

UNIX input and output

Disk files

In UNIX a disk file is a finite sequence of bytes, usually stored on some nonvolatile medium.

Disk files have names, which are called ‘paths’. We won’t discuss file naming or permissions here

Opening a file

In UNIX, files must be “opened” before being read and “closed” after.

Here is a function call to open a disk file named “/Users/alice/adventures.txt” for reading.

```
int fd = open("/Users/alice/adventures.txt", O_RDONLY) ;
```

Notice that `open` returns an integer.

- If the integer is negative, the file could not be opened.
- If the integer is not negative, it is a **file descriptor**.

A file descriptor is an index into a table called the “descriptor table”.

Each process has its own descriptor table, so the file descriptor only has meaning within the context of a given process.

Reading a file

Once a file is open, the process uses the file descriptor to refer to the open file.

For example, to read up to 10 bytes from the file, I can use

```
unsigned char buffer[10] ;  
int count = read( fd, buffer, 10 ) ;
```

The call to read will return

- -1 if there was an error
- 0 if we have reached the end of the file
- a positive number giving the number of bytes successfully read

Here is a complete program that reads a file and prints its contents in hexadecimal.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <sys/stat.h>
5 #include <unistd.h>
6 #include <errno.h>
7 #include <string.h>
8 #include <stdbool.h>
9
10 void check( bool condition, char* message ) {
11     if( !condition ) {
12         printf( "%s: %s\n", message, strerror( errno ) ) ; }
13     exit( 1 ) ; }
14
15 int main( int argc, char **argv ) {
16     if( argc != 2 ) { printf( "Usage: hexdump file\n" ) ; exit(1) ; }
17     int fd = open( argv[1], O_RDONLY ) ;
18     check( fd >= 0, "Error on open" ) ;
19     const int BUFSIZE = 1024, LINELEN = 20 ;
20     unsigned char buffer[ BUFSIZE ] ;
21     int outCount = 0 ;
22     while( 1 ) {
23         int count = read( fd, buffer, BUFSIZE ) ;
24         check( count >= 0, "Error on read" ) ;
25         if( count == 0 ) break ;
26         for( int i=0 ; i < count ; ++i ) {
27             printf( "%2.2x", buffer[i] ) ;
28             outCount = (outCount + 1) % LINELEN ;
29             if( outCount == 0 ) printf("\n") ; else printf(" ") ; } }
30     if( outCount != 0 ) printf( "\n" ) ;
31     return 0 ; }

```

Notes:

- On line 16, the file is opened, returning a file descriptor.
- On line 19, I allocated space to hold the data read
- On line 22, up to 1024 bytes are read from the file into array buffer. The number of characters read is saved in variable count.
- On line 24, I checked for the end of the file.
- One lines 25–28, each character read by the previous read, is output.
- On line 25 the value of count is between 1 and 1024

inclusive.

Writing files

Writing files is similar to reading.

The following function call will open a file "out.txt" for reading or writing

```
const int flags = O_WRONLY | O_CREAT | O_TRUNC ;
const int mode = S_IRUSR | S_IWUSR ;
int outFD = open( argv[2], flags, mode ) ;
```

The flags indicate that

- The process will only write to the file
- The file should be created if it does not already exist
- The file's length should be truncated to 0 if it already exists.

The mode indicates that if a new file is created then:

- The owner of the file should be allowed to read it.
- The owner of the file should be allowed to write to it.

As before the open method returns a file descriptor or an error indicator.

We can write to a file using the file descriptor.

```
unsigned char buffer[ 10 ] ;
/* Put up to 10 bytes of data into buffer. */
int writeCount = write( outFD, buffer, 10 ) ;
```

This call will write up to 10 bytes to the file.

The result is either

- Less than 0, indicating an error.

- Greater or equal to 0, indicating how many bytes were written.

Here is a program that simply copies the contents of a file to a new file.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <sys/stat.h>
5 #include <unistd.h>
6 #include <errno.h>
7 #include <string.h>
8 #include <stdbool.h>
9
10 void check( bool condition, char* message ) {
11     if( !condition ) {
12         printf( "%s: %s\n", message, strerror( errno ) );
13         exit( 1 ); } }
14
15 int main( int argc, char **argv ) {
16     if( argc != 3 ) { printf( "Usage: copy inFile outFile\n" ); exit(1); }
17     int inFD = open( argv[1], O_RDONLY );
18     check( inFD >= 0, "Error on open" );
19     int outFD = open( argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR );
20     check( outFD >= 0, "Error on open" );
21     const int BUFSIZE = 1024;
22     unsigned char buffer[ BUFSIZE ];
23     while( 1 ) {
24         int count = read( inFD, buffer, BUFSIZE );
25         check( count >= 0, "Error on read" );
26         if( count == 0 ) break;
27         for( int i = 0; i < count; ) {
28             int writeCount = write( outFD, buffer+i, count-i );
29             if( writeCount < 0 && (errno == EINTR || errno == EAGAIN) ) {
30                 writeCount = 0; }
31             check( writeCount >= 0, "Error on write" );
32             i += writeCount; } }
33     return 0; }

```

Notes on the program

- On line 19, I open a file for writing.
- Lines 27 to 32, repeatedly call write until all data in the buffer is written.
- Lines 29 to 30 deal with errors that we can recover from by trying again.

