OSD 19CS2106S
Session – 33
Lecture Notes

UNIX provides several different IPC mechanisms.

Interprocess interactions have several distinct purposes:

- **Data transfer —** One process may wish to send data to another process. The amount of data sent may vary from one byte to several megabytes.
- **Sharing data —** Multiple processes may wish to operate on shared data, such that if a process modifies the data, that change will be immediately visible to other processes sharing it.
- **Event notification —** A process may wish to notify another process or set of processes that some event has occurred. For instance, when a process terminates, it may need to inform its parent process. The receiver may be notified asynchronously, in which case its normal processing is interrupted. Alternatively, the receiver may choose to wait for the notification.
- **Resource sharing —** Although the kernel provides default semantics for resource allocation, they are not suitable for all applications. A set of cooperating processes may wish to define their own protocol for accessing specific resources. Such rules are usually implemented by a locking and synchronization scheme, which must be built on top of the basic set of primitives provided by the kernel.
- **Process control —** A process such as a debugger may wish to assume complete control over the execution of another (target) process. The controlling process may wish to intercept all traps and exceptions intended for the target and be notified of any change in the target's state.

IPC structure

The three types of IPC that we call XSI IPC - message queues, semaphores, and shared memory have many similarities. The XSI IPC functions are based closely on the System V IPC functions. message queues, semaphores, and shared memory are defined as XSI extensions in the Single UNIX Specification.

Identifiers and Keys

Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer identifier. To send or fetch a message to or from a message queue, for example, all we need know is the identifier for the queue. Unlike file descriptors, IPC identifiers are not small integers. Indeed, when a given IPC structure is created and then removed, the identifier associated with that structure continually increases until it reaches the maximum positive value for an integer, and then wraps around to 0.

The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a key that acts as an external name. Whenever an IPC structure is being created (by calling `msgget`, `semget`, or `shmget`), a key must be specified. The data type of this key is the primitive system data type `key_t`, which is often defined as a long integer in the header `<sys/types.h>`. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

1. The server can create a new IPC structure by specifying a key of `IPC_PRIVATE` and store the returned identifier somewhere (such as a file) for the client to obtain. The key `IPC_PRIVATE` guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.
   The `IPC_PRIVATE` key is also used in a parent child relationship. The parent creates a new IPC structure specifying `IPC_PRIVATE`, and the resulting identifier is then available to the child after the `fork`. The child can pass the identifier to a new program as an argument to one of the `exec` functions.
2. The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the `get` function (`msgget`, `semget`, or `shmget`) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.
3. The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. This key is then used in step 2. The only service provided by `ftok` is a way of generating a key from a pathname and project ID.
4. The three `get` functions (`msgget`, `semget`, and `shmget`) all have two similar arguments: a key and an integer flag. A new IPC structure is created (normally, by a server) if either key is `IPC_PRIVATE` or key is not currently associated with an IPC structure of the particular type and the `IPC_CREAT` bit of flag is specified. To reference an existing queue (normally done by a client), key must equal the key that was specified when the queue was created, and `IPC_CREAT` must not be specified.
5. Note that it's never possible to specify `IPC_PRIVATE` to reference an existing queue, since this special key value always creates a new queue. To reference an existing queue that was created with a key of `IPC_PRIVATE`, we must know the associated identifier and then use that identifier in the other IPC calls (such as `msgsnd` and `msgrcv`), bypassing the `get` function.
6. If we want to create a new IPC structure, making sure that we don't reference an existing one with the same identifier, we must specify a flag with both the `IPC_CREAT` and `IPC_EXCL` bits set. Doing this causes an error return of `EEXIST` if the IPC structure already exists. (This is similar to an `open` that specifies the `O_CREAT` and `O_EXCL` flags.)

Permission Structure

XSI IPC associates an `ipc_perm` structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

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

Each implementation includes additional members. See `<sys/ipc.h>` on your system for the complete definition.
All the fields are initialized when the IPC structure is created. At a later time, we can modify the `uid`, `gid`, and `mode` fields by calling `msgctl`, `semctl`, or `shmctl`. To change these values, the calling process must be either the creator of the IPC structure or the superuser.

Advantages and Disadvantages

A fundamental problem with XSI IPC is that the IPC structures are system wide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling `msgrcv` or `msgctl`, by someone executing the `ipcrm`(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.
Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions we described in File I/O. Almost a dozen new system calls (`msgget`, `semop`, `shmat`, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an `ls` command, we can't remove them with the `rm` command, and we can't change their permissions with the `chmod` command. Instead, two new commands `ipcs`(1) and `ipcrm`(1)were added.

