UNIT-5: Inter Process Communication (IPC)

Introduction to Inter Process Communication

Pipes

Introduction:

- There are two types of pipes for two-way communication: anonymous pipes and named pipes.
- Anonymous pipes enable related processes to transfer information to each other. Typically, an anonymous pipe is used for redirecting the standard input or output of a child process so that it can exchange data with its parent process.
- To exchange data in both directions (duplex operation), you must create two anonymous pipes. The parent process writes data to one pipe using its write handle, while the child process reads the data from that pipe using its read handle. Similarly, the child process writes data to the other pipe and the parent process reads from it.
- Anonymous pipes cannot be used over a network, nor can they be used between unrelated processes.

Properties of Pipe:

1) Pipes do not have a name. For this reason, the processes must share a parent process. This is the main drawback to pipes. However, pipes are treated as file descriptors, so the pipes remain open even after fork and exec.

2) Pipes do not distinguish between messages; they just read a fixed number of bytes. Newline (\n) can be used to separate messages. A structure with a length field can be used for message containing binary data.

3) Pipes can also be used to get the output of a command or to provide input to a command.

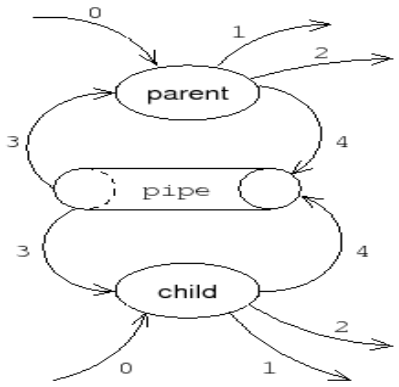
Creation of Pipes

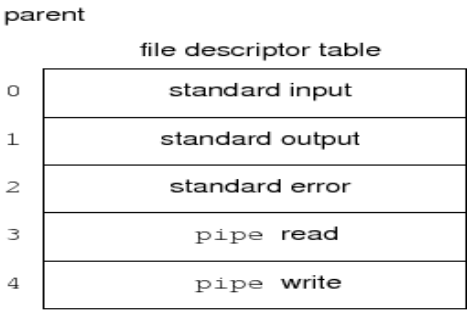
Since A pipe provides a one-way flow of data.

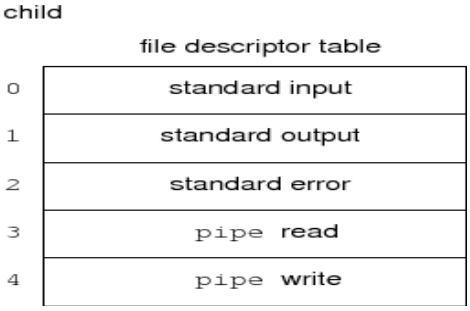
int pipe (int * *filedes*);

int pipefd[2]; /* pipefd[0] is opened for reading; pipefd[1] is opened for writing */

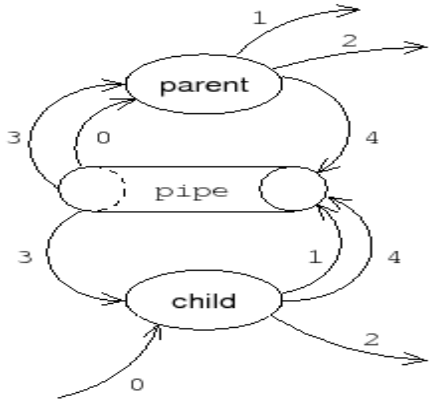
- after call to `fork`







- after both calls to `dup2`



parent

file descriptor table

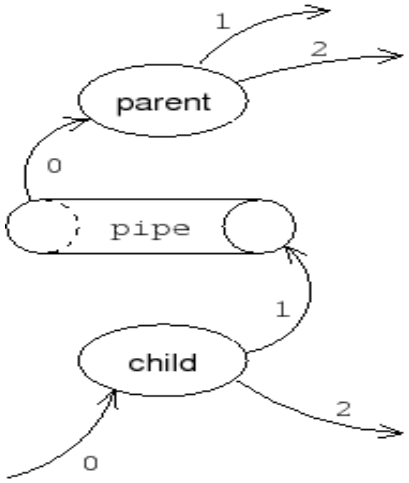
0	pipe read
1	standard output
2	standard error
3	pipe read
4	pipe write

