INTER-PROCESS COMMUNICATION USING PIPES

THE PIPE

- A pipe is typically used as a one-way communications channel which couples one related process to another.
- UNIX deals with **pipes** the same way it deals with **files.**
- A process can send data ‘down’ a pipe using a **write** system call and another process can receive the data by using **read** at the other end.

PIPES AT THE COMMAND LEVEL

- `$ who | sort`
- This command causes the shell to start the commands **who** and **sort** simultaneously.
- The '|' symbol in the command line tells the shell to create a pipe to couple the standard output of **who** to the standard input of **sort**.
- The final result of this command should be a nicely ordered list of the users logged onto this machine.

DISSECTING THE PIPE

- The **who** command on the left side of the pipe does not know that its **stdout** is being sent to a pipe.
- The **who** command simply writes to **stdout**.
- Similarly the **sort** command does not know that it is getting its input from a pipe.
- The **sort** command simply reads the **stdin**.
- Both commands behave normally!

DISSECTING THE PIPE

- The overall effect is logically as if the following sequence has been executed.

```
$ who > sometempfile.txt

$ sort sometempfile.txt

$ rm sometempfile.txt
```

- Flow control is maintained automatically by the OS (if **who** writes faster than **sort** can read, **who** is suspended, until **sort** catches up).

PROGRAMMING WITH PIPES

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

- Within programs a pipe is created using the system call `pipe()`.

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

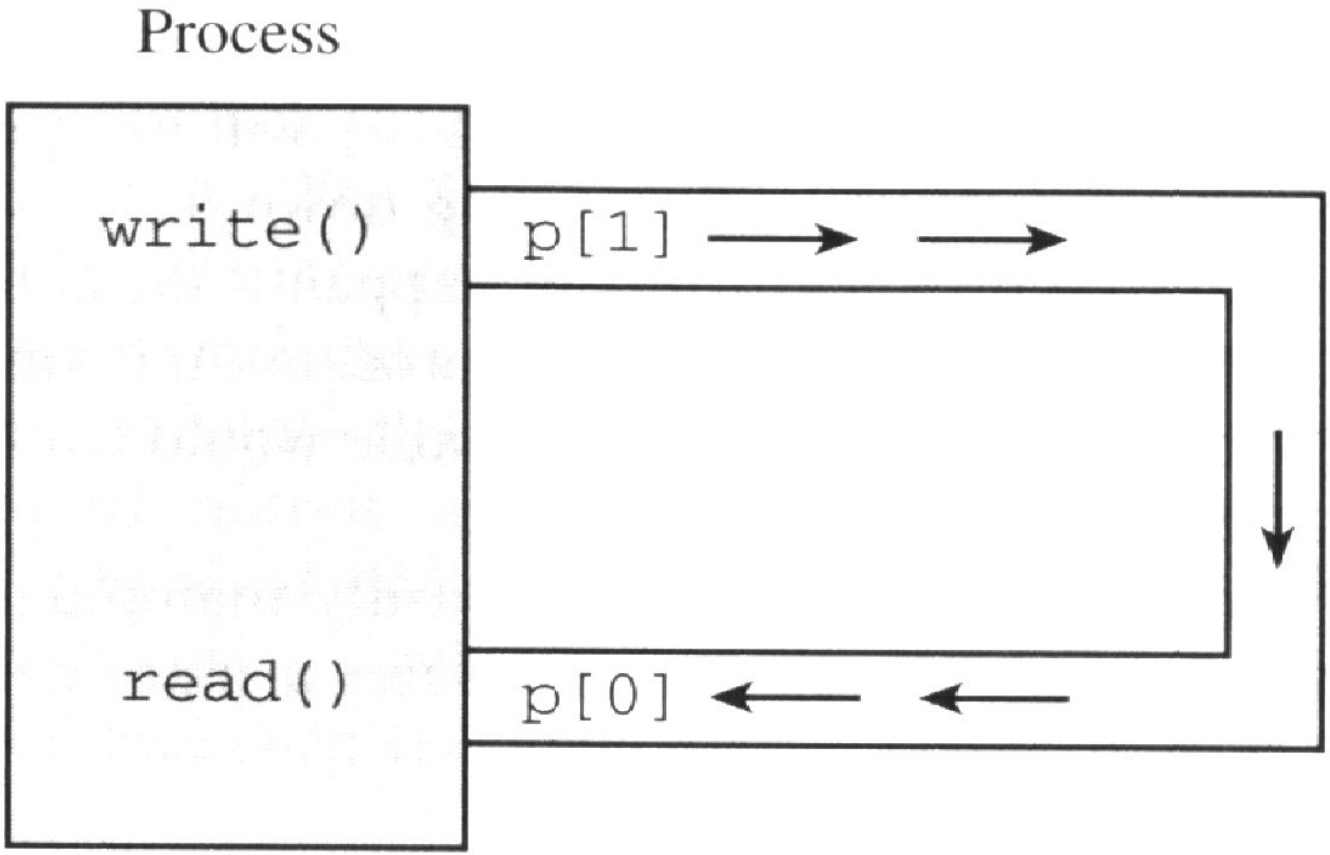
- If successful, this call returns two file descriptors:
  - one for writing down the pipe,
  - one for reading from it.

