UNIT-IV (INTER PROCESSES COMMUNICATION)

1Q) Describe about Pipes?

Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.

- Historically, they have been half duplex (i.e., data flows in only one direction).
- Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.

A pipe is created by calling the `pipe` function.

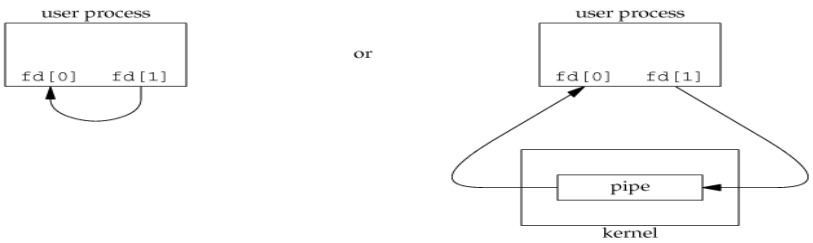
```
#include <unistd.h>

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

Returns: 0 if OK, 1 on error.

Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

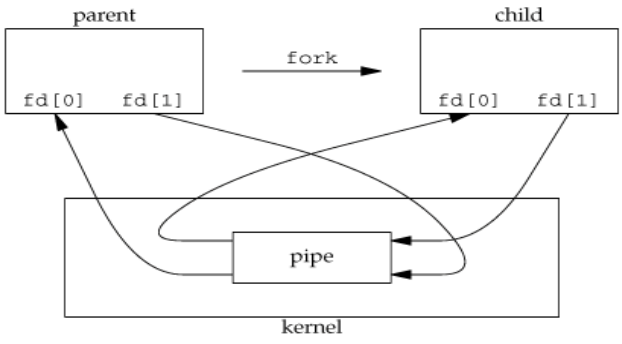
Two ways to picture a half-duplex pipe are shown in Figure . The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

Two ways to view a half-duplex pipe



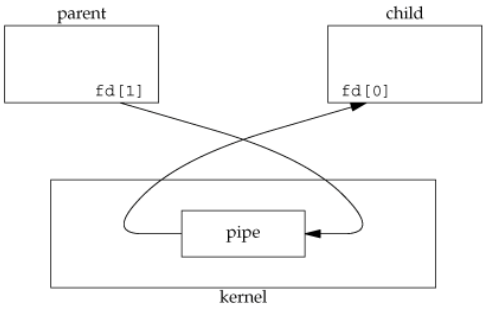
A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child or vice versa. Figure shows this scenario.

Figure 15.3 Half-duplex pipe after a `fork`



What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Figure shows the resulting arrangement of descriptors.

Figure 15.4. Pipe from parent to child



For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`. When one end of a pipe is closed, the following two rules apply.

- If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all the data has been read.
- If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, `write` returns 1 with `errno` set to `EPIPE`.

2Q) Explain about popen () and pclose () system calls?

Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the `popen` and `pclose` functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, `fork`ing a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>
```

Returns: file pointer if OK, `NULL` on error

```
int pclose(FILE *fp);
```

Returns: termination status of cmdstring, or 1 on error

The function `popen` does a `fork` and `exec` to execute the cmdstring, and returns a standard I/O file pointer. If type is `"r"`, the file pointer is connected to the standard output of cmdstring

Result of `fp = popen`(cmdstring, `"r"`)

If type is `"w"`, the file pointer is connected to the standard input of cmdstring, as shown:

Result of `fp = popen`(cmdstring, `"w"`)

3Q) Explain about FIFO Pipe?

FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.

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

int mkfifo(const char *pathname, mode_t mode);
```

Returns: 0 if OK, 1 on error

Once we have used `mkfifo` to create a FIFO, we open it using `open`. When we `open` a FIFO, the nonblocking flag (`O_NONBLOCK`) affects what happens.

- In the normal case (`O_NONBLOCK` not specified), an `open` for read-only blocks until some other process opens the FIFO for writing. Similarly, an `open` for write-only blocks until some other process opens the FIFO for reading.
- If `O_NONBLOCK` is specified, an `open` for read-only returns immediately. But an `open` for write-only returns 1 with `errno` set to `ENXIO` if no process has the FIFO open for reading.

There are two uses for FIFOs.

- ✓ FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
- ✓ FIFOs are used as rendezvous points in client-server applications to pass data between the clients and the servers.

Example Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file. Consider a procedure that needs to process a filtered input stream twice. Figure shows this arrangement.

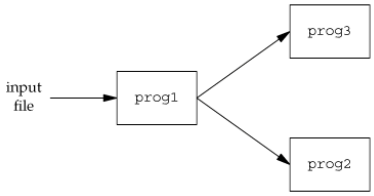
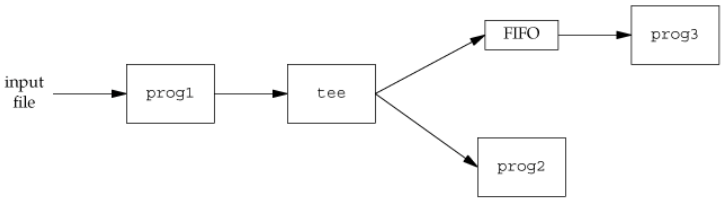


FIGURE : Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program `tee`(1), we can accomplish this procedure without using a temporary file. (The
`tee` program copies its standard input to both its standard output and to the file named on its command line.)

