Interprocess Communication

1. Pipe

➔ In general pipe means connecting a data flow from one process to another.

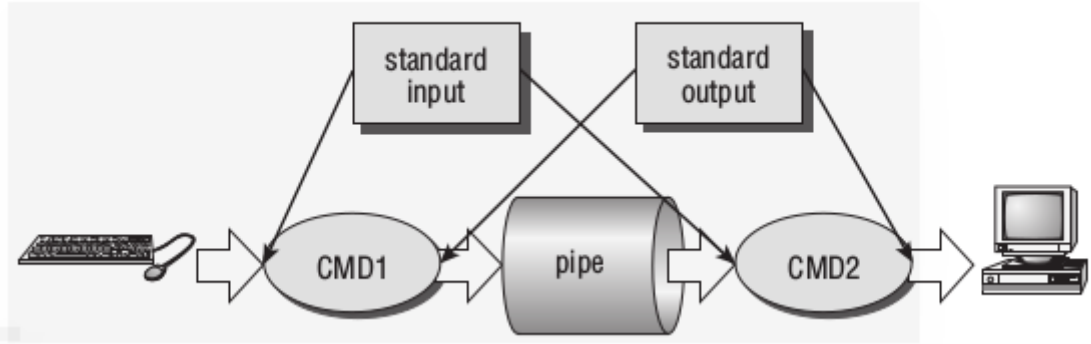


Fig. **Pipe**

➔ **Pipes can be used in threads and processes.**

➔ **A new process can be created using the system call fork().**

➔ **It returns two differnt values to the child and parent.**

➔ **The value 0 is returned to the child (new) process and the PID (Process ID) of the child is returned to the parent process. This is used to distinguish between the two processes.**

```
int pipe(int pipefd[2]);
int pipe2(int pipefd[2], int flags);
```

➔ **pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication.**

➔ **The array pipefd is used to return two file descriptors referring to the ends of the pipe.**

➔ **pipefd[0] refers to the read end of the pipe.**

➔ **pipefd[1] refers to the write end of the pipe.**

➔ **Data written to the write end of the pipe is buffered by the kernel until. it is read from the read end of the pipe.**

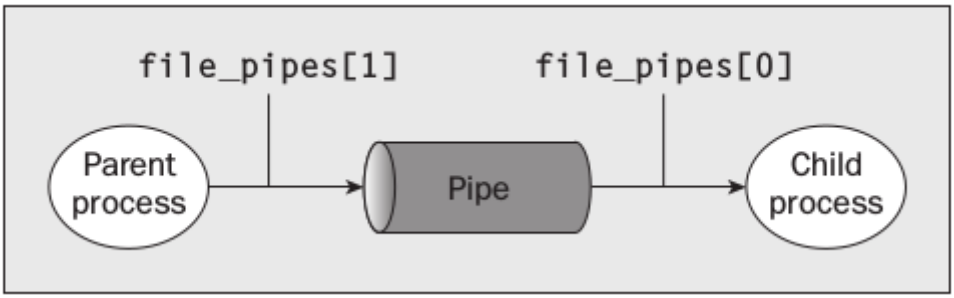
➔ **If flags is 0, then pipe2() is the same as pipe().**

➔ **A simple example of pipe.**

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
int main()
{
        int fd[2], nbytes;
        pid_t childpid;
        char string[]= “hello World !”;
        char readbuffer[100];
        pipe(fd);
        childpid=fork();
        switch(childpid)
        {
                case -1: perror(“Process creation failed”);
                          exit(1);
                case 0: close(fd[0]);
                        write(fd[1],string,(strlen(string)+1));
                        exit(0);
                default:
                         close(fd[1]);