PROGRAMMING WITH PIPES

- `filedes` is a two-integer array that will hold the file descriptors that will identify the pipe.
- If successful, `filedes[0]` will be open for reading from the pipe and `filedes[1]` will be open for writing down it.
- `pipe()` can fail (returns -1) if it cannot obtain the file descriptors (exceeds user-limit or kernel-limit).

```
/* first pipe example */
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

/* this figure includes terminating null */
#define MSGSIZE 16

char *msg1 = "Hello, World #1";
char *msg2 = "Hello, World #2";
char *msg3 = "Hello, World #3";

main()
{
 char inbuf[MSGSIZE];
 int p[2], j;

 /* open pipe */
 if(pipe(p) == -1)
 {
   perror("pipe call");
   exit(1);
 }
```

```
 /* write down pipe */
 write(p[1],msg1,MSGSIZE);
 write(p[1],msg2,MSGSIZE);
 write(p[1],msg3,MSGSIZE);

 /* read from pipe */
 for(j = 0; j < 3; j++)
 {
   read(p[0],inbuf,MSGSIZE);
   printf("%s\n",inbuf);
 }

 exit(0);
}
```



- The output of **`pipes.c`**:

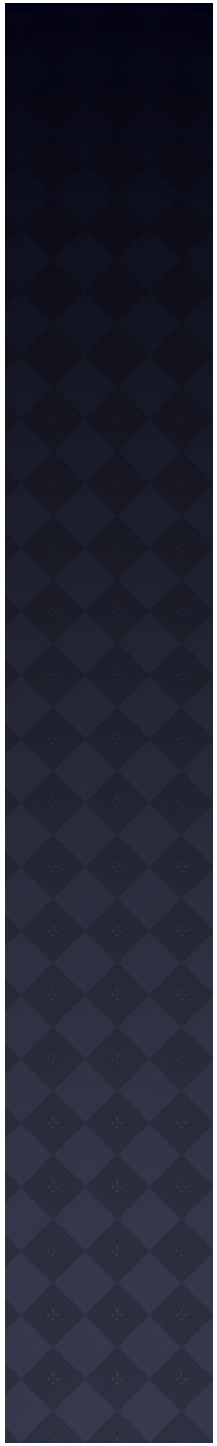
```
Hello, World #1
Hello, World #2
Hello, World #3
```

