UNIX Programming(18CS56)
Module 4

Changing User IDs and Group IDs, Interpreter Files, system Function, Process Accounting, User Identification, Process Times, I/O Redirection.

Overview of IPC Methods, Pipes, popen, pclose Functions, Coprocesses, FIFOs, System V IPC, Message Queues, Semaphores.

Shared Memory, Client-Server Properties, Stream Pipes, Passing File Descriptors, An Open Server-Version 1, Client-Server Connection Functions.

CHANGING USER IDs AND GROUP IDs

- When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID **to an ID that has the appropriate privilege or access.**
- Similarly, when our programs **need to lower their privileges or prevent access to** certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Both return: 0 if OK, 1 on error

There are **rules** for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

- If the process has **superuser privileges**, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- If the process **does not have superuser privileges**, but uid equals either the real user ID or the saved set-user-ID, setuid sets only the effective user ID to uid. The real user ID and the saved set-user-ID are not changed.
- If neither of these two conditions is true, errno is set to EPERM, and 1 is returned.

We can make a few statements about the three user IDs that the kernel maintains.

- Only a **superuser process can change the real user ID**. Normally, the real user ID is set by the **login(1)** program when we log in and never changes. Because login is a superuser process, it sets all three user IDs when it calls setuid.
- The **effective user ID is set by the exec functions only if the set-user-ID bit is set for the program file.** If the set-user-ID bit is not set, the exec functions leave the effective user ID as its current value. We can call setuid at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
- The saved set-user-ID is copied from the effective user ID by **exec**. If the file's set-user-ID bit is set, this copy is saved after exec stores the effective user ID from the file's user ID.

is saved after exec stores the effective user ID from the file's user ID.

ID	exec		setuid(uid)	
				user
real user ID	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program file	set to uid	set to uid
saved set-user ID	copied from effective user ID	copied from effective user ID	set to uid	unchanged

setreuid and setregid Functions

Swapping of the real user ID and the effective user ID with the setreuid function.

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Both return : 0 if OK, -1 on error

We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged.

The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID.

This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user- ID operations

seteuid and setegid functions :

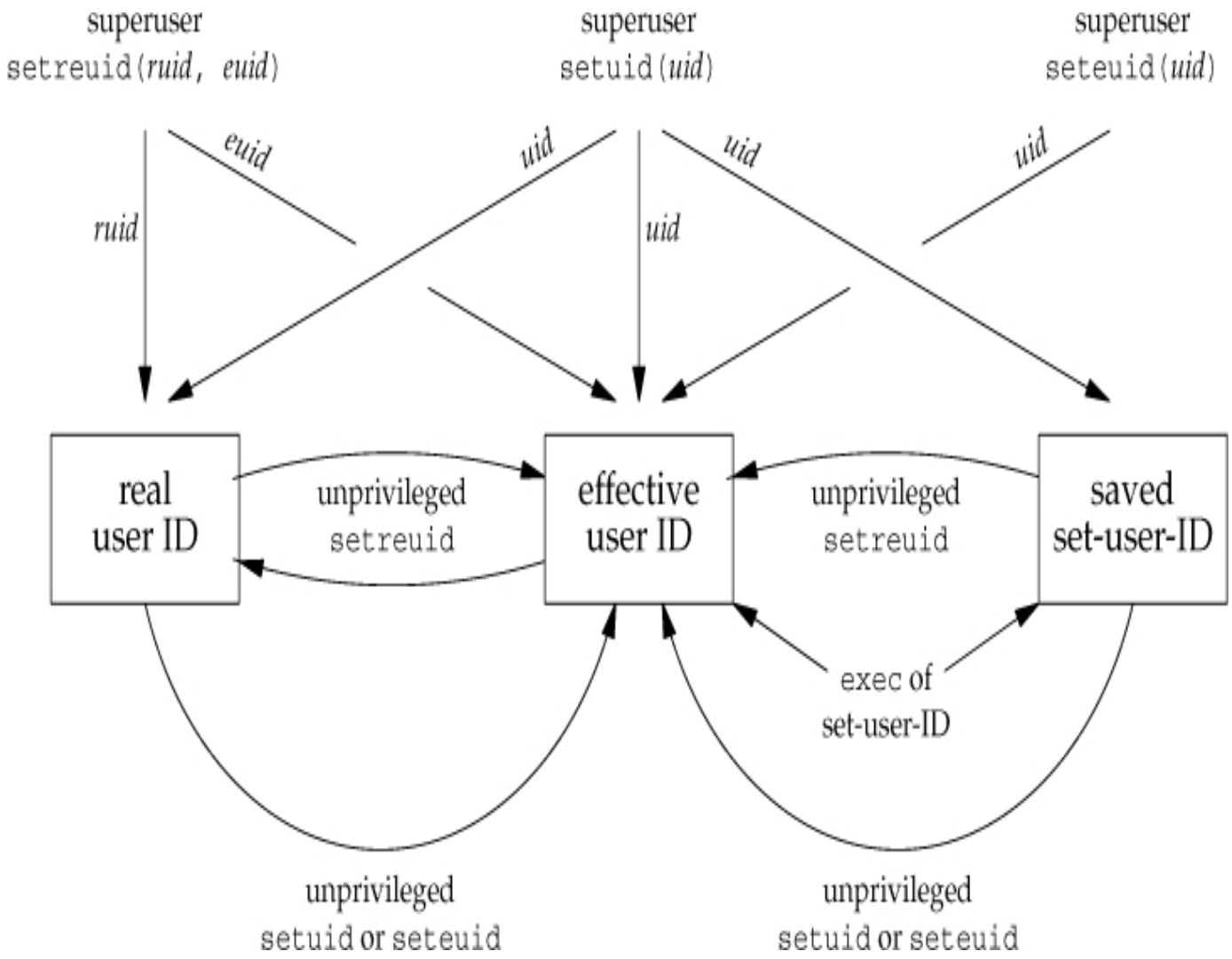
- POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Both return : 0 if OK, 1 on error

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID.

For a privileged user, only the effective user ID is set to uid. (This differs from the setuid function, which changes all three user IDs.)


superuser
setreuid(ruid, euid)
superuser
setuid(uid)
superuser
seteuid(uid)
euid
ruid
uid
uid
uid
uid
real
user ID
unprivileged
setreuid
effective
user ID
unprivileged
setreuid
saved
set-user-ID
exec of
set-user-ID
unprivileged
setuid or seteuid
unprivileged
setuid or seteuid

system FUNCTION

```
#include <stdlib.h>

int system(const char *cmdstring);
```

If cmdstring is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, system is always available.
Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

1. If either the fork fails or waitpid returns an error other than EINTR, system returns 1 with errno set to indicate the error.
2. If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
3. Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from system is the termination status of the shell, in the format specified for waitpid.

