File descriptors

Most I/O on Unix systems takes place through the read and write system calls. All read and write operations must be performed on file descriptors, non-negative integers which are created via a call to open (see below). File descriptors remain bound to files even when files are renamed or deleted or undergo permission changes that revoke access. By convention, file descriptors numbers 0, 1, and 2 correspond to standard input, standard output, and standard error respectively. Thus a call to printf will result in a write to file descriptor 1. type.c (below) shows a very simple program that prints the contents of files to the standard output—just like the UNIX cat command. The function typefile uses four system calls to copy the contents of a file to the standard output.High-level I/O functions such as fread and fprintf are implemented in terms of read and write. Note that not all IPC mechanisms properly implement these semantics.

- int open(char *path, int flags, ...);
  The open system call requests access to a particular file. path specifies the name of the file to access; flags determines the type of access being requested—in the case of this example read-only access. open ensures that the named file exists (or can be created, depending on flags) and checks that the invoking user has sufficient permission for the mode of access. If successful, open returns a file descriptor. If unsuccessful, open returns −1 and sets the global variable errno to indicate the nature of the error. The routine perror will print “filename: error message” to the standard error based on errno.

- int read (int fd, void *buf, int nbytes);
  read will read up to nbytes bytes of data into memory starting at buf. It returns the number of bytes actually read, which may very well be less than nbytes. The case in which read returns fewer than nbytes is often called a “short read” and is a common source of errors. If read returns 0, this indicates an end of file. If it returns −1, this indicates an error.

- int write (int fd, void *buf, int nbytes);
  write will write up to nbytes bytes of data at buf to file descriptor fd. It returns the number of bytes actually written, which unfortunately may be less than nbytes if the file descriptor is non-blocking. Write returns −1 to indicate an error.

- int close (int fd);
  close deallocates a file descriptor. Systems typically limit each process to 64 file descriptors by default (though the limit can sometimes be raised substantially with the setrlimit(2) system call). Thus, it is a good idea to close file descriptors after their last use so as to prevent “too many open files” errors.

type.c: Copy file to standard output

```
#include <unistd.h>
```

```
#include <fcntl.h>
#include <stdio.h>

void typefile (char *filename) {
      int fd, nread;
      char buf[1024];
      fd = open (filename, O_RDONLY);
      if (fd == -1) {
            perror (filename); 2 return;
      }
      while ((nread = read (fd, buf, sizeof (buf))) > 0)
            write (1, buf, nread);
      close (fd);
}

int main (int argc, char **argv) {
      int argno;
      for (argno = 1; argno < argc; argno++)
            typefile (argv[argno]); exit (0);
}
```

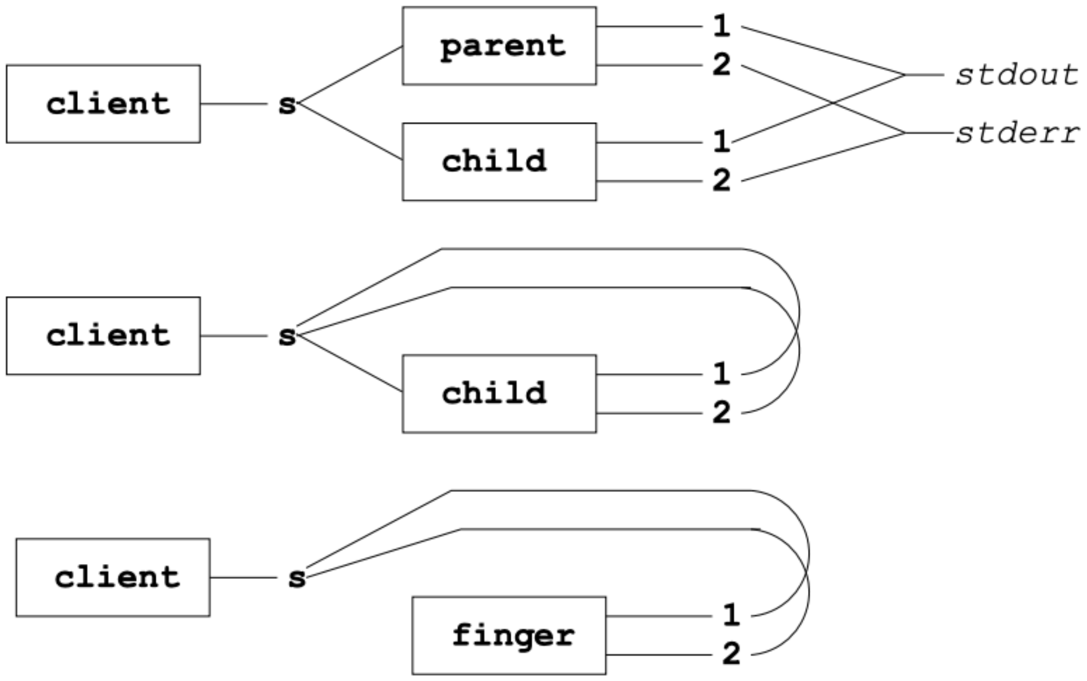

Figure 1: File descriptors are inherited by the child processes.

Figure 2: The child process has called dup2 to connect file descriptors 1 and 2 which normally correspond to standard output and standard error to the network socket.

Figure 3: The child has called execl and the local finger process runs, sending its standard output over s.

- int fork (void);
  fork creates a new process, identical to the current one. fork returns twice: in the old process, fork returns a process ID of the new process. In the new or “child” process, fork returns 0. fork returns −1 if there is an error.

- int dup2(int oldfd, int newfd);
  dup2 closes file descriptor number newfd, and replaces it with a copy of oldfd. When the second argument is 1, this changes the destination of the standard output. When that argument is 2, it changes the standard error.

- int execl(char *path, char *arg0, ..., NULL);
  The execl system call runs a command—as if you had typed it on the command line. The command executed will inherit all file descriptors except those with the close-onexec flag set. execl replaces the currently executing program with the one specified by path. On success, it therefore doesn’t return. On failure, it returns −1.

Additional Resources

See the man pages for…
File Descriptors: open(2), close(2), dup(2), dup2(2), lseek(2), read(2), write(2)
Pipe: pipe(2), pipe(7), popen(2)

open(2) is a must read