Preview

- FIFOs
- XSI IPC
  - Message Queue
  - Semaphore
  - Shared Memory

FIFOs

- **Pipe**s can be used only between related processes when a common ancestor has created the pipe.
- **FIFO**s (Named pipes) allow two unrelated processes to communicate with each other.
- Since **FIFO** is a type of file, creating a FIFO is similar to creating a file.
- Two unrelated processes can open a FIFO and begin communication.

FIFOs

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
                                    Return 0 if Ok, -1 error
```

- The specification of mode argument is the same as for the open system call.
- The rule for user and group ownership of a **FIFO** are the same as in a file.
- Once we create a **FIFO** by using mkfifo, we can open it by using open().

FIFOs

- A FIFO supports blocked read and write operations by default: if a process opens the FIFO for reading, it is blocked until another process opens the FIFO for writing, and vice versa.
- However, it is possible to make FIFOs support non-blocking operations by specifying the O_NONBLOCK flag while opening them.
- A FIFO 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.

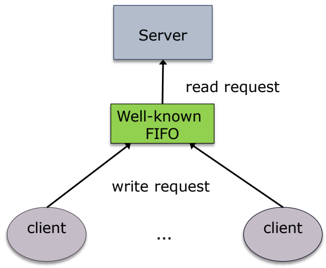
FIFOs

- In the normal case (O_NONBLOCK not specified)
  - An FIFO open for read-only blocks until some other process opens the FIFO for writing.
  - An FIFO open for write-only blocks until some other process open the FIFO for reading.
- If O_NONBLOCK is specified
  - An open for read-only returns immediately.
  - An open for write-only returns –l with errno set to ENXIO if no process has the FIFO open for reading.

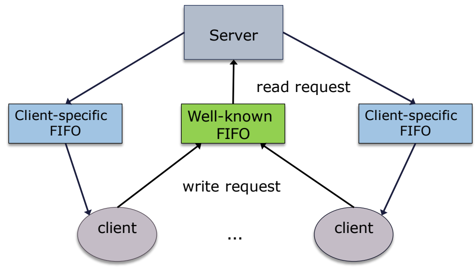
FIFOs

- **FIFO**s are used by shell commands to pass data from one shell pipleline to another without creating intermediate temporary files.
- **FIFO**s are used as rendezvous point in client-server applications to pass data between the clients and the servers.

FIFOs



FIFOs




```c
 /* server.c create a FIFO to communicate with client*/
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#define HALF_DUPLEX  "halfduplex"
#define MAX_BUF_SIZE 255
int main(int argc, char *argv[])
{
    int fd, ret_val, count, numread;
    char buf[MAX_BUF_SIZE];
    /* Create the FIFO(named - pipe) */
    ret_val = mkfifo(HALF_DUPLEX, 0666);
    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }
    /* Open the FIFO for reading */
    fd = open(HALF_DUPLEX, O_RDONLY);
    /* Read from the FIFO */
    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);
    return 0;
}
```

FIFOs

```c
 /* client.c write a string to FIFO */
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#define HALF_DUPLEX "halfduplex"
#define MAX_BUF_SIZE 255

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]));
    return 0;
}
```

```c
/*server1.c which receive two integer through FIFO and calculate it's sum.
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#define HALF_DUPLEX        "halfduplex"
#define BFFERSIZE          20

int main(int argc, char *argv[])
{
    int fd, ret_val, count, numread;
    char line[BFFERSIZE];

    /* Create the named - pipe */
    ret_val = mkfifo(HALF_DUPLEX, 0666);

    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }

    /* Open the FIFO for reading */
    fd = open(HALF_DUPLEX, O_RDONLY);
    int int1, int2, n, sum;
      while((n=read(fd, line, BFFERSIZE))>0)
      {
         line[n]='\0';
         if (sscanf(line, "%d %d", &int1, &int2)== 2)
         {
                    sprintf(line, "The sum is %d\n", int1+int2);
                    write(1, line, strlen(line));
         }
         else
                     write(1, "invalid arguments\nEnter two integers\n", 37);
      }
     return o;
}
```


FIFOs

```c
/*client1.c which send two integer through FIFO to server1.
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#define HALF_DUPLEX                "halfduplex"
#define BFFERSIZE      20

int main(int argc, char *argv[])
{
    int fd, n;
    char line[BFFERSIZE];