child

file descriptor table

0	standard input
1	pipe write
2	standard error
3	pipe read
4	pipe write

- after all calls to `close`



parent

file descriptor table

0	pipe read
1	standard output
2	standard error

child

file descriptor table

0	standard input
1	pipe write
2	standard error

- If the parent wants to receive data from the child, it should close fd1, and the child should close fd0.
- If the parent wants to send data to the child, it should close fd0, and the child should close fd1. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with. On a technical note, the EOF will never be returned if the unnecessary ends of the pipe are not explicitly closed.

Example:

Example to show how to create and use a pipe:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <stdlib.h>
int main(void)
{
    int fd[2], nbytes;
    pid_t childpid;
    char string[] = "Hello, world!\n";
    char readbuffer[80];
    pipe(fd);
    if((childpid = fork()) == -1)
    {
       perror("fork");
       exit(1);
    }

    if(childpid == 0)
    {
       /* Child process closes up input side of pipe */
       close(fd[0]);
       /* Send "string" through the output side of pipe */
       write(fd[1], string, (strlen(string)+1));
       exit(0);
    }
    else
    {
       /* Parent process closes up output side of pipe */
       close(fd[1]);
       /* Read in a string from the pipe */
       nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
       printf("Received string: %s", readbuffer);
    }
    return(0);
}
```

Execution of Program & Result:

result:

hello world

Named Pipes

Introduction

- Named pipes allow two unrelated processes to communicate with each other. They are also known as FIFOs (first-in, first-out) and can be used to establish a one-way (half-duplex) flow of data.
- Named pipes are identified by their access point, which is basically in a file kept on the file system. Because named pipes have the pathname of a file associated with them, it is possible for unrelated processes to communicate with each other; in other words, two unrelated processes can open the file associated with the named pipe and begin communication.
- Unlike anonymous pipes, which are process-persistent objects, named pipes are file system-persistent objects, that is, they exist beyond the life of the process. They have to be explicitly deleted by one of the processes by calling "unlink" or else deleted from the file system via the command line.
- In order to communicate by means of a named pipe, the processes have to open the file associated with the named pipe. By opening the file for reading, the process has access to the reading end of the pipe, and by opening the file for writing, the process has access to the writing end of the pipe.
- A named pipe supports blocked read and write operations by default: if a process opens the file for reading, it is blocked until another process opens the file for writing, and vice versa. However, it is possible to make named pipes support non-blocking operations by specifying the O_NONBLOCK flag while opening them. A named pipe must be opened either read-only or write-only. It must not be opened for read-write because it is half-duplex, that is, a one-way channel.
- Shells make extensive use of pipes; for example, we use pipes to send the output of one command as the input of the other command. In real-life UNIX® applications, named pipes are used for communication, when the two processes need a simple method for synchronous communication.

Creating a Named Pipe

A named pipe can be created in two ways -- via the command line or from within a program.

From the Command Line

A named pipe may be created from the shell command line. For this one can use either the "mknod" or "mkfifo" commands.

Example:

To create a named pipe with the file named "npipe" you can use one of the following commands:

% mknod npipe p

or

% mkfifo npipe

You can also provide an absolute path of the named pipe to be created.

Now if you look at the file using "ls -l", you will see the following output:

prw-rw-r-- 1 secf other 0 Jun 6 17:35 npipe

The 'p' on the first column denotes that 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.

Within a Program

- The function "mkfifo" can be used to create a named pipe from within a program. The signature of the function is as follows:

  int mkfifo(const char *path, mode_t mode)

- The mkfifo function takes the path of the file and the mode (permissions) with which the file should be created. It creates the new named pipe file as specified by the path.

- The function call assumes the O_CREATE|O_EXCL flags, that is, it creates a new named pipe or returns an error of EEXIST if the named pipe already exists. The named pipe's owner ID is set to the process' effective user ID, and its group ID is set to the process' effective group ID, or if the S_ISGID bit is set in the parent directory, the group ID of the named pipe is inherited from the parent directory.

Opening a Named Pipe