Interprocess Communication

Processes have to communicate to exchange data and to synchronize operations. Several forms of interprocess communication are: pipes, named pipes and signals. But each one has some limitations.

System V IPC

The UNIX System V IPC package consists of three mechanisms:

1. Messages allow process to send formatted data streams to arbitrary processes.
2. Shared memory allows processes to share parts of their virtual address space.
3. Semaphores allow processes to synchronize execution.

The share common properties:

- Each mechanism contains a table whose entries describe all instances of the mechanism.
- Each entry contains a numeric *key*, which is its user-chosen name.
- Each mechanism contains a "get" system call to create a new entry or to retrieve an existing one, and parameters to the calls include a key and flags.
- The kernel uses the following formula to find the index into the table of data structures from the descriptor: `index = descriptor modulo (number of entries in table);`
- Each entry has a permissions structure that includes the user ID and group ID of the process that created the entry, a user and group ID set by the "control" system call (studied below), and a set of read-write-execute permissions for user, group, and others, similar to the file permission modes.
- Each entry contains other information such as the process ID of the last process to update the entry, and time of last access or update.
- Each mechanism contains a "control" system call to query status of an entry, to set status information, or to remove the entry from the system.

Messages

There are four system calls for messages:

1. *msgget* returns (and possibly creates) a message descriptor.
2. *msgctl* has options to set and return parameters associated with a message descriptor and an option to remove descriptors.
3. *msgsnd* sends a message.
4. *msgrcv* receives a message.

Syntax of *msgget*:

```
msgqid = msgget(key, flag);
```

where `msgqid` is the descriptor returned by the call, and *key* and *flag* have the semantics described above for the general "get" calls. The kernel stores messages on a linked list (queue) per descriptor, and it uses *msgqid* as an index into an array of message queue headers. The queue structure contains the following fields, in addition to the common fields:

- Pointers to the first and last messages on a linked list.

- The number of messages and total number of data bytes on the linked list.
- The maximum number of bytes of data that can be on the linked list.
- The process IDs of the last processes to send and receive messages.
- Time stamps of the last *msgsnd*, *msgrcv*, and *msgctl* operations.

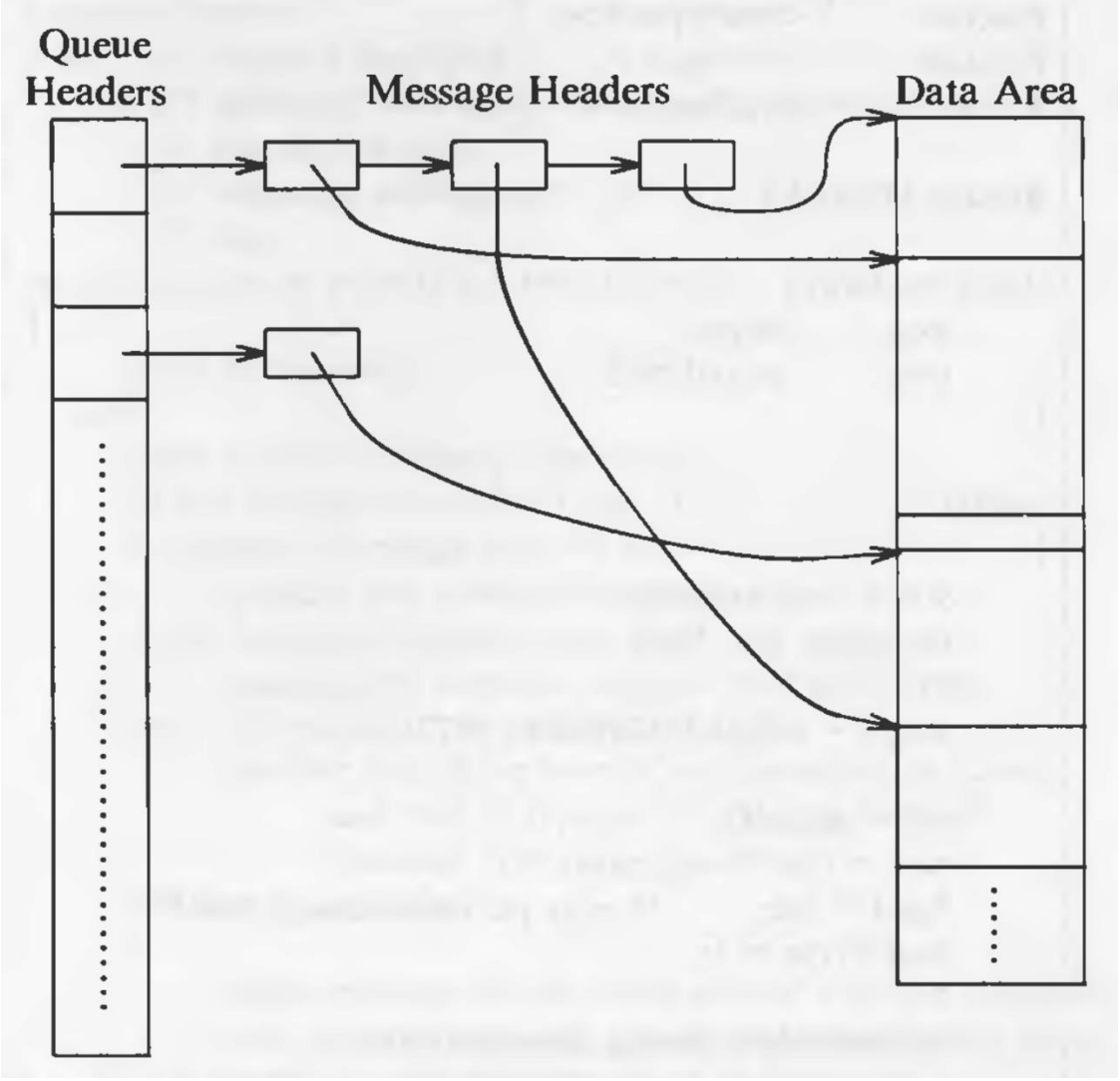
Syntax of *msgsnd*:

```
msgsnd(msgqid, msg, count, flag);
```

