

# CS810D - Advanced Programming in the UNIX Environment

—

## Interprocess Communication

---

Department of Computer Science  
Stevens Institute of Technology  
Jan Schaumann

`jschauma@cs.stevens.edu`

`http://www.cs.stevens.edu/~jschauma/810-APUE/`

## Pipes: `pipe(2)`

---

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Returns: 0 if OK, -1 otherwise

- oldest and most common form of UNIX IPC
- half-duplex
- can only be used between processes that have a common ancestor

### Behavior after closing one end:

- `read(2)` from a pipe whose write end has been closed returns 0 after all data has been read
- `write(2)` to a pipe whose read end has been closed generates SIGPIPE signal. If caught or ignored, `write(2)` returns an error and sets `errno` to EPIPE.

## Pipes: `popen(3)` and `pclose(3)`

---

```
#include <stdio.h>

FILE *popen(const char *cmd, const char *type);
           Returns: file pointer if OK, NULL otherwise

int pclose(FILE *fp);
           Returns: termination status cmd or -1 on error
```

- historically implemented using unidirectional pipe
- *type* one of “r” or “w”
- *cmd* passed to `/bin/sh -c`

## FIFOs: `mkfifo(2)`

---

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

```
int mkfifo(const char *path, mode_t mode);
```

Returns: 0 if OK, -1 otherwise

- aka “named pipes”
- allows unrelated processes to communicate
- just a type of file – test for using `S_ISFIFO(st_mode)`
- *mode* same as for `open(2)`
- use regular I/O operations (ie `open(2)`, `read(2)`, `write(2)`, `unlink(2)` etc.)
- main uses for FIFOs:
  - used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files
  - pass data between client and server

## Sockets: `socketpair(2)`

---

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

int socketpair(int d, int type, int protocol, int *sv);
```

The `socketpair(2)` call creates an unnamed pair of connected sockets in the specified domain `d`, of the specified `type`, and using the optionally specified `protocol`.

The descriptors used in referencing the new sockets are returned in `sv[0]` and `sv[1]`. The two sockets are indistinguishable.

## Sockets: `socket(2)`

---

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

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

Some of the currently supported domains are:

Domain	Description
PF_LOCAL	local (previously UNIX) domain protocols
PF_INET	ARPA Internet protocols
PF_INET6	ARPA IPv6 (Internet Protocol version 6) protocols
PF_ISDN	Integrated Services Digital Network
PF_ARP	RFC 826 Ethernet Address Resolution Protocol

Some of the currently defined types are:

Type	Description
SOCK_STREAM	sequenced, reliable, two-way connection based byte streams
SOCK_DGRAM	connectionless, unreliable messages of a fixed (typically small) maximum length
SOCK_SEQPACKET	sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length
SOCK_RAW	access to internal network protocols and interfaces

## Sockets: Datagrams in the UNIX domain

---

`udgramread.c` and `udgramsend.c`

- create socket using `socket(2)`
- attach to a socket using `bind(2)`
- binding a name in the UNIX domain creates a socket in the file system
- both processes need to agree on the name to use
- these files are only used for rendezvous, not for message delivery once a connection has been established
- sockets must be removed using `unlink(2)`

## Sockets: Datagrams in the Internet Domain

---

`dgramread.c` and `dgramsend.c`

- Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed.
- The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value `INADDR_ANY`.
- “well-known” ports (range 1 - 1023) only available to super-user
- request any port by calling `bind(2)` with a port number of 0
- determine used port number (or other information) using `getsockname(2)`
- convert between network byteorder and host byteorder using `htons(3)` and `ntohs(3)` (which may be noops)

## Sockets: Connections using stream sockets

---

`streamread.c` and `streamwrite.c`

- connections are asymmetrical: one process requests a connection, the other process accepts the request
- one socket is created for each accepted request
- mark socket as willing to accept connections using `listen(2)`
- pending connections are then `accept(2)`ed
- `accept(2)` will block if no connections are available
- `select(2)` to check if connection requests are pending

## Sockets: Other Useful Functions

---

I/O on sockets is done on descriptors, just like regular I/O, ie the typical `read(2)` and `write(2)` calls will work. In order to specify certain flags, some other functions can be used:

- `send(2)`, `sendto(2)` and `sendmsg(2)`
- `recv(2)`, `recvfrom(2)` and `recvmsg(2)`

To manipulate the options associated with a socket, use `setsockopt(2)`:

Option	Description
<code>SO_DEBUG</code>	enables recording of debugging information
<code>SO_REUSEADDR</code>	enables local address reuse
<code>SO_REUSEPORT</code>	enables duplicate address and port bindings
<code>SO_KEEPALIVE</code>	enables keep connections alive
<code>SO_DONTROUTE</code>	enables routing bypass for outgoing messages
<code>SO_LINGER</code>	linger on close if data present
<code>SO_BROADCAST</code>	enables permission to transmit broadcast messages
<code>SO_OOBINLINE</code>	enables reception of out-of-band data in band
<code>SO_SNDBUF</code>	set buffer size for output
<code>SO_RCVBUF</code>	set buffer size for input
<code>SO_SNDLOWAT</code>	set minimum count for output
<code>SO_RCVLOWAT</code>	set minimum count for input
<code>SO_SNDTIMEO</code>	set timeout value for output
<code>SO_RCVTIMEO</code>	set timeout value for input
<code>SO_TIMESTAMP</code>	enables reception of a timestamp with datagrams
<code>SO_TYPE</code>	get the type of the socket (get only)
<code>SO_ERROR</code>	get and clear error on the socket (get only)

## System V IPC

---

Three more types of IPC originating from System V:

- Message Queues
- Semaphores
- Shared Memory

All three use *IPC structures*, referred to by an *identifier* and a *key*.

Since these structures are not known by name, special system calls (`msgget(2)`, `semop(2)`, `shmat(2)`, etc.) and special userland commands (`ipcrm(1)`, `ipcs(1)`, etc.) are necessary.

## System V IPC: Message Queues

---

- linked list of messages stored in the kernel
- create or open existing queue using `msgget(2)`
- add message at end of queue using `msgsnd(2)`
- control queue properties using `msgctl(2)`
- receive messages from queue using `msgrcv(2)`

The message itself is contained in a user-defined structure such as

```
struct mymsg {
    long mtype;    /* message type */
    char mtext[1]; /* body of message */
};
```

## System V IPC: Semaphores

---

A semaphore is a counter used to provide access to a shared data object for multiple processes. To obtain a shared resource a process needs to do the following:

1. Test semaphore that controls the resource.
2. If value of semaphore  $> 0$ , decrement semaphore and use resource; increment semaphore when done
3. If value  $== 0$  sleep until value  $> 0$

Semaphores are obtained using `semget(2)`, properties controlled using `semctl(2)`, operations on a semaphore performed using `semop(2)`.

## System V IPC: Shared Memory

---

- fastest form of IPC
- access to shared region of memory often controlled using semaphores
- obtain a shared memory identifier using `shmget(2)`
- catchall for shared memory operations: `shmctl(2)`
- attach shared memory segment to a processes address space by calling `shmat(2)`
- detach it using `shmdt(2)`

## More Information

---

- `/usr/share/doc/psd/20.ipctut`
- `/usr/share/doc/psd/21.ipc`
- `http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/0650/ \`  
`bks/SGI_Developer/books/T_IRIX_Prog/sgi_html/ch02.html`