Systems Software

Week 5: IPC and Pipes

Notes By: Jonathan McCarthy

Overview

- Inter Process Communication (IPC)
- File Descriptors
- Pipes
- Pipes Example
- Named Pipes (FIFO)

IPC

- Inter Process Communication (IPC)
- Facilitating communication between different processes.
- IPC can be offered between:
  - Related processes
  - Unrelated processes

File Streams and File Descriptors

- If we need to perform file operations (IO) to a file, there are 2 options available:
  - Streams
  - File Descriptors
- Streams are represented as File * objects.
- File descriptors are represented as objects of type int.

File Streams

- A stream offers a high level interface that is layered on top of the file descriptors.
- There are more features and functionality that comes with the stream interface for file IO.

File Descriptors

- File descriptors offer a primitive low lever interface for IO operations.
- A file descriptor can connect to a file, a device (terminal), or a pipe or socket to communicate with another process.
- File descriptors should be used for IO with devices, and for nonblocking IO operations.

fopen and fdopen

- The **fopen()** function opens the file whose name is the string pointed to by path and associates a stream with it.
- The **fdopen()** function associates a stream with the existing file descriptor, fd. The mode of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor.

fgetc

- fgetc() reads the next character from stream and returns it as an unsigned char cast to an int, or EOF on end of file or error.

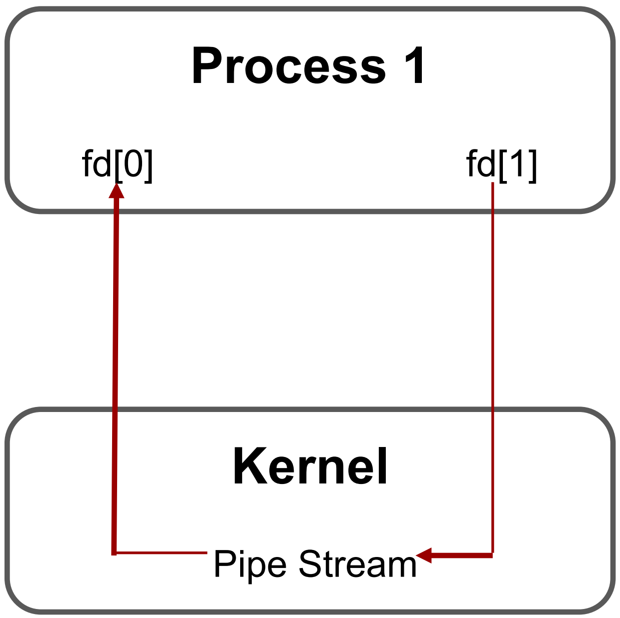
fstat

- stat() retrieve information about the file pointed to by pathname; the differences for fstatat() are described below.
- fstat() is identical to stat(), except that the file about which information is to be retrieved is specified by the file descriptor fd.

