Inter Process Communication (IPC)

On Linux

Prayas Mohanty (Red Hat Certified Instructor)

Red Hat Certification ID: 100-005-594

Objective

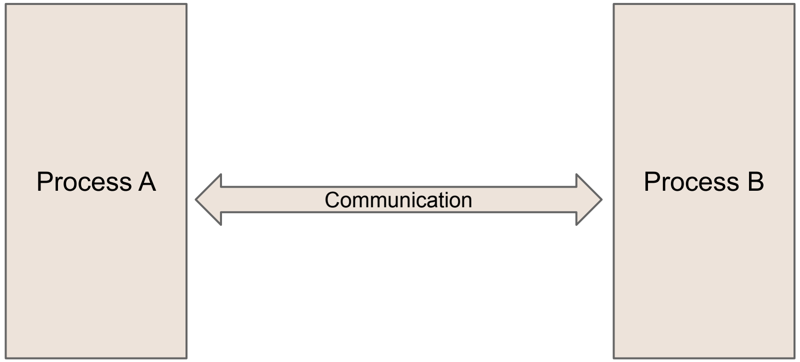
- What is IPC
- What is pipe
- What is fifo
- How to use Message queue
- How to use shared Memory
- How to use Semaphore
- Semaphore Vs Mutex

Prerequisite of Participants

- Having Knowledge on Linux Platform
- Having familiarity with Linux Command line
- Basic Knowledge on vi Editor
- Proper Knowledge on C programming
- Basic Knowledge on Process Handling in C

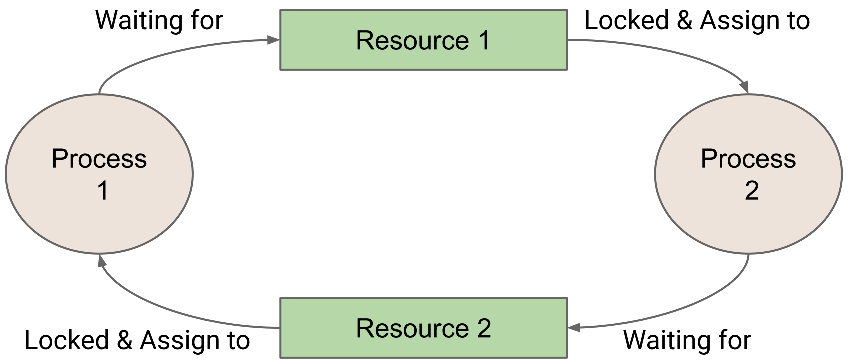
What is IPC

- Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions.
- It refer to mechanisms an operating system provides to allow the processes to manage shared data.
- A common example of this need is managing access to a given system resource. To carry out IPC, some form of active or passive communication is required.

