UNIX Networking 2

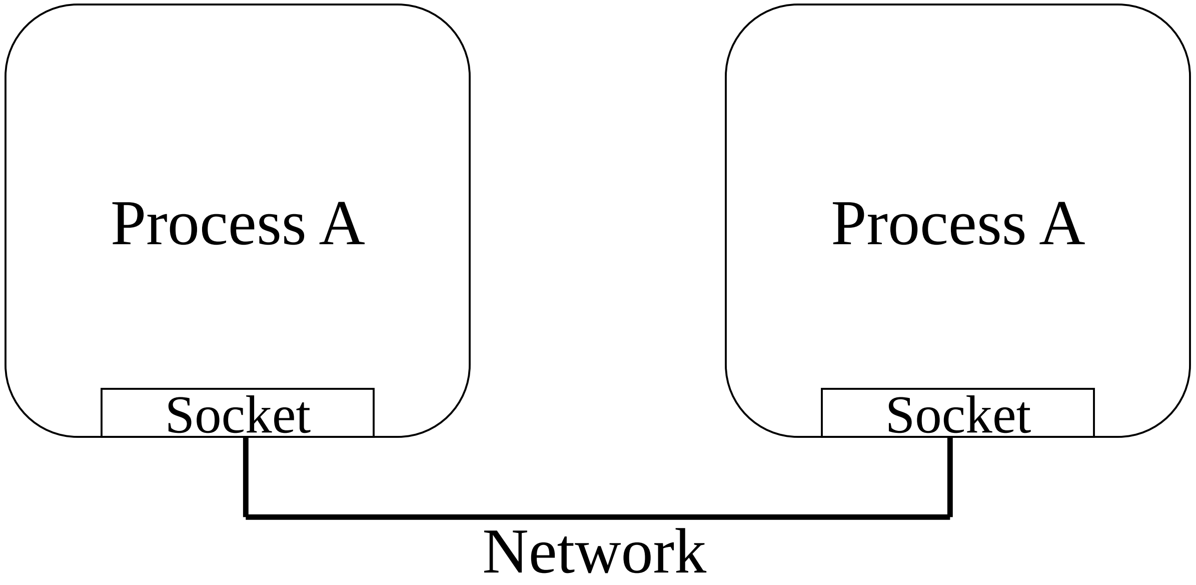
Benjamin Brewster

Adapted from Jon Herlocker, OSU

Network PIs

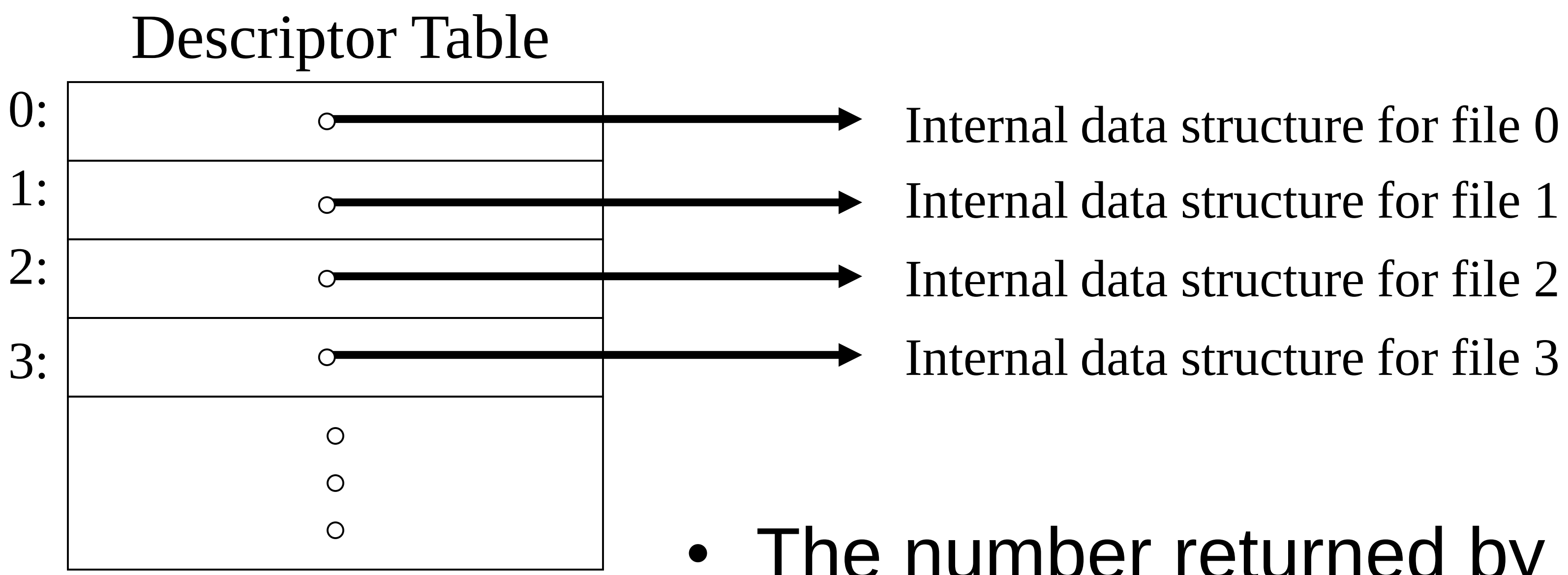
- UNIX Network **P**rogramming **I**nterfaces
  - X/Open Transport Interface (XTI)
  - Berkeley Sockets

