MODULE 4

CHAPTER 1

PROCESS CONTROL

PROCESS CONTROL

Process control is concerned about creation of new processes, program execution, and process termination.

PROCESS IDENTIFIERS

#include <unistd.h>

pid_t getpid(void);

Returns: process ID of calling process

pid_t getppid(void);

Returns: parent process ID of calling process

uid_t getuid(void);

Returns: real user ID of calling process

uid_t geteuid(void);

Returns: effective user ID of calling process

gid_t getgid(void);

Returns: real group ID of calling process

gid_t getegid(void);

Returns: effective group ID of calling process

fork Fuction

An existing process can create a new one by calling the fork function.
Returns: 0 for child process 1 on error.

- The new process created by fork is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID

0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

- Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a copy of the parent.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment.

Example programs:

Program 1: Program to demonstrate fork function Program name – fork1.c */

```
#include<sys/types.h>
#include<unistd.h>
int main()
{
    fork( );
    printf("\n hello UP");
}
```

Output :

```
$ cc fork1.c
$ ./a.out

hello UP
hello UP
```

Note: The statement hello UP is executed twice as both the child and parent has executed that instruction.

Program 2

/* Program name – fork2.c */

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

int main( )
{
      printf("\n 5th sem ");
      fork( );
      printf("\n hello UP");
}
```

Output :

```
5th sem
hello UP
hello UP
```

Note: The statement 5th sem is executed only once by the parent because it is called before fork and statement hello UP is executed twice by child and parent.

vfork Function

- The function vfork has the same calling sequence and same return values as fork.
- The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork.
- Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.
- Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.

Example of vfork function

```
#include "apue.h"
int  glob = 6;     /* external variable in initialized data */

int main(void)
{
        int        var;                                      /* automatic variable on the stack */
        pid_t    pid;
        var = 88;
        printf("before vfork\n");

        if ((pid = vfork()) < 0)
        {
              err_sys("vfork error");
        }

       else if (pid == 0)
       {        /* child */
             glob++;                                        /* modify parent's variables */ var++;
            _exit(0);                                      /* child terminates */
       }

printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);           /* Parent continues here */
exit(0);
}
```

Output:

```
 before vfork
         pid = 29039, glob = 7, var = 89
```

exit() Function

A process can terminate normally in five ways:

- Executing a return from the main function.
- Calling the exit function.
- Calling the _exit or _Exit function.
- Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
- Calling the pthread_exit function from the last thread in the process.

The three forms of abnormal termination are as follows:

- Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
- When the process receives certain signals. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.
- The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

wait() and waitpid()

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.

A process that calls wait or waitpid can:

- ✓ Block, if all of its children are still running
- ✓ Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- ✓ Return immediately with an error, if it doesn't have any child processes.

Syntax/Prototype of wait and waitpid:

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0, or 1 on error.

The differences between these two functions are as follows.

- ➢ The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- ➢ The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

Print a description of the exit status

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

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

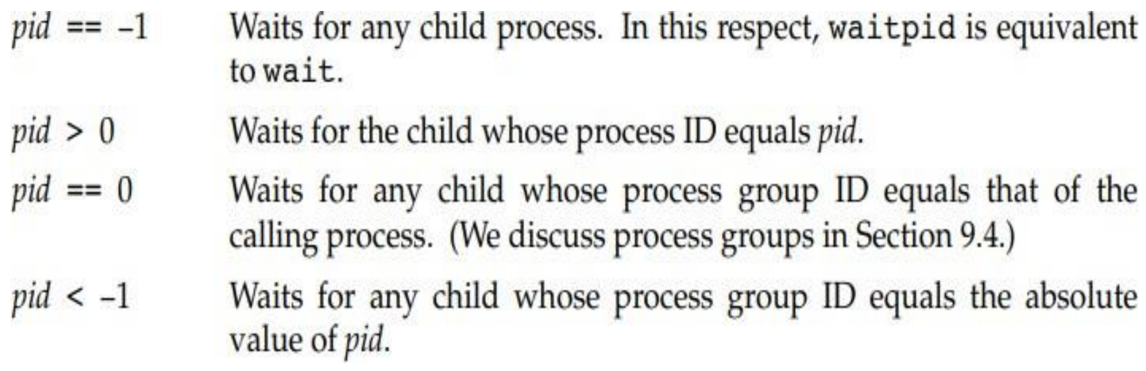
Program to Demonstrate various exit status

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

int
main(void)
{
    pid_t   pid;
    int     status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

The program shown above calls the pr_exit function, demonstrating the various values for the termination status. If we run the program we get-

```
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

The interpretation of the pid argument for waitpid depends on its value:

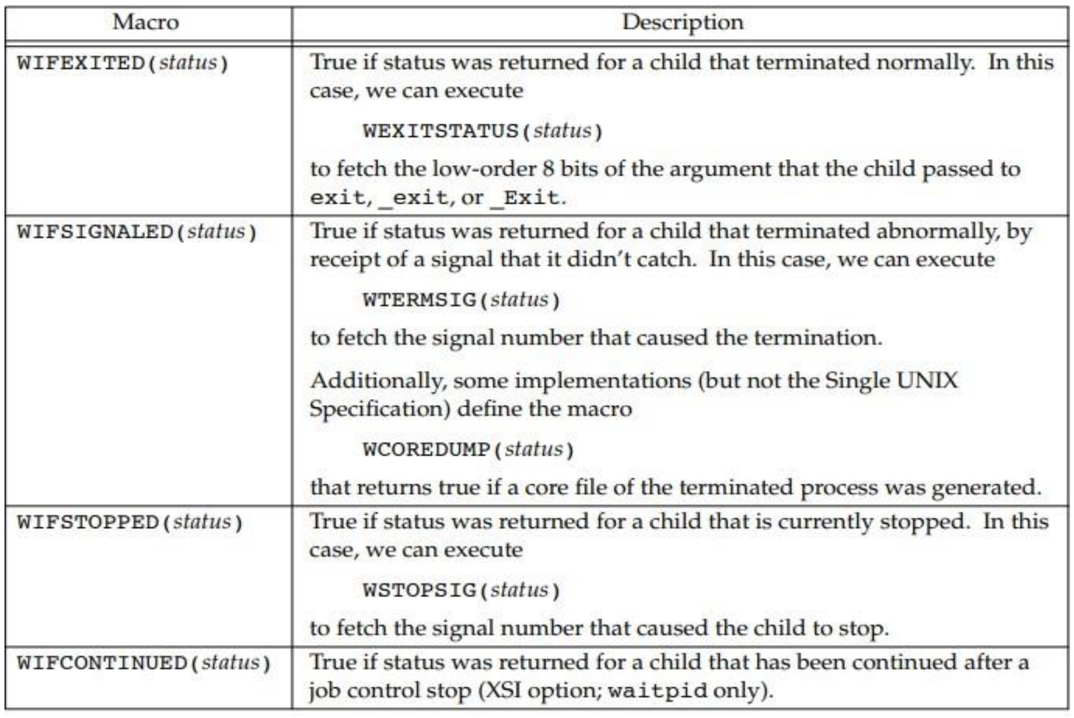


pid == –1	Waits for any child process. In this respect, `waitpid` is equivalent to `wait`.
pid > 0	Waits for the child whose process ID equals *pid*.
pid == 0	Waits for any child whose process group ID equals that of the calling process. (We discuss process groups in Section 9.4.)
pid < –1	Waits for any child whose process group ID equals the absolute value of *pid*.

The waitpid function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by statloc.

With wait, the only real error is if the calling process has no children. (Another error return is possible, in case the function call is interrupted by a signal. With waitpid, however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process).

The options argument lets us further control the operation of waitpid. This argument either is 0 or is constructed from the bitwise OR of the constants.



Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.
`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute `WTERMSIG(`*status*`)` to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro `WCOREDUMP(`*status*`)` that returns true if a core file of the terminated process was generated.
`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`

The options constants for waitpid	
Constant	**Description**
WCONTI NUED	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned.
WNOHAN G	The waitpid function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
WUNTRA CED	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The WIFSTOPPED macro determines whether the return value corresponds to a stopped child process.

The waitpid function provides three features that aren't provided by the wait function.

- The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child.
- The waitpid function provides a nonblocking version of wait. There are times when we want to fetch a child's status, but we don't want to block.
- The waitpidfunction provides support for job control with the WUNTRACED and WCONTINUED options.

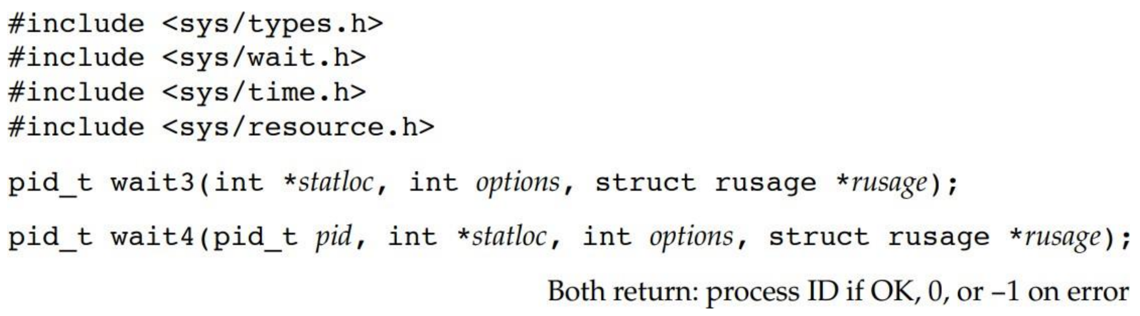
Program to Avoid zombie processes by calling fork twice

```
#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)
           {
                 if ((pid = fork()) < 0)                          /* first child */
                     err_sys("fork error"); else if (pid > 0)
                      exit(0);                              /* parent from second fork == first child */

printf("second child, parent pid = %d\n", getppid());
exit(0);
}

if (waitpid(pid, NULL, 0) != pid)                  /* wait for first child */
     err_sys("waitpid error");
}
```

Output: second child, parent pid = 1

wait3 and wait4 function

The only feature provided by these two functions that isn't provided by the wait, waitid, and waitpid functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.
The prototypes of these functions are:



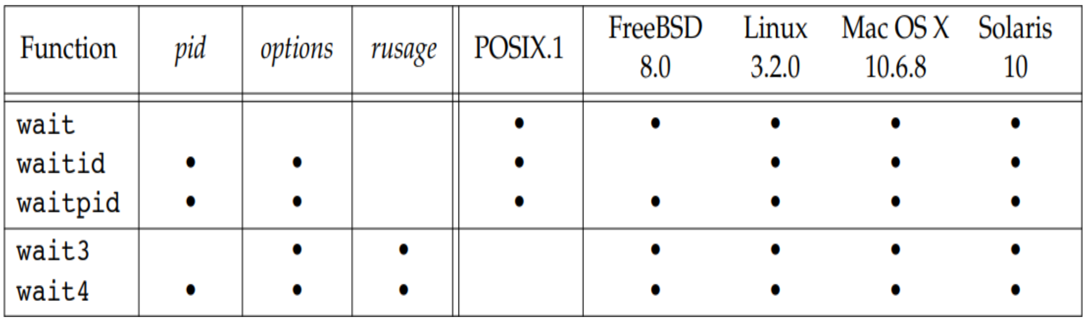
```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK, 0, or –1 on error

Both return: process ID if OK,-1 on error

The resource information includes such statistics as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received etc. the resource information is available only for terminated child process not for the process that were stopped due to job control.

Arguments supported by wait functions on various systems

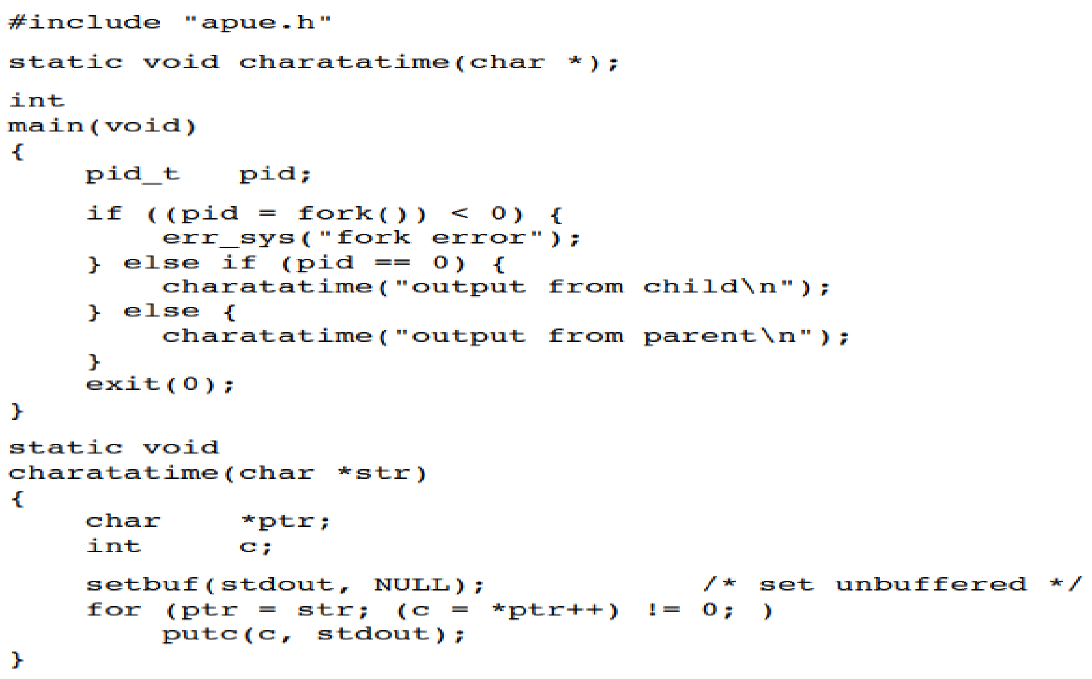


Function	*pid*	*options*	*rusage*	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`wait`				•	•	•	•	•
`waitid`	•	•		•		•	•	•
`waitpid`	•	•		•	•	•	•	•
`wait3`		•	•		•	•	•	•
`wait4`	•	•	•		•	•	•	•

Race Condition

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the process runs.

Example: The program below outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.