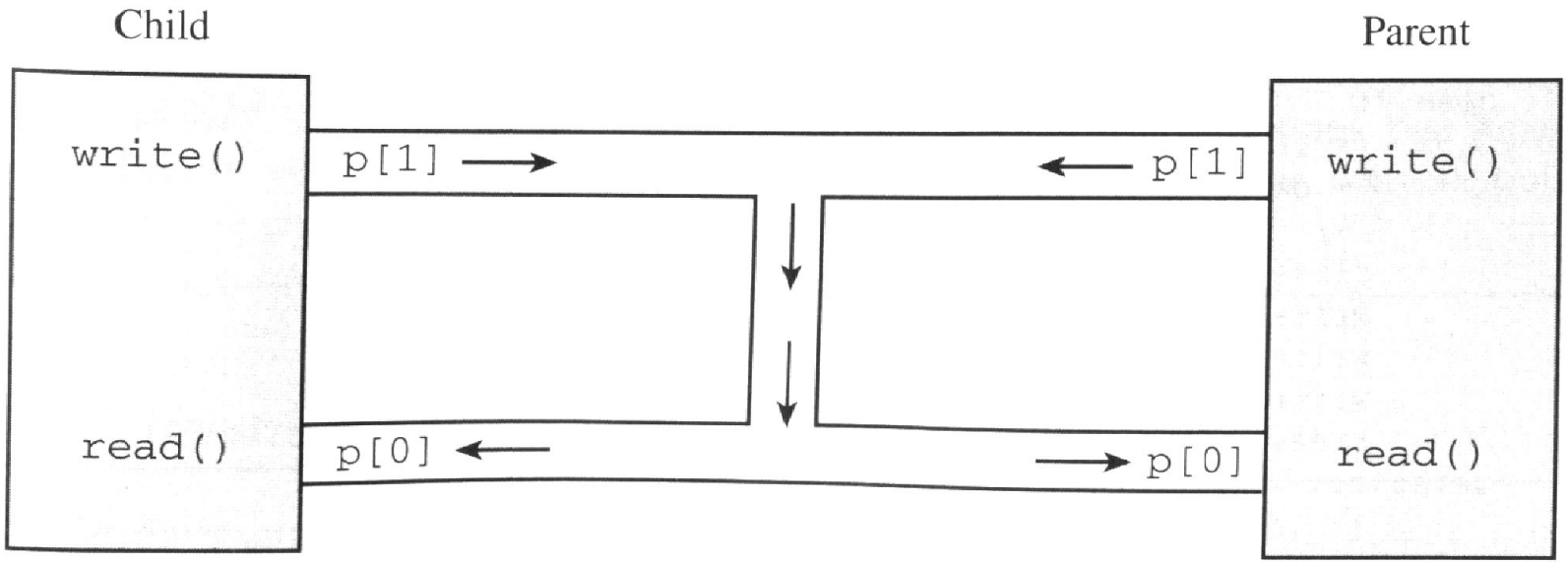
MORE ABOUT PIPES

- Pipes behave *first-in-first-out*, this cannot be changed; `lseek()` will not work in pipes!
- The size of the read and write don't have to match (For example, you can write 512 bytes at a time while reading 1 byte at a time).
- The previous example is trivial as there is only one process involved and the process is sending messages to itself!
- Pipes become powerful when used with `fork()`.

```c
/* pipes_2.c -- second pipe example */
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

#define MSGSIZE 16

char *msg1 = "Hello, World #1";
char *msg2 = "Hello, World #2";
char *msg3 = "Hello, World #3";

main()
{
 char inbuf[MSGSIZE];
 int p[2], j;
 pid_t pid;

 /* open pipe */
 if(pipe(p) == -1)
 {
   perror("pipe call");
   exit(1);
 }

 switch(pid = fork())
 {
     case -1:
          perror("Fork Failure");
          exit(2);
```

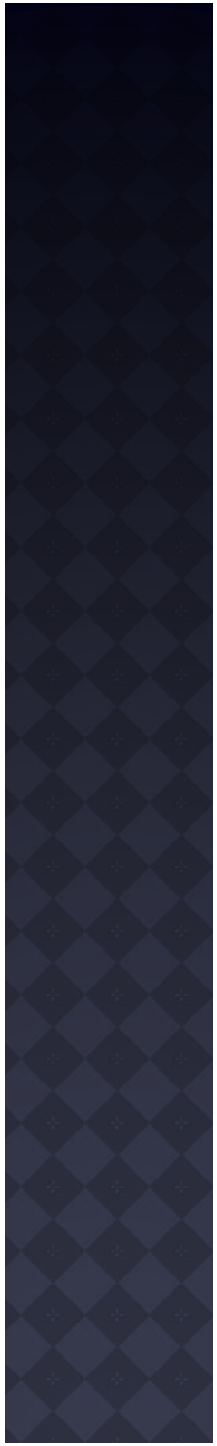
```
        case 0:
                /* if child, then write down pipe */
                write(p[1],msg1,MSGSIZE);
                write(p[1],msg2,MSGSIZE);
                write(p[1],msg3,MSGSIZE);

                break;

        default:
                /* if parent, then read from pipe */
                for(j = 0; j < 3; j++)
                {
                        read(p[0],inbuf,MSGSIZE);
                        printf("%s\n",inbuf);
                }

                wait(NULL);
  }

  exit(0);
}
```