Network Sockets



- Berkeley Socket API
  - A "socket" is the endpoint of a communication link between two processes
  - The socket API treats network connections like files as much as possible
  - Developed in the early 1980s for BSD Unix under a grant from DARPA

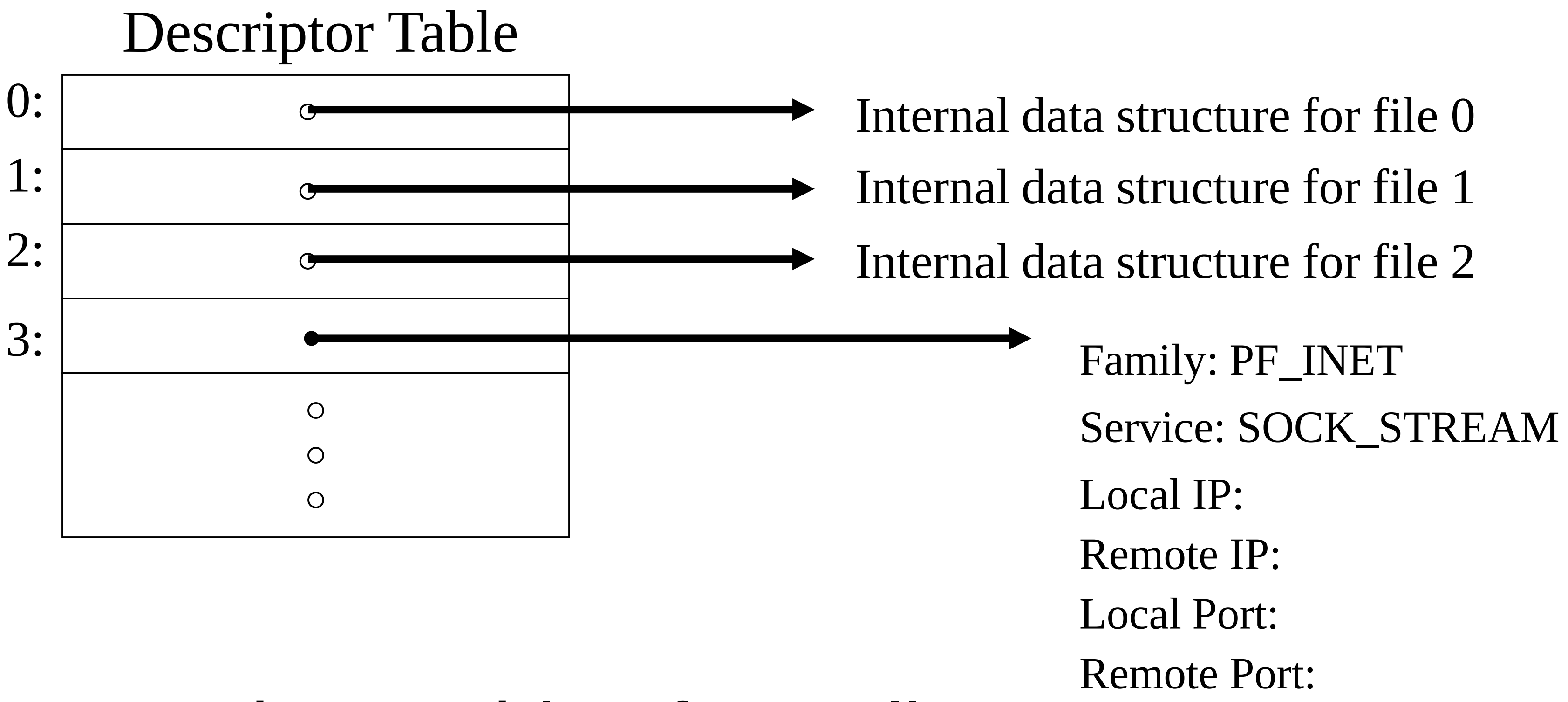
File Descriptor Table


Descriptor Table
0:
1:
2:
3:
Internal data structure for file 0
Internal data structure for file 1
Internal data structure for file 2
Internal data structure for file 3

- The number returned by `open()` is an index into an array of pointers to internal OS data structures

