CSCI 4061: Input/Output with Files, Pipes

Chris Kauffman

Last Updated:
Mon Feb 15 05:02:57 PM CST 2021

Logistics

Reading

Stevens/Rago Ch 3, 4, 5, 6

Goals

- Standard IO library
- File Descriptors
- `open()/close()`
- `read()/write()`
- `dup2() / dup()`
- Pipes

Assignments

- Lab03: Expand buffer with `realloc()`
- HW03: Ditto + `read()` + child control
- All have important techniques necessary for P1
- P1 Commando Ongoing: finish techniques required by Mon 2/15 lecture

Exercise: C Standard I/O Functions

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

1. Printing things to the screen?
2. Opening a file?
3. Closing a file?
4. Printing to a file?
5. Scanning from terminal or file?
6. Get whole lines of text?
7. Names for standard input, output, error

Give samples of function calls

Write your answers in a text file so a team member can share screens

Answers: C Standard I/O Functions

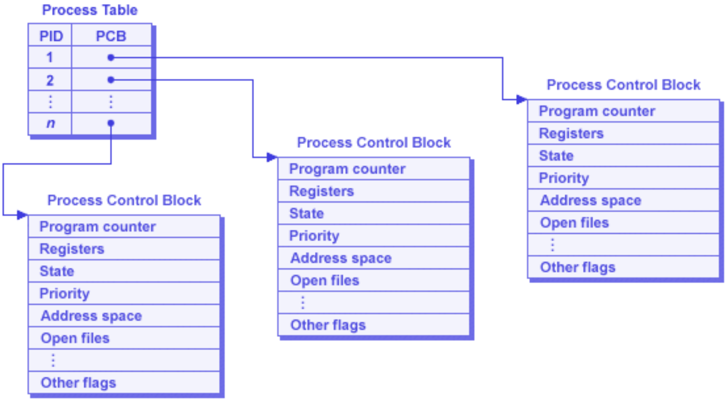
Recall basic I/O functions from the C Standard Library header `stdio.h`

1	`printf("%d is a number",5);`	Printing things to the screen?
2	`FILE *file = fopen("myfile.txt","r");`	Opening a file?
3	`fclose(file);`	Close a file?
4	`fprintf(file,"%d is a number",5);`	Printing to a file?
5	`scanf("%d %f",&myint,&mydouble);`	Scanning from terminal
	`fscanf(file2,"%d %f",&myint,&mydouble);`	or file?
6	`result = fgets(charbuf, 1024, file);`	Get whole lines of text?
7	`FILE *stdin, *stdout, *stderr;`	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)
- Library Functions: available with any compliant C compiler
- On Unix systems, `fscanf()`, `FILE*`, and the like are backed by lower level System Calls and Kernel Data Structures

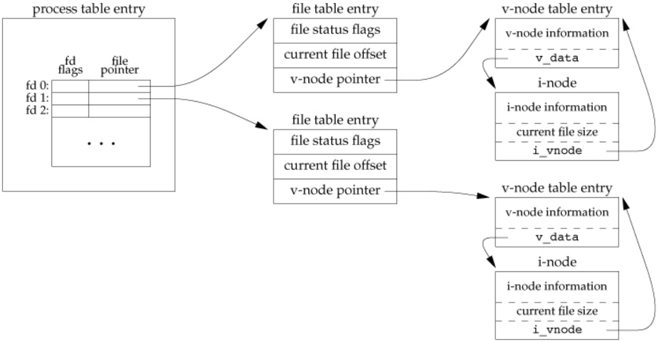
The Process Table



Source: SO What is the Linux Process Table?

- OS maintains data on all processes in a Process Table
- Process Table Entry ≈ Process Control Block
- Contains info like PID, instruction that process is executing*, Virtual Memory Address Space and **Files in Use**

File Descriptors



- Each Process Table entry contains a table of open files
- A use program refers to these via **File Descriptors**
- File Descriptor is an integer index into Kernel's table

```
int fd = open("some_file.txt", O_RDONLY);
```

- FD Table entry refers to other Kernel/OS data structures

File Descriptors are Multi-Purpose

- Unix tries to provide most things via files/file descriptor
- Many Unix system actions are handled via `read()`-from or `write()`-to file descriptors
- File descriptors allow interaction with standard like `myfile.txt` or `commando.c` to read/change them
- FD's also allow interaction with many other things
  - Pipes for interprocess communication
  - Sockets for network communication
  - Special files to manipulate terminal, audio, graphics, etc.
  - Raw blocks of memory for Shared Memory communication
  - Even processes themselves have special files in the file system: ProcFS in `/proc/PID#`, provide info on running process
- We will focus on standard File I/O using FDs Now and touch on some broader uses Later
- Also must discuss interactions between previous and new System Calls like
  What happens with `open()` files when calling `fork()`?

