MODULE - 5

Syllabus:

Signals: The UNIX Kernel Support for Signals, signal, Signal Mask, sigaction, The SIGCHLD Signal and the waitpid Function, The sigsetjmp and siglongjmp Functions, Kill, Alarm, Interval Timers, Timers. Daemon Processes: Introduction, Daemon Characteristics, Coding Rules, Error Logging, Client-Server Model. Inter process Communication: Overview of IPC Methods, Pipes, popen, pclose Functions, Coprocesses, FIFOs, System V IPC, Message Queues.

5.1 Signals and Daemon Processes

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGEMT	hardware fault	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGFREEZE	checkpoint freeze	ignore
SIGHUP	Hangup	terminate
SIGILL	illegal instruction	terminate+core
SIGINFO	status request from keyboard	ignore
SIGINT	terminal interrupt character	terminate
SIGIO	asynchronous I/O	terminate/ignore
SIGIOT	hardware fault	terminate+core
SIGKILL	Termination	terminate
SIGLWP	threads library internal use	ignore
SIGPIPE	write to pipe with no readers	terminate
SIGPOLL	pollable event (poll)	terminate
SIGPROF	profiling time alarm (setitimer)	terminate
SIGPWR	power fail/restart	terminate/ignore
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	invalid memory reference	terminate+core
SIGSTKFLT	coprocessor stack fault	terminate
SIGSTOP	stop	stop process

SIGSYS	invalid system call	terminate+core
SIGTERM	Termination	terminate
SIGTHAW	checkpoint thaw	ignore
SIGTRAP	hardware fault	terminate+core
SIGTSTP	terminal stop character	stop process
SIGTTIN	background read from control tty	stop process
SIGTTOU	background write to control tty	stop process
SIGURG	urgent condition (sockets)	ignore
SIGUSR1	user-defined signal	terminate
SIGUSR2	user-defined signal	terminate
SIGVTALRM	virtual time alarm (setitimer)	terminate
SIGWAITING	threads library internal use	ignore
SIGWINCH	terminal window size change	ignore
SIGXCPU	CPU limit exceeded (setrlimit)	terminate+core/ignore
SIGXFSZ	file size limit exceeded (setrlimit)	terminate+core/ignore
SIGXRES	resource control exceeded	Ignore

When a signal is sent to a process, it is pending on the process to handle it. The process can react to pending signals in one of three ways:

- Accept the **default action** of the signal, which for most signals will terminate the process.
- **Ignore** the signal. The signal will be discarded and it has no affect whatsoever on the recipient process.
- Invoke a **user-defined** function. The function is known as a signal handler routine and the signal is said to be ***caught*** when this function is called.

5.1.1 The UNIX Kernel Support for Signals:

- When a signal is generated for a process, the kernel will set the corresponding signal flag in the process table slot of the recipient process.
- If the recipient process is asleep, the kernel will awaken the process by scheduling it.
- When the recipient process runs, the kernel will check the process U-area that contains an array of signal handling specifications.
- If array entry contains a zero value, the process will accept the default action of the signal.
- If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.
- If array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine.

5.1.2 Signal:

The function prototype of the signal API is:

```
#include <signal.h>
void (*signal(int sig_no, void (*handler)(int)))(int);
```

The formal arguments of the API are: sig_no is a signal identifier like SIGINT or SIGTERM. The handler argument is the function pointer of a user-defined signal handler function.

The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, and accepts the default action of the SIGSEGV signal. The pause API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return:

```
#include<iostream.h>
#include<signal.h>

/*signal handler function*/
void catch_sig(int sig_num)
{
        signal (sig_num,catch_sig);
        cout<<"catch_sig:"<<sig_num<<endl;
}

/*main function*/

int main()
{
        signal(SIGTERM,catch_sig);
        signal(SIGINT,SIG_IGN);
        signal(SIGSEGV,SIG_DFL);
        pause( );                       /*wait for a signal interruption*/
}
```

The SIG_IGN specifies a signal is to be ignored, which means that if the signal is generated to the process, it will be discarded without any interruption of the process.

The SIG_DFL specifies to accept the default action of a signal.

5.1.3 Signal Mask:

A process initially inherits the parent's signal mask when it is created, but any pending signals for the parent process are not passed on. A process may query or set its signal mask via the sigprocmask API:

#include <signal.h>

int sigprocmask(int cmd, const sigset_t *new_mask, sigset_t *old_mask);

Returns: 0 if OK, 1 on error

The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, and the cmd argument specifies how the new_mask value is to be used by the API.

