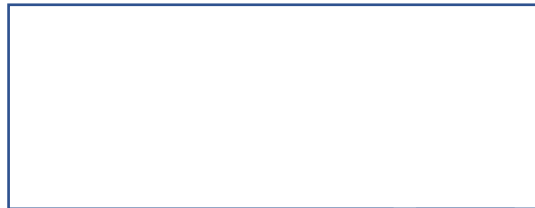


MODULE 4

CHAPTER 01

Changing User IDs & 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.



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.

ID	exec		setuid(uid)	
	set-user-ID bit off	set-user-ID bit on	superuser	Unprivileged 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

The above Table summarises the various ways these three user IDs can be changed

setreuid & setregid Functions:

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

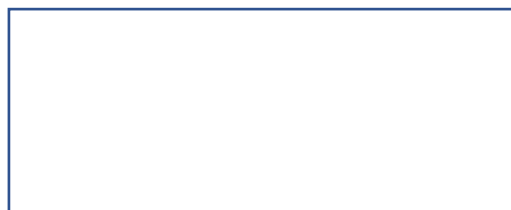


Both return : 0 if OK, -1 on error.

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 & 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.



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.).

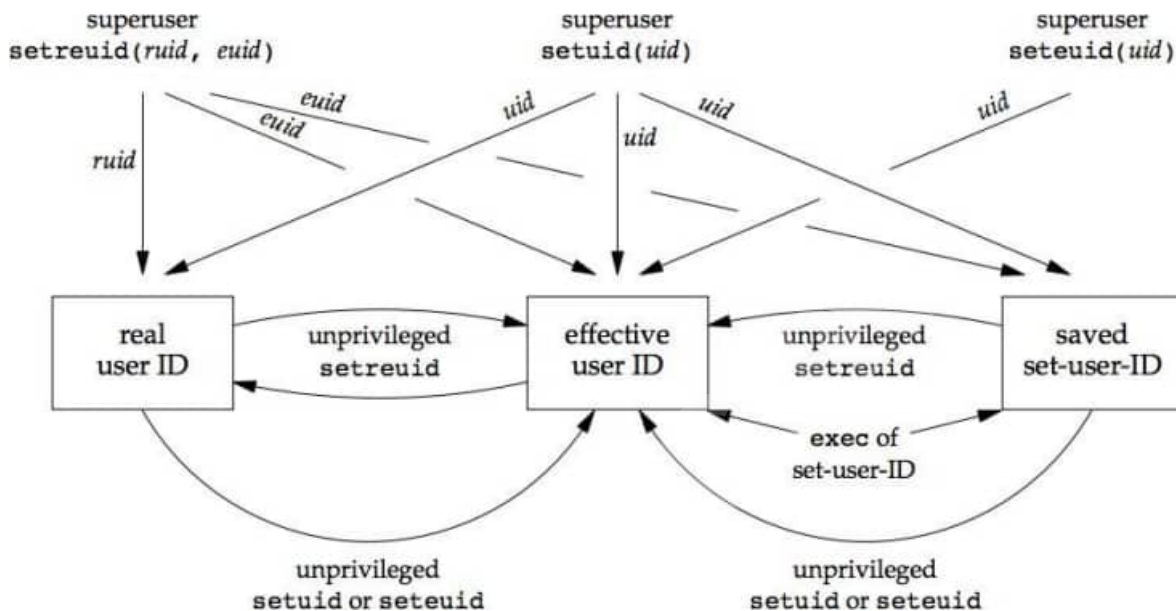


Figure: Summary of all the functions that set the various user IDs.

Interpreter Files

These files are text files that begin with a line of the form
`#! pathname [ optional-argument ]`

The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line `#!/bin/sh`.

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., `PATH` is not used). The recognition of these files is done within the kernel as part of processing the `exec` system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter file a text file that begins with `#!` and the interpreter, which is specified by the pathname on the first line of the interpreter file. Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the `#!`, the pathname, the optional argument, the terminating newline, and any spaces.

A program that execs an interpreter file.

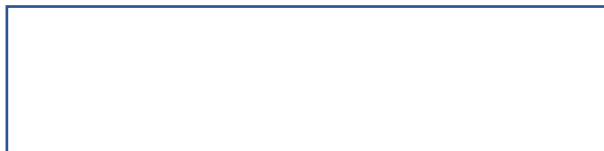
```
#include "apue.h"
#include <sys/wait.h>
int main(void)
{
    pid_t pid;
    if ((pid = fork()) < 0)
    {
        err_sys("fork error");
    }
    else if (pid == 0)
    { /* child */
        if (execl("/home/sar/bin/testinterp", "testinterp", "myarg1", "MY
        ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Output:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

system Function:

It helps us execute a command string within a program.
System is implemented by calling fork, exec & waitpid.



Returns a value of system function.

-1 if either fork fails or waitpid returns an error other than EINTR.

127 – If exec fails [as if shell has executed exit].

Termination status of the shell – if all three function succeed.

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);
}
```

Program: Execute the command-line argument using system

```
#include "apue.h"
int main(int argc, char *argv[])
{
    int status;
    if (argc < 2)
        err_quit("command-line argument required");
    if ((status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);
    exit(0);
}
```

Program: Print real and effective user IDs

```
#include "apue.h"
int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    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 and looks something like:

```
typedef u_short comp_t; /* 3-bit base 8 exponent; 13-bit fraction */
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

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 */
    }

    /* second child */
    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 */
}

```

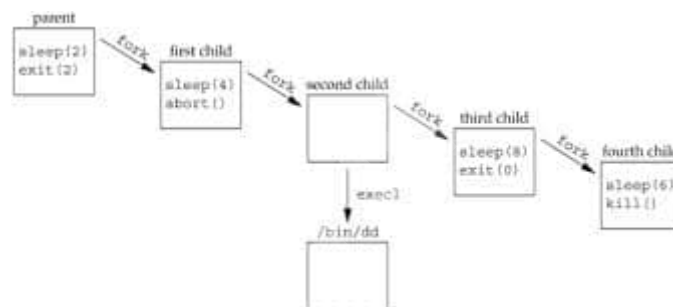
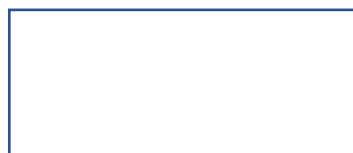


Figure: Process structure for accounting example

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.



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

We describe three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the times function to obtain these values for itself and any terminated children.



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 */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value.

INTERPROCESS COMMUNICATION

Introduction

1. We have seen the process control primitives and saw how to invoke multiple processes.
2. The only way for these processes to exchange information is by passing open files across a fork or an exec or through the file system.
3. We will now describe other techniques for processes to communicate with each other: IPC, or inter-process communication.
4. In the past, UNIX System IPC was a hodgepodge of various approaches, few of which were portable across all UNIX system implementations.
5. Through the POSIX and The Open Group (formerly X/Open) standardization efforts, the situation has improved, but differences still exist.
6. Table 7.1 summarizes the various forms of IPC.

Sno.	IPC type
1.	half-duplex pipes
2.	FIFOs
3.	full-duplex pipes
4.	named full-duplex pipes
5.	message queues
6.	Semaphores
7.	shared memory
8.	Sockets
9	STREAMS

7. The first seven forms of IPC in Table 7.1 are usually restricted to IPC between processes on the same host.
8. The final two rows—sockets and STREAMS—are the only two that are generally supported for IPC between processes on different hosts.

Pipes

1. Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.
 - a.) Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
 - b.) 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.
2. Half-duplex pipes are the most commonly used form of IPC.
3. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one to the standard input of the next using a pipe.
4. A pipe is created by calling the pipe

```
function.#include <unistd.h>
int pipe(int filed[2]);
```

Returns: 0 if OK, -1 on error

5. Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing.
6. The output of filedes[1] is the input for filedes[0].
7. Two ways to picture a half-duplex pipe are shown in figure 7.1.

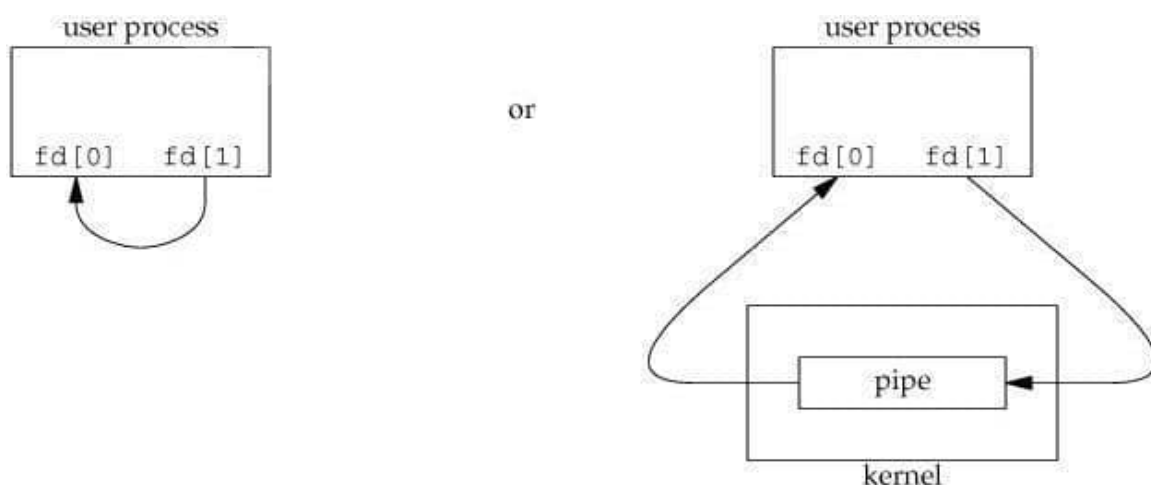


Figure 7.1: Two ways to view a half-duplex pipe

8. The left half of the figure 7.1 shows the two ends of the pipe connected in a single process.
9. The right half of the figure 7.1 emphasizes that the data in the pipe flows through the kernel.
10. The `fstat` function returns a file type of `FIFO` for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.
11. A pipe in a single process is next to useless.
12. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child or vice versa.
13. What happens after the fork depends on which direction of data flow we want.
14. 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]`).
15. Figure 7.2 shows the resulting arrangement of descriptors.

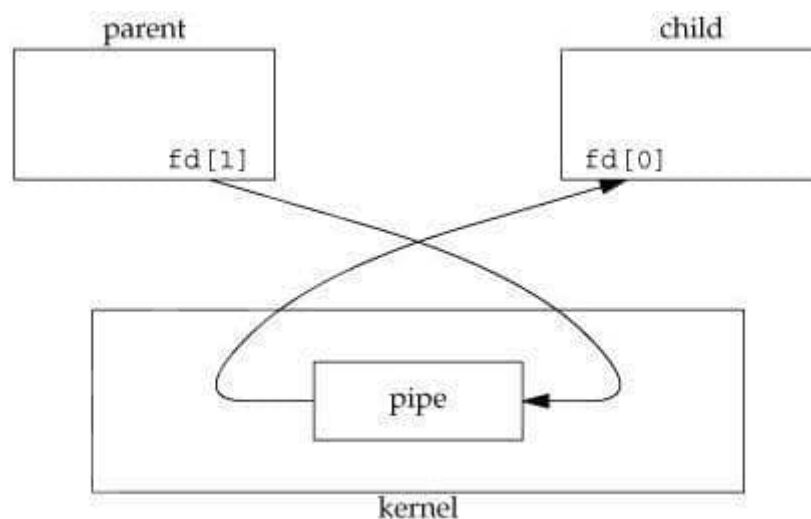


Figure 7.2: Pipe from parent to child

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

2. 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.
18. When we are writing to a pipe (or FIFO), the constant PIPE_BUF specifies the kernel's pipe buffer size.

Example

19. Program 7.1 shows the code to create a pipe between a parent and its child and to send data down the pipe.

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

int main(void)
{
    int    n;
    int    fd[2];
    pid_t  pid;
    char   line[MAXLINE];

    if (pipe(fd) < 0)
        perror("pipe error");
    if ((pid = fork()) < 0) {
        perror("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);
}
```

Program 7.1: Send data from parent to child over a pipe

20. In Program 7.1, we called read and write directly on the pipe descriptors.
21. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output.

22. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Example

23. Recall the five functions `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD`.

24. Program code 7.2 shows an implementation of the functions using pipes.

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

static int pfd1[2], pfd2[2];
void TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        perror("pipe error");
}
void TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        perror("write error");
}
void WAIT_PARENT(void)
{
    char c;
    if (read(pfd1[0], &c, 1) != 1)
        perror("read error");
    if (c != 'p') {
        printf("WAIT_PARENT: incorrect data");
        exit(1);
    }
}
void TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        perror("write error");
}
void WAIT_CHILD(void)
{
    char c;
    if (read(pfd2[0], &c, 1) != 1)
        perror("read error");
}
```

```

    if (c != 'c')
        printf("WAIT_CHILD: incorrect data");
        exit(1);
    }
}

```

Program 7.2: Routines to let a parent and child synchronize

25. We create two pipes before the fork, as shown in program 7.2.

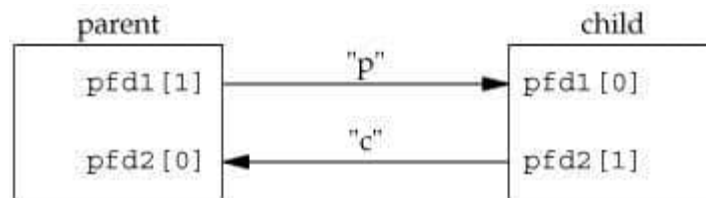


Figure 7.3: Using two pipes for parent—child synchronization

26. The parent writes the character "p" across the top pipe when TELL_CHILD is called, and the child writes the character "c" across the bottom pipe when TELL_PARENT is called.
27. The corresponding WAIT_xxx functions do a blocking read for the single character.
28. Note that each pipe has an extra reader, which doesn't matter. That is, in addition to the child reading from pfd1[0], the parent also has this end of the top pipe open for reading. This does not affect us, since the parent doesn't try to read from this pipe.

popen and pclose Functions

1. 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 provides the popen and pclose functions.
2. These two functions handle all the dirty work that we have been doing ourselves:

```

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.

```

3. The prototypes of these functions are:

```
#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

4. The function popen does a fork and exec to execute the cmdstring, and returns a standard I/O file pointer.
5. If argument type is "r", the file pointer is connected to the standard output of cmdstring (figure 7.4)



Figure 7.4: Result of fp = popen(cmdstring, "r")

6. If type is "w", the file pointer is connected to the standard input of cmdstring, as shown in figure 7.5.



Figure 7.5. Result of fp = popen(cmdstring, "w")

7. The pclose function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell.
8. If the shell cannot be executed, the termination status returned by pclose is as if the shell had executed exit(127).
9. The cmdstring is executed by the Bourne shell, as in

```
sh -c cmdstring
```

10. 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");
```

or

```
fp = popen("cmd 2>&1", "r");
```

Example: Implementation of popen and pclose Functions

11. Program code 7.3 shows the implementation of popen and pclose.

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

/* Pointer to array allocated at run-time. */
static pid_t *childpid = NULL;
static int maxfd;

FILE * popen(const char *cmdstring, const char *type)
{
    int i;
    int pfd[2];
    pid_t pid;
    FILE *fp;
    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0)
        { errno = EINVAL; /* required by POSIX */
          return(NULL);
        }

    if (childpid == NULL) { /* first time through */
        allocate zeroed out array for child pids */
        maxfd = sysconf(_SC_OPEN_MAX);
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }
    if (pipe(pfd) < 0)
        return(NULL); /* errno set by pipe() */

    if ((pid = fork()) < 0) {
        return(NULL); /* errno set by fork() */
    } else if (pid == 0) { /* child */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
```

```

        close(pfd[1]);
        if (pfd[0] != STDIN_FILENO) {
            dup2(pfd[0], STDIN_FILENO);
            close(pfd[0]);
        }
    }
    /* close all descriptors in childpid[] */
    for (i = 0; i < maxfd; i++)
        if (childpid[i] > 0)
            close(i);

    execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
    _exit(127);
}
/* parent continues... */
if (*type == 'r') {
    close(pfd[1]);
    if ((fp = fdopen(pfd[0], type)) == NULL)
        return(NULL);
} else {
    close(pfd[0]);
    if ((fp = fdopen(pfd[1], type)) == NULL)
        return(NULL);
}
childpid[fileno(fp)] = pid; /* remember child pid for this fd */
return(fp);
}

int pclose(FILE *fp)
{
    int    fd, stat;
    pid_t  pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1); /* popen() has never been called */
    }

    fd = fileno(fp);
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1); /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;

```

```

    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */
    return(stat); /* return child's termination status */
}

```

Program 7.3 The popen and pclose functions

Co-processes

1. A UNIX system filter is a program that reads from standard input and writes to standard output.
2. Filters are normally connected linearly in shell pipelines. A filter becomes a co-process when the same program generates the filter's input and reads the filter's output.
3. The Korn shell provides co-processes. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as co-processes.
4. A co-process normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.
5. Co-processes are also useful from a C program.
6. Whereas popen gives us a one-way pipe to the standard input or from the standard output of another process, with a co-process, we have two one-way pipes to the other process: one to its standard input and one from its standard output.
7. We want to write to its standard input, let it operate on the data, and then read from its standard output.

Example

8. Let's look at co-processes with an example. The process creates two pipes: one is the standard input of the co-process, and the other is the standard output of the co-process. Figure 7.7 shows this arrangement.

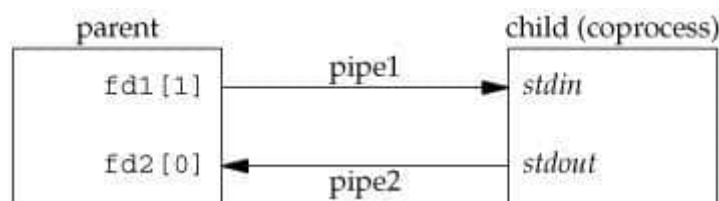


Figure 7.7. Driving a co-process by writing its standard input and reading its standard output

9. The program 7.4 is a simple co-process that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output. We compile this program and leave the executable in the file add2.

```
#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);
}
```

Program 7.4. Simple filter to add two numbers

The program 7.5 invokes the add2 co-process after reading two numbers from its standard input. The value from the co-process is written to its standard output.

```
#include "apue.h"

static void sig_pipe(int); /* our signal handler */

int main(void)
{
    int    n, fd1[2], fd2[2];
    pid_t  pid;
    char   line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");
```

```

if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid > 0) {          /* parent */
    close(fd1[0]);
    close(fd2[1]);
    while (fgets(line, MAXLINE, stdin) != NULL)
        { n = strlen(line);
          if (write(fd1[1], line, n) != n)
              err_sys("write error to pipe");
          if ((n = read(fd2[0], line, MAXLINE)) < 0)
              err_sys("read error from pipe");
          if (n == 0) {
              err_msg("child closed pipe");
              break;
          }
          line[n] = 0; /* null terminate */
          if (fputs(line, stdout) == EOF)
              err_sys("fputs error");
        }
    if (ferror(stdin))
        err_sys("fgets error on stdin");
    exit(0);
} else {                      /* child */
    close(fd1[1]);
    close(fd2[0]);
    if (fd1[0] != STDIN_FILENO) {
        if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        close(fd1[0]);
    }
    if (fd2[1] != STDOUT_FILENO) {
        if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        close(fd2[1]);
    }
    if (execl("./add2", "add2", (char *)0) < 0)
        err_sys("execl error");
    }
    exit(0);
}

static void sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}

```

Program 7.5. Program to drive the add2 filter

10. Here, we create two pipes, with the parent and the child closing the ends they don't need.
11. We have to use two pipes: one for the standard input of the co-process and one for its standard output.
12. The child then calls `dup2` to move the pipe descriptors onto its standard input and standard output, before calling `execl`.
13. If we compile and run the program 7.5, it works as expected. Furthermore, if we kill the `add2` co-process while the program 7.5 is waiting for our input and then enter two numbers, the signal handler is invoked when the program writes to the pipe that has no reader.

FIFOs

1. FIFOs are sometimes called named pipes.
2. Pipes can be used only between related processes when a common ancestor has created the pipe.
3. With FIFOs, however, unrelated processes can exchange data.
4. We saw earlier that a FIFO is a type of file. One of the encodings of the `st_mode` member of the `stat` structure indicates that a file is a FIFO.
5. We can test for this with the `S_ISFIFO` macro.
6. Creating a FIFO is similar to creating a file. Indeed, the pathname for a FIFO exists in the file system.

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

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

Returns: 0 if OK, 1 on error

7. The specification of the mode argument for the `mkfifo` function is the same as for the `open` function.
8. The rules for the user and group ownership of the new FIFO are the same as we described earlier.
9. Once we have used `mkfifo` to create a FIFO, we open it using `open`.
10. Indeed, the normal file I/O functions (`close`, `read`, `write`, `unlink`, etc.) all work with FIFOs.

11. 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.

12. As with a pipe, if we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated.

13. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.