What are Pipes

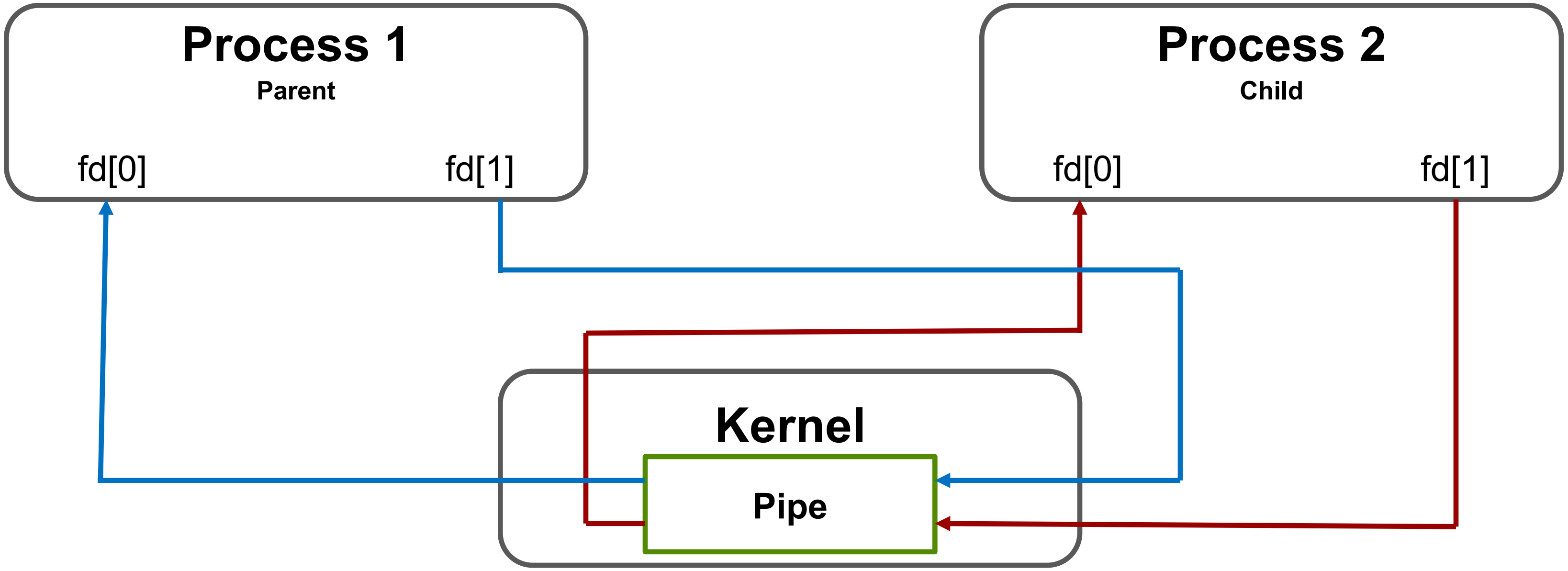
- A pipe allows processes to communicate with each other (Inter Process Communication - IPC)
- A process can write date to a pipe and this can be read by another process.
- The data passes to the pipe follows the FIFO algorithm.
- The pipe has no name.
- The pipe as created by a parent process and uses it to communicate with a child process.

Simple Pipe Architecture Example



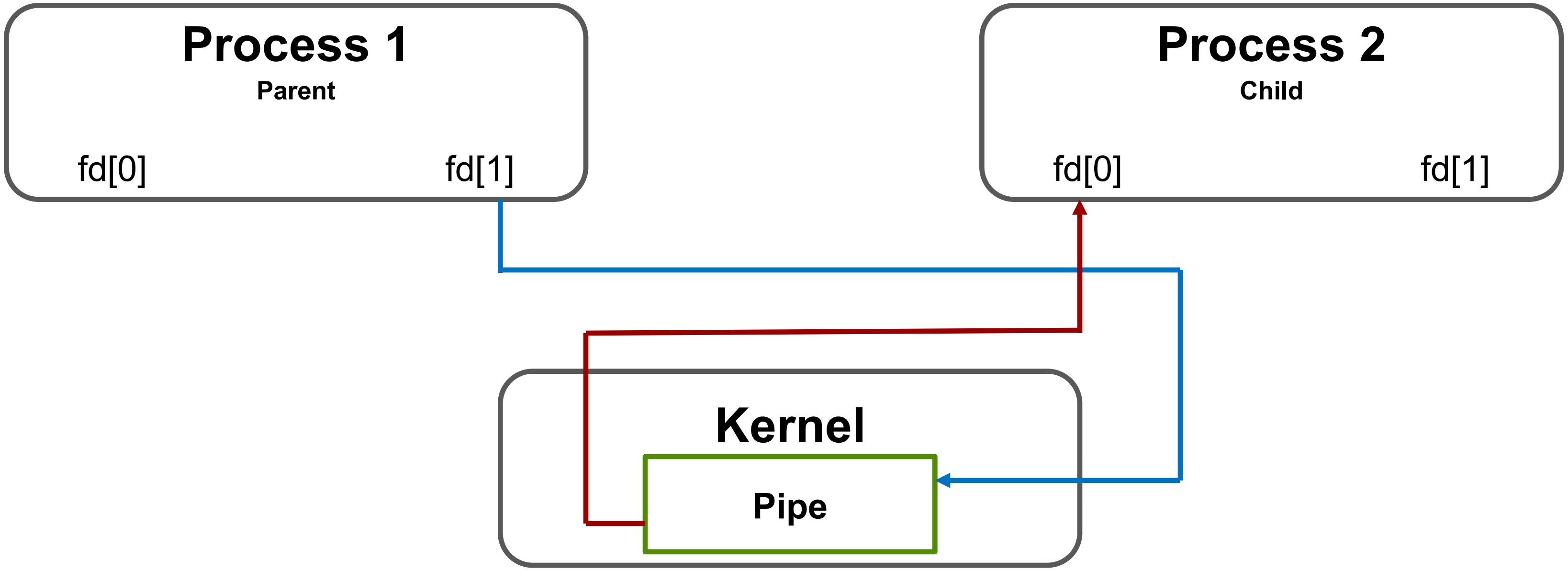
- Where fd = file descriptor
- fd[0] for reading
- fd[1] for writing
- The pipe is implemented using kernel memory.
- A pipe is a channel with two ends
- This example is a half-duplex pipe. Why??
- This example is of no real practical use, why would a single process need to communicate with itself?

