: system buffer space will be freed when someone does a read, so trying again later works.

More general idea, I/O is normally blocking,
the program waits until the read/write is done.
You can set I/O to be non-blocking
If the I/O would block an error (< `0`) is returned and `errno` is set to `EAGAIN`

The `fcntl` command is used to set the non-blocking flag on a socket (or any descriptor).

Example (spinlock loop):

```
fcntl(ssock,F_SETFL,O_NONBLOCK);
int result;
while (1){
   result = read(ssock,buf,sizeof(buf));
   if (result < 0){
     if (errno != EAGAIN) {
       errexit();
     }
     else sleep(1); //blocked, wait and try again
   }
   else break; // good read
}
```

`select` is a better way to go with reads.

Non-blocking writes

```
int sendsize = 999;
int strt = 0;
int result;
fcntl(ssock,F_SETFL,O_NONBLOCK);
while (sendsize > 0) {
  result = write(ssock,&(buf[strt]),sendsize);
  if (result < 0){
     if (errno != EAGAIN) {
         errexit();
     }
  }
  else
  {
    strt += result;
    sendsize -=result;
  }
}
```

Non-blocking write:
On `EAGAIN` we sent nothing, send again.
On `result>=0` might have written less than `sendsize` so track (`strt`) where the unsent portion of the buffer begins and send the remainder of the buffer on later writes.

Interrupt driven I/O

You can set a descriptor so a signal is sent when I/O is available.

```
// Send me an SIGIO signal on I/O events
fcntl(ssock,F_SETFL,O_ASYNC);
// instead of SIGIO, send this sig
fcntl(ssock,F_SETSIG,SIGHUP);
```

Asynchronous I/O must use the sigaction form of the signal handler.

```
void ioHdlr(int sig);
struct sigaction act;
act.sa_handler = ioHdlr;
act.sa_mask = SIGHUP;
act.sa_flags = 0;
sigaction(SIGIO,&act,NULL);
```

Call the function `ioHdlr` when a `SIGIO` happens.
Mask `SIGHUP` (in addition to `SIGIO`).

Do the `sigaction` before the `fcntl`.

Simplistic handler:

```
void ioHdlr(int sig) {
  read(ssock, buf,sizeof(buf));
};
```

Unix (local) domain sockets

A non-network socket implementation is available under Unix.
These are called Unix domain or local domain sockets.
They are invisible from outside your machine.

`AF_LOCAL` or `AF_UNIX`
`PF_LOCAL` or `PF_UNIX`

```
#include <sys/un.h>
int msock = socket(PF_LOCAL, SOCK_STREAM, 0);
struct sockaddr_un serveraddress;
serveraddress.sun_family = AF_LOCAL;
strcpy(serveraddress.sun_path,"/tmp/socketfile");
bind(msock, (struct sockaddr *)&serveraddress,
  SUN_LEN(&serveraddress));
```

The Unix version of `sockaddr` is used. A path (file name) replaces the internet address/port pair.

Client side:

```
int ssock = socket(PF_LOCAL, SOCK_STREAM, 0);
serveraddress.sun_family = AF_LOCAL;
strcpy(serveraddress.sun_path,"/tmp/socketfile");
connect(ssock,(struct sockaddr *)&serveraddress,
  SUN_LEN(&serveraddress));
```

Unix domain semantics

If the "socket" file exists it is assumed that a server is running.

1) You cannot bind to an existing file.

2) After the socket close, the `unlink` system call still needs to be used to remove the file.

3) A crashed server leaves a dangling file, which needs to be removed by hand (bad OS design on this one).

The sample code is "primitive",
the `bind`, `connect` and other commands should have
`if (... < 0) errexit(...);`
code attached.
Otherwise you won't know what went wrong.

Other IP protocols

These are available although implementations may be limited

`socket(PF_LOCAL,SOCK_SEQPACKET,0)`

TCP with boundaries preserved.
Your `read` be big enough to read the whole packet.
Not available with `AF_INET`.

`socket(PF_INET,SOCK_RDM,0)`

Reliable UDP.
Packets delivery guaranteed.
Order of delivery is not guaranteed.