```
mkfifo fifo1 prog3 <
fifo1 &

prog1 < infile | tee
fifo1 | prog2
```

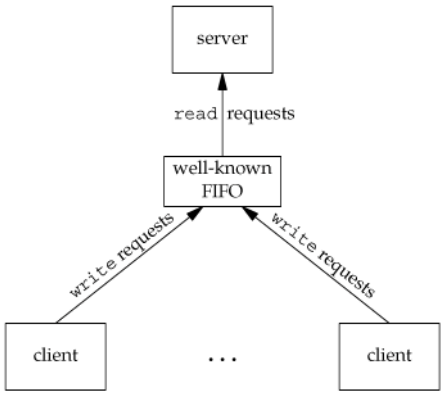
We create the FIFO and then start `prog3` in the background, reading from the FIFO. We then start `prog1` and use
`tee`to send its input to both the FIFO and `prog2`. Figure shows the process arrangement.



Using a FIFO and `tee` to send a stream to two different processes

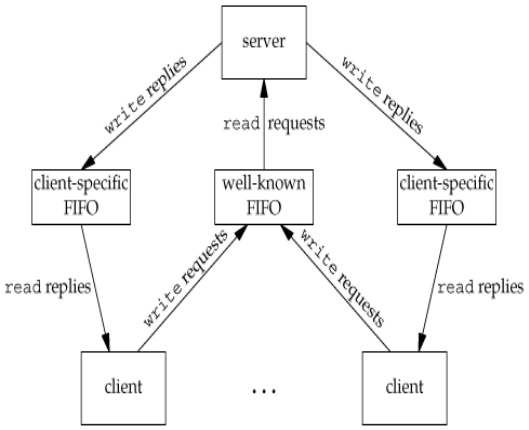
4Q) Explain with an example the CLIENT/SERVER communication using FIFO Pipe?

- FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.
- A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client'sprocess ID.
- For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the filesystem.
- The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.



Clients sending requests to a server using a FIFO

Client-server communication using FIFOs



5Q) Explain about Message Queues?

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

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.

Each queue has the following `msqid_ds` structure associated with it:

```
struct msqid_ds

{
        struct      msg perm;   /* see Section 15.6.2 */
        msgqnum t   msg qnum;   /* # of messages on queue */
        msglen_t    msg_qbytes  /* max # of bytes on queue */
        pid t       msg lspid;  /* pid of last msgsnd() */
        pid_t       msg_lrpid;  /* pid of last msgrcv() */
```

```
        time_t          msg_stime;     /* last-msgsnd() time */
        time_t          msg_rtime;     /* last-msgrcv() time */
        time_t          msg_ctime;    /* last-change time */

        .

        .

        .

};
```

This structure defines the current status of the queue.

The first function normally called is `msgget` to either open an existing queue or create a new

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

queue.
Returns: message queue ID if OK, 1 on error

When a new queue is created, the following members of the `msqid_ds` structure are initialized.

- ✓ The `ipc_perm` structure is initialized. The `mode` member of this structure is set to the corresponding permission bits of flag.
- ✓ `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_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

The `msgctl` function performs various operations on a queue.

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

Returns: 0 if OK, 1 on error.

The cmd argument specifies the command to be performed on the queue specified by msqid.

Table 9.7.2 POSIX:XSI values for the cmd parameter of msgctl.

cmd	description
IPC_RMID	remove the message queue msqid and destroy the corresponding msqid_ds
IPC_SET	set members of the msqid_ds data structure from buf
IPC_STAT	copy members of the msqid_ds data structure into buf

Data is placed onto a message queue by calling `msgsnd`.

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Returns: 0 if OK, 1 on error.

Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg
{
  long  mtype;      /* positive message type */
  char  mtext[512]; /* message data, of length nbytes */
};
```

The ptr argument is then a pointer to a `mymesg` structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

Messages are retrieved from a queue by `msgrcv`.

