Pipes in C

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

March 13, 2023

Lesson Plan

1. Review Week 7: Low-Level I/O & Signals
2. Review Week 8: Processes
3. Pipes in C & IPC
4. Practice by Mimicry

Acknowledgements

Parts of the slides are borrowed from Andi Bergen.

Section 1

Review Weeks 7+8: Low-Level I/O, Signals, Processes

Review Weeks 7+8: Low-Level I/O, Signals, Processes

Week 7: Low-Level I/O & Signals

1. Bitwise algebra and manipulation
2. Deep dive into low-level file IO
3. Error handling
4. Signals (not too important for today)

See: `www.cs.toronto.edu/~rupert/209/lec07.pdf`

Week 8: Processes

1. Introduction to Processes
   - `fork`
2. Termination & Status
3. Testing Your Understanding
4. Loading into a Process
   - `exec`
5. Putting it Together

See: `www.cs.toronto.edu/~rupert/209/lec08.pdf`

Section 2

Pipes in C

Pipes in C

In C programming, pipes are a one-way and first-in first-out (FIFO) mechanism for inter-process communication (IPC). Data is passed between two processes by connecting the output of one to the input of process using a pipe.

- Pipes live in main memory as "virtual files".
- Just like real files, pipes can be used by not only the process that creates them, but also their children.

Are they the same as UNIX pipes?

You might recall that in the UNIX shell, we also used "pipes". Those (using the character |) are simpler, and redirect the output of one command as the input of another. Pipes in C are more complex and powerful than that!

Pipes in C: Anatomy

A pipe `p` has two ends, a read end `p[0]` and a write end `p[1]`.

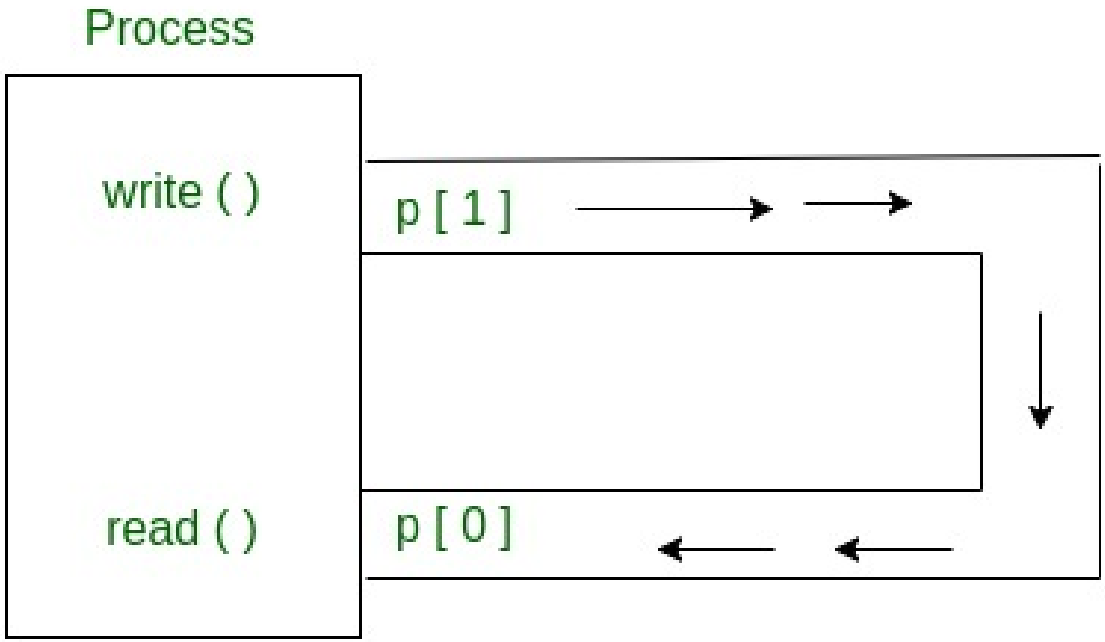

Figure 1: pipes from/to a process

These two ends are typically used on different processes, such as a parent and its child, or related children processes.

