Universidade de Coimbra
Departamento de Engenharia Electrotécnica e de Computadores

Sistemas Operativos - 2019/2020
Support Document Nº1
Files, Pipes, FIFOs, I/O Redirection, and Unix Sockets

1 Objectives

The objective of this document is to present information concerning files, pipes, I/O Redirection, and Unix Sockets.

2 Material for Preparation

- [Robbins e Robbins, 2003,
  Chapter 4, "UNIX I/O";
  Chapter 5, "Files and Directories";
  Chapter 6, "UNIX Special Files";
  Chapter 18, "Connection-Oriented Communication";
  Chapter 20, "Connectionless Communication and Multicast";
  Appendix B, "Restart Library"].
- [Marshall, 1999,
  "Input and Output (I/O): `stdio.h`";
  "Interprocess Communication (IPC), Pipes";
  "IPC: Sockets"].
- [Stevens, 1990,
  Section 3.4 "Pipes" (pp. 102-109);
  Section 3.5 "FIFOs" (pp. 110-115);
  Chapter 6 "Berkeley Sockets"].
- [Kernighan e Ritchie, 1988,
  Chapter 7, "Input and Output" (pp. 151-168);
  Chapter 8, "The Unix System Interface" (pp. 169-189);
  Appendix B, "Standard Library" (pp. 241-258)].
- [Araújo, 2014,
  "UNIX Programming: Files and Pipes"].

Students that do not have knowledge of programming in C should overcome this situation. Reading the book [Kernighan e Ritchie, 1988] (ANSI C) is strongly recommended to obtain knowledge about C programming, and in particular about pointers and the main standard C functions (mathematical, string manipulation, and file manipulation).

3 Files in C and Unix

In C, files are referred either by pointers or by (file) descriptors. The ISO C standard library I/O functions such as `fopen()`, `fscanf()`, `fprintf()`, `fread()`, `fwrite()`, and `fclose()` use pointers. The UNIX I/O functions such as `open()`, `read()`, `write()`, `close()`, and `ioctl()` use file descriptors.

A file descriptor is an integer value and is a pointer to a table of file descriptors (Figure 1) that exists in the memory space of each program. This table is accessed through the use of certain functions that use the descriptor (e.g. "`int fp = open("myfile.c", O_RDONLY);`"). Each entry in the file descriptors