- A named pipe can be opened for reading or writing, and it is handled just like any other normal file in the system. For example, a named pipe can be opened by using the open() system call, or by using the fopen() standard C library function.

- As with normal files, if the call succeeds, you will get a file descriptor in the case of open(), or a 'FILE' structure pointer in the case of fopen(), which you may use either for reading or for writing, depending on the parameters passed to open() or to fopen().

- Therefore, from a user's point of view, once you have created the named pipe, you can treat it as a file so far as the operations for opening, reading, writing, and deleting are concerned.

Reading From and Writing to a Named Pipe

- Reading from and writing to a named pipe are very similar to reading and writing from or to a normal file. The standard C library function calls read( ) and write( ) can be used for reading from and writing to a named pipe. These operations are blocking, by default.

The following points need to be kept in mind while doing read/writes to a named pipe:

- A named pipe cannot be opened for both reading and writing. The process opening it must choose either read mode or write mode. The pipe opened in one mode will remain in that mode until it is closed.
- Read and write operations to a named pipe are blocking, by default. Therefore if a process reads from a named pipe and if the pipe does not have data in it, the reading process will be blocked. Similarly if a process tries to write to a named pipe that has no reader, the writing process gets blocked, until another process opens the named pipe for reading. This, of course, can be overridden by specifying the O_NONBLOCK flag while opening the named pipe.

Seek operations (via the Standard C library function lseek) cannot be performed on named pipes.

Full-Duplex Communication Using Named Pipes

- Although named pipes give a half-duplex (one-way) flow of data, you can establish full-duplex communication by using two different named pipes, so each named pipe provides the flow of data in one direction. However, you have to be very careful about the order in which these pipes are opened in the client and server, otherwise a deadlock may occur.

  For example, let us say you create the following named pipes:

  NP1 and NP2

- In order to establish a full-duplex channel, here is how the server and the client should treat these two named pipes:
- Let us assume that the server opens the named pipe NP1 for reading and the second pipe NP2 for writing. Then in order to ensure that this works correctly, the client must open the first named pipe NP1 for writing and the second named pipe NP2 for reading. This way a full-duplex channel can be established between the two processes.
- Failure to observe the above-mentioned sequence may result in a deadlock situation.

Benefits of Named Pipes

- Named pipes are very simple to use.
- mkfifo is a thread-safe function.
- No synchronization mechanism is needed when using named pipes.
- Write (using write function call) to a named pipe is guaranteed to be atomic. It is atomic even if the named pipe is opened in non-blocking mode.

- Named pipes have permissions (read and write) associated with them, unlike anonymous pipes. These permissions can be used to enforce secure communication.

Limitations of Named Pipes

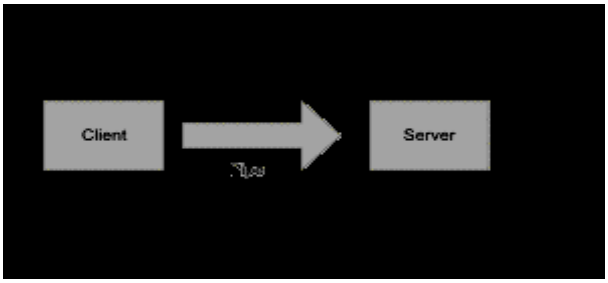
- Named pipes can only be used for communication among processes on the same host machine.
- Named pipes can be created only in the local file system of the host, that is, you cannot create a named pipe on the NFS file system.
- Due to their basic blocking nature of pipes, careful programming is required for the client and server, in order to avoid deadlocks.
- Named pipe data is a byte stream, and no record identification exists.

Code Samples

- The code samples given here were compiled using the GNU C compiler version 3.0.3 and were run and tested on a SPARC processor-based Sun Ultra 10 workstation running the Solaris 8 Operating Environment.
- The following code samples illustrate half-duplex and full-duplex communication between two unrelated processes by using named pipes.

Example of Half-Duplex Communication

- In the following example, a client and server use named pipes for one-way communication. The server creates a named pipe, opens it for reading and waits for input on the read end of the pipe. Named-pipe reads are blocking by default, so the server waits for the client to send some request on the pipe. Once data becomes available, it converts the string to upper case and prints via STDOUT.
- The client opens the same named pipe in write mode and writes a user-specified string to the pipe (see Figure 1).



- The following table shows the contents of the header file used by both the client and server. It contains the definition of the named pipe that is used to communicate between the client and the server.

Filename : half_duplex.h

```
#define HALF_DUPLEX "/tmp/halfduplex"
#define MAX_BUF_SIZE 255
```

Server Code

The following table shows the contents of Filename : hd_server.c.

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "half_duplex.h" /* For name of the named-pipe */
#include <stdlib.h>
int main(int argc, char *argv[])
{
int fd, ret_val, count, numread;
char buf[MAX_BUF_SIZE];
/* Create the named - pipe */

ret_val = mkfifo(HALF_DUPLEX, 0666);
if ((ret_val == -1) && (errno != EEXIST)) {
perror("Error creating the named pipe\n");
exit (1);
}
/* Open the pipe for reading */
fd = open(HALF_DUPLEX, O_RDONLY);
/* Read from the pipe */
numread = read(fd, buf, MAX_BUF_SIZE);
buf[numread] = '\0';
printf("Half Duplex Server: Read From the pipe : %s\n",buf);
/* Convert to the string to upper case */
count = 0;
while (count < numread) {
buf[count] = toupper(buf[count]);
count++;
}
printf("Half Duplex Server: Converted String : %s\n", buf);
}
```

Client Code

The following table shows the contents of Filename : hd_client.c

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

```
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include "half_duplex.h" /* For name of the named-pipe */
#include <stdlib.h>
int main(int argc, char *argv[])
{
int fd;
/* Check if an argument was specified. */

if (argc != 2) {
printf("Usage: %s<string to be sent to the server>n",argv[0]);
exit (1);
}
/* Open the pipe for writing */
fd = open(HALF_DUPLEX, O_WRONLY);
/* Write to the pipe */
write(fd, argv[1], strlen(argv[1]));
}
```

Running the Client and the Server

When you run the server, it will block on the read call and will wait until the client writes something to the named pipe. After that it will print what it read from the pipe, convert the string to upper case, and then terminate. In a typical implementation this server will be either an iterative or a concurrent server. But for simplicity and to demonstrate the communication through the named pipe, we have kept the server code very simple. When you run the client, you will need to give a string as an argument.

First compile the programs and save the executable file as hd_server and hd_client respectively. Make sure you run the server first, so that the named pipe gets created.

Expected output:

1. Run the server: % hd_server &
   The server program will block here, and the shell will return control to the command line.
2. Run the client: % hd_client hello
3. The server prints the string read and terminates:

   Half Duplex Server : Read From the pipe : hello

   Half Duplex Server : Converted String : HELLO

Message Queue

Introduction

A message queue works kind of like a FIFO, but supports some additional functionality. Generally messages are taken off the queue in the order they are put on. Specifically, however, there are ways to pull certain messages out of the queue before they reach the front.

In terms of usage, a process can create a new message queue, or it can connect to an existing one. In this, the latter, way two processes can exchange information through the same message queue.

One more thing about System V IPC: when you create a message queue, it doesn't go away until you destroy it. All the processes that have ever used it can quit, but the queue will still exist. A good practice is to use the *ipcs* command to check if any of your unused message queues are just floating around out there. You can destroy them with the *ipcrm* command.