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, fails if not found
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

// 'man 3 open' will list all the O_xxx options when opening.
//  Other common options: O_RDONLY, O_RDWR, O_EXEC

...;                            // 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'");
}
```

`open()` / `close()` show common features of many system calls

- Returns -1 on errors
- Show errors using the `perror()` function
- Use of vertical pipe (|) to bitwise-OR several options

read() from File Descriptors

```c
// read_some.c: Basic demonstration of reading data from
// a file using open(), read(), close() system calls.

#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`
- `read()` does NOT null terminate, add \0 manually if needed

Exercise: Behavior of `read()`

```c
// count_bytes.c
#define BUFSIZE 4

int main(int argc, char *argv[]){
  char *infile = argv[1];
  int in_fd = open(infile,O_RDONLY);
  char buf[BUFSIZE];
  int nread, total=0;
  while(1){
    nread = read(in_fd,buf,BUFSIZE-1);
    if(nread == 0){
      break;
    }
    buf[nread] = '\0';
    total += nread;
    printf("read: '%s'\n",buf);
  }
  printf("%d bytes total\n",total);
  close(in_fd);
  return 0;
}
```

Run `count_bytes.c` on file `data.txt`

```
> cat data.txt
ABCDEFGHIJ
> gcc count_bytes.c
> ./a.out data.txt
???
```

1. Explain control flow within program
2. Predict output of program

Answers: Behavior of read()

```
==INITIAL STATE==
data.txt: ABCDEFGHIJ\n
position: ^
buf:     |? ? ? ? |
          0 1 2 3
nread:   0
total:   0
```

```
==ITERATION 1==
nread = read(in_fd,buf,3);
buf[nread] = '\0'
total+= nread;
printf("read: '%s'\n",buf);

data.txt: ABCDEFGHIJ\n
position:    ^
buf:     |A B C \0|
          0 1 2 3
nread:    3
total:    3
output:  'ABC'
```

```
==ITERATION 2==
nread = read(in_fd,buf,3);
buf[nread] = '\0'
total+= nread;
printf("read: '%s'\n",buf);

data.txt: ABCDEFGHIJ\n
position:       ^
buf:     |D E F \0|
          0 1 2 3
nread:    3
total:    6
output:  'DEF'
```

```
==ITERATION 3==
nread = read(in_fd,buf,3);
buf[nread] = '\0'
total+= nread;
printf("read: '%s'\n",buf);

data.txt: ABCDEFGHIJ\n
position:          ^
buf:     |G H I \0|
          0 1 2 3
nread:    3
total:    9
output:  'GHI'
```

```
==ITERATION 4==
nread = read(in_fd,buf,3);
buf[nread] = '\0'
total+= nread;
printf("read: '%s'\n",buf);

data.txt: ABCDEFGHIJ\n
position:             ^
buf:     |J \n\0\0|
          0 1 2 3
nread:    2
total:    11
output:   'J\n'
```

```
==ITERATION 5==
nread = read(in_fd,buf,3);
if(nread == 0){
  break;
}

data.txt: ABCDEFGHIJ\n
position:             ^
buf:     |J \n\0\0|
          0 1 2 3
nread:    0
total:    11
output:   11 bytes total
```

Answers: Behavior of `read()`

Take-Aways from `count_bytes.c` include

- OS maintains **file positions** for each open File Descriptor
- I/O functions like `read()` use/change position **in a file**
- `read()`'ing into program arrays overwrites data there
- OS **does not** update positions in user arrays: programmer must do this in their program logic
- `read()` returns # of bytes read, may be less than requested
- `read()` returns 0 when at end of a file

Exercise: write() to File Descriptors

```c
#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 on write_then_read.c

- Download, Compile, Run:
  https://z.umn.edu/write_then_read
- Explain Output, differences between write() / printf()

Answers: `write()` to File Descriptors

