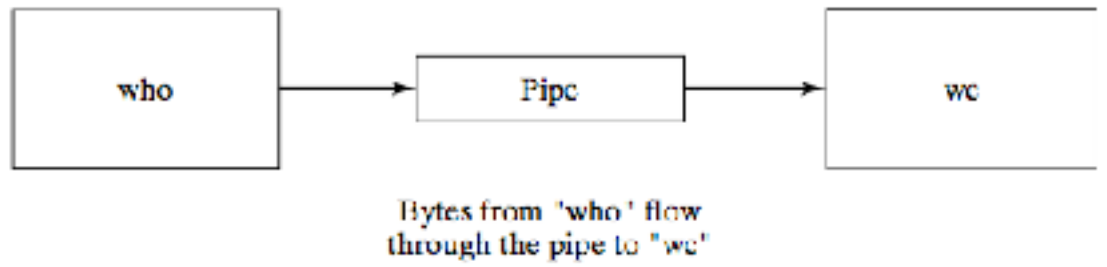

Figure 12–52 A simple pipe.

System Call: int pipe (int *fd* [2])

pipe () creates an unnamed pipe and returns two file descriptors; the descriptor associated with the "read" end of the pipe is stored in *fd*[0], and the descriptor associated with the "write" end of the pipe is stored in *fd*[1].

The following rules apply to processes that read from a pipe:

- If a process reads from a pipe whose write end has been closed, the read () returns a 0, indicating end-of-input.
- If a process reads from an empty pipe whose write end is still open, it sleeps until some input becomes available.
- If a process tries to read more bytes from a pipe than are present, all of the current contents are returned and read () returns the number of bytes actually read.

The following rules apply to processes that write to a pipe:

- If a process writes to a pipe whose read end has been closed, the write fails and the writer is sent a SIGPIPE signal. The default action of this signal is to terminate the writer.
- If a process writes fewer bytes to a pipe than the pipe can hold, the write () is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process. If a process writes more bytes to a pipe than the pipe can hold, no similar guarantees of atomicity apply.

Since access to an unnamed pipe is via the file descriptor mechanism, typically only the process that creates a pipe and its descendants may use the pipe.[a] lseek () has no meaning when applied to a pipe.

If the kernel cannot allocate enough space for a new pipe, pipe () returns -1; otherwise, it returns 0.

a. In advanced situations, it is actually possible to pass file descriptors to unrelated processes via a pipe.

Figure 12–53 Description of the pipe () system call.

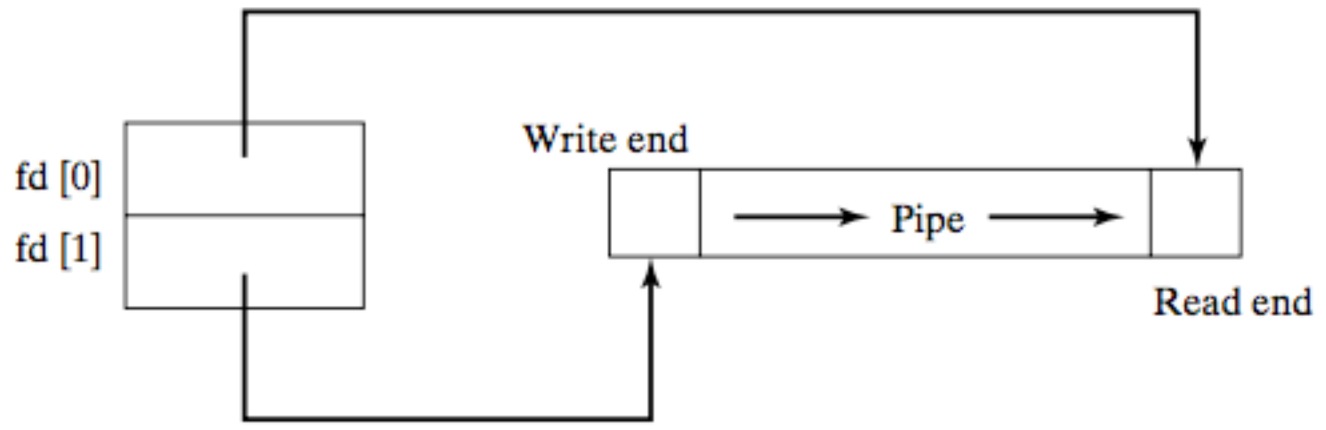

Figure 12–54 An unnamed pipe.