Creating a message queue:

First of all, you want to connect to a queue, or create it if it doesn't exist. The call to accomplish this is the ***msgget()*** system call:

```
int msgget(key_t key, int msgflg);
```

msgget() returns the message queue ID on success, or −1 on failure (and it sets errno, of course.).

The arguments are a little weird, but can be understood with a little brow−beating. The first, *key* is a system−wide unique identifier describing the queue you want to connect to (or create). Every other process that wants to connect to this queue will have to use the same *key*.

The other argument, *msgflg* tells msgget() what to do with queue in question. To create a queue, this field must be set equal to IPC_CREAT bit−wise OR'd with the permissions for this queue. (The queue permissions are the same as standard file permissions−−queues take on the user−id and group−id of the program that created them.)

Well, since the type key_t is actually just a long, you can use any number you want. But what if you hard−code the number and some other unrelated program hardcodes the same number but wants another queue? The solution is to use the ftok() function which generates a key from two arguments:

```
key_t ftok(const char *path, int id);
```

Basically, *path* just has to be a file that this process can read. The other argument, *id* is usually just set to some arbitrary char, like 'A'. The ftok() function uses information about the named file (like inode number, etc.) and the *id* to generate a probably−unique *key* for msgget(). Programs that want to use the same queue must generate the same *key*, so they must pass the same parameters to ftok().

Finally, it's time to make the call:

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

key = ftok("/home/beej/somefile", 'b');

msqid = msgget(key, 0666 | IPC_CREAT);
```

Sending to the queue

Once you've connected to the message queue using msgget(), you are ready to send and receive messages.

First, the sending:

Each message is made up of two parts, which are defined in the template structure struct msgbuf, as defined in sys/msg.h:

```
struct msgbuf {

long mtype;

char mtext[1];

};
```

The field mtype is used later when retrieving messages from the queue, and can be set to any positive number. mtext is the data this will be added to the queue.

Syntax: `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);`

msqid is the message queue identifier returned by msgget(). The pointer *msgp* is a pointer to the data you want to put on the queue. *msgsz* is the size in bytes of the data to add to the queue. Finally, *msgflg* allows you to set some optional flag parameters.

Receiving from the queue

`int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);`

msqid is the message queue identifier returned by msgget(). The pointer *msgp* is a pointer to the data you want to read from the queue. *msgsz* is the size in bytes of the data from the queue. *msgtyp* is the type of the message to be retrieved from the queue. Finally, *msgflg* allows you to set some optional flag parameters.

Actually, the behavior of msgrcv() can be modified drastically by choosing a *msgtyp* that is positive, negative, or zero:

Zero	Retrieve the next message on the queue, regardless of its `mtype`.
Positive	Get the next message with an `mtype` **equal to** the specified *`msgtyp`*.
Negative	Retrieve the first message on the queue whose `mtype` field is less than or equal to the absolute value of the *`msgtyp`* argument.

Destroying a message queue

There comes a time when you have to destroy a message queue. Like before, they will stick around until you explicitly remove them. There are two ways:

1. Use the Unix command *ipcs* to get a list of defined message queues, then use the command *ipcrm* to delete the queue.

2. Write a program to do it for you.

Often, the latter choice is the most appropriate, since you might want your program to clean up the queue at some time or another. To do this requires the introduction of another function: msgctl().

The synopsis of msgctl() is:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Of course, *msqid* is the queue identifier obtained from msgget(). The important argument is **cmd** which tells msgctl() how to behave. It can be a variety of things, but we're only going to talk about IPC_RMID, which is used to remove the message queue. The *buf* argument can be set to NULL for the purposes of IPC_RMID.

A program that submits messages to a message queue - server

The following program shows the contents of `Filename : mg_server.c`.

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <string.h>
#include <stdio.h>
int main (void) {
     key_t ipckey;
     int mq_id;
     struct { long type; char text[100]; } mymsg;
     /* Generate the ipc key */
     ipckey = ftok("/tmp/foo", 42);
     printf("My key is %d\n", ipckey);
     /* Set up the message queue */
     mq_id = msgget(ipckey, IPC_CREAT | 0666);
     printf("Message identifier is %d\n", mq_id);
     /* Send a message */
```

```
     memset(mymsg.text, 0, 100); /* Clear out the space */
     strcpy(mymsg.text, "Hello, world!");
     mymsg.type = 1;
     msgsnd(mq_id, &mymsg, sizeof(mymsg), 0);
}
```