Process forks



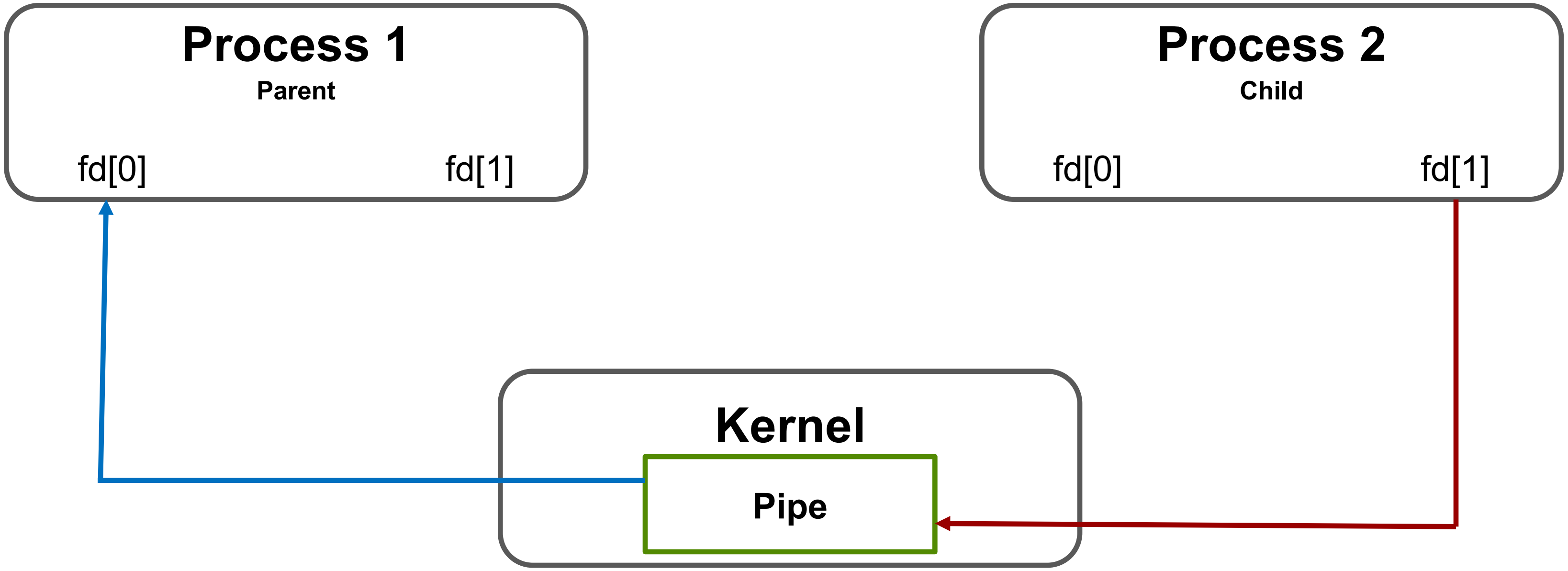
- After a fork we have to decide in which direction the data should flow
  - Parent to child
  - Child to parent

Parent to Child



- Parent must close fd[0]
- Child must close fd[1]

Child to Parent



- Parent must close fd[1]
- Child must close fd[0]

Rules for closing pipes::

- If we read from a pipe that was closed, the return should be 0
- If we write to a pipe with read closed on the other end, the SIGPIPE signal will be sent. This will need to be dealt with.