`flag` describes the action the kernel should take if it runs out of internal buffer space. The algorithm is given below:

```
/*  Algorithm: msgsnd
 *  Input: message queue descriptor
 *         address of message structure
 *         size of message
 *         flags
 *  Output: number of bytes send
 */
{
        check legality of descriptor, permissions;
        while (not enough space to store message)
        {
                if (flags specify not to wait)
                        return;
                sleep(event: enough space is available);
        }
        get message header;
        read message text from user space to kernel;
        adjust data structures: enqueue message header,
                message header points to data, counts,
                time stamps, process ID;
        wakeup all processes waiting to read message from queue;
}
```

The diagram shows the structure of message queues:



Syntax for *msgrcv*:

```
count = msgrcv(id, msg, maxcount, type, flag);
```

The algorithm is given below:

```
/*  Algorithm: msgrcv
 *  Input: message descriptor
 *         address of data array for incoming message
 *         size of data array
 *         requested message type
 *         flags
 *  Output: number of bytes in returned message
 */

{
        check permissions;
 loop:
        check legality of message descriptor;
        // find message to return to user
        if (requested message type == 0)
                consider first message on queue;
        else if (requested message type > 0)
                consider first message on queue with given type;
        else                    // requested message type < 0
                consider first of the lowest typed messages on queue,
                        such that its type is <= absolute value of requested type;
        if (there is a message)
        {
                adjust message size or return error if user size too small;
                copy message type, text from kernel space to user space;
                unlink message from queue;
                return;
        }
        // no message
        if (flags specify not to sleep)
                return with error;
        sleep(event: message arrives on queue);
        goto loop;
}
```

If processes were waiting to send messages because there was no room on the list, the kernel awakens them after it removes a message from the message queue. If a message is bigger than *maxcount*, the kernel returns an error for the system call leaves the message on the queue. If a process ignores the size constraints (*MSG_NOERROR* bit is set in *flag*), the kernel truncates the message, returns the requested number of bytes, and removes the entire message from the list.

If the *type* is a positive integer, the kernel returns the first message of the given type. If it is a negative, the kernel finds the lowest type of all message on the queue, provided it is less than or equal to the absolute value of the *type*, and returns the first message of that type. For example, if a queue contains three messages whose types are 3, 1, and 2, respectively, and a user requests a message with type -2, the kernel returns the message of type 1.

The syntax of *msgctl*:

```
msgctl(id, cmd, mstatbuf);
```

where `cmd` specifies the type of command, and `mstatbuf` is the address of a user data structure that will contain control parameters or the results of a query.

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.

The Single UNIX Specification includes an alternate IPC message queue implementation in the message-passing option of its real-time extensions. We do not cover the real-time extensions in this text.

