

CSCI 4061: Input/Output with Files, Pipes

Chris Kauffman

*Last Updated:
Tue Feb 12 09:44:23 CST 2019*

Logistics

Reading

- ▶ Stevens/Rago
Ch 3, 4, 5, 6
- ▶ OR Robbins and Robbins
Ch 4, 5

Goals

- ▶ Assignment 1 Questions
- ▶ Standard IO library
- ▶ `open()/close()`
- ▶ `read()/write()`

Assignment 1

- ▶ Questions?

Lab03:

- ▶ `wait() + NOHANG`
- ▶ `pipe()`
- ▶ All things you'll need in first project
- ▶ How did it go?

Exercise: C Standard I/O Functions

Recall basic I/O functions from the C Standard Library header `stdio.h`

- ▶ Printing things to the screen?
- ▶ Opening a file?
- ▶ Closing a file?
- ▶ Printing to a file?
- ▶ Scanning from terminal or file?
- ▶ Get whole lines of text?
- ▶ Names for standard input, output, error

Give samples of function calls

Answers: C Standard I/O Functions

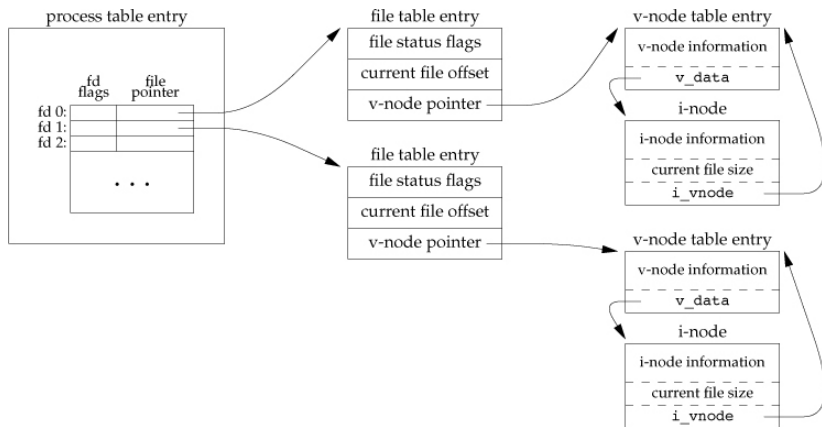
Recall basic I/O functions from the C Standard Library header `stdio.h`

<code>printf("%d is a number",5);</code>	Printing things to the screen?
<code>FILE *file = fopen("myfile.txt","r");</code>	Opening a file?
<code>fclose(file);</code>	Close a file?
<code>fprintf(file,"%d is a number",5);</code>	Printing to a file?
<code>fscanf(file2,"%d %f",&myint,&mydouble);</code>	Scanning from terminal or file?
<code>result = fgets(charbuf, 1024, file);</code>	Get whole lines of text?
<code>FILE *stdin, *stdout, *stderr;</code>	Names for standard input, etc

*The standard I/O library was written by Dennis Ritchie around 1975.
–Stevens and Rago*

- ▶ Assuming you are familiar with these and could look up others like `fgetc()` (single char) and `fread()` (read binary)
- ▶ Standard C: available wherever there is compiler
- ▶ On Unix systems, `fscanf()`, `FILE*`, the like are backed by underlying system calls and concepts

File Descriptors



- ▶ OS maintains data on all processes in Process Table
- ▶ Data includes file descriptors, refer to other OS tables
- ▶ Program deals with `int fd; : index into table`

File Descriptors are Multi-Purpose

- ▶ Unix tries to provide most things via files/file descriptor
- ▶ Many interactions created via `read()/write()` from/to file descriptors
- ▶ Get file descriptors from standard files like `myfile.txt` or `commando.c` to read/change them
- ▶ Also get file descriptors for many other things
 - ▶ Pipes for interprocess communication
 - ▶ Sockets for network communication
 - ▶ Special files to manipulate terminal, audio, graphics, terminal
- ▶ Even processes themselves have special files in the file system: **ProcFS** in `/proc/PID#`, provide info on running process

Open and Close: File Descriptors for Files

```
#include <sys/stat.h>
#include <fcntl.h>

int fd1 = open("firstfile", O_RDONLY); // read only
if(fd1 == -1){                          // check for errors on open
    perror("Failed to open 'firstfile'");
}

int fd2 = open("secndfile", O_WRONLY); // write only, better be present
int fd3 = open("thirdfile", O_WRONLY | O_CREAT); // write only, create if needed
int fd4 = open("forthfile", O_WRONLY | O_CREAT | O_APPEND); // append if existing

// 5 options for first arg: open for what ...
// Around 13 options for 2nd argument to open...

...;                                // Do stuff with open files

int result = close(fd1); // close the file associated with fd1
if(result == -1){        // check for an error
    perror("Couldn't close 'firstfile'");
}
```