Program: The system function, without signal handling

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

Int system(const char *cmdstring)    /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                                /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Program: Calling the system function

```
#include "apue.h"
#include <sys/wait.h>

Int main(void)
{
    int      status;

    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

   if ((status = system("nosuchcommand")) < 0)
       err_sys("system() error");
   pr_exit(status);

   if ((status = system("who; exit 44")) < 0)
       err_sys("system() error");
   pr_exit(status);

   exit(0);
}
```

PROCESS ACCOUNTING

- Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates.
- These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.
- A superuser executes accton with a pathname argument to enable accounting.
- The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.
- The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork.
- Each accounting record is written when the process terminates.
- This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.
- The accounting records correspond to processes, not programs.
- A new record is initialized by the kernel for the child after a fork, not when a new program is executed.

The structure of the accounting records is defined in the header <sys/acct.h> and looks something like

```
struct   acct
{
 char    ac_flag;       /* flag   */
 char    ac_stat;       /* termination status (signal & core flag only) */
                        /* (Solaris only) */
 uid_t   ac_uid;        /* real user ID */
 gid_t   ac_gid;        /* real group ID */
 dev_t   ac_tty;        /* controlling terminal */
  time_t ac_btime;      /* starting calendar time */
  comp_t ac_utime;      /* user CPU time (clock ticks) */
  comp_t ac_stime;      /* system CPU time (clock ticks) */
  comp_t ac_etime;      /* elapsed time (clock ticks) */
  comp_t ac_mem;        /* average memory usage */
  comp_t ac_io;         /* bytes transferred (by read and write) */
                        /* "blocks" on BSD systems */
  comp_t ac_rw;         /* blocks read or written */
                        /* (not present on BSD systems) */
 char    ac_comm[8];    /* command name: [8] for Solaris, */
                        /* [10] for Mac OS X, [16] for FreeBSD, and */
                        /* [17] for Linux */
};
```

Values for ac_flag from accounting record

Values for `ac_flag` from accounting record

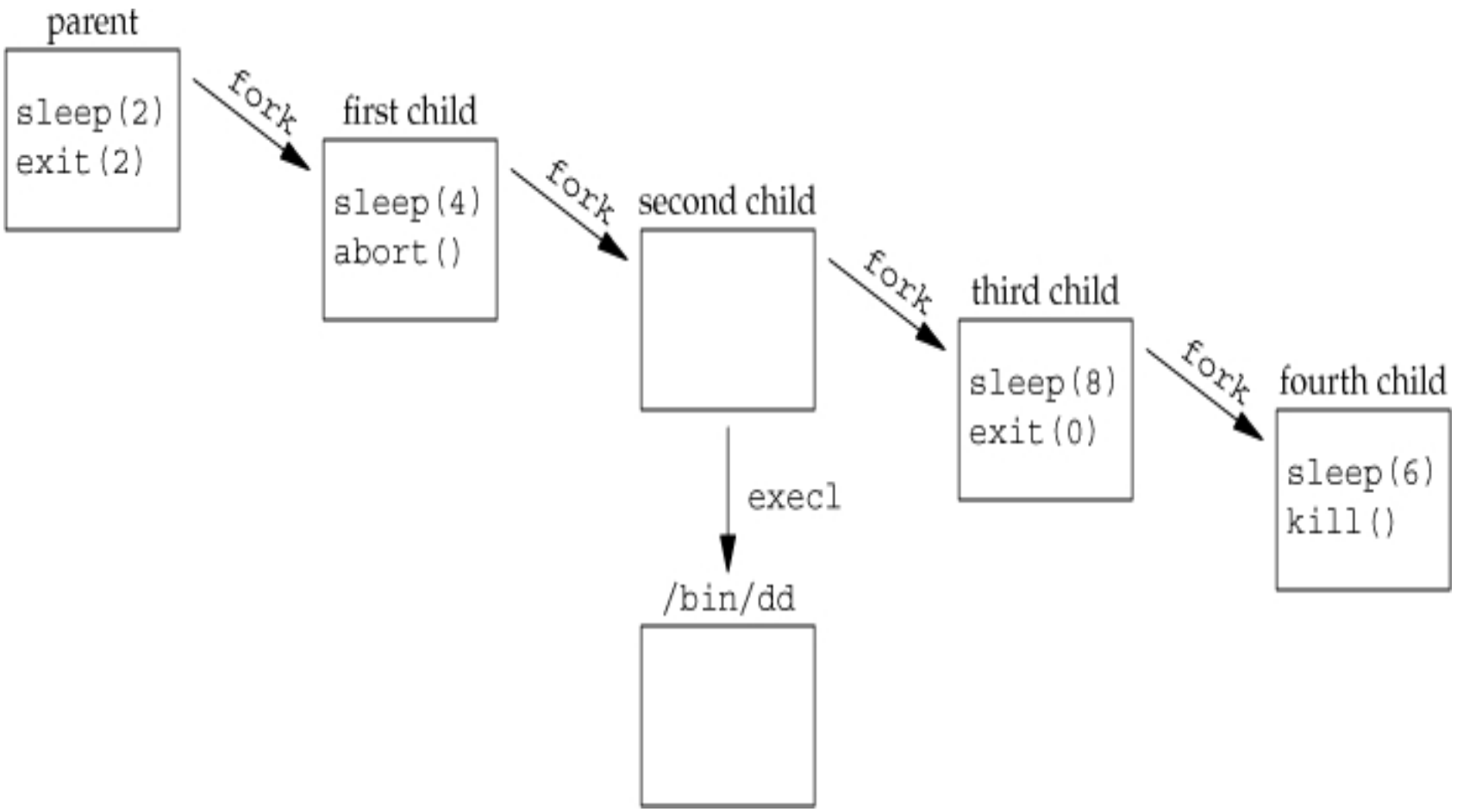
ac_flag	Description
AFORK	process is the result of `fork`, but never called `exec`
ASU	process used superuser privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Program to generate accounting data

```
#include "apue.h"

Int main(void)
{
        pid_t pid;
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid != 0)
        { /* parent */
                sleep(2);
                exit(2); /* terminate with exit status 2 */
        } /* first child */
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid != 0)
        {
                sleep(4);
                abort(); /* terminate with core dump */
        }
```

```
if ((pid = fork()) < 0)
        err_sys("fork error");
        else if (pid != 0)
        {
                execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
                exit(7); /* shouldn't get here */
        } /* third child */
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid != 0)
        {
                sleep(8);
                exit(0); /* normal exit */
        } /* fourth child */
        sleep(6);
        kill(getpid(), SIGKILL); /* terminate w/signal, no core dump */
        exit(6); /* shouldn't get here */
        }
```

Process structure for accounting example



USER IDENTIFICATION

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call getpwuid(getuid()), but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in and the getlogin function provides a way to fetch that login name.

```
#include <unistd.h>
char *getlogin(void);
```

Returns : pointer to string giving login name if OK, NULL on error

This function can fail if the process is not attached to a terminal that a user logged in to.

