Inter Process Communications

Inter Process Communications

Introduction

- In a multiprocessing environment, often many processes are in need to communicate with each other and share some of the resources.
- IPC mechanisms have many distinct purposes:

  for example

- Data transfer
- Event notification
- Synchronisation
- Sharing data
- Resource sharing

- The UNIX operating system provides a rich set of features that allows processes to communicate with each other.

IPC Mechanisms

- Primitive
  - Unnamed pipe
  - Named pipe (FIFO)

- System V IPC
  - Message queues
  - Shared memory
  - Semaphores

- POSIX 1.b
  - Message queues
  - Shared memory
  - Semaphores

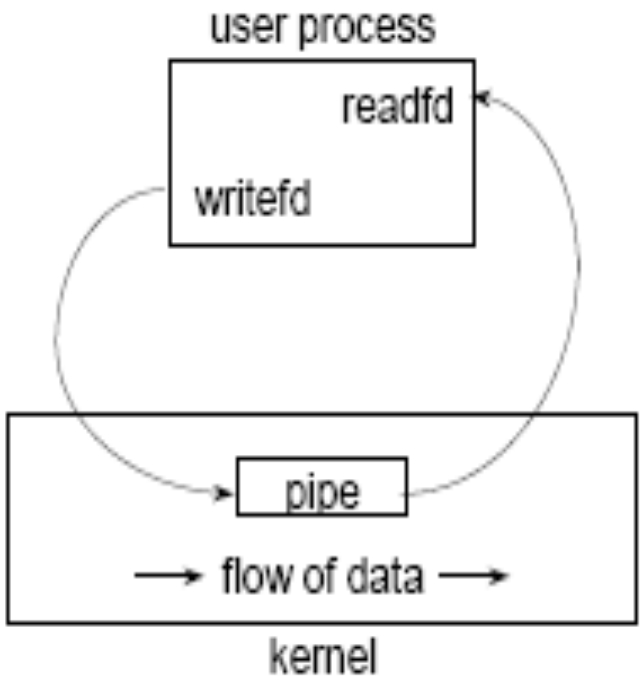
- Socket Programming

IPC

- pipes & fifos are simplest & fastest amongst all IPCs
- shared memory is fastest amongst sysVor POSIX IPCs
- but shared memory without synchronization is dangerous
- message queues maintain boundary for messages
- messages once read is deleted in pipes/fifos/message_queues

Unnamed Pipes

- There is no form of IPC that is simpler than pipes.
- In Unix, a *pipe* is a unidirectional, stream communication that allows related-processes to communicate.
- One process writes to the ``write end" of the pipe, and a second process reads from the ``read end" of the pipe.
- The order in which data is written to the pipe, is the same order as that in which data is read from the pipe.
- pipe is a data structure in the kernel.



Unnamed Pipes

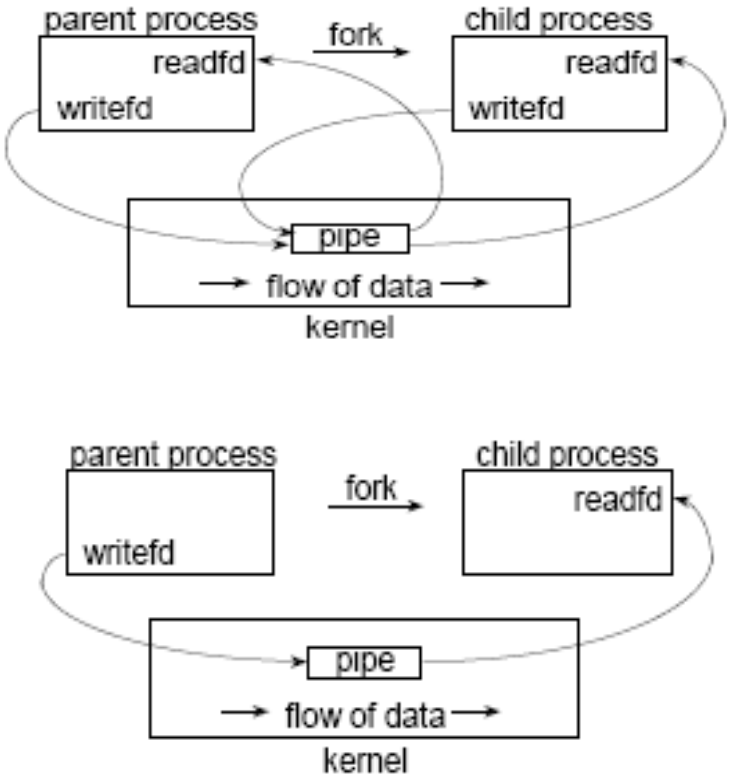
- A pipe is created by using the pipe system call

  int pipe(int* filedes);