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


ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int
flag);
```

Returns: size of data portion of message if OK, 1 on error. The type argument lets us specify which message we want.

type == 0 The first message on the queue is returned.

type > 0 The first message on the queue whose message type equals type is returned.

type < 0 The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

6Q) Write short Notes on Shemaphores?

A semaphore is a counter used to provide access to a shared data object for multiple processes.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

A common form of semaphore is called a ***binary semaphore***. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

The first function to call is semget to obtain a semaphore ID.

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

Returns: semaphore ID if OK, 1 on error

When a new set is created, the following members of the semid_ds structure are initialized.

- The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.
- sem_otime is set to 0.
- sem_ctime is set to the current time.
- sem_nsems is set to nsems.

The number of semaphores in the set is nsems. If a new set is being created (typically in the server), we must specify nsems. If we are referencing an existing set (a client), we can specify nsems as 0.

The semctl function is the catchall for various semaphore operations.

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

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of
various command-specific arguments:

```
union semun

{

  int   val;              /* for SETVAL */

  struct semid_ds *buf;    /* for
  IPC_STAT and IPC_SET */ unsigned
  short          *array;  /* for
  GETALL and SETALL */

};
```

Table 9.8.1 POSIX:XSI values for the cmd parameter of semctl.

cmd	description
GETALL	return values of the semaphore set in arg.array
GETVAL	return value of a specific semaphore element
GETPID	return process ID of last process to manipulate element
GETNCNT	return number of processes waiting for element to increment
GETZCNT	return number of processes waiting for element to become 0
IPC_RMID	remove semaphore set identified by semid
IPC_SET	set permissions of the semaphore set from arg.buf
IPC_STAT	copy members of semid_ds of semaphore set semid into arg.buf
SETALL	set values of semaphore set from arg.array
SETVAL	set value of a specific semaphore element to arg.val

The *cmd* argument specifies one of the above ten commands to be performed on the set specified by semid.

The function `semop` atomically performs an array of operations on a semaphore set.

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, 1 on error.

The semoparray argument is a pointer to an array of semaphore operations, represented by

sembuf structures:

```
struct sembuf {

  unsigned short sem_num; /* member # in set (0, 1,
  ..., nsems-1) */ shortsem_op;        /*   operation
  (negative, 0, or positive) */ short sem_flg;      /*
  IPC_NOWAIT, SEM_UNDO */

