CS3071 : Operating Systems Laboratory

A.Y. 2021-2022 (Autumn)

Faculty : Dr. Bidyut Kumar Patra

Inter-Process Communication (Pipe - Unnamed)

27/09/2021 , 13:15 - 16:15 hours

What is Inter Process Communication (IPC) ?

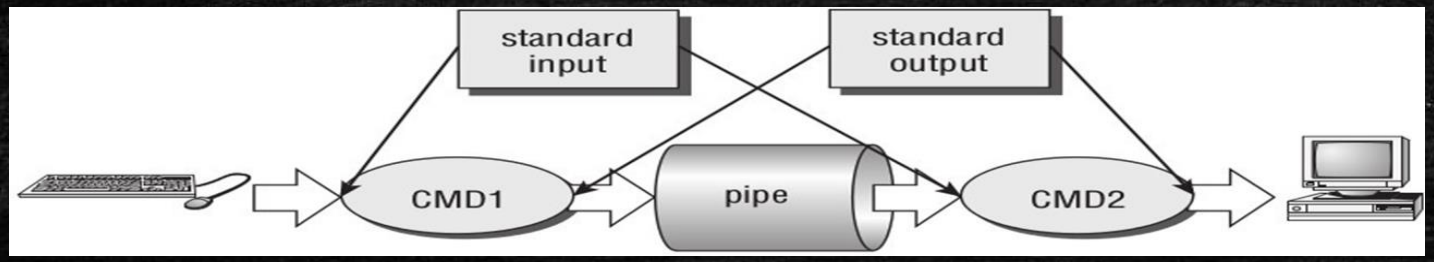
- An Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other.
- This involves synchronizing their actions and managing shared data .

IPC Methodology

Sr. No.	Methodology	Description
1.	**Pipe (unnamed/simple)**	Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process.
2.	**Pipe (named/FIFO)**	Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.
3.	**Message Queues**	Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.
4.	**Shared Memory**	Communication between two or more processes is achieved through a shared piece of memory among all processes.
5.	**Semaphores**	Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed. This needs to be repeated by all the processes to secure data.
6.	**Signals**	Signal is a mechanism to communication between multiple processes by way of signaling.

Pipes

- Takes output from one program, or process, and sends it to another.
- Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and so on.
- Pipes are unidirectional i.e. data flows from left to right through the pipeline.
- “ | ” is used to implement pipes for shell commands.



Example of using Pipe in shell

Open pipeeg.sh Save

```
1 #!/bin/bash
2
3 filename=test.txt
4
5 cat $filename
6
7 ls -l | find ./ -type f -name "*.txt" | wc -l
8
9 ls -l | find ./ -type f -name "*.txt" >> $filename
10
11 cat $filename
```

sh Tab Width: 8 Ln 11, Col 14 INS

sumanto@sumanto-VirtualBox: ~

```
sumanto@sumanto-VirtualBox:~$ chmod 775 pipeeg.sh
sumanto@sumanto-VirtualBox:~$ ./pipeeg.sh
line 1
line 2
15
line 1
line 2
./xx.txt
./.mozilla/firefox/vxd49l4r.default-release/SecurityPreloadSt
ate.txt
./.mozilla/firefox/vxd49l4r.default-release/AlternateServices
.txt
./.mozilla/firefox/vxd49l4r.default-release/pkcs11.txt
./.mozilla/firefox/vxd49l4r.default-release/SiteSecurityServi
ceState.txt
./.mozilla/firefox/vxd49l4r.default-release/serviceworker.txt
./.mozilla/firefox/vxd49l4r.default-release/TRRBlacklist.txt
./.mozilla/firefox/vxd49l4r.default-release/cert_override.txt
./.cache/tracker/locale-for-miner-apps.txt
./.cache/tracker/last-crawl.txt
./.cache/tracker/first-index.txt
./.cache/tracker/db-version.txt
./.cache/tracker/parser-version.txt
./.cache/tracker/db-locale.txt
./test.txt
sumanto@sumanto-VirtualBox:~$
```

pipe(int file_descriptor[2]) – System call

- Pipe function provides a means of passing data between two programs, without the overhead of invoking a shell to interpret the requested command.

int pipe(int file_descriptor[2])

- Pipe is passed (a pointer to) an *array of two integer file descriptors*.
- It fills the array with two new file descriptors and *returns a zero on success*.
- ***On failure, it returns -1*** and sets errno to indicate the reason for failure.
- *file_descriptor[0]* is for reading and *file_descriptor[1]* is for writing.
- Whatever is written into *file_descriptor[1]* can be read from *file_descriptor[0]*.
- Header file to include - *#include <unistd.h>*