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

`IPC_STAT` Fetch the `msqid_ds` structure for this queue, storing it in the structure pointed to by buf.

`IPC_SET` Copy the following fields from the structure pointed to by buf to the `msqid_ds` structure associated with this queue: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges. Only the superuser can increase the value of `msg_qbytes`.

`IPC_RMID` Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of `EIDRM` on its next attempted operation on the queue. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges.

POSIX:XSI Interprocess Communication

The POSIX interprocess communication (IPC) is part of the POSIX:XSI Extension and has its origin in UNIX System V interprocess communication. IPC, which includes message queues, semaphore sets and shared memory, provides mechanisms for sharing information among processes on the same system. These three communication mechanisms have a similar structure, and this chapter emphasizes the common elements of their use. Table given below summarizes the POSIX:XSI interprocess communication functions.

POSIX:XSI interprocess communication functions.

mechanism	POSIX function	meaning
message queues	`msgctl`	control
	`msgget`	create or access
	`msgrcv`	receive message
	`msgsnd`	send message
semaphores	`semctl`	control
	`semget`	create or access
	`semop`	execute operation (wait or post)
shared memory	`shmat`	attach memory to process
	`shmctl`	control
	`shmdt`	detach memory from process
	`shmget`	create and initialize or access

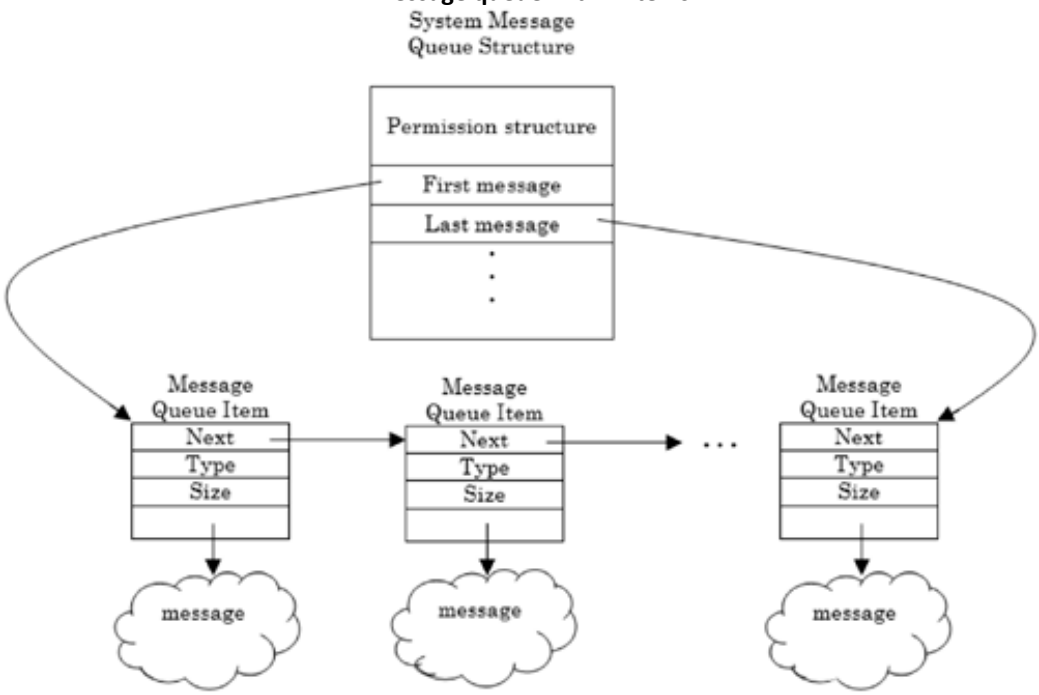
The designers of UNIX found the types of interprocess communications that could be implemented using signals and pipes to be restrictive. To increase the flexibility and range of interprocess communication, supplementary communication facilities were added. These facilities, added with the release of System V in the 1970s, are grouped under the heading IPC (Interprocess Communication). In brief, these are

- Message queues— Information to be communicated is placed in a predefined message structure. The process generating the message specifies its type and places the message in a system-maintained message queue. Processes accessing the message queue can use the message type to selectively read messages of specific types in a first in first out (FIFO) manner. Message queues provide the user with a means of asynchronously multiplexing data from multiple processes.
- Semaphores— Semaphores are system-implemented data structures used to communicate small amounts of data between processes. Most often, semaphores are used for process synchronization.
- Shared memory— Information is communicated by accessing shared process data space. This is the fastest method of interprocess communication. Shared memory allows participating processes to randomly access a shared memory segment. Semaphores are often used to synchronize the access to the shared memory segments.