};
```

The nops argument specifies the number of operations (elements) in the array.

The sem_op element operations are values specifying the amount by which the semaphore value is to be changed.

- If sem_op is an integer ***greater than zero***, semop adds the value to the corresponding semaphore element value and awakens all processes that are waiting for the element to increase.
- If sem_op is ***0*** and the semaphore element value is not 0, semop blocks the calling process (waiting for 0) and increments the count of processes waiting for a zero value of that element.
- If sem_op is a ***negative*** number, semop adds the sem_op value to the corresponding semaphore element value provided that the result would not be negative. If the operation would make the element value negative, semop blocks the process on the event that the semaphore element value increases. If the resulting value is 0, semop wakes the processes waiting for 0.

7Q) Explain about interprocess communication in Linux?

Inter process communication (IPC) is a mechanism whereby two or more processes communicate with each other to perform tasks. These processes may interact in a client/server manner or in a peer to peer fashion.

Examples: Database servers, E-Mails, IPC is supported is supported by all UNIX systems. However, different UNIX systems implement different methods for IPC. i.e BSD UNIX-Sockets, Unix System V.5 for Pipes, FIFOs, messages, semaphores and shared memory.

In computer science, inter-process communication or interprocess communication (IPC) refers specifically to the mechanisms an operating system provides to allow processes it manages to share data. Typically, applications can use IPC categorized as clients and servers, where the client requests data and the server responds to client requests. Many applications are both clients and servers, as commonly seen in distributed computing. Methods for achieving IPC are divided into categories which vary based on software requirements, such as performance and modularity requirements, and system circumstances, such as network bandwidth and latency.

Examples: These processes are not bound to one computer, and can run on various computers connected by network. Inter-process communication techniques can be divided into various types. These are:

1. Pipes
2. FIFO
3. Shared memory
4. Mapped memory
5. Message queues
6. Sockets

Pipes: The most basic versions of the UNIX operating system gave birth to pipes. These were used to facilitate one-directional communication between single-system processes. We can create a pipe by using the pipe system call, thus creating a pair of file descriptors

FIFO: A FIFO or 'first in, first out' is a one-way flow of data. FIFOs are similar to pipes, the only difference being that FIFOs are identified in the file system with a name. In simple terms, FIFOs are 'named pipes'.
Shared memory: Shared memory is an efficient means of passing data between programs. An area is created in memory by a process, which is accessible by another process. Therefore, processes communicate by reading and writing to that memory space.
Mapped memory: This method can be used to share memory or files between different processors in a Windows environment. A 32-bit API can be used with Windows. This mechanism speeds up file access, and also facilitates inter-process communication.
Message queues: By using this method, a developer can pass messages between messages via a single queue or a number of message queues. A system kernel manages this mechanism. An application program interface (API) coordinates the messages.
Sockets: We use this mechanism to communicate over a network, between a client and a server. This method facilitates a standard connection that is independent of the type of computer and the type of operating system used.

8Q) Explain the Properties of pipes?

1) Pipes do not have a name. For this reason, the processes must share a parent process. This is the main drawback to pipes. However, pipes are treated as file descriptors, so the pipes remain open even after fork and exec.

2) Pipes do not distinguish between messages; they just read a fixed number of bytes. Newline (\n) can be used to separate messages. A structure with a length field can be used for message containing binary data.

3) Pipes can also be used to get the output of a command or to provide input to a command

9Q) Write the Difference between named pipes and unnamed pipes?

A *pipe* is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.

A *FIFO* *special file* is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.
A pipe or FIFO has to be open at both ends simultaneously. If you read from a pipe or FIFO file that doesn't have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file. Writing to a pipe or FIFO that doesn't have a reading process is treated as an error condition; it generates a SIGPIPE signal, and fails with error code EPIPE if the signal is handled or blocked.
Neither pipes nor FIFO special files allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.

10Q) Write short notes on kernel support for Message Queues?

Kernel support for messages:
Message queues allow one or more processes to write messages, which will be read by one or more reading processes. Linux maintains a list of message queues, the msgque vector; each element of which points to a msqid_ds data structure that fully describes the message queue. When message queues are created a new msqid_ds data structure is allocated from system memory and inserted into the vector.

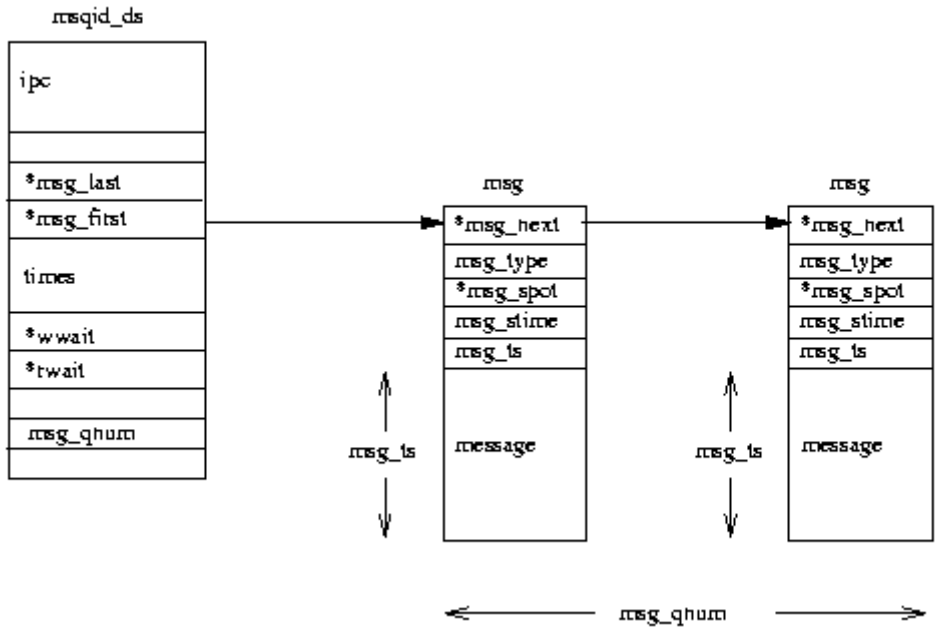


Figure: System V IPC Message Queues

Each msqid_ds data structure contains an ipc_perm data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. The msqid_ds also contains two wait queues; one for the writers to the queue and one for the readers of the message queue.

Each time a process attempts to write a message to the write queue its effective user and group identifiers are compared with the mode in this queue's ipc_perm data structure. If the process can write to the queue then the message may be copied from the process's address space into a msg data structure and put at the end of this message queue. Each message is tagged with an application specific type, agreed between the cooperating processes. However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be woken up when one or more messages have been read from this message queue.

Reading from the queue is a similar process. Again, the processes access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match this criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be woken up and run again.

11Q) Explain the Message Queues API?

Initializing the Message Queue

The msgget() function initializes a new message queue:

int msgget(key_t key, int msgflg)

It can also return the message queue ID (msqid) of the queue corresponding to the key argument. The value passed as the msgflgargument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the msgget() function.

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

...