                         nbytes=read(fd[0],readbuffer,sizeof(readbuffer));
                         printf(“Reaceived string : %s “,readbuffer);
        }

        return 0;
}
```



2. Named Pipe

- ➔ **A Named Pipe is also called as FIFO.**
- ➔ **The principal difference is that a FIFO has name within the file system and is opened in the same way as a regular file.**
- ➔ **This allows a FIFO to be used for communication between unrelated processes (e.g., a client and server).**
- ➔ **Once a FIFO has been opened, we use the same I/O system calls as are used with pipes and other files (i.e., read(), write(), and close()).**
- ➔ **Just as with pipes, a FIFO has a write end and a read end, and data is read from the pipe in the same order as it is written.**
- ➔ **This fact gives FIFOs their name: first in, first out.**
- ➔ **A FIFO is created using the `mkfifo()` library function.**

➔ **A simple example for writing and reading using mkfifo:**

1. Writer.c

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

int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";
  /* create the FIFO (named pipe) */
    mkfifo(myfifo, 0666);

    /* write "Hi" to the FIFO */
    fd = open(myfifo, O_WRONLY);
    write(fd, "Hi", sizeof("Hi"));
    close(fd);
    /* remove the FIFO */
    unlink(myfifo);
 return 0;
}
```

2. Reader.c

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#define MAX_BUF 1024

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

    /* open, read, and display the message from the FIFO */
    fd = open(myfifo, O_RDONLY);
    read(fd, buf, MAX_BUF);
    printf("Received: %s\n", buf);
   close(fd);

  return 0;
}
```

3. Shared Memory

- **One of the simplest interprocess communication methods is using shared memory.**
- **Shared memory allows two or more processes to access the same memory.**
- **Shared memory is the fastest form of interprocess communication because all processes share the same piece of memory. Access to this shared memory is as fast as accessing a process's nonshIt also avoids copying data unnecessarily.ared memory, and it does not require a system call or entry to the kernel.**
- **Because the kernel does not synchronize accesses to shared memory, you must provide your own synchronization.**
- **For example, a process should not read from the memory until after data is written there, and two processes must not write to the same memory location at the same time.**
- **A common strategy to avoid these race conditions is to use semaphores.**
- **To use a shared memory segment, one process must allocate the segment.**
- **Then each process desiring to access the segment must attach the segment.**
- **After finishing its use of the segment, each process detaches the segment. At some point, one process must deallocate the segment.**
- **Allocating a new shared memory segment causes virtual memory pages to be created. Because all processes desire to access the same shared segment, only one process should allocate a new shared segment.**
- **Allocating an existing segment does not create new pages, but it does return an identifier for the existing pages.**
- **To permit a process to use the shared memory segment, a process attaches it, which adds entries mapping from its virtual memory to the segment's shared pages.**
- **When finished with the segment, these mapping entries are removed.**
- **When no more processes want to access these shared memory segments, exactly one process must deallocate the virtual memory pages.**

➔ **A process allocate shared memory using shmget system call.**

int shmget(key_t key, size_t size, int shmflg);

➔ **Call shmget() to create a new shared memory segment or obtain the identifier of an existing segment (i.e., one created by another process). This call returns a shared memory identifier for use in later calls.**

➔ **Use shmat() to attach the shared memory segment; that is, make the segment part of the virtual memory of the calling process.**

➔ **Call shmdt() to detach the shared memory segment.**

➔ **Call shmctl() to delete the shared memory segment.**

➔ **A simple example using shared memory :**

1. write_shm.c

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
main()
{
key_t key=1235;
int shm_id;
void *shm;
char *message = "hello";
shm_id = shmget(key,10*sizeof(char),IPC_CREAT);
shm = shmat(shm_id,NULL,NULL);
sprintf(shm,"%s",message);
}
```

2. read_shm.c

```
#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>


main()

 {

key_t key=1235;

int shm_id;

void *shm;

char *message;

message = malloc(10*sizeof(char));

shm_id = shmget(key,10*sizeof(char),NULL);

shm = shmat(shm_id,NULL,NULL);

if(shm == NULL)

{

printf("error");

}

sscanf(shm,"%s",message);

printf("\n message = %s\n",message);

}
```