```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Output:

ooutput from child utput from parent

output from child output from parent

Program modification to avoid Race Condition

```
        #include "apue.h"
        static void charatatime(char *);
        int
        main(void)
        {
            pid_t   pid;
+           TELL_WAIT();
+
            if ((pid = fork()) < 0) {
                err_sys("fork error");
            } else if (pid == 0) {
+               WAIT_PARENT();      /* parent goes first */
                charatatime("output from child\n");
            } else {
                charatatime("output from parent\n");
+               TELL_CHILD(pid);
            }
            exit(0);
        }
        static void
        charatatime(char *str)
        {
            char    *ptr;
            int     c;
            setbuf(stdout, NULL);           /* set unbuffered */
            for (ptr = str; (c = *ptr++) != 0; )
                putc(c, stdout);
        }
```

In the above program the parent goes first. The child goes first if we change the lines following the fork to be

```
} else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();          /* child goes first */
    charatatime("output from parent\n");
}
```

exec() Function

When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.

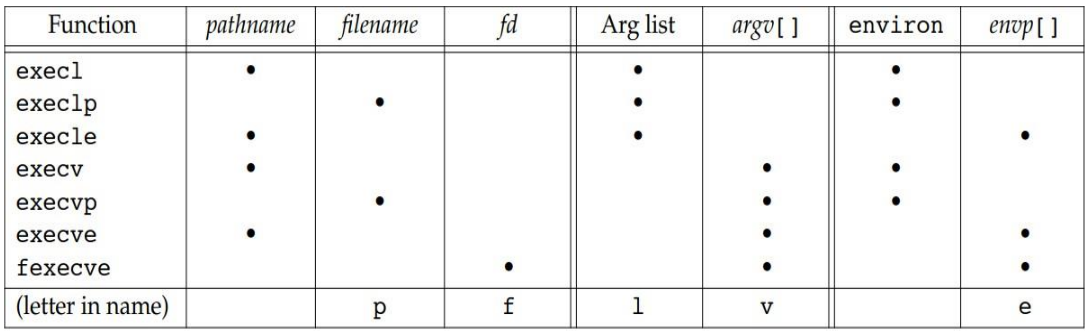
There are 6 exec functions:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0,... /*(char *)0, char* const envp */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv []);
```

All six return: -1 on error, no return on success

- The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified.If filename contains a slash, it is taken as a pathname otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
- The next difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execlerequire each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- The final difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

The below table shows the differences among the 7 exec functions.



Function	*pathname*	*filename*	*fd*	Arg list	*argv*[ ]	environ	*envp*[ ]
execl	•			•		•	
execlp		•		•		•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•				•		•
fexecve			•		•		•
(letter in name)		p	f	l	v		e

Figure 8.14 Differences among the seven exec functions

Process ID does not change after an exec, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Values for tms_utime, tms_stime, tms_cutime, and tms_cstime.

Relationship of the six exec functions

```
#include "apue.h"
#include <sys/wait.h>
char          *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main(void)
{
            pid_t pid;
            if ((pid = fork()) < 0)
            {
                err_sys("fork  error");
             }
             else if (pid == 0)
             {                                          /* specify pathname, specify environment */
                     if (execle("/home/sar/bin/echoall", "echoall", "myarg1", "MY ARG2", (char
*)0, env_init) < 0)
                      err_sys("execle error");
}
              if (waitpid(pid, NULL, 0) < 0)
                 err_sys("wait error");

             if ((pid = fork()) < 0)
             {
                   err_sys("fork error");
              }
              else if (pid == 0) { /* specify filename, inherit environment */
              if (execlp("echoall", "echoall", "only 1 arg", (char *)0) <0)
                   err_sys("execlp error");
}
exit(0);
}
```

Output:

```
argv[0]: echoall argv[1]: myarg1 argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall argv[1]:
only 1 arg USER=sar LOGNAME=sar
SHELL=/bin/bash
HOME=/home/sar       47 more lines that aren't shown
```

Note that the shell prompt appeared before the printing of argv[0] from the second exec. This is because the parent did not wait for this child process to finish.

CHAPTER 2

OVERVIEW OF IPC METHODS

PIPES

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems.

Pipes have two limitations.

1. 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.

2. Pipes can be used only between processes that have a common ancestor.

Normally,apipe is created by a process, that process calls fork, and the pipe is

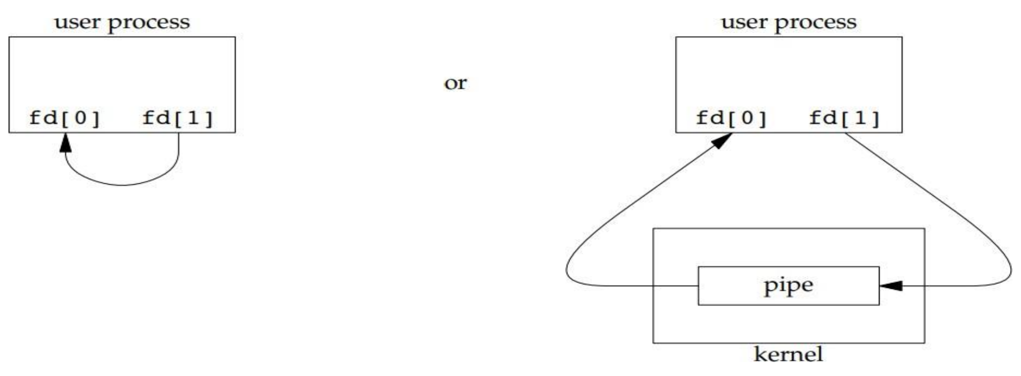
used between the parent and the child.

A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int fd[2]);
Returns: 0 if OK, −1 on error
```