```
> gcc write_then_read.c
> ./a.out
0. Recreating empty existing.txt
1. Opening file existing.txt for writing
2. Writing to file existing.txt
3. Wrote 128 bytes to existing.txt
4. Opening existing.txt for reading
5. Reading up to 128 bytes from existing.txt
6. Read 127 chars, printf()'ing:
here is some text to write
7. printf()'ing 127 characters individually
here is some text to write\0\0\0hello\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
8. write()'ing 127 characters to screen
here is some text to write^@^@^@hello^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
```

`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

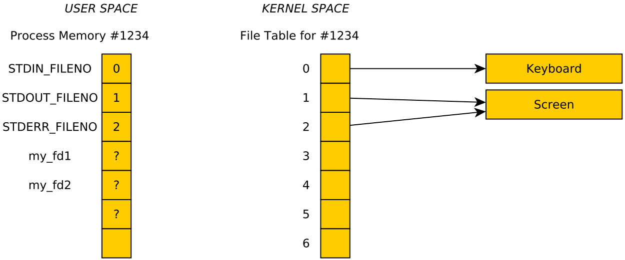
- When a process is born, comes with 3 open file descriptors
- Related to `FILE*` streams in Standard C I/O library
- Traditionally have FD values given but use the Symbolic name to be safe

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)

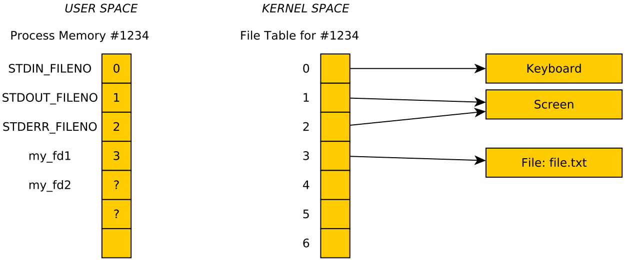
```
// Low level printing to the screen
char message[] = "Wubba lubba dub dub!\n";
int length = strlen(message);
write(STDOUT_FILENO, message, length);
```

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

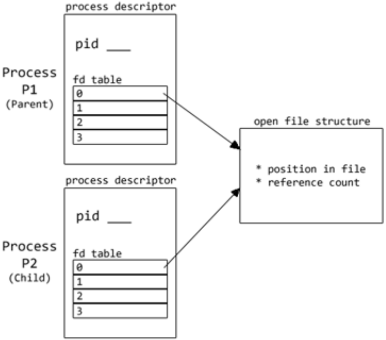
- Shells can direct input / output for programs using < and >
- Most common conventions are as follows

```
$> 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

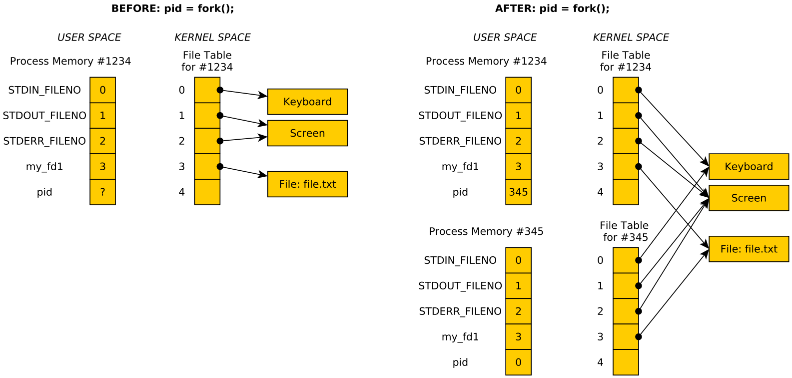
- Shells start child processes with `fork()`
- Child processes share all open file descriptors with parents
- By default, Child prints to screen / reads from keyboard input
- Redirection requires manipulation prior to `fork()`
- See: `open_fork.c`
- Experiment with order
  1. `open()` then `fork()`
  2. `fork()` then `open()`



Source: Eddie Kohler Lecture Notes

Examine: fork-open-file.pdf for picture explaining effects of `open()` vs `fork()` order differences

Processes Inherit Open FDs: Diagram

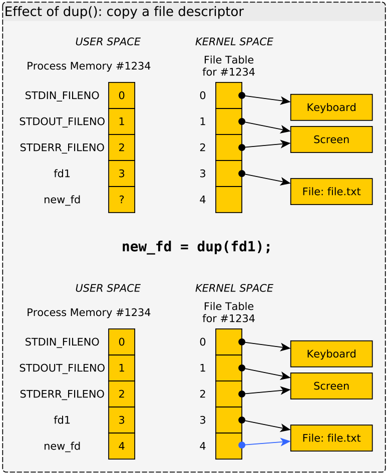


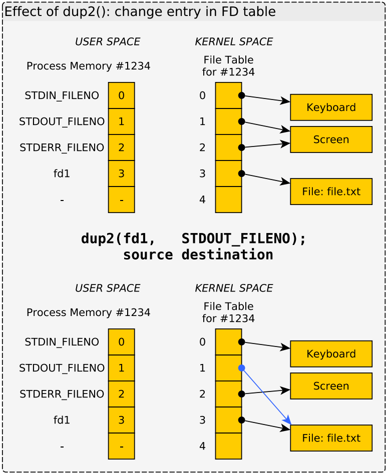
Typical sequence:

- 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()`

Manipulating the File Descriptor Table

- System calls `dup()` and `dup2()` manipulate the FD table
- `int backup_fd = dup(fd);` : copy a file descriptor
- `dup2(src_fd, dest_fd);` : `src_fd` copied to `dest_fd`





