MDP Guide to Network Programming

Massimo Di Pierro

October 11, 2007

Abstract

This notes are a work in progress, please report any error.

Contents

1 Introduction

I use both C and C++. Compile with

```
g++ filename.cpp -o filename.exe -lpthread
```

1.1 Header Files

I will assume the following headers and included

```
// BEGIN FILE: mdp_all.h
// C headers
#include "sys/types.h"
#include "sys/socket.h"
#include "sys/time.h"
#include "time.h"
#include "netinet/in.h"
#include "arpa/inet.h"
#include "errno.h"
#include "fcntl.h"
#include "netdb.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "sys/stat.h"
#include "sys/uio.h"
#include "unistd.h"
#include "sys/wait.h"
#include "sys/un.h"
#include "sys/select.h"
#include "poll.h"
#include "strings.h"
#include "pthread.h"

// C++ headers and STL headers
#include "iostream"
#include "string"
#include "vector"
```

```
#include "deque"
#include "map"
using namespace std;

#ifndef HAVE_INET_NTOP
#define inet_ntop(a,b) inet_ntoa(b)
#define inet_pton(a,b,c) inet_aton(b,c)
#endif

void exit_message(int en, string message) {
  cerr << "FROM PROCESS PID: " << getpid() << endl;
  cerr << "CHILD OF PROCESS PID: " << getppid() << endl;
  cerr << "FATAL ERROR: " << message << endl;
  cerr << "EXITING WITH ERROR NUMBER: " << en << endl;
  exit(en);
}
```

In all the examples below we will use exit_message to notify the user when an error occurs. In real life we may want to deal with error is some more sophisticated way.

1.2 File descriptors

In Unix/Linux everything is a file (a file is a file, a socket is a file, a device is a file). From the point of view of a program a file (including a socket or a device) are identified by a single integer number called file descriptor. For example to write a buffer to the file "myfile.dat":

```
int main() {
   int fd=open("myfile.dat",O_WRONLY);
   if(fd<0)
      exit_message(1,"unable to open file");
   char buffer[128];
   int buffer_size=128;
   if(write(fd,buffer,buffer_size)!=buffer_size)
      exit_message(1,"write failuer");
   close(fd);
}
```

In the example fd is the file descriptor associated to the file. If you have a pointer to file (FILE* fp) and not an integer file descriptor you can get it with

```
fd=fileno(fp);
```

2 Dealing with IP addresses

Relevant commands:

- `htonl`: converts 32bits int from host to network
- `ntohl`: converts 32bits int from network to host
- `htons`: converts 16bits int from host to network
- `ntohs`: converts 16bits int from network to host
- `inet_pton`: converts IP address (char[]) to 32bits network
- `inet_aton`: same as above (only ipv4)
- `inet_ntop`: converts 32bits network to IP address (char[])
- `inet_ntoa`: same as above (only ipv4)
- `inet_addr`: do not use, problems with broadcasting.
- `gethostbyname`: get the ip address from the hostname

2.1 Representation of addresses

There are at least four representations for an address:

- Host Name: www.yahoo.com
- IP Address: 216.109.118.77
- 32bits long network: $216 \cdot 2^{24} + 109 \cdot 2^{16} + 118 \cdot 2^8 + 77 = 3631052365$
- 32bits long host: the host representation of the long integer above.

Functions that deal with socket need to know: the address, the port number and the socket type. The structure `sockaddr_in` has member variables to hold the socket type, the port number and the ip address in 32bits long network representation.

```
struct sockaddr_in  {
  short           sin_family;  // address family
  u_short         sin_port;    // socket port
  struct in_addr  sin_addr;    // address (32bits net)
  char            sin_zero[8]; // padding
};
```

2.2 Storing an IP address (ipv4)

```
// input variables:
char ip_address[16]="216.109.118.77";
int  port_numer=80;
// ouput variable:
struct sockaddr_in address;
// store IP address in 'address':
memset(&address,0,sizeof(address));
address.sin_family=AF_INET;
address.sin_port=htons(port_number);
if(inet_pton(AF_INET,ip_address,&address.sin_addr)<=0)
  exit_message(1,"inet_pton");
```

2.3 Retrieving an IP address

```
// input variable:
struct sockaddr_in address; // as returned by recvfrom
// output variables:
char ip_address[16];
int  port_numer=80;
// retrieve IP address from 'address':
port_number=ntohs(address.sin_port);
strncpy(ip_address,inet_ntop(AF_INET,address.sin_addr),16);
```

2.4 Convert host names to IP address

```
// output variable:
char ip_address[16];
// get host ip:
struct hostent* hptr=gethostbyname(name.c_str());
if(hptr==0)
  exit_message(1,"gethostbyname, invalid address");
if(hptr->h_length!=4)
  exit_message(1,"gethostbyname, not supported");
strncpy(ip_address,
        inet_ntop(family,*((struct in_addr*)
                           *(hptr->h_addr_list))),16);
```

3 TCP Sockets (SOCK_STREAM)

TCP is a reliable protocol. A connection is established between a client and a server. When the client sends something to the server the server acknowledges receiving and vice versa. Telnet, ssh, tcp, pop, imap are all based on TCP.

Relevant commands:

- `socket`: creates a socket
- `bind`: binds socket to a port
- `listen`: a server listen for connection
- `accept`: a server accepts a client
- `connect`: a client attempt to connect to a server
- `recv`: read from a TCP socket
- `read`: same as above but default flags (avoid)
- `send`: write to a TCP socket
- `write`: same as above but default flags (avoid)
- `setsockopt`: set socket options.