All three of these facilities can be used by related and unrelated processes, but these processes must be on the same system (machine).
Like a file, an IPC resource must be generated before it can be used. Each IPC resource has a creator, owner, and access permissions. These attributes, established when the IPC is created, can be modified using the proper system calls. At a system level, information about the IPC facilities supported by the system can be obtained with the `ipcs` command.

Summary of the System V IPC Calls.			
Functionality	Message Queue	System Call Semaphore	Shared Memory
Allocate an IPC resource; gain access to an existing IPC resource.	`msgget`	`semget`	`shmget`
Control an IPC resource: obtain/modify status information, remove the resource.	`msgctl`	`semctl`	`shmctl`
IPC operations: send/receive messages,perform semaphore operations, attach/free a shared memory segment.	`msgsnd` `msgrcv`	`semop`	`shmat` `shmdt`

A message queue with N items



Message Queues IPC

Where's my queue?

First of all, you want to connect to a queue, or create it if it doesn't exist. The call to accomplish this is the **`msgget()`** system call: **`int msgget(key_t key, int msgflg);`**

`msgget()` returns the message queue ID on success, or `-1` on failure (and it sets *`errno`*, of course.)

The first arguments, *`key`* is a system-wide unique identifier describing the queue you want to connect to (or create). Every other process that wants to connect to this queue will have to use the same *`key`*.

The other argument, *`msgflg`* tells **`msgget()`** what to do with queue in question. To create a queue, this field must be set equal to `IPC_CREAT` bit-wise OR'd with the permissions for this queue. (The queue permissions are the same as standard file permissions—queues take on the user-id and group-id of the program that created them.)

A sample call is given in the following section.

"Are you the Key Master?"

`key_t ftok(const char *path, int id);`

The **`ftok()`** function uses information about the named file (like inode number, etc.) and the *`id`* to generate a probably-unique *`key`* for **`msgget()`**. Programs that want to use the same queue must generate the same *`key`*, so they must pass the same parameters to **`ftok()`**.

Finally, it's time to make the call:

```
#include <sys/msg.h>
key = ftok("/home/vishnu/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

In the above example, I set the permissions on the queue to `666` (or `rw-rw-rw-`). And now we have *`msqid`* which will be used to send and receive messages from the queue.

Sending to the queue

Once you've connected to the message queue using **`msgget()`**, you are ready to send and receive messages. First, the sending: Each message is made up of two parts, which are defined in the template structure `struct msgbuf`, as defined in *`sys/msg.h`*:

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

The field *mtype* is used later when retrieving messages from the queue, and can be set to any positive number. *mtext* is the data this will be added to the queue.
You can use any structure you want to put messages on the queue, as long as the first element is a long. For instance, we could use this structure to store all kinds of goodies:

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    struct pirate_info {
        char name[30];
        char ship_type;
        int notoriety;
        int cruelty;
        int booty_value;
    } info;
};
```

msgsnd():

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

msqid is the message queue identifier returned by **msgget()**. The pointer *msgp* is a pointer to the data you want to put on the queue. *msgsz* is the size in bytes of the data to add to the queue (not counting the size of the *mtype* member). Finally, *msgflg* allows you to set some optional flag parameters, which we'll ignore for now by setting it to 0. When to get the size of the data to send, just take the size of the second field:

```
struct cheese_msgbuf {
    long mtype;
    char name[20];
};
/* calculate the size of the data to send: */
struct cheese_msgbuf mbuf;
int size;
size = sizeof mbuf.name;

#include <sys/msg.h>
#include <stddef.h>
key_t key;
int msqid;
struct pirate_msgbuf pmb = {2, { "vishnu", 'S', 80, 10, 12035 } };
key = ftok("/home/vishnu/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
/* stick him on the queue */
/* struct pirate_info is the sub-structure */
msgsnd(msqid, &pmb, sizeof(struct pirate_info), 0);
```

Receiving from the queue

A call to **msgrcv()** that would do it looks something like this:

```
#include <sys/msg.h>
#include <stddef.h>
key_t key;
int msqid;
struct pirate_msgbuf pmb; /* where vishnu is to be kept */
key = ftok("/home/vishnu/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
/* get him off the queue! */
msgrcv(msqid, &pmb, sizeof(struct pirate_info), 2, 0);
```

