8. Pipes and Unbuffered IO

Pipes consist of two file descriptors, a file descriptor to write to the pipe and another to read from it. These two together form an unbuffered communication channel that can be used for inter-process communication.

The `pipe` system call can be used to form a pipe.

When all the write ends of the pipe are closed, `EOF` will be sent to the pipe and calling `read` on it will return `0`.

Data in a pipe can be read only once. If multiple processes call `read` on the read end of the same pipe, then the operating system determines which process reads what data.

A forked process inherits the parent's open file descriptors. When a process terminates, all its open file descriptors are closed.

A `read` call to an empty pipe and a `write` call to a full pipe will block.

A file descriptor is an index into the process' open file descriptor table. This table stores pointers to data structures containing information about open files.

Useful System Calls

`open` and `close`

The prototype of `open` is `int open(const char *path, int oflag, ...);`

The file name specified by `path` is opened for reading and/or writing, as specified by the argument `oflag`; the file descriptor is returned to the calling process. The `oflag` argument may indicate that the file is to be created if it does not exist (by specifying the `O_CREAT` flag). In this case, `open` requires an additional argument `mode_t mode`.

The flags specified for the `oflag` argument must include exactly one o the following file access modes:

- `O_RDONLY` - open for reading only
- `O_WRONLY` - open for writing only
- `O_RDWR` - open for reading and writing

In addition any combination of the following values can be or'ed in `oflag`:

- `O_APPEND` - append on each write
- `O_CREAT` - create file if it does not exist
- `O_TRUNC` - truncate size to `0`

If successful, `open` returns a non-negative integer, termed a file descriptor. It returns `-1` on failure. The file pointer (used to mark the current position within the file) is set to the beginning of the file. When a new file is created, it is given the group of the directory which contains it.

The prototype of `close` is `int close(int fildes);`

`close` deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated. For example, on the last close of a file the current seek pointer associated with the file is lost; on the last close of a socket associated naming information and queued data are discarded

Upon successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

---

dup2

The prototype of `dup2` is `int dup2(int fildes, int fildes2);`

`dup2` duplicates an existing object descriptor and returns its value to the calling process. If both are equal, then just `dup2` returns the second file descriptor. Otherwise, it duplicates the first file descriptor. If the second file descriptor is already in use, it is first deallocated as if a `close` call had been done first. However, it does not close the first file descriptor.

Upon successful completion, the new file descriptor is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

```
// Redirecting a file to standard input
int fd = open("file", O_RDONLY);
dup2(fd, STDIN_FILENO);
close(fd);
```

---

read

The prototype of `read` is `ssize_tread(int fildes, void *buf, size_t nbyte);`

`read` attempts to read `nbyte` bytes of data from the object referenced by the descriptor `fildes` into the buffer pointed to by `buf`.

On objects capable of seeking, the `read` starts at a position given by the pointer associated with `fildes`. Upon return from `read`, the pointer is incremented by the number of bytes actually read. Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

`read` will fail if the parameter `nbyte` exceeds `INT_MAX`, and it will not attempt a partial read.

Upon successful completion, `read` returns the number of bytes actually read and placed in the buffer. Upon reading end-of-file, `0` is returned. Otherwise, a `-1` is returned and the global variable errno is set to indicate the error. `ssize_t` is signed `size_t`.

`write`

The prototype of `write` is `ssize_t write(int fildes, const void *buf, size_t nbyte);`

`write` attempts to write `nbyte` bytes of data to the object referenced by the descriptor `fildes` from the buffer pointed to by `buf`.

On objects capable of seeking, `write` starts at a position given by the pointer associated with `fildes`. Upon return from `write`, the pointer is incremented by the number of bytes which were written. Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

When using non-blocking I/O on objects, such as sockets (refer to 9. Sockets), that are subject to flow control, `write` and may write fewer bytes than requested.

`write` will fail if the parameter `nbyte` exceeds `INT_MAX`, and it will not attempt a partial write.

Upon successful completion the number of bytes which were written is returned. Otherwise, a `-1` is returned and the global variable errno is set to indicate the error.

`pipe`

The prototype of `pipe` is `int pipe(int fildes[2]);`

`pipe` creates a pipe (an object that allows unidirectional data flow) and allocates a pair of file descriptors. Memory for the `fildes` array needs to be allocated before `pipe` is called. The first descriptor connects to the read end of the pipe; the second connects to the write end. Data written to `fildes[1]` appears on (i.e., can be read from) `fildes[0]`.

The pipe persists until all of its associated descriptors are closed. A pipe whose read or write end has been closed is considered widowed. Writing on such a pipe causes the writing process to receive a `SIGPIPE` signal. Widowing a pipe is the only way to deliver end-of-file to a reader: after the reader consumes any buffered data, reading a widowed pipe returns a zero count.

On successful creation of the pipe, `0` is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error.

`select`, `FD_SET`, `FD_ISSET`, `FD_CLR`, `FD_ZERO`

The prototype of `select` is `int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds, fd_set *restrict errorfds, struct timeval *restrict timeout);`

`select` examines the I/O descriptor sets whose addresses are passed in `readfds`, `writefds`, and `errorfds` to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first `nfds` descriptors are checked in each set; i.e., the descriptors from `0` through `nfds-1` in the descriptor sets are examined.

On return, `select` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. `select` returns the total number of ready descriptors in all the sets.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets:

- `FD_ZERO(&fdset)` initialises a descriptor set `fdset` to the null set.
- `FD_SET(fd, &fdset)` includes a particular descriptor `fd` in `fdset`.
- `FD_CLR(fd, &fdset)` removes `fd` from `fdset`.
- `FD_ISSET(fd, &fdset)` is non-zero if `fd` is a member of `fdset`, `0` otherwise.

The behaviour of these macros is undefined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`, which is normally at least equal to the maximum number of descriptors supported by the system.

If timeout is not a null pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a null pointer, the select blocks indefinitely. To effect a poll, the timeout argument should be not be a null pointer, but it should point to a zero-valued `timeval` structure.

For example, a timeout of `1` second can be set using

```
struct timeval timeout = { .tv_sec = 1, .tv_usec = 0 };
select(nfds, &read_fds, &write_fds, &errorfds, &timeout);
```

Any of `readfds`, `writefds`, and `errorfds` may be given as null pointers if no descriptors are of interest.

`select` returns the number of ready descriptors that are contained in the descriptor sets, or `-1` if an error occurred. If the time limit expires, `select` returns `0`. If `select` returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified and `errno` will be set to indicate the error.
