SOKETLER

Bu dertse soket programlamaya odaklanılacak,bu program tarafından uygulanacaktır.**socketfun.c** :

```
#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <stdio.h>

#include <netdb.h>

#include <signal.h>


static char *

inadport_decimal(sad)

        struct sockaddr_in *sad;

{

        static char buf[32];

        int a;


        a = ntohl(0xffffffff & sad->sin_addr.s_addr);

        sprintf(buf, "%d.%d.%d.%d:%d",

                        0xff & (a >> 24),

                        0xff & (a >> 16),

                        0xff & (a >> 8),

                        0xff & a,
```

```
                0xffff & (int)ntohs(sad->sin_port));
        return buf;
}

int serve_socket(hn, port)
char *hn;
int port;
{

  struct sockaddr_in sn;
  int s;
  struct hostent *he;

  if (!(he = gethostbyname(hn))) {
    puts("can't gethostname");
    exit(1);
  }

  bzero((char*)&sn, sizeof(sn));
  sn.sin_family = AF_INET;
  sn.sin_port     = htons((short)port);
  sn.sin_addr = *(struct in_addr*)(he->h_addr_list[0]);

  if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
```

```
  perror("socket()");
  exit(1);
 }
 if (bind(s, (struct sockaddr*)&sn, sizeof(sn)) == -1) {
  perror("bind()");
  exit(1);
 }

 return s;
}

int accept_connection(s)
int s;
{
 int l;
 struct sockaddr_in sn;
 int x;

 sn.sin_family = AF_INET;

 if (listen(s, 1) == -1) {
  perror("listen()");
  exit(1);
 }
```

```
  l = sizeof(sn);
  if ((x = accept(s, (struct sockaddr*)&sn, &l)) == -1) {
    perror("accept()");
    exit(1);
  }
  return x;
}

#ifndef        ALB

int request_connection(hn, port)
char *hn;
int port;
{
  struct sockaddr_in sn;
  int s, ok;
  struct hostent *he;

  if (!(he = gethostbyname(hn))) {
    puts("can't gethostname");
    exit(1);
  }
  ok = 0;
```

```
  while (!ok) {
    sn.sin_family = AF_INET;
    sn.sin_port  = htons((short)port);
    sn.sin_addr.s_addr = *(u_long*)(he->h_addr_list[0]);

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket()");
      exit(1);
    }
    ok = (connect(s, (struct sockaddr*)&sn, sizeof(sn)) != -1);
    if (!ok) sleep (1);
  }
  return s;
}

#else  ALB

int request_connection(hn, port)
char *hn;
int port;
{
        char easy[BUFSIZ];
        int ok;
```

```
        sprintf(easy, "proxy!%s!%d", hn, port);
        while ((ok = ipcopen(easy, "")) < 0)
                sleep(1);
        return ok;
}

#endif        ALB
```

Unix içinde interprocess iletişimi(genel kısaltması IPC) gerçekleştirmenin en genel şekli soketlerdir.Biz IPC gerçekleştirmenin diğer yollarını gördük.Örneğin,prosesler dosya aracılığıyla ve pipes(paylaşılan bölgeler) aracılığıyla iletişim kurabilir. Ancak,bunların her ikiside sınırlıdır.Örneğin,proseslerden birinin diğeri tarafından çatallı(forked) olması durumunda iki proses arasında iletişim kurabilmenin tek yolu pipe aracılığıyladır.Sıklıkla biz 2 proses bağlantısız olduğu zaman birbirleriyle konuşmasını istiyoruz .Örneğin,bunlar farklı kabuklar,farklı kullanıcılar ya da farklı makineler tarafından oluşturulabilir.Soketler bize bunu yapma imkanı sağlarlar.

Unix soket arayüzü şimdiye kadar çalıştığın şeylerden daha gizemli şeylerden biridir.O sebeple bunun yerine bu doğrudan kullanılarak, **socketfun.c** aşağıdaki üç prosedürü kullanacağız:

(Solaris üzerinde, **gcc** ve **socketfun.c** kullanıyorsanız bazı özel bağlantı seçenekleri kullanmak zorundasınız.Son yürütülebilir yaptığınız zaman **"-lsocket -lnsl"** belirtin.Bu derste bu **makefile** içinde yapılır.SunOS gibi bir işletim sistemi ya da **cc** kullanırsanız bunu belirtmeniz gerekmez.)

```
extern int serve_socket(char *hn, int port);
                /* This serves a socket at the given host name and port
```

```
        number.  It returns an fd for that socket.  Hn should
        be the name of the server's machine. */

extern int accept_connection(int s);
        /* This accepts a connection on the given socket (i.e.
        it should only be called by the server).  It returns
        the fd for the connection.  This fd is of course r/w */

extern int request_connection(char *hn, int port);
        /* This is what the client calls to connect to a server's
        port.  It returns the fd for the connection. */
```