PROCESS TIMES

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

clock_t times(struct tms *buf);
```

- Returns: elapsed wall clock time in clock ticks if OK, 1 on error
- This function fills in the tms structure pointed to by buf:
- **struct tms {**
- **clock_t tms_utime; /* user CPU time */ clock_t tms_stime; /* system CPU time */**
- **clock_t tms_cutime; /* user CPU time, terminated children */ clock_t tms_cstime; /* system CPU time, terminated children */**
- **};**

INTERPROCESS COMMUNICATION

- **INTRODUCTION**
- IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems. The various forms of IPC that are supported on a UNIX system are as follows :

1) Half duplex Pipes.
2) FIFO's
3) Full duplex Pipes.
4) Named full duplex Pipes.
5) Message queues.
6) Shared memory.
7) Semaphores.
8) Sockets.
9) STREAMS.

- The first seven forms of IPC are usually restricted to IPC between processes on the same host. The final two i.e. Sockets and STREAMS are the only two that are generally supported for IPC between processes on different hosts.

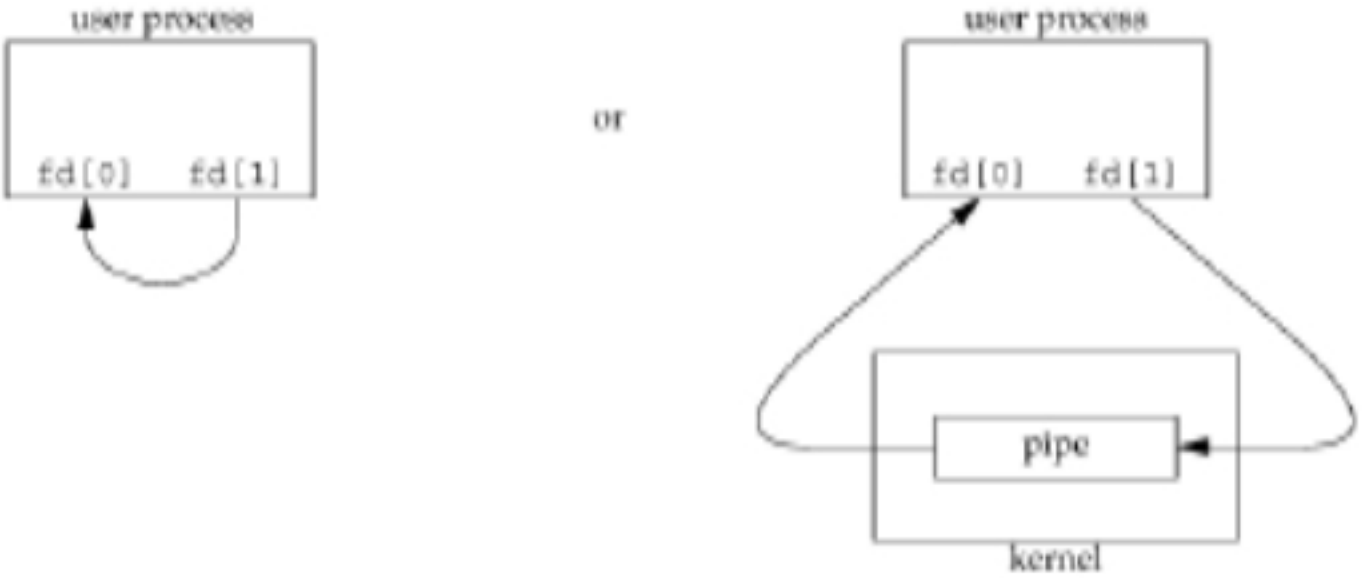
PIPES

- 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

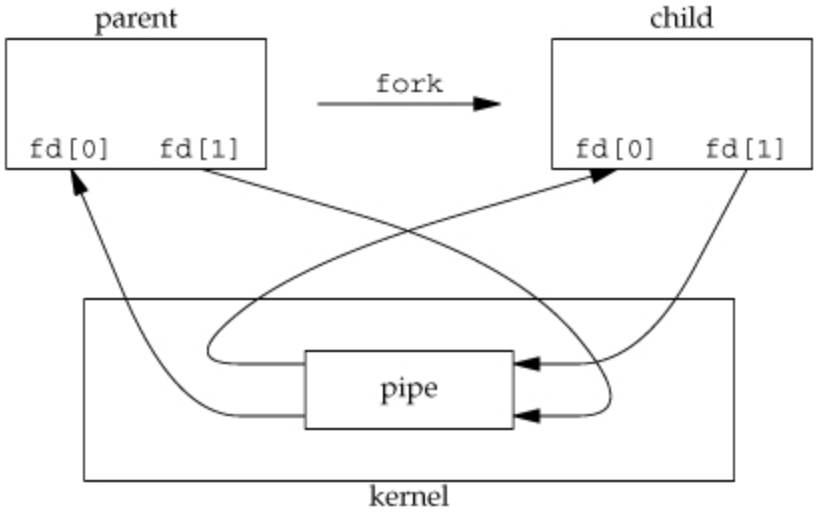
```
#include <unistd.h>

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

Returns: 0 if OK, 1 on error.

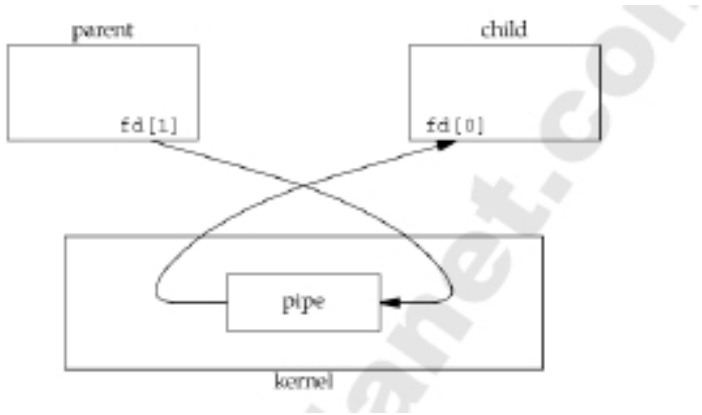
- Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].
- Two ways to picture a half-duplex pipe are shown in Figure. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.