The possible values of cmd and the corresponding use of the new_mask value are:

cmd value	Meaning
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the new_mask argument
SIG_BLOCK	Adds the signals specified in the new_mask argument to the calling process signal mask
SIG_UNBLOCK	Removes the signals specified in the new_mask argument from the calling process signal mask

- If the actual argument to new_mask argument is a NULL pointer, the cmd argument will be ignored and the current process signal mask will not be altered.
- If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.
- The sigset_t contains a collection of bit flags.

The BSD UNIX and POSIX.1 define a set of API known as sigsetops functions:

```
#include<signal.h>

int sigemptyset (sigset_t* sigmask);
int sigaddset (sigset_t* sigmask, const int sig_num);
int sigdelset (sigset_t* sigmask, const int sig_num);
int sigfillset (sigset_t* sigmask);
int sigismember (const sigset_t* sigmask, const int sig_num);
```

- The sigemptyset API clears all signal flags in the sigmask argument.
- The sigaddset API sets the flag corresponding to the signal_num signal in the sigmask argument.
- The sigdelset API clears the flag corresponding to the signal_num signal in the sigmask argument.
- The sigfillset API sets all the signal flags in the sigmask argument. [all the above functions return 0 if OK, -1 on error]
- The sigismember API returns 1 if flag is set, 0 if not set and -1 if the call fails.
- The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask if it is not there.

```
#include<stdio.h>
#include<signal.h>
int main()
{
 sigset_t        sigmask;
 sigemptyset(&sigmask);              /*initialise set*/

if(sigprocmask(0,0,&sigmask)==-1)        /*get current signal mask*/
{
perror("sigprocmask");
exit(1);
}
else sigaddset(&sigmask,SIGINT);  /*set SIGINT flag*/
    sigdelset(&sigmask, SIGSEGV);          /*clear SIGSEGV flag*/
if(sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
    perror("sigprocmask");
}
```

A process can query which signals are pending for it via the sigpending API:

#include<signal.h>
int *sigpending*(sigset_t* sigmask);
Returns 0 if OK, -1 if fails.

The sigpending API can be useful to find out whether one or more signals are pending for a process and to set up special signal handling methods for these signals before the process calls the sigprocmask API to unblock them.

5.1.4 sigaction:

The sigaction API blocks the signal it is catching allowing a process to specify additional signals to be blocked when the API is handling a signal.

The sigaction API prototype is:

#include<signal.h>

int sigaction(int signal_num, struct sigaction* action, struct sigaction* old_action);

Returns: 0 if OK, 1 on error

The struct sigaction data type is defined in the <signal.h> header as:

```
struct sigaction
{
        void    (*sa_handler)(int);
        sigset_t        sa_mask;
        int     sa_flag;
}
```

The following program illustrates the uses of sigaction:

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void callme(int sig_num)
{
        cout<<"catch signal:"<<sig_num<<endl;
}

int main(int argc, char* argv[])
{
        sigset_t sigmask;
        struct sigaction action,old_action;

        sigemptyset(&sigmask);

        if(sigaddset(&sigmask,SIGTERM)==-1                                    ||
        sigprocmask(SIG_SETMASK,&sigmask,0)==-1)

                perror("set signal mask");
        sigemptyset(&action.sa_mask);
        sigaddset(&action.sa_mask,SIGSEGV);
        action.sa_handler=callme;
        action.sa_flags=0;
        if(sigaction(SIGINT,&action,&old_action)==-1)

                perror("sigaction");
                pause();
                cout<<argv[0]<<"exists\n";
                return 0;
}
```

5.1.5 The SIGCHLD Signal and waitpid Function:

When a child process terminates or stops, the kernel will generate a SIGCHLD signal to its parent process. Depending on how the parent sets up the handling of the SIGCHLD signal, different events may occur:

- Parent accepts the **default action** of the SIGCHLD signal:
  - SIGCHLD does not terminate the parent process
  - Parent process will be awakened
  - API will return the child's exit status and process ID to the parent
  - Kernel will clear up the Process Table slot allocated for the child process
  - Parent process can call the waitpid API repeatedly to wait for each child it created

- Parent ignores the SIGCHLD signal:
  - SIGCHLD signal will be discarded
  - Parent will not be disturbed even if it is executing the waitpid system call
  - If the parent calls the waitpid API, the API will suspend the parent until all its child processes have terminated
  - Child process table slots will be cleared up by the kernel
  - API will return a -1 value to the parent process

- Process **catches** the SIGCHLD signal:
  - The signal handler function will be called in the parent process whenever a child process terminates
  - If the SIGCHLD arrives while the parent process is executing the waitpid system call, the waitpid API may be restarted to collect the child exit status and clear its process table slots
  - Depending on parent setup, the API may be aborted and child process table slot not freed

5.1.6 The sigsetjmp and siglongjmp Functions:

The function prototypes of the APIs are:

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
int siglongjmp(sigjmp_buf env, int val);
```

The sigsetjmp and siglongjmp are created to support signal mask processing. Specifically, it is implementation- dependent on whether a process signal mask is saved and restored when it invokes the setjmp and longjmp APIs respectively.