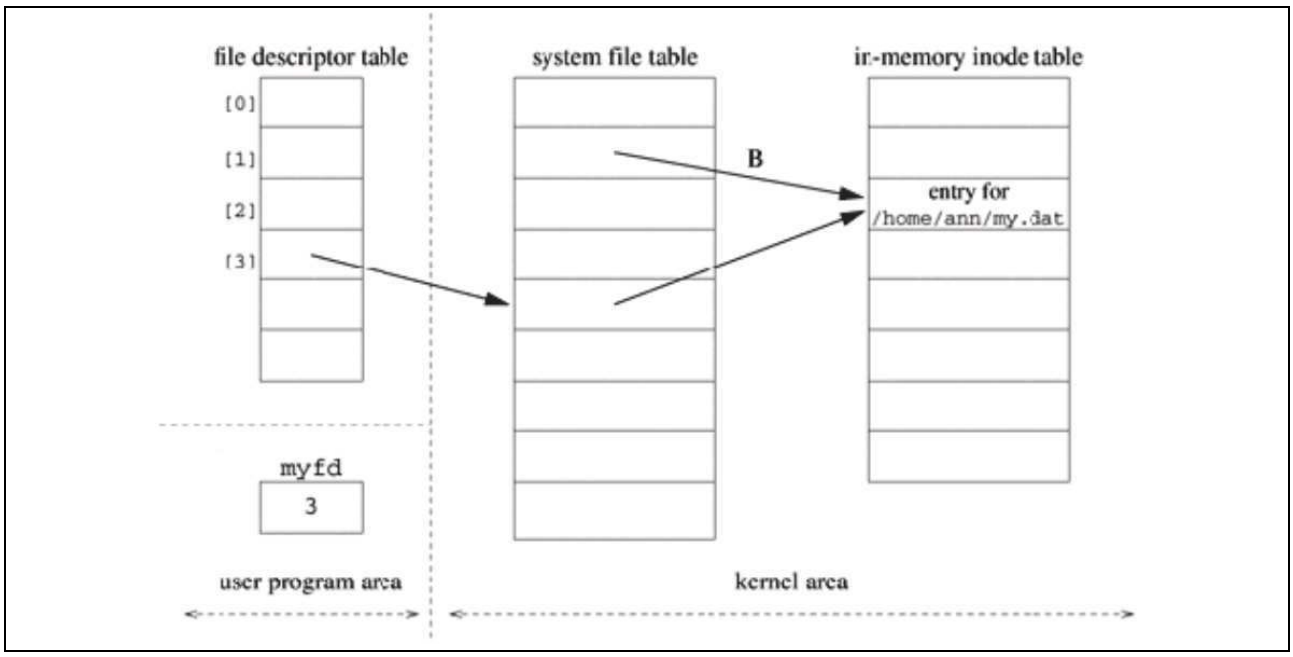

Figure 1: *File descriptor table*, *system file table*, *inode table*.

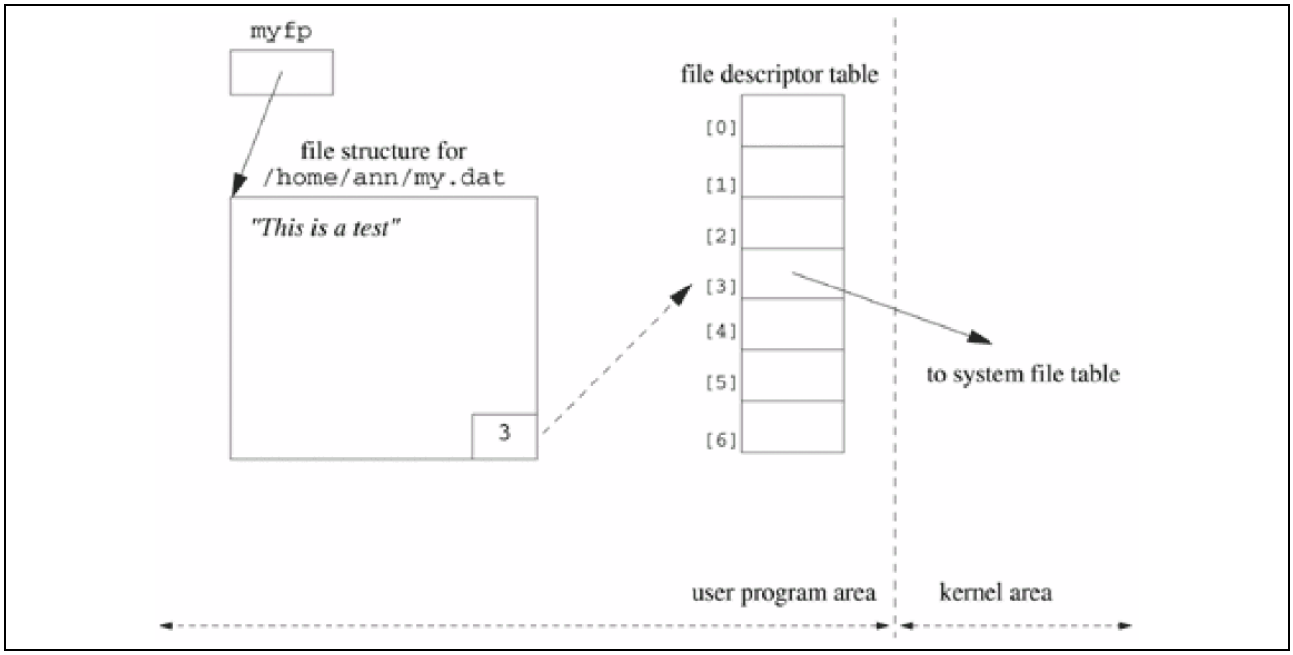

Figure 2: Diagram of a pointer to a file after a `fopen()`.

table is a reference for the *System File Table* that maintains a list of all the references to all the files that are being presently referenced and the present write or read position in each of these references to files. Each element of this table makes a reference the a *In-memory inode table* that maintains a list of all the active files in the system.