There is something new to note in the **msgrcv()** call: the 2! What does it mean? Here's the synopsis of the call:

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

The 2 we specified in the call is the requested *msgtyp*. Recall that we set the *mtype* arbitrarily to 2 in the **msgsnd()** section of this document, so that will be the one that is retrieved from the queue.
Actually, the behavior of **msgrcv()** can be modified drastically by choosing a *msgtyp* that is positive, negative, or zero:

msgtyp	**Effect on msgrcv()**
Zero	Retrieve the next message on the queue, regardless of its *mtype*.
Positive	Get the next message with an *mtype* ***equal to*** the specified *msgtyp*.
Negative	Retrieve the first message on the queue whose *mtype* field is less than or equal to the absolute value of the *msgtyp* argument.

So, what will often be the case is that you'll simply want the next message on the queue, no matter what *mtype* it is. As such, you'd set the *msgtyp* parameter to 0.

Destroying a message queue

There comes a time when you have to destroy a message queue. There are two ways:

1. Use the Unix command **ipcs** to get a list of defined message queues, then use the command **ipcrm** to delete the queue.

2. Write a program to do it for you.

Often, the latter choice is the most appropriate, since you might want your program to clean up the queue at some time or another. To do this requires the introduction of another function: **`msgctl()`**.

The synopsis of **`msgctl()`** is:

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

Of course, *`msqid`* is the queue identifier obtained from **`msgget()`**. The important argument is *`cmd`* which tells **`msgctl()`** how to behave. It can be a variety of things, but we're only going to talk about `IPC_RMID`, which is used to remove the message queue. The *`buf`* argument can be set to `NULL` for the purposes of `IPC_RMID`.

Say that we have the queue we created above to hold the pirates. You can destroy that queue by issuing the following call:

```
#include <sys/msg.h>
.
msgctl(msqid, IPC_RMID, NULL);
```

And the message queue is no more.

`/*** kirk.c --demonstrate Message Queues IPC - adds messages to the message queue */`

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

#include  <errno.h>
#include  <string.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;
    if ((key = ftok("kirk.c", 'B')) == -1) {

         perror("ftok");
         exit(1);
    }

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
         perror("msgget");

         exit(1);
    }
    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */
    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {

      int len = strlen(buf.mtext);
      /* ditch newline at end, if it exists */
      if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

      if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
          perror("msgsnd");

   }
```

```
   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl");

      exit(1);
   }

  return 0;
}
/*[vishnu@mannava ~]$ cc kirk.c -o kirk

[vishnu@mannava ~]$ ./kirk
Enter lines of text, ^D to quit:

hello how are you
type ipcs -q and check
third line

hello
*/

/*
[vishnu@mannava ~]$ ipcs -q
------ Message Queues --------

key        msqid      owner      perms      used-bytes   messages
0x42004205 0          vishnu     644        43           2
```

/*spock.c - demonstrate Message Queues IPC - retrieves messages from the message queue */

```
#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;

    if ((key = ftok("kirk.c", 'B')) == -1) {   /* same key as kirk.c */
         perror("ftok");
         exit(1);

    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue
         */ perror("msgget");
         exit(1);
    }

    printf("spock: ready to receive messages,
    captain.\n"); for(;;) { /* Spock never quits! */
```

```
        if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1)
             { perror("msgrcv");
             exit(1);

        }
        printf("spock: \"%s\"\n", buf.mtext);

    }
    return 0;
}

/*[vishnu@mannava ~]$ ./spock

spock: ready to receive messages, captain. spock: "hello how are you"
spock: "type ipcs -q and check"
spock: "third line"
spock: "hello"
msgrcv: Identifier
removed */

[vishnu@team-osd ~]$ ipcs -l

------ Messages Limits --------
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 1940588
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 100
semaphore max value = 32767

[vishnu@team-osd ~]$ ipcs -u

------ Messages Status --------
allocated queues = 9
used headers = 4
used space = 378 bytes

------ Shared Memory Status --------
segments allocated 96
pages allocated 14011
pages resident  3191
pages swapped   6271
Swap performance: 0 attempts     0 successes

------ Semaphore Status --------
used arrays = 0
allocated semaphores = 0

[vishnu@team-osd ~]$ ipcs -q -c

------ Message Queues Creators/Owners --------
msqid      perms      cuid       cgid       uid        gid
12         644        osd-kiran3168 osd-kiran3168 osd-kiran3168 osd-kiran3168
13         644        vishnu     vishnu     vishnu     vishnu
```