CS631 - Advanced Programming in the UNIX Environment
Interprocess Communication II

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@cs.stevens.edu
https://stevens.netmeister.org/631/

Pipes: `pipe(2)`

```
#include <unistd.h>

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

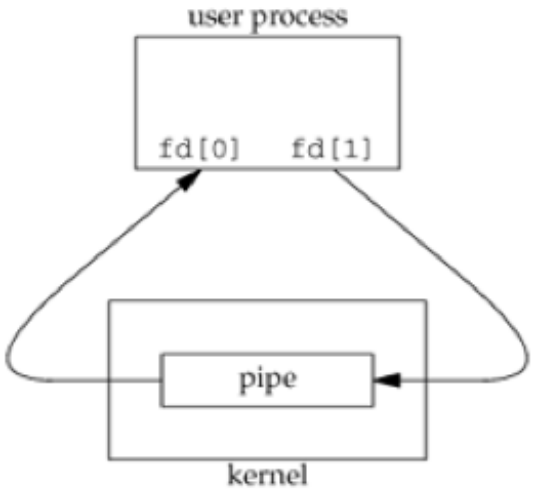
Returns: 0 if OK, -1 otherwise

- oldest and most common form of UNIX IPC
- half-duplex (on some versions full-duplex)
- can only be used between processes that have a common ancestor
- can have multiple readers/writers (`PIPE_BUF` bytes are guaranteed to not be interleaved)

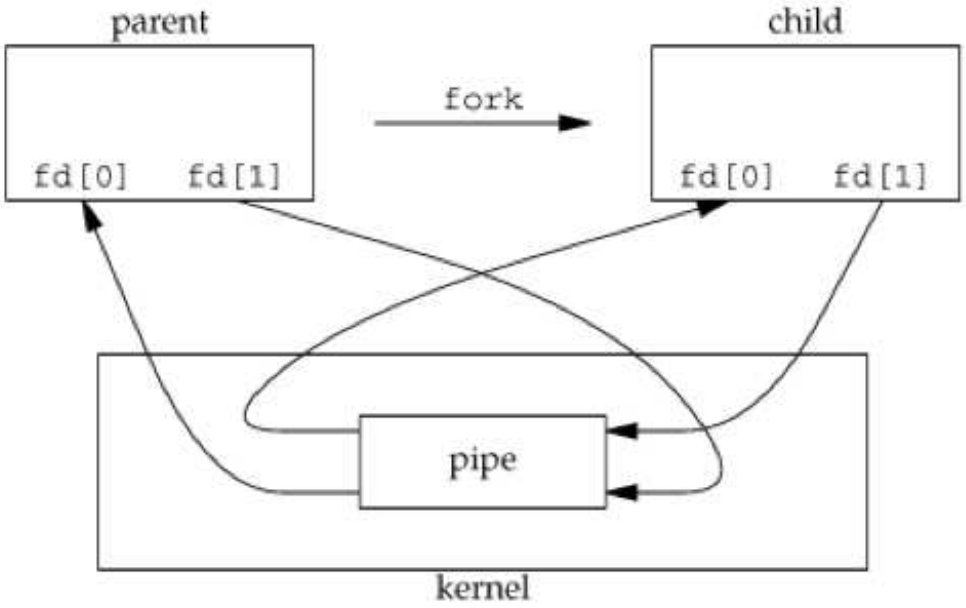
Behavior after closing one end:

- `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`.

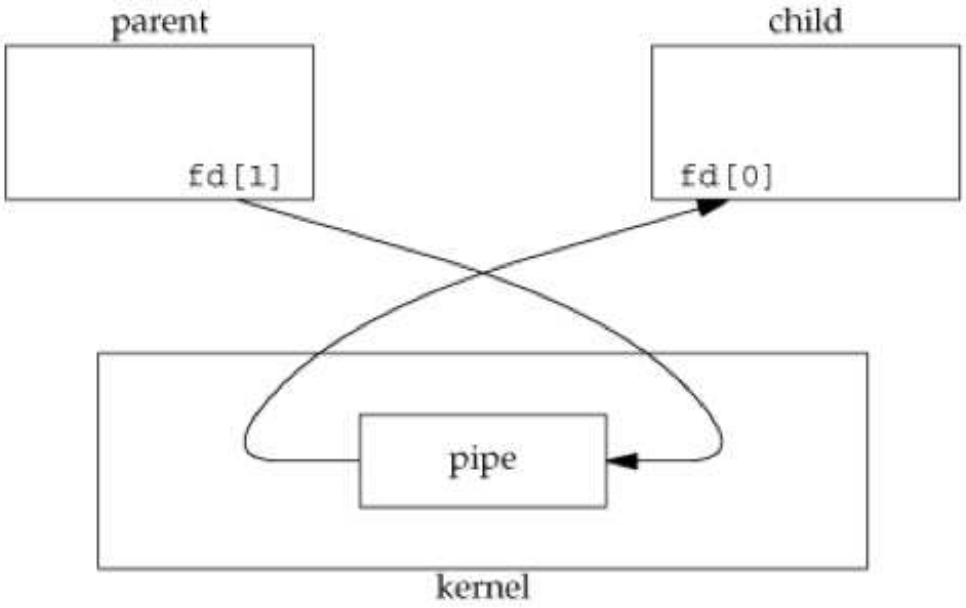
Pipes: `pipe(2)`



Pipes: `pipe(2)`



Pipes: `pipe(2)`



Sockets: `socketpair(2)`

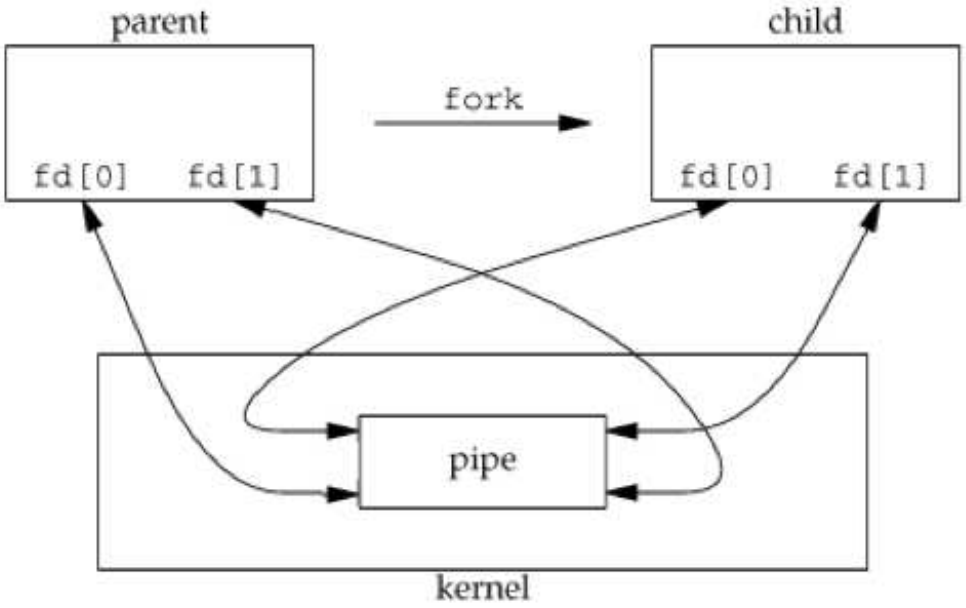
```
#include <sys/socket.h>