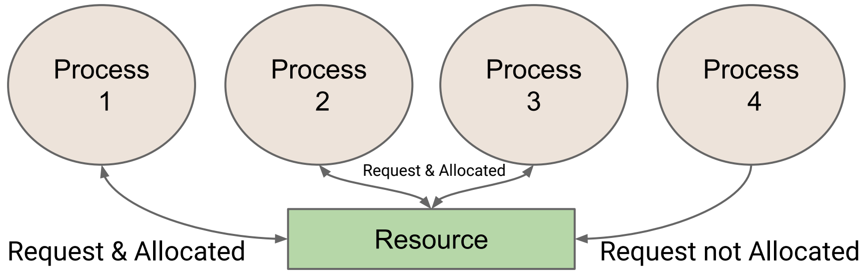


Multithreading pitfalls

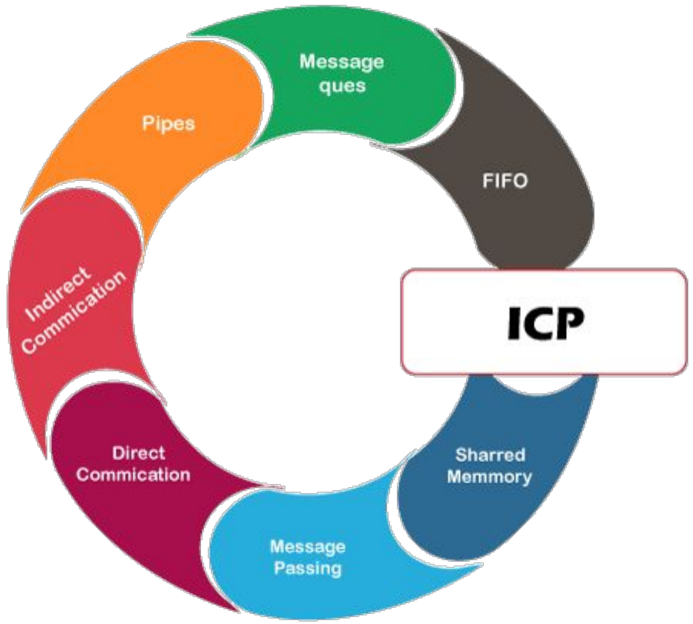
- deadlock :
  - Deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process.



- Starvation:
  - It is the problem that occurs when low priority processes get jammed for an unspecified time as the high priority processes keep executing.



Overview of Linux IPC Mechanisms



- Pipe
- FIFO
- Semaphore
- Message Queue
- Shared Memory

Using Pipe

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";
int main() {
        char inbuf[MSGSIZE];
        int pd[2], i;

        if (pipe(pd) < 0)
                exit(1);
        write(pd[1], msg1, MSGSIZE);
        write(pd[1], msg2, MSGSIZE);
        write(pd[1], msg3, MSGSIZE);

        for (i = 0; i < 3; i++) {
                /* read pipe */
                read(pd[0], inbuf, MSGSIZE);
                printf("% s\n", inbuf);
        }
        return 0;
}
```

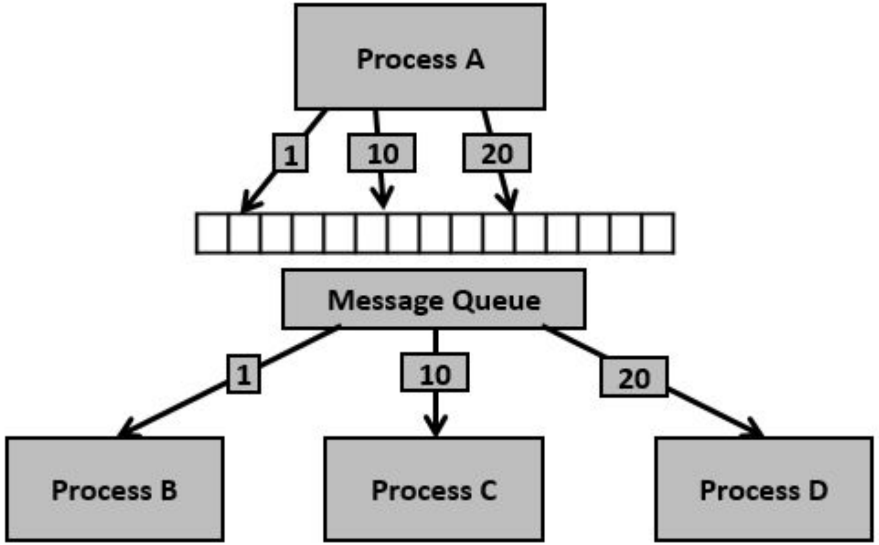
- Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a “virtual file”.
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this “virtual file” or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe system call finds the first two available positions in the process’s open file table and allocates them for the read and

Using FIFO the named pipe

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

int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";
    mkfifo(myfifo, 0666);
    char arr1[80], arr2[80];
    while (1)    {
        fd = open(myfifo, O_WRONLY);
        fgets(arr2, 80, stdin);
        write(fd, arr2, strlen(arr2)+1);
        close(fd);

        fd = open(myfifo, O_RDONLY);
        read(fd, arr1, sizeof(arr1));
        printf("User2: %s\n", arr1);
        close(fd);
    }
}
```

