Pipes: pipe(2)

```c
#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)
Pipes: `pipe(2)`
Pipes: pipe(2)
Pipes: pipe(2)
Pipes: pipe(2)

$ cc -Wall pipe1.c
$ ./a.out
P=> Parent process with pid 23988 (and its ppid 7474).
P=> Sending a message to the child process (pid 23989):
C=> Child process with pid 23989 (and its ppid 23988).
C=> Reading a message from the parent (pid 23988):
Hello child! I’m your parent pid 23988!
$
Pipes: pipe(2)

$ cc -Wall pipe1.c
$ ./a.out

P=> Parent process with pid 23984 (and its ppid 7474).
P=> Sending a message to the child process (pid 23985):
C=> Child process with pid 23985 (and its ppid 1).
C=> Reading a message from the parent (pid 1):
Hello child! I’m your parent pid 23984!
$
Pipes: pipe(2)

$ cc -Wall pipe1.c
$ ./a.out

P=> Parent process with pid 23986 (and its ppid 7474).
P=> Sending a message to the child process (pid 23987):
C=> Child process with pid 23987 (and its ppid 23986).
C=> Reading a message from the parent (pid 1):
Hello child! I'm your parent pid 23986!
$
Pipes: pipe(2)

A more useful example: displaying some content using the user's preferred pager. (Look, Ma, no temporary files!)

```bash
$ cat pipe2.c | ${PAGER:-/usr/bin/more}
$ cc -Wall pipe2.c
$ echo $PAGER

$ ./a.out pipe2.c
[...
^Z
$ ps -o pid,ppid,command
  PID   PPID  COMMAND
22306  26650  ./a.out pipe2.c
22307  22306  more
23198  26650  ps -o pid,ppid,command
26650  26641  -ksh
$ fg
$ env PAGER=/bin/cat ./a.out pipe2.c
```
Pipes: `pipe(2)`

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

```c
#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 (nowadays frequently implemented using sockets or full-duplex pipes)
- `type` one of “r” or “w” (or “r+” for bi-directional communication, if available)
- `cmd` passed to `/bin/sh -c`
Pipes: `popen(3)` and `pclose(3)`

```bash
$ cc -Wall popen.c
$ echo $PAGER

$/a.out popen.c
[...]
$ env PAGER=/bin/cat ./a.out popen.c
[...]
$
```
Pipes: `popen(3)` and `pclose(3)`

```
$ cc -Wall popen.c
$ echo $PAGER

$ ./a.out popen.c
[...]
$ env PAGER=/bin/cat ./a.out popen.c
[...]
$ env PAGER=/bin/cat/foo ./a.out popen.c
sh: /bin/cat/foo: Not a directory
$ env PAGER="more; touch /tmp/boo" ./a.out popen.c
$ env PAGER="more; rm /etc/passwd 2>/dev/null" ./a.out popen.c
```
FIFOs: `mkfifo(2)`

```c
#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 (i.e., `open(2)`, `read(2)`, `write(2)`, `unlink(2)` etc.)
- used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files
FIFOs: `mkfifo(2)`

Example: split input into sets

![Diagram showing input file feeding into `prog1` which then distributes input to `prog2` and `prog3`.](image-url)
FIFOs: `mkfifo(2)`

Example: split input into sets

$ mkfifo fifo
$ grep pattern fifo > match &
$ gzcat file.gz | tee fifo | grep -v pattern > nomatch
System V IPC

Three types of IPC originating from System V:
- Semaphores
- Shared Memory
- Message Queues

All three use *IPC structures*, referred to by an *identifier* and a *key*; all three are (necessarily) limited to communication between processes on one and the same host.

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: 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: Semaphores

$ cc -Wall semdemo.c
1$ ./a.out

2$ ./a.out

$ ipcs -s
IPC data flow
System V IPC: Shared Memory
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)`
System V IPC: Shared Memory

$ cc -Wall shmdemo.c
$ ./a.out "Cow says: 'Moo!'"
$ ./a.out
$ ipcs -m
System V IPC: Shared Memory

$ cc -Wall memory-layout.c
$ ./a.out
array[] from 804a080 to 8053cc0
stack around bffff9e4
malloced from 8053cc8 to 806c368
shared memory attached from 40162000 to 4017a6a0
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

```c
struct mymsg {
    long mtype;  /* message type */
    char mtext[512]; /* body of message */
};
```
System V IPC: Message Queues

$ cc -Wall msgsend.c -o msgsend
$ cc -Wall msgrecv.c -o msgrecv
$ ipcs -q
$ ./msgsend 1
$ ipcs -q
$ ./msgsend 1
$ ipcs -q
$ ./msgrecv 1
$ ipcs -q
$ ./msgrecv 1
$ ipcs -q
$ ./msgrecv 1
^C
$ ipcs -q
$ ./msgsend 2
$ ipcrm -q <msqid>
Sockets: `socketpair(2)`

```c
#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.

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 Kublai Khan . . .
78483 --> sending: A stately pleasure dome decree . . .
78483 --> reading: In Xanadu, did Kublai 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:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_LOCAL</td>
<td>local (previously UNIX) domain protocols</td>
</tr>
<tr>
<td>PF_INET</td>
<td>ARPA Internet protocols</td>
</tr>
<tr>
<td>PF_INET6</td>
<td>ARPA IPv6 (Internet Protocol version 6) protocols</td>
</tr>
<tr>
<td>PF_ARP</td>
<td>RFC 826 Ethernet Address Resolution Protocol</td>
</tr>
</tbody>
</table>

Some of the currently defined types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCK_STREAM</td>
<td>sequenced, reliable, two-way connection based byte streams</td>
</tr>
<tr>
<td>SOCK_DGRAM</td>
<td>connectionless, unreliable messages of a fixed (typically small) maximum length access to internal network protocols and interfaces</td>
</tr>
<tr>
<td>SOCK_RAW</td>
<td>...</td>
</tr>
</tbody>
</table>

Lecture 07: Interprocess Communication

October 24, 2016
Sockets: Datagrams in the UNIX/LOCAL domain

1$ cc -Wall udgramsndsend.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$
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 ntohss(3) (which may be noops)
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 accepted.
- `accept(2)` will block if no connections are available.
- `select(2)` to check if connection requests are pending.
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)`:

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

- http://www.cs.cf.ac.uk/Dave/C/node25.html

HW#3: write the basic framework for your final project.