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

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

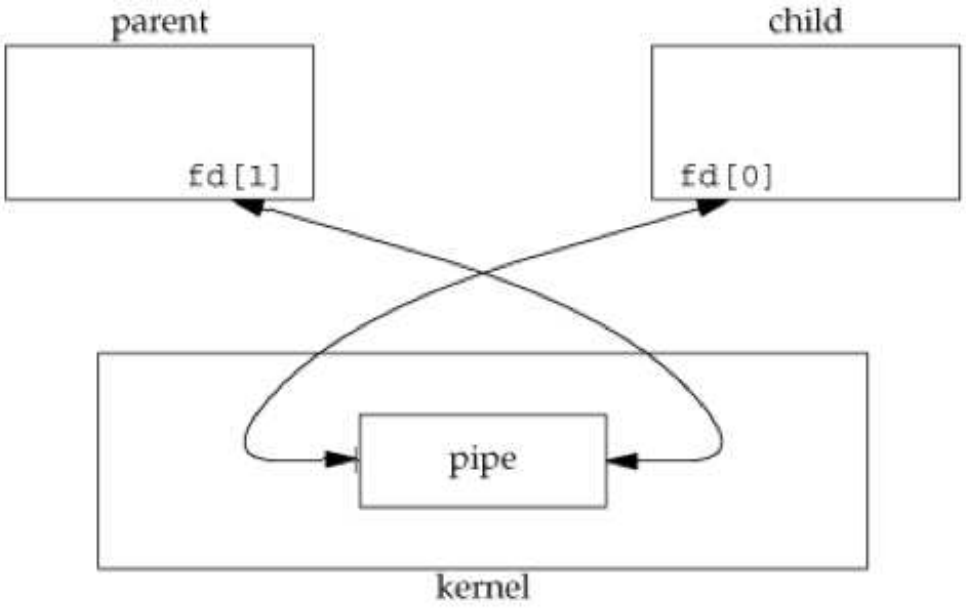
The descriptors used in referencing the new sockets are returned in *sv[0]* and *sv[1]*. The two sockets are indistinguishable.

`This call is currently implemented only for the UNIX domain.`

Sockets: `socketpair(2)`



Sockets: `socketpair(2)`



Sockets: `socketpair(2)`

```
$ cc -Wall socketpair.c
$ ./a.out
78482 --> sending: In Xanadu, did Kubla Khan . . .
78483 --> sending: A stately pleasure dome decree . . .
78483 --> reading: In Xanadu, did Kubla Khan . . .
78482 --> reading: A stately pleasure dome decree . . .
$
```

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_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_RAW	access to internal network protocols and interfaces
...	...