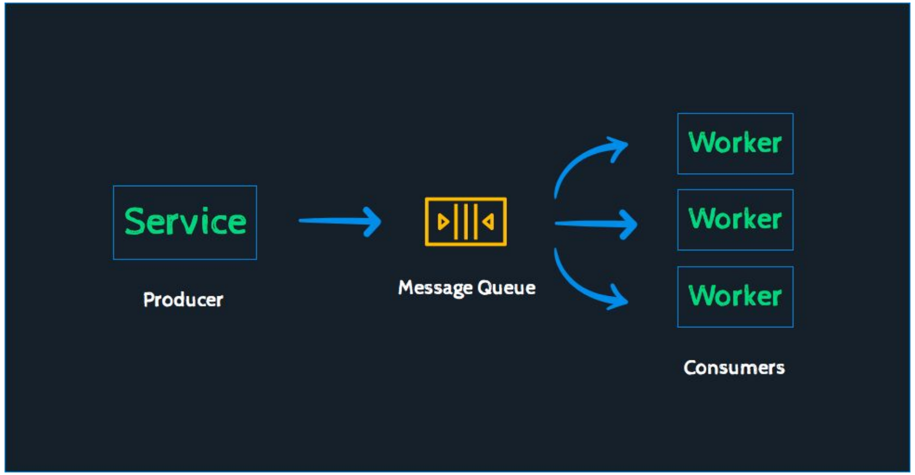
```c
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
       int fd1;
       char * myfifo = "/tmp/myfifo";
       mkfifo(myfifo, 0666);
       char str1[80], str2[80];
       while (1)     {
       fd1 = open(myfifo,O_RDONLY);
       read(fd1, str1, 80);
       printf("User1: %s\n", str1);
       close(fd1);

       fd1 = open(myfifo,O_WRONLY);
       fgets(str2, 80, stdin);
       write(fd1, str2, strlen(str2)+1);
       close(fd1);
       }
}
```

What is Message Queue



- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
- Message queues allow one or more processes to write messages which will be read by one or more reading processes.
- Each message is given an identification or type so that processes can select the appropriate message.
- Process must share a common key in order to gain access to the queue in the first place.

Why message queue?



- The main features of applications that use message queuing techniques are:
  - There are no direct connections between programs.
  - Communication between programs can be independent of time.
  - Work can be carried out by small, self-contained programs.
  - Communication can be driven by events.
  - Applications can assign a priority to a message.
  - High Security.
  - Data integrity.

Message Queue basics

- A new queue is created or an existing queue opened by msgget().
- New messages are added to the end of a queue by msgsnd().
- Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue.
- Messages are fetched from a queue by msgrcv().
  - We don’t have to fetch the messages in a first-in, first-out order.
  - Instead, we can fetch messages based on their type field.

Messageq relevant function calls

- ftok(): is use to generate a unique key.
- msgget(): either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
- msgsnd(): Data is placed on to a message queue by calling msgsnd().
- msgrcv(): messages are retrieved from a queue.
- msgctl(): It performs various operations on a queue. Generally it is use to destroy message queue.

An Application using Message queue.