3.1 TCP server

Any TCP server performs the following operations in the same order:

- Create socket (family AF_NET for ipv4 and AF_INET6 for ipv6)
- Bind socket to local port
- *Optional:* set socket options (using setsockopt or fcntl)
- Listen for connection
- *Optional:* fork or create_pthread
- Accept a connection (blocking unless otherwise specified)
- Read/write or recv/send from/to socket
- Close socket

```
int sfd=socket(AF_INET,SOCK_STREAM,0);
if(sfd<0) exit_message(1,"socket");
int  port_numer=80; // port you want to server from
int  maxn=10;       // max number of connections
struct sockaddr_in address;
memset(&address,0,sizeof(address));
address.sin_family=AF_INET;
address.sin_port=htons(port_number);
address.sin_addr.s_addr=htonl(INADDR_ANY);
if(bind(sfd,(struct sockaddr*)&address,sizeof(address)))
  exit_message(1,"bind");
if(listen(sfd,maxc))
  exit_message(1,"bind, too many connections?");
struct sockaddr_in client_address;
socklen_t length=sizeof(client_address);
int cfd=accept(sfd,(struct sockaddr*)&client_address,&length);
if(cfd<0) exit_message(1,"accept");
char buffer[128];
int buffer_size;
if(send(cfd,buffer,buffer_size,0)!=buffer_size)
  exit_message(1,"send");
```

```
if(recv(cfd,buffer,buffer_size,0)!=buffer_size)
  exit_message(1,"recv");
close(cfd); // close connection to client
close(sfd); // stop listening
```

NOTE: We bind to local address INADDR_ANY because we assume we have only one network card. If this is not the case we have to specify the IP address associated to the network card from which we wish to serve.

3.2 TCP client

Any TCP client performs the following operations in the same order:

- Create socket (family AF_NET for ipv4 and AF_INET6 for ipv6)
- *Optional:* set socket options (setsockopt or fcntl)
- Connect to server
- read/write or recv/send from/to socket
- Close socket

```
int sfd=socket(AF_INET,SOCK_STREAM,0);
if(sfd<0) exit_message(1,"bind");
// fill address of remote server
memset(&address,0,sizeof(address));
address.sin_family=AF_INET;
address.sin_port=htons(port_number);
if(inet_pton(AF_INET,ip_address,&address.sin_addr)<=0)
  exit_message(1,"inet_pton");
int ce=connect(sfd,(struct sockaddr*)&address,
   sizeof(address));
if(ce!=0 && errono==ECONNREFUSED) exit_message(1,"connection refused");
if(ce!=0 && errono==ETIMEDOUT) exit_message(1,"timeout");
// else if ce==0
char buffer[128];
int buffer_size;
if(recv(sfd,buffer,buffer_size,0)!=buffer_size)
  exit_message(1,"recv");
```

```
if(send(sfd,buffer,buffer_size,0)!=buffer_size)
  exit_message(1,"send");
close(sfd); // close connection to client
```

3.3 Staying alive

When TCP connections can be idle for long time they may timeout. To prevent timeout, after creating a socket, set the SO_KEEPALIVE option of the socket to ON.

```
static int alive_on=1;
setsockopt(sfd,SOL_SOCKET,SO_KEEPALIVE,
           &alive_on,sizeof(alive_on));
```

One should check if the return value of setsockopt is different from zero and eventually return error.

3.4 How to copy a socket

It is unsafe to copy a file descriptor (that may be associated to a socket) in the following way

```
int sfd2=sfd;
```

because one may lose track of the copies and accidentally close the same file descriptor twice, close(sfd) and close(sfd2). This would be incorrect. To prevent the problem, always copy a file descriptor using the `dup` function:

```
int sfd2=dup(sfd);
```

Now one can safely close(sfd) and close(sfd2) as if they were different file descriptors, even if they are connected to the same file or socket.

3.5 Dealing with SIGHUP

When a process tries to perform IO on a socket while the connection is closed by the peer, the process receives a SIGHUP signal. If this happens one may want to try to connect again or just abort. See examples below on how to catch a signal.

3.6 Terminators and marshalling

If a host (let's say the client) sends data to a peer (let's say the server) how does the server knows the size of the data to read? There are two solutions to the problem:

- Using a terminator character or a sequence of characters as a terminator sequence.
- Using marshalling, i.e. send the size of the data as an int (32 bits) and then the data.

Different solutions correspond to different protocols. Typically for text based protocols (such as HTML and XML) the former solution is adopted. For ASCII based protocols (such as ASCII transfer in FTP) the latter solution is adopted.

4 UDP Sockets (SOCK_DGRAM)

UDP is an unreliable protocol. No connection is established between a client and a server. When the client sends something to the server there is no guarantee that the message has been received, and vice versa. UDP is typically used in internet radio/video broadcasting or for some other type of broadcasting when the failure rate is considered acceptable (for example, skipping a video frame once in a while).

Relevant commands:

- `sendto`: send a datagram to a remote host
- `recvfrom`: receive a datagram from a remote host
- `fcntl`: the most general way to set options on a socket or any file description
- `poll`: wait from something to happen on a set of file descriptors
- `select`: same as poll but old and cumbersome
- `sigemptyset, sigaddset, sigaction, sigprocmask`: deal with signals

4.1 Sending a datagram with sendto

To send a UDP datagram contained in buffer

- Create Socket
- Bind socket to port (port 0 if you do not care)
- Store destination (remote) address
- call sendto

```
sfd=socket(AF_INET, SOCK_DGRAM, 0);
// if sfd<0 error
// set local port and bind to it
local_address.sin_family = AF_INET;
local_address.sin_port =htons(local_port);
local_address.sin_addr.s_addr = INADDR_ANY;
if(bind(sfd,(const sockaddr*)&local_address,
   sizeof(local_address))<0)
  exit_message(1,"bind");
// set remote address and store it
remote_address.sin_family = AF_INET;
remote_address.sin_port = htons(remote_port_number);
if(inet_pton(AF_INET,remote_ip_address,
   &remote_address.sin_addr)<=0)
  exit_message(1,"inet_pton");
// send buffer to remote address
char buffer[128];
int buffer_size=128;
sendto(sfd,buffer,buffer_size,0,
       (const sockaddr*)&remote_address,sizeof(remote_address));
// close socket
close(sfd);
```