Soketlerin çalışması şöyledir:

Bir proses "hizmet veren" bir sokettir.Bunu yaparak,bir port numarası ile tanımlanan "port" kurmak için işletim sistemi anlatıyor.Bu hangi diğer proseslerin tanıtıcı olduğu proses hizmeti ile bağlantılı olabilir. Telefon soketleri için hoş bir benzetmedir.Bir soket bir telefon gibidir ve port numarasıda telefon numarası gibidir.Makine adı alan kodu gibidir.Böylece,biri "seni ararım" dediğinde,sadece telefon görüşmesi gibi ,alan kodunuzu ve port numaranızla bir bağlantı istemeleri gerekir.

Soket "hizmette" te iken bir telefon ve telefon numarası ile kendiniz kurabilirsiniz.Hizmet sonucu soket bir tanımlayıcısıdır.Ancak,okuma ve yazma dosya tanımlayıcısı gibi değil.Bunun yerine,gerçekten yapabileceğiniz tek şey onu(diğer proseslerin bu porta bağlanması Unix in anlattığı değil)kapatmaktır,ya da onun üzerinden bağlantıları kabul etmektir.

Sunucu **accept_connection()** çağırarak soket dosya tanımlayıcısı üzerinden bir bağlantıyı kabul eder.Başka bir proses bu sokete bağlandığı zaman **accept_connection()** sadece döndürür.Bu bağlantı yeni bir dosya tanıtıcı ile döner.Bu tanımlayıcı r/w ' dır.Diğer bir değişle sen ona **write()** ile bu baytlar

bağlı bu prosese gidecek ve ondan **read()** ile o başka bir proses tarafından yazılan baytları okuyacaktır.

Request_connection() ile olanlar servis ile non-serving prosesler bağlıdır.O bu proses ile bağlanmak için bu verilen ana makine üzerinde verilen port numarası ile soket hizmet etmiştir der.Bu bir telefon görüşmesi başlatmak gibidir. Bu bağlantının sonuna bir dosya tanıtıcı ile döner.

Bir soket servis edildikten sonra,birden fazla kez bağlı olabilir.Diğer prosesler var olduğu sürece calling **request_connection()** bu makine ve port numarasına bu hizmet prosesi herhangi sayıda **accept_connection()** çağırabilir. Bu bir telefon numarası ile birden fazla telefon hatları olması gibi bir türdendir.2 kişiden fazla kişi ile **jtalk** yazarken bu faydalı olacaktır.

Şimdi **serve1.c** ve **client1.c** dosyalarını inceleyelim.

----serve1.c----

```
#include <stdio.h>

#include <stdlib.h>

//#include "socketfun.h"


int inout(int in, int out)

{

  char s[1000];

  int i;


  i = 0;
```

```
  while(read(in, s+i, 1) != 0) {
    if (s[i] == '\n') {
      write(out, s, i+1);
      return i+1;
    }
    i++;
  }
  return 0;
}

main(int argc, char **argv)
{
  char *hn, *un;
  int port, sock, fd;
  int i;
  char s[1000];

  if (argc != 3) {
    fprintf(stderr, "usage: serve1 hostname port\n");
    exit(1);
  }

  hn = argv[1];
  port = atoi(argv[2]);
```

```
 if (port < 5000) {
   fprintf(stderr, "usage: serve1 hostname port\n");
   fprintf(stderr, "      port must be > 5000\n");
   exit(1);
 }
 un = getenv("USER");

 sock = serve_socket(hn, port);

 fd = accept_connection(sock);

 printf("Connection established.  Sending `Server: %s'\n", un);
 sprintf(s, "Server: %s\n", un);
 write(fd, s, strlen(s));

 printf("Receiving:\n");

 inout(fd, 1);
}
```

Yukarıda görüldüğü gibi **Serve1.c** iki parametre almaktadır. Sunucu olabilmesi için host adı ve soketin port numarası gerekmektedir. Host adı makinenin çalıştığı server1 adı olmalıdır(örneğin eğer cetus1c ile çalışıyorsanız serve1 in ilk parametresi cetus1c yada cetus1c.cs.utk.edu olabilir. Yapmanız gereken

makineyi yeteri bir şekilde nitelendirebilmektir.) Port numarası 5000 den büyük olmalıdır. Çünkü Unix programları küçük port numaraları kullanmaktadır.