Table from Comer & Stevens,
Internetworking with TCP/IP

File Descriptor Table


Descriptor Table
0:
1:
2:
3:
Internal data structure for file 0
Internal data structure for file 1
Internal data structure for file 2
Family: PF_INET
Service: SOCK_STREAM
Local IP:
Remote IP:
Local Port:
Remote Port:

- Descriptor table after call to socket() command

Client Sockets & TCP/IP

- Client initializing connection with server
  - Server must already be running and expecting connections

- Since the client is using the TCP protocol, it needs to know the IP address and port of the server beforehand

Ports

- Many different processes can be running on one computer
- However, an IP address only identifies the machine, not the process
- Ports are used to reach a specific process on a machine
- Each process listens on a unique port - similar to a PO Box (ports are part of TCP)
- So a complete address is a IP address combined with a port number

Socket Documentation

- Most socket related man pages are in the "3n" section
  - `man -s 3n socket`
  - `man -k socket`

- All the info you need to use the network library is scattered across different man pages
  - Hard to use man pages for overall information
  - Best to have a book, and/or sample code available

Client Sockets

- Process:
  - 1. Create the socket endpoint - socket()
  - 2. Connect the socket to the server - connect()
  - 3. Use read() and write(), or send() and recv() to transfer data to and from the socket

Creating the Socket

```
int socket(int domain, int type, int protocol);
```

For IP, use AF_INET

For TCP, use SOCK_STREAM
For UDP, use SOCK_DGRAM

Returns file descriptor or -1

For IP, use 0

```
if ((sockfd = socket(AF_INET,SOCK_STREAM, 0)) == -1)
{
        perror("socket");
        exit(1);
}
```

Connecting the Socket

connect() connects the socket to a remote address

1. Sock file descriptor

```
int connect(int sockfd,
            struct sockaddr *address,
            size_t address_size);
```

2. Network address to connect to

Returns 0 on success or -1 on failure

3. Size of structure specified in 2.

```
if (connect(sockfd, (struct sockaddr *) &server, sizeof(server)) == -1)
{
        perror("connect");
        exit(1);
}
```

Filling the address struct
IP Number Address

- Client connecting to server

```
struct sockaddr_in server;

server.sin_family = AF_INET;
server.sin_port = htons(7000);
server.sin_addr.s_addr = inet_addr("192.168.1.1");
```

The IP address field

- Suppose you have declared

  `struct sockaddr_in server;`

- If you acquire the address in the form of

  `in_addr_t`

  then you assign directly:

  ```
  server.sin_addr.s_addr = inet_addr("192.168.1.1");
  ```

The IP address field

- If you have anything else, such as

```
int ipaddress;
```

- then you should be careful that the data doesn't get changed during type conversion

- memcpy is one solution:

```
memcpy(&server.sin_addr, &ipaddress, sizeof(ipaddress));
```

Setting up an address

- Client connecting to server:

```
struct hostent *server_ip_address;
server_ip_address = gethostbyname("www.engr.orst.edu");

if (server_ip_address == NULL)
{
  fprintf(stderr, "could not resolve server host name\n");
  exit(1);
}

server.sin_family = AF_INET;
server.sin_port = htons(80);

memcpy(&server.sin_addr, server_ip_address->h_addr,
         server_ip_address->h_length);
```

Setting up an address

- Server accepting connections:

```
struct sockaddr_in server;

server.sin_family = AF_INET;
server.sin_port = htons(7000);
server.sin_addr.s_addr = INADDR_ANY;
```

Sending data

1. Socket file descriptor

2. Pointer to data that should be sent

```
ssize_t send(int sockfd, void *message,
         size_t message_size, int flags);
```

Returns number of bytes sent, or 0

3. Number of bytes to send, starting at address in 2

4. Configuration flags

```
char request[1024];
r = send(sockfd, request, 1024, 0);
if (r < 1024)
     {} // handle possible error
```

Send

- Send will block until all the data has been sent, or the connection goes away
- Remember that internet connections fail all the time
  - Client intentionally disconnects (STOP button in a web browser)
  - Network partitions
  - Network failure
  - etc.

Receiving data

2. Pointer to buffer for storing data

1. Sock file descriptor

```
ssize_t recv(int sockfd, void *buffer,
             size_t buffer_size, int flags);
```

4. Configuration flags

Returns number of bytes sent, or 0

3. Max number of bytes to retrieve

```
char buffer[1024];
r = recv(sockfd, buffer, 1024, 0);
if (r < 1024)
{
        // error if r == -1
        // if 0 < r < 1024, may be more data
        // if r == 0, end of data
}
```

Receiving data

- Data may arrive in odd size bundles
- recv() or read() will return exactly the amount of data that has already arrived
- more data may be coming as long as the return value is greater than 0
- recv() and read() will block if the connection is open but no data is available

Demo Client/Server

```
% gcc -o client client.c
% gcc -o server server.c
% ./server 51717 &
% ./client localhost 51717
Please enter the message: test
Here is the message: test
I got your message
[1]+  Done                    ./server 51717
```