4.2 Receiving a datagram with recvfrom

To receive a UDP datagram and store it in buffer

- Create Socket

- Bind socket to port (port 0 if you do not care)
- Call recvfrom
- *Optional:* retrieve destination (remote) address

```
sfd=socket(AF_INET, SOCK_DGRAM, 0);
// if sfd<0 error
// set local port and bind to it
local_address.sin_family = AF_INET;
local_address.sin_port =htons(local_port);
local_address.sin_addr.s_addr = INADDR_ANY;
if(bind(sfd,(const sockaddr*)&local_address,
   sizeof(local_address))<0)
  exit_message(1,"bind");
socklet_t addrsize=sizeof(remote_address);
char buffer[128];
int buffer_size=128;
recvfrom(sfd,buffer,buffer_size,0,
         (const sockaddr*)&remote_address,&addrsize);
// Optional retrieve remote address
int remote_port_number;
char remote_ip_address[16];
remote_port_number=ntohs(remote_address.sin_port);
strncpy(remote_ip_address,
        inet_ntop(AF_INET,remote_address.sin_addr),16);
// close socket
close(sfd);
```

4.3 Non-blocking IO

Non blocking IO works for both UDP and TCP, but it is more commonly used for UDP.

To set a socket nonblocking, immediately after creating the socket

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

When one does this than one should put the recv/recvfrom in a loop and periodically try to read from the socket. If there is nothing to read

from the socket recv/recvfrom will return with an error status different from EWOULDBLOCK. If recv/recvfrom return EWOULDBLOCK than a socket error has occurred.

```
while(1) {
   int e=recvfrom(sfd,buffer,buffer_size,0,
                  (struct sockaddr*)&address,&address_size);
   if(e<0 && errono=EWOULDBLOCK)
      exit_message(1,"recvfrom->EWOULDBLOCK");
   // do something such as sleep(1) and try again
   ...
}
```

4.4 Asynchronous IO

Asynchronous IO works for both UDP and TCP, but it is more commonly used for UDP. Asynchronous IO means that the socket is set to non-blocking and the OS is requested to notify the process when there is something to read from the socket. The notification is a SIGIO signal.

To set a socket asynchronous, immediately after creating the socket

```
fcntl(sfd,F_SETFL,O_NONBLOCK);
fcntl(sfd,F_SETOWN,getpid());
fcntl(sfd,F_SETFL,FASYNC);
```

If something happens to the socket the process itself, getpid(), receives a SIGIO signal. To catch the signal:

```
void sigio_handler(int singnum) {
   // take action
}

...
struct sigaction action;
action.sa_handler=sigio_handler;
if(sigemptyset(&action.sa_mask)<0)
  exit_message(1,"sigemptyset");
if(sigaddset(&action.sa_mask,SIGIO))
  exit_message(1,"sigaddset");
```

```
if(sigaction(SIGIO,&action,NULL)<0)
  exit_message(1,"sigaction");
```

4.5 Using poll

The poll function is a blocking function that allows execution to wait for something to happen to any of a set of the file descriptors.

Given two file descriptors (sfda and sfdb)

```
struct pollfd fds[2];
fds[0].fd=sfda;
fds[0].events=POLLRDNORM | POLLERR;
fds[1].fd=sfdb;
fds[1].events=POLLRDNORM | POLLERR;

while(1) {
    int n=poll(fds,2,-1); // -1 means poll forever
    if(n==-1) exit_message(1,"poll");
    if(n==0) exit_message(1,"timeout");
    if(fds[0].revents & POLLRDNORM) {
       // something to read from sfda, deal with it
    }
    if(fds[0].revents & POLLERR) {
       // error from sfda, deal with it
    }
    if(fds[1].revents & POLLRDNORM) {
       // something to read from sfdb, deal with it
    }
    if(fds[1].revents & POLLERR) {
       // error from sfda, deal with it
    }
}
```

The lines:

```
fds[0].fd=sfda;
fds[0].events=POLLRDNORM | POLLERR;
```

indicate that we are registering `sfda` with poll and poll will to return when POLLRDNORM (something to read) or POLLERR (socket error).

4.6 Using select

4.7 UDP broadcasting with sendto

Broadcasting is a way to send a single datagram to every machine in a subnetwork. All machines that listen for the broadcast will receive the same datagram. Each network has a broadcast address. To get the broadcast address

```
> ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:51:9D:F7:AC:A0
          inet addr:140.192.36.40  Bcast:140.192.36.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:162132 errors:0 dropped:0 overruns:0 frame:0
          TX packets:78186 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:69309232 (66.0 MiB)  TX bytes:9676649 (9.2 MiB)
          Interrupt:18 Base address:0x9000 Memory:fc000000-fc020000
```

Notice:

```
Bcast:140.192.36.255
```

That is by sending a datagram to this address all nodes in the subnetwork will receive the dgram.

To enable broadcasting on a socket, after creating the socket, before sendto

```
static int bc_on=1;
setsockopt(sfd,SOL_SOCKET,SO_BROADCAST,
           &bc_on,sizeof(bc_on));
```

4.8 UDP broadcasting and recvfrom

In order be able to be able to receive broadcasted datagrams, after creating the socket

```
static int reuse_on=1;
setsockopt(sfd,SOL_SOCKET,SO_REUSEADDR,
           &reuse_on,sizeof(reuse_on));
```

also, very importantly, bind the socket to the broadcasting address (bc_ip_address, bc_port_number):

```
// set local port and bind to it
struct sockaddr_in bc_address;
bc_address.sin_family = AF_INET;
bc_address.sin_port =htons(bc_port);
if(inet_pton(AF_INET,bc_ip_address,
   &bc_address.sin_addr)<=0)
  exit_message(1,"inet_pton");
if(bind(sfd,(const sockaddr*)&bc_address,
   sizeof(local_address))<0)
  exit_message(1,"bind");
```

Routers may reroute broadcasted datagrams. The old way to broadcast to a subnetwork was to sendto 255.255.255.255. If you use this address do not use the function inet_addr(). In general, if you can, avoid using 255.255.255.255.