```
key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
{
perror("msgget: msgget failed");
exit(1);
} else
(void) fprintf(stderr, &ldquo;msgget succeeded");
...
```

IPC Functions, Key Arguments, and Creation Flags: <sys/ipc.h>

Processes requesting access to an IPC facility must be able to identify it. To do this, functions that initialize or provide access to an IPC facility use a key_t key argument. (key_t is essentially an int type defined in <sys/types.h>

The key is an arbitrary value or one that can be derived from a common seed at run time. One way is with ftok() , which converts a filename to a key value that is unique within the system. Functions that initialize or get access to messages (also semaphores or shared memory see later) return an ID number of type int. IPC functions that perform read, write, and control operations use this ID. If the key argument is specified as IPC_PRIVATE, the call initializes a new instance of an IPC facility that is private to the creating process. When the IPC_CREAT flag is supplied in the flags argument appropriate to the call, the function tries to create the facility if it does not exist already. When called with both the IPC_CREAT and IPC_EXCL flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with IPC_EXCL in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If IPC_CREAT is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/tmp",
key), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a key based on the string ("/tmp"). The second argument evaluates to the combined permissions and control flags.

Controlling message queues

The msgctl() function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using msgctl() Also, any process with permission to do so can use msgctl() for control operations.

The msgctl() function is prototypes as follows:

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

The msqid argument must be the ID of an existing message queue. The cmd argument is one of:

IPC_STAT

-- Place information about the status of the queue in the data structure pointed to by buf. The process must have read permission for this call to succeed.

IPC_SET

-- Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC_RMID

-- Remove the message queue specified by the msqid argument.

The following code illustrates the msgctl() function with all its various flags:

```
#include<sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
if (msgctl(msqid, IPC_STAT, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
if (msgctl(msqid, IPC_SET, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
```

Sending and Receiving Messages

The msgsnd() and msgrcv() functions send and receive messages, respectively:

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

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

The msqid argument must be the ID of an existing message queue. The msgp argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
long    mtype;   /* message type */
char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

The msgsz argument specifies the length of the message in bytes.

The structure member msgtype is the received message's type as specified by the sending process.

The argument msgflg specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to msg_qbytes.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (msgflg & IPC_NOWAIT) is non-zero, the message will not be sent and the calling process will return immediately.

- If (msgflg & IPC_NOWAIT) is 0, the calling process will suspend execution until one of the following occurs:
    - The condition responsible for the suspension no longer exists, in which case the message is sent.
    - The message queue identifier msqid is removed from the system; when this occurs, errno is set equal to EIDRM and -1 is returned.
    - The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

Upon successful completion, the following actions are taken with respect to the data structure associated with msqid:

- msg_qnum is incremented by 1.
- msg_lspid is set equal to the process ID of the calling process.
- msg_stime is set equal to the current time.

The following code illustrates msgsnd() and msgrcv():

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

...

int msgflg; /* message flags for the operation */
struct msgbuf *msgp; /* pointer to the message buffer */
int msgsz; /* message size */
long msgtyp; /* desired message type */
int msqid /* message queue ID to be used */

...

msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
- sizeof msgp->mtext + maxmsgsz));

if (msgp == NULL) {
(void) fprintf(stderr, "msgop: %s %d byte messages.\n",
"could not allocate message buffer for", maxmsgsz);
exit(1);

...

msgsz = ...
msgflg = ...

if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
perror("msgop: msgsnd failed");
...
msgsz = ...
msgtyp = first_on_queue;
msgflg = ...
```

if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
perror("msgop: msgrcv failed");