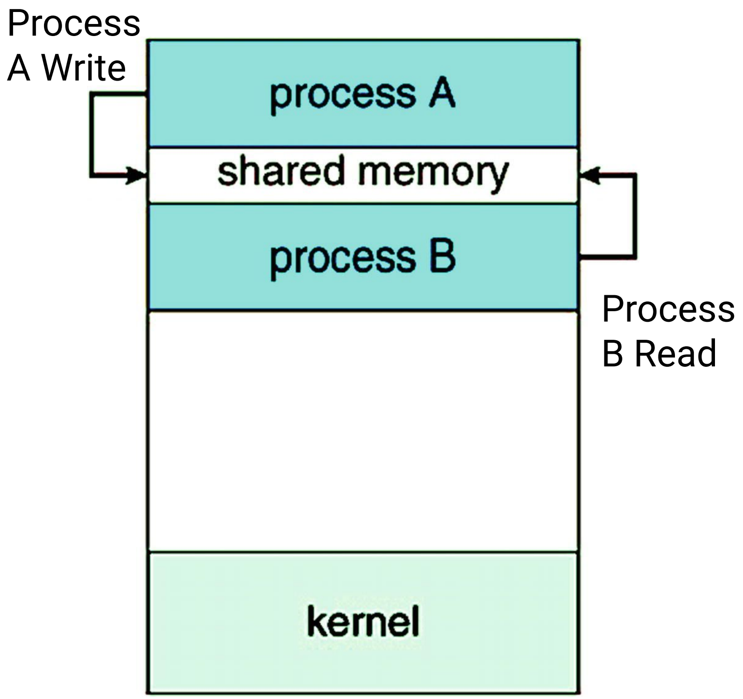
```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10

struct mesg buffer {
       long mesg type;
       char mesg_text[100];
} message;

int main() {
       key t key;
       int msgid;
       key = ftok("progfile", 65);
       msgid = msgget(key, 0666 | IPC_CREAT);
       message.mesg_type = 1;

       printf("Write Data : ");
       fgets(message.mesg text,MAX,stdin);
       msgsnd(msgid, &message, sizeof(message), 0);
       printf("Data send is : %s \n", message.mesg_text);
}
```

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

struct mesg buffer {
       long mesg type;
       char mesg_text[100];
} message;

int main() {
       key t key;
       int msgid;
       key = ftok("progfile", 65);
       msgid = msgget(key, 0666 | IPC_CREAT);

       msgrcv(msgid, &message, sizeof(message), 1, 0);
       printf("Data Received is : %s \n", message.mesg_text);
       msgctl(msgid, IPC_RMID, NULL);
}
```

Message queue specific coding guidelines

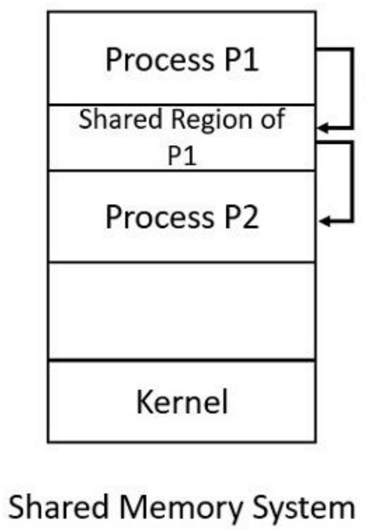
- Handle errors after system call.
- Remove the queue after use.
- Use appropriate mode flags – (O_CREAT, O_EXECL, etc.)

What is shared memory?

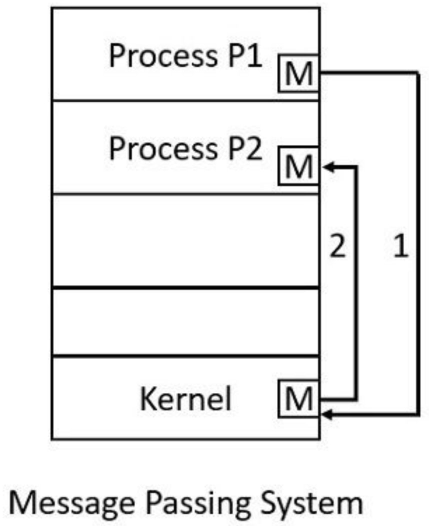


- Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.
- Shared memory is an efficient means of passing data between programs.
- shared memory is a IPC concept where two or more process can access the common memory.
- The communication is done via this shared memory where changes made by one process can be viewed by another process.

Why shared memory?



Shared Memory System



Message Passing System

- The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.
    - Server reads from the input file.
    - The server writes this data in a message using either a pipe, fifo or message queue.
    - The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
    - Finally the data is copied from the client's buffer.
    - A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment.

Share Memory Deployment

- Create the shared memory segment or use an already created shared memory segment (shmget())
- Attach the process to the already created shared memory segment (shmat())
- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())

Share Memory related Function Calls

- ftok(): is use to generate a unique key.
- shmget(): int shmget(key_t,size_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.
- shmat(): Before you can use a shared memory segment, you have to attach yourself to it using shmat(). void *shmat(int shmid ,void *shmaddr ,int shmflg);
  - shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.
- shmdt(): When you’re done with the shared memory segment, your program should detach itself from it using shmdt(). int shmdt(void *shmaddr);
- shmctl(): when you detach from shared memory,it is not destroyed. So, to destroy shmctl() is used. shmctl(int shmid,IPC_RMID,NULL);

Develop an application using Shared Memory

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAX 10

int main()
{
    key t key = ftok("shmfile",65);
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Write Data : ");
    fgets(str,MAX,stdin);

    printf("Data written in memory: %s\n",str);
    shmdt(str);

    return 0;
}
```