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. As shown in this scenario.



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]). Following Figure shows the resulting arrangement of descriptors.



- 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.

PROGRAM: shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"
 int main(void)
  { int n; int fd[2]; pid_t pid; char line[MAXLINE];
  if (pipe(fd) < 0)
            err_sys("pipe error");
  if ((pid = fork()) < 0)
            { err_sys("fork error"); }
  else if (pid > 0)
  { /* parent */
            close(fd[0]);
            write(fd[1], "hello world\n", 12); }
   else { /* child */
            close(fd[1]);
            n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line, n);
            }
exit(0); }
```

popen and pclose Functions

- 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 work of : creating a pipe, forking 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>

FILE *popen(const char *cmdstring, const char *type);
```

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

- 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

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

fp = popen(cmdstring, "r")

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

fp = popen(cmdstring, "w")

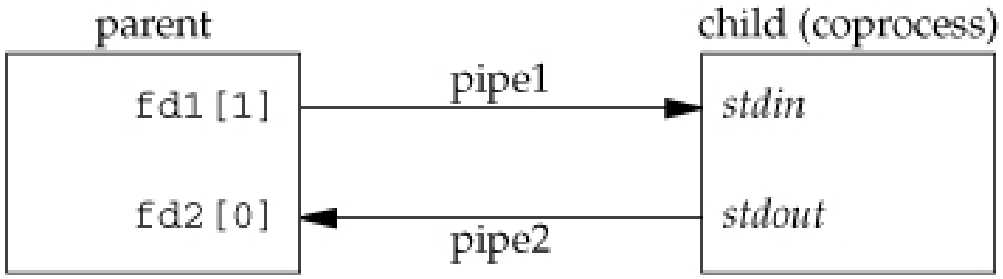
sh -c *cmdstring*

This means that the shell expands any of its special characters in *cmdstring*. This allows us to say, for example,

fp = popen("ls *.c", "r");

COPROCESSES

- A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines.
- A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output.
- A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.
- The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess. Figure 15.16 shows this arrangement.



Driving a coprocess by writing its standard input and reading its standard output

- 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
```

Program: Simple filter to add two numbers

```
#include "apue.h"
int main(void)
{
int n, int1, int2; char line[MAXLINE];
while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0)
{
      line[n] = 0; /* null terminate */
      if (sscanf(line, "%d%d", &int1, &int2) == 2)
      { sprintf(line, "%d\n", int1 + int2);
      n = strlen(line);
      if (write(STDOUT_FILENO, line, n) != n)
      err_sys("write error");
} else
{ if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
      err_sys("write error");
}
} exit(0); }
```

FIFOs

- 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

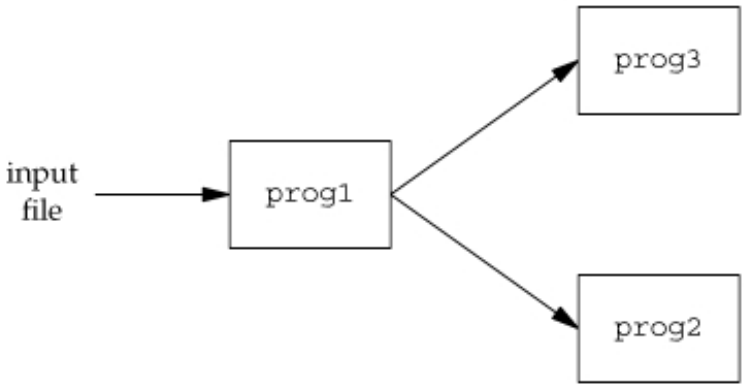
Once we have used mkfifo to create a FIFO, we open it using open.

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.

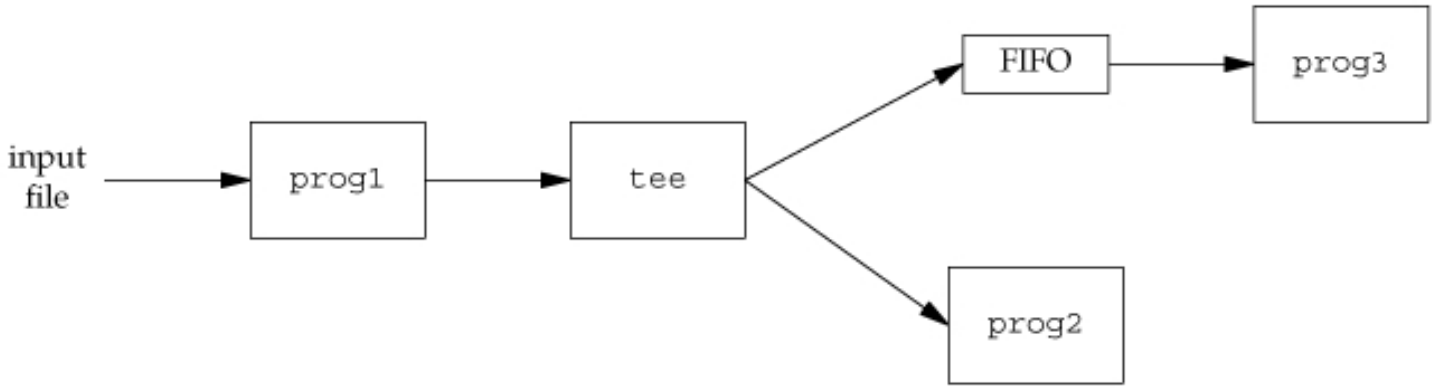
- There are two uses for FIFOs.
- FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
- FIFOs are used as rendezvous points in client-server applications to pass data between the clients and the servers.
- **Example Using FIFOs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



- 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
```