Two entries in the *System File Table* may reference the same element of the *inodes* table, because one or more programs may have open the same file for reading and may be reading from different positions. Typically, if two programs are writing to the same file there are no guarantees that they are not overwriting one over the other.

A pointer to a file is a pointer to a structure (Figure 2) called `FILE` that resides in the memory area of the program that may be directly accessed by the user (contrary to the area where the descriptors table resides which may no be directly accessed by the user). The `FILE` structure contains a buffer and

a file descriptor. It is this file descriptor that is used to write to (and read from) disk. In some way the pointer to a file is a handle for a handle.

When a program calls for example a function "`fprintf(myfp, "This is a test");`", a message "This is a test" is not immediately written to disk. It is writen in the buffer of the `FILE` structure. When this buffer is full the I/O subsystem calls function `write()` with the file descriptor of the file, and only in this instant are the data writen to disk. A call to function `fflush()` forces the system to empty the buffer and to write its contents to disk, even if the buffer is not full.

The `chmod` command can be used to change file access permissions. The permissions of files can be seen using the "`ls -l`" command, for example.

The following Linux system calls can be used for file manipulation (see the suggested literature and the man pages):

- `creat()` - create a file.
- `open()` - specifies a file to be manipulated.
- `close()` - release file descriptor.
- `read()` - input a block of information from a file.
- `write()` - output a block of information to a file.
- `lseek()` - reposition read/write file offset.
- `ioctl()` - control device.
- `unlink()` - delete a name and possibly the file it refers to.

The following C library functions can be used for file manipulation: `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fprintf()`, `fscanf()`, `feof()`, `fflush()`, `fsync()`, `fseek()`, `ftell()`, `rewind()`, `fgetpos()`, `fsetpos()`.

Observe the following example concerning the use of the write and read functionalities using the standard C library.

```
/*--------------------------------------------------------------------------*/
#include <stdio.h>

/*--------------------------------------------------------------------------*/
/*                  write_to_file()                                         */
/*--------------------------------------------------------------------------*/
int write_to_file(){

  int account;       /* account number */
  char name[30];     /* account name */
  double balance;    /* account balance */

  FILE *cfPtr;       /* cfPtr = clients.dat file pointer */

  /* fopen opens file. Exit program if unable to create file  */
  if ( ( cfPtr = fopen("clients.dat","w") ) == NULL ) {
    printf("File could not be opened\n");
  }else{ /* end if */
    printf("Enter the account, name, and balance.\n");
    printf("Enter EOF to end input.\n");
    printf("? ");
    scanf("%d%s%lf",&account,name,&balance);
    /* write account, name and balance into file with fprintf */
    while ( !feof( stdin ) ){
```

```
      fprintf(cfPtr,"%d %s %.2f\n",account,name,balance);
      printf("? ");
      scanf("%d%s%lf",&account,name,&balance);
    } /* end while */
    fclose( cfPtr ); /* fclose closes file */
  } /* end else */
  return 0; /* indicates successful termination */
}

/*-------------------------------------------------------------------------*/
/*                  read_from_file()                                       */
/*-------------------------------------------------------------------------*/
int read_from_file(){

  int account;       /* account number */
  char name[30];     /* account name */
  double balance;    /* account balance */

  FILE *cfPtr;       /* cfPtr = clients.dat file pointer */

  /* fopen opens file; exits program if file cannot be opened */
  if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
    printf( "File could not be opened\n" );
  }else{ /* end if */ /* read account, name and balance from file */
    printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
    fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
    /* while not end of file */
    while ( !feof( cfPtr ) ) {
      printf( "%-10d%-13s%7.2f\n", account, name, balance );
      fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
    } /* end while */
    fclose( cfPtr ); /* fclose closes the file */
  } /* end else */
  return 0; /* indicates successful termination */
}
/*-------------------------------------------------------------------------*/
```

4 Pipes an FIFOs

A pipe is a unidirectional data channel that can be used for interprocess communication. It is created using the `pipe()` system call (cf. `man pipe`). "`int pipe(int filedes[2])`" returns two file descriptors `fd[0]` and `fd[1]`. The data written to `fd[1]` can be read from `fd[0]` on a first-in-first-out (FIFO) basis.