4.9 UDP multicasting and sendto

Multicasting is based on the same idea of broadcasting but it allows a program to send one datagram to a set of hosts, including hosts across the sub-network. Multicasting is essentially a method to implement a virtual network (referred to as a "group") and broadcast within the group. Every IP multicast has a group address. It is not necessary to be a member of the group to send a datagram to the group.

Multicast is not supported by old routers and is an optional feature in IPV4.

To send a multicast datagram one sends a regular datagram to the multicast address. The only difference is that one has to set the Time-To-Live option of the datagram using the setsockopt

```
unsigned char ttl=1; // 0,1,32,64,128 or 255
setsockopt(sfd,IPPROTO_IP,IP_MULTICAST_TTL,&ttl,sizeof(ttl));
```

If ttl is set to 0 the multicast is restricted to the host; if ttl is set to 1 the multicast is restricted to the subnet; if ttl is set to 32 to the site, 64 to the region, 128 to the continent and 255 to the world. Yes, you sent ttl to 255 and you are sending the datagram to everybody.

Note that multicast address are class D addresses (range 224.0.0.0 to 239.255.255.255).

4.10 UDP multicast and recvfrom

To receive a multicast address one has:

- to be in the are reached by the datagram (in the same subnet if tt was set to 1 by the sender);
- join the group (identified by a multicast_ip_address and a multicast_port_number).

One joins the group by

```
// set socket reusable
static int reuse_on=1;
setsockopt(sfd,SOL_SOCKET,SO_REUSEADDR,&reuse_on,sizeof(reuse_on));
// setup local address and bind to port
local_address.sin_family=AF_INET;
local_address.sin_port=htons(multicast_port_number);
local_address.sin_port.s_addr=htonl(INADDR_ANY);
if(bind(sfd,(struct sockaddr*)&local_address,
   sizeof(local_address))<0)
  exit_message(1,"bind");
// request kernel to join group
struct ip_mreq mreq;
mreq.imr_multiaddr.s_addr=inet_addr(multicast_ip_address);
mreq.imr_interface.s_addr=htonl(INADDR_ANY);
setsockopt(sfd,IPPROTO_IP,IP_ADD_MEMBERSHIP,&mreq,sizeof(mreq));
```

Now you recvfrom in the usual way.

Note that a multicast protocol is identified by a port (to which the clients/receivers must bind) and a multicast address (the same address used by the server to multicast a message and used by the clients/receivers to join the group).

5 Signals, Processes and Threads

Relevant commands:

- `alarm`: sets an alarm
- `fork`: fork a process
- `system`: executes a shell comand
- `execl`: repleces process with a shell comand
- `pthread_create, pthread_detach, pthread_join, pthread_cancel`: threads
- `getpeerbyname`: get the address and port of the peer connected to a socket (typically used in deamon).

5.1 Using Alarms

Alarm is a system to ask the OS to send us a signal SIGALRM after xxx seconds in time.

To set an alarm:

```
// create handler
void alarm_handler(int signum) {
   // this is called when the alarm occurs
   // do something
}


...
// register signal SIGALRM
struct sigaction action;
action.sa_handler=alarm_handler;
if(sigemptyset(&action.sa_mask)<0)
  exit_message(1,"sigemptyset");
if(sigaddset(&action.sa_mask,SIGALRM))
  exit_message(1,"sigaddset");
if(sigaction(SIGALRM,&action,NULL)<0)
  exit_message(1,"sigaction");
// set alarm
int seconds=5
alarm(seconds);
// do something while you wait for the alarm
```

```
...
```

Note that all blocking functions return when the alarm occurs. Each of them may return in a different way. Do not set alarms that may go off during console IO functions.

5.2 Ignore Signals

To ignore a signal, just register the signal with the macro SIG_IGN as handler.

```
...
// ignore signal SIGALRM
struct sigaction action;
action.sa_handler=SIG_IGN;
if(sigemptyset(&action.sa_mask)<0)
  exit_message(1,"sigemptyset");
if(sigaddset(&action.sa_mask,SIGALRM))
  exit_message(1,"sigaddset");
if(sigaction(SIGALRM,&action,NULL)<0)
  exit_message(1,"sigaction");
...
```

5.3 Blocking a signal

Sometimes you need to block signals. For example, if you set an alarm and you perform IO you may want to block the alarm during the IO. In the same fashion if you are using signal driven IO and you expect a SIGIO you may want to block the SIGIO during some other IO. The process of blocking one or more signals usually works as follows:

- Register the present signal mask
- Block all signals that need to be blocked
- Perform the operation (...) that should not be interrupted by the signal
- Check if a signal is pending and take appropriate action
- Restore the initial signal mask

```
// register present signal mask in oset
// select new signal mask in set
sigset_t set, oset;
if(sigemptyset(&set)<0)
  exit_message(1,"sigemptyset");
// add one or more signals to block
if(sigaddset(&set,SIGALRM)<0)
  exit_message(1,"sigaddset");
sigprocmask(SIG_BLOCK,&set,&oset);
// perform the operation in question
...
// check if signal is pending
sigset_t pending;
sigpending(&pending);
// check for each possible pending signal
if(sigismember(&pending,SIGALRM) {
  // deal with pending signal
}
// restore signal mask (IMPORTANT)
sigprocmask(SIG_BLOCK,&oset,NULL);
```

5.4 Creating a process

The simplest way to do concurrent programming is to create a child process that runs in parallel with the parent process. For example, one process does socket IO and another process does console IO at the same time. The console IO is not blocked by the blocking socket IO functions.

To create a child process:

```
pid=fork();
if(pid<0) {
  exit_message(1,"fork");
} else if(pid) {
   // parent process
   // (pid contains the pid of child)
} else {
   // child process
}
```

```
// code executed by both parent and child
```

This is okay. However, interprocess communications may be difficult, processes may become zombies and you have to use signals. Programs that fork, if written well, are more complex than programs with threads.