    /* Open the FIFO for writing */
    fd = open(HALF_DUPLEX, O_WRONLY);

     printf("Enter to integers\n Press Ctrl+D to exit\n");
      while(fgets(line, BFFERSIZE, stdin)!=NULL)
      {
         n=strlen(line);
         write(fd,line,n); //pass information from  to child through FIFO
      }
```

XSI Interprocess Communication

- There are three types of XSI IPC
  - Message queue
  - Semaphore
  - Shared memory
- Each IPC structure in the kernel is referred to by a **non-negative identifier**.
- When a given IPC structure is created and then removed, the identifier associated with that structure continually increase up to the maximum positive integer, and then wraps around to 0.

XSI Interprocess Communication

- When an XSI IPC structure is created (by calling **msgget()**, **semget()** or **shmget()**), a **key** must be specified.
- The data type **key_t** for a key is specified in the header file <sys/types.h>.

XSI Interprocess Communication

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
                              Return key if Ok, -1 error
```

- The *ftok*() function return a key based on path and id that is usable in subsequent calls to *msgget*(), *semget*(), and *shmget*().
- The application shall ensure that the *path* argument is the pathname of an existing file.
- Only lower 8 bit of id are used when generating a queue (∴we can path a character).

XSI Interprocess Communication

- XSI IPC associated with ipc_perm structure.
- This structure defines the permissions and owner and so on.

```
struct ipc_perm {
   uid_t uid; /* owner's effective user ID*/
   gid_t gid; /* owner's effective group ID */
   uid_t cuid; /* creator effective user ID */
   gid_t cgid; /* creator effective group ID */
   mode_t mode /* access mode */
};
```


XSI Interprocess Communication

- All the fields are initialized when the IPC structure is created.
- We can modify the uid, gid, and mode filed by calling msgctl(), semctl() or shmctl().
- The value of mode fields are:

Operation permissions	Octal value
Read by user	00400 = 000 000 100 000 000
Write by user	00200 = 000 000 010 000 000
Read by group	00040 = 000 000 000 100 000
Write by group	00020 = 000 000 000 010 000
Read by others	00004 = 000 000 000 000 100
Write by others	00002 = 000 000 000 000 010

XSI IPC (Message Queue)

- A message queue is a linked list of message stored within the kernel's space and identified by a message queue ID.
- A new message queue is created or opened by **msgget()**.
- A new messages are added to the end of a queue by **msgsnd()**.
- Messages are fetched from a queue by **msgrcv()**.
- We don't have to fetch the message in a First In First Out order. Instead, we can fetch messages based on their type field.

XSI IPC (Message Queue)

- Each queue has the msqid_ds structure associated with it.

```
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue */
    struct msg *msg_last;   /* last message in queue */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;        /* number of message on queue */
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* last receive pid */
};
```


XSI IPC (Message Queue)

```
/* one msg structure for each message */
struct msg {
    struct msg *msg_next;   /* next message on queue */
    long  msg_type;
    char *msg_spot;         /* message text address */
    short msg_ts;           /* message text size */
};
```

- **msg_next:** This is a pointer to the next message in the queue
- **msg_type:** This is the message type, as assigned in the user structure msgbuf
- **msg_spot:** A pointer to the beginning of the message body.
- **msg_ts:** The length of the message text, or body.

XSI IPC (Message Queue)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t key, int msgflg);
                              Return key if Ok, -1 error
```

- We can create or open a message queue.
- If a new queue is created, the msqid_ds structure are initiated.
  - msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are all set to 0.
  - msg_ctime is set to the current time
  - msg_qbyte is set to the system limit.

XSI IPC (Message Queue)

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
                                        Return key if Ok, -1 error
```

- The `msgctl()` system call provides a variety of message control operations as specified by *cmd*.
  - **IPC_STAT** Copies the current attributes of the message queue associated with *msqid* into the structure that *buf* points to
  - **IPC_SET** Sets the attributes of the associated with *msqid* from the values found in the structure that *buf* points to
  - **IPC_RMID** Removes the message queue identifier specified by *msqid* from the system and destroys the message queue

XSI IPC (Message Queue)

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
                                              Return key if Ok, -1 error
```

- The msgsnd() function sends a message to the queue associated with message queue identifier *msqid.*
- If the call completes successfully, the following actions are taken with respect to msqid_ds associated with *msqid*:
  - msg_qnum is incremented by 1.
  - msg_lspid is set to the process ID of the calling process.
  - msg_stime is set to the current time.

XSI IPC (Message Queue)

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
                                              Return key if Ok, -1 error
```

- The argument *msgp* must point to a user-defined buffer that must contain first a field of type long int that specifies the type of the message, and then a data portion that holds the data bytes of the message.

```
struct mymsg {
        long int mtype;  /* positive message type */
        char mtext[n];   /* message data of n bytes */
}
```

XSI IPC (Message Queue)

```
#include <sys/msg.h>
     int msgrcv(int msqid, void *msgp, int msgsz, long msgtyp, int msgflg);
                                                 Return key if Ok, -1 error
```

- The msgrcv() function reads a message from the queue associated with *msqid* and places it in the user-defined structure that *msgp* points to.
- When successfully completed, the following actions are taken with respect to the data structure associated with *msqid*:
  - msg_qnum is decremented by 1.
  - msg_lrpid is set to the process ID of the calling process.
  - msg_rtime is set to the current time.

XSI IPC (Message Queue)

```
#include <sys/msg.h>
     int msgrcv(int msqid, void *msgp, int msgsz, long msgtyp, int msgflg);
                                                 Return key if Ok, -1 error
```

- *msgtyp* Specifies the type of message requested as follows:
  - If *msgtyp* is 0, the first message on the queue is received.
  - If *msgtyp* is greater than 0, the first message of type equal to *msgtyp* is received.
  - If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.
- *msgflg* Specifies the action to be taken if a message of the desired type is not in the queue.

XSI IPC (Message Queue)

- Kirk
  - Create a message queue and send messages as many as possible.
  - Message queue created by Kirk will save messages.
- Spock
  - Open the message queue.
  - Receive messages from message queue.


```c
/* kirt.c get lines of text and added into the message queue */
/* Then, the message queue is then read by spock.c           */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
/* user message type with 200 byte per message */
struct my_msgbuf {
    long mtype;
    char mtext[200];
};
int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;
    /*create a key for create message queue */
    if ((key = ftok("kirk.c", 'B')) == -1) {
        perror("ftok error");
        exit(1);
    }
    /*create a message queue */
    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget error");
        exit(1);
    }
    printf("Enter lines of text, ^D to quit:\n");
    buf.mtype = 1; /* we don't really care in this case, just used as FIFO*/
    while(gets(buf.mtext), !feof(stdin)) {
        if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0) == -1)
            perror("msgsnd error");
    }
    /* Now remove message queue by calling msgclt */
    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl error");
        exit(1);
    }
    exit(0);
}
```



```c
/*spock.c read message from the message queue */
/* created by kirt.c                          */
#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;

    /* create a key same as kirt.c */
    if ((key = ftok("kirk.c", 'B')) == -1) {
        perror("ftok error");
        exit(1);
    }
    /* open message queue already created by kirk.c */
    if ((msqid = msgget(key, 0644)) == -1) {
        perror("msgget error");
        exit(1);
     }
    printf("spock: ready to receive messages, captain.\n");

    for(;;) {
        /* get each message from the message queue */
        if (msgrcv(msqid, (struct msgbuf *)&buf, sizeof(buf), 0, 0) == -1) {
            perror("msgrcv error");
            exit(1);
        }
        printf("spock: \"%s\"\n", buf.mtext);
    }

    return 0;
}
```

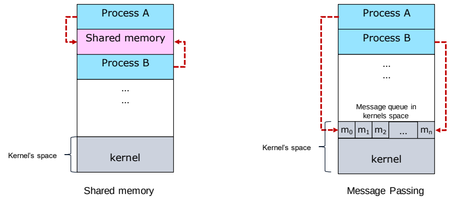

XSI IPC (Shared Memory)

- There are two fundamental models of interprocess communication:
  - Shared Memory- a region of memory is shared by processes with read /write operations. It is useful for exchanging smaller amount of data since no conflicts need be avoided.
  - Message Passing - communication takes place by means of messages exchanged between the cooperating processes (Message Queue). It is also easier to implement in a distributed system than shared memory.

XSI IPC (Shared Memory)

- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls (shared memory are located in user's space).
- In shared-memory systems, system calls are required only to establish shared memory regions.
- Once shared memory is established, all accesses are treated as routine memory accesses, without kernel's assistance.

XSI IPC (Shared Memory)



Communication Model

XSI IPC (Shared Memory)

- Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server (or between processes).
- The only trick in using shared memory is synchronizing access to a given region among multiple processes.
- Since OS does not support mutual exclusion, programmer must take care mutual exclusion of the region between multiple processes by using a semaphore.
- The kernel maintains a structure `shmid_ds` with at least the following members for each shared memory segment:

XSI IPC (Shared Memory)

```
struct shmid_ds {
        struct ipc_perm shm_perm;    /* Ownership and permissions */
        size_t          shm_segsz;   /* Size of segment (bytes) */
        time_t          shm_atime;   /* Last attach time */
        time_t          shm_dtime;   /* Last detach time */
        time_t          shm_ctime;   /* Last change time */
        pid_t           shm_cpid;    /* PID of creator */
        pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
        shmatt_t        shm_nattch;  /* No. of current attaches */
        ...
    };
```

```
struct ipc_perm {
        key_t          __key;    /* Key supplied to shmget(2) */
        uid_t          uid;      /* Effective UID of owner */
        gid_t          gid;      /* Effective GID of owner */
        uid_t          cuid;     /* Effective UID of creator */
        gid_t          cgid;     /* Effective GID of creator */
        unsigned short mode;     /* Permissions + SHM_DEST and
                                    SHM_LOCKED flags */
        unsigned short __seq;    /* Sequence number */
    };
```


XSI IPC (Shared Memory)

- Before using the shared memory what we needs to be done with the system calls,
  - 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()`)

XSI IPC (Shared Memory)

```
#include <sys/shm.h>
#include <sys/ipc.h>
int shmget (key_t key, size_t size, int shmflg);
                              Return shared memory ID if Ok, -1 error
```

- We can create or open a shared memory with shmget() system call.
- The key can be either an arbitrary value or one that can be derived from the library function ftok().
- The size parameter is the size of the shared memory segment in bytes.
  - If a new segment is being created (server), we must specify its size.
  - If we are referencing an existing segment (a client), we can specify size as 0.
- The shmflg parameter specifies the required shared memory flags such as
  - IPC_CREAT :creating new segment
  - IPC_EXCL: used with IPC_CREAT to create new segment and the call fails, if the segment already exists).

XSI IPC (Shared Memory)

```
#include <sys/shm.h>
#include <sys/ipc.h>
int shmget (key_t key, size_t size, int shmflg);
                              Return shared memory ID if Ok, -1 error
```

- If a new shared memory is created, the `ipc_perm` structure are initiated.
  - shm_lpid, shm_nattch, shm_atime, and shm_dtime are all set to 0.
  - shm_ctime is set to the current time.
  - shm_segsz is set to the size requested.

XSI IPC (Shared Memory)

- Once a shared memory segment has been created, a process attaches it to its address space by calling system call `shmat()`.

```
#include <sys/shm.h>
#include <sys/ipc.h>
void *shmat(int shmid, const void *addr, int flag);

         Return the address of attached shared memory if Ok, -1 error
```

  - `shmid`: ID return by `shmget()` system call.
  - `addr`: is to specify the attaching address. If `addr` is `NULL`, the system chooses the suitable address to attach the segment by default. If it is not `NULL` and `SHM_RND` is specified in `flag`, attach is equal to the address of the nearest multiple of `SHMLBA`(Lower Boundary Address).

XSI IPC (Shared Memory)

```
#include <sys/shm.h>
#include <sys/ipc.h>
void *shmat(int shmid, const void *addr, int flag);

         Return the address of attached shared memory if Ok, -1 error
```

- flag: specifies the required shared memory flags
  - `SHM_RND` (rounding off address to SHMLBA)
  - `SHM_EXEC` (allows the contents of segment to be executed)
  - `SHM_RDONLY` (attaches the segment for read-only purpose, by default it is read-write)
  - `SHM_REMAP` (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).

XSI IPC (Shared Memory)

```
#include <sys/shm.h>
#include <sys/ipc.h>
int shmdt(const void *addr);

                                   Return 0 if Ok, -1 error
```

- `shmdt()` system call detach the shared memory segment from the address space of calling process.

XSI IPC (Shared Memory)

- The shmctl function is used for various shared memory operations.

```
#include <sys/shm.h>
int shctl (int shmid, int cmd, struct shmid_ds *buf);
              Return shared memory ID or 0 if Ok, -1 error
```

- The cmd argument specifies one of the following five commands to be performed, on the segment specified by shmid.
  - IPC_STAT : Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by buf.
  - IPC_SET: Set the three fields from the structure pointed to by buf: shm_perm.uid, shm_perm.gid, and shm_perm.mode. (only possible to modify when a process is supper user or effective user id is same as shm_perm.cuid or shm_perm.uid)

XSI IPC (Shared Memory)

```
#include <sys/shm.h>
int shctl (int shmid, int cmd, struct shmid_ds *buf);
                  Return shared memory ID if Ok, -1 error
```

- The cmd argument continue
  - IPC_RMID – Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.
  - IPC_INFO – Returns the information about the shared memory limits and parameters in the structure pointed by buf.
  - SHM_LOCK :Lock the shared memory segment in memory. This command can be executed only by the superuser
  - SHM_UNLOCK :Unlock the shared memory segment in memory. This command can be executed only by the superuser

XSI IPC (Shared Memory)

```
// header.h
#define  NOT_READY  -1
#define  FILLED      0 //when sender fill data
#define  TAKEN       1 //when receiver take data
#define  GO          2 // when sender keep sending
#define  STOP        3 // when sender stop sending data

struct student {
        int id;
        char lname[20];
        char fname[20];
};
struct Memory {
     int  status; //FILLED or TAKEN
     int  gostop; //GO or STOP
     struct student data;
};
```



```
//buildsm.sh
#include<stdio.h>
#include<stdlib.h>
#include<sys/shm.h>
#include<errno.h>
#include "header.h"

int main(int argc, char *argv[])
{
          int shmid;
          key_t key;
          struct Memory *shm;
          key = ftok(".", 'x'); //create a key value
          //create a shared memory
          if ((shmid = shmget(key, sizeof(struct Memory), IPC_CREAT | 0666)) <0)
          {
                    perror("shmget error \n");
                    exit (1);
          }
          shm = (struct Memory *) shmat(shmid, NULL, 0); //attach to shared memory
          if ((long)shm == -1)
          {
                    perror("shmat error \n");
                    exit (1);
          }
          shm->status = NOT_READY;
          shm->gostop = GO;
          return 0;
}
```


XSI IPC (Shared Memory)

```
// removesm.c
#include<stdio.h>
#include<sys/shm.h>
#include<errno.h>
#include<stdlib.h>
#include "header.h"


int main(int argc, char *argv[]) {
   key_t key;
   int shmid;
   struct Memory shm;
   key = ftok(".", 'x');
   if ((shmid = shmget(key, sizeof(struct Memory), 0)) <0)
   {
          perror("shmget error \n");
          exit (1);
   }

    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```



```
//sender.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/shm.h>
#include<errno.h>
#include "header.h"

int main(int argc, char *argv[])
{
          int shmid;
          key_t key;
          struct Memory *shm;
          char name[20];
          int n, id, more, i;

          key = ftok(".", 'x'); //get key value
          if ((shmid = shmget(key, sizeof(struct Memory), 0)) <0) //open shared memory
          {
                    perror("shmget error \n");
                    exit (1);
          }
          shm = (struct Memory *) shmat(shmid, NULL, 0); //attach to shared memory
          if ((long)shm == -1)
          {
                    perror("shmat error \n");
                    exit (1);
          }
          shm->gostop = GO;
          shm->status = NOT_READY;
```


XSI IPC (Shared Memory)

```
          printf("Number of Student Data?");
                    scanf("%d",&more);
          for (i=0; i <more; i++)
          {
                    printf("Student's ID ? ");
                    scanf("%d", &id);
                    shm->data.id = id;
                    printf("Last Name? ");
                    scanf("%s",name);
                    strcpy(shm->data.lname, name);
                    printf("First Name? ");
                    scanf("%s",name);
                    strcpy(shm->data.fname, name);
                    shm->status = FILLED;

                    while (shm->status != TAKEN)
                       ;
                    printf("Data is taken by other process\n");
          }
          shm->gostop = STOP;
          shmdt((void *) shm); //detach
          return 0;
}
```



```
//receiver.c
#include<stdio.h>
#include<stdlib.h>
#include<sys/shm.h>
#include<errno.h>
#include "header.h"

int main(int argc, char *argv[])
{
          int shmid, n, int1, int2;
          key_t key;
          struct Memory *shm;
          key = ftok(".", 'x'); //get key value
          // open existing shared memory
          if ((shmid = shmget(key, sizeof(struct Memory), 0)) <0)
          {
                    perror("shmget error \n");
                    exit (1);
          }
           //a pointer is attach to shared memory
          shm = (struct Memory *) shmat(shmid, NULL, 0);
          if ((long)shm == -1)
          {
                    perror("shmat error \n");
                    exit (1);
          }
```

XSI IPC (Shared Memory)

```
//continue…
        // read from the shared memory
        while (shm->gostop == GO)
        {
                while (shm->status != FILLED)
                {
                        if (shm->gostop == STOP)
                                break;
                        ;
                }
                printf ("Student ID: %d \n", shm->data.id);
                printf ("Student Last Name: %s\n", shm->data.lname);
                printf ("Student First Name: %s\n",shm->data.fname);
                shm->status = TAKEN;
        }
        shmdt((void *) shm); //detach

        return 0;
}
```



```
//server.c create a shared memory and write on shared memory
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<errno.h>
#define SHSIZE 100
int main(int argc, char *argv[])
{
        int shmid;
        key_t key;
        char *shm, *s;
        key = ftok(".", 'x'); //create a key value
        //create a shared memory with size 100 byte
        if ((shmid = shmget(key, SHSIZE, IPC_CREAT | 0666)) <0)
        {
                perror("shmget error \n");
                exit (1);
        }
        shm = shmat(shmid, NULL, 0); //attach pointer to the shared memory
        if (shm == (char*) -1)
        {
                perror("shmat error \n");
                exit (1);
        }
        memcpy (shm, "Hello World", 11); //write to shared memory you can use write system call

        s = shm;
        s+=11;
        *s = 0;
        while (*shm != '*') //server will wait until client read and type * in shared memory
                sleep (1);

        printf("Server has detected the completion of its child...\n");
        shmdt((void *) shm); //detach shared memory
        printf("Server has detached its shared memory...\n");
        shmctl(shmid, IPC_RMID, NULL); //remove shared memory
        printf("Server has removed its shared memory...\n");
        return 0;
}
```



```
// client.c ; open shared memory and read data
#include<string.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<errno.h>

#define SHSIZE 100

int main(int argc, char *argv[])
{
        int shmid;
        key_t key;
        char *shm, *s;
        key = ftok(".", 'x'); //create a key value

        if ((shmid = shmget(key, SHSIZE,0666)) <0) //open shared variable created by server
        {
                perror("shmget error \n");
                exit (1);
        }
        shm = shmat(shmid, NULL, 0); // attach a pointer to shared memory
        if (shm == (char*) -1)
        {
                perror("shmat error \n");
                exit (1);
        }
        for (s =shm; *s != 0; s++) //read available data from the shared memory
                printf("%c", *s);
        printf("\n");
        *shm = '*'; // write a '*' to shared memory which inform to server that client done its job
        return 0;
}
```


XSI IPC (Semaphore)

- A semaphore is not a form of IPC similar to the others (pipes, FIFOs or message queue, shared memory).
- A semaphore is a counter used to protect to a shared data object for multiple processes.
- To access (read or write) a shared data object, a process must check semaphore.
- Modification to the a semaphore are executed **indivisibly**.

XSI IPC (Semaphore)

- To access a shared resources, a process needs to do the followings:
  - Test the semaphore that controls the resources.
  - If the value of the semaphore is >0, the process reduce the value by 1 and access resources. Check and modification to the a semaphore are executed indivisibly.
  - If the value of the semaphore is 0, the process need go to sleep on the semaphore until the value becomes greater than 0.

XSI IPC (Semaphore)

Ex)

- Lets there are two processes P_1, P_2 working on their job and , and two resource R_1 and R_2.
- Both P_1 and P_2 need R_1 and R_2 to finish their job.
- Each resource is associated with a semaphore.

XSI IPC (Semaphore)

Case 1)

```
semaphore R1;
semaphore R2;
void process_P1()
{
   down(&R1);
   down(&R2);
   use_both_resource();
   up(&R2);
   up(&R1);
}

void process_P2()
{
   down(&R1);
   down(&R2);
   use_both_resource();
   up(&R2);
   up(&R1);
}
```


XSI IPC (Semaphore)

Case 2)

```
semaphore R1;
semaphore R2;
void process_P1()
{
   down(&R1);
   down(&R2);
   use_both_resource();
   up(&R2);
   up(&R1);
}

void process_P2()
{
   down(&R2);
   down(&R1);
   use_both_resource();
   up(&R1);
   up(&R2);
}
```