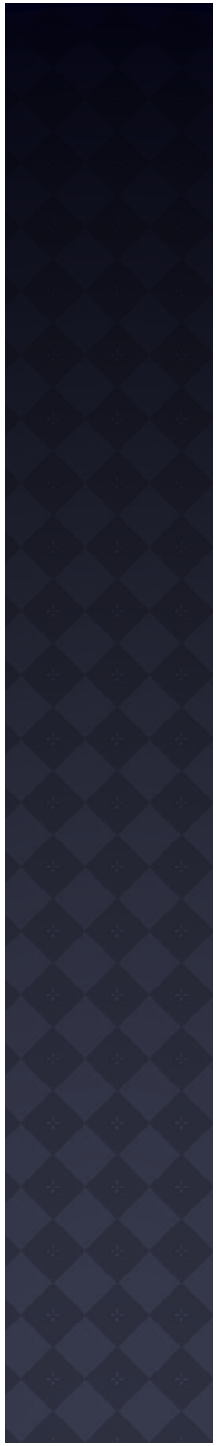


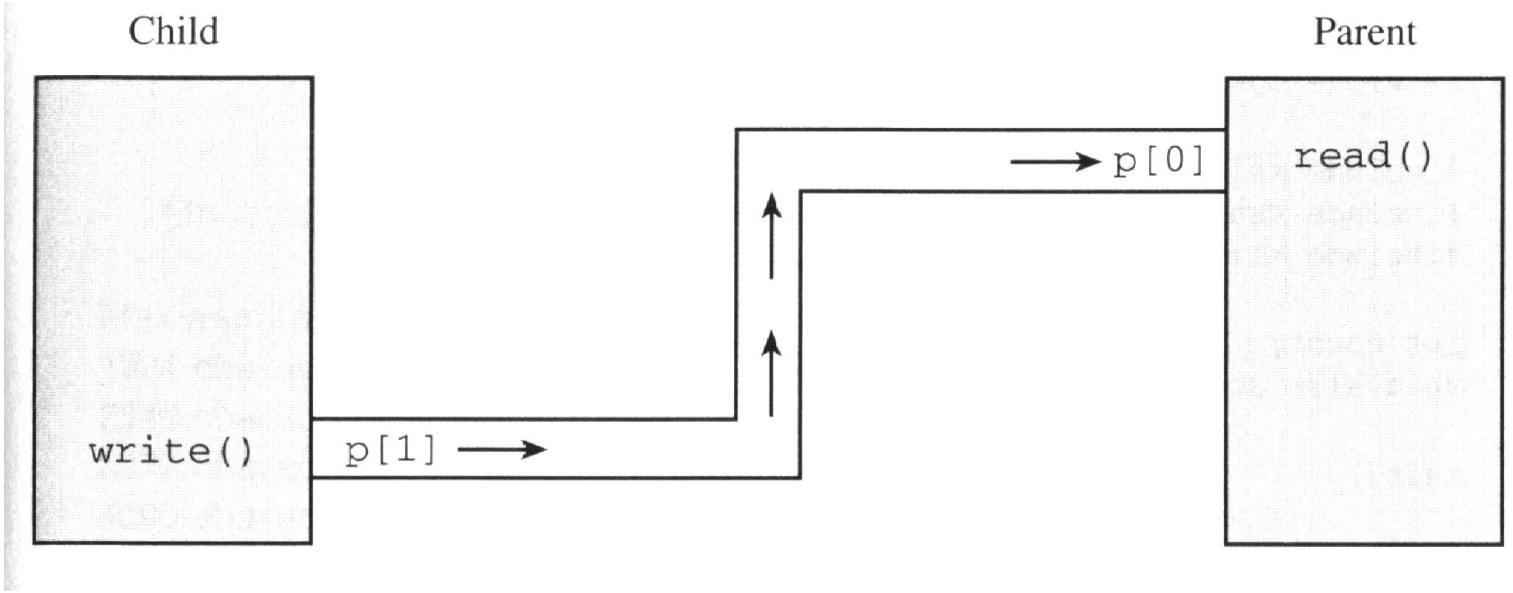
- Do you see a problem here?
- What happens if both attempt to write and read at the same time?
- Pipes are meant as uni-directional communication devices.

```
/* pipes_3.c -- third pipe example */
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

#define MSGSIZE 16

char *msg1 = "Hello, World #1";
char *msg2 = "Hello, World #2";
char *msg3 = "Hello, World #3";

main()
{
 char inbuf[MSGSIZE];
 int p[2], j;
 pid_t pid;

 /* open pipe */
 if(pipe(p) == -1)
 {
   perror("pipe call");
   exit(1);
 }

 switch(pid = fork())
 {
       case -1:
              perror("Fork Failure");
              exit(2);
```



```
        case 0:
                /* if child, close read file,
                 * then write down pipe
                 */
                close(p[0]);
                write(p[1],msg1,MSGSIZE);
                write(p[1],msg2,MSGSIZE);
                write(p[1],msg3,MSGSIZE);

                break;

        default:
                /* if parent, close write file,
                 * then read from pipe
                 */
                close(p[1]);
                for(j = 0; j < 3; j++)
                {
                        read(p[0],inbuf,MSGSIZE);
                        printf("%s\n",inbuf);
                }

                wait(NULL);
 }

 exit(0);
}
```



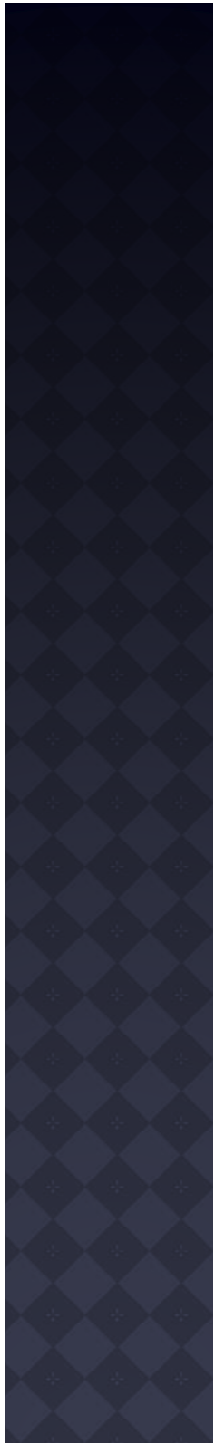


- We now have a uni-directional pipe connecting two processes!

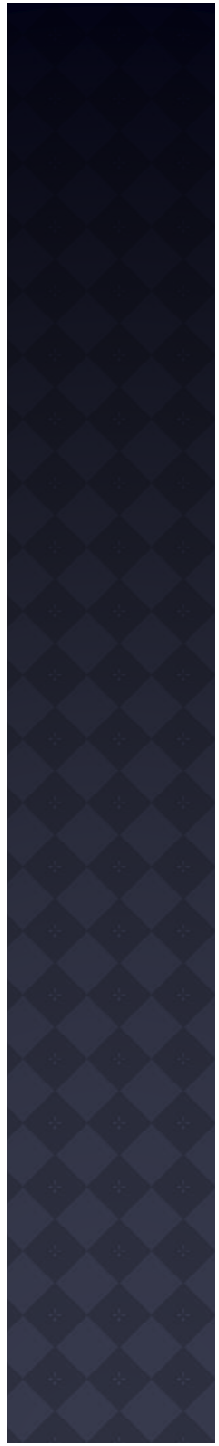
CLOSING PIPES

- Closing the **write** file descriptor
  - If all processes close the *write-only end* of the pipe and the pipe is empty, any process attempting a read will return no data (will return 0 like EOF).
- Closing the **read** file descriptor
  - If all processes close the *read-only end* of the pipe and there are processes waiting to write to the pipe, the kernel will send a *SIGPIPE* signal.
    - If the signal is not caught the process will terminate.
    - If the signal *is* caught, then after the interrupt routine has completed, *write* will return -1.

```
/* pipes_4.c -- fourth pipe example */
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

#define MSGSIZE 16

char *msg1 = "Hello, World #1";
char *msg2 = "Hello, World #2";
char *msg3 = "Hello, World #3";

main()
{
 char inbuf[MSGSIZE];
 int p[2], j;
 pid_t pid;

 /* open pipe */
 if(pipe(p) == -1)
 {
   perror("pipe call");
   exit(1);
 }

 switch(pid = fork())
 {
       case -1:
              perror("Fork Failure");
              exit(2);
```



```
        case 0:
                /* if child, close read file,
                 * then write down pipe,
                 * then close write file.
                 */
                close(p[0]);
                write(p[1],msg1,MSGSIZE);
                write(p[1],msg2,MSGSIZE);
                write(p[1],msg3,MSGSIZE);
                close(p[1]);
                break;

        default:
                /* if parent, close write file,
                 * then read from pipe until
                 * pipe is empty.
                 */
                close(p[1]);
                while(read(p[0],inbuf,MSGSIZE) != 0)
                        printf("%s\n",inbuf);

                wait(NULL);
 }

 exit(0);
}
```



THE ASSIGNMENT

- write a C program that allows communication between a parent process and child process using pipe.
  - Parent process has to take input n from the user, where n is an integer. The parent process should then write n to the pipe
  - The child process has to read the contents of the pipe and then print all the odd numbers till the limit n.

- write a C program that allows communication between a parent process and child process using pipe.
  - Parent process has to take input string from the user
  - The child process has to read the contents of the pipe and convert it to capital letters