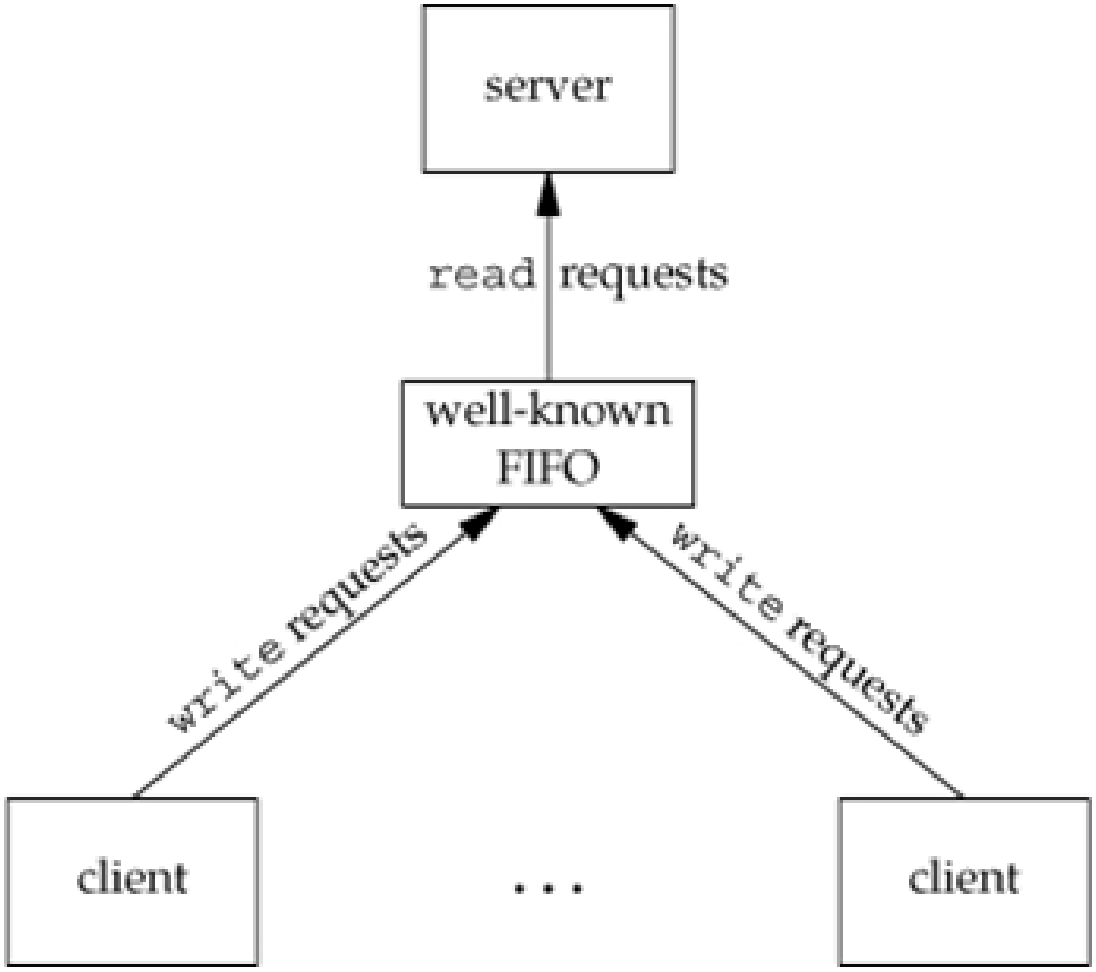
- We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2. Figure shows the process arrangement.



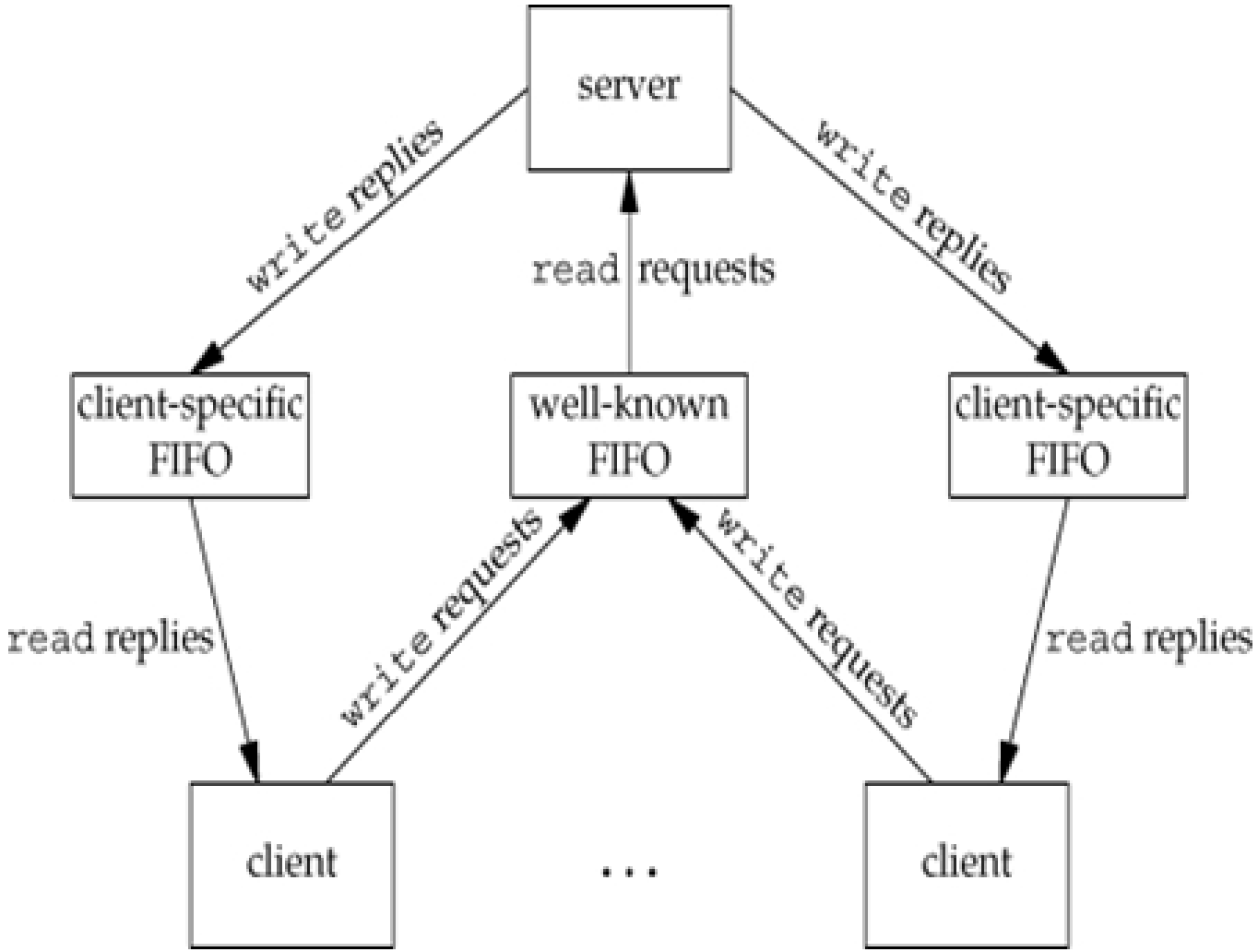
Example Client-Server Communication Using a FIFO

