# UNIX Domain Protocols

Sanjaya Kumar Jena

Computer Sc. & Engineering
Faculty of Engineering, ITER
SOA, Deemed To Be University
Bhubaneswar-30

# Introduction

- The Unix domain protocols are not an actual protocol suite, but a way of performing client/server communication on a single host using the same API that is used for clients and servers on different hosts.

- The Unix domain protocols are an alternative to the interprocess communication (IPC) methods, when the client and server are on the same host.

- Two types of sockets are provided in the Unix domain: stream sockets (similar to TCP) and datagram sockets (similar to UDP).

- Even though a **raw socket** is also provided, its semantics have never been documented as per infromation till now.

# Reasons for Unix Domain Protocols

- On Berkeley-derived implementations, Unix domain sockets are often twice as fast as a TCP socket when both peers are on the same host.

- Unix domain sockets are used when passing **descriptors** between processes on the same host.

- Implementations of Unix domain sockets provide the client's credentials (user ID and group IDs) to the server, which can provide additional security checking.

# Protocol addrersses: Unix Domain Protocols

- The protocol addresses used to identify clients and servers in the Unix domain are **pathnames within the normal filesystem**.

- These pathnames are not normal Unix files: We cannot read from or write to these files except from a program that has associated the pathname with a Unix domain socket.

- IPv4 uses a combination of 32-bit addresses and 16-bit port numbers for its protocol addresses.

- IPv6 uses a combination of 128-bit addresses and 16-bit port numbers for its protocol addresses.

# Unix Domain Socket Address Structure

The Unix domain socket address structure is defined in the header **<sys/un.h>**.

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

struct sockaddr_un {

  sa_family_t sun_family; /* AF_LOCAL */

  char sun_path[104];     /* null-terminated pathname */

};
```

- The pathname stored in the `sun_path` array must be null-terminated.
- The macro `SUN_LEN` is provided and it takes a pointer to a **`sockaddr_un`** structure and returns the length of the structure, including the number of non-null bytes in the pathname.
- The **unspecified** address is indicated by a **null string as the pathname**, that is, a structure with `sun_path[0]` equal to 0. This is the Unix domain equivalent of the IPv4 `INADDR_ANY` constant and the IPv6 `IN6ADDR_ANY_INIT` constant.
- POSIX renames the Unix domain protocols as “local IPC,” to remove the dependence on the Unix OS.
- The historical constant `AF_UNIX` becomes `AF_LOCAL`.

# Unix Domain Socket Address Structure Fillin

## UNSPECIFIED CASE

```
#include<sys/un.h>
int listenfd;
struct sockaddr_un  servaddr;
listenfd = socket(AF_LOCAL, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
servaddr.sun_path[0]=0; /* for unspecified address */
```

# Unix Domain Socket Address Structure Fillin

## UNSPECIFIED CASE

```
#include<sys/un.h>
int listenfd;
struct sockaddr_un  servaddr;
listenfd = socket(AF_LOCAL, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
servaddr.sun_path[0]=0; /* for unspecified address */
```

## SPECIFIED CASE

```
#include<sys/un.h>
int sockfd;
struct sockaddr_un servaddr;
sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
unlink(argv[1]);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
strcpy(servaddr.sun_path,argv[1]);/* for specified */
```

# bind, listen, getsockname in Unix Domain Socket Address Structure

```
int main(int argc, char **argv){
int sockfd;   socklen_t len;
struct sockaddr_un addr1, addr2;
if (argc != 2){
 printf("usage: unixbind <pathname>");
 return -1;
}
sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
unlink(argv[1]); /* OK if this fails */
bzero(&addr1, sizeof(addr1));
addr1.sun_family = AF_LOCAL;
strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) - 1);
bind(sockfd, (struct sockaddr *) &addr1, SUN_LEN(&addr1));
len = sizeof(addr2);
getsockname(sockfd, (struct sockaddr * *) &addr2, &len);
printf("bound name = %s, returned len = %d\n", addr2.sun_path
    , len);
listen(sockfd,5);
exit(0);
}
```

# Unix Domain Stream Server