5.5 Killing zombies

When a child process dies before the parent, it sends a signal SIGCHLD to the parent and becomes a zombie. The zombie status means that the OS does not clean-up the memory used by the process and is waiting for the parent to authorize the cleanup. To prevent a child from becoming a zombie the parent should kill the child when it receives SIGCHLD. To achieve this, before forking the parent you should:

```
void zombie_handler(int s) {
  pid_t pid;
  int status;
  pid=wait(&status);
  // pid is the pid of dead child
  // status holds the return error of the dead child
}
...
// before forking register handler
struct sigaction action;
action.sa_handler=zombie_handler;
if(sigemptyset(&action.sa_mask)<0)
  exit_message(1,"sigemptyset");
if(sigaddset(&action.sa_mask,SIGCHLD))
  exit_message(1,"sigaddset");
if(sigaction(SIGCHLD,&action,NULL)<0)
  exit_message(1,"sigaction");
```

5.6 Creating a deamon process (fork twice)

A deamon is a process not connected to console and with no parent shell. To detach a process from shell fork twice.

```
pid=fork()
```

```
if(pid<0) exit_message(1,"fork");
if(pid>0) exit(0);
pid=fork()
if(pid<0) exit_message(1,"fork");
if(pid>0) exit(0);
// we are now detached from shell
```

To close the console IO close stdin, stderr and stdout.

```
close(fileno(stdin));
close(fileno(stderr));
close(fileno(stdout));
```

5.7 Running commands withing programs: system and execl

Sometimes you want to run a shell command within a program. For example, to list all running processes,

```
// do something
...
system("/usr/bin/ps aux");
// do something else
...
```

To perform the same operation and substitute the call ps for the present process:

```
// do something
...
execl("/usr/bin/ps","aux",0); // all arguments ended by 0
// it never makes it here
...
```

5.8 Creating a deamon process for inetd

The inetd is a system deamon that accepts connection on multiple ports to serve a set of registered services (ssh, tcp, etc.). When a connection occurs a new process is forked, the socket is renamed 0 and the proper application is execl-ed. Here is a service (built for inetd) that sends to the connected peer the peer's ip address:

```
int main(int argc, char** argv) {
  string message="You are ";
  struct sockaddr_in remote;
  int addrlen;
  getpeerbyname(0,(struct sockaddr*) remote, (socklen_t*) &addrlen);
  message=message+inet_ntop(AF_INET,remote.sin_addr);
  send(0,message.c_str(),message.length(),0);
  close(0);
}
```

The above process did not create the socket itself, nor did it call bind, listen and accept. It is started by inetd with a peer already connected to the socket with `sfd=0`. To determine the ip address, port and protocol of the peer connected to 0, the program needs to call `getpeerbyname(0,...)`.

After a valid inetd service is created and compiled (let's say in `/usr/local/bin/myservice.exe`) the service has to be registered with inetd. This is a 3-step process:

- Pick a port for the service that does no conflict with other services (let's say 1234).
- Tell inetd to listen from that port.
- Associtate you compiled program to the service.
- Restart inetd.

Step 1 implies looking into /etc/services and finding an unused port number.

Step 2 implies editing the file /etc/services and adding a new line/entry:

```
\texttt{%myservicename 1234/tcp}
```

Step 3 imples editing the file /etc/inetd.conf and adding a new line/entry:

```
myservicename stream tcp nowait nobody /usr/local/bin/myservice.exe
```

5.9 Locking a file

When one writes to a file one may want to lock the file so that no other process can write to the same file. For an integer file descriptor fd:

```
// lock fd for writing
fcntl(fd,F_SETLK,F_WRLCK);
// use fd
...
// unlock fd
fcntl(fd,F_SETLK,F_UNLCK);
```

To lock for reading F_RDLCK and to lock for both reading and writing F_RWLCK.

5.10 Getting file info

Use the command "stat" as follows:

```
struct stat file_info;
stat("path/myfile.dat",&file_info);
```

Now struct stat file_info contans the following information about the file:

```
struct stat {
    dev_t    st_dev;    /* device inode resides on */
    ino_t    st_ino;    /* inode's number */
    mode_t   st_mode;   /* inode protection mode */
    nlink_t  st_nlink;  /* number or hard links to the file */
    uid_t    st_uid;    /* user-id of owner */
    gid_t    st_gid;    /* group-id of owner */
    dev_t    st_rdev;   /* device type, for special file inode */
    struct timespec st_atimespec;  /* time of last access */
    struct timespec st_mtimespec;  /* time of last data modification */
    struct timespec st_ctimespec;  /* time of last file status change */
    off_t    st_size;   /* file size, in bytes */
    quad_t   st_blocks; /* blocks allocated for file */
    u_long   st_blksize;/* optimal file sys I/O ops blocksize */
    u_long   st_flags;  /* user defined flags for file */
    u_long   st_gen;    /* file generation number */
};
```

5.11 Creating threads

This is an example that uses two threads.

```
void* f(void* p) {
  sleep(2);
  cerr << "Hello from a thread\n";
  pthread_exit(NULL);
  return 0;
}

int main() {
  pthread_t thread_number;
  pthread_attr_t thread_attr;
  pthread_attr_init(&thread_attr);
  pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_JOINABLE);

  cout << "Starting thread\n";

  if(pthread_create(&thread_number, &thread_attr, f, (void*) NULL))
    exit_message(1,"pthread_create: Unable to create thread");

  cout << "Thread started. waiting ...\n";

  int status;
  if(pthread_join(thread_number,(void**) &status))
    exit_message(1,"pthread_join: Unable to join thread");

  cout << "Thread joined\n";
}
```

The advantage of two threads over two processes is that threads share the same memory and see the same variables. This makes communication between threads easier than communication between processes. Because they share the same memory one has to prevent two threads from writing to the same variable simultaneously. To avoid this, use the mutex functions. They work very much like a file locking but they lock variables, not files.