Sockets: Datagrams in the UNIX/LOCAL domain

```
1$ cc -Wall udgramsend.c -o send
1$ cc -Wall udgramread.c -o read
1$ ./read
socket --> socket

2$ ls -l socket
srwxr-xr-x  1 jans  users   0 Oct 31 19:17 socket
2$ ./send socket
2$

--> The sea is calm tonight, the tide is full...
1$
```

Sockets: Datagrams in the UNIX/LOCAL domain

- 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

```
1$ cc -Wall dgramsend.c -o send
1$ cc -Wall dgramread.c -o read
1$ ./read
Socket has port #64293

2$ netstat -na | grep 64293
udp4       0      0  *.64293                *.*
2$ ./send localhost 64293
2$

--> The sea is calm tonight, the tide is full...
1$
```

(Compare observed packets via `tcpdump(8)`.)

Sockets: Datagrams in the Internet Domain

- 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)
- you can (try to) send packets without anything listening (connectionless, unreliable)

Sockets: Connections using stream sockets

```
1$ cc -Wall streamread.c -o read
1$ cc -Wall streamwrite.c -o write
1$ ./read
Socket has port #65398

2$ ./write localhost 65398
2$ ./write localhost 65398
--> Half a league, half a league . . .
Ending connection
--> Half a league, half a league . . .
Ending connection

2$ nc localhost 65398
moo
2$
```

Sockets: Connections using stream sockets

- 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

I/O Multiplexing

Standard I/O loop:

```
while ((n = read(fd1, buf, BUFFSIZE)) > 0) {
        if (write(fd2, buf, n) != n) {
                fprintf(stderr, "write error\n");
                exit(1);
         }
}
```

Suppose you want to read from multiple file descriptors - now what?

I/O Multiplexing

When handling I/O on multiple file descriptors, we have the following options:

- blocking mode: open one fd, block, wait (possibly forever), then test the next fd
- fork and use one process for each, communicate using signals or other IPC
- non-blocking mode: open one fd, immediately get results, open next fd, immediately get results, sleep for some time
- asynchronous I/O: get notified by the kernel when either fd is ready for I/O

I/O Multiplexing

Instead of blocking forever (undesirable), using *non-blocking* mode (busy-polling is inefficient) or using *asynchronous I/O* (somewhat limited), we can:

- build a set of file descriptors we're interested in
- call a function that will return if any of the file descriptors are ready for I/O (or a timeout has elapsed)

I/O Multiplexing

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

int select(int maxfdp1, fd_set *readfds, fd_set *writefds
           fd_set *exceptfds, struct timeval *tvptr);
```

Returns: count of ready descriptors, 0 on timeout, -1 otherwise

Arguments passed:

- which descriptors we're interested in
- what conditions we're interested in
- how long we want to wait
  - `tvptr == NULL` means wait forever
  - `tvptr->tv_sec == tvptr->tv_usec == 0` means don't wait at all
  - wait for specified amount of time

`select(2)` tells us both the total count of descriptors that are ready as well as which ones are ready.

I/O Multiplexing

- filedescriptor sets are manipulated using the `FD_*` functions/macros
- read/write sets indicate readiness for read/write; *except* indicates an exception condition (for example OOB data, certain terminal events)
- EOF means ready for read - `read(2)` will just return 0 (as usual)
- `pselect(2)` provides finer-grained timeout control; allows you to specify a signal mask (original signal mask is restored upon return)
- `poll(2)` provides a conceptually similar interface

See also: `https://daniel.haxx.se/docs/poll-vs-select.html`

Sockets: Connections using stream sockets

```
1$ cc -Wall strchkread.c -o read
1$ ./read
Socket has port #65398
Do something else
Do something else
2$ ./write localhost 65398
2$ ./write localhost 65398
-> Half a league, half a league . . .
Ending connection
Do something else
--> Half a league, half a league . . .
Ending connection
^C
1$
```

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

Final Project

Write a simple web server.

`https://stevens.netmeister.org/631/f19-group-project.html`

More Information

Reading:

- `https://stevens.netmeister.org/631/ipc.pdf`
- `https://ops.tips/blog/how-linux-creates-sockets/`
- `https://ops.tips/blog/how-linux-tcp-introspection/`
- `https://beej.us/guide/bgipc/html/single/bgipc.html`

Exercises:

- Revisit HW2; try to implement it using a `socketpair(2)`
- What happens if you change the `domain`, `type`, and `protocol` arguments to `socketpair(2)`?
- Our examples use the `INET` domain. Change the examples to be dual-stack (`INET` and `INET6`).
- Even though communications via `localhost` work just fine, make sure to verify network communications across the internet.