14. It is common to have multiple writers for a given FIFO. This means that we have to worry about atomic writes if we do not want the writes from multiple processes to be interleaved.

15. As with pipes, the constant PIPE_BUF specifies the maximum amount of data that can be written atomically to a FIFO.

16. There are two uses for FIFOs.

1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
2. 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

17. FIFOs can be used to duplicate an output stream in a series of shell commands.

18. This prevents writing the data to an intermediate disk file (similar to using pipes to avoid intermediate disk files).

19. But whereas pipes can be used only for linear connections between processes, a FIFO has a name, so it can be used for nonlinear connections.

20. Consider a procedure that needs to process a filtered input stream twice. Figure 7.8 shows this arrangement.

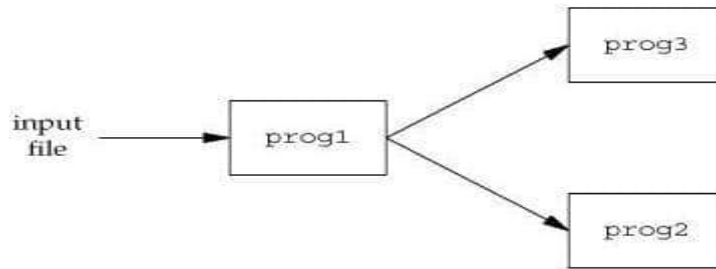


Figure 7.8. Procedure that processes a filtered input stream twice

21. 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
  
```

22. 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.

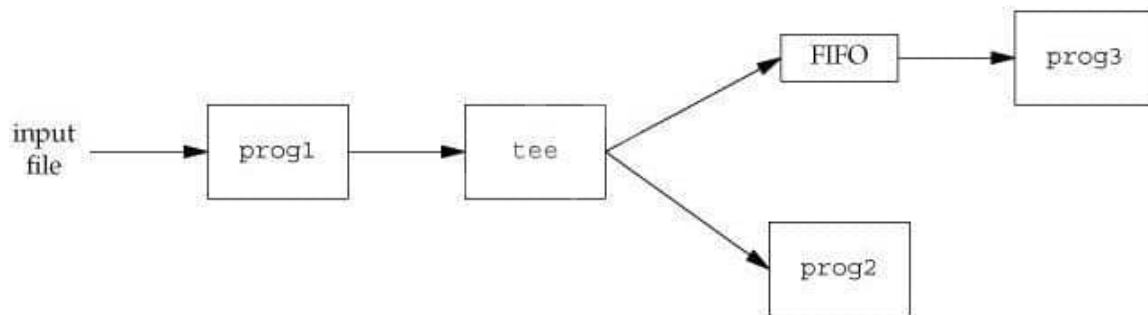


Figure 7.9 shows the process arrangement.

Figure 7.9. Using a FIFO and tee to send a stream to two different processes

Example Client Server Communication Using a FIFO

23. Another use for FIFOs is to send data between a client and a server.
24. 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. (By "well-known" we mean that the pathname of the FIFO is known to all the clients that need to contact the server.)
25. Figure 7.10 shows this arrangement.

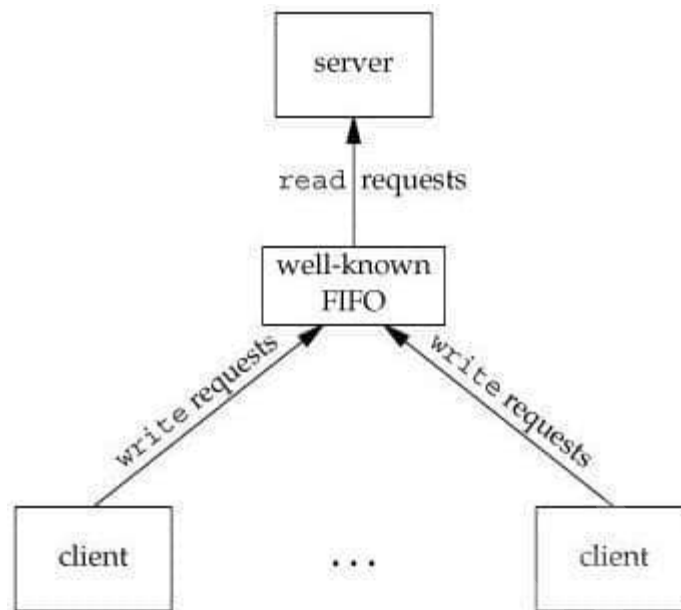


Figure 7.10. Clients sending requests to a server using a FIFO

26. 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.
27. The problem in using FIFOs for this type of client-server communication is how to send replies back from the server to each client.
28. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients.
29. 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.
30. For example, the server can create a FIFO with the name /tmp/serv1.XXXXXX, where XXXXXX is replaced with the client's process ID.
31. Figure 7.11 shows this arrangement.

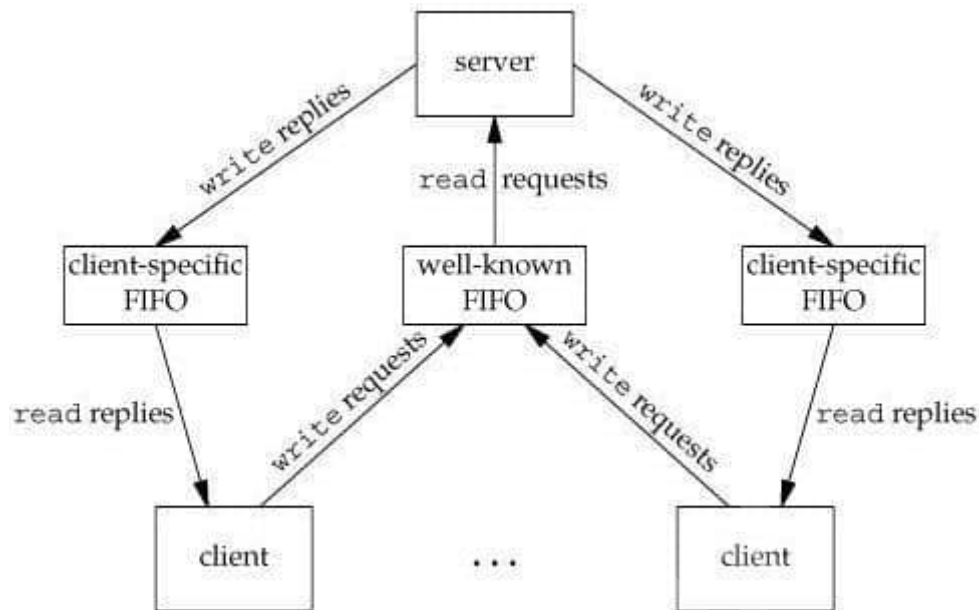


Figure 7.11. Clientserver communication using FIFOs

32. This arrangement works, although it is impossible for the server to tell whether a client crashes.
33. This causes the client-specific FIFOs to be left in the file system. 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.
34. With the arrangement shown in Figure 7.11, if the server opens its well-known FIFO read-only (since it only reads from it) each time the number of clients goes from 1 to 0, the server will read an end of file on the FIFO. To prevent the server from having to handle this case, a common trick is just to have the server open its well-known FIFO for readwrite.

7.6. XSI- X/Open System Interface: IPC

1. The three types of IPC that are called XSI IPC are message queues, semaphores, and shared memory.

Identifiers and Keys

2. Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer identifier.

3. To send or fetch a message to or from a message queue, for example, all we need know is the identifier for the queue.
4. 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.
5. The identifier is an internal name for an IPC object.
6. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object.
7. For this purpose, an IPC object is associated with a key that acts as an external name.
8. Whenever an IPC structure is being created (by calling `msgget`, `semget`, or `shmget`), a key must be specified.
9. 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>`.
10. This key is converted into an identifier by the kernel.
11. There are various ways for a client and a server to rendezvous at the same IPC structure.
 - a. 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 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.
 - b. 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.
 - c. 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

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

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

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

12. The path argument must refer to an existing file. Only the lower 8 bits of id are used when generating the key.
13. The key created by ftok is usually formed by taking parts of the st_dev and st_ino fields in the stat structure corresponding to the given pathname and combining them with the project ID.
14. If two pathnames refer to two different files, then ftok usually returns two different keys for the two pathnames.
15. However, because both i-node numbers and keys are often stored in long integers, there can be information loss creating a key. This means that two different pathnames to different files can generate the same key if the same project ID is used.
16. The three get functions (msgget, semget, and shmget) all have two similar arguments: a key and an integer flag.
17. 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.
18. 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.
19. 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.
20. 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.
21. 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.

Permission Structure

22. XSI IPC associates an ipc_perm structure with each IPC structure.

23. 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 */
    .
    .
    .
};
```

24. Each implementation includes additional members. See <sys/ipc.h> on your system for the complete definition.
25. All the fields are initialized when the IPC structure is created.
26. Later, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl.
27. To change these values, the calling process must be either the creator of the IPC structure or the superuser.
28. Changing these fields is similar to calling chown or chmod for a file.
29. The values in the mode field are as shown below.

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

30. Below are the six permissions for each form of IPC.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Configuration Limits

31. All three forms of XSI IPC have built-in limits that we may encounter.

32. Most of these limits can be changed by reconfiguring the kernel.

Advantages and Disadvantages

33. A fundamental problem with XSI IPC is that the IPC structures are system wide and do not have a reference count.

34. 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.

35. They remain in the system until specifically read or deleted by some process calling `msgrcv` or `msgctl`, by someone executing the `ipcrm` command, or by the system being rebooted.

36. Another problem with XSI IPC is that these IPC structures are not known by names in the file system.

37. We can't access them and modify their properties.

38. Almost a dozen new system calls (`msgget`, `semop`, `shmat`, and so on) were added to the kernel to support these IPC objects.

39. 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.

40. Instead, two new commands `ipcs(1)` and `ipcrm(1)` were added.

41. Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (select and poll) with them.
42. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O.
43. Other advantages for message queues are that they're reliable, flow controlled; record oriented, and can be processed in other than first-in, first-out order.
44. The following compares some of the features of these various forms of IPC.

IPC type	Connectionless?	Reliable?	Flow control?	Records?	Message types or priorities?
message queues	no	yes	yes	yes	yes
STREAMS	no	yes	yes	yes	yes
UNIX domain stream socket	no	yes	yes	no	no
UNIX domain datagram socket	yes	yes	no	yes	no
FIFOs (non-STREAMS)	no	yes	yes	no	no

Message Queues

1. A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
2. We will call the message queue just a queue and its identifier a queue ID.
3. A new queue is created or an existing queue opened by msgget.
4. New messages are added to the end of a queue by msgsnd.
5. 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.
6. Messages are fetched from a queue by msgrcv.
7. We do not have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.
8. Each queue has the following msqid_ds structure associated with it:

```

struct msqid_ds {
    struct ipc_perm    msg_perm;
    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 */
    .
    .
    .
};

```

9. This structure defines the current status of the queue. The members shown are the ones defined by the Single UNIX Specification. Implementations include additional fields not covered by the standard.
10. 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

11. We have already seen the rules for converting the key into an identifier and discussed whether a new queue is created or an existing queue is referenced.
12. 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.