Creating a Pipe:: Step 1

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
```

Step 2:: Variables, pipe and fork

```
int       fd[2], nbytes;
pid_t     pid;
char      string[] = "Howya Friend!!!\n";
char      readbuffer[100];

// Create the pipe
pipe(fd);

// create a child process
pid = fork();
```

Step 3: Setup Child file Descriptors

```
if(pid == 0) // This is the child
{
        // Take no input, close fd[0]
        close(fd[0]);

        // Send output to parent via write
        write(fd[1], string, (strlen(string)+1));
        exit(0);
}
```

Step 4: Setup Child file Descriptors

```
else // This is the parent
{
        // Send no output, close fd[1]
        close(fd[1]);

        // Get input from the pipe via read
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Message from Child: %s", readbuffer);
}
```

Run

jmccarthy@debianJMC2017: ~/Documents/Apps/week5/pipes

File Edit View Search Terminal Help

```
jmccarthy@debianJMC2017:~/Documents/Apps/week5/pipes$ gcc -o pipe2 pipes2.c
jmccarthy@debianJMC2017:~/Documents/Apps/week5/pipes$ ./pipe2
Message from Child: Howya Friend!!!
jmccarthy@debianJMC2017:~/Documents/Apps/week5/pipes$ 
```

Pipe Example 2

- Create a C program to replicate the following:
  - **ps aux | grep login**

- How can pipes be used to solve this problem??

dup command

```
DUP(2)                      Linux Programmer's Manual                      DUP(2)

NAME
       dup, dup2, dup3 - duplicate a file descriptor

SYNOPSIS
       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int dup3(int oldfd, int newfd, int flags);
```

dup command

dup2()

The **dup2**() system call performs the same task as **dup**(), but instead of using the lowest-numbered unused file descriptor, it uses the descriptor number specified in newfd. If the descriptor newfd was previously open, it is silently closed before being reused.

The steps of closing and reusing the file descriptor newfd are performed atomically. This is important, because trying to implement equivalent functionality using **close**(2) and **dup**() would be subject to race conditions, whereby newfd might be reused between the two steps. Such reuse could happen because the main program is interrupted by a signal handler that allocates a file descriptor, or because a parallel thread allocates a file descriptor.

Note the following points:

* If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.

* If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then **dup2**() does nothing, and returns newfd.

dup command

```
dup3()
    dup3() is the same as dup2(), except that:

    *  The  caller  can  force the close-on-exec flag to be set for the new
       file descriptor by specifying O_CLOEXEC in flags.  See the  descrip-
       tion of the same flag in open(2) for reasons why this may be useful.

    *  If oldfd equals newfd, then dup3() fails with the error EINVAL.
```

Pipes IPC Example 2

```c
#include <stdlib.h>
#include <unistd.h>

void exec1();
void exec2();


int pid;
int pipefd[2];
```

- Init variables and functions.
- We have two functions that will use the exec command to perform different tasks in the process.
- Pid stores the process ID
- Pipefd array stores the file descrpitors.

Pipes IPC Example 2

```
void main() {

  // create pipe1
  if (pipe(pipefd) == -1) {
    perror("Error Init Pipe");
    exit(1);
  }
```

- Start the program (main)
- Create a pipe and store the associated file descriptors in pipefd array

Pipes IPC Example 2

```
// fork (ps aux)
if ((pid = fork()) == -1) {
  perror("Error Init Fork 1");
  exit(1);
} else if (pid == 0) {
  exec1();
}
```

- Fork to create a child process
- If the pid is 0, we are dealing with the child process. Call the exec1 function to get the child process to swap to a different task.
- (exec1 function described on slide 27)

Pipes IPC Example 2