Pipes in C: Creation

To create a pipe in C, the `pipe()` system call is used.

```c
#include "unistd.h"
int pipe(int filedes[2]);
```

Its parameters are two integers, each describing the file descriptor (FD) of an end of the pipe; these FDs are how C programmers interact with pipes.

```c
int p[2];   // usually they're just declared on the stack,
pipe(p);    // and the pipe() system call initializes them.
```

When called, `pipe()` finds the first two available positions in the process's open file table and allocates them for the two ends of the pipe. Returns `0` on success.

Error Checking

Creating pipes might fail so checking is a good idea, even if you don't want to crash.

```c
if (pipe(p) < 0) {
    perror("failed to open pipe");
    exit(1); // you could do other handling instead...
}
```

Pipes in C: Reading & Writing

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Once a pipe has been created, it can be used to communicate between processes. To send data through a pipe,

- One process writes data to the write end `write_fd` (or `p[1]`) of the pipe using the `write()` system call.

  ```c
  write(write_fd, data, sizeof(data));
  ```

- The other process can then read the data from the read end `read_fd` (or `p[0]`) of the pipe using the `read()` system call.

  ```c
  read(read_fd, buffer, sizeof(buffer));
  ```

The size given in the the `read()`/`write()` calls must be passed in the function call.

- Can the passed size be smaller than the actual size? What about larger?

Pipes in C: On a Single Process

It's possible to operate on both ends of the same pipe in a single process.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSG_SIZE 13
char *msg = "hello, world\n";
int main(void) {
    char buf[MSG_SIZE];
    int p[2];
    if (pipe(p) == -1) { perror("pipe"); exit(1); }
    write(p[1], msg, MSG_SIZE); // No error checking: Bad
    read(p[0], buf, MSG_SIZE); // No error checking: Bad
    write(STDOUT_FILENO, buf, MSG_SIZE);
    return 0;
}
```

This can be used as a form of memory.

Arnold's Example: `pipes1.c`

`www.cs.toronto.edu/~arnold/209/12s/lectures/pipes/pipes1.c`

Pipes in C: Safe Communications

There are many safety/security considerations when using pipes. The major sources of bugs we've touched on before are:

- Strings manipulation functions.
- Process creation and loading: `fork()`/`exec*()`.
- Low-level file I/O: `write()`, `read()`.

Even if your code runs without errors, there might be unexpected/undesired behaviour and security compromises.

Indicating the wrong number of bytes

You *can* read/write the wrong number of bytes to a pipe. But should you?