12Q) Write a program which shows the CLIENT/SERVER communication with Message Queues?

```
        #include<string.h>#include<time.h>
#include<sys/ipc.h>#include<sys/msg.h>
#include<sys/wait.h>#include<sys/errno.h>
 extern int errno;        // error NO.
#define MSGPERM 0600    // msg queue permission
#define MSGTXTLEN 128   // msg text length
int msgqid, rc;
int done;
struct msg_buf {
  long mtype;
  char mtext[MSGTXTLEN];
} msg;
int main(int argc,char **argv)
{
  // create a message queue. If here you get a invalid msgid and use it in msgsnd() or
msgrcg(), an Invalid Argument error will be returned.
  msgqid = msgget(IPC_PRIVATE, MSGPERM|IPC_CREAT|IPC_EXCL);
  if (msgqid < 0) {    perror(strerror(errno));
    printf("failed to create message queue with msgqid = %d\n", msgqid);
    return 1;  }  printf("message queue %d created\n",msgqid);
  // message to send
  msg.mtype = 1; // set the type of message
  sprintf (msg.mtext, "%s\n", "a text msg..."); /* setting the right time format by means
of ctime() */
  // send the message to queue
  rc = msgsnd(msgqid, &msg, sizeof(msg.mtext), 0); // the last param can be: 0, IPC_NOWAIT,
MSG_NOERROR, or IPC_NOWAIT|MSG_NOERROR.
  if (rc < 0) {    perror( strerror(errno) );
    printf("msgsnd failed, rc = %d\n", rc);
    return 1;
  }  // read the message from queue
  rc = msgrcv(msgqid, &msg, sizeof(msg.mtext), 0, 0);
  if (rc < 0) {    perror( strerror(errno) );
    printf("msgrcv failed, rc=%d\n", rc);
    return 1;  }
  printf("received msg: %s\n", msg.mtext);
  // remove the queue
  rc=msgctl(msgqid,IPC_RMID,NULL);
  if (rc < 0) {    perror( strerror(errno) );
    printf("msgctl (return queue) failed, rc=%d\n", rc);
    return 1;  }
  printf("message queue %d is gone\n",msgqid);
  return 0;
}
```

13Q) Write short notes on Kernel support for semaphores?

System V IPC semaphore objects each describe a semaphore array and Linux uses the semid_ds data structure to represent this. All of the semid_ds data structures in the system are pointed at by the semary, a vector of pointers. There are sem_nsems in each semaphore array, each one described by a semdata structure pointed at by sem_base. All of the processes that are allowed to manipulate the semaphore array of a System V IPC semaphore object may make system calls that perform operations on them. The system call can specify many operations and each operation is described by three inputs; the

semaphore index, the operation value and a set of flags. The semaphore index is an index into the semaphore array and the operation value is a numerical value that will be added to the current value of the semaphore. First Linux tests whether or not all of the operations would succeed. An operation will succeed if the operation value added to the semaphore's current value would be greater than zero or if both the operation value and the semaphore's current value are zero. If any of the semaphore operations would fail Linux may suspend the process but only if the operation flags have not requested that the system call is non-blocking. If the process is to be suspended then Linux must save the state of the semaphore operations to be performed and put the current process onto a wait queue. It does this by building a sem_queue data structure on the stack and filling it out. The new sem_queue data structure is put at the end of this semaphore object's wait queue (using the sem_pending and sem_pending_last pointers). The current process is put on the wait queue in the sem_queue data structure (sleeper) and the scheduler called to choose another process to run.

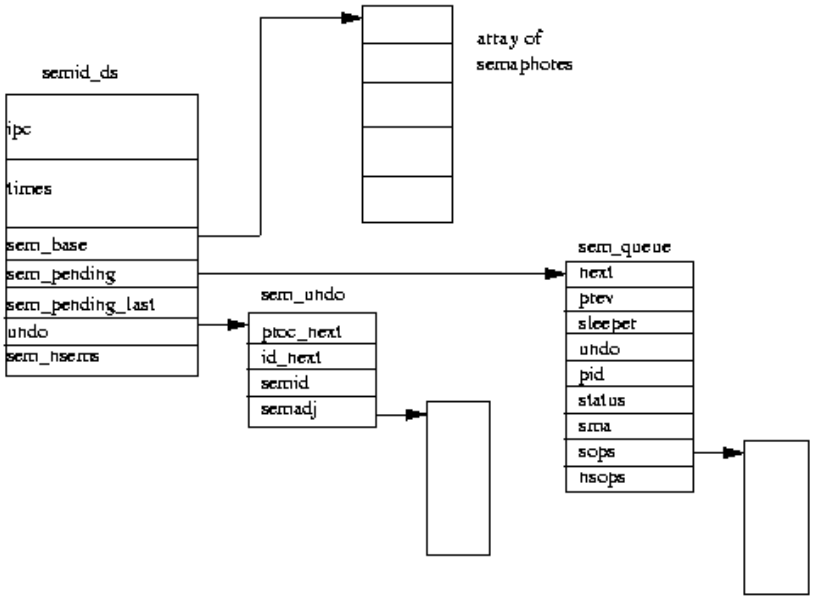
If all of the semaphore operations would have succeeded and the current process does not need to be suspended, Linux goes ahead and applies the operations to the appropriate members of the semaphore array. Now Linux must check that any waiting, suspended, processes may now apply Their semaphore operations. It looks at each member of the operations pending queue (sem_pending) in turn, testing to see if the semaphore operations will succeed this time. If they will then it removes the sem_queue data structure from the operations pending list and applies the semaphore operations to the semaphore array. It wakes up the sleeping process making it available to be restarted the next time the scheduler runs. Linux keeps looking through the pending list from the start until there is a pass where no semaphore operations can be applied and so no more processes can be woken.