```
  // fork (grep login)
  if ((pid = fork()) == -1) {
    perror("Error Init Fork 1");
    exit(1);
  } else if (pid == 0) {
    exec2();
  }

  close(pipefd[0]);
  close(pipefd[1]);
} // close main
```

- Fork again to create a child process
- If the pid is 0, we are dealing with the child process. Call the exec2 function to get the child process to swap to a different task.
- (exec1 function described on slide 28)

Pipes IPC Example 2

```
void exec1() {
  // input from stdin - ok
  // output to pipe
  dup2(pipefd[1], 1);
  // close fds
  close(pipefd[0]);
  close(pipefd[1]);
  // exec
  execlp("ps", "ps", "aux", NULL);
  // exec didn't work, exit
  perror("Error with ps aux");
  exit(1);
}
```

Pipes IPC Example 2

```
void exec2() {
  // input from pipe
  dup2(pipefd[0], 0);
  // output to stdout - default

  // close fds
  close(pipefd[0]);
  close(pipefd[1]);

  // exec
  execlp("grep", "grep", "login", NULL);
  // exec didn't work, exit
  perror("Error with grep login");
  exit(1);
}
```

Pipes – Making Life Easier!!

- The popen() function can be used to automate the creation of the pipe and the forking process.
- fp = popen("ls *", "r");
- Popen will set the file descriptors to facilitate the communication between parent and child or child and parent as required.

Pipes – Making Life Easier!!

popen.c ✕ | *otherProcess.c ✕

```c
#include <stdio.h>

int main() {
   FILE *fp;
   int status;
   char path[1024];

   fp = popen("ls *", "r");

   while (fgets(path, 1024, fp) != NULL)
      printf("%s", path);

   status = pclose(fp);

}
```

- Popen
  1. Setup Pipes
  2. Fork()
  3. Child process runs **ls *** command and returns results to parent
  4. Parent reads with fgets and displays with printf

Named Pipes

- A named pipe operates like a normal pipe
- The main difference is the named pipe exists as a special file
- The special file is a FIFO file, and doesn't contain any user info
- Processes of different ancestry can share date using named pipes.

FIFO file for a Named Pipe

- A named pipe operates like a file
- We can use the standead file IO system calls to operate a named pipe.
  - int open(const char *pathname, int flags);
  - int read(int fd, void *buf, size_t count);
  - int write(int fd, const void *buf, size_t count);
  - int close(fd);

FIFO Named Pipe using mkfifo

- The mkfifo function can be used to create a FIFO file.
- **mkfifo(fifoFile, 0666);**
- Use read and write operations to send information between processes.
- The processes don’t need to be related (ancestors)
- Unlink can be called to remove the file.
- **unlink(fifoFile);**

FIFO Named Pipe Example

Open | mainProcess.c ~/Documents/Apps/week5/pipes/namedPipes

mainProcess.c | otherProcess.c

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd;
    char * fifoFile = "/tmp/fifoFile";

    /* create the FIFO (named pipe) */
    mkfifo(fifoFile, 0666);

    fd = open(fifoFile, O_WRONLY);
    write(fd, "The truth is out there!!", sizeof("The truth is out there!!"));
    close(fd);

    unlink(fifoFile);

    return 0;
}
```

FIFO Named Pipe Example

Open

~/D

*otherProcess.c

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

#define MAX_BUF 1024

int main()
{
    int fd;
    char * fifoFile = "/tmp/fifoFile";
    char buf[MAX_BUF];

    fd = open(fifoFile, O_RDONLY);
    read(fd, buf, MAX_BUF);
    printf("Message in: %s\n", buf);
    close(fd);

    return 0;
}
```

Run Example

```
jmccarthy@debianJMC2017: ~/Documents/
File  Edit  View  Search  Terminal  Help
$./mainProcess
$
```

```
jmccarthy@debianJMC2017: ~/Docu
File  Edit  View  Search  Terminal  Help
$./otherProcess
Message in: The truth is out there!!
$
```

Questions