```
char buf[MSG_SIZE];
write(STDOUT_FILENO, buf, MSG_SIZE+1); // same as printf()
```

While `clang` wouldn't catch this mistake, `gcc` would! How?

Demo: `rainy.c`

`github.com/rhubarbwu/csc209/blob/master/lectures/lec09/rainy.c`

Pipes in C: Inter-Process Communication (IPC)

Recall that upon calling `fork()` the child process gets a copy of the parent process' file descriptor (FD) table. So if a pipe is created before the fork, the child process after the fact also has access to the pipe.

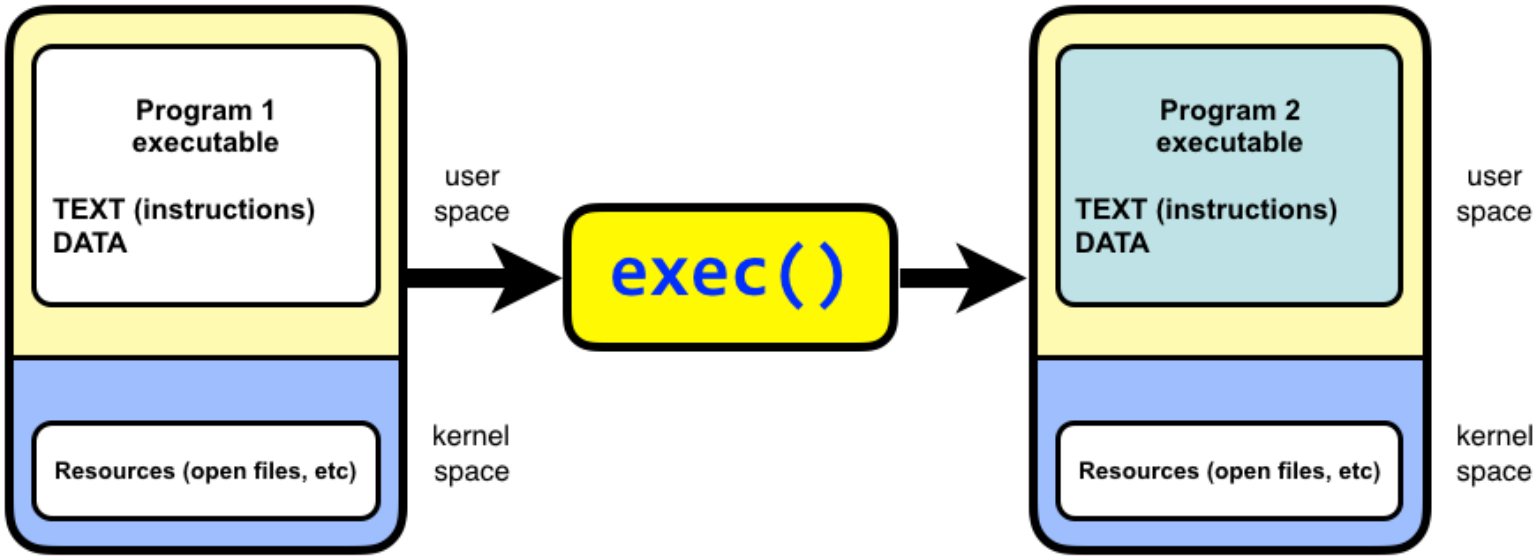

Figure 2: `exec*()`

While `exec*()` doesn't create new processes, the same FD table is retained after the program is replaced (important because `fork()` and `exec*()` are often used together).

Pipes in C: IPC from Child to Parent

Here's a simple program where a child writes to its parent.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSG_SIZE 13
char *msg = "hello, world\n";
int main(void) {
char buf[MSG_SIZE]; int p[2];
    if (pipe(p) == -1) { perror("pipe"); exit(1); }
    if (fork() == 0) // child writes
        write(p[1], msg, MSG_SIZE);
    else { // parent reads from pipe, and prints to stdout
        read(p[0], buf, MSG_SIZE);
        printf("%s\n", buf);
    }
    return 0;
}
```

As an exercise, identify safety checks you might want to perform.

Pipes in C: IPC between Children

You can communicate between children processes too!

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSG_SIZE 20
char *msg = "good morning, bruv\n";
int main(void) {
    char buf[MSG_SIZE]; int p[2];
    if (pipe(p) == -1) { perror("pipe"); exit(1); }
    if (fork() == 0) write(p[1], msg, MSG_SIZE);
    else if (fork() == 0) {
        sleep(1);
        read(p[0], buf, MSG_SIZE);
        printf("%s\n", buf);
    }
    return 0;
}
```

What could go wrong? What's missing from the code?

Pipes in C: Closing

```c
#include <unistd.h>
int close(int fd);
```

Each process should `close()` usused file descriptors, especially with pipes.

- This can save a bit of memory.
- More importantly, `read()` can stall otherwise. If a process calls `read()` on a pipe, but there is no data ready to be read, it will **stall forever**, unless:
  - New data arrives on the pipe; or
  - **All** file descriptors (across all processes) referring to the write end of the pipe have been closed.

This means that not only should processes that never use an FD close them immediately, but all processes should close each existing FD after its last use.

Read more on `man 7 pipe`.

Blocking system calls

`read()` is an example of a blocking system call. What happens to the process when `read()` stalls as described above?

Pipes in C: Closing

Does this program ever terminate?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define MSG_SIZE 21
char *msg = "get me the generals!";
int main(void) {
    char buf[MSG_SIZE];
    int p[2];
    if (pipe(p) != 0) { perror("pipe"); exit(1); }
    if (fork() == 0) {
        read(p[0], buf, MSG_SIZE);
        exit(0);
    }
    int status; wait(&status);
    return 0;
}
```

Which pipes should be closed?