- ▶ Note use of vertical pipe (|) to bitwise-OR several options
- ▶ Common for system calls

read() from File Descriptors

```
#define SIZE 128

int in_fd = open(in_name, O_RDONLY);
char buffer[SIZE];
int bytes_read = read(in_fd, buffer, SIZE);
```

- ▶ Read up to SIZE from an open file descriptor
- ▶ Bytes stored in buffer, overwrite it
- ▶ Return value is number of bytes read, -1 for error
- ▶ SIZE commonly defined but can be variable, constant, etc
- ▶ Examine read_some.c : explain what's happening

Warnings

- ▶ Bad things happen if buffer is actually smaller than SIZE
- ▶ NOT null terminated: must add a \0 if this is desired

Exercise: `write()` to File Descriptors

```
#define SIZE 128

int out_fd = open(out_name, O_WRONLY);
char buffer[SIZE];
int bytes_written = write(out_fd, buffer, SIZE);
```

- ▶ Write up to SIZE bytes to open file descriptor
- ▶ Bytes taken from buffer, leave it intact
- ▶ Return value is number of bytes written, -1 for error

Questions

- ▶ Examine `write_then_read.c` for additional details
- ▶ Make sure `existing.txt` is present, empty
- ▶ Compile and run
- ▶ Use `cat existing.txt`: **explain contents**

read()/write() work with bytes

- ▶ In C, general correspondence between byte and the char type
- ▶ Not so for other types: int is often 4 bytes
- ▶ Requires care with non-char types
- ▶ All calls read/write actual bytes

```
#define COUNT 16
int out_ints[COUNT];           // array of 16 integers
int bufsize = sizeof(int)*COUNT; // size in bytes of array
...;
write(out_fd, out_ints, bufsize); // write whole buffer

int in_ints[COUNT];
...;
read(in_fd, in_ints, bufsize);    // read to capacity of in_ints
```

Questions

- ▶ Examine write_read_ints.c, compile/run
- ▶ Examine contents of integers.dat
- ▶ Explain what you see

Standard File Descriptors

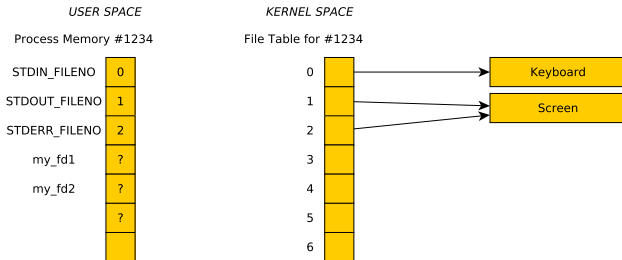
- ▶ When a process is born, comes with 3 open file descriptors
- ▶ Related to FILE* streams in Standard C I/O library
- ▶ Traditionally have FD values given but use the Symbolic name to be safe

Symbol	#	FILE*	FD for...
STDIN_FILENO	0	stdin	standard input (keyboard)
STDOUT_FILENO	1	stdout	standard output (screen)
STDERR_FILENO	2	stderr	standard error (screen)

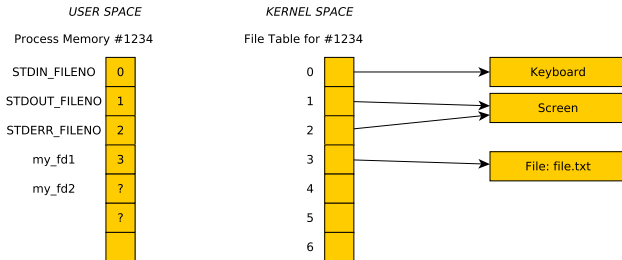
```
// Low level printing to the screen
char message[] = "Wubba lubba dub dub!\n";
int length = strlen(message);
write(STDOUT_FILENO, message, length);
```

See `low_level_interactions.c` to gain an appreciation for what `printf()` and its kin can do for you.