The only difference between these functions and the setjmp and longjmp functions is that sigsetjmp has an additional argument. If savemask is nonzero, then sigsetjmp also saves the current signal mask of the process in env. When siglongjmp is called, if the env argument was saved by a call to sigsetjmp with a nonzero savemask, then siglongjmp restores the saved signal mask. The siglongjmp API is usually called from user-defined signal handling functions. This is because a process signal mask is modified when a signal handler is called, and siglongjmp should be called to ensure the process signal mask is restored properly when "jumping out" from a signal handling function.

5.1.7 Kill:

A process can send a signal to a related process via the kill API. This is a simple means of inter-process communication or control. The function prototype of the API is:

#include<signal.h>
int kill(pid_t pid, int signal_num);
Returns: 0 on success, -1 on failure.

The signal_num argument is the integer value of a signal to be sent to one or more processes designated by pid. The possible values of pid and its use by the kill API are:

pid > 0	The signal is sent to the process whose process ID is pid
pid==0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal
pid==1	The signal is sent to all processes on the system for which the sender has permission to send the signal

The following program illustrates the implementation of the UNIX kill command using the kill API:

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<signal.h>

int main(int argc,char** argv)
{
int pid, sig = SIGTERM;
if(argc==3)
{
if(sscanf(argv[1],"%d",&sig)!=1)
{
cerr<<"invalid number:" << argv[1] << endl;
return -1;
}
```

```
argv++,argc--;
}
while(--argc>0)
if(sscanf(*++argv, "%d", &pid)==1)
{
}
else
if(kill(pid,sig)==-1)
    perror("kill");
cerr<<"invalid pid:" << argv[0] <<endl;
return 0;
}
```

The UNIX kill command invocation syntax is:

Kill [-<signal_num>] <pid>……

Where signal_num can be an integer number or the symbolic name of a signal. <pid> is process ID.

5.1.8 Alarm:

The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype of the API is:

```
#include<signal.h>
unsigned int alarm(unsigned int time_interval);
```

5.1.9 Interval Timers:

The interval timer can be used to schedule a process to do some tasks at a fixed time interval, to time the execution of some operations, or to limit the time allowed for the execution of some tasks.

The following program illustrates how to set up a real-time clock interval timer using the alarm API:

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

#define INTERVAL 5

void callme(int sig_no)
{
        alarm(INTERVAL);
         /*do scheduled tasks*/
}

int main()
{
        struct sigaction action; sigemptyset(&action.sa_mask);
        action.sa_handler=(void(*)( )) callme;
        action.sa_flags=SA_RESTART;
        if(sigaction(SIGALARM,&action,0)==-1)
        {
                perror("sigaction");
                return 1;
        }
        if(alarm(INTERVAL)==-1)
                 perror("alarm");
        else while(1)
        {
                /*do normal operation*/
        }
        return 0;
}
```

In addition to alarm API, UNIX also invented the setitimer API, which can be used to define up to three different types of timers in a process:

- Real time clock timer
- Timer based on the user time spent by a process
- Timer based on the total user and system times spent by a process

The getitimer API is also defined for users to query the timer values that are set by the setitimer API. The setitimer and getitimer function prototypes are:

The struct itimerval datatype is defined as:

#include<sys/time.h>

int setitimer(int which, const struct itimerval * val, struct itimerval * old);

int getitimer(int which, struct itimerval * old);

The *which* arguments to the above APIs specify which timer to process. Its possible values and the corresponding timer types are:

ITIMER_REAL	Decrements in real time and generates a SIGALARM signal when it expires
ITIMER_VIRTUAL	Decrements in virtual time (time used by the process) and generates a SIGVTALRM signal when it expires
ITIMER_PROF	Decrements in virtual time and system time for the process and generates a SIGPROF signal when it expires

5.2 Daemon Processes:

5.2.1 Introduction:

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down.