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

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



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

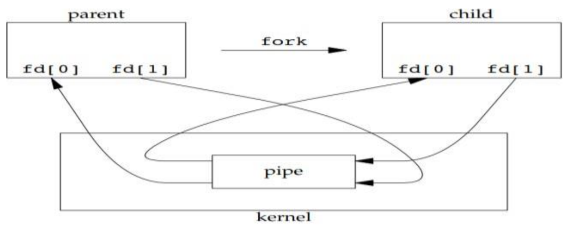


Figure 15.3 Half-duplex pipe after a fork

What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Figure 15.4 shows the resulting arrangement of descriptors.

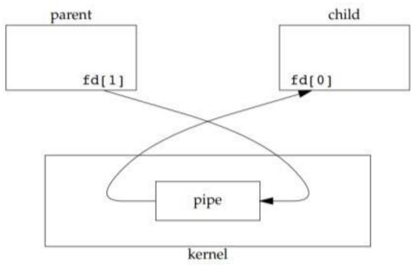


Figure 15.4 Pipe from parent to child

For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

When one end of a pipe is closed, two rules are applie

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. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe.
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.

Example

The below code shows how to send data from parent to child process through a pipe

```
#include "apue.h"

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

        if (pipe(fd) < 0)
            err_sys("pipe error");
        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid > 0) {           /* parent */
            close(fd[0]);
            write(fd[1], "hello world\n", 12);
        } else {                        /* child */
            close(fd[1]);
            n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line, n);
        }
        exit(0);
}
```

popen and pclose Functions

Since a common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions. These two functions handle all the dirty work that we've 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.

#include <stdio.h>

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

Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);

Returns: termination status of cmdstring, or −1 on error

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

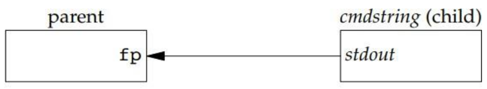


Figure 15.9 Result of fp = popen(*cmdstring*, "r")

- If type is "w", the file pointer is connected to the standard input of cmdstring, as shown in Figure.

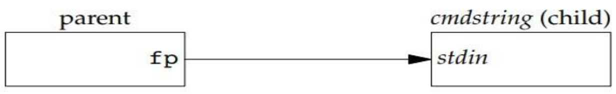


Figure 15.10 Result of fp = popen(*cmdstring*, "w")

One way to remember the final argument to popen is to remember that, like fopen, the returned file pointer is readable if type is "r" or writable if type is "w". The pclose function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell. If the shell cannot be executed, the termination status returned by pclose is as if the shell had executed exit(127).

The cmdstring is executed by the Bourne shell, as in sh -c cmdstring. This means that the shell expands any of its special characters in cmdstring. This allows us to say, for example,

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

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

Coprocesses:

- A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a coprocess when the same program generates the filter's input and reads the filter 's output.
- A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe. Although the shell syntax required to initiate a coprocess and connect its input and output to other processes is quite contorted, coprocesses are also useful from a C program.
- Whereas popen gives us a one-way pipe to the standard input or from the standard output of another process, with a coprocess we have two one-way pipes to the other process: one to its standard input and one from its standard output. We want to write to its standard input, let it operate on the data, and then read from its standard output.
- Let's look at coprocesses with an example. The process creates two pipes: one is the standard input of the coprocess and the other is the standard output of the coprocess. Figure shows this arrangement.

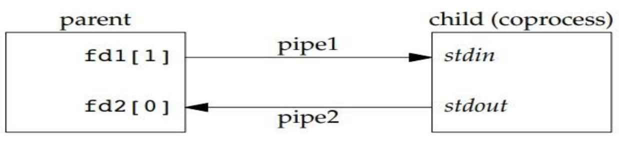


Figure 15.16 Driving a coprocess by writing its standard input and reading its standard output

- The below program is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output.

```
#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);
}
```

Figure 15.17 Simple filter to add two numbers

- Filters to add 2 numbers using standard input:

```
#include "apue.h"

int
main(void)
{
    int     int1, int2;
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("printf error");
        } else {
            if (printf("invalid args\n") == EOF)
                err_sys("printf error");
        }
    }
    exit(0);
}
```

FIFOs

FIFOs are sometimes called named pipes. Unnamed pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data. Creating a FIFO is similar to creating a file. Indeed, the pathname for a FIFO exists in the file system.

Prototype of FIFO Function:

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

int mkfifo(const char *path, mode_t mode);

int mkfifoat(int fd, const char *path, mode_t mode);

Both return: 0 if OK, −1 on error

- The mkfifoat function is similar to the mkfifo function, except that it can be used to create a FIFO in a location relative to the directory represented by the fd file descriptor argument. Like the other *at functions, there are three cases:
  1. If the path parameter specifies an absolute pathname, then the fd parameter is ignored and the mkfifoat function behaves like the mkfifo function.
  2. If the path parameter specifies a relative pathname and the fd parameter is a valid file descriptor for an open directory, the pathname is evaluated relative to this directory.
  3. If the path parameter specifies a relative pathname and the fd parameter has the special value AT_FDCWD, the pathname is evaluated starting in the current working directory, and mkfifoat behaves like mkfifo. Once we have used mkfifo or mkfifoat to create a FIFO, we open it using open. Indeed, the normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.

Once we have used mkfifo or mkfifoat to create a FIFO, we open it using open. Indeed, the normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.

When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

• In the normal case (without O_NONBLOCK), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for writeonly 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.

As with a pipe, if we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.

It is common to have multiple writers for a given FIFO. This means that we have to worry about atomic writes if we don't want the writes from multiple processes to be interleaved. As with pipes, the constant PIPE_BUF specifies the maximum amount of data that can be written atomically to a FIFO.