1. The parent process creates an unnamed pipe using pipe ().
2. The parent process forks.
3. The writer closes its read end of the pipe, and the designated reader closes its write end of the pipe.
4. The processes communicate by using write () and read () calls.
5. Each process closes its active pipe descriptor when finished with it.

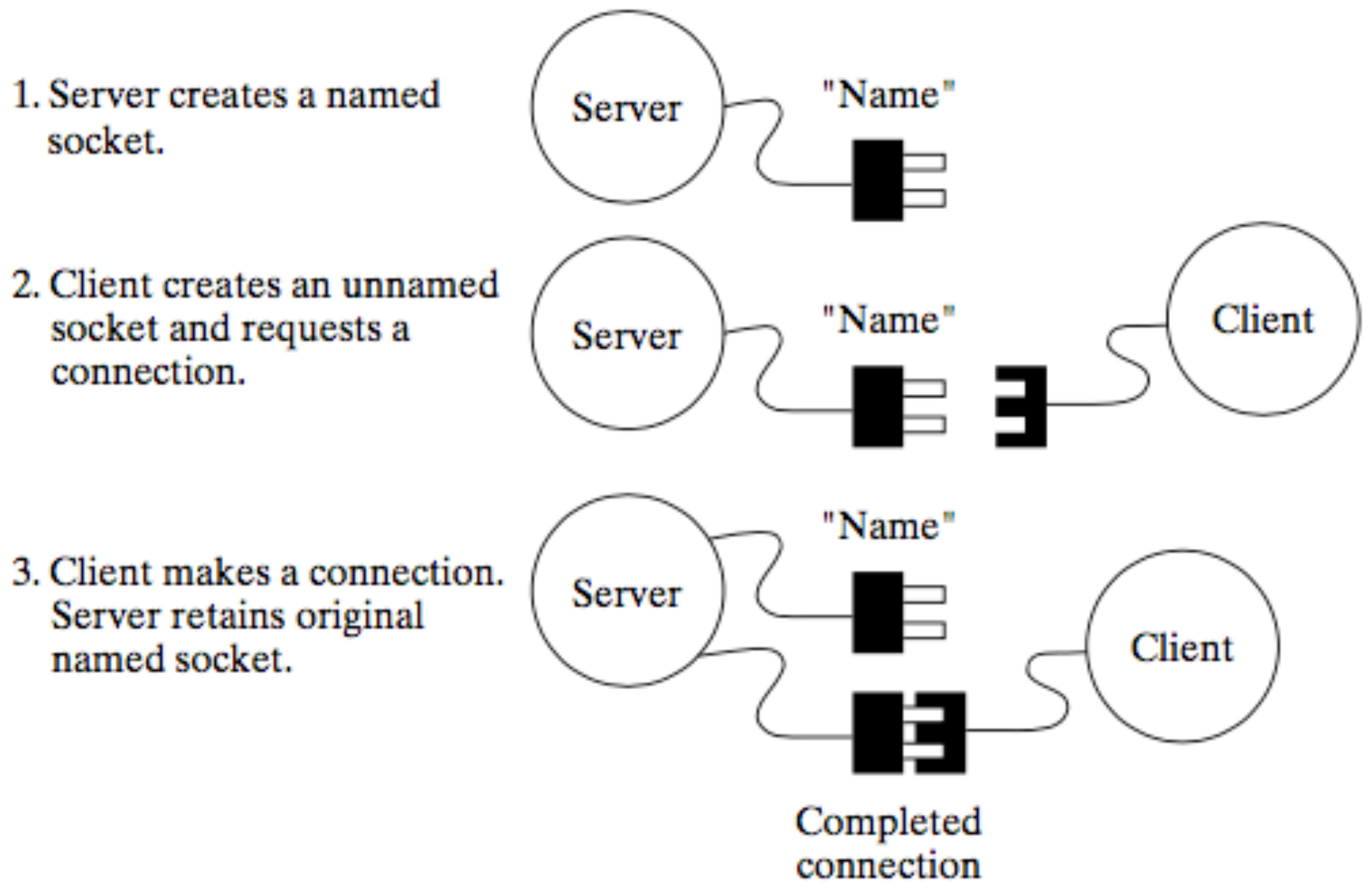
1. Server creates a named socket.
Server
"Name"
2. Client creates an unnamed socket and requests a connection.
Server
"Name"
Client
3. Client makes a connection. Server retains original named socket.
Server
"Name"
Client
Completed connection