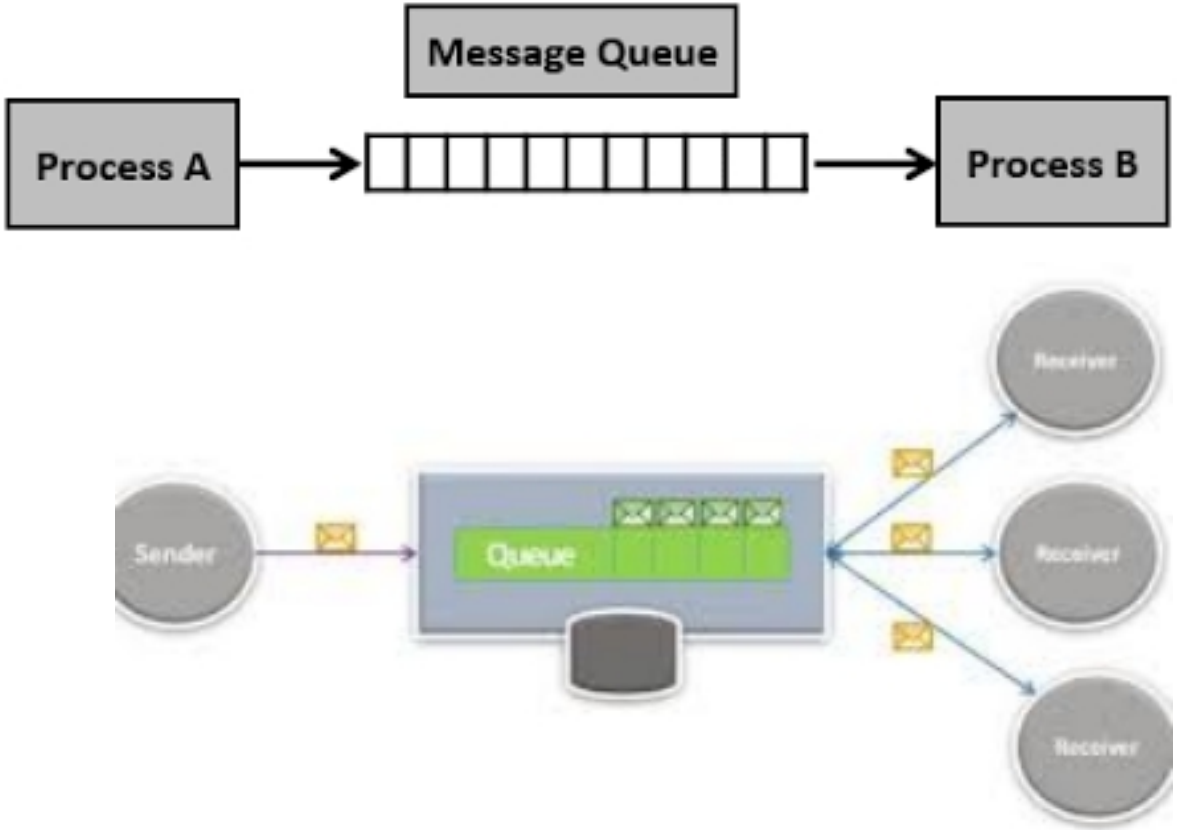
- FIFO's can be used to send data between a client and a server.
- If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- 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's process ID.
- 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




Message Queue
Process A
Process B
Sender
Queue
Receiver
Receiver
Receiver

MESSAGE QUEUES

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
- 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,
  - actual data bytes (corresponding to the length),
  - All this is specified to msgsnd when the message is added to a queue.
- Messages are fetched from a queue by msgrcv.

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 */
        .
        .
        .
};
```

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

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 queue.

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

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

Returns: message queue ID if OK, 1 on error

- The first argument, key, recognizes the message queue. The key can be either an **arbitrary value** or one that can be derived from the library function **ftok().**
- The second argument, flag, specifies the required message queue flag/s such as **IPC_CREAT (creating message queue if not exists) or IPC_EXCL** (Used with IPC_CREAT to create the message queue and the call fails, if the message queue already exists). Need to pass the permissions as well.

- 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>

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

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.

This system call sends/appends a message into the message queue (System V). Following arguments need to be passed −

- The first argument, msqid, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, ptr, is the pointer to the message, sent to the caller, defined in the structure of the following form −

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

- The third argument, mbytes, is the size of message (the message should end with a null character)
- The fourth argument, flag, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in queue or MSG_NOERROR (truncates message text, if more than msgsz bytes)

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);
```

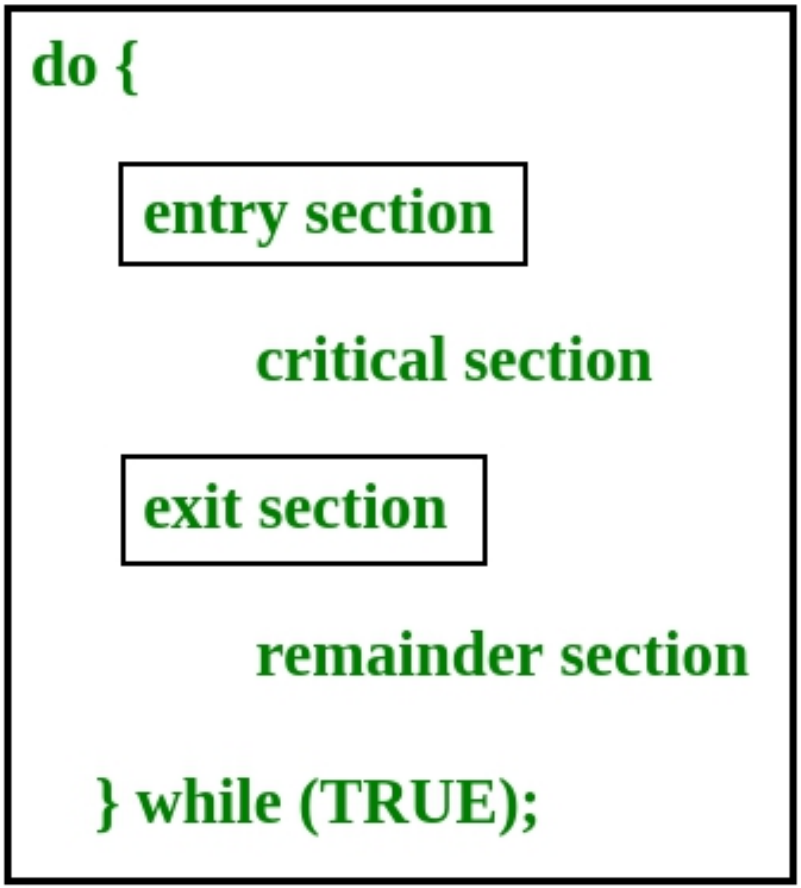
Returns: size of data portion of message if OK, 1 on error.