6 Appendix A - Working Examples

6.1 Gethostbyname

This program returns the IP address of a host.

```
// BEGIN FILE: gethostbyname.cpp
#include "mdp_all.h"
int main(int argc, char** argv) {
  if(argc<2) {
    cout << "usage: name host_name\n";
    exit(1);
  } else {

    struct hostent* hptr=gethostbyname(argv[1]);
    if(hptr==0) cout << "erreor\n";
    if(hptr->h_length!=4) cout << "it is ipv6\n";

    char** q=hptr->h_aliases;
    cout << "Aliases:\n";
    while(*q!=0) {
      cout << *q << endl;
      q++;
    }

    char** p=hptr->h_addr_list;
    cout << "Addresses:\n";
    while(*p!=0) {
      cout << inet_ntop(family,*((struct in_addr*)*(p))) << endl;
      p++;
    }
  }
  return 0;
}
```

6.2 TCP server with fork

This server sends a buffer containing "Here I am" to the clients that connect.

```
// BEGIN FILE: plain_server.cpp
#include "mdp_all.h"
int main(int argc, char** argv) {
  if(argc<2) {
    cout << "USAGE: server local_port\n";
    return 0;
  }
  sockaddr_in clientaddr;
  sockaddr_in servaddr;
  int port=atoi(argv[1]);
  int sfd=socket(AF_INET,SOCK_STREAM,0);
  memset(&servaddr,0,sizeof(sockaddr_in));
  servaddr.sin_family=AF_INET;
  servaddr.sin_port=htons(port);
  servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  if(bind(sfd, (sockaddr*) &servaddr, sizeof(sockaddr_in)))
    exit_message(1,"bind: failure to connect");
  if(listen(sfd,1024))
    exit_message(1,"listen: max connections exceeded");

  char data[]="HERE I AM\n";
  int connfd;
  int length=sizeof(clientaddr);
  int pid;
  while(1) {
    cout << "waiting...\n";
    connfd=accept(sfd,(sockaddr*) &clientaddr, (socklen_t*) &length);

    pid=fork();
    if(pid==0) {
      close(sfd);
      cout << "serving...\n";
      send(connfd,data,strlen(data),0);
      close(connfd);
      cout << "done!\n";
      exit(0); // becase of fork this will make a zombie!
    }
  }
```

```
}
```

6.3 TCP client

This client connects to a server and expects to receive a '\ n' terminated string from the server.

```
// BEGIN FILE: plain_client.cpp
#include "mdp_all.h"
int main(int argc, char** argv) {
  if(argc<3) {
    cout << "USAGE: client address port\n";
    return 0;
  }
  sockaddr_in remote_sockaddr;
  int sfd;
  int port=atoi(argv[2]);
  sfd=socket(AF_INET,SOCK_STREAM,0);
  memset(&remote_sockaddr,0,sizeof(sockaddr_in));
  remote_sockaddr.sin_family=AF_INET;
  remote_sockaddr.sin_port=htons(port);
  inet_pton(AF_INET,argv[1],&remote_sockaddr.sin_addr);
  int ce=connect(sfd,(sockaddr*) &remote_sockaddr, sizeof(sockaddr_in));
  if(ce!=0 && (errono==ECONNREFUSED || errno==ETIMEDOUT)) {
    close(sfd);
  } else {

    int bytes;
    string s;
    char c;
    while(1) {
      bytes=recv(sfd,&c,1,0);
      if(bytes<0) {
        cerr << "CONNECTION ERROR\n";
        break;
      } else if(bytes==0) {
        cerr << "END OF INPUT\n";
        break;
```

```
      }
      else if(c=='\n') break;
      s=s+c;
    }
    cout << s << endl;
    close(sfd);
  }
  return  0;
}
```

6.4 UDP server

This UDP server waits to receive datagrams. It displays each datagram it receives.

```
// BEGIN FILE: dgram_server.cpp
#include "mdp_all.h"
int main(int argc, char** argv) {
  if(argc!=2) {
    cout << "USAGE: ./a.out serving_port\n";
    return 0;
  }

  int sockint, s, namelen, client_address_size;
  struct sockaddr_in client, server;
  char buf[32];

  s = socket(AF_INET, SOCK_DGRAM, 0);
  if(s==-1) exit_message(1,"Socket was not created.\n");

  server.sin_family = AF_INET;
  server.sin_port = htons(atoi(argv[1]));
  server.sin_addr.s_addr = INADDR_ANY;

  if(bind(s,(const sockaddr*)&server,sizeof(server))<0)
    exit_message(3,"Error binding server");

  client_address_size = sizeof( client );
```

```
  printf("Waiting for a message to arrive.\n");
  if(recvfrom(s,buf,sizeof(buf),0,(struct sockaddr *)&client,
     (socklen_t*)&client_address_size)<0)
    exit_message(4,"Error in receive from\n");

  cout << "I received: `` << buf << endl;
  close(s);
}
```

6.5 UDP client

This UDP client sends a datagram "Hello" to the server.

```
// BEGIN FILE: dgram_client.cpp
#include "mdp_all.h"
int main(int argc, char** argv) {
  if(argc!=3) {
    cout << "Usage: ./a.out remote_address remote_port\n";
    return 0;
  }

  int s;
  struct sockaddr_in server;
  char buf[32];

  s = socket(AF_INET, SOCK_DGRAM, 0);
  if(s == -1) exit_message(1,"Socket was not created.\n");
  server.sin_family = AF_INET;
  server.sin_port = 0;
  server.sin_addr.s_addr = INADDR_ANY;
  if(bind(s,(const sockaddr*)&server, sizeof(server))<0)
    exit_message(3,"Error binding server.\n");

  server.sin_family = AF_INET;
  server.sin_port = htons(atoi(argv[2]));
  server.sin_addr.s_addr = inet_addr(argv[1]);

  strcpy( buf, "Hello" );
```

```
  cout << "Sending data to the socket.\n";
  sendto(s,buf,(strlen(buf)+1),0,
         (const sockaddr*)&server,sizeof(server));
  cout << "Data has been sent to the socket\n";
  close(s);
}
```