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

- `dup()`, `dup2()`, and `fork()` can be combined in interesting ways
- **Diagram** `fork-dup.pdf` shows how to redirect standard out to a file like a shell does in: `ls -l > output.txt`

Write a program which

1. Prints PID to screen
2. Opens a file named `write.txt`
3. Forks a Child process
4. Child: **redirect standard output** into `write.txt`

   Parent: does no redirection
5. Both: `printf()` their PID
6. Child: **restore** standard output to screen

   Parent: makes no changes
7. Both: `printf()` "All done"

```
> gcc duped_child.c

> ./a.out
BEGIN: Process 1913588
MIDDLE: Process 1913588
END: Process 1913588 All done
END: Process 1913590 All done

> cat write.txt
MIDDLE: Process 1913590
```

Answers: Redirecting Output with dup() / dup2()

```c
// duped_chld.c: solution to in-class activity on redirecting output
// in child process.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char *argv[]){
  system("echo '' > write.txt");       // ensure file exists, is empty
  printf("BEGIN: Process %d\n",getpid());
  int fd = open("write.txt",O_WRONLY); // open a file
  int backup;
  pid_t child = fork();                // fork a child, inherits open file
  if(child == 0){                      // child only redirects stdout
    backup = dup(STDOUT_FILENO);       // make backup of stdout
    dup2(fd,STDOUT_FILENO);            // dup2() alters stdout so child printf() goes into file
  }
  printf("MIDDLE: Process %d\n",getpid());
  if(child == 0){
    dup2(backup,STDOUT_FILENO);        // restore stdout
  }
  printf("END: Process %d All done\n",getpid());
  close(fd);
  return 0;
}
```

Pipes

- A mechanism 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 familiar with many Unix utilities: string together programs with |, output from first becomes input for second

```
> 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 System 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]);
```

pipe-dup.pdf diagram to shows how to redirect standard output to a pipe so printf() would go into the pipe for later reading

C Standard I/O Implementation

Typical Unix implementation of standard I/O library `FILE` is

- A file descriptor
- Some buffers with positions
- Some options controlling buffering

~~From `/usr/lib/libio.h`~~

From `/usr/include/bits/types/struct_FILE.h`

```
struct _IO_FILE {
  int _flags;                 // options
  char* _IO_read_ptr;         // buffers for read/write and
  char* _IO_read_end;         // positions within them
  char* _IO_read_base;
  char* _IO_write_base;
  ...;
  int _fileno;                // unix file descriptor
  ...;
  _IO_lock_t *_lock;          // locking
};
```

Exercise: Subtleties of Mixing Standard and Low-Level I/O

```
3K.txt:
 1 2 3 4 5 6 7 8 9 10 11 12 13 14...
37 38 39 40 41 42 43 44 45 46 47 ...
70 71 72 73 74 75 76 77 78 79 80 ...
102 103 104 105 106 107 108 109 1...
...
```

```c
// mixed_std_low.c: mix C Standard
// and Unix I/O calls. pain++;
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]){
  FILE *input = fopen("3K.txt","r");
  int first;
  fscanf(input, "%d", &first);
  printf("FIRST: %d\n",first);

  int fd = fileno(input);
  char buf[64];
  read(fd, buf, 63);
  buf[63] = '\0';
  printf("NEXT: %s\n",buf);

  return 0;
}
```

Sample compile/run:

```
> gcc mixed_std_low.c
> ./a.out
FIRST: 1
NEXT: 41 1042 1043 1044 1045...
```

- Explain output of program given input file
- Use knowledge that **buffering** occurs internally for standard I/O library

Answers: Subtleties of Mixing Standard and Low-Level I/O

- C standard I/O calls like `printf` / `fprintf()` and `scanf()` / `fscanf()` use internal buffering
- A call to `fscanf(file, "%d", &x)` will read a large chunk from a file but only process part of it
- From OS perspective, associated file descriptor has advanced forwards / read a bunch
- The data is in a hidden "buffer" associated with a `FILE *file`, used by `fscanf()`

Output Also buffered, Always `fclose()`

- Output is also buffered: `output_buffering.c`
- Output may be lost if `FILE*` are not `fclose()`'d: closing will flush remaining output into a file
- See `fail_to_write.c`
- File descriptors always get flushed out by OS when a program ends BUT `FILE*` requires user action

Controlling FILE Buffering

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Above functions change buffering behavior of standard C I/O
Examples:

```
// 1. Set full "block" buffering for stdout, use outbuf
#define BUFSIZE 64
char outbuf[BUFSIZE] = {};
setvbuf(stdout, outbuf, _IOFBF, BUFSIZE);

// 2. Turn off buffering of stdout, output immediately printed
setvbuf(stdout, NULL, _IONBF, 0);
```

ALL of you will write the 2nd example in a program soon. What program?