The loop on lines 27–32 is not strictly needed. We could replace it with

```
int writeCount = write( outFD, buffer, count ) ;  
check( writeCount >= 0, "Error on write" ) ;
```

because for disk files, it is always the case that there is an error or all bytes are written. However, as we will see later, we can also use `write` to write to sockets and pipes, and in those cases the `write` may only write some data. So using a loop to write is the preferred method.

Closing files

We can close a file when we no longer need to read or write to it.

```
int err = close( fileDescriptor ) ;
```

After a file is closed its descriptor can not be used until the descriptor is recycled later.

Files are also closed when the process terminates; so I didn't put `close` in my programs.

The wonderful thing about file descriptors

In UNIX we can open things that aren't disk files. For example devices.

A device might be the keyboard, a mouse, a printer, a terminal, a disk drive, a memory card.

We can use the `read` and `write` system calls to read from and write data to devices.

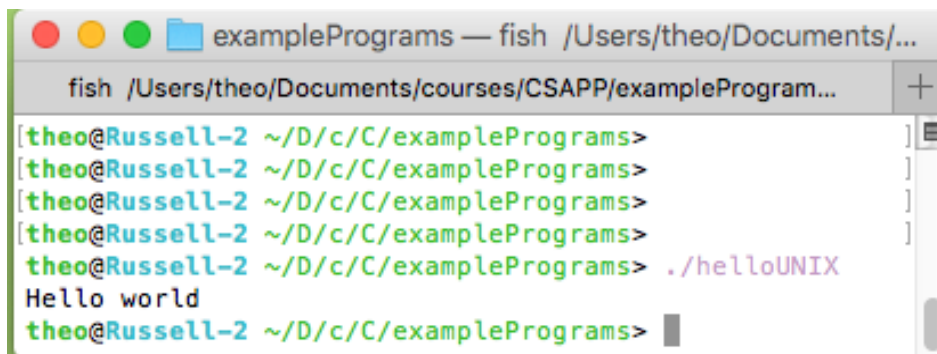
Standard input, standard output, and

standard error

By convention whenever a Unix program is started, it will have 3 files open with descriptors 0, 1, 2

- 0 represents the ‘standard input stream’
- 1 represents the ‘standard output stream’
- 2 represents the ‘standard error stream’

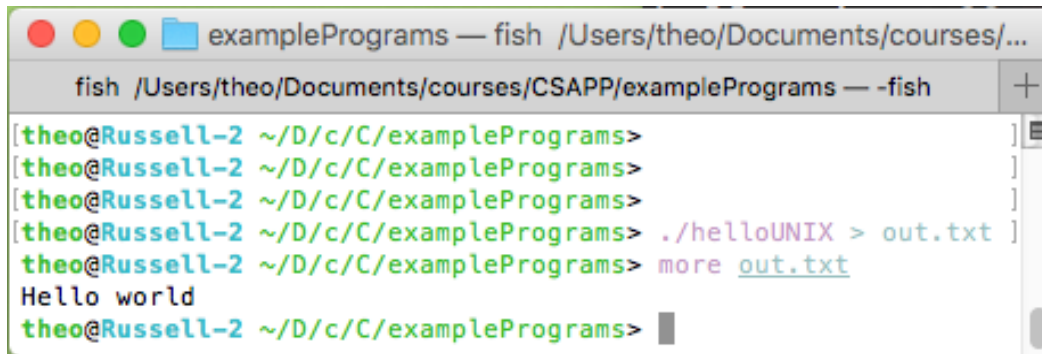
If you start a UNIX program from shell in a terminal window, then 0, 1, and 2 represent that window. Running the helloUNIX program (see below) on a OS X terminal window, we get:

A screenshot of a terminal window on OS X. The window title is "examplePrograms — fish /Users/theo/Documents/...". The terminal shows a series of shell prompts: [theo@Russell-2 ~/D/c/C/examplePrograms>]. The user enters the command `./helloUNIX`, and the terminal outputs "Hello world". The prompt returns to [theo@Russell-2 ~/D/c/C/examplePrograms>].

```
[theo@Russell-2 ~/D/c/C/examplePrograms>
[theo@Russell-2 ~/D/c/C/examplePrograms>
[theo@Russell-2 ~/D/c/C/examplePrograms>
[theo@Russell-2 ~/D/c/C/examplePrograms>
[theo@Russell-2 ~/D/c/C/examplePrograms> ./helloUNIX
Hello world
[theo@Russell-2 ~/D/c/C/examplePrograms>
```

This works because the program writes to file descriptor 1 and file descriptor 1 represents the Terminal window. There is no need for the program to open the terminal for writing because the UNIX shell arranges that when the helloUNIX program starts, file descriptor 1 (and 0 and 2) represent the terminal.