6.6 UDP server using non-blocking IO

A UDP server that uses non-blocking IO.

```
// BEGIN FILE: dgram_server_unblock.cpp
#include "mdp_all.h"
int main(int argc, char** argv) {
  if(argc!=2) {
    cout << "USAGE: ./a.out serving_port\n";
    return 0;
  }

  int sockint, s, namelen, client_address_size;
  struct sockaddr_in client, server;
  char buf[32];

  s = socket(AF_INET, SOCK_DGRAM, 0);
  if(s==-1) exit_message(1,"Socket was not created.\n");

  server.sin_family = AF_INET;
  server.sin_port = htons(atoi(argv[1]));
  server.sin_addr.s_addr = INADDR_ANY;

  if(bind(s,(const sockaddr*)&server,sizeof(server))<0)
    exit_message(3,"Error binding server");

  fcntl(s,F_SETFL,O_NONBLOCK);

  client_address_size = sizeof( client );
  printf("Waiting for a message to arrive.\n");
  while(1) {
```

```
    sleep(1);
    cout << "tick\n";
    int e=recvfrom(s,buf,sizeof(buf),0,
       (struct sockaddr *)&client,
       (socklen_t*)&client_address_size);
    if(e==EWOULDBLOCK) exit_message(1,"socket error\n");
    else if(e>0) {
      cout << "I received: " << buf << endl;
      break;
    }
  }

  close(s);
}
```

6.7 UDP server using asynchronous IO

A UDP server that uses asynchronous IO.

```
// BEGIN FILE: dgram_server_sigio.cpp
#include "mdp_all.h"

int s;
void sigio_handler(int signum) {
  char buf[32];
  struct sockaddr_in client;
  int client_address_size = sizeof( client );
  cout << "I got a signal SIGIO\n";
  int e=recvfrom(s,buf,sizeof(buf),0,
    (struct sockaddr *)&client,
    (socklen_t*)&client_address_size);
  cout << e << endl;
  if(e==EWOULDBLOCK) exit_message(1,"socket error\n");
  else if(e>0)
    cout << "I received: " << buf << endl;
}

int main(int argc, char** argv) {
```

```
if(argc!=2) {
  cout << "USAGE: ./a.out serving_port\n";
  return 0;
}

int sockint, namelen;
struct sockaddr_in server;

s = socket(AF_INET, SOCK_DGRAM, 0);
if(s==-1) exit_message(1,"Socket was not created.\n");

server.sin_family = AF_INET;
server.sin_port = htons(atoi(argv[1]));
server.sin_addr.s_addr = INADDR_ANY;

if(bind(s,(const sockaddr*)&server,sizeof(server))<0)
  exit_message(3,"Error binding server");

fcntl(s,F_SETFL,O_NONBLOCK);
fcntl(s,F_SETOWN,getpid());
fcntl(s,F_SETFL,FASYNC);

cout << "Waiting for a message to arrive.\n";

struct sigaction action;
action.sa_handler=sigio_handler;
if(sigemptyset(&action.sa_mask)<0)
   exit_message(1,"sigemptyset");
if(sigaddset(&action.sa_mask,SIGIO))
   exit_message(1,"sigaddset");
if(sigaction(SIGIO,&action,NULL)<0)
   exit_message(1,"sigaction");

while(1) {
  sleep(1);
  cout << "tick\n";
}
```

```
  close(s);
}
```

6.8 UDP server using poll

A UDP server that uses poll.

```
// BEGIN\ FILE: dgram_server_poll.cpp
#include "mdp_all.h"

int main(int argc, char** argv) {
  if(argc!=2) {
    cout << "USAGE: ./a.out serving_port\n";
    return 0;
  }

  int sockint, s, namelen, client_address_size;
  struct sockaddr_in client, server;
  char buf[32];

  s = socket(AF_INET, SOCK_DGRAM, 0);
  if(s==-1) exit_message(1,"Socket was not created.\n");

  server.sin_family = AF_INET;
  server.sin_port = htons(atoi(argv[1]));
  server.sin_addr.s_addr = INADDR_ANY;

  if(bind(s,(const sockaddr*)&server,sizeof(server))<0)
    exit_message(3,"Error binding server");

  struct pollfd fds[2];
  fds[0].fd=fileno(stdin);
  fds[0].events=POLLRDNORM;
  fds[1].fd=s;
  fds[1].events=POLLRDNORM | POLLERR;

  string text;
  char c;
```

```
  cout << "type something:\n";
  while(1) {
    int n=poll(fds,2,-1);
    if(n==0) exit_message(0,"timeout");
    if(n==-1) exit_message(1,"auch!");

    if(fds[0].revents) {
      read(fileno(stdin),&c,1);
      if(c=='\n') {
        cout << "You typed: " << text << endl;
        cout << "type something:\n";
        text="";
      } else {
        text=text+c;
      }
    }

    if(fds[1].revents) {
      int e=recvfrom(s,buf,sizeof(buf),0,
        (struct sockaddr *)&client,
        (socklen_t*)&client_address_size);
      if(e==EWOULDBLOCK) exit_message(1,"socket error\n");
      cout << "From: " << inet_ntoa(client.sin_addr) << endl;
      cout << "I received: " << buf << endl;
    }
  }
  close(s);
}
```