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.

Critical Section Problem

- The critical section is a code segment where the shared variables can be accessed.
- An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time.
- All the other processes have to wait to execute in their critical sections.



SEMAPHORES

- 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:

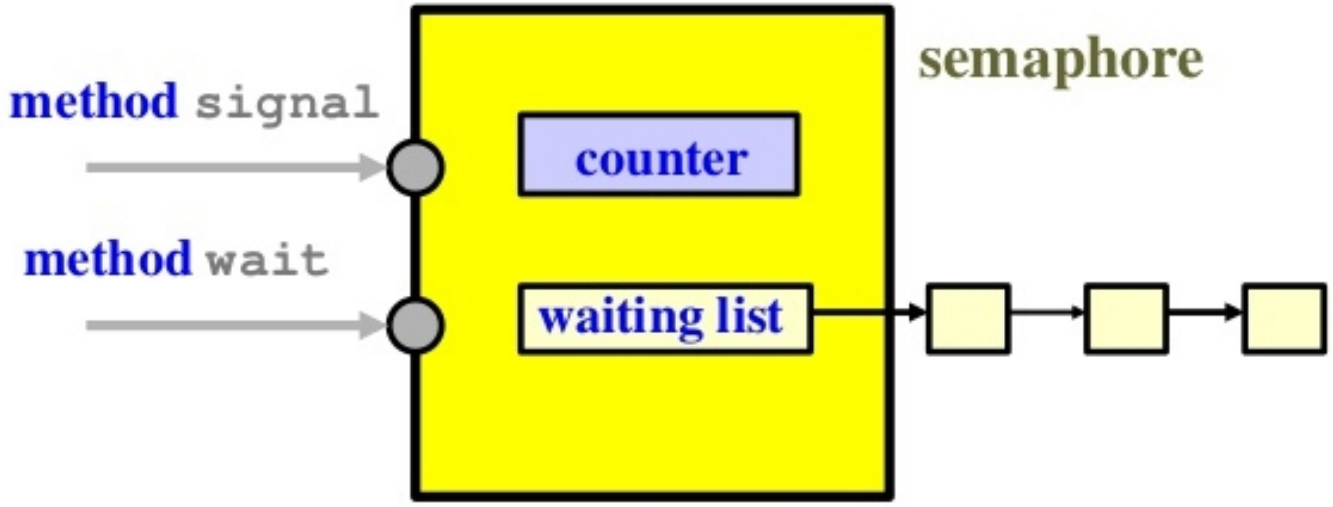
  1. Test the semaphore that controls the resource.

  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.

Semaphores

- A *semaphore* is an object that consists of a counter, a waiting list of processes and two methods (*e.g.*, functions): `signal` and `wait`.

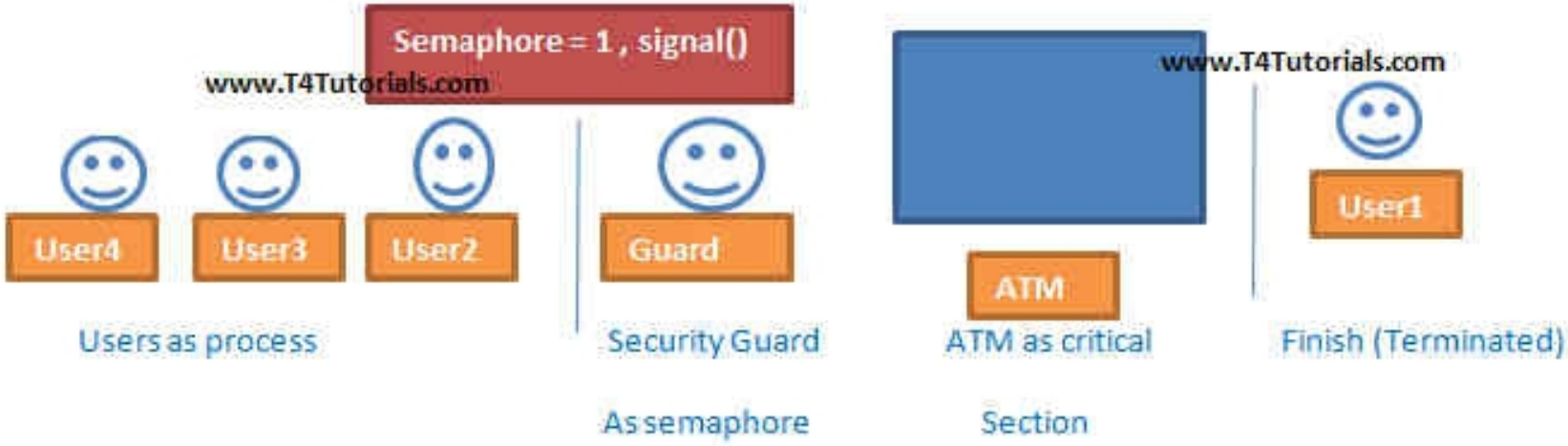


- When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1.
- If any other processes are asleep, waiting for the semaphore, they are awakened. A common form of semaphore is called a ***binary semaphore***.
- It controls a single resource, 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