5.2.1.1 Daemon Characteristics:

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session and group leaders.

5.2.1.2 Coding Rules:

- **Call umask to set the file mode creation mask to 0**. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions.
- **Call fork and have the parent exit**. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader.
- **Call setsid to create a new session**. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
- **Change the current working directory to the root directory**. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.
- **Unneeded file descriptors should be closed.** This prevents the daemon from holding open any descriptors that it may have inherited from its parent.
- **Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect**. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed;

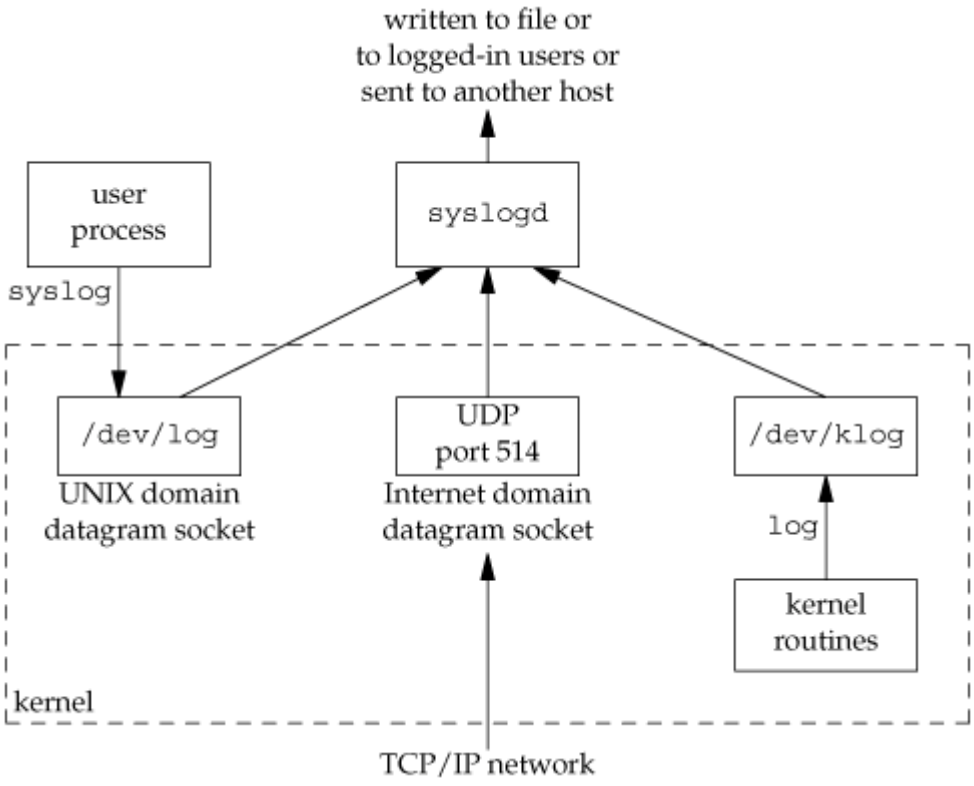
nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

5.2.2 Daemon Conventions:

- If the daemon uses a lock file, the file is usually stored in /var/run. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually name.pid, where name is the name of the daemon or the service. For example, the name of the cron daemon's lock file is /var/run/crond.pid
- If the daemon supports configuration options, they are usually stored in /etc. The configuration file is named name.conf, where name is the name of the daemon or the name of the service. For example, the configuration for the syslogddaemon is /etc/syslog.conf
- Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (/etc/rc* or /etc/init.d/*). If the daemon should be restarted automatically when it exits, we can arrange for initto restart it if we include a respawnentry for it in /etc/inittab
- If a daemon has a configuration file, the daemon reads it when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch SIGHUP and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive SIGHUP. Thus, they can safely reuse it.

5.2.3 Error Logging

- One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system. We also don't want each daemon writing its own error messages into a separate file. It would be a headache for anyone administering the system to keep up with which daemon writes to which log file and to check these files on a regular basis. A central daemon error-logging facility is required.
- The BSD syslog facility was developed at Berkeley and used widely in 4.2BSD. Most systems derived from BSD support syslog. Until SVR4, System V never had a central daemon logging facility.
- The syslog function is included as an XSI extension in the Single UNIX Specification. The BSD syslog facility has been widely used since 4.2BSD. Most daemons use this facility. The following figure illustrates its structure.