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int main()
{
    key t key = ftok("shmfile",65);
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    char *str = (char*) shmat(shmid,(void*)0,0);
    printf("Data read from memory: %s\n",str);

    shmdt(str);
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```

Shared memory specific coding guidelines

- Handle errors after system call.
- Unmap memory after use
- Synchronize access to shared memory if required

What is semaphore?

- a semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system.
- Semaphores are a type of synchronization primitives.
- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and post

Using semaphore function calls

- A semaphore is initialised by using sem_init(for processes or threads) or sem_open (for IPC).
- To lock a semaphore or wait we can use the sem_wait function.
- To release or signal a semaphore, we use the sem_post function.
- To destroy a semaphore, we can use sem_destroy.

Use semaphore calls to updates global variable.

```c
sem t lock;
int balance = 0;
void mypThreadFun(void *targ) {
        int b;
        printf("Hello Balance \n");
        sem wait(&lock);
        b = balance;
        b += 10;
        balance = b;
        sem post(&lock);
        printf("done\n");
}
int main() {
        int i;
        void *myThreadFun = &mypThreadFun;
        pthread t thread id[200];
        sem init(&lock,0,1);
        printf("Balance Before Thread: %d\n", balance);
        for (i=0; i<200; i++) {
            pthread_create(&thread_id[i], NULL, myThreadFun, NULL); }
        for (i=0; i<200; i++){pthread_join(thread_id[i], NULL); }
        sem destroy(&lock);
        printf("Balance After Thread: %d\n", balance);
        exit(0);
}
```

- initialize a semaphore Variable
    - sem_t lock;
- initialize a semaphore
    - sem_init(&lock,0,1);
- Lock semaphore before Critical Section
    - sem_wait(&lock);
- Unlock semaphore after Critical Section
    - sem_post(&lock);
- Destroy semaphore to release memory
    - sem_destroy(&lock);

Semaphore vs. mutex

- Mutex is a locking mechanism whereas Semaphore is a signaling mechanism
- Mutex is just an object while Semaphore is an integer
- Mutex has no subtype whereas Semaphore has two types, which are counting semaphore and binary semaphore.
- Semaphore supports wait and signal operations modification, whereas Mutex is only modified by the process that may request or release a resource.
- Semaphore value is modified using wait () and signal () operations, on the other hand, Mutex operations are locked or unlocked.

Summary

- What is IPC
- What is pipe
- What is fifo
- How to use Message queue
- How to use shared Memory
- How to use Semaphore
- Semaphore Vs Mutex

Any Questions ?

Thank you!