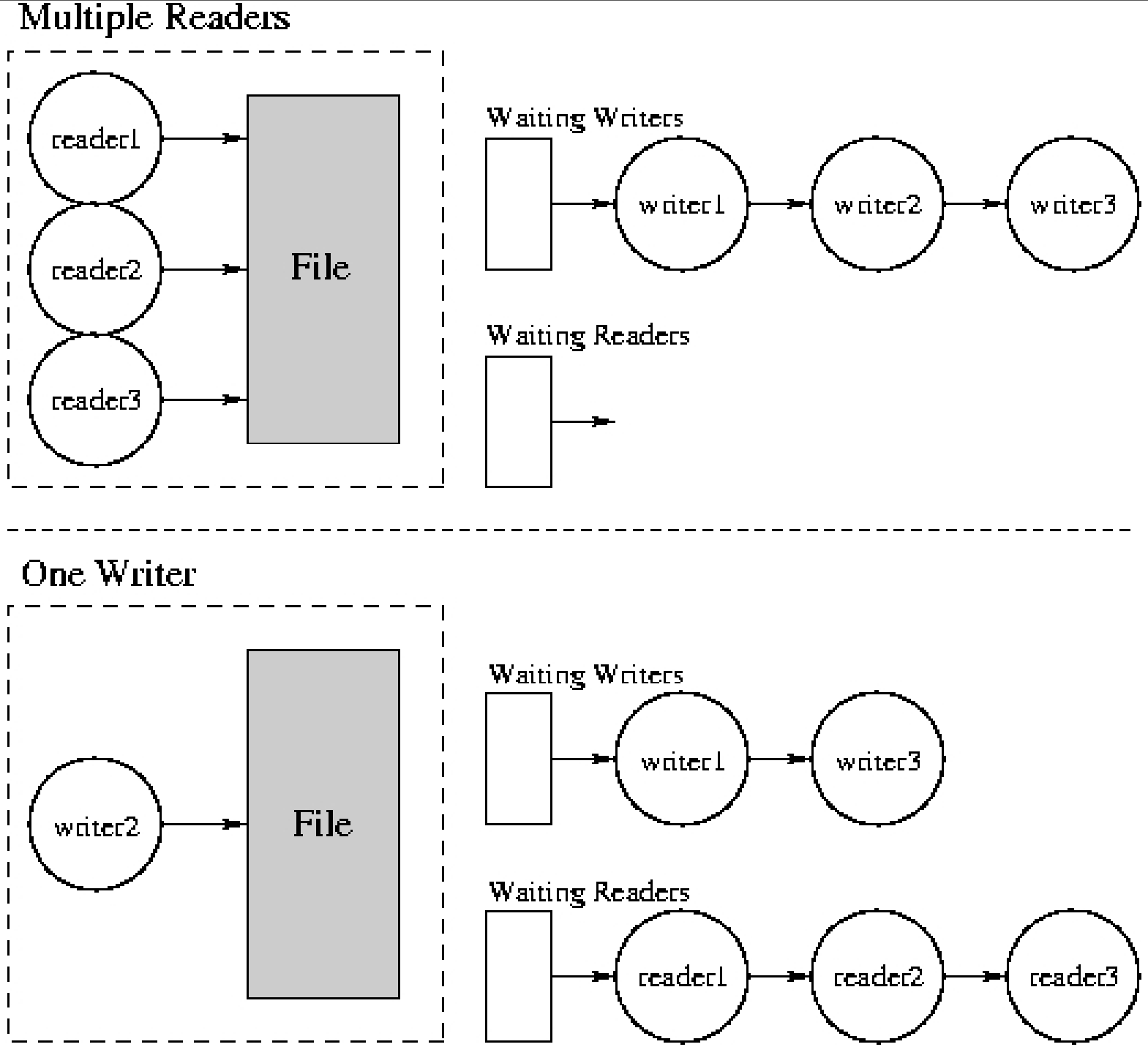
Semaphore

Binary Semaphore

User1(Process 1) is using ATM(Critical section), Guard (Semaphore) is performing wait operation and stops the user2, user3 and user 4



User 1 finish its work and ATM (critical section) is free. Signal is given to all other process.


Multiple Readers
reader1
reader2
reader3
File
Waiting Writers
writer1
writer2
writer3
Waiting Readers
One Writer
writer2
File
Waiting Writers
writer1
writer3
Waiting Readers
reader1
reader2
reader3

- 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 kernel maintains a semid_ds structure for each semaphore set:

```
struct semid_ds {
    struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
    unsigned short   sem_nsems; /* # of semaphores in set */
    time_t           sem_otime; /* last-semop() time */
    time_t           sem_ctime; /* last-change time */
    .
    .
    .
};
```

Each semaphore is represented by an anonymous structure containing at least the following members:

struct {

```
    unsigned short  semval;   /* semaphore value, always >= 0 */
    pid_t           sempid;   /* pid for last operation */
    unsigned short  semncnt;  /* # processes awaiting semval>curval */
    unsigned short  semzcnt;  /* # processes awaiting semval==0 */
    .
    .
    .
};
```

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

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

int semget(key_t key, int nsems, int flag);
```

Returns: semaphore ID if OK, 1 on error

- The **semget**() system call returns the semaphore set identifier associated with the argument *key*.
- A new set of *nsems* semaphores is created if *key* has the value **IPC_PRIVATE** or if no existing semaphore set is associated with *key* and **IPC_CREAT** is specified in *flag*.
- If *flag* specifies both **IPC_CREAT** and **IPC_EXCL** and a semaphore set already exists for *key*, then **semget**() fails with *errno* set to **EEXIST**.

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>

int semctl(int semid, int semnum, int  cmd,... /* union semun arg */);
```

- **semctl**() performs the control operation specified by *cmd* on the semaphore set identified by *semid*, or on the *semnum*-th semaphore of that set. (The semaphores in a set are numbered starting at 0.)
- This function has three or four arguments, depending on *cmd*.
- When there are four, the fourth has the type *union semun*. The *calling program* must define this union as follows:

- 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) */
  short           sem_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

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.
- The only restriction in using shared memory is synchronizing access to a given region among multiple processes.
- If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done.
- Often, semaphores are used to synchronize shared memory access

- The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
  .
  .
  .
};
```

```
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 */
  .
  .
  .
};
```

- The type shmatt_t is defined to be an unsigned integer at least as large as an unsigned short. Figure below lists the system limits that affect shared memory.

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum size in bytes of a shared memory segment	33,554,432	33,554,432	4,194,304	8,388,608
The minimum size in bytes of a shared memory segment	1	1	1	1
The maximum number of shared memory segments, systemwide	192	4,096	32	100
The maximum number of shared memory segments, per process	128	4,096	8	6

The first function called is usually shmget, to obtain a shared memory identifier.

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

int shmget(key_t key, size_t size, int flag);
```

Returns: shared memory ID if OK, 1 on error

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

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

Returns: key if OK, `(key_t)`-1 on error

- When a new segment is created, the following members of the shmid_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*. shm_lpid, shm_nattach, 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.

- The shmctl function is the catchall for various shared memory operations

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Returns: 0 if OK, 1 on 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 following three fields from the structure pointed to by *buf* in the shmid_ds structure associated with this shared memory segment: shm_perm.uid, shm_perm.gid, and shm_perm.mode.
- IPC_RMID Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments the segment is not removed until the last process using the segment terminates or detaches it.