

BITS Pilani - Hyderabad Campus

Advanced Operating Systems

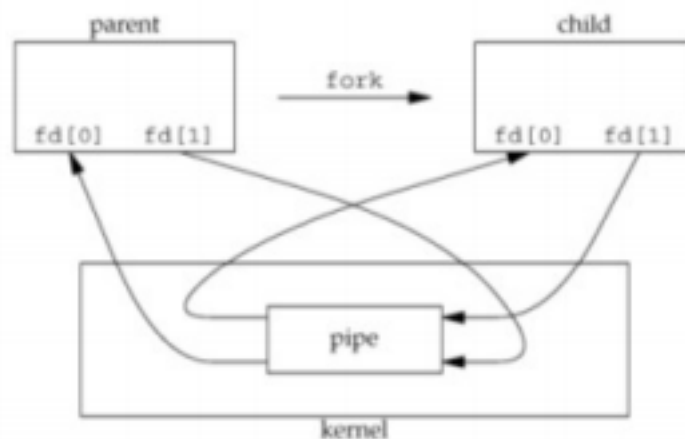
Lab Assignment - 2

1st Semester 2025 - 26

Exercise 1. Write a program that takes a file name as an argument, opens the file, reads it, and closes the file. The file should contain a string with the name of another application (e.g., 'ls' or 'ps' or any of your own applications) and the program forks a new process that executes the application named in the file.

1. Pipes :

A call to `pipe()` returns a pair of file descriptors. After the call to `fork()` following `pipe()`, both the parent and child share the pipe as shown in the figure below. So, we need to close the unwanted ends as in the program above. Code to implement "ls | wc -l". You need to use `dup()` or `dup2()` system call. Read man page on how to use them.



```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<errno.h>
#include<sys/wait.h>
int main(int argc, char *argv[]) {
    pid_t pid;
    int pfd[2];
    int s;
    pipe(pfd);
    pid = fork();
    if (pid == 0) {
        dup2(pfd[1], 1);
        close(pfd[0]);
        close(pfd[1]);
        if (execlp("ls", "ls", NULL) == -1) /* first child runs "ls" */
            perror("execlp ls");
    }
    else {
        if (fork() == 0) {
            dup2(pfd[0], 0);
            close(pfd[0]);
            close(pfd[1]);
        }
    }
}
```

```

        if (execlp("wc", "wc", "-l", NULL) == -1) /*second child runs wc */
            perror("execlp wc");
    }
    else {
        close(pfd[0]);
        close(pfd[1]);
        wait(&s);
        wait(&s); /* need to wait for both the children */
    }
}
}
}

```

2. Message Queues :

Message queues provide an additional technique for IPC. The main advantage of using Message Queues is that they provide “Asynchronous Communication Protocols” i.e; the sender and the receiver do not need to be active at the same time. The messages sent by a process are stored at a location which can be read at a later time by the receiver. Basic Message Passing IPC lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

For sending/receiving messages from a queue you need to first create it. The creation is done by the `msgget()` function which looks as follows :

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

Once the queue has been created, you can use `msgsnd()` to send messages into the queue. A message typically consists of two parts. For example the structure below can be used to send and receive the messages.

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

It is possible to send only one-byte arrays into a queue. So now that we know how the queue is created and how a message looks like, let us send one using the `msgsnd()` system call. The function looks as follows:

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

Receiving from queue:

The `msgrcv()` function which looks as follows:

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

How is a message queue deleted using the `msgctl()` command .?

The `msgctl()` call has the following syntax:

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

Let us create two programs which will communicate amongst themselves through message queues:

To use this program first compile and run `write.c` to add a message to the message queue.

To see the Message Queue type `ipcs -q` on your Ubuntu Terminal.

// write.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXSIZE 128
```

```
void die(char *s)
{
    perror(s);
    exit(1);
}
```

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

```
main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    struct msgbuf sbuf;
    size_t buflen;
```

```
    key = 1234;
```

```
    if ((msqid = msgget(key, msgflg )) < 0) //Get the message queue ID for the given
        key die("msgget");
```

```
    //Message Type
    sbuf.mtype = 1;
```

```
    printf("Enter a message to add to message queue : ");
    scanf("%s", sbuf.mtext);
    getchar();
```

```
    buflen = strlen(sbuf.mtext) + 1 ;
```

```
    if (msgsnd(msqid, &sbuf, buflen, IPC_NOWAIT) < 0)
    {
        printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext,
        buflen); die("msgsnd");
    }
```

```
    else
        printf("Message Sent\n");
```

```
    exit(0);
}
```

//read.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE 128

void die(char *s)
{
    perror(s);
    exit(1);
}

typedef struct msgbuf
{
    long mtype;
    char mtext[MAXSIZE];
};

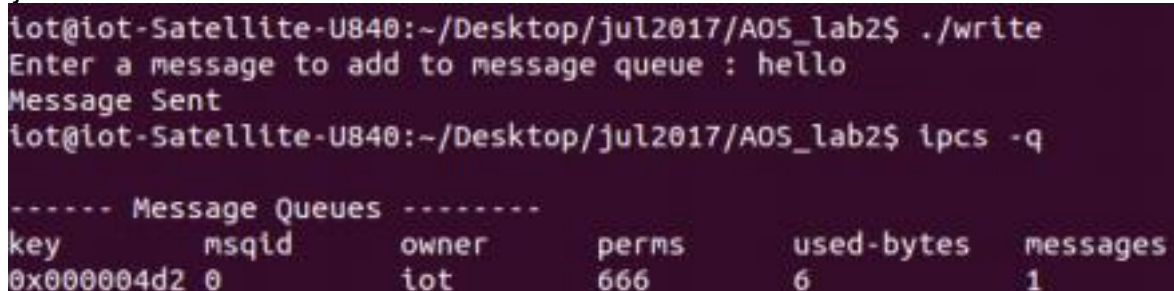
main()
{
    int msqid;
    key_t key;
    struct msgbuf rcvbuffer;

    key = 1234;

    if ((msqid = msgget(key, 0666)) < 0)
        die("msgget()");

    //Receive an answer of message type 1.  if
    (msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0) < 0)
        die("msgrcv");

    printf("%s\n", rcvbuffer.mtext);
    exit(0);
}
```



iot@iot-Satellite-U840:~/Desktop/jul2017/AOS_lab2\$./write
Enter a message to add to message queue : hello
Message Sent
iot@iot-Satellite-U840:~/Desktop/jul2017/AOS_lab2\$ lpcs -q

----- Message Queues -----					
key	msqid	owner	perms	used-bytes	messages
0x000004d2	0	iot	666	6	1

3. Shared Memory:

Shared memory is another way by which two or more processes can communicate. The basic idea is that one program will create a shared memory portion where it will put a certain

amount of data while the other process will read it. The method used for creating and sending messages by using shared memory is similar to that of message queues.

The creation of the shared memory space is done using `shmget()` which looks as follows: **`int shmget(key_t key, size_t size, int shmflg);`**

Before sharing any message, the process has to attach/connect itself to that shared memory space. That is done via the `shmat()` call which looks as follows:

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

After that, the sharing of message is trivial. You can just use a `strncpy()` or any other `cpy()` statement to copy the message into that memory location which is returned by `shmat()`. The other process can then read it. Similar to message queues, each process needs to detach itself from the shared memory once the job is done and then destroy the occupied memory block. This is achieved through `shmdt()` and `shmctl()` calls.

`int shmdt(void *shmaddr);`

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

Now let's look at a program where a parent and a child communicate through the shared memory.

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

#include <sys/shm.h>

#include <sys/stat.h>
int main ()
{
    int segment_id;
    char* shared_memory;
    struct shmid_ds shmbuffer;
    int segment_size;
    const int shared_segment_size = 0x6400;

    /* Allocate a shared memory segment. */
    segment_id = shmget (IPC_PRIVATE, shared_segment_size,
IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);

    /* Attach the shared memory segment. */
    shared_memory = (char*) shmat (segment_id, 0, 0);
    printf ("shared memory attached at address %p\n", shared_memory);

    /* Determine the segment's size. */
    shmctl (segment_id, IPC_STAT, &shmbuffer);
    segment_size = shmbuffer.shm_segsz;
    printf ("segment size: %d\n", segment_size);

    /* Write a string to the shared memory segment. */
    sprintf (shared_memory, "Hello, world.");

    /* Detach the shared memory segment. */
    shmdt (shared_memory);

    /* Reattach the shared memory segment, at a different address. */
    shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
    printf ("shared memory reattached at address %p\n", shared_memory);
}
```

```

/* Print out the string from shared memory. */
printf ("%s\n", shared_memory);

/* Detach the shared memory segment. */
shmdt (shared_memory);

/* Deallocate the shared memory segment. */
shmctl (segment_id, IPC_RMID, 0);
return 0;
}

```

The `ipcs` command provides information on interprocess communication facilities, including shared segments. Use the `-m` flag to obtain information about shared memory.

% `ipcs -m`

```

iot@iot-Satellite-U840:~/Desktop/jul2017/A05_lab2$ ./shm
shared memory attached at address 0x7f229c537000
segment size: 25600
shared memory reattached at address 0x50000000
Hello, world.
iot@iot-Satellite-U840:~/Desktop/jul2017/A05_lab2$ ipcs -m
----- Shared Memory Segments -----
key          shmid    owner    perms    bytes    nattch   status
0x00000000   65536    iot      600      524288    2        dest
0x00000000   163841   iot      600      524288    2        dest
0x00000000   950274   iot      600      524288    2        dest
0x00000000   360451   iot      600      524288    2        dest

```

Practice Questions:

Exercise 1. Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronise and notify each process when operations such as memory loaded and memory read have been performed.

Exercise 2 Write a program to create pipe.c to simulate the behaviour shown in the below figure.

