Inter-process communication (IPC)

Outline

Inter-Process communication (IPC)

Shared Memory Segments

- Introduction
- Pros and Cons
- How to create and use

Named Pipes

- What is named pipes and how to create
- Half-Duplex Named Pipes
- Full-Duplex named pipes

Inter-Process Communication (IPC)

IPC is a set of methods for the exchange of data among multiple threads or processes.

Why do we need IPC?

- Information sharing
- Computational Speedup
  - When parallelizing programs
- Modularity
- Convenience
- Privilege separation
  - Ability to provide access control

- **PIPE**: Only two related (eg: parent & child) processess can be communicated. Data reading would be first in first out manner.
- **Named PIPE or FIFO** : Only two processes (can be related or unrelated) can communicate. Data read from FIFO is first in first out manner.
- **Message Queues**: Any number of processes can read/write from/to the queue.
- **Shared Memory**: Part of process's memory is shared to other processes. other processes can read or write into this shared memory area based on the permissions. Accessing Shared memory is faster than any other IPC mechanism as this does not involve any kernel level switching(Shared memory resides on user memory area).
- **Semaphore**: Semaphores are used for process synchronization. This can't be used for bulk data transfer between processes.

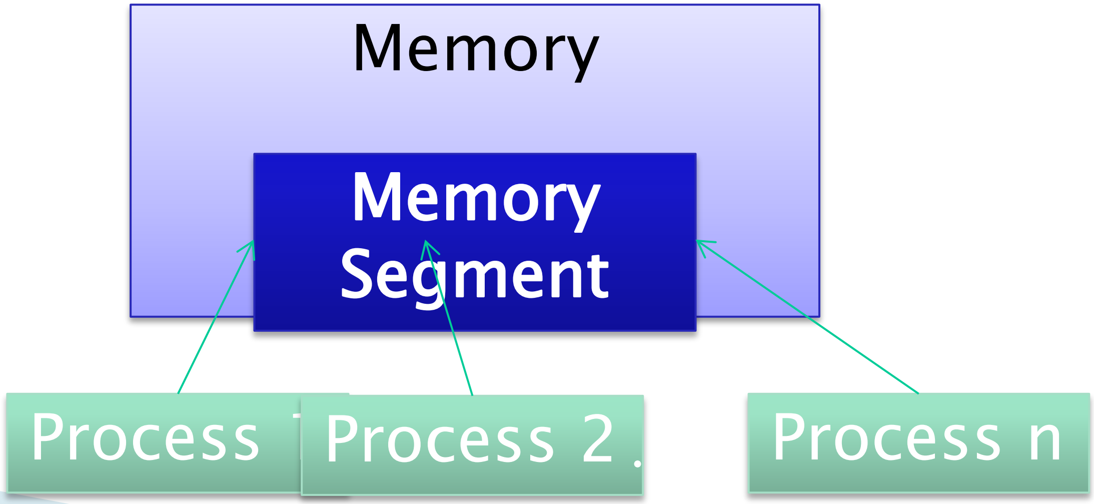
Methods of IPC

- **Shared memory**
- **Named Pipes**
- Message passing (MPI)
- Synchronization
- Remote procedure calls (RPC)
- Etc.

Shared Memory Segment

What is shared memory?

Shared memory (SHM) is one method of inter-process communication (IPC) whereby 2 or more processes share a single chunk of memory to communicate.



Steps of Shared Memory IPC

1. Creating the segment and connecting
2. Getting a pointer to the segment
3. Reading and Writing
4. Detaching from and deleting segments

1. Creating the segment and connecting

System V IPC is used in these examples

A shared memory segment is ‘created’ and ‘connected to’ via the **shmget()** call

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

The *key* argument should be created using **ftok()**.

The *size*, is the size in bytes of the shared memory segment.

The *shmflg* should be set to the permissions of the segment bitwise-ORd with IPC_CREAT if you want to create the segment, but can be 0 otherwise.

Upon successful completion, **shmget()** returns an identifier for the shared memory segment.

1. Creating the segment and connecting Cont.

Here's an example call that creates a 1K segment with **644 permissions** (rw-r--r--)

```
key_t key;
int shmid;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

A **Sticky bit** is a **permission bit** that is set on a file or a directory that lets only the owner of the file/directory or the root user to delete or rename the file.

2. Getting a pointer to the segment

The shared memory segment must be attached using **shmat()** before its used.

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

shmid is the shared memory ID from the **shmget()** call.

shmaddr, which you can use to tell **shmat()** which specific address to use. When set it to 0, the OS will decide the address.

shmflg can be set to SHM_RDONLY if you only want to read from it, 0 otherwise.

2. Getting a pointer to the segment Cont.

Here's a more complete example of how to get a pointer to a shared memory segment:

```
key_t key;
int shmid;
char *data;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

3. Reading and Writing

The *data* pointer from the above example is a char pointer. Thus it reads chars from it.

lets say the 1K shared memory segment contains a null-terminated string.

It can be printed like this:

```
printf("shared contents: %s\n", data);
```

And we could store something in it as easily as this:

```
printf("Enter a string: ");
gets(data);
```

4. Detaching from and deleting segments

When you're done with the shared memory segment, your program should detach itself from it using the **shmdt()** call:

```
int shmdt(void *shmaddr);
```

The only argument, *shmaddr*, is the address you got from **shmat()**.

The function returns -1 on error, 0 on success.

4. Detaching from and deleting segments Cont.

Remember! When you detach from the segment, it isn't destroyed. Nor is it removed when *everyone* detaches from it.

You have to specifically destroy it using a call to **shmctl()**

```
shmctl(shmid, IPC_RMID, NULL);
```

The above call deletes the shared memory segment, assuming no one else is attached to it.

Code Example

As always, you can destroy the shared memory segment from the command line using the **ipcrm** Unix command.

```
ipcrm [-m shmid]
```

Also, be sure that you don't leave any unused shared memory segments sitting around wasting system resources.

All the System V IPC objects you own can be viewed using the **ipcs** command.

Shared Memory, Pros and Cons

Pros

- Fast bidirectional communication among any number of processes
- Saves Resources

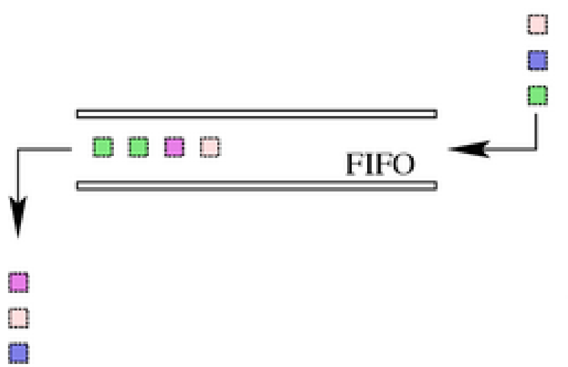
Cons

- Needs concurrency control (leads to data inconsistencies like ‘Lost update’)
- Lack of data protection from Operating System (OS)

Named Pipes

Named pipes also allow two unrelated processes to communicate with each other.

They are also known as FIFOs (first-in, first-out)

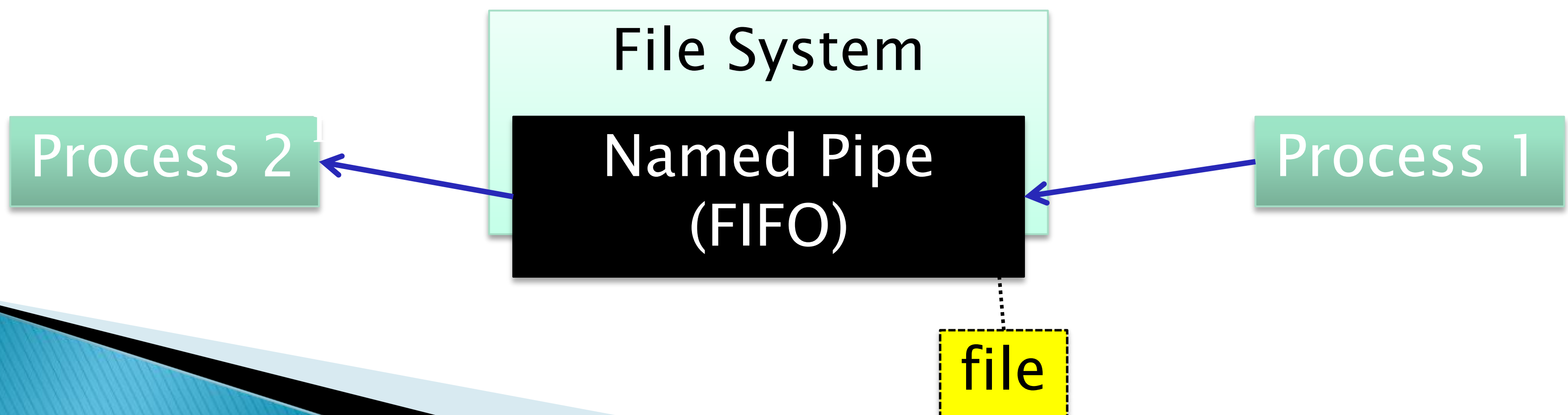


Used to establish a one-way (**half-duplex**) flow of data.

Named Pipes Cont.

How does it work?

- Uses an **access point (A file on the file system)**
- Two unrelated processes opens this file to communicate
- One process writes and the other reads, thus it is a half-duplex communication



Named Pipes – Properties

Named pipes are **system-persistent objects.**

- i.e. They exist beyond the life of the process
- In contrary, anonymous pipes are process-persistent objects

Named Pipes must be explicitly deleted by one of the process by calling "unlink"

By default, it supports **blocked** read and writes.

- i.e. if a process opens the file for reading, it is blocked until another process opens the file for writing, and vice versa.
- This can be overridden by calling O_NONBLOCK flag when opening them.

A named pipe must be opened either **read-only or write-only** because it is **half-duplex**, that is, a one-way channel

1. Creating a Named Pipe

The function "mkfifo" can be used to create a named pipe

```
int mkfifo(const char *path, mode_t mode)
```

It creates the new named pipe file as specified by the path.

mode parameter used to set the permissions of the file.

This function creates a new named pipe or returns an error of EEXIST if the named pipe already exists.

2. Opening Named Pipe

- A named pipe can be opened for reading or writing

- It is handled just like any other normal file in the system.

- Ex: a named pipe can be opened by using the `open()` system call, or by using the `fopen()` standard C library function.

- 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.

3. 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.

3. Reading From and Writing to a Named Pipe Cont.

- 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.

3. Reading From and Writing to a Named Pipe Cont.

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.

Code Example of Half-Duplex Communication

How to obtain Full-Duplex using Named Pipes?

A full-duplex communication can be established using different named pipes so each named pipe provides the flow of data in one direction.

Care should be taken to avoid deadlock.

An Example

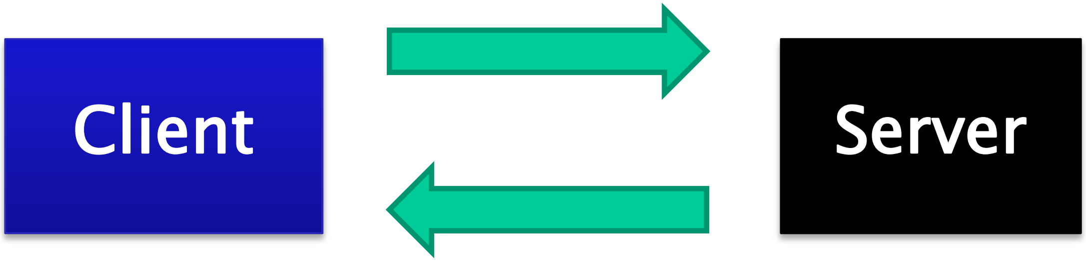
Let us assume that the **server** opens the named pipe NP1 for reading and the second pipe NP2 for writing.

Then to ensure this works correctly, the **client** must open the NP1 for writing and the 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.

Code Example of Full-Duplex Communication



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, (i.e. cannot create a named pipe on the NFS file system.)
- 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.

References

Shared Memory Under Linux by Mike Perry

http://fscked.org/writings/SHM/

Beej's Guide to Unix IPC by Brian "Beej Jorgensen" Hall

Section 9. Shared Memory Segments: http://beej.us/guide/bgipc/output/html/multipage/shm.html

Introduction to Interprocess Communication Using Named Pipes by Faisal Faruqui, July 2002 at Oracle

http://developers.sun.com/solaris/articles/named_pipes.html

Message Queues

Unix IPC Package

- Unix System V IPC package consists of three things
  - **Messages** – allows processes to send formatted data streams to arbitrary processes
  - **Shared memory** allows processes to share parts of their virtual address space.
  - **Semaphores** allow processes to synchronise execution.

Message Queues

- A message queue works like a FIFO.
- A process can create a **new message queue**, or it can connect to an existing one.
- When you create a message queue, it doesn't go away until you destroy it.
  - Can use the *ipcs* command to check if any of the unused message queues are just floating around.
  - Can destroy them with the *ipcrm* command

- To connect to a queue
  - **int msgget(key_t key, int msgflg);**
  - msgget() returns the message queue ID on success, or -1 on failure
- Arguments
  - *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.
  - *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.

- type *key_t* is actually just a long, you can use any number you want.
- use the ftok() function which generates a key from two arguments:
  - key_t ftok(const char *path, int id);
  - *path* is the file that this process can read, *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().

To make the call:

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

key = ftok("/home/abc/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

Sending to the Queue

- 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];
 };
```

- ***mtype*** is used later when retrieving messages from the queue, set to any positive number. ***mtext*** is the data this will be added to the queue.

- You can use any structure you want to put messages on the queue, as long as the first element is a long.

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    char name[30];
    char ship_type;
    int notoriety;
    int cruelty;
    int booty_value;
};
```

- **To send msg**

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

- **Arguments**
  - msqid is the message queue identifier returned by msgget().
  - 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.
  - msgflg allows you to set some optional flag parameters, which we'll ignore for now by setting it to 0.

Sending to the Queue

```
key_t key;
int msqid;
struct pirate_msgbuf  pmb = {2, "L'Olonais", 'S', 80, 10, 12035};

key = ftok("/home/csec/writeq.c", 'b');

msqid = msgget(key, 0666 | IPC_CREAT);

msgsnd(msqid, &pmb, sizeof(pmb), 0);
```

Receiving from the Queue

- **int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);**
  - Here msgtyp corresponds to the mtype we set in the msgsnd()
- **msgtyp Effect on msgrcv()**
  - 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.

Receiving from a Queue

```
key_t key;
int msqid;
struct pirate_msgbuf pmb; /* where L'Olonais is to be kept */
key = ftok("/home/abc/somefile", 'b');
msqid = msgget(key, 0666);
msgrcv(msqid, &pmb, sizeof(pmb), 2, 0);
```

Destroying a message queue

- There are two ways of destroying a queue:
  - Use the Unix command *ipcs* to get a list of defined message queues, then use the command
  - *ipcrm* to delete the queue.
  - Write a program to do it for you.

```
//writes to a message queue
//write queue

#include <stdio.h>
   #include <stdlib.h>
   #include <errno.h>
   #include <sys/types.h>
   #include <sys/ipc.h>
   #include <sys/msg.h>


   struct my_msgbuf {
      long mtype;
      char mtext[200];
   };


   int main(void)
   {
      struct my_msgbuf buf;
      int msqid;
      key_t key;


      if ((key = ftok("writeq.c", 'B')) == -1) {
         perror("ftok");
         exit(1);
      }


      if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1)
{
         perror("msgget");
         exit(1);
      }

printf("Enter lines of text, ^D to quit:\n");


   buf.mtype = 1; /* we don't really care in this case */
   while(gets(buf.mtext), !feof(stdin)) {
      if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0) == -1)
         perror("msgsnd");
   }


   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl");
      exit(1);
         }


  return 0;
 }
```

```
//reads from the msg queue

    #include <stdio.h>
    #include <stdlib.h>
    #include <errno.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>

    struct my_msgbuf {
        long mtype;
        char mtext[200];
    };

    int main(void)
    {
        struct my_msgbuf buf;
        int msqid;
        key_t key;

        if ((key = ftok("writeq.c", 'B')) == -1) {  /* same key
as writeq.c */
            perror("ftok");
            exit(1);
        }

        if ((msqid = msgget(key, 0644)) == -1) { /* connect to
the queue */
            perror("msgget");
            exit(1);
        }

printf("Enter lines of text, ^D to quit:\n");
printf("readq: ready to receive messages....\n");

        for(;;) {
if (msgrcv(msqid, (struct msgbuf *)&buf, sizeof(buf), 0, 0) == -1) {
                perror("msgrcv");
                exit(1);
            }
            printf("readq: \"%s\"\n", buf.mtext);
        }

        return 0;
    }
```

- **int msgctl(int msqid, int cmd, struct msqid_ds *buf);**
- Arguments
  - **msqid is the queue identifier obtained from msgget().**
  - **cmd tells msgctl() how to behave. IPC_RMID is used to remove the message queue.**
  - **Buf argument can be set to NULL for the purposes of IPC_RMID.**
- msgctl(msqid, IPC_RMID, NULL);

- **PIPE**: Only two related (eg: parent & child) processess can be communicated. Data reading would be first in first out manner.

- **Named PIPE or FIFO** : Only two processes (can be related or unrelated) can communicate. Data read from FIFO is first in first out manner.

- **Message Queues**: Any number of processes can read/write from/to the queue.

- **Shared Memory**: Part of process's memory is shared to other processes. other processes can read or write into this shared memory area based on the permissions. Accessing Shared memory is faster than any other IPC mechanism as this does not involve any kernel level switching(Shared memory resides on user memory area).

- **Semaphore**: Semaphores are used for process synchronization. This can't be used for bulk data transfer between processes.