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. We discuss each of these uses with an example.

Example : Using FIFOs to Duplicate Output Streams

Consider a procedure that needs to process a filtered input stream twice.

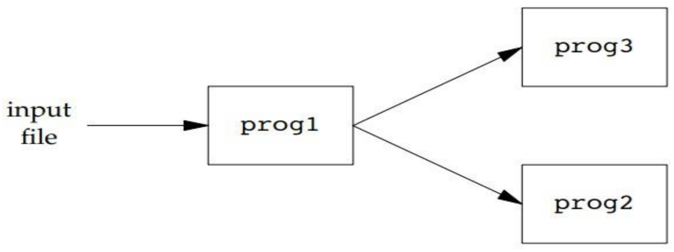


Figure 15.20 Procedure that processes a filtered input stream twice

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

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2. Figure 15.21 shows the process arrangement.

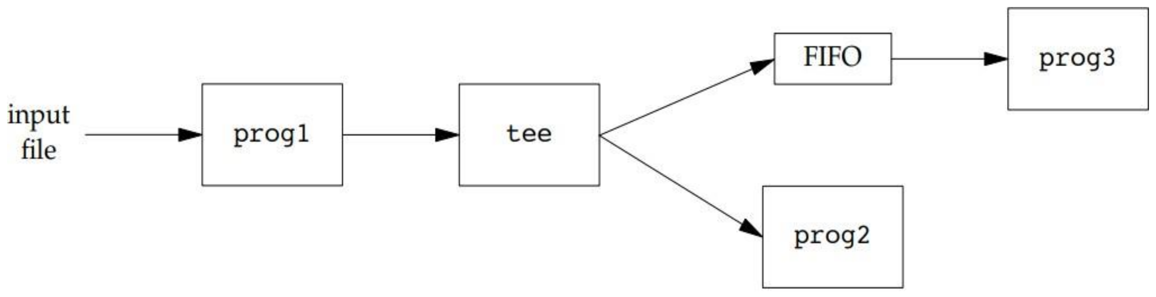


Figure 15.21 Using a FIFO and `tee` to send a stream to two different processes

Example : Client–Server Communication Using a FIFO

Another use for FIFOs is to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. (By ‘‘well-known,’’ we mean that the pathname of the FIFO is known to all the clients that need to contact the server.)

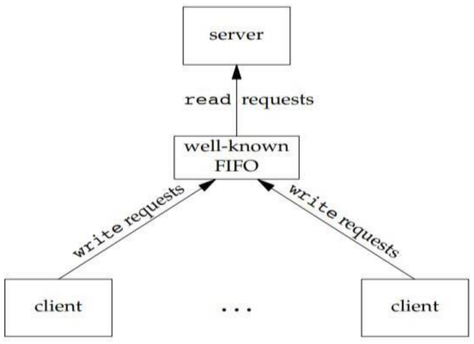


Figure 15.22 Clients sending requests to a server using a FIFO

Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size. This prevents any interleaving of the client writes.

The problem in using FIFOs for this type of client–server communication is how to send replies back from the server to each client. A single FIFO can’t be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client’s process ID.

For example, the server can create a FIFO with the name /tmp/serv1.XXXXX, where XXXXX is replaced with the client’s process ID. This arrangement is shown in Figure 15.23.

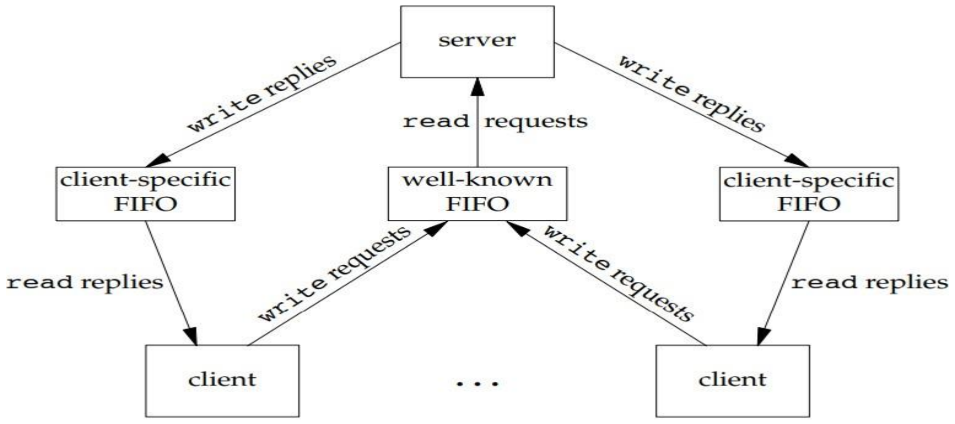


Figure 15.23 Client–server communication using FIFOs

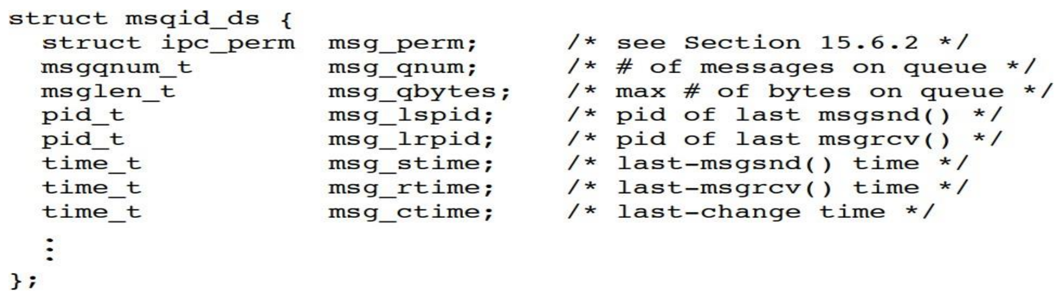
This arrangement works, although it is impossible for the server to tell whether a client crashes. A client crash leaves the client-specific FIFO in the file system. The server also must catch SIGPIPE, since it’s possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader. With the arrangement shown in Figure 15.23, 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 read–write.

Message Queues

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

A new queue is created or an existing queue opened by **msgget**. New messages are added to the end of a queue by **msgsnd**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd when the message is added to a queue. Messages are fetched from a queue by **msgrcv**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following msqid_ds structure associated with it:



```
struct msqid_ds {
  struct ipc_perm  msg_perm;      /* see Section 15.6.2 */
  msgqnum_t        msg_qnum;      /* # of messages on queue */
  msglen_t         msg_qbytes;    /* max # of bytes on queue */
  pid_t            msg_lspid;     /* pid of last msgsnd() */
  pid_t            msg_lrpid;     /* pid of last msgrcv() */
  time_t           msg_stime;     /* last-msgsnd() time */
  time_t           msg_rtime;     /* last-msgrcv() time */
  time_t           msg_ctime;     /* last-change time */
    ⋮
};
```

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

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

• The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.

• msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are all set to 0.

• msg_ctime is set to the current time.

• msg_qbytes is set to the system limit

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

The **msgctl** function performs various operations on a queue. This function and the related functions for semaphores and shared memory (semctl and shmctl) are the ioctl-like functions for XSI IPC (i.e., the garbage-can functions).

#include<sys/msg.h>

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

Returns: 0 if OK, −1 on error

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

1. **IPC_STAT** Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.
2. **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.
3. **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.

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

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

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

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

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

A flag value of **IPC_NOWAIT** can be specified. This is similar to the nonblocking I/O flag for file I/O . If the message queue is full (either the total number of messages on the queue equals the system limit, or the total number of bytes on the queue equals the system limit), specifying IPC_NOWAIT causes msgsnd to return immediately with an error of EAGAIN.

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. In the second case, an error of EIDRM is returned (‘‘identifier removed’’); in the last case, the error returned is EINTR.

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).

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

If the message is too big and this flag value is not specified, an error of E2BIG is returned instead (and the message stays on the queue).

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 absolute value of the type is returned.

When msgrcv succeeds, the kernel updates the msqid_ds structure associated with the message queue to indicate the caller’s process ID (msg_lrpid), the time of the call (msg_rtime), and that one less message is on the queue (msg_qnum).

Semaphores:

A semaphore isn't a form of IPC similar to the others that we've described (pipes, FIFOs, and message queues). A semaphore is a counter used to provide access to a shared data object for multiple processes.

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

1. Test the semaphore that controls the resource.

2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.

3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

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

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.

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

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

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.

2. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.

3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

The kernel maintains a **semid_ds structure** for each semaphore set:

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

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

When we want to use XSI semaphores, we first need to obtain a semaphore ID by calling the **semget** function.

#include int semget(key_t key, int nsems, int flag);

Returns: semaphore ID if OK, −1 on error

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

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

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

The **semctl** function is the catchall for various semaphore operations.

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

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

                                                   Returns: (see following)
```

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

1) **IPC_STAT** Fetch the semid_ds structure for this set, storing it in the structure pointed to by arg.buf.
2) **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.

3) **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.
4) **GETVAL** Return the value of semval for the member semnum.
5) **SETVAL** Set the value of semval for the member semnum. The value is specified by arg.val.
6) **GETPID** Return the value of sempid for the member semnum.
7) **GETNCNT** Return the value of semncnt for the member semnum. GETZCNT Return the value of semzcnt for the member semnum.
8) **GETZCNT** Return the value of semzcnt for the member semnum.
9) **GETALL** Fetch all the semaphore values in the set. These values are stored in the array pointed to by arg.array.
10) **SETALL** Set all the semaphore values in the set to the values pointed to by arg.array.

For all the GET commands other than GETALL, the function returns the corresponding value. For the remaining commands, the return value is 0 if the call succeeds. On error, the semctl function sets errno and returns −1.

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

#include <sys/sem.h>

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

Returns: 0 if OK, −1 on error

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

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

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

The operation on each member of the set is specified by the corresponding sem_op value. This value can be negative, 0, or positive.

1. 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.
2. 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 the resulting semaphore value 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.

3. 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.

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

CHAPTER 3

SHARED MEMORY

Shared Memory

Shared memory allows two or more processes to shareagiven region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access.

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

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

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

#include<sys/shm.h>

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

Returns: shared memory ID if OK, −1 on error

When a new segment is created, the following members of the **shmid_ds structure are initialized.**

- The ipc_perm structure The mode member of this structure is set to the corresponding permission bits of flag.
- shm_lpid, shm_nattch, shm_atime, and shm_dtime are all set to 0.
- shm_ctime is set to the current time.
- shm_segsz is set to the size requested.

If a new segment is being created (typically by the server), we must specify its size. If we are referencing an existing segment (a client), we can specify size as 0. When a new segment is created, the contents of the segment are initialized with zeros.

The **shmctl** function is the catchall for various shared memory operations.

#include <sys/shm.h>

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

Returns: 0 if OK, −1 on error

The cmd argument specifies one of the following five commands to be performed, on the segment specified by shmid.

IPC_STAT Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by buf.

IPC_SET Set the following three fields from the structure pointed to by buf in the shmid_ds structure associated with this shared memory segment: shm_perm.uid, shm_perm.gid, and shm_perm.mode. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.

IPC_RMID Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the shm_nattch field in the shmid_ds structure), the segment is not removed until the last process using the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that shmat can no longer attach the segment. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.

Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.

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

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

Returns: pointer to shared memory segment if OK, −1 on error
```

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.

• If addr is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.

• If addr is nonzero and SHM_RND is not specified, the segment is attached at the address given by addr. • If addr is nonzero and SHM_RND is specified, the segment is attached at the address given by (addr − (addr modulus SHMLBA)). The SHM_RND command stands for ‘‘round.’’ SHMLBA stands for ‘‘low boundary address multiple’’ and is always a power of 2.

When we're done with a shared memory segment, we call **shmdt** to detach it. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling shmctl with a command of IPC_RMID.

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

int shmdt(const void *addr);

Returns: 0 if OK, −1 on error
```

The addr argument is the value that was returned by a previous call to shmat. If successful, shmdt will decrement the shm_nattch counter in the associated shmid_ds structure.

CLIENT SERVER PROPERTIES

The properties of clients and servers, along with their interactions, are deeply influenced by the type of inter-process communication (IPC) mechanisms employed. Let's analyze the key points and implications based on the given IPC scenarios:

Fork and Exec Model with Half-Duplex Pipes

1. **Setup and Communication:**
    - Two half-duplex pipes are created before the fork to facilitate bidirectional communication.
    - The server, which is executed after the fork, can have enhanced permissions (e.g., being a set-user-ID program).
    - The server can identify the client using its real user ID.
2. **Advantages:**
    - Simple and straightforward for parent-child communication.
    - Allows the server to enforce additional permissions and control access to resources based on the client's identity.
    - Can create open servers that offer functionality like opening files for clients.
3. **Limitations:**
    - Pipes are suitable for simple data exchange but lack advanced features like message prioritization or asynchronous communication.
    - Pipes cannot effectively handle special device files; they are more suited to transferring regular file data.

Message Queues:

Message queues offer flexibility and scalability for client-server communication. Two main models are possible:

Single Message Queue for All Clients

1. **Setup:**
    - A single queue is shared between the server and all clients.
    - The type field in messages is used to differentiate between clients and to route messages.

2. **Message Flow:**
    - Clients send requests with a specific type (e.g., type = 1).
    - Each message includes the client's process ID.
    - The server responds with messages tagged with the client's process ID, ensuring only the intended client retrieves it.
3. **Advantages:**
    - Easy to manage since only one queue is maintained.
    - Efficient for a moderate number of clients where message routing based on the type field suffices.
4. **Limitations:**
    - Potential contention when many clients share the same queue.
    - Complexity increases as the number of clients grows.

Individual Message Queues for Each Client

1. **Setup:**
    - Each client creates its own message queue and uses it to communicate with the server.
2. **Advantages:**
    - Eliminates contention issues seen with a shared queue.
    - Provides a direct, isolated communication channel for each client.
3. **Limitations:**
    - More resources are required to manage multiple queues.
    - Additional complexity in the setup and teardown of queues.
    -

Open Server Concept

1. **Functionality:**
    - The server acts as an intermediary, performing actions like opening files for clients.
    - It enforces additional access checks based on the real user ID of the client.
    - This design can grant specific permissions (e.g., root-level privileges) to clients under controlled conditions.

2. **Security Implications:**
    - By being a set-user-ID program, the server operates with elevated privileges, enabling it to bypass standard user/group/other permission checks.
    - The server must validate requests rigorously to prevent privilege escalation or misuse.
3. **Use Cases:**
    - Ideal for environments where clients require limited access to resources beyond their normal permissions.
    - Useful in implementing advanced access control mechanism
    -

Overall Implications

- **Performance and Scalability:**
    - Single message queues offer simplicity but may struggle with high concurrency.
    - Multiple message queues scale better but demand more system resources.
- **Security and Flexibility:**
    - Set-user-ID programs provide flexibility in access control but require careful handling to avoid vulnerabilities.
    - The fork and exec model with pipes is simpler and suitable for controlled environments but lacks the flexibility of message queues.

Passing File Descriptor:

Passing file descriptors between processes is a powerful IPC technique that enables the sharing of open file states without duplicating the effort or context required to open and configure the file.

Concept: Passing an Open File Descriptor

1. **What is Passed?**
    - Technically, a pointer to an open file table entry is passed, not the file descriptor itself.
    - The receiving process assigns this pointer to its first available descriptor slot.
2. **Behavior of the Descriptor:**
    - The file descriptor in the receiving process may have a different number than in the sending process.
    - The processes share the same file table entry, maintaining consistency for operations like reading, writing, and seeking.

Advantages of Passing File Descriptors

1. **Encapsulation of Complexity:**
    - The server can perform all the complex tasks (e.g., resolving names, configuring files, or negotiating locks).
    - The client can directly use the descriptor without handling these complexities.
2. **Resource Sharing:**
    - Multiple processes can access the same file or socket with identical state information (e.g., current offset).
3. **Efficient Communication:**
    - File descriptor passing avoids duplicating file state, reducing the overhead of re-opening or configuring the file/device.

Mechanics of Passing File Descriptors

1. **Underlying Mechanism:**
    - File descriptors are passed over UNIX domain sockets using ancillary data (struct msghdr with msg_control).
2. **APIs for Sending and Receiving:**
    - **send_fd**(int fd, int fd_to_send): Sends a file descriptor.
    - **recv_fd**(int fd, ssize_t (*userfunc)(int, const void *, size_t)): Receives a file descriptor.
    - **send_err**(int fd, int status, const char *errmsg): Sends an error status/message alongside the descriptor.
3. **Lifecycle of the Descriptor:**
    - The sender typically closes its local descriptor after passing it.
    - The actual file or device remains open until all references (from all processes) are closed.

Usage Scenarios

1. **Client-Server Models:**
    - Servers open resources on behalf of clients and pass descriptors to clients for direct access.
    - Example: A print server that opens a print job file and passes the descriptor to a client program.

2. **Process Specialization:**
   - A parent process sets up a file and passes descriptors to worker processes, each handling specific tasks.
3. **Modular Applications:**
   - Modular components of a system share file descriptors to maintain a consistent state across modules.

Example Workflow

1. **Server-Side:**
   - Opens or configures a resource.
   - Passes the descriptor to a client using send_fd.
2. **Client-Side:**
   - Receives the descriptor using recv_fd.
   - Performs operations (e.g., read, write, or ioctl) on the resource.
3. **Descriptor Sharing:**
   - Both processes can manipulate the shared file state (e.g., seek positions are updated for both).

Security and Limitations

1. **Security Considerations:**
   - The server must verify the client's identity before passing sensitive file descriptors.
   - File descriptor passing should not expose the server to unintended file operations by untrusted clients.
2. **Limitations:**
   - This technique is limited to systems that support UNIX domain sockets and ancillary data.
   - Resource leakage can occur if passed descriptors are not properly closed by all processes.

To exchange file descriptors using UNIX domain sockets, we call the sendmsg(2) and recvmsg(2) functions . Both functions take a pointer to a msghdr structure that contains all the information on what to send or receive. The structure on your system might look similar to the following:

```
struct msghdr {
    void          *msg_name;         /* optional address */
    socklen_t      msg_namelen;      /* address size in bytes */
    struct iovec  *msg_iov;          /* array of I/O buffers */
    int            msg_iovlen;       /* number of elements in array */
    void          *msg_control;      /* ancillary data */
    socklen_t      msg_controllen;   /* number of ancillary bytes */
    int            msg_flags;        /* flags for received message */
};
```

The first two elements are normally used for sending datagrams on a network connection, where the destination address can be specified with each datagram. The next two elements allow us to specify an array of buffer. The msg_flags field contains flags describing the message received.

Two elements deal with the passing or receiving of control information. The msg_control field points to a **cmsghdr** (control message header) structure, and the **msg_controllen** field contains the number of bytes of control information.