Pipes in C: Closing

Does this program ever terminate? **No it wouldn't. Let's close some FDs...**

```
int main(void) {
    char buf[MSG_SIZE]; int p[2];
    if (pipe(p) != 0) { perror("pipe"); exit(1); }
    if (fork() == 0) {
        close(p[1]); // closing from the child
        read(p[0], buf, MSG_SIZE);
        exit(0);
    }
    close(p[1]); // closing from parent
    int status; wait(&status);
    return 0;
}
```

Which pipes should be closed? **The write end `p[1]` should be closed in both the parent and the child processes. After all (both) ends have been closed, read() will return the number of bytes thus far read.**

Arnold's Example: `pipes2.c`

`http://www.cs.toronto.edu/~arnold/209/12s/lectures/pipes/pipes2.c`

Pipes in C: Some Trivia

Here are some facts from `man 7 pipe`.

Under *I/O on pipes and FIFOs*:

1. Pipes have no message boundaries. If you `write()` to a pipe twice and then `read()` from it, don't assume that you will just receive the first chunk of data that you wrote.
2. `write()` is guaranteed to be *atomic* only if the data being written is smaller than a certain number of bytes (on Linux, 4KB).

Under *Pipe capacity*:

3. Pipes have a finite size: Default is 64KB on Linux.
4. If a `write()` would overflow a pipe, the process blocks (or fails, if `O_NONBLOCK` flag is used when creating the pipe) until another process empties some of the data from the pipe by calling `read()`.

Section 3

Practice by Mimicry

Practice by Mimicry

One way to learn is by imitating or mimicking something/someone else. Consider the following Python code:

```
from sys import argv # this is analogous to the int **argv in C
argslen = sum([len_verbose(arg) for arg in argv])
print(f"The length of all the args is {argslen}")
```

As a quick review, put this code in a file called `argslens` and make it executable with `python`. See if you remember how!

Practice by Mimicry

One way to learn is by imitating or mimicking something/someone else. Consider the following Python code:

```
from sys import argv # this is analogous to the int **argv in C
argslen = sum([len_verbose(arg) for arg in argv])
print(f"The length of all the args is {argslen}")
```

As a quick review, put this code in a file called `argslens` and make it executable with `python`. See if you remember how!

1. Find `python` (`which python`) and prepend a shebang: `#!/usr/bin/python`
2. Make it executable: `chmod a+x argslens`

Now rewrite it in C. Use `fork()` and `pipe()` to mimic this behaviour such that each child process handles one argument.

Karen's Solutions

`github.com/rhubarbwu/csc209/blob/master/lectures/lec09/argslens.c`

Practice by Mimicry

Now, try writing programs using C pipes to mimic UNIX pipes.

```
$ sort -n < infile > outfile
$ sort -n | tail -3
$ sort -n < infile | tail -3
$ sort -n < infile | tail -3 > outfile
$ head -10 < infile | sort -n | tail -3 > outfile
```

Hint: each UNIX pipe | or redirection </> might involve reading/writing of some file descriptor. And each separate command might involve calling `fork()`/`exec*()`.

Remember that UNIX pipes are simpler: they only deal with input and output of whole commands and not their subprocesses. On the other hand, mimicking them with C still requires a fair bit of code...

Arnold's Solutions

`www.cs.toronto.edu/~arnold/209/12s/lectures/pipesAndRedirection`