CSSE2310 — 6b.1

File descriptors and pipes

“Everything is a file”?

- Unix systems make many aspects of the system available by file interface.
  - eg `cat /proc/cpuinfo`
- IO devices
  - eg Disks `/dev/sda1`

Just because you can see it on the filesystem, doesn’t mean it is actually stored there (or anywhere).

- eg: `/dev/random`

Unix files

To be treated as a file, the kernel needs to know what to do with a few basic requests:

- open, close
- read, write — bytes only
- seek — move around in the file

Note that some calls might return errors.
Eg moving backwards in `stdin` from keyboard won't work.

File descriptors

- Unix systems (at a lower level than the standard IO functions) use integers to refer to open files.
- When you ask the kernel to open a file, it gives back a "file-descriptor".
- When ever the program wants to interact with that file, it includes that number in requests to the kernel.

open()

See fd.c.

- open() takes numeric constants instead of the string form used by fopen()
  - O_RDONLY, O_WRONLY, O_RDWR ← be careful
  - Can | in other flags:
    - O_APPEND
    - O_CREAT
- read() call deals in fixed numbers of bytes.

See fd2.c

Writing

See `write.c`.

- We need to specify what “mode” (ie permissions) the newly created file.
- If we want the file created, we need to ask for that.
- If we want the file truncated on creation, we need to ask for that.

fds and `fork()`

See `fdc.c`

- Any open files in the parent at time of `fork()` will also be open in the child.
- The parent and child share the same offset.
- If opening the same file multiple times (ie different file descriptors) moving one does not move the other.

IO redirection

Let's send output to a file.

See `redir.c`

- `stdin` → descriptor 0
- `stdout` → descriptor 1
- `stderr` → descriptor 2

Note that `dup2()` is "copy" not "move", so it is a good idea to close the original.
There is also a `dup()` which copies a descriptor but doesn't put it in a specified place.

Piping

```
ls | sort -r
```

The (standard) output of `ls` is connected "somehow" to the (standard) in of the second process.

- `write(1,...` → `read(0...`

What about:

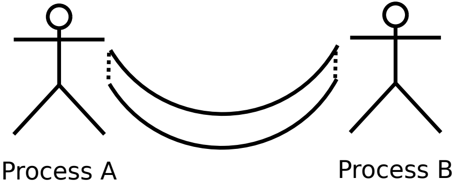
```
(ls > tempfile &) ; (sort -r < tempfile)
```

Problems:

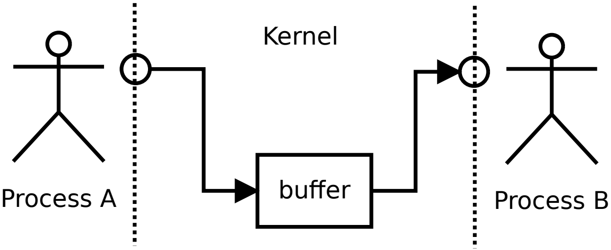
- Assumes the process has write permission to the current directory.
- The second process could hit EOF before the first process has finished writing.

We want something which doesn't use "disk" files.

Pipes in C — behaviour



Pipes in C — behaviour



Pipes in C

Reading from a pipe:

- If no bytes in the pipe → block
- EOF only when it is impossible to write to the pipe.
  - In the simple case when the writing process closes its fd. But beware later.

Writing to a pipe

- If the pipe is full → block
- If it is impossible to read from pipe (and you are attempting to write) → you win a SIGPIPE from the kernel.

Physical intuition

Note:

- Bytes that are written go into a “sump” (buffer)[1], they do not travel directly to the other process.
- The other process must actively read (there is no push delivery).
- There is no concept of pressure (you can’t write bytes harder to push them into the other process).
- You can’t connect two pipes together (end-to-end) and read from the far end to suck bytes out of the earlier pipe.

[1] Don’t ever call it a sump in programming

Pipes in C

You can't[2] pass a file descriptor to an arbitrary process.

- It's just a number, the kernel may have something completely different associated with that slot.

You can't force an fd onto another process:

- You can't force a pipe through the wall of someone's house
- If the program is not expecting to interact with it, it won't ever refer to it.

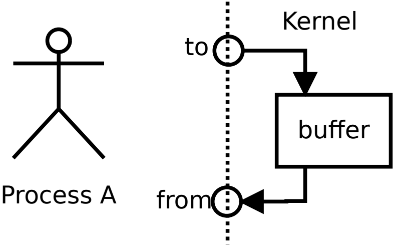
[2] As far as you know from the material in this course

Pipes How?

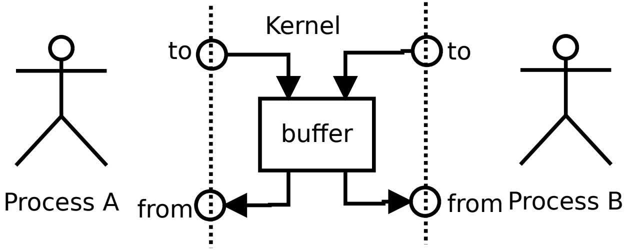
```
int pipe(int fd[2]);
```

- Pass in an array to hold two `fds` (ints).
- A successful `pipe` call fills in this array.
- fd[0] is the read end and fd[1] is the write end.
- Congratulations, you are now holding both ends.

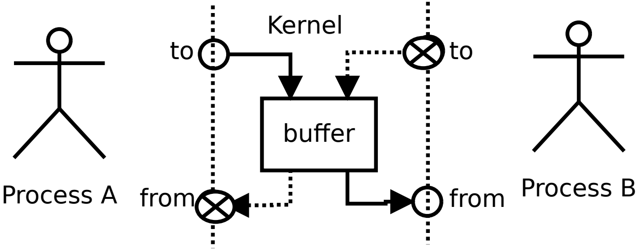
Demo time



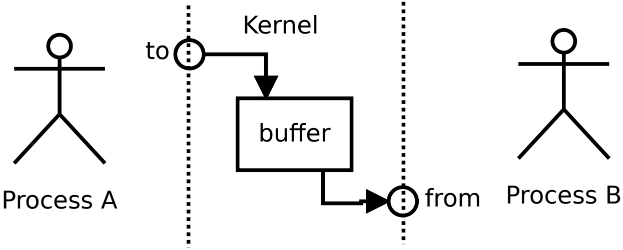
Demo time



Demo time



Demo time



Notes

- You need to create the pipe in the parent before you fork (otherwise the child won't inherit the fds.
- You need to close off the ends of the pipe each process won't be using.
  - Otherwise the reader won't know when the writer has finished.
- If you `dup2()` any `fds` make sure to close the original.
- SIGPIPE could kill your program?
  - Look at the return value of write/printf?

FILE*?

How do fds and FILE* interact?

- If you are setting up fds and then exec-ing on top, there is no issue (since the new process will create its own FILE* stdin, stdout, stderr).
- FILE* fdopen(int fd, const char *mode) will wrap a FILE* around an fd.
- int fileno(FILE *stream) will get the file descriptor.
- If you call fclose(), then the underlying descriptor will be close()d.

Don't do this:

```c
int get_int(int fd) {
    FILE* f = fdopen(fd, "r");
    int result;
    fscanf(f, "%d", &result);
    return result
}
```

Once you wrap an fd, you should forget about it and use the FILE* instead.

In the above code, every time you call `get_int`, a new buffer will be created. This probably results in input going “missing”.