13. On success, msgget returns the non-negative queue ID. This value is then used with the other three message queue functions.
14. 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

15. 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.

16. 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

17. As we mentioned earlier, 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).

18. Messages are always placed at the end of the queue.

19. 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.)

20. If the largest message we send is 512 bytes, we can define the following structure:

```

struct mtypes {
    long    mtype;
    char    mtext[512];
};

```

21. The ptr argument is then a pointer to a mtypes structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.
22. A flag value of IPC_NOWAIT can be specified. This is similar to the nonblocking I/O flag for file I/O.
23. If the message queue is full, specifying IPC_NOWAIT causes msgsnd to return immediately with an error of EAGAIN.
24. If IPC_NOWAIT is not specified, we are blocked until there is room for the message, the queue is removed from the system, or a signal is caught and the signal handler returns.
25. In the second case, an error of EIDRM is returned ("identifier removed"); in the last case, the error returned is EINTR.
26. Note how ungracefully the removal of a message queue is handled. Since a reference count is not maintained with each message queue, the removal of a queue simply generates errors on the next queue operation by processes still using the queue.
27. When msgsnd returns successfully, the msqid_ds structure associated with the message queue is updated to indicate the process ID that made the call (msg_lspid), the time that the call was made (msg_stime), and that one more message is on the queue (msg_qnum).
28. 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

29. The ptr argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data.
30. nbytes specifies the size of the data buffer. If the returned message is larger than nbytes and the MSG_NOERROR bit in flag is set, the message is truncated.
31. 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.

32. A nonzero type is used to read the messages in an order other than first in, first out.
33. For example, the type could be a priority value if the application assigns priorities to the messages. Another use of this field is to contain the process ID of the client if a single message queue is being used by multiple clients and a single server (as long as a process ID fits in a long integer).
34. We can specify a flag value of IPC_NOWAIT to make the operation nonblocking, causing msgrcv to return -1 with errno set to ENOMSG if a message of the specified type is not available.
35. If IPC_NOWAIT is not specified, the operation blocks until a message of the specified type is available, the queue is removed from the system (-1 is returned with errno set to EIDRM), or a signal is caught and the signal handler returns (causing msgrcv to return 1 with errno set to EINTR).
36. When msgrcv succeeds, the kernel updates the msqid_ds structure associated with the message queue to indicate the caller's process ID (msg_lpid), the time of the call (msg_rtime), and that one less message is on the queue (msg_qnum).

Semaphores

1. A semaphore is not a form of IPC. A semaphore is a counter used to provide access to a shared data object for multiple processes.
2. To obtain a shared resource, a process needs to do the following:
 - a.) Test the semaphore that controls the resource.
 - b.) 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.
 - c.) 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.
3. 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.

4. To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.
5. A common form of semaphore is called a binary semaphore. It controls a single resource, and its value is initialized to 1.
6. 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.
7. XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.
 - a.) 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.
 - b.) 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.
 - c.) 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.
8. The kernel maintains a `semid_ds` structure for each semaphore set:

```

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

```

9. The Single UNIX Specification defines the fields shown, but implementations can define additional members in the `semid_ds` structure.
10. 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 */
    .
    .
    .
};

```

11. The table below lists the system limits that affect semaphore sets.

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum value of any semaphore	32,767	32,767	32,767	32,767
The maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	16,384
The maximum number of semaphore sets, systemwide	10	128	87,381	10
The maximum number of semaphores, systemwide	60	32,000	87,381	60
The maximum number of semaphores per semaphore set	60	250	87,381	25
The maximum number of undo structures, systemwide	30	32,000	87,381	30
The maximum number of undo entries per undo structures	10	32	10	10
The maximum number of operations per semop call	100	32	100	10

12. 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

13. We have already seen the rules for converting the key into an identifier and discussed whether a new set is created or an existing set is referenced.
14. 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`.

15. The number of semaphores in the set is `nsems`.
16. 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.
17. 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 */);
```

18. 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 */
};
```

19. Note that the optional argument is the actual union, not a pointer to the union.
20. The `cmd` argument specifies one of the following ten commands to be performed on the set specified by `semid`.
21. The five commands that refer to one particular semaphore value use `semnum` to specify one member of the set. The value of `semnum` is between 0 and `nsems-1`, inclusive.

IPC_STAT	Fetch the <code>semid_ds</code> structure for this set, storing it in the structure pointed to by <code>arg.buf</code> .
----------	--

IPC_STAT	Fetch the semid_ds structure for this set, storing it in the structure pointed to by arg.buf.
IPC_SET	Set the sem_perm.uid, sem_perm.gid, and sem_perm.mode fields from the structure pointed to by arg.buf in the semid_ds structure associated with this set. This command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or by a process with superuser privileges.
IPC_RMID	Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of EIDRM on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or by a process with superuser privileges.
GETVAL	Return the value of semval for the member semnum.
SETVAL	Set the value of semval for the member semnum. The value is specified by arg.val.
GETPID	Return the value of sempid for the member semnum.
GETNCNT	Return the value of semncnt for the member semnum.
GETZCNT	Return the value of semzcnt for the member semnum.
GETALL	Fetch all the semaphore values in the set. These values are stored in the array pointed to by arg.array.
SETALL	Set all the semaphore values in the set to the values pointed to by arg.array.

22. For all the GET commands other than GETALL, the function returns the corresponding value. For the remaining commands, the return value is 0.

23. 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

24. 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
    */
};

```

25. The nops argument specifies the number of operations (elements) in the array.
26. The operation on each member of the set is specified by the corresponding sem_op value.
27. This value can be negative, 0, or positive.

a.) The easiest case is when sem_op is positive. This case corresponds to the returning of resources by the process. The value of sem_op is added to the semaphore's value. If the undo flag is specified, sem_op is also subtracted from the semaphore's adjustment value for this process.

b.) If sem_op is negative, we want to obtain resources that the semaphore controls. If the semaphore's value is greater than or equal to the absolute value of sem_op (the resources are available), the absolute value of sem_op is subtracted from the semaphore's value. This guarantees that the resulting value for the semaphore is greater than or equal to 0. If the undo flag is specified, the absolute value of sem_op is also added to the semaphore's adjustment value for this process.

If the semaphore's value is less than the absolute value of sem_op (the resources are not available), the following conditions apply.

- a. If IPC_NOWAIT is specified, semop returns with an error of EAGAIN.
- b. If IPC_NOWAIT is not specified, the semncnt value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.
 - i. The semaphore's value becomes greater than or equal to the absolute value of sem_op (i.e., some other process has released some resources). The value of semncnt for this semaphore is decremented (since the calling process is done waiting), and the absolute value of sem_op is subtracted from the semaphore's value. If the undo flag is specified, the absolute value of sem_op is also added to the semaphore's adjustment value for this process.
 - ii. The semaphore is removed from the system. In this case, the function returns an error of EIDRM.
 - iii. A signal is caught by the process, and the signal handler returns. In this case, the value of semncnt for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of EINTR.
- c.) If sem_op is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.

If the semaphore's value is currently 0, the function returns immediately.

If the semaphore's value is nonzero, the following conditions apply.

- a. If IPC_NOWAIT is specified, return is made with an error of EAGAIN.
- b. If IPC_NOWAIT is not specified, the semzcnt value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.
 - i. The semaphore's value becomes 0. The value of semzcnt for this semaphore is decremented (since the calling process is done waiting).
 - ii. The semaphore is removed from the system. In this case, the function returns an error of EIDRM.
 - iii. A signal is caught by the process, and the signal handler returns. In this case, the value of semzcnt for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of EINTR.

28. The semop function operates atomically; it does either all the operations in the array or none of them.

Semaphore Adjustment on exit

29. It is a problem if a process terminates while it has resources allocated through a semaphore.

30. Whenever we specify the SEM_UNDO flag for a semaphore operation and we allocate resources (a sem_op value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of sem_op).

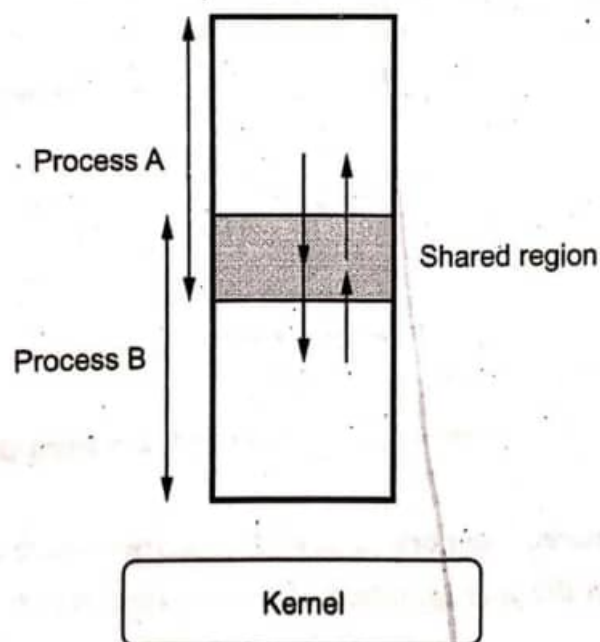
31. When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore.

32. If we set the value of a semaphore using semctl, with either the SETVAL or SETALL commands, the adjustment value for that semaphore in all processes is set to 0.

5.8 Shared Memory

VTU : Jan.-15, 18

- 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.
- Access to this shared memory is as fast as accessing a process's non-shared memory, and it does not require a system call or entry to the kernel. It also avoids copying data unnecessarily.
- Because the kernel does not synchronize accesses to shared memory, you must provide your own synchronization. For example, a process should not read from the memory until after data is written there, and two processes must not write to the same memory location at the same time. A common strategy to avoid these race conditions is to use semaphores.
- To use a shared memory segment, one process must allocate the segment. Then each process desiring to access the segment must attach the segment. After finishing its use of the segment, each process detaches the segment. At some point, one process must de-allocate the segment.
- If a process writes to a shared memory location, the new contents of that location are immediately visible to all processes sharing the region.
- Fig. 5.8.1 shows concept of shared memory.



- Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process. Other processes can then attach the same shared memory segment into their own address space.
- Shared memory provides an efficient way of sharing and passing data between multiple processes. By itself, shared memory does not provide any synchronization facilities.
- There is no automatic facility to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It is responsibility of the programmer to synchronize access.
- There can be multiple shared memory regions existing in a given system at any one time.

5.8.1 Kernel Support for UNIX Shared Memory

- In UNIX System V.3 and V.4, there is a shared memory table in the kernel address space that keeps track of all shared memory regions created in the system.
- Each entry of the table stores the data for one shared memory region :
 1. A name that an integer ID key assigned by the process that created the shared memory.
 2. The creator user and group IDs.
 3. The assigned owner user and group IDs.
 4. Read-write access permission of the region for owner, group members and others.
 5. The time when the last process attached to the region.
 6. The time when the last process detached from the region.
 7. The time when the last process changed control data of the region.
 8. The size, in number of bytes of the shared memory region.
- Fig. 5.8.2 shows kernel data structure for shared memory.

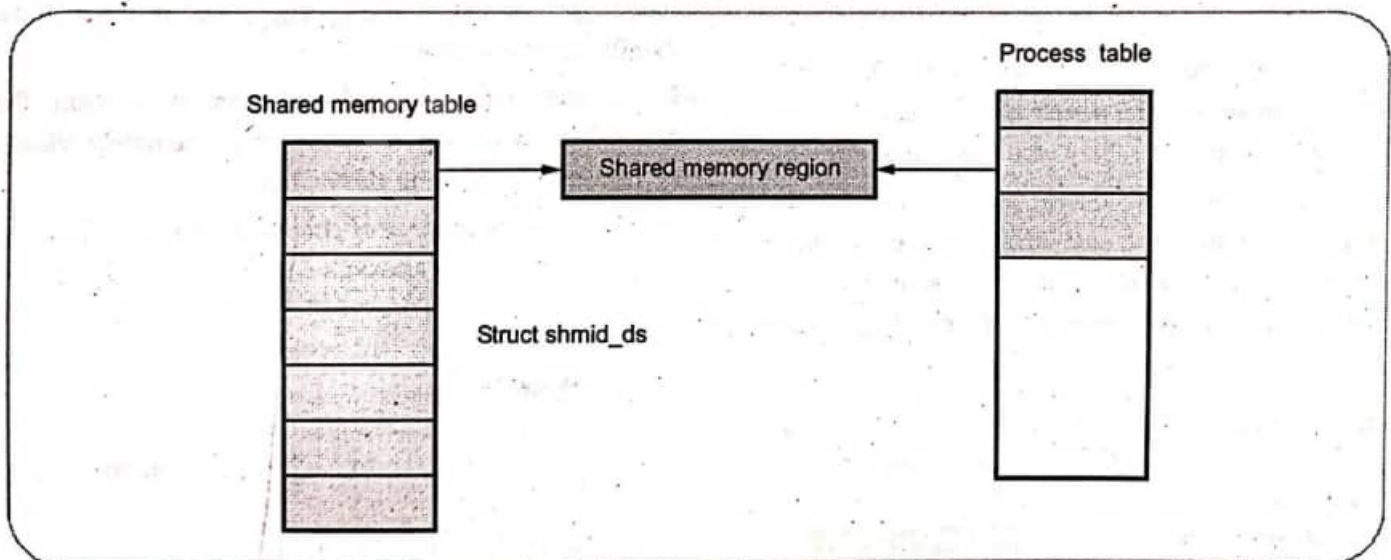


Fig. 5.8.2 Kernel data structure for shared memory

- Shared memory is stored in kernel address space and they are persistent. There are several imposed limit on the manipulation of shared memory. these limits are defined in the `<sys/shm.h>` header :

System limit	Meaning
SHMMNI	The maximum number of shared memory regions that may exist at any given time in a system.
SHMMIN	The minimum size, in number of bytes, of a shared memory region.
SHMMAX	The maximum size, in number of bytes, of a shared memory region.

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

```

struct shmid_ds {
    struct ipc_perm shm_perm;
    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 */
    .
    .
    .
};

```

5.8.2 UNIX System V APIs for Shared Memory

- Each entry in the shared memory table is of type *struct shmid_ds* , which is defined in the *<sys/shm.h>* header.
- The data fields of the structure and the corresponding data it stores are as follows :

Data field	Data stored
shm_perm	Data stored in a struct ipc_perm record.
shm_segsz	The shared memory region size in bytes.
shm_lpid	Process ID of the last process that attaches to the region.
shm_cpid	Creator process ID.
shm_nattch	Number of processes currently attached to the region.
shm_atime	Time when the last process attached to the region.
shm_dtime	Time when the last process detached from the region.
shm_ctime	The time when the last process changed control data of the region.

- UNIX System V shared memory APIs are :

Shared memory APIs	Uses
shmget	Open and create a shared memory.
shmat	Attach a shared memory to a process virtual address space, so that the process can read and/or write data in the shared memory.
shmdt	Detach a shared memory from the process virtual address space.
shmctl	Query or change control data of a shared memory or delete the memory.

- Shared memory required following header files :

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

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

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

5.8.2.1 shmget

- A process allocates a shared memory segment using *shmget*.
- The first function called is usually *shmget*, to obtain a shared memory identifier.

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

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

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

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

- Its first parameter is an integer key that specifies which segment to create. Unrelated processes can access the same shared segment by specifying the same key value.
- Its second parameter specifies the number of bytes in the segment. It is the size of the shared memory segment in bytes. Because segments are allocated using pages, the number of actually allocated bytes is rounded up to an integral multiple of the page size.
- The third parameter is flag values that specify options to *shmget*. The flag values include these :

Flag	Uses
IPC_CREAT	This flag indicates that a new segment should be created. This permits creating a new segment while specifying a key value.
IPC_EXCL	This flag, which is always used with IPC_CREAT, causes <i>shmget</i> to fail if a segment key is specified that already exists. Therefore, it arranges for the calling process to have an "exclusive" segment. If this flag is not given and the key of an existing segment is used, <i>shmget</i> returns the existing segment instead of creating a new one.
Mode flags	This value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment. Execution bits are ignored.

- For example, *shmget* creates a new shared memory segment that's readable and writeable to the owner but not other users.

```
int segment_id = shmget (shm_key, getpagesize (), IPC_CREAT | S_IRUSR | S_IWUSER);
```

- If the call succeeds, *shmget* returns a segment identifier. If the shared memory segment already exists, the access permissions are verified and a check is made to ensure that the segment is not marked for destruction.
- *shmget*() is used to obtain access to a shared memory segment.

The following code illustrates *shmget*() :

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

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

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

```
...
```

```
key_t key;      /* key to be passed to shmget() */
```

```
int shmflg;     /* shmflg to be passed to shmget() */
```

```
int shmid;      /* return value from shmget() */
```

```
int size;       /* size to be passed to shmget() */
```

```
...
```

```

key = ...
size = ...
shmflg) = ...

if ((shmid = shmget (key, size, shmflg)) == -1)
{
    perror("shmget: shmget failed");
    exit(1);
}
else
{
    (void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
    exit(0);
}
...

```

5.8.2.2 shmat

- Once a shared memory segment has been created, a process attaches it to its address space by calling `shmat`. The function prototype of the `shmat` API is :


```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *addr, int flag);

```
- The process can read / write data in that shared memory. Note that if this is newly created shared memory, the kernel does not actually allocate the memory region until the first process calls this function to attach to it.
- The address in the calling process at which the segment is attached depends on the `addr` argument and whether the `SHM_RND` bit is specified in `flag`.

Addr value	Meaning
<code>addr = 0</code>	The segment is attached at the first available address selected by the kernel. This is the recommended technique.
<code>addr is nonzero</code>	1. <code>SHM_RND</code> is not specified, the segment is attached at the address given by <code>addr</code> . 2. <code>SHM_RND</code> is specified, the segment is attached at the address given by $(addr - (addr \text{ modulus } SHMLBA))$. (<code>SHMLBA</code> stands for "low boundary address multiple" and is always a power of 2.)

- If the `SHM_RDONLY` bit is specified in `flag`, the segment is attached read-only. Otherwise, the segment is attached read-write.

5.8.2.3 shmdt

- `Shmdt` function is used to detach the process. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process specifically removes it by calling `shmctl` with a command of `IPC_RMID`.
- The function prototype of the `shmdt` API is :


```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt (void* addr);

```

- This function detaches shared memory from the specified *addr* virtual address of the calling process. The *addr* value should be obtained from a *shmat* call prior to this function call.
- The return value of the function is 0 if it succeeds or -1 if it fails.
- Fig. 5.8.3 shows memory layout on an Intel-based Linux system.

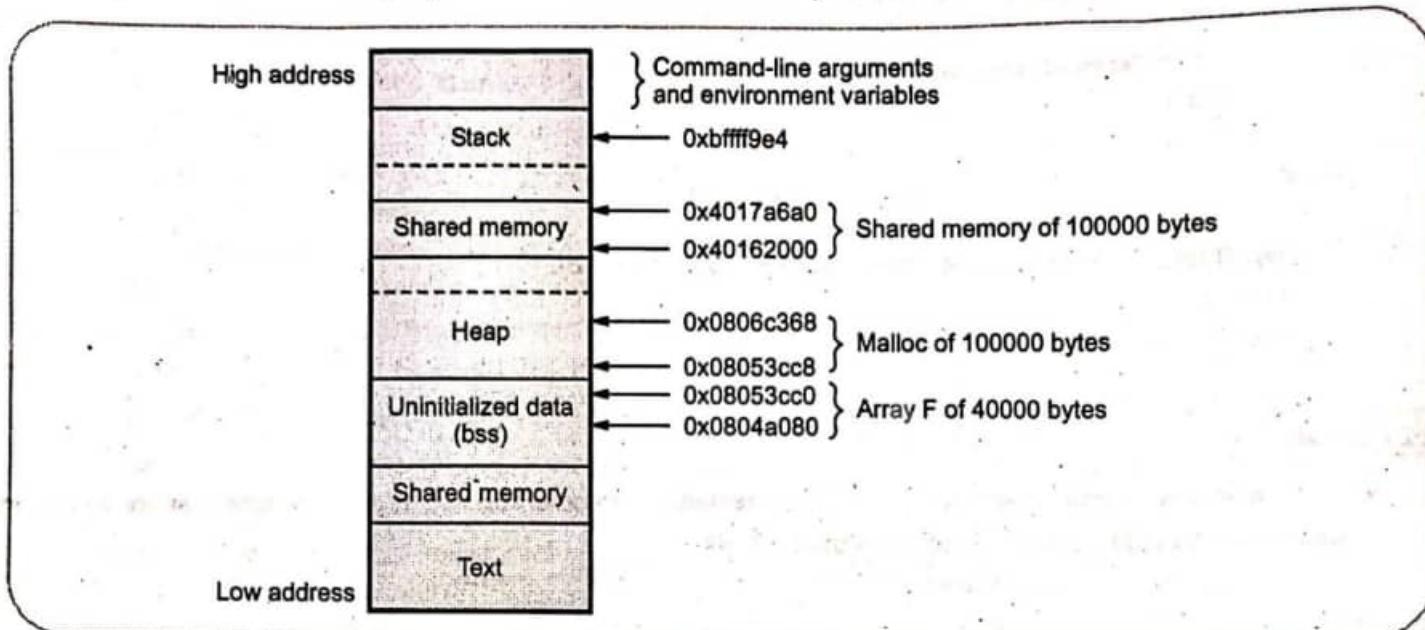


Fig. 5.8.3 Memory layout

Use of Shared Memory

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

int main ()
{
    int segment_id;
    char* shared_memory;
    struct shmid_ds shmbuffer;
    int segment_size;
    const int shared_segment_size = 0x6400;