The code in Listing 2 includes the necessary header files and then defines the variables to be used within the main function. The first order of business is to determine the IPC key using */tmp/foo* as the common file and the number 42 as the ID. For demonstration purposes, this key is displayed on the screen using printf(3c). Next, the message queue is created using *msgget*. The first parameter to *msgget* is the IPC key, and the second is a set of flags. In the example, the flags include the octal permissions, which allow anyone with the IPC key to fully use this IPC, and the IPC_CREAT flag, which causes *msgget* to create the queue. Again, the result is printed to the screen.

Sending the message to the queue is simple. After zeroing out the memory space in the message, a familiar string is copied to the text part of the buffer. The message type is set to 1, and then *msgsnd* is called. *msgsnd* expects to be passed the queue ID, a pointer to the data, the size of the data, and a flag indicating whether the call should block or not. If the flag is IPC_NOWAIT, the call returns even if the queue is full. If the flag is 0, the call blocks until space is free on the queue, the queue is deleted, or the application receives a signal.

The client side of the equation is similar. Listing 3 shows code that retrieves the message that the server sent.

Program Code to retrieve a message from a queue- client

The following program shows the contents of `Filename : mg_client.c`.

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <string.h>
#include <stdio.h>
int main (void)
{
     key_t ipckey;
     int mq_id;
     struct { long type; char text[100]; } mymsg;
     int received;
     /* Generate the ipc key */
```

```
        ipckey = ftok("/tmp/foo", 42);
        printf("My key is %d\n", ipckey);
        /* Set up the message queue */
        mq_id = msgget(ipckey, 0);
        printf("Message identifier is %d\n", mq_id);
        received = msgrcv(mq_id, &mymsg, sizeof(mymsg), 0, 0);
        printf("%s (%d)\n", mymsg.text, received);
}
```

The procedure to get the IPC key and the message queue identifier is similar to the server's code. The call to msgget does not specify any flags, because the server has already created the queue. If the application were designed such that the client might be started before the server, then both the client and server would have to specify permissions and the IPC_CREAT flag so that whichever application started first would create the queue.

mq_client.c then calls msgrcv to pull a message off the queue. The first three arguments specify the message queue identifier, a pointer to the memory space that will hold the message, and the size of the buffer. The fourth parameter is the type parameter, and it allows you to be selective about which messages you get:

If the type is 0, the first message in the queue is returned.

If the type is a positive integer, the first message in the queue with the same type is returned.

If the type is a negative integer, the first message in the queue with the lowest value that is less than or equal to the absolute value of the specified type is returned. For example, if 2 and then 1 were to be added to the queue, callingmsgrcv with a type of -2 would return the 1 because it is the smallest, even though it was second in the queue.

The fifth parameter to msgrcv is the blocking flag again. Listing 4 shows the client and server in action.

Execution and result:

sunbox$ ./mq_server

My key is 704654099

Message identifier is 2

sunbox$ ./mq_client

My key is 704654099

Message identifier is 2

Hello, world! (104)

The output of the client and server show that they both came up with the same IPC key because they referenced the same file and identifier. The server created the IPC instance, which the kernel assigned the value of 2, and the client application learned this. It should then come as no surprise that the client pulls "Hello, world!" back from the message queue.

This example shows the simplest of situations. A message queue would be helpful for a short-lived process such as a Web transaction that submits work to a heavier back-end application such as some batch processing. A client can also be a server, and multiple applications can submit messages to the queue. The message type field allows applications to direct messages to particular readers.