There is a problem with semaphores, *deadlocks*. These occur when one process has altered the semaphores value as it enters a critical region but then fails to leave the critical region because it crashed or was killed. Linux protects against this by maintaining lists of adjustments to the semaphore arrays. The idea is that when these adjustments are applied, the semaphores will be put back to the state that they were in before the a process's set of semaphore operations were applied. These adjustments are kept in sem_undo data structures queued both on the semid_ds data structure and on the task_struct data structure for the processes using these semaphore arrays.

Each individual semaphore operation may request that an adjustment be maintained. Linux will maintain at most one sem_undo data structure per process for each semaphore array. If the requesting process does not have one, then one is created when it is needed. The new sem_undo data structure is queued both onto this process's task_struct data structure and onto the semaphore array's semid_ds data structure. As operations are applied to the semphores in the semaphore array the negation of the operation value is added to this semphore's entry in the adjustment array of this process's sem_undo data structure. So, if the operation value is 2, then -2 is added to the adjustment entry for this semaphore.

When processes are deleted, as they exit Linux works through their set of sem_undo data structures applying the adjustments to the semaphore arrays. If a semaphore set is deleted, the sem_undo data structures are left queued on the process's task_struct but the semaphore array identifier is made invalid. In this case the semaphore clean up code simply discards the sem_undo data structure.

Figure: System V IPC Semaphores



14Q) Explain the APIS for Semaphores?

The semget function creates a semaphore set or accesses an existing semaphore set.

```
#include <sys/sem.h>
int semget(key-t key, int nsems, int oflag);
```

Returns: nonnegative identifier if OK, -1 on error

Arguments:

Key –returns from ftok()

Nsem-number of semaphore sets are created

Oflag – Open flag for accessing is IPC_CREAT|0666

Program:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int main()
{
Int semid;
Semid=semget(15,2,IPC_CREAT|0644);
If(semid==-1)
{
Printf("semget error");
exit(1);
}
else
{
printf("semaphore set created :");
Printf(semid=%d",semid);
exit(1);
}
Return 0;
```

```
}
```

Semop()

Once a semaphore set is opened with semget, operations are performed on one or more of the semaphores in the set using the semop function.

```
#include <sys/sem.h>
int semop (int semid, struct sembuf *opsptr, size-t nops) ;
```

Returns: 0 if OK, -1 on error

Arugments:

Semid: returns from semget()

opsptr points to an array of the following structures:

```
struct sembuf {
short sem-num; /* semaphore number: 0, 1, ..., nsems-1 */
short sem-op; /* semaphore operation: <0, 0, >O */
short sem-flg; /* operation flags: 0, IPC-NOWAIT, SEM-UNDO */
};
```

Nops: specifies how many entries are in the array pointed to by opPtr

Program

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
Strct semid_ds sbuf[2]={{0,-1,SEM_UNDO||IPC_NOWAIT},1,0,0}};
Int main()
{
Int perms=S_IRWSU|S_IRWXG|S_IRWXO;
Int fd=semget(100,2,IPC_CREAT|S_IRWXG|S_IRWXO);
If(fd==-1)
Perrror("semget");
Exit(1);
If(semop(fd,sbuf,2==-1)
Perror("semop");
Return 0;
}
```

Semctl()

The semctl function performs various control operations on a semaphore.

```
#include <sys/sem.h>
int semctl (int semid, int semnum, int cmd, union semun arg  ) ;
```

Returns: nonnegative value if OK (see text), -1 on error

Arguments:

Semid –returns from semget()

Semnum –is a semaphore index where the next argument cmd specifies an operation to be performed.