```
struct cmsghdr {
    socklen_t  cmsg_len;    /* data byte count, including header */
    int        cmsg_level;  /* originating protocol */
    int        cmsg_type;   /* protocol-specific type */
    /* followed by the actual control message data */
};
```

To send a file descriptor, we set cmsg_len to the size of the cmsghdr structure, plus the size of an integer (the descriptor). The cmsg_level field is set to SOL_SOCKET, and cmsg_type is set to SCM_RIGHTS, to indicate that we are passing access rights. (SCM stands for socket-level control message.) Access rights can be passed only across a UNIX domain socket. The descriptor is stored right after the cmsg_type field, using the macro CMSG_DATA to obtain the pointer to this integer.

Three macros are used to access the control data, and one macro is used to help calculate the value to be used for cmsg_len.

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

unsigned char *CMSG_DATA(struct cmsghdr *cp);

                         Returns: pointer to data associated with cmsghdr structure

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);

                              Returns: pointer to first cmsghdr structure associated
                                 with the msghdr structure, or NULL if none exists

struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp,
                            struct cmsghdr *cp);

                         Returns: pointer to next cmsghdr structure associated with
                               the msghdr structure given the current cmsghdr
                                       structure, or NULL if we're at the last one

unsigned int CMSG_LEN(unsigned int nbytes);

                                    Returns: size to allocate for data object nbytes large
```

The **CMSG_LEN** macro returns the number of bytes needed to storeadata object of size nbytes, after adding the size of the cmsghdr structure, adjusting for any alignment constraints required by the processor architecture, and rounding up. The send_fd function, which passes a file descriptor over a UNIX domain socket. In the sendmsg call, we send both the protocol data (the null and the status byte) and the descriptor.

To receive a descriptor we allocate enough room for a cmsghdr structure and a descriptor, set msg_control to point to the allocated area, and call recvmsg. We use the **CMSG_LEN** macro to calculate the amount of space needed. We read from the socket until we read the null byte that precedes the final status byte.

Note that we are always prepared to receive a descriptor (we set msg_control and msg_controllen before each call to recvmsg), but only if **msg_controllen** is nonzero on return did we actually receive a descriptor.

OPEN SERVER VERSION-1

An **open server** is designed to provide a flexible and reusable mechanism for managing file access requests from client processes. The key functionality of the server is to open files (or devices/sockets) on behalf of clients and return the file descriptors, rather than the file contents, using **file descriptor passing**.

Key Features

1. **File Descriptor Passing**:
    - The server opens the requested file and passes the file descriptor back to the client.
    - This enables the client to use the file descriptor as if it opened the file itself.
2. **Minimal IPC Communication**:
    - Only essential information is exchanged: the file path and open mode from the client, and the resulting file descriptor or error response from the server.
3. **Flexible File Handling**:
    - Supports all file types (e.g., regular files, devices, sockets) since the server sends a file descriptor, not the file contents.
4. **Independent Server**:
    - The server is a separate executable, allowing it to act as a reusable utility for multiple clients.
    - Being set-user-ID capable, it can provide elevated permissions unavailable to the client.

Application Protocol

1. **Request from Client**:
   - The client sends a null-terminated string in the format:

   open <pathname> <openmode>\0

     - <pathname>: Full file path.
     - <openmode>: Numeric mode (e.g., O_RDONLY, O_WRONLY) in ASCII decimal.
2. **Response from Server**:
   - The server sends:
     - The file descriptor on success using send_fd().
     - An error response using send_err() if the operation fails.

Implementation Steps

Client Process

1. **Setup IPC**:
   - Create a **file descriptor pipe (fd-pipe)** for communication between client and server.
2. **Invoke Server**:
   - Use fork() and exec() to start the server executable.
3. **Send Request**:
   - Write the request string ("open <pathname> <openmode>\0") to the pipe.
4. **Receive Response**:
   - Read the server's response:
     - On success: A valid file descriptor.
     - On failure: An error message.

Server Process

1. **Receive Request**:
   - Read the request string from the fd-pipe.
2. **Parse the Request**:
   - Extract the file path and open mode.

3. **Open the File**:
    - Call open() using the parsed arguments.
4. **Send Response**:
    - If the file is successfully opened:
        - Pass the file descriptor to the client using send_fd().
    - If an error occurs:
        - Use send_err() to notify the client.

Advantages

1. **Reusability**:
    - The server acts as a standalone program that can be used by multiple clients without embedding logic into applications.
2. **Simplified Maintenance**:
    - Updates to the server logic do not require changes to the client applications.
3. **Elevated Permissions**:
    - As a set-user-ID program, the server can perform privileged operations that clients cannot directly execute.
4. **Enhanced Security**:
    - The server can validate requests and enforce additional permission checks beyond standard UNIX file permissions.

CLIENT CODE:

```
#include "apue.h"
int main(int argc, char *argv[])
{
   int fd_pipe[2];
   pid_t pid;
   // Create a pipe for communication
   if (pipe(fd_pipe) < 0)
      err_sys("pipe error");
```

```
    // Fork a child to execute the server
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  // Child process
        close(fd_pipe[1]); // Close write end in child
        if (fd_pipe[0] != STDIN_FILENO) {
            dup2(fd_pipe[0], STDIN_FILENO);
            close(fd_pipe[0]);
        }
        execl("./open_server", "open_server", (char *)0);
        err_sys("execl error");
    }
    // Parent process (client)
    close(fd_pipe[0]); // Close read end in parent
    char request[1024];
    snprintf(request, sizeof(request), "open %s %d\0", "/path/to/file", O_RDONLY);
    write(fd_pipe[1], request, strlen(request) + 1);
    int fd = recv_fd(fd_pipe[1], NULL);
    if (fd < 0)
        err_sys("failed to receive file descriptor");
    printf("Received file descriptor: %d\n", fd);
    close(fd_pipe[1]);
    return 0;
}
```

SEVER CODE:

```
#include "apue.h"
int main(void)
 {
    char buf[1024];
    ssize_t n;
    // Read request from the client
    if ((n = read(STDIN_FILENO, buf, sizeof(buf))) <= 0)
        err_sys("read error");

    char pathname[512];
    int openmode;
    sscanf(buf, "open %s %d", pathname, &openmode);

    // Open the requested file
    int fd = open(pathname, openmode);
    if (fd < 0) {
        send_err(STDOUT_FILENO, errno, "open failed");
        exit(1);
    }
    // Send back the file descriptor
    send_fd(STDOUT_FILENO, fd);
    close(fd);
    return 0;
}
```

OUTPUT:

```
File content: Hello, Open Server!
```