Serve1.c sunucusu bir socket açar ve bir bağlantı kurulumu için bekler. Bir bağlantı isteği gelirse karşı tarafa **"Server: "** karakterini, sunucu işlemimin sağlandığı kullanıcı adını ve bağlantı sonu için yenibir satır gönderir. Daha sonra bağlantıdan bytelar okunur ve yeni satır gelene kadar ekrana çıktı olarak verir.

---client1.c---

```
#include <stdio.h>

#include <stdlib.h>

//#include "socketfun.h"


int inout(int in, int out)

{

 char s[1000];

 int i;


 i = 0;

 while(read(in, s+i, 1) != 0) {

  if (s[i] == '\n') {

   write(out, s, i+1);

   return i+1;

  }

  i++;
```

```
  }
  return 0;
}


main(int argc, char **argv)
{
  char *hn, *un;
  int port, fd;
  int i;
  char s[1000];

  if (argc != 3) {
    fprintf(stderr, "usage: client1 hostname port\n");
    exit(1);
  }

  hn = argv[1];
  port = atoi(argv[2]);
  un = getenv("USER");

  fd = request_connection(hn, port);

  printf("Connection established.  Receiving:\n");
```

```
 inout(fd, 1);


 printf("Sending `Client: %s'\n", un);
 sprintf(s, "Client: %s\n", un);
 write(fd, s, strlen(s));
}
```

Client1.c ,Serve1.c ile benzerdir. Paramatre olarak host adı ve port numarası alır, fakat soket sunucusu yerine, host adı ve port numarası ile sokete bağlantı sağlamaya çalışır. Bağlantı sağlandığında bağlantıdan gelen bytelar okunur ve yeni bir satır gelene kadar çıktı olarak verilir. Daha sonra karakter dizisi yollanır **"Client: "**, sonra kendi kullanıcı adı ve sunucu işlemi için yenibir satır yollanır.

Şimdi bunları örnekle gösterelim.(Makine olarak cetus1c de olduğunuzu varsayalım).

UNIX1> serve1 cetus1c 5001

Soket sunar ve bir bağlantı beklenir. **Client1** çalıştırılır.

UNIX2> client1 cetus1c 5001

Bunu yaptığımızda aşağıdaki sonuçlar elde edilir.

UNIX1> serve1 cetus1c 5001

Bağlantı sağlanır. 'Server: plank' yollanır.

Alınan:

Client: plank

UNIX2>

diğer tarafta elde edilen:

UNIX2> client1 cetus1c 5001

Bağlantı sağlanır.

Alınan:

Server: plank

Yollanan `Client: plank'

UNIX2>

Programları anlayana kadar inceleyin. Eğer **“client1”** i **“serve1”** den önce çalıştırırsanız client1 soket sunulana kadar bekler.

Şimdi uğraştığınızı varsayalım.

UNIX1> serve1 cetus2f 5001

bind(): İstenen adrese atama yapılamadı.

UNIX1>

burada **serve_socket()** ile söylenmek istenen **cetus2f** e soket sağlayamazsınız **cetus1c** de çalışırken.

Client1 herhangi bir makine yada kullanıcı tarafından çalıştırılabilir. Farklı makineden ilk deneme, Örneğin **servel** çalıştıralım.

UNIX1> serve1 cetus1c 5001

ve farklı makineye giriş yapalım **client1** i çalıştırmak için

UNIX2> rlogin cetus2e

....

UNIX2> client1 cetus1c 5001

Bağlantı sağlanır.

Alınan:

Server: plank

Yollanan `Client: plank'

UNIX2>

Harika bir şekilde çalışmalı. Şimdi aynısını farklı kullanıcı için deneyin. Örneğin **serve1** önce çalıştırın:

UNIX1> serve1 cetus1c 5001

Sonra **client1** için bir arkadaş edinin. Mesela Ben Dr.Booth u makinesini kullanması için edindim.

UNIX> client1 cetus1c 5001

Ona ulaşan:

UNIX> client1 cetus1c 5001

Bağlantı sağlanır.

Alınan:

Server: plank

Yollanan `Client: booth'

UNIX>

ve Bana ulaşan,

UNIX1> serve1 cetus1c 5001

Bağlantı sağlanır. Yollanan `Server: plank'

Alınan:

Client: booth

UNIX1>

ALTERNATE

Şimdi “talk” un sınırlı bir bölümünü yazmak istediğinizi varsayalım. Bağlantı kurmak ve gönderilen satırları başka bir alternatif sunucu/istemci çifti içerisinde bildirelim. Bu **alternate.c** dosyasıdır.