The possible values of cmd are

IPC_STAT,IPC_SET,IPC_RMID,GETALL,SETALL,GETVAL,SETVAL,GETPID,GETNCNT,GETZCNT.

Union semun arg-is a union typed object that may be used to specify r retrieve the control data of one or more semaphores in the set.

```
Union semun
{
Int val; //a semaphore value
Struct semid_ds *buf; //control data of a semaphore set
Ushort *array; //an array of semaphore values
};
/* ** semrm.c -- removes a semaphore */
 #include <stdio.h>
 #include <stdlib.h>
 #include <errno.h>
#include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/sem.h>
 int main(void)
 {
 key_t key;
 int semid; if ((key = ftok("semdemo.c", 'J')) == -1)
 {
 perror("ftok");
 exit(1);
} /* grab the semaphore set created by seminit.c: */
 if ((semid = semget(key, 1, 0)) == -1)
{ perror("semget");
exit(1);
 } /* remove it: */
if (semctl(semid, 0, IPC_RMID) == -1)
{
 perror("semctl");
 exit(1); }
 return 0;
 }
```

15Q) Explain File Locking with Semaphores?

There are two of them. The first, *semdemo.c*, creates the semaphore if necessary, and performs some pretend file locking on it in a demo very much like that in the File Locking document. The second program, *semrm.c* is used to destroy the semaphore (again, **ipcrm** could be used to accomplish this.)

The idea is to run run *semdemo.c* in a few windows and see how all the processes interact. When you're done, use *semrm.c* to remove the semaphore. You could also try removing the semaphore while running *semdemo.c* just to see what kinds of errors are generated.

Here's *semdemo.c*, including a function named **initsem()** that gets around the semaphore race conditions, Stevens-style:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define MAX_RETRIES 10

union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};

/*
** initsem() -- more-than-inspired by W. Richard Stevens' UNIX Network
** Programming 2nd edition, volume 2, lockvsem.c, page 295.
*/
int initsem(key_t key, int nsems)  /* key from ftok() */
{
    int i;
    union semun arg;
    struct semid_ds buf;
    struct sembuf sb;
    int semid;

    semid = semget(key, nsems, IPC_CREAT | IPC_EXCL | 0666);

    if (semid >= 0) { /* we got it first */
        sb.sem_op = 1; sb.sem_flg = 0;
        arg.val = 1;

        printf("press return\n"); getchar();

        for(sb.sem_num = 0; sb.sem_num < nsems; sb.sem_num++) {
            /* do a semop() to "free" the semaphores. */
            /* this sets the sem_otime field, as needed below. */
            if (semop(semid, &sb, 1) == -1) {
                int e = errno;
                semctl(semid, 0, IPC_RMID); /* clean up */
                errno = e;
                return -1; /* error, check errno */
            }
        }

    } else if (errno == EEXIST) { /* someone else got it first */
        int ready = 0;

        semid = semget(key, nsems, 0); /* get the id */
        if (semid < 0) return semid; /* error, check errno */

        /* wait for other process to initialize the semaphore: */
        arg.buf = &buf;
        for(i = 0; i < MAX_RETRIES && !ready; i++) {
            semctl(semid, nsems-1, IPC_STAT, arg);
            if (arg.buf->sem_otime != 0) {
                ready = 1;
```

```
            } else {
                sleep(1);
            }
        }
        if (!ready) {
            errno = ETIME;
            return -1;
        }
    } else {
        return semid; /* error, check errno */
    }

    return semid;
}

int main(void)
{
    key_t key;
    int semid;
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;  /* set to allocate resource */
    sb.sem_flg = SEM_UNDO;

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

    /* grab the semaphore set created by seminit.c: */
    if ((semid = initsem(key, 1)) == -1) {
        perror("initsem");
        exit(1);
    }

    printf("Press return to lock: ");
    getchar();
    printf("Trying to lock...\n");

    if (semop(semid, &sb, 1) == -1) {
        perror("semop");
        exit(1);
    }

    printf("Locked.\n");
    printf("Press return to unlock: ");
    getchar();

    sb.sem_op = 1; /* free resource */
    if (semop(semid, &sb, 1) == -1) {
        perror("semop");
        exit(1);
    }

    printf("Unlocked\n");
```

```
    return 0;
}
```

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

int main(void)
{
    key_t key;
    int semid;
    union semun arg;

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

    /* grab the semaphore set created by seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    /* remove it: */
    if (semctl(semid, 0, IPC_RMID, arg) == -1) {
        perror("semctl");
        exit(1);
    }

    return 0;
}
```