*size_t write(int fildes, const void *buf, size_t nbytes)* – System call

*size_t write(int fildes, const void *buf, size_t nbytes)*

- It arranges for the first nbytes bytes from buf to be written to the file associated with the file descriptor fildes.
- It returns the number of bytes actually written. This may be less than n-bytes if there has been an error in the file descriptor.
- If the function returns 0, it means no data was written.
- if it returns –1, there has been an error in the write call.
- Header file to include - *#include <unistd.h>*

*size_t read(int fildes, void *buf, size_t nbytes)* – System call

*size_t read(int fildes, void *buf, size_t nbytes)*

- It reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf.
- It returns the number of data bytes actually read, which may be less than the number requested.
- If a read call returns 0, it had nothing to read; it reached the end of the file.
- Again, an error on the call will cause it to return –1.
- Header file to include - *#include <unistd.h>*

Example of reading and writing into pipe

Open | pipe1.c | Save

```c
#include<stdio.h>
#include<string.h>
#include<unistd.h>
int main()
{
        int file_pipes[2], data_pro;
        char data[] = "Hello OS LAB";
        char buffer[20];

        if (pipe(file_pipes) == 0)
        {
                //Writing from pipe using file_pipes[1]
                data_pro = write(file_pipes[1], data, strlen(data));
                printf("Wrote %d bytes\n", data_pro);

                //Reading from pipe using file_pipes[0]
                data_pro = read(file_pipes[0], buffer, strlen(data));
                printf("Read %d bytes: %s\n", data_pro, buffer);
        }
return 0;
}
```

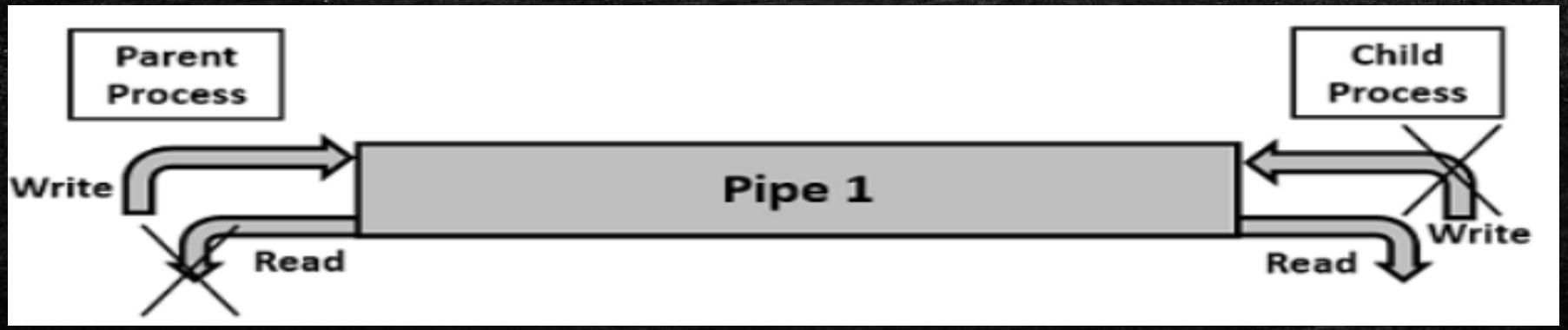
C | Tab Width: 8 | Ln 17, Col 68 | INS

sumanto@sumanto-VirtualBox: ~

```
sumanto@sumanto-VirtualBox:~$ gcc pipe1.c
sumanto@sumanto-VirtualBox:~$ ./a.out
Wrote 12 bytes
Read 12 bytes: Hello OS LAB\U
sumanto@sumanto-VirtualBox:~$
```

Pipe with fork

- A process creates a pipe just before it forks one or more child processes.
- The pipe is then used for communication either between the parent or child processes, or between two sibling processes.
- To ensure pipe work properly, you should: Always be sure to close the end of pipe you aren't concerned with. That is, if the parent wants to receive data from the child, it should close file_pipes[1], and the child should close file_pipes[0]. When processes finish reading or writing, close related file descriptors. Otherwise, there will be undesired synchronization problems.



Example of reading and writing into pipe with fork

Open pipe2.c Save

```c
#include<stdio.h>
#include<string.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
        int file_pipes[2], data_pro, pid;
        char data[] = "Hello OS LAB", buffer[20];

        pipe(file_pipes); // Pipe is created

        pid = fork();
        if (pid == 0)
        {
                close(file_pipes[0]); //parent close the read end.
                data_pro = write(file_pipes[1], data, strlen(data));
                printf("Wrote %d bytes\n", data_pro);
                //after finishing writing, parent close the write end
                close(file_pipes[1]);
                wait(NULL);
        }
        else
        {
                close(file_pipes[1]); //child close the write end.
                data_pro = read(file_pipes[0], buffer, 20);
                printf("Read %d bytes: %s\n", data_pro, buffer);
                //after finishing reading, child close the read end
                close(file_pipes[0]);
        }
return 0;
}
```

C Tab Width: 8 Ln 13, Col 22 INS

sumanto@sumanto-VirtualBox: ~

```
sumanto@sumanto-VirtualBox:~$ gcc pipe2.c
sumanto@sumanto-VirtualBox:~$ ./a.out
Wrote 12 bytes
Read 12 bytes: Hello OS LABQV
sumanto@sumanto-VirtualBox:~$
```

Example of child forget to close write end

pip3.c

```c
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char* argv[])
{
    int pipefds[2];
    pid_t pid;
    char buf[30];

    pipe(pipefds); //create pipe

    memset(buf,0,30);
    pid = fork();
    if (pid > 0) {

      printf(" PARENT write in pipe\n");

          close(pipefds[0]); //parent close the read end
          write(pipefds[1], "CS3071OSLAB", 11); //write into pipe
          close(pipefds[1]); //after finishing writing, parent close the write end

      wait(NULL);//parent wait for child
    }
    else {
      //child read from the pipe read end until the pipe is empty
      while(read(pipefds[0], buf, 1)==1)
                printf("CHILD read from pipe -- %s\n", buf);

      close(pipefds[0]); //after finishing reading, child close the read end

      printf("CHILD: EXITING!");
      exit(EXIT_SUCCESS);
    }
    return 0;
}
```

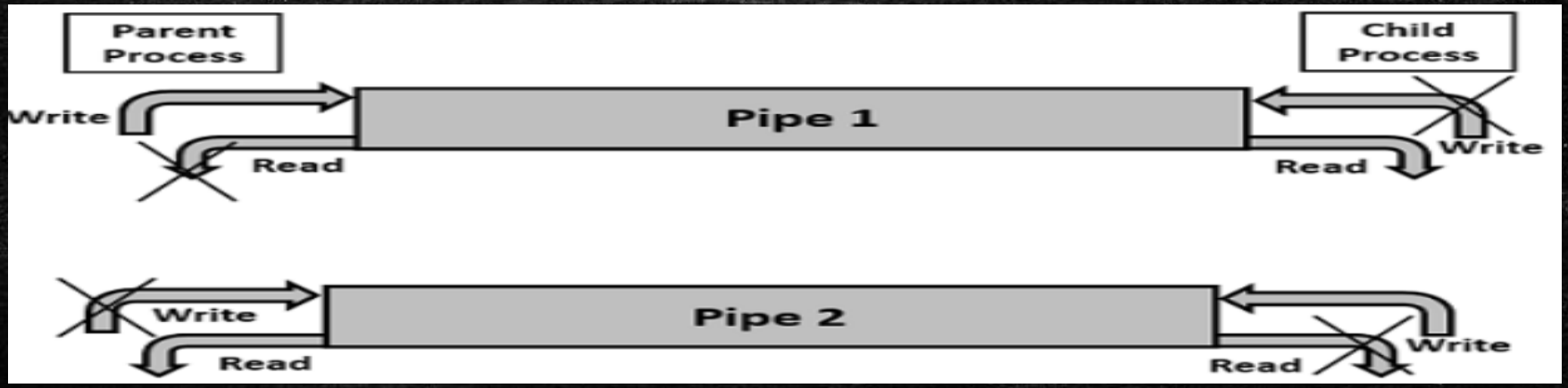
C Tab Width: 8 Ln 15, Col 18 INS

sumanto@sumanto-VirtualBox: ~

```
sumanto@sumanto-VirtualBox:~$ gcc pip3.c
sumanto@sumanto-VirtualBox:~$ ./a.out
 PARENT write in pipe
CHILD read from pipe -- C
CHILD read from pipe -- S
CHILD read from pipe -- 3
CHILD read from pipe -- 0
CHILD read from pipe -- 7
CHILD read from pipe -- 1
CHILD read from pipe -- O
CHILD read from pipe -- S
CHILD read from pipe -- L
CHILD read from pipe -- A
CHILD read from pipe -- B
```

Achieving two-way communication using pipes

1. Create pipe1 for the parent process to write and the child process to read.
2. Create pipe2 for the child process to write and the parent process to read.
3. Close the unwanted ends of the pipe from the parent and child side.
4. Parent process to write a message and child process to read and display on the screen.
5. Child process to write a message and parent process to read and display on the screen.



Thank You.