# Lecture 6

## Inter Process Communication

Inter Process Communication (IPC) is a set of techniques for establishing communication among multiple processes during runtime by exchanging data among them. IPC techniques consist of using pipes, shared memory, message passing etc.

## Pipes

A pipe is a unidirectional or bidirectional communication channel that enables the transfer of data between two related processes. A pipe has two ends: one for sending data (write end) and one for receiving data (read end). The data written by one process can be read by another process. A pipe is a simple way for two related processes to communicate with each other.

The process that creates the pipe is called the parent process, and the two processes that communicate through the pipe are called the child processes. Pipes are created using the pipe() system call. A pipe returns two file descriptors, one for the read end and one for the write end of the pipe. For unidirectional communication a pipe is required and for bidirectional two pipes are required.

**Creating and closing a pipe:**

```
#include<unistd.h>

int pipe(int pipedes[2]);
```

This system call will create a pipe for one-way communication i.e., it creates two descriptors, the first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor pipedes[0] is for reading/receiving and pipedes[1] is for writing/sending. Whatever is written into pipedes[1] can be read from pipedes[0].

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with perror() function.

```
#include<unistd.h>

int close(int pipedes)
```

The above system call closing already opened file descriptor. This implies the file is no longer in use and resources associated can be reused by any other process. This system call returns 0 on success and -1 in case of error.

**Sending/writing data through pipe:**

```
#include<unistd.h>

ssize_t write(int fd, void *buf, size_t count)
```

The above system call is to write to the specified file with arguments of the file descriptor fd, a proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call.

The file needs to be opened before writing the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes written (or 0 in case nothing is written) on success and -1 in case of failure. Proper error number is set in case of failure.

**Receiving/reading data from pipe:**

```
#include<unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

The above system call is to read from the specified file with arguments of file descriptor fd, proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call. The file needs to be opened before reading the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes read (or 0 in case of encountering the end of the file) on success and -1 in case of failure. The return bytes can be smaller than the number of bytes requested, just in case no data is available, or file is closed. Proper error number is set in case of failure.

For bidirectional communication using pipes it is required to create two pipes. First one is for the parent to write and the child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

# Shared Memory

Separate processes run in separate address spaces. A shared memory segment is a piece of memory that can be allocated and attached to an address space. Thus, processes that have this memory segment attached will have access to it. Procedures of using shared memory for IPC are given below.

To use shared memory these header files must be included:

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

#include <sys/ipc.h>

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

**Generating a key:** Unix uses this key for identifying shared memory segments.

A key is a value of type `key_t`. There are three ways to generate a key:

1. **Explicit:**
   ```
   key_t    SomeKey;
   SomeKey = 1234;
   ```
2. `ftok()` **to generate:**
   ```
   key_t = ftok(char *path, int ID);
   ```
   - `path` is a path name (e.g., "`./`")
   - `ID` is an integer (e.g., '`a`')
   - Function `ftok()` returns a key of type `key_t`:
     ```
     SomeKey = ftok("./", 'x');
     ```
3. **System generated:**
   ```
   IPC_PRIVATE
   ```

Keys are global entities. If other processes know your key, they can access the shared memory.

**Allocating a shared memory:**

```
int shm_id = shmget(key_t key, int size, int flag);
```

Here, first parameter is contains the key of the shared memory, second parameter includes the size of it and the third one includes the necessary flag which indicates whether the shared memory needs to be created or the process is going to access an existing shared memory and permission access to the shared memory to the process. Consider, the shared memory needs to be created then the flag will be `IPC_CREAT | 0666` which means a new shared memory with read, write access to user, group and other will be provided. And if the process is willing to access on an existing memory then the flag will be only `0666` which means that process obtained read and write permissions for user, group and others on an existing shared memory. Upon successful execution

`shmget()` returns a shared memory ID which is a positive integer. At failure in returns negative value.

**Attaching the shared memory to an address space:**

```
void *shm_ptr = shmat(int  shm_id, char *ptr, int  flag);
```

- `shm_id` is the shared memory ID returned by `shmget()`.
- For second and third parameters `NULL` and `0` should be used respectively as we are going to use system assigned address spaces for the shared memory.
- `shmat()` returns a `void` pointer to the memory. If unsuccessful, it returns a negative integer.

After attaching processes can communicate among them using that shared memory.

**Removing the shared memory:**

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

Here, `shm_ID` is the shared memory ID returned by `shmget()` and `IPC_RMID` is the flag responsible for removing the shared memory containing `shm_ID`. After all the tasks are done by all the cooperating processes, the shared memory should be removed. After a shared memory is removed, it no longer exists.

## Message Passing

Communication among processes by message passing is done by creating message queues in the kernel and exchanging messages through these queues. Processes having access of a message queue can communicate among them by reading and writing data using the queue. Procedures of using message queue for IPC are given below.

To use message queue these header files must be included:

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

**Creating or allocating a message queue:**

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

- The first argument, key, recognizes the message queue. The key can be derived explicitly or by using `ftok()` or system generated.
- The second one includes the necessary flag which indicates whether the message queue needs to be created or the process is going to access an existing message queue and permission access to the message queue to the process. For both cases, the flag will be

`IPC_CREAT | 0666` which means with a new message queue with read, write access to user, group and other will be provided. In case an existing queue `IPC_CREAT` will be ignored by the system, permissions will be considered.

- This call would return a valid message queue identifier (used for further calls of message queue) on success and -1 in case of failure.

**The message structure:**

```
struct msgbuf {
   long mtype;
   char mtext[1];
};
```

- The variable `mtype` is used for communicating with different message types which is a positive number (cannot be zero).
- The variable `mtext` is the message body array. Usually, larger than one byte.

**Writing messages in the message queue:**

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

- The first argument, `msgid`, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of `msgget`().
- The second argument, `msgp`, is the pointer to the message sent to the caller, defined in the structure of the form of `msgbuf`.
- The third argument, `msgsz`, is the size of message.
- The fourth argument, `msgflg`, indicates certain flags such as `IPC_NOWAIT` (returns immediately when no message is found in queue or `0` by which the process waits if there are no messages in the queue.
- This call would return 0 on success and -1 in case of failure.

**Reading messages from the queue:**

```
int msgrcv(int msgid, const void *msgp, size_t msgsz,
long msgtype, int msgflg)
```

- The first argument, `msgid`, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of `msgget`().
- The second argument, `msgp`, is the pointer to the message sent to the caller, defined in the structure of the form of `msgbuf`.
- The third argument, `msgsz`, is the size of the message received.
- The fourth argument, `msgtype`, indicates the type of message which should be read:
  - 0: The first message of the queue will be read.
  - A positive integer: Reads the first message in the queue of type `msgtype` (if `msgtype` is 10, then reads only the first message of type 10 even though other types may be in the queue at the beginning).
  - A negative integer: Reads the first message of lowest type less than or equal to the absolute value of message type (say, if `msgtype` is -5, then it reads first message of type less than 5 i.e., message type from 1 to 5).
- The fifth argument, `msgflg`, indicates certain flags such as:
  - `0`: Waits if there are no messages in the queue.
  - `IPC_NOWAIT`: Returns immediately when no message is found in the queue.
  - `MSG_EXCEPT`: If the message type parameter is a positive integer, then return the first message whose type is NOT equal to the given integer.
  - `MSG_NOERROR`: If a message with a text part larger than `msgsz` matches what we want to read, then truncate the text when copying the message to our msgbuf structure. If this flag is not set and the message text is too large, the system call returns '-1', and errno is set to E2BIG.

**Removing the shared memory:**

```
int msgctl(int msgid, IPC_RMID, NULL)
```

Here, `msgid` is the queue ID returned by `msgget`() and `IPC_RMID` is the flag responsible for removing the message queue containing `msgid`. After all the tasks are done by all the cooperating processes, the message queue should be removed. After a message queue is removed, it no longer exists.