Figure 12–56 The socket connection.

Domains The domain of a socket indicates where the server and client sockets may reside; the domains that are currently supported include:

- PF_LOCAL (the clients and server must be in the same machine, also called PF_UNIX)
- PF_INET (the clients and server are on the network)
- PF_INET6 (the clients and server are on an IPv6 network)

Other domains are listed in the socket man page. PF stands for “Protocol Family.” This book contains examples of PF_LOCAL and PF_INET sockets.

Types The type of a socket determines the type of communication that can exist between the client and server; the two main types that are currently supported are:

- SOCK_STREAM: sequenced, reliable, two-way connection based, variable-length streams of bytes
- SOCK_DGRAM: like telegrams; connectionless, unreliable, fixed-length messages

Other types that are either in the planning stages or implemented only in some domains include:

- SOCK_SEQPACKET: sequenced, reliable, two-way connection based, fixed-length packets of bytes
- SOCK_RAW: provides access to internal network protocols and interfaces

Name	Meaning
socket	Creates an unnamed socket.
bind	Gives the socket a name.
listen	Specifies the maximum number of pending connections.
accept	Accepts a socket connection from a client.

Figure 12–57 System calls used by a typical Linux daemon process.

System Call: int **socket** (int *domain*, int *type*, int *protocol*)

socket () creates an unnamed socket of the specified domain, type, and protocol. The legal values of these parameters were described earlier in this section.

If socket () is successful, it returns a file descriptor associated with the newly created socket; otherwise, it returns -1.

Figure 12–58 Description of the socket () system call.

System Call: int **bind** (int *fd*, const struct sockaddr* *address*, size_t *addressLen*)

bind () associates the unnamed socket represented by file descriptor *fd* with the socket address stored in *address*. *addressLen* must contain the length of the address structure. The type and value of the incoming address depend on the socket domain.

If the socket is in the PF_LOCAL domain, a pointer to a **sockaddr_un** structure (defined in "/usr/include/sys/un.h") must be cast to a (**sockaddr***) and passed in as *address*. This structure has two fields that should be set as follows:

FIELD	ASSIGN THE VALUE
sun_family	PF_LOCAL
sun_path	the full pathname of the socket (absolute or relative), up to 108 characters long

If the named PF_LOCAL socket already exists, an error occurs, so it's a good idea to unlink () a name before attempting to bind to it.

If the socket is in the PF_INET domain, a pointer to a **sockaddr_in** structure must be cast to a (**sockaddr***) and passed in as *address*. This structure has four fields, which should be set as follows:

FIELD	ASSIGN THE VALUE
sin_family	PF_INET
sin_port	the port number of the Internet socket
sin_addr	a structure of type in_addr that holds the Internet address
sin_zero	leave empty

For more information about Internet ports and addresses, please consult the Internet-specific part of this section.