In many aspects FIFOs are similar to pipes, but there are also differences - study the suggested literature and the man pages. In a C program, named pipes (FIFOs) can be created with the `mknod()` or `mkfifo()` functions (cf. `man 2 mknod`, `man 3 mkfifo`). In the shell, the commands `mknod`, and `mkfifo` (cf. `man 1 mknod`, `man 1 mkfifo`) can be used to create FIFOs. To communicate, processes write and read to a pipe or to a FIFO using the `write()` and `read()` system calls, respectively. If `fifo01` is a FIFO that has already been created, then, in the shell, the command `echo Hello FIFO! >fifo01` will write the string "`Hello FIFO!`" (and an ending newline character) to the `fifo01`, provided that the permissions of the FIFO are appropriately set.

5 I/O Redirection

If a process performs input or output operations to some file descriptor, it is possible to redirect such operations to other file descriptors. This is called I/O redirection. The `dup()`, and `dup2()` system calls

create a copy of a file descriptor and are useful to perform I/O redirection:

- `int dup(int oldfd)` - returns the lowest numbered unused descriptor for the new descriptor.
- `int dup2(int oldfd, int newfd)` - makes `newfd` be the copy of `oldfd`, closing `newfd` first if necessary.

`dup()` and `dup2()` create a copy of file descriptor `oldfd`. After successful return of `dup()` or `dup2()`, the old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using `lseek()` on one of the descriptors, the position is also changed for the other. See usage examples on the suggested literature.

6 Unix Sockets

Sockets provide a very versatile point-to-point, two-way communication between two processes [Stevens, 1990], [Robbins e Robbins, 2003], [Marshall, 1999]. Sockets are a basic component of interprocess and intersystem communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.

Sockets exist in communication domains or protocol families. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. There are several socket domains (see `<sys/socket.h>`). The most used domains are the UNIX (`AF_UNIX`) and Internet (`AF_INET`) domains. The UNIX domain provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. Sockets can also be used to communicate between processes on different systems. The mostly used socket address space between connected systems is called the Internet domain. Internet domain communication uses the TCP/IP Internet protocol suite. In this course we will only deal with sockets in the UNIX domain.

There are several types of sockets. In this "Operating Systems" course we will deal with sockets of type stream (`SOCK_STREAM`). See "`man 2 socket`" for more information on socket domains (protocol families) and socket types.

There are several functions that are used by user processes to send or receive packets and to do other socket operations. For more information see their respective manual pages.

`socket()` creates a socket, `connect()` connects a socket to a remote socket address, the `bind()` function binds a socket to a local socket address, `listen()` tells the socket that new connections shall be accepted, and `accept()` is used to get a new socket with a new incoming connection. `socketpair()` returns two connected anonymous sockets (only implemented for a few local families like `AF_UNIX`).

`send()`, `sendto()`, and `sendmsg()` send data over a socket, and `recv()`, `recvfrom()`, `recvmsg()` receive data from a socket. `poll()` and `select()` wait for arriving data or a readiness to send data. In addition, the standard I/O operations like `write()`, `writev()`, `sendfile()`, `read()`, and `readv()` can be used to read and write data.

`getsockname()` returns the local socket address and `getpeername()` returns the remote socket address. `getsockopt()` and `setsockopt()` are used to get or set socket layer or protocol options. `ioctl()` can be used to set or read some other options.

`close()` is used to close a socket. `shutdown()` closes parts of a full-duplex socket connection.

See the suggested literature (Sec. 2) and the manual pages for more information and usage examples.

References

[Araújo, 2014] Rui Araújo. *Operating Systems*. DEEC-FCTUC, 2014.

[Kernighan e Ritchie, 1988] Brian W. Kernighan e Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, USA, Second ed., 1988.

[Marshall, 1999] A. Dave Marshall. *Programming in C: UNIX System Calls and Subroutines Using C*. Cardiff University, UK, 1999. [Online]. Available: "`http://www.cs.cf.ac.uk/Dave/C/`".

[Robbins e Robbins, 2003] Kay A. Robbins e Steven Robbins. *Unix Systems Programming: Communication Concurrency, and Threads*. Prentice–Hall, Inc, Upper Saddle River, NJ, USA, 2003.

[Stevens, 1990] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, USA, First ed., 1990.