6.9 UDP server using select

Same as above but using select isstead of poll.

```
// BEGIN FILE: dgram_server_select.cpp
#include "mdp_all.h"

int max(int a, int b) {
```

```
  if(a>b) return a;
  return b;
}

int main(int argc, char** argv) {
  if(argc!=2) {
    cout << "USAGE: ./a.out serving_port\n";
    return 0;
  }

  int sockint, s, namelen, client_address_size;
  struct sockaddr_in client, server;
  char buf[32];

  s = socket(AF_INET, SOCK_DGRAM, 0);
  if(s==-1) exit_message(1,"Socket was not created.\n");

  server.sin_family = AF_INET;
  server.sin_port = htons(atoi(argv[1]));
  server.sin_addr.s_addr = INADDR_ANY;

  if(bind(s,(const sockaddr*)&server,sizeof(server))<0)
    exit_message(3,"Error binding server");

  fd_set rset;
  fd_set eset;

  string text;
  char c;
  int maxn=max(fileno(stdin),s)+1;
  cout << "type something:\n";
  while(1) {

    FD_ZERO(&rset);
    FD_SET(fileno(stdin),&rset);
    FD_SET(s,&rset);
    FD_ZERO(&eset);
    FD_SET(fileno(stdin),&eset);
```

```
    FD_SET(s,&eset);

    int n=select(maxn,&rset,0,&eset,0);
    if(n==0) exit_message(0,"timeout");
    if(n==-1) exit_message(1,"auch!");

    if(FD_ISSET(fileno(stdin),&rset)) {
      read(fileno(stdin),&c,1);
      if(c=='\n') {
        cout << "You typed: " << text << endl;
        cout << "type something:\n";
        text="";
      } else {
        text=text+c;
      }
    }

    if(FD_ISSET(s,&rset) ||
       FD_ISSET(s,&eset)) {
      int e=recvfrom(s,buf,sizeof(buf),0,
        (struct sockaddr *)&client,
        (socklen_t*)&client_address_size);
      if(e==EWOULDBLOCK) exit_message(1,"socket error\n");
      cout << "From: " << inet_ntoa(client.sin_addr) << endl;
      cout << "I received: " << buf << endl;
    }
  }
  close(s);
}
```

6.10 Setting an alarm

A program that asks the OS to send an alarm to the program.

```
// BEGIN FILE: test_alarm.cpp
#include "mdp_all.h"

void handler(int s) {
```

```
  if(s==SIGALRM)
    cout << "I got a SIGALRM signal\n";
}

int main() {
  struct sigaction action;
  action.sa_handler=handler;
  if(sigemptyset(&action.sa_mask)<0)
    exit_message(1,"sigemptyset");
  if(sigaddset(&action.sa_mask,SIGALRM))
    exit_message(1,"sigaddset");
  if(sigaction(SIGALRM,&action,NULL)<0)
    exit_message(1,"sigaction");

  cout << "Setting the alarm\n";
  alarm(5);
  cout << "Waiting\n";
  if(sleep(10)!=0)
    cout << "I have been awakened\n";
  else
    cout << "I woke up naturally\n";
  cout << "Done\n";
  return 0;
}
```

6.11 Killing zombies

A program that forks and prevents its child from becoming a zombie.

```
// BEGIN FILE: test_zombie.cpp
#include "mdp_all.h"

void handler(int s) {
  if(s==SIGCHLD) {
    pid_t pid;
    int status;
    pid=wait(&status);
    cout << "Process " << pid << " terminated with status " << status << "\n";
```

```
  }
}

int main() {
  int pid=fork();
  if(pid==0) {
    // the child process dies!
    sleep(3);
    exit(1);
  } else {
    cout << "My child has pid=" << pid << endl;
    struct sigaction action;
    action.sa_handler=handler;
    if(sigemptyset(&action.sa_mask)<0)
      exit_message(1,"sigemptyset");
    if(sigaddset(&action.sa_mask,SIGCHLD))
      exit_message(1,"sigaddset");
    if(sigaction(SIGCHLD,&action,NULL)<0)
      exit_message(1,"sigaction");

    if(sleep(10)!=0)
      cout << "I have been woken up\n";
    else
      cout << "I woke up naturally\n";
  }
  return 0;
}
```

7 Appendix B - OSI Model

The OSI, or Open System Interconnection, model defines a networking framework for implementing protocols in seven layers. Control is passed from one layer to the next, starting at the application layer in one station, proceeding to the bottom layer, over the channel to the next station and back up the hierarchy.

7.1 Physical

(Layer 1) This layer conveys the bit stream - electrical impulse, light or radio signal – through the network at the electrical and mechanical level. It provides the hardware means of sending and receiving data on a carrier, including defining cables, cards and physical aspects. Fast Ethernet, RS232, and ATM are protocols with physical layer components.

7.2 Data Link

(Layer 2) At this layer, data packets are encoded and decoded into bits. It furnishes transmission protocol knowledge and management and handles errors in the physical layer, flow control and frame synchronization. The data link layer is divided into two sublayers: The Media Access Control (MAC) layer and the Logical Link Control (LLC) layer. The MAC sublayer controls how a computer on the network gains access to the data and permission to transmit it. The LLC layer controls frame synchronization, flow control and error checking.

7.3 Network

(Layer 3) This layer provides switching and routing technologies, creating logical paths, known as virtual circuits, for transmitting data from node to node. Routing and forwarding are functions of this layer, as well as addressing, internet.

7.4 Transport

(Layer 4) This layer provides transparent transfer of data between end systems, or hosts, and is responsible for end-to-end error recovery and flow control. It ensures complete data transfer. working, error handling, congestion control and packet sequencing.

7.5 Session

(Layer 5) This layer establishes, manages and terminates connections between applications. The session layer sets up, coordinates, and terminates conversations, exchanges, and dialogues between the applications at each end. It deals with session and connection coordination.

7.6 Presentation

(Layer 6) This layer provides independence from differences in data representation (e.g., encryption) by translating from application to network format, and vice versa. The presentation layer works to transform data into the form that the application layer can accept. This layer formats and encrypts data to be sent across a network, providing freedom from compatibility problems. It is sometimes called the syntax layer.

7.7 Application

(Layer 7) This layer supports application and end-user processes. Communication partners are identified, quality of service is identified, user authentication and privacy are considered, and any constraints on data syntax are identified. Everything at this layer is application-specific. This layer provides application services for file transfers, e-mail, and other network software services. Telnet and FTP are applications that exist entirely in the application level. Tiered application architectures are part of this layer.

8 Appendix B - UNIX Commands

- ping
- arp
- netstat
- telnet
- tcpdump
- ifconfig
- ifup