The BSD syslog facility

- Three ways to generate log messages:
    1. Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device. We won't describe this function any further, since we're not interested in writing kernel routines.
    2. Most user processes (daemons) call the syslog(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
    3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

5.2.4 Client Server Model:

- A common use for a daemon process is as a server process. Indeed, in the above figure (The BSD syslog facility), we can call the syslogd process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.
- In general, a server is a process that waits for a client to contact it, requesting some type of service.
- In the above figure (The BSD syslog facility), the service being provided by the syslogd server is the logging of an error message. Also, the communication between the client and the server is one-way. The client sends its service request to the server; the server sends nothing back to the client.

5.3 Overview of IPC Methods:

IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems.

The various forms of IPC that are supported on a UNIX system are as follows:

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

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

5.3.1 Pipes:

Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.
Historically, they have been half duplex (i.e., data flows in only one direction).
Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

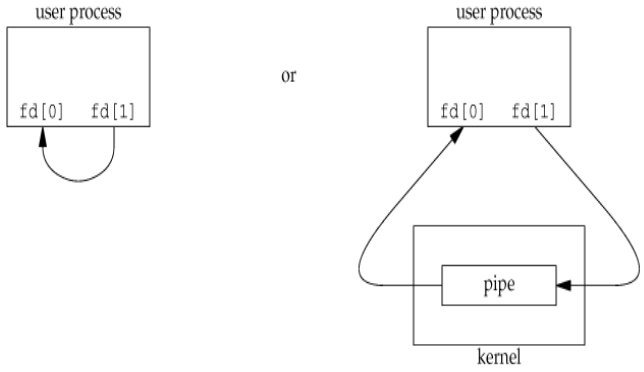
A pipe is created by calling the pipe function.

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

Returns: 0 if OK, 1 on error.

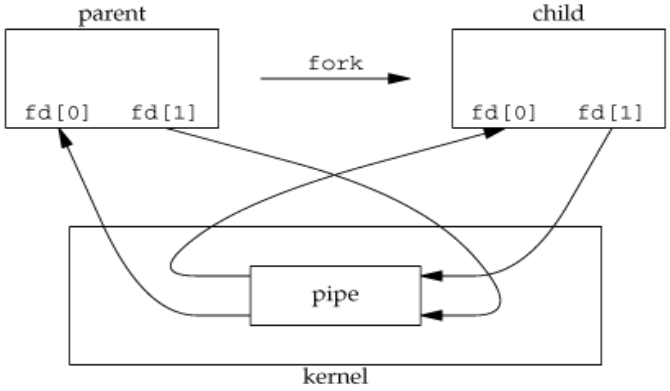
Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

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



Two ways to view a half-duplex pipe

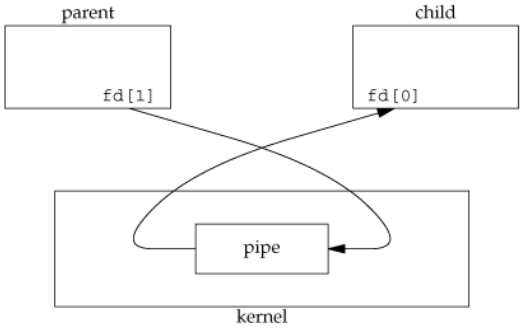
A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa. The foll Figure shows this scenario



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]).

The foll figure shows the resulting arrangement of descriptors



Pipe from parent to child

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

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

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

```
#include "apue.h"
int main(void)
{
    int         n;
    int         fd[2];
    pid_t       pid;
    char        line[MAXLINE];

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

```
else
{                       /* child */
      close(fd[1]);
      n = read(fd[0], line, MAXLINE);
      write(STDOUT_FILENO, line, n);
}
exit(0);
}
```

5.3.1.1 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, NULLon error

int pclose(FILE *fp);
Returns: termination status of cmdstring or 1 on error

The function popendoes a forkand execto 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: Result of fp = popen(cmdstring, "r")

If type is “w”, the file pointer is connected to the std input of cmdst as shown:

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

5.3.1.2 Coprocesses:

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.

The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess. Figure 15.16 shows this arrangement.

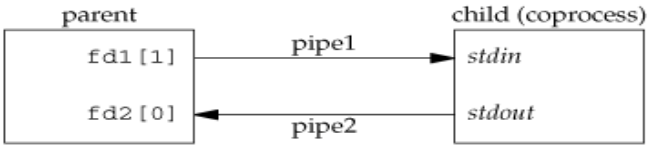


Figure: Driving a coprocess by writing its standard input and reading its standard output

Program: Simple filter to add two numbers

```
#include "apue.h"

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

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;            /* null terminate */

        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

5.3.2 FIFOs:

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

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```
Returns: 0 if OK, 1 on error

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

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

Uses of FIFOs:

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

5.3.2.1 Example Client-Server Communication Using a FIFO

- FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.
- A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client'sprocess ID.
- For example, the server can create a FIFO with the name /vtu/ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the 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.