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

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

    shmctl (segment_id, IPC_STAT, &shmbuffer); /* Determine the segment's size. */
    segment_size = shmbuffer.shm_segsz;
    printf ("segment size: %d\n", segment_size);
    sprintf (shared_memory, "thank you."); /* Write a string to the shared memory segment. */

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

```

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

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

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

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

```

5.8.2.4 shmctl

- The function prototype of the shmctl API is :

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

```

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

- shmctl () performs the control operation specified by cmd on the shared memory segment whose identifier is given in shmid.
- This API can either query or change the control data of a shared memory designated by shmid, or delete the memory altogether.
- The buf argument is a pointer to a shmid_ds structure, defined in <sys/shm.h> as follows :

```

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

```

- The possible value of cmd and their meaning are as follows :

cmd value	Meaning
IPC_STAT	Copy control data of the shared memory to the object pointed to by buf. The calling process must have read permission to the set.
IPC_SET	Change the control data of the shared memory by the data specified in the object pointed to by buf. The calling process must be the superuser, creator etc.
IPC_RMID	Remove the shared memory from the system. The calling process must be the superuser, creator etc.
SHM_LOCK	Lock the shared memory in memory. The calling process must have superuser privileges to perform this task.
SHM_UNLOCK	Unlock the shared memory in memory. The calling process must have superuser privileges to perform this task.

5.8.3 Program Examples

The semaphore must be initialised with the `shmget.c` program. The effects of controlling shared memory and accessing can be investigated with following programs.

`shmget()` program

/ This is a simple exerciser of the shmget() function. */*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
extern void exit();
extern void perror();
```

```
main()
```

```
{
    key_t key;      /* key to be passed to shmget() */
    int shmflg;     /* shmflg to be passed to shmget() */
    int shmid;      /* return value from shmget() */
    int size;       /* size to be passed to shmget() */
```

```
(void) fprintf(stderr, "All numeric input is expected to follow C conventions:\n");
(void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
```

```
/* Get the key. */
```

```
(void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
(void) fprintf(stderr, "Enter key: ");
(void) scanf("%li", &key);
```

```
/* Get the size of the segment. */
```

```
(void) fprintf(stderr, "Enter size: ");
(void) scanf("%i", &size);
```

```
/* Get the shmflg value. */
```

```
(void) fprintf(stderr, "Expected flags for the shmflg argument are:\n");
(void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
(void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
(void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
(void) fprintf(stderr, "\towner write = \t%#8.8o\n", 0200);
(void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
(void) fprintf(stderr, "\tgroup write = \t%#8.8o\n", 020);
(void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
(void) fprintf(stderr, "\tother write = \t%#8.8o\n", 02);
(void) fprintf(stderr, "Enter shmflg: ");
(void) scanf("%i", &shmflg);
```

```
/* Make the call and report the results. */
```

```
(void) fprintf(stderr, "shmget: Calling shmget(%#lx, %d, %#o)\n", key, size, shmflg);
```

```
if ((shmctl = shmget (key, size, shmflg)) == -1)
{
    perror("shmget: shmget failed");
    exit(1);
}
```

```
Else
```

```
{
    (void) fprintf(stderr,
        "shmget: shmget returned %d\n", shmctl);
    exit(0);
}
```

```
shmctl() program
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>
```

```
static void do_shmctl();
extern void exit();
extern void perror();
```

```
main()
```

```
{
    int cmd;          /* command code for shmctl() */
    int shmctl;       /* segment ID */
    struct shmctl_ds shmctl_ds; /* shared memory data structure to hold results */
```

```
fprintf(stderr, "All numeric input is expected to follow C conventions:\n");
fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
fprintf(stderr, "\t0... is interpreted as octal,\n");
fprintf(stderr, "\totherwise, decimal.\n");
```

```
/* Get shmctl and cmd. */
```

```
fprintf(stderr, "Enter the shmctl for the desired segment: ");
scanf("%i", &shmctl);
fprintf(stderr, "Valid shmctl cmd values are:\n");
fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
fprintf(stderr, "Enter the desired cmd value: ");
scanf("%i", &cmd);
```

```
switch (cmd)
```

```

    case IPC_STAT:    /* Get shared memory segment status. */
        break;
    case IPC_SET:     /* Set owner UID and GID and permissions. */
        /* Get and print current values. */
        do_shmctl(shmid, IPC_STAT, &shm_ds);
        /* Set UID, GID, and permissions to be loaded. */
        (void) fprintf(stderr, "\nEnter shm_perm.uid: ");
        (void) scanf("%hi", &shm_ds.shm_perm.uid);
        (void) fprintf(stderr, "Enter shm_perm.gid: ");
        (void) scanf("%hi", &shm_ds.shm_perm.gid);
        (void) fprintf(stderr,);
        (void) fprintf(stderr, "Enter shm_perm.mode: ");
        (void) scanf("%hi", &shm_ds.shm_perm.mode);
        break;
    case IPC_RMID:
        /* Remove the segment when the last attach point is detached. */
        break;
    case SHM_LOCK:    /* Lock the shared memory segment. */
        break;
    case SHM_UNLOCK:  /* Unlock the shared memory segment. */
        break;
    default:          /* Unknown command will be passed to shmctl. */
        break;
}
do_shmctl(shmid, cmd, &shm_ds);
exit(0);
}

/* Display the arguments being passed to shmctl(), call shmctl(), and report the
   results. */
static void
do_shmctl(shmid, cmd, buf)
int shmid,          /* attach point */
    cmd;            /* command code */
struct shm_ds *buf; /* pointer to shared memory data structure */
{
    register int  rtn; /* hold area */
    fprintf(stderr, "shmctl: Calling shmctl(%d, %d, buf)\n", shmid, cmd);
    if (cmd == IPC_SET)
    {
        fprintf(stderr, "\tbuf->shm_perm.uid == %d\n", buf->shm_perm.uid);
        fprintf(stderr, "\tbuf->shm_perm.gid == %d\n", buf->shm_perm.gid);
        fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n", buf->shm_perm.mode);
    }
    if ((rtn = shmctl(shmid, cmd, buf)) == -1)
    {
        perror("shmctl: shmctl failed");
        exit(1);
    }
}

```

```

{
    (void) fprintf(stderr, "shmctl: shmctl returned %d\n", rtn);
}
if (cmd != IPC_STAT && cmd != IPC_SET)
    return;
/* Print the current status. */
fprintf(stderr, "\nCurrent status:\n");
fprintf(stderr, "\tshm_perm.uid = %d\n", buf->shm_perm.uid);
fprintf(stderr, "\tshm_perm.gid = %d\n", buf->shm_perm.gid);
fprintf(stderr, "\tshm_perm.cuid = %d\n", buf->shm_perm.cuid);
fprintf(stderr, "\tshm_perm.cgid = %d\n", buf->shm_perm.cgid);
fprintf(stderr, "\tshm_perm.mode = %#o\n", buf->shm_perm.mode);
fprintf(stderr, "\tshm_perm.key = %#x\n", buf->shm_perm.key);
fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
fprintf(stderr, "\tshm_atime = %s", buf->shm_atime ? ctime(&buf->shm_atime) :
"Not Set\n");
fprintf(stderr, "\tshm_dtime = %s", buf->shm_dtime ? ctime(&buf->shm_dtime) :
"Not Set\n");
fprintf(stderr, "\tshm_ctime = %s", ctime(&buf->shm_ctime));
}

```

University Questions

1. Write the prototypes of system library calls available to manipulate shared memory and semaphores.

VTU : Jan.-15, Marks 7

2. Write short notes on shared memory.

VTU : Jan.-18, Marks 5

5.9 Client - Server Properties

- Following are the some of the properties of client - server architecture :
 1. Clients and servers are separate processes.
 2. They may run on the same or different machines.
 3. Each process can hide internal information.
 4. Each process can implement its own set of business rules.
 5. They communicate by peer to peer protocols.
- The simplest example of this type is client's "fork" and "exec" of server. Two half-duplex pipes can be created before the fork to allow data to be transferred in both directions. Open server is created using this type of arrangement.
- With FIFOs, an individual per client FIFO is also required if the server is to send data back to the client. If the client server application sends data only from the client to the server, a single well-known FIFO suffices.
- Multiple possibilities exist with message queues.
 1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient.

2. An individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue with a key of `IPC_PRIVATE`. The server also has its own queue, with a key or identifier known to all clients.
- One problem with this technique is that each client-specific queue usually has only a single message on it : A request for the server or a response for a client. Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method.

5.10 Stream Pipes

- A stream pipe is a bidirectional pipe. For bidirectional data flow between a parent and child, only a single stream pipe is required. Fig. 5.10.1 shows stream pipe.
- Stream pipe is full duplex. The `fd[0]` is used by parent process and the child uses only `fd[1]`. Each end of the stream pipe is full duplex, the parent reads and writes `fd[0]` and the child duplicates `fd[1]` to both standard input and standard output. Fig. 5.10.2 shows the resulting descriptors.
- The function `s_pipe` is similar to the standard `pipe` function. It takes the same argument as `pipe`, but the returned descriptors are open for reading and writing.
- `s_pipe` function under SVR4 :

```
#include "ourhdr.h"
int s_pipe (int fd[2])
{
    return ( pipe (fd) );
}
```

- Fig. 5.10.3 shows the stream pipe under SVR4.

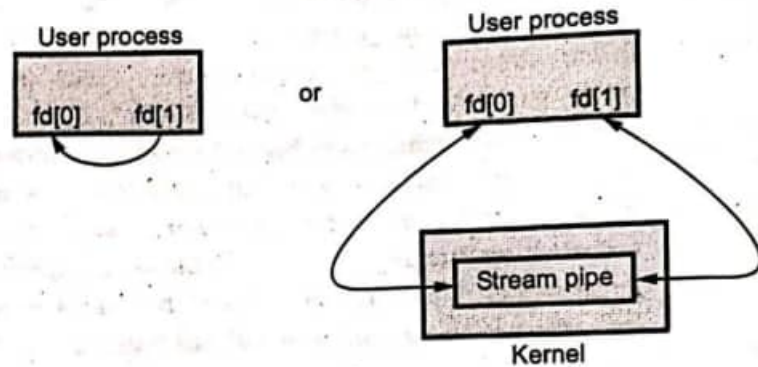


Fig. 5.10.1

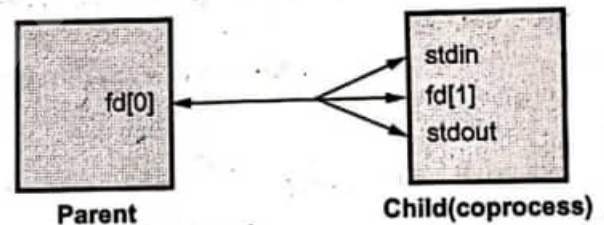


Fig. 5.10.2 Arrangement of descriptors for processes

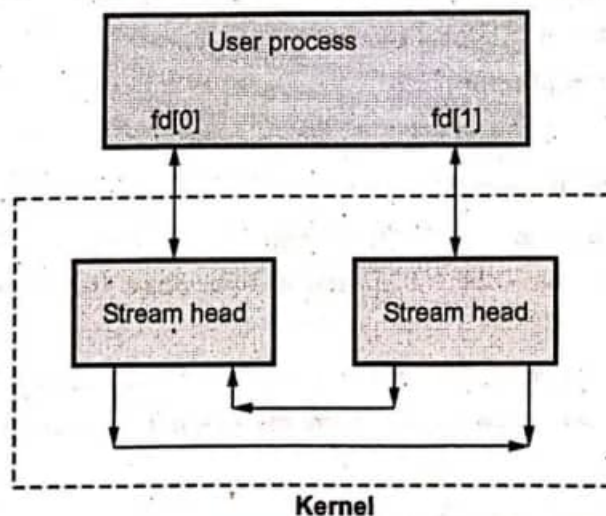


Fig. 5.10.3 Stream pipe under SVR4

- Pipe is stream device, so data can push onto either end of the pipe.
- Following is the code for `s_pipe` function in BSD version. It creates a pair of connected UNIX domain stream sockets.

```
#include <sys/types.h>
#include <sys/socket.h>
int s_pipe (int fd[2])
{
    return ( socketpair (AF_UNIX , SOCK_STREAM, 0, fd );
}
```

Stream-pipe connections have the following advantages :

1. Stream pipes do not pose the security risk of being overwritten or read by other programs that explicitly access the same portion of shared memory.
2. Stream-pipe connections allow distributed transactions between database servers that are on the same computer.

Stream-pipe connections have the following disadvantages :

1. Stream-pipe connections might be slower than shared-memory connections on some computers.
2. Stream pipes are not available on all platforms.
3. When you use stream pipes for client/server communications, the hostname entry is ignored.

University Questions

1. What are pipes ? Explain the different ways to view a half duplex pipe. Write a program to create a pipe between a parent and its child and to send data down the pipe. **VTU : July-14, Marks 10**
2. Explain with a neat diagram, how STREAM PIPES can be used to implement client server model. **VTU : Jan.-15, Marks 10**
3. What are pipes ? Write a C/C++ program to send data from parent to child over a pipe. **VTU : Jan.-17, Marks 10**
4. Explain STREAMs - based pipes. Write a C function that is used by a server to wait for a client's connect request to arrive. **VTU : July-17, Marks 10**
5. Write short notes on any two of the following : Stream pipes **VTU : Jan.-18, Marks 10**

5.11 Passing File Descriptor

- The normal inheritance of file descriptors by child processes suffices for many purposes, one of the most typical being when servers use child processes to deal with clients. When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to also share the same file table entry.
- Fig. 5.11.1 shows passing open file from one process to another process. (See Fig. 5.11.1 on next page)
- When a descriptor is passed from one process to another, sending process closes the descriptor after passing the descriptor. Closing the descriptor by the sender does not really close the file or device, since the descriptor is still considered open by the receiving process.
- In order for a file descriptor to be passed, the sender and receiver must already share a pipe or pump stream for its conveyance. Given a writable file descriptor `pd1` for such an open stream and a file descriptor `fd1` to be passed, the sender executes the command.

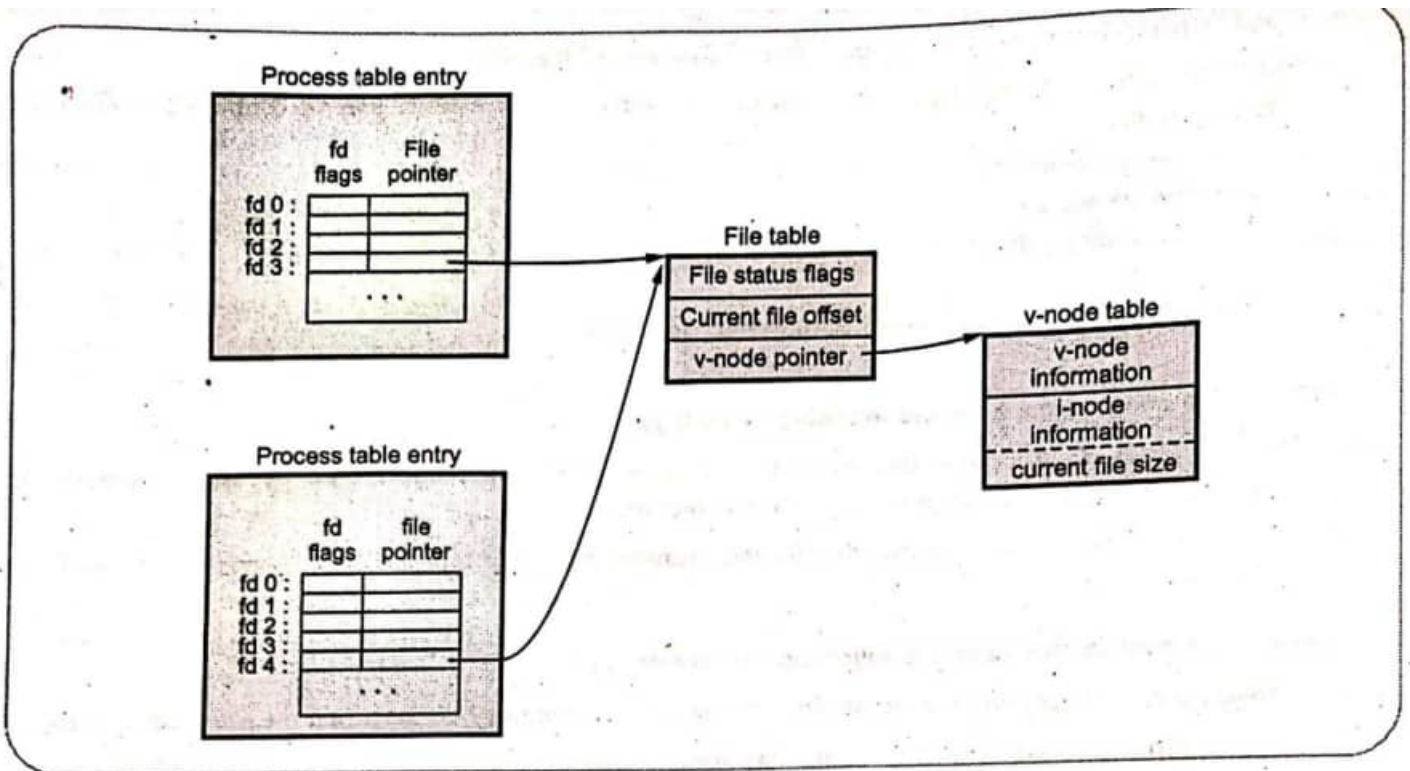


Fig. 5.11.1 Passing an open file

```
#include "ourhdr.h"
```

```
int send_fd (int spipecfd, int filedес);
```

```
int send_err (int spipecfd, int status, const char *errmsg);
```

```
int recv_fd (int spipecfd, ssize_t (*userfun) (int, const void *, size_t));
```

- When a process wants to pass a descriptor to another process it calls either `send_fd` or `send_err`. The process waiting to receive the descriptor calls `recv_fd`.
- The function `send_err` calls the `send_fd` function after writing the error message to the `s_pipe`.

```
int send_err(int fd, int errcode, const char *msg)
```

```
{
    int n;
    if ((n = strlen(msg)) > 0)
        if (written(fd, msg, n) != n) /* send the error message */
            return(-1);
    if (errcode >= 0)
        errcode = -1; /* must be negative */
    if (send_fd(fd, errcode) < 0)
        return(-1);
    return(0);
}
```

5.11.1 System V Release 4

- File descriptors are exchanged using `ioctl` command `I_SENDFD` and `I_RECVFD` in SRV4 on stream pipe. Third argument is used for send descriptor. Following is the program for these purposes.

- The `send_fd` function for STREAMS pipes :

```
#include "apue.h"
#include <stropts.h>

int send_fd(int fd, int fd_to_send)
{
    char buf[2]; /* send_fd()/recv_fd() 2-byte protocol */
    buf[0] = 0; /* null byte flag to recv_fd() */
    if (fd_to_send < 0)
    {
        buf[1] = -fd_to_send; /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    }
    else
    {
        buf[1] = 0; /* zero status means OK */
    }

    if (write(fd, buf, 2) != 2)
        return(-1);
    if (fd_to_send >= 0)
        if (ioctl(fd, I_SENDFD, fd_to_send) < 0)
            return(-1);
    return(0);
}
```

The `strrecvfd` structure :

```
struct strrecvfd
{
    int fd; /* new descriptor */
    uid_t uid; /* effective user ID of sender */
    gid_t gid; /* effective group ID of sender */
    char fill[8];
};
```

- The `recv_fd` function reads the STREAMS pipe until the first byte of the 2-byte protocol is received. When we issue the `I_RECVFD` ioctl command, the next message on the stream head's read queue must be a descriptor from an `I_SENDFD` call, or we get an error.

5.12 An Open Server Version 1

- To develop a open server program i.e. a program that is executed by a process to open one or more files. But instead of sending the contents of the file back to the calling process, the server sends back an open file descriptor.
- Following are the advantages for designing separate program on server are as follows :
 1. The server can easily be contacted by any client, similar to the client calling a library function.
 2. If we need to change the server, only a single program is affected.
 3. The server can be a set-user-ID program, providing it with additional permissions that the client does not have.

- The client process creates an `s_pipe` and then calls `fork` and `exec` to invoke the server. The client sends requests across the `s_pipe`, and the server sends back responses across the `s_pipe`. Following are the application protocol between the client and the server.
 1. The client sends a request of the form "open <pathname> <openmode>\0" across the `s_pipe` to the server.
 2. The server sends back an open descriptor or an error by calling either `send_fd` or `send_err`.
- The client "main" function, version 1 is as follows :

```
#include "open.h"
#include <fcntl.h>

#define BUFFSIZE 8192
int main(int argc, char *argv[ ])
{
    int n, fd;
    char buf[BUFFSIZE], line[MAXLINE];

    /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL)
    {
        if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* replace newline with null */

        /* open the file */
        if ((fd = csopen(line, O_RDONLY)) < 0)
            continue; /* csopen() prints error from server */

        /* and cat to stdout */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
            if (n < 0)
                err_sys("read error");
            close(fd);
    }
    exit(0);
}
```

- The server main function, version 1

```
#include "opend.h"

char errmsg[MAXLINE];
int oflag;
char *pathname;

int main(void)
{
    int nread;
    char buf[MAXLINE];
```

```

for (;;)
{ /* read arg buffer from client, process request */
  if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
    err_sys("read error on stream pipe");
  else if (nread == 0)
    break; /* client has closed the stream pipe */
  request(buf, nread, STDOUT_FILENO);
}
exit(0);
}

```

5.13 Client Server Connection Functions

VTU : Jan.-16, 17

- Here we will discuss three functions that can be used by a client server to establish the functions per client connections.

```
#include "ourhdr.h"
```

```
int serv_listen (const char *name);
```

- Server listens for client connection on a well known name by calling *serv_listen*. Here name is the well known name of the server. When client wants to connect with server then client used this name. The return value is the file descriptor for the server's end of the named stream pipe.
- Once a server has called *serv_listen*, it calls *serv_accept* to wait for a client connection to arrive.

```
#include "ourhdr.h"
```

```
int serv_accept (int listenfd, uid_t *uidptr);
```

where *listenfd* is a descriptor from *serv_listen*.

- When the client does connect to the server, a new stream pipe is automatically created, and the new descriptor is returned as the value of the function. The effective user ID of the client is stored through the pointer *uidptr*.
- A client just calls *cli_conn* to connect to a server.

```
#include "ourhdr.h"
```

```
int cli_conn (const char *name);
```

where name specified by the client must be the same name that was advertised by the server's call to *serv_listen*.

- All above three functions are used for writing server daemons to handle multiple clients. The number of descriptor available to a single process is only limit to handle the multiple clients. Server requires one descriptor for each client connection. The client-server connections are all stream pipes, open descriptors can be passed across the connections.

5.13.1 System V Release 4

- SVR4 provides mounted streams and a streams processing module named *connl*d that we can use to provide a named stream pipe with unique connections for the server. Initially server creates an unnamed stream pipe and pushes the stream processing module *connl*d on one end. Fig. 5.13.1 shows SVR4 pipe after pushing *connl*d.
- SVR4 uses *fattach* function to attach a pathname to the end of the pipe that has the *connl*d pushed onto it.

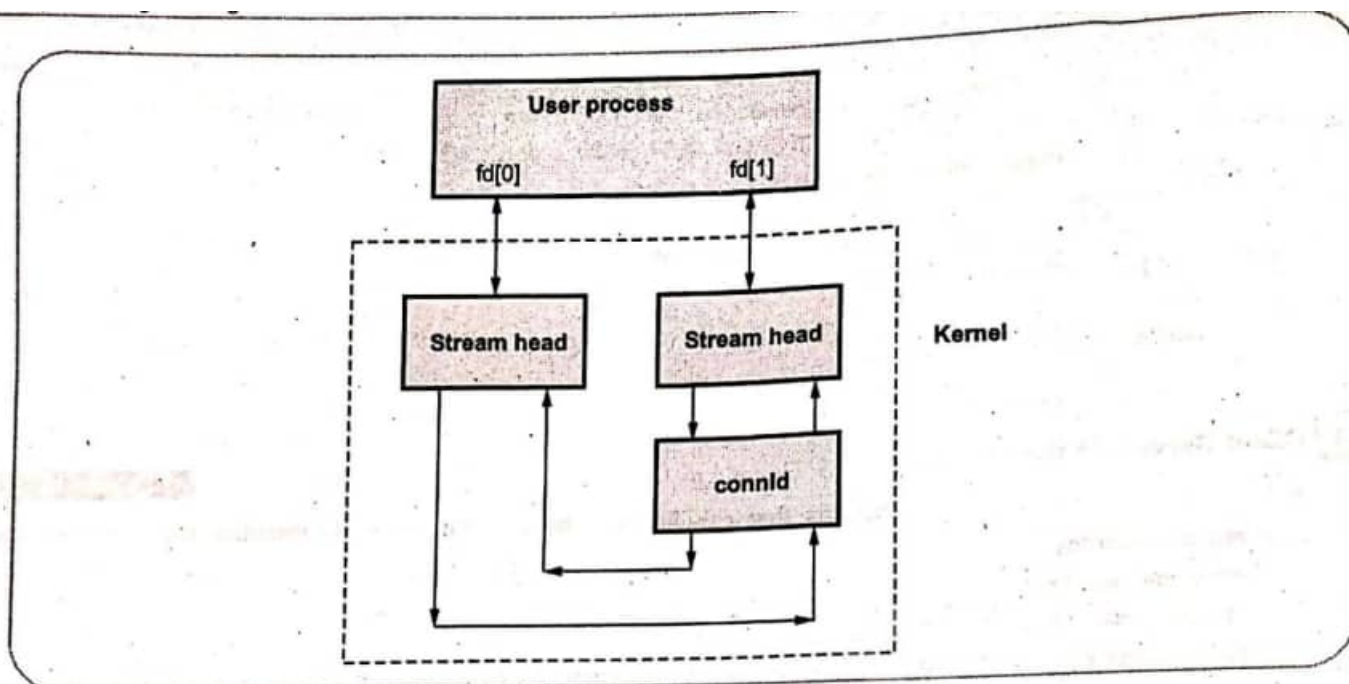


Fig. 5.13.1 SVR4 pipe after pushing connld

- The serv_listen function using STREAMS pipes is as follows :

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>
```

```
/* pipe permissions: user rw, group rw, others rw */
```

```
#define FIFO_MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
```

```
/* Establish an endpoint to listen for connect requests.*/
```

```
int serv_listen(const char *name)
```

```
{
```

```
    int  tempfd;
```

```
    int  fd[2];
```

```
/* Create a file: mount point for fattach( ). */
```

```
unlink(name);
```

```
if ((tempfd = creat(name, FIFO_MODE)) < 0)
```

```
    return(-1);
```

```
if (close(tempfd) < 0)
```

```
    return(-2);
```

```
if (pipe(fd) < 0)
```

```
    return(-3);
```

```
/* Push connld & fattach() on fd[1]. */
```

```
    if (ioctl(fd[1], I_PUSH, "connld") < 0)
```

```
    {
```

```
        close(fd[0]);
```

```
        close(fd[1]);
```

```
        return(-4);
```

```
    }
```

```

if (fattach(fd[1], name) < 0)
{
    close(fd[0]);
    close(fd[1]);
    return(-5);
}

```

```

    close(fd[1]); /* fattach holds this end open */

```

```

return(fd[0]); /* fd[0] is where client connections arrive */
}

```

- When another process call a open for the named end of the pipe, the following occurs :
 1. A new pipe is created.
 2. One descriptor for new pipe is passed back to the client as the return value from open.
 3. The other descriptor is passed to the server on the other end of the named pipe.
- Fig. 5.13.2 shows client-server connection on a named stream pipe. The pipe between the client and server is created by the open. The file descriptor in the client (fd) is returned by the open. The new file descriptor in the server is received by the server using an *ioctl* of *I_RECVFD* on the descriptor fd[0]. Once the server has pushed *connld* onto fd[1] and attached a name to fd[1], it never specifically uses fd[1] again.

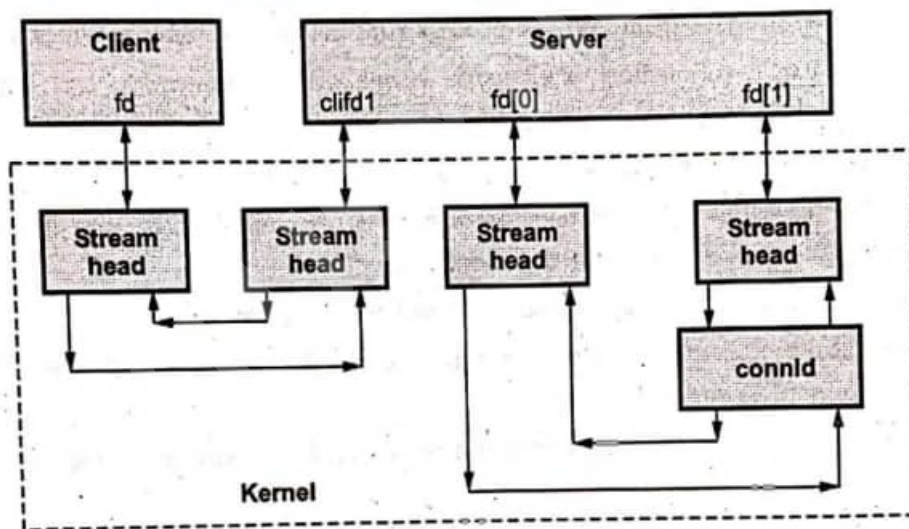


Fig. 5.13.2 Client-server connection on a named stream pipe

- The server waits for a client connection to arrive by calling the *serv_accept* function. It shown in the following program.

```

#include <sys/types.h>
#include <sys/stat.h>
#include <stropts.h>
#include "ourhdr.h"

```

```

/* Wait for a client connection to arrive, and accept it. */
int serv_accept(int listenfd, uid_t *uidptr)
/* returns new fd if all OK, -1 on error */
{
    struct strrecvfd recvfd;

```

```
if (ioctl(listenfd, I_RECVFD, &recvfd) < 0)
    return(-1); /* could be ENTER if signal caught */

if (uidptr != NULL)
    *uidptr = recvfd.uid; /* effective uid of caller */

return(recvfd.fd); /* return the new descriptor */
}
```

University Questions

1. Write short notes on client server properties.
2. Explain the different client server connection functions, with example program.

VTU : Jan.-16, Marks 8

VTU : Jan.-17, Marks 10