```
int main(int argc, char **argv){
int listenfd, connfd; socklen_t clilen,len;
struct sockaddr_un cliaddr, servaddr;
len=sizeof(struct sockaddr_un);
listenfd = socket(AF_LOCAL, SOCK_STREAM, 0);
//unlink(argv[1]);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
//strcpy(servaddr.sun_path,argv[1]);// for specified
servaddr.sun_path[0]=0; //for unspecified address
bind(listenfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
getsockname(listenfd, (struct sockaddr *)&servaddr, &len);
printf("bound name for client= %s\n", servaddr.sun_path);
listen(listenfd, 5);
for ( ; ; ) {
    clilen = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &
        clilen);
    write(connfd,"YES\n",4);
    close(connfd);
}
return 0;}
```

# Unix Domain Stream Client

```
int main(int argc, char **argv){
 int listenfd, connfd,cr,n;
 char buf[12];
 socklen_t clilen;
 struct sockaddr_un cliaddr, servaddr;
 listenfd = socket(AF_LOCAL, SOCK_STREAM, 0);
 bzero(&servaddr, sizeof(servaddr));
 servaddr.sun_family = AF_LOCAL;
 strcpy(servaddr.sun_path, argv[1]);
 cr=connect(listenfd,(struct sockaddr *)&servaddr,sizeof(
     servaddr));
 if(cr==0){
   printf("connect success\n");
 }
 else{
  printf("Unable to Connect:\n");
  return -1;
 }
 n=read(listenfd,buf,4);
 write(1,buf,n);
 return 0;
}
```

# Unix Domain Datagram Server

```
int main(int argc, char **argv){
int listenfd, connfd,sr;
char buf[20];
socklen_t clilen,len;
struct sockaddr_un cliaddr, servaddr;
len=sizeof(struct sockaddr_un);
listenfd = socket(AF_LOCAL, SOCK_DGRAM, 0);
unlink(argv[1]);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
strcpy(servaddr.sun_path,argv[1]);// for specified
bind(listenfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
getsockname(listenfd,(struct sockaddr *)&servaddr, &len);
printf("bound name for client= %s\n", servaddr.sun_path);
for ( ; ; ) {
  sr=recvfrom(listenfd,buf,sizeof(buf),0,(struct sockaddr *)&
      cliaddr,&len);
  write(1,buf,sr);
  write(1,"\n",1);
}
return 0;
}
```

# Unix Domain Datagram Client

```
int main(int argc, char **argv)
{
 int sockfd,cr,n;
 char buf[12];
 socklen_t len;
 struct sockaddr_un cliaddr;
 len=sizeof(cliaddr);
 sockfd = socket(AF_LOCAL, SOCK_DGRAM, 0);
 bzero(&cliaddr, sizeof(cliaddr));
 cliaddr.sun_family = AF_LOCAL;
 strcpy(cliaddr.sun_path, argv[1]);
 sendto(sockfd,"SOA---ITER",10,0,(struct sockaddr *)&cliaddr,
     len);
 write(1,"Me done\n",8);
 return 0;
}
```

# `socketpair` Function

The **`socketpair`** function creates two sockets that are then connected together. This function applies only to Unix domain sockets.

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

int
socketpair(int family,int type,int protocol,int sockfd[2]);

Returns: nonzero if OK, -1 on error
```

- The family must be **`AF_LOCAL`**
- The **`type`**, however, can be either **`SOCK_STREAM`** or **`SOCK_DGRAM`**.
- The **`protocol`** must be 0.
- The two socket descriptors that are created are returned as **`sockfd[0]`** and **`sockfd[1]`**.
- The two created sockets are unnamed; that is, there is no implicit **`bind`** involved.
- The result of socketpair with a type of **`SOCK_STREAM`** is called a **stream pipe**. It is similar to a regular Unix pipe (created by the pipe function), but a **stream pipe** is **full-duplex**; that is, both descriptors can be read and written.

# Unix Domain Socket Uses: Passing Descriptors

When we think of passing an open descriptor from one process to another, we normally think of either

- A child sharing all the open descriptors with the parent after a call to **`fork`**
- All descriptors normally remaining open when **`exec`** is called

Current Unix systems provide a way to pass any open descriptor from one process to any other process. That is, there is no need for the processes to be related, such as a parent and its child. The technique requires us to first establish a Unix domain socket between the two processes and then use **`sendmsg`** to send a special message across the Unix domain socket. This message is handled specially by the kernel, passing the open descriptor from the sender to the receiver.

# Descriptor Passing Example SELF-TRY.....

# Receiving Sender Credentials SELF-TRY.....

# THANK YOU