If bind () succeeds, it returns a 0; otherwise, it returns -1.

Figure 12–59 Description of the bind () system call.

System Call: int **listen** (int *fd*, int *queueLength*)

listen () allows you to specify the maximum number of pending connections on a socket. Right now, the maximum queue length is 5. If a client attempts a connection to a socket whose queue is full, it is denied.

Figure 12–60 Description of the listen () system call

System Call: int **accept** (int *fd*, struct sockaddr* *address*, int* *addressLen*)

accept () listens to the named server socket referenced by *fd* and waits until a client connection request is received. When this occurs, accept () creates an unnamed socket with the same attributes as the original named server socket, connects it to the client's socket, and returns a new file descriptor that may be used for communication with the client. The original named server socket may be used to accept more connections.

The *address* structure is filled with the address of the client, and is normally only used in conjunction with Internet connections. The *addressLen* field should be initially set to point to an integer containing the size of the structure pointed to by *address*. When a connection is made, the integer that it points to is set to the actual size, in bytes, of the resulting *address*.

If accept () succeeds, it returns a new file descriptor that may be used to talk with the client; otherwise, it returns -1.

Figure 12–61 Description of the accept () system call.

Name	Meaning
socket	Creates an unnamed socket.
connect	Attaches an unnamed client socket to a named server socket.

Figure 12–62 System calls used by a typical Linux client process.

System Call: int **connect** (int *fd*, struct sockaddr* *address*, int *addressLen*)

connect () attempts to connect to a server socket whose address is contained within a structure pointed to by *address*. If successful, fd may be used to communicate with the server's socket. The type of structure that *address* points to must follow the same rules as those stated in the description of bind ():

- If the socket is in the PF_LOCAL domain, a pointer to a **sockaddr_un** structure must be cast to a (**sockaddr***) and passed in as *address*.
- If the socket is in the PF_INET domain, a pointer to a **sockaddr_in** structure must be cast to a (**sockaddr***) and passed in as *address*.

addressLen must be equal to the size of the address structure.

If the connection is made, connect () returns 0. If the server socket doesn't exist or its pending queue is currently filled, connect () returns -1.

Figure 12–63 Description of the connect () system call.

Library Function: in_addr_t **inet_addr** (const char* *string*)

inet_addr () returns the 32-bit IP address that corresponds to the A.B.C.D format *string*. The IP address is in network byte order.

Figure 12–64 Description of the inet_addr () library function.

System Call: int **gethostname** (char* *name*, int *nameLen*)

gethostname () sets the character array pointed to by *name* of length *nameLen* to a null-terminated string equal to the local host's name.

Figure 12–65 Description of the gethostname () system call.

Library Function: struct hostent* **gethostbyname** (const char* *name*)

gethostbyname () searches the "/etc/hosts" file (and/or DNS database if the host is a DNS client) and returns a pointer to a **hostent** structure that describes the file entry associated with the string *name*.

If *name* is not found in the "/etc/hosts" file, NULL is returned.

Figure 12–66 Description of the gethostbyname () library function.

Library Function: char* **inet_ntoa** (struct in_addr *address*)

inet_ntoa () takes a structure of type *in_addr* as its argument and returns a pointer to a string that describes the address in the format A.B.C.D.

Figure 12–67 Description of the inet_ntoa () library function.

Library Function: void **memset** (void* *buffer*, int *value*, size_t *length*)

memset () fills the array *buffer* of size *length* with the value of *value*.

Figure 12–68 Description of the memset () library function.

Library Function: void **bzero** (void* *buffer*, size_t *length*)

bzero () fills the array *buffer* of size *length* with zeroes (ASCII NULL).

Figure 12–69 Description of the bzero () library function.

Library Function: in_addr_t **htonl** (in_addr_t *hostLong*)
in_port_t **htons** (in_port_t *hostShort*)
in_addr_t **ntohl** (in_addr_t *networkLong*)
in_port_t **ntohs** (in_port_t *networkShort*)

Each of these functions performs a conversion between a host-format number and a network-format number. For example, htonl () returns the network-format equivalent of the host-format unsigned long *hostLong*, and ntohs () returns the host-format equivalent of the network-format unsigned short *networkShort*.

Figure 12–70 Description of the htonl (), htons (), ntohl (), and ntohs () library functions.

```
int serverFd; /* Server socket
struct sockaddr_in serverINETAddress; /* Server Internet address */
struct sockaddr* serverSockAddrPtr; /* Pointer to server address */
struct sockaddr_in clientINETAddress; /* Client Internet address */
struct sockaddr* clientSockAddrPtr; /* Pointer to client address */
int port = 13; /* Set to the port that you wish to serve */
int serverLen; /* Length of address structure */

serverFd = socket (PF_INET, SOCK_STREAM, DEFAULT_PROTOCOL);/* Create */
serverLen = sizeof (serverINETAddress); /* Length of structure */

bzero ((char*) &serverINETAddress, serverLen); /* Clear structure */
serverINETAddress.sin_family = PF_INET; /* Internet domain */
serverINETAddress.sin_addr.s_addr = htonl (INADDR_ANY);/* Accept all */
serverINETAddress.sin_port = htons (port); /* Server port number */
```

When the address is created, the socket is bound to the address, and its queue size is specified in the usual way:

```
serverSockAddrPtr = (struct sockaddr*) &serverINETAddress;
bind (serverFd, serverSockAddrPtr, serverLen);
listen (serverFd, 5);
```

```
clientLen = sizeof (clientINETAddress);
clientSockAddrPtr = (struct sockaddr*) clientINETAddress;
clientFd = accept (serverFd, clientSockAddrPtr, &clientLen);
```

Port # / Layer	Name	Comment
1	tcpmux	TCP port service multiplexer
5	rje	Remote Job Entry
7	echo	Echo service
9	discard	Null service for connection testing
11	systat	System Status service for listing connected ports
13	daytime	Sends date and time to requesting host
17	qotd	Sends quote of the day to connected host
18	msp	Message Send Protocol
19	chargen	Character Generation service; sends endless stream of characters
20	ftp-data	FTP data port
21	ftp	File Transfer Protocol (FTP) port; sometimes used by File Service Protocol (FSP)
22	ssh	Secure Shell (SSH) service
23	telnet	The Telnet service
25	smtp	Simple Mail Transfer Protocol (SMTP)
37	time	Time Protocol
39	rlp	Resource Location Protocol
42	nameserver	Internet Name Service
43	nicname	WHOIS directory service

49	tacacs	Terminal Access Controller Access Control System for TCP/IP based authentication and access
50	re-mail-ck	Remote Mail Checking Protocol
53	domain	domain name services (such as BIND)
63	whois++	WHOIS++, extended WHOIS services
67	bootps	Bootstrap Protocol (BOOTP) services; also used by Dynamic Host Configuration Protocol (DHCP)
68	bootpc	Bootstrap (BOOTP) client; also used by Dynamic Host Control Protocol (DHCP) clients
69	tftp	Trivial File Transfer Protocol (TFTP)
70	gopher	Gopher Internet document search and retrieval
71	netrjs-1	Remote Job Service
72	netrjs-2	Remote Job Service
73	netrjs-3	Remote Job Service
73	netrjs-4	Remote Job Service
79	finger	Finger service for user contact information
80	http	HyperText Transfer Protocol (HTTP) for World Wide Web (WWW) services
88	kerberos	Kerberos network authentication system
95	supdup	Telnet protocol extension
101	hostname	Hostname services on SRI-NIC machines
102/tcp	iso-tsap	ISO Development Environment (ISODE) network applications
105	csnet-ns	Mailbox nameserver; also used by CSO nameserver