- A pipe consists of
  - two descriptors, one for reading, one for writing.
  - reading from the pipe advances the read cursor
  - writing to the pipe advances the write cursor
  - operating system blocks reads of empty pipe
  - operating system blocks writes to full pipe
  - pipe data consists of unstructured character *stream*

Pipe Creation & Usage

- First, a process creates a pipe, and then forks to create a copy of itself.
- **Parent opens & writes file, child reads file**
  - parent closes read end of pipe
  - child closes write end of pipe



EXAMPLE

```
#define DATA "hello world"
#define BUFFSIZE 1024

int rgfd[2]; /* file descriptors
                of streams */
main()
      {
      char sbBuf[BUFFSIZE];
      pipe(rgfd);
      if (fork())
          { /* parent, read from pipe */
          close(rgfd[1]); /* close write end */
          read(rgfd[0],
          sbBuf, BUFFSIZE);
          printf("-->%s\n", sbBuf);
          close(rgfd[0]);
          }
      else
          { /* child, write data to pipe */
          close(rgfd[0]); /* close read end */
          write(rgfd[1], DATA,
          sizeof(DATA));
          close(rgfd[1]);
          exit(0);
          }
      }
```

Named pipe (FIFO)

- One limitation of anonymous pipes is that only processes 'related' to the process that created the pipe may communicate using them.
- If we want two un-related processes to communicate via pipes, we need to use named pipes.
- A named pipe (FIFO) is a pipe whose access point is a file kept on the file system.
- By opening this file for reading, a process gets access to the reading end of the pipe.
- By opening the file for writing, the process gets access to the writing end of the pipe.
- If a process opens the file for reading, it is blocked until another process opens the file for writing. The same goes the other way around.

Creating A Named Pipe With The mknod Command

- A named pipe may be created either via the 'mknod' (or its newer replacement, 'mkfifo'), or via the mknod() system call (or by the POSIX-compliant mkfifo() function).
- To create a named pipe with the file named 'prog_pipe', we can use the following command:

  mknod prog_pipe p
- We could also provide a full path to where we want the named pipe created. If we then type 'ls -l prog_pipe', we will see something like this:

  prw-rw-r-- 1 choo choo 0 Nov 7 01:59 prog_pipe
- The 'p' on the first column denotes this is a named pipe. Just like any file in the system, it has access permissions, that define which users may open the named pipe, and whether for reading, writing or both.

Opening/Reading/Writing From/To A Named Pipe

- Opening a named pipe is done just like opening any other file in the system, using the open() system call, or using the fopen() standard C function.
- If the call succeeds, we get a file descriptor (in the case of open()), or a 'FILE' pointer (in the case of fopen()).
- Reading from a named pipe is very similar to reading from a file, and the same goes for writing to a named pipe. Yet there are several differences:
  - Either Read Or Write - a named pipe cannot be opened for both reading and writing. The process opening it must choose one mode, and stick to it until it closes the pipe.
  - Read/Write Are Blocking - when a process reads from a named pipe that has no data in it, the reading process is blocked. It does not receive an end of file (EOF) value, like when reading from a file. When a process tries to write to a named pipe that has no reader (e.g. the reader process has just closed the named pipe), the writing process gets blocked, until a second process re-opens the named pipe.

EXAMPLE

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/stat.h>
int main()
    {
    int x,y,z;
    mknod("./fif",S_IFIFO|0666,0);
    x=open("fif",O_RDONLY);
    write(x,"hello",5);
    close(x);
    unlink("fif");
    }
```

```
#include<stdio.h>
#include<fcntl.h>

int main()
    {
    int x,y,z;
    char a[10];
    x=open("fif",O_WRONLY);
    read(x,a,5);
    write(1,a,5);
    close(x);
    }
```