-----alternate.c----

```
#include <stdio.h>

#include <stdlib.h>

#include "socketfun.h"

int inout(int in, int out)

{

  char s[1000];

  int i;

  i = 0;

  while(read(in, s+i, 1) != 0) {

    if (s[i] == '\n') {

      write(out, s, i+1);

      return i+1;

    }

    i++;

  }

  return 0;

}

main(int argc, char **argv)

{
```

```
char *hn, *un;
int port, sock, fd;
int i;
char s[1000];
if (argc != 4) {
  fprintf(stderr, "usage: alternate hostname port s|c\n");
  exit(1);
}

hn = argv[1];
port = atoi(argv[2]);
if (port < 5000) {
  fprintf(stderr, "usage: alternate hostname port\n");
  fprintf(stderr, "       port must be > 5000\n");
  exit(1);
}
un = getenv("USER");
if (argv[3][0] == 's') {
  sock = serve_socket(hn, port);
  fd = accept_connection(sock);
} else {
  fd = request_connection(hn, port);
}
printf("Connection established.  Client should start talking\n", un);
```

```
 if (argv[3][0] == 's') {
  if (inout(fd, 1) == 0) exit(0);
 }
 while(1) {
  if (inout(0, fd) == 0) exit(0);
  if (inout(fd, 1) == 0) exit(0);
 }
}
```

Burada procedure **inout()** ifadesinin ne olduğunu tam anlamıyla bilmek gerekir. Burada yapacağımız her işlem, **stdin**'den bir satır okuma ve soket boyunca gönderme daha sonra her soketten satır okuma ve **stdout** boyunca gönderme işleminden oluşur.

Bu alternate dosyalarını çalıştırırken aynı makineden iki faklı kullanıcı, iki farklı makine veya iki farklı kullanıcı ile farklı makineler üzerinde çalıştırma seçeneklerini denemeliyiz.

MİNİTALK

Alternate'ler tamamen faydalı dosyalar değillerdir, bu yüzden kullanırken dikkatli olunmalıdır. Burada aynı anda soket üzerinden bir bağlantı veya **stdin** ile kullanıcı olarak bir bağlantı yapılabilir, bir tercih yapmalıyız. Bu işlemi bir sonraki dersin konusu olan **select()** ile gerçekleştirmekte mümkündür. **talk()** için toplamda 4 tane proses mevcuttur, bunlar iki tane sunucu iki tanede kullanıcı içindir. Bir sunucu/istemci çiftinde bir istemci tipi bağlantıyı sağlar sunucuya yazdırır diğerinde ise sunucu bağlantıyı sağlar ve istemciye yazdırır. Bu işlemi **minitalk.c** dosyası ile yapmak mümkündür.

----minitalk.c-----

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "socketfun.h"
main(int argc, char **argv)
{
  char *hn;
  int port, sock, fd;
  if (argc != 4) {
    fprintf(stderr, "usage: minitalk hostname port s|c\n");
    exit(1);
  }
  hn = argv[1];
  port = atoi(argv[2]);
  if (port < 5000) {
    fprintf(stderr, "usage: minitalk hostname port s|c\n");
    fprintf(stderr, "       port must be > 5000\n");
    exit(1);
  }
  if (strcmp(argv[3], "s") == 0) {
    sock = serve_socket(hn, port);
    fd = accept_connection(sock);
    close(sock);
  } else if (strcmp(argv[3], "c") == 0) {
    fd = request_connection(hn, port);
  } else {
    fprintf(stderr, "usage: minitalk hostname port s|c\n");
```

```
  fprintf(stderr, "      last argument must be `s' or `c'\n");
  exit(1);
 }

 printf("Connection made -- input goes to remote site\n");
 printf("                output comes from remote site\n");
 if (fork() == 0) {
  dup2(fd, 0);
 } else {
  dup2(fd, 1);
 }
 close(fd);
 execlp("cat", "cat", 0);
 fprintf(stderr, "Exec failed.  Drag\n");
}
```

Burada minitalk'ın 4 proses içerisinde **dups fd** 0 veya 1 olması exac'in kendi işlemi içerisinde anne-çocuk(üst düğüm veya alt düğüm) olmasına göre değişir. stdin'den okuma yaparak ve istemci çocuğa(alt düğüm) bağlantı boyunca çıkış gönderen bir anne sunucu(üst düğüm) vardır ve aynı şekilde istemci anneye bağlantı boyunca çıkış veren bir sunucu çocukta diyebiliriz. Bu tercihlere karar vermeli ve bunu bir denemeliyiz.

minitalk'ın tek sorunu proseslerden birisi öldüğünde diğer prosesler ölen prosesin iş yükünü karşılamazlar. Bu sizce neden, düşünün. minitalk'ı bitirmek ve prosesi öldürmek için "ps" ve "cat" kullanarak dosya içerisinde ne var görebiliriz.