File descriptors 0, 1, and 2 can be bound to actual disk files, if you like.



```

examplePrograms — fish /Users/theo/Documents/courses/...
fish /Users/theo/Documents/courses/CSAPP/examplePrograms — -fish
[theo@Russell-2 ~/D/c/C/examplePrograms>
[theo@Russell-2 ~/D/c/C/examplePrograms>
[theo@Russell-2 ~/D/c/C/examplePrograms>
[theo@Russell-2 ~/D/c/C/examplePrograms> ./helloUNIX > out.txt
[theo@Russell-2 ~/D/c/C/examplePrograms> more out.txt
Hello world
[theo@Russell-2 ~/D/c/C/examplePrograms> █

```

This time the shell created a file called “out.txt” and bound it to file descriptor 1 before starting the program. UNIX pipes work a similar way. The UNIX shell command

```
fred | ginger
```

starts processes running programs fred and ginger at the same time. Before starting them:

- It creates a ‘pipe’, which is a sort of a queue of bytes.
- It arranges that in the fred process, 1 represents the pipe.
- It arranges that in the ginger process, 0 represents the pipe.

Thus the standard output of the first process is the standard input of the second.

Object oriented I/O

So file descriptors can represent

- Disk files.
- Devices such as printers, keyboards, and entire disks.
- Virtual devices such as terminal windows, and pipes.

A program (or subroutine) can work with any of these objects without the programmer having to worry about what sort of object it is. The programmer just uses read and write and assumes the object will interpret the call reasonably.

See the RIO functions in the text book, for example. They will work with any appropriate object. They are “polymorphic” functions.

Soon we will look at how we can use this idea to perform input and output across the internet.

The C way of I/O vs, the UNIX way of I/O

[Not to be covered in class. Please read anyway. It's good stuff to know]

Standard C functions

You may be wondering about how write relates to C's printf function and to C++'s cout object.

Operating systems and languages both define interfaces, but they are not the same interfaces.

open, read, write, and close are defined in the UNIX standard. They may not be available or have the same meaning in other operating systems (for example Windows).

Of course C and C++ are intended to be portable across many operating systems.

So the C language standard defines its own, but different, library functions, such as fopen, getc, putc, fclose. C also

has functions that build on top of these functions such as `fwrite`, `fread`, `fprintf`, `fscanf`, `printf`, and `scanf`.

If a C program is running on UNIX, all these functions ultimately use UNIX functions like `open`, `read`, `write`, and `close`.

But if a C program is running on Windows, these function use Windows system calls.

Similarly C++ defines portable input/output objects such as `cin`, `cout`, `cerr` on top of the C functions.

Don't mix UNIX and standard C functions on the same input or output stream.

FILE objects and FILE* pointers.

In place of file descriptors, C uses objects of type `FILE`.

For example the C function `fopen` creates a `FILE` object and returns a pointer to that object.

The C function `putc` takes a pointer to a `FILE` object and a character and outputs the character.

Before the `main` function is called, 3 `FILE` objects created and three pointers to them are initialized.

- `stdin`
- `stdout`
- `stderr`

So if a C program runs on UNIX, we have a choice.

We could write

```
1 // Standard C include files
2 #include <stdlib.h>
3 #include <string.h>
4
5 // UNIX include files.
6 #include <unistd.h>
7 #include <errno.h>
8
9 int main( int argc, char **argv ) {
10     char *data = "Hello world\n" ;
11     int count = strlen( data ) ;
12     for( int i = 0 ; i < count ; ) {
13         int writeCount = write( 1, data+i, count-i ) ;
14         if( writeCount < 0 && (errno == EINTR || errno == EAGAIN) ) {
15             writeCount = 0 ; }
16         if( writeCount < 0 ) exit(1) ;
17         i += writeCount ; }
18     return 0 ; }
```

Or we could be more portable and write

```
1 // Standard C include file
2 #include <stdio.h>
3
4 int main( int argc, char **argv ) {
5     fprintf( stdout, "Hello world\n" ) ;
6     return 0 ; }
```

The C `printf(...)` function is the same as `fprintf(stdout, ...)`

For more on the C functions see

- <http://www.cplusplus.com/reference/cstdio/>