File Descriptors refer to Kernel Structures



```
my_fd1 = open("file.txt", O_RDONLY);
```



Shell I/O Redirection

- ▶ Shells can direct input / output for programs using `<` and `>`
- ▶ Most common conventions are as follows

```
$> some_program > output.txt  
# output redirection to output.txt
```

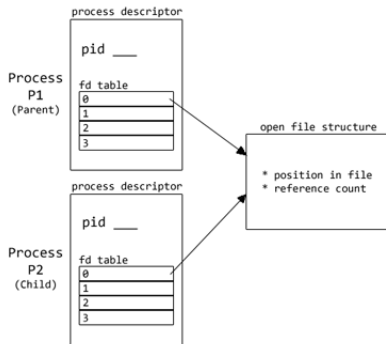
```
$> interactive_prog < input.txt  
# read from input.txt rather than typing
```

```
$> some_program >& everthing.txt  
# both stdout and stderr to file
```

```
$> some_program 2> /dev/null  
# stderr silenced, stdout normal
```

- ▶ Long output can be saved easily
- ▶ Can save typing input over and over
- ▶ Gets even better with pipes (soon)

Processes Inherit Open FDs

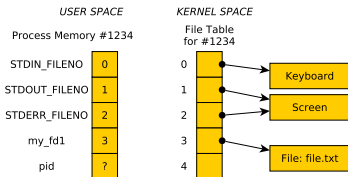


Source: Eddie Kohler Lecture Notes

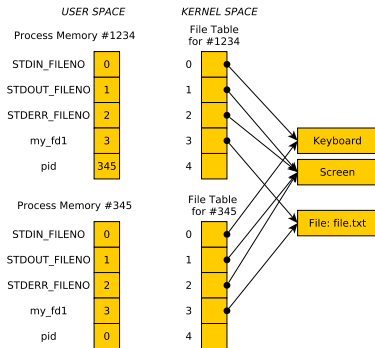
- ▶ Shells start child processes with `fork()`
- ▶ Child processes share all open file descriptors with parents
- ▶ Child prints to screen by default, reads from keyboard
- ▶ Redirection requires manipulation prior to `fork()`

Processes Inherit Open FDs: Diagram

BEFORE: pid = fork();



AFTER: pid = fork();



Typical sequence:

- ▶ Parent creates an output_fd and/or input_fd
- ▶ Call fork()
- ▶ Child changes standard output to output_fd and/or input_fd
- ▶ Changing means calls to dup2()

Redirecting Output with `dup()` / `dup2()`

- ▶ System calls `dup()` and `dup2()` allow for manipulation of the file descriptor table.
- ▶ `int backup_fd = dup(fd);` creates a copy of the file descriptor
- ▶ `dup2(from_fd, to_fd);` causes `to_fd` to refer to the same spot as `from_fd`

Diagrams

- ▶ [fork-dup.pdf](#) diagram to shows how to redirect standard out to a file like a shell `ls -l > output.txt`
- ▶ [pipe-dup.pdf](#) diagram to shows how to redirect standard output to a pipe so `printf()` would go into the pipe for later reading

Pipes

- ▶ A vehicle for one process to communicate with another
- ▶ Uses internal OS memory rather than temporary files
- ▶ A great Unix innovation which allows small programs to be strung together to produce big functionality
- ▶ Leads to smaller programs that cooperate
- ▶ Preceding OS's lacked communication between programs meaning programs grew to unmanageable size

Pipes on the Command Line

Super slick for those that know what they are doing: string programs with |

```
> ls | grep pdf
00-course-mechanics.pdf
01-introduction.pdf
02-unix-basics.pdf
03-process-basics.pdf
04-making-processes.pdf
05-io-files-pipes.pdf
99-p1-commando.pdf
header.pdf
> ls | grep pdf | sed 's/pdf/PDF/'
00-course-mechanics.PDF
01-introduction.PDF
02-unix-basics.PDF
03-process-basics.PDF
04-making-processes.PDF
05-io-files-pipes.PDF
99-p1-commando.PDF
header.PDF
```

```
cat file.txt |           # Feed input \
tr -sc 'A-Za-z' '\n' |  # Translate non-alpha to newline \
tr 'A-Z' 'a-z' |        # Upper to lower case \
sort |                   # Duh \
uniq -c |                # Merge repeated, add counts \
sort -rn |               # Sort in reverse numerical order \
head -n 10               # Print only top 10 lines
```

Pipe C function Calls

- ▶ Use the `pipe()` system call
- ▶ Argument is an array of 2 integers
- ▶ Filled by OS with file descriptors of opened pipe
- ▶ 0th entry is for reading
- ▶ 1th entry is for writing

```
int my_pipe[2];           // array of 2 file descriptors
int result = pipe(my_pipe); // now filled with 2 fds by system

char msg[128] = "hello world";
int nwritten = write(my_pipe[1], msg, strlen(msg)+1);

char buffer[128];
int nread = read(my_pipe[0], buffer, 128);

close(my_pipe[0]);
close(my_pipe[1]);
```