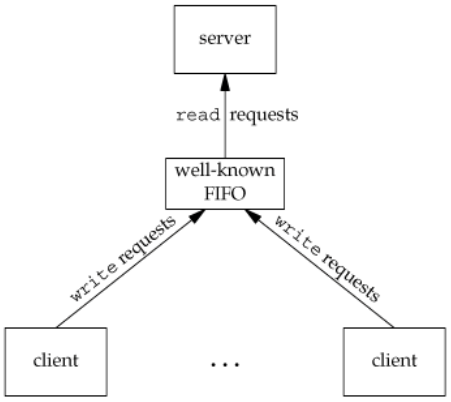


Figure: Clients sending requests to a server using a FIFO

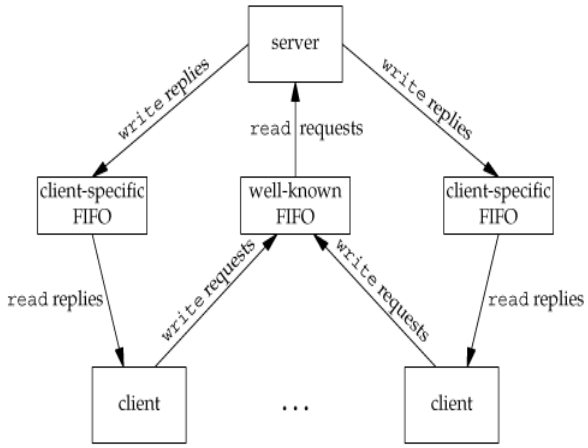


Figure: Client-server communication using FIFOs

5.3.3 Identifiers and Keys

Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer identifier. The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a key that acts as an external name.

Whenever an IPC structure is being created, a key must be specified. The data type of this key is the primitive system data type key_t, which is often defined as a long integer in the header <sys/types.h>. This key is converted into an identifier by the kernel.

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

- The server can create a new IPC structure by specifying a key of IPC_PRIVATE and store the returned identifier somewhere (such as a file) for the client to obtain. The key IPC_PRIVATE guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.
- The IPC_PRIVATE key is also used in a parent-child relationship. The parent creates a new IPC structure specifying IPC_PRIVATE, and the resulting identifier is then available to the child after the fork. The child can pass the

identifier to a new program as an argument to one of the exec functions.

- 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.
- 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 ftokis 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

The path argument must refer to an existing file. Only the lower 8 bits of id are used when generating the key.

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. If two pathnames refer to two different files, then ftok usually returns two different keys for the two pathnames. 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.

5.3.4 Permission Structure

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

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

All the fields are initialized when the IPC structure is created. At a later time, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling chown or **chmod**for a file.

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

IPC permissions

5.3.5 Configuration Limits

All three forms of XSI IPC have built-in limits that we may encounter. Most of these limits can be changed by reconfiguring the kernel. We describe the limits when we describe each of the three forms of IPC.

5.3.6 Advantages and Disadvantages

- A fundamental problem with XSI IPC is that the IPC structures are systemwide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling msgrcv or msgctl, by someone executing the ipcrm(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.
- Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions. Almost a dozen new system calls (msgget, semop, shmat, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an ls command, we can't remove them with the rm command, and we can't change their permissions with the chmod command. Instead, two new commands ipcs(1) and ipcrm(1)were added.
- Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (select and poll) with them. 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. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busywait loop.

5.3.7 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 msgid_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 msgnd() */
Pid_t msg_lrpid; /*d of last msgrcv()*/

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

This structure defines the current status of the queue.

The first function normally called is msggetto 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_rtimeare all set to 0.
- msg_ctimeis set to the current time.
- msg_qbytesis set to the system limit.

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

The msgctlfunction performs various operations on a queue.

#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
Returns: 0 if OK, 1 on error.

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

Data is placed onto a message queue by calling msgsnd.

#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
Returns: 0 if OK, 1 on error

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

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

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

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

Messages are retrieved from a queue by msgrcv.

#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);

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

The type argument lets us specify which message we want.

type==0	First message on the queue is returned
type>0	First message on the queue whose message type equals type is returned
type<0	First message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned