8.2.6.2 UNIX Command Pipes

The UNIX shell command pipes execute multiple commands concurrently, such that the standard output of one command (specified to the left of a "|" symbol) is connected directly to the standard input of the next command (specified to the right of a "|" symbol). For example, given the following command:

```
%    ls   -l   |   sort  -r
```

the UNIX shell forks two child processes. One executes the *ls -l* command and the other executes the *sort* command. Furthermore, the standard output data of the child process executing the *ls -l* command will be directed to the standard input port of the one executing the *sort* command. The output of these two commands is a sorted list of the current directory content.

The following *command_pipe.C* program illustrates how to execute two commands concurrently with a command pipe:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define CLOSE_ALL()          close(fifo[0]), close(fifo[1])

int main( int argc, char* argv[])
{
     int fifo[2];
     pid_tpid1, pid2;

     if(pipe(fifo)) perror("pipe"), _exit(2); /* create a command pipe */

     switch (pid1=fork())  {      /* execute command to the left of pipe */
        case -1:      perror("fork"), exit(3);
        case 0:       if (dup2(fifo[1],STDOUT_FILENO)==-1)
                          perror("dup2"), exit(4);
                      CLOSE_ALL();
                      if (execlp("/bin/sh","sh","-c",argv[1],0)==-1)
                          perror("execl"), exit(5);
     }

     switch (pid2=fork())  {      /* execute command to the right of pipe */
        case -1:      perror("fork"), exit(6);
        case 0:       if (dup2(fifo[0],STDIN_FILENO)==-1)
                          perror("dup2"), exit(7);
                      CLOSE_ALL();
```

```
                    if (execlp("/bin/sh","sh","-c",argv[2],0)==-1)
                       perror("execl"), exit(8);
        }

        CLOSE_ALL();
        if (waitpid(pid1,&fifo[0],0)!=pid1 || waitpid(pid2,&fifo[1],0)!=pid2)
            perror("waitpid");
        return fifo[0]+fifo[1];
}
```

This program takes two command line arguments. Each argument specifies a shell command to be executed. If a shell command includes arguments, the command and its arguments must be enclosed by a pair of quotation marks so that the UNIX shell will pass them as one argument to the program. For example, if the following shell command is specified, the *argv[1]* of the program will be *ls -l*, and the *argv[2]* of the program will be *sort -r*.

```
%    a.out    "ls -l"    "sort -r"
```

When the program is run, it creates an unnamed pipe to connect the standard input and output ports of the two commands to be executed. If the *pipe* system call fails, the program calls *perror* to print a diagnostic message and then exits.

After a pipe is created, the program forks a child process to *exec* the shell command specified in *argv[1]*. However, before the *execl* call, the child process redirects its standard output port to the pipe's write end, then closes its copy of the pipe file descriptor. Thus, when the *argv[1]* shell command is executed, the standard output data will be directed into the pipe.

The program also forks another child process to *exec* a Bourne shell that executes the shell command specified in *argv[2]*. However, before the *execl* call the child process redirects its standard input port to the pipe's read end and closes its copy of the pipe file descriptor. Thus, when the Bourne shell executes *argv[2]*, the standard input data will be from the pipe.

After creating two child processes, the parent process closes the file descriptors of the pipe. This is to ensure that when the first child process (which executes *argv[1]*) terminate, there will be no file descriptor referencing the pipe's read end, and the second child process (which executes *argv[2]*) will eventually read the end-of-file indicator from the pipe and know when to terminate itself. The parent process waits for the two child processes to terminate and checks their exit status.

The above program is not POSIX.1-compliant, as the Bourne shell is not defined in POSIX.1. However, it is compliant to POSIX.2, where shells and command pipes are defined

The sample output of the program may be:

```
%    CC -o command pipe command_pipe.C
%    command_pipe    "ls -l"   wc
52   410    3034
%    command_pipe    pwd    cat
/home/book/chapter10
%    command_pipe    cat    "sort -r"
Hello world
Bye-Bye
^D
Bye-Bye
Hello world
%
```

The above program can be further enhanced to accept an arbitrary number of shell commands and pipes. Thus, if the new program is run with the following command:

```
%    command_pipe    "ls -l"    "sort -r"    "wc"    "cat"
```

it is similar to the following shell command:

```
%    ls -l    | sort -r    |    wc  |    cat
```

As a general rule, if a UNIX command contains *N* "|" symbols, the shell must create *N* pipes (created via the *pipe* API) and *N+1* child processes. Because each pipe call consumes two file descriptors and because a process may use, at most, OPEN_MAX file descriptors at any one time, a shell process must recycle its file descriptors. This enables the setting up of unnamed pipes to handle an unlimited number of command pipes.

The new program shows how to handle arbitrary numbers of command pipes and shell commands:

```
/* command_pipe2.C */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#define CLOSE(fd)  if (fd < -1) close(fd), fd=-1
static int fifo[2][2] = { -1, -1, -1, -1 }, cur_pipe=0;
```

```
int main( int argc, char* argv[])
{
    for (int i=1; i < argc; i++)    {
      if (pipe(fifo[cur_pipe])) perror("pipe"), _exit(2);
      switch (fork())  {          /* execute command to the left of pipe */
        case -1:      perror("fork"), exit(3);
        case 0:       if (i > 1)   {              /* not first command */
                        dup2(fifo[1-cur_pipe][0],STDIN_FILENO);
                        CLOSE(fifo[1-cur_pipe][0]);
                      }
                      if (i < argc-1)    {    /* not the last command */
                        dup2(fifo[cur_pipe][1],STDOUT_FILENO);
                        CLOSE(fifo[cur_pipe][0]);
                        CLOSE(fifo[cur_pipe][1]);
                      }
                      if (execlp("/bin/sh","sh","-c",argv[i],0)==-1)
                        perror("execl"), exit(5);
      }
      CLOSE(fifo[1-cur_pipe][0]);
      CLOSE(fifo[cur_pipe][1]);
      cur_pipe = 1 - cur_pipe;
    }
    CLOSE(fifo[1-cur_pipe][0]);
    while (waitpid(-1,0,0))
      ;
    return 0;
}
```

The above program creates an unique pipe between every two consecutive commands. All commands except the very first one will get the standard input data from their pipe's read end (the first command standard input is from the console). Similarly, all commands except the very last one will send their standard output data to the write end of the next command pipe (the last one will send its standard output to the console). The only difficult part here is that the parent process must close all the unnecessary pipe descriptors after each child is created. This is to ensure that it does not use up all the allowable file descriptors and that no unnecessary file descriptors are used up in the new child process. Furthermore, the parent process must make sure only one child process has a file descriptor referencing one end of a pipe at any time. The end-of-file indicator will eventually be passed from the first command process to the next, and so on, until all the child processes receive it and terminate properly.

The sample output of the program may be:

```
%   CC command_pipe2.C
%   a.out   ls   sort   wc
150 50 724
```

```
%    a.out    date
Sun Jan 16 13:0:0 PST 1994
%    a.out    date    wc
1  6  29
%    a.out pwd sort cat
/home/book/chapter11
```

8.2.6.3 The popen and pclose Functions

This section depicts more advanced examples on the usage of *fork*, *exec*, and *pipe*. Specifically, it will show how to use these APIs to implement the C library functions *popen* and *pclose*.

The *popen* function is used to execute a shell command within a user program. The function prototype of the *popen* function is:

```
FILE*    popen (const char* shell_cmd, const char* mode);
```

The first argument *shell_cmd* is a character string that contains any shell command a user may execute in a Bourne shell. The function will invoke a Bourne shell to interpret and execute that command.

The second argument *mode* is either "*r*" or "*w*." It specifies that the stream pointer that is returned from the function can be used to either read data from the standard output (if *mode* is "*r*") or to write data to the standard input (if *mode* is "*w*") to the Bourne shell process that executes the *shell_cmd*.

popen returns a NULL value if the *shell_cmd* cannot be executed. Possible causes of failure may be that the *shell_cmd* is invalid or that the process lacks permission to execute the command.

The *pclose* function accepts a stream pointer that is returned from a *popen* function call. It will close the stream associated with the stream pointer and then wait for the corresponding Bourne shell process to terminate. The function prototype of the *pclose* function is:

```
int        pclose (FILE *fstream):
```

pclose returns the exist status of the command being executed if it succeeds or a -1 if it

fails. Possible cause of failure may be that the *fstream* actual value is invalid or not defined by a *popen* call.

The *popen* function is implemented by calling *fork* to create a child process, which in turn will *exec* a Bourne shell (*/bin/sh*) to interpret and execute the *shell_cmd*. Additionally, the parent process will call *pipe* to establish a connection between either the pipe read end and the child standard output (if *mode* is "*r*"), or between the pipe write end and the child standard input (if *mode* is "*w*"). The file descriptor of the pipe other end is converted to a stream pointer, via the *fdopen* function, and is returned to the caller of the *popen* function.

The *popen* function may be implemented as the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <limits.h>

struct sh_rec
{
      pid_t   sh_pid;
      FILE*   stream;
}               sh_info[OPEN_MAX];
static int      num_sh;

FILE* popen(const char* shell_cmd, const char* mode)
{
      int fifo[2];
      if ((strcmp(mode,"r") && strcmp(mode,"w")) || pipe(fifo)==-1)
         return 0;

      switch (sh_info[num_sh].sh_pid=fork())      {
         case -1:      perror("fork"); return 0;
         case 0:       (*mode=='r') ?  dup2(fifo[1],STDOUT_FILENO) :
                                       dup2(fifo[0],STDIN_FILENO);
                       close(fifo[0]);
                       close(fifo[1]);
                       execl("/bin/sh", "sh", "-c", shell_cmd, 0);
                       exit(5);
      }
      if (*mode=='r')  {
         close(fifo[1]);
         return (sh_info[num_sh++].stream=fdopen(fifo[0],mode));
```

```
        } else  {
            close(fifo[0]);
            return (sh_info[num_sh++].stream=fdopen(fifo[1],mode));
        }
}
```

Note that each child process PID and its corresponding stream pointer from each *popen* call are recorded in a global array *sh_info*. The *sh_info* array has OPEN_MAX-1 entries, as a process can, at most, call *popen* OPEN_MAX-1 times to allocate stream pointers referencing *exec*'ed shell commands. The data stored in the *sh_info* array is used by the *pclose* function in the following manner: When *pclose* is called, it finds an entry in the *sh_info* array that *stream* value matches with the *fstream* argument value. If there is no match, the *fstream* value is invalid, and the *pclose* function will return a -1 failure value. If there is a match, the *pclose* function will close the stream (a pipe end) referenced by *fstream*, then wait for the corresponding child process (whose PID is given by the *sh_pid* variable in the *sh_info* entry) to terminate before the function returns a zero success value.

The *pclose* function may be implemented as follows:

```
int pclose (FILE *fstream )
{
    int i, status, rc=-1;
    for (i=0; i < num_sh; i++)
        if (sh_info[i].stream==fstream) break;
    if (i == num_sh) return -1;         /* invalid fstream value */
    fclose(fstream);
    if (waitpid(sh_info[i].sh_pid,&status,0)==sh_info[i].sh_pid ||
                                    WIFEXITED(status))
        rc = WEXITSTATUS(status);
    for (i++; i < num_sh; i++) sh_info[i-1] = sh_info[i];
    num_sh--;
    return rc;
}
```

The *popen* and *pclose* functions are defined in POSIX.2, but not in POSIX.1. The following *test_popen.C* program illustrates the use of the *popen* and *pclose* functions:

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

int main (int argc, char* argv[])
{
    charbuf[256], *mode= (argc>2) ? "w" : "r";
```

```
        FILE*fptr;

        if (argc>1 && (fptr = popen(argv[1],mode)))    {
           switch (*mode) {
                case 'r':   while (fgets(buf,256,fptr)) fputs(buf,stdout);
                            break;
                case 'w':   fprintf(fptr,"%s\n",argv[2]);
           }
           return pclose(fptr);
        }
        return 5;
}
```

The test program invocation syntax is:

```
%    a.out <shell_cmd> [ <data to write to shell_cmd> ]
```

If the test program is invoked with a shell command only (that is, one argument), the program will call *popen* to execute the command and print the *exec*'ed command's standard output data to the console. If, however, the program is invoked with a shell command and second argument, the program will execute the shell command, via *popen*, and supply the second argument as data to the standard input port of the *exec*'ed command.

The program calls *pclose* to terminate the *popen* call. The sample output of the test program may be:

```
%    CC test_popen.C
%    a.out date
Sat Jan 15 21:42:22 PST 1994
%    a.out   "Hello world"   wc
1    2       11
```

8.3 Process Attributes

A discussion of process will not be complete without mentioning the various APIs to query and set some of the process attributes. This section will depict the process attribute inquiry APIs that are common for both POSIX.1 and UNIX. The next section will depict the POSIX.1 and UNIX APIs that change process attributes.

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

pid_t    getpid    (void);
pid_t    getppid   (void);
pid_t    getpgrp   (void);
uid_t    getuid    (void);
uid_t    geteuid   (void);
gid_t    getgid    (void);
gid_t    getegid   (void);
```

Note that in older versions of UNIX all the above APIs return values that are of type *int*. The *pid_t, uid_t,* and *gid_t* types are defined in the <sys/types.h> header.

The *getpid* and *getppid* APIs return the calling process PID and its parent process ID, respectively. No arguments are needed for these system calls.

The *getpgrp* API returns the calling process's process group ID. Every process in a UNIX or POSIX system belongs to a process group. When a user logs onto a system, the login process becomes a session leader and a process group leader. The session ID and the process group ID of a session leader is the same as its process ID. If the login process creates new child processes to execute jobs, these child processes will be in the same session and process group as the login process. However, if the login process moves some jobs to the background, the process associated with each background job will be assigned a different process group ID. Furthermore, if a background job is executed by more than one process, then the process that created all the other processes for the job will become the process group leader of the job.

The *getuid* and *getgid* system calls return the real user ID and real group ID of the calling process, respectively. The real user-ID and group-ID are the UID and GID of a person who created the process. For example, when you login to a UNIX system, the login shell's real UID and real GID are your UID and GID, respectively. All the child processes created by this login shell will also have your real UID and GID. The real UID and real GID are used by the UNIX kernel to keep track of which user created a process in the system.

The *geteuid* and *getegid* system calls return the effective user ID and effective group ID of the calling process. The effective user ID and group ID are used by the kernel to determine the access permission of the calling process in accessing files. These attributes are also assigned to the UID and GID attributes of files created by the process. In normal situations, a process effective UID is the same as its real UID. However, if the *set-UID* flag of the executable file is set, the process effective user ID will take on the executable file UID. This gives the process the access privileges of the user who owns the executable file. A similar mecha-

nism applies to the effective GID, where a process effective GID is different from its real GID if the corresponding executable file *set-GID* flag is set.

An example of the *set-UID* flag use is the */bin/passwd* command. This command aids users changing their passwords in the */etc/passwd* file. Since the */etc/passwd* file is read-only for all users, the process created by executing the */bin/passwd* program must have superuser privileges to be able to write data to the */etc/passwd* file. Thus, the */bin/passwd* file's UID is the superuser user ID, and its *set-UID* flag is ON. A process created by executing the program */bin/passwd* will have the effective UID of the superuser, and it has the privilege to modify the */etc/passwd* file in the course of its run.

The *set-UID* and *set-GID* flags of an executable file may be changed via either the *chmod* UNIX command or the *chmod* API.

The following *getproc.C* program illustrates how to obtain process attributes:

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

int main ( )
{
    cout << "Process PID: " << (int)getpid() << endl;
    cout << "Process PPID: " << (int)getppid() << endl;
    cout << "Process PGID: " << (int)getpgrp() << endl;
    cout << "Process real-UID: " << (int)getuid() << endl;
    cout << "Process real-GID: " << (int)getgid() << endl;
    cout << "Process effective-UID: " << (int)geteuid() << endl;
    cout << "Process effective-GID: " << (int)getegid() << endl;
}
```

The output of the program may be:

```
%    CC getproc.C
%    a.out
Process PID: 1254
Process PPID: 200
Process PGID: 50
Process real-UID:751
Process real-GID: 77
Process effective-UID: 205
Process effective-GID: 77
%
```

8.4 Change Process Attributes

This section depicts the POSIX.1 and UNIX APIs that change process attributes. Note that some of the process attributes, such as PID and parent PID, can be queried but not changed, whereas some attributes, such as a process session ID, cannot be queried but can be changed.

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

int     setsid    (void);
int     setpgid   (pid_t pid, pid_t pgid);
int     setuid    (uid_t uid);
int     seteuid   (uid_t uid);
int     setgid    (gid_t gid);
int     setegid   (gid_t gid);
```

The *setsid* API sets the calling process to be a new session leader and a new process group leader. The session ID and the process group ID of the process are the same as its PID. The process will also be disassociated with its controlling terminal. This API is commonly called by a daemon process after it is created, so that it can run independently from its parent process. This is a POSIX.1-specific API.

The *setpgid* API sets the calling process to be a new process group leader. The process group ID of the process is the same as its PID. The process will be the only member in the new group. This call will fail if the calling process is already the session leader of a process session. In UNIX System V, the *setpgrp* API is the same as the *setpgid* API.

If the effective UID of a process is a superuser, the *setuid* system call will set the real-UID, effective-UID, and the saved set-UID attributes of the process to the *uid* argument value. If, however, the calling process does not have the effective UID of a superuser, then if *uid* is either the real-UID or saved set-UID value, the API will change the effective UID of the process to *uid*. Otherwise, the API will return a -1 failure status.

If the effective GID of a process is a superuser, the *setgid* system call will set the real-GID, effective-GUID, and the saved set-GID attributes of the process to the *gid* argument value. If, however, the calling process does not have the effective UID of a superuser and *gid* is either the real-GID or saved set-GUID value, the API will change the effective GID of the calling process to *gid*. Otherwise, the API will return a -1 failure status.

The *seteuid* API changes the calling process effective UID to *uid*. If the process has no superuser privilege, the *uid* value must be either the real UID or the saved set-UID of the process. If, on the other hand, the process has superuser privileges, the *uid* may be any value.

The *setegid* API changes the calling process effective GID to *gid*. If the process has no superuser privilege the *gid* must be either the real GID or the saved set-GID of the process. If, on the other hand, the process has superuser privileges, the *gid* may be any value.

8.5 A Minishell Example

To illustrate the use of process APIs, this section depicts a simple UNIX shell program that can create processes which execute any number of UNIX commands, either serially or concurrently. Specifically, the minishell program can execute any user-specified commands with optional input and/or output redirection and can also execute commands in the background and/or foreground. The only limitation of this minishell program is that it does not support any shell variables or metacharacters.

The design of the minishell program is as follows: A simple UNIX command consists of a command name, optional switches, and any number of arguments. Furthermore, input and output redirections may be specified with any command, command pipes may be specified to chain multiple simple commands together, and "&" may be specified to execute a command in the background. The following are examples of UNIX commands that are acceptable by the minishell program:

```
%   pwd > foo                     # a simple command with output redirection
%   sort-r /etc/passwd &          # a command executed in background
%   cat -n < abc > foo            # a command with input/output redirection
%   ls -l | sort-r | cat -n       # two command pipes with three commands
%   cat foo; date; pwd > zz       # three commands executed in sequence
%   (spell /etc/motd | sort ) > xx; date   # execute two cmds in a sub-shell
```

To capture these commands to be executed, a CMD_INFO class is defined, where each of its objects store simple command information. The definition of the CMD_INFO class is:

```
class CMD_INFO
{
   public:
      char**       argv;       // command and argument list
      char*        infile;     // std input re-directed file
      char*        outfile;    // std. output re-directed file
      int          backgrnd;   // 1 if cmd to run in background
      CMD_INFO*    pSubcmd;    // cmds to be run in a sub-shell
      CMD_INFO*    Pipe;       // next command after '|'
      CMD_INFO*    Next;       // next command after ';'
};
```

Specifically, the *argv* variable stores a command and its switches and arguments in a vector of character strings. Thus, given a command as follows:

% sort -r /etc/motd > foo &

the *argv* variable of a CMD_INFO object that executes this command will be:

argv[0] = "sort"
argv[1] = "-r"
argv[2] = "/etc/motd"
argv[3] = 0;

The *infile* and *outfile* variables store the file names of any input and output redirection files, respectively, that are specified in a command. For the *sort* command example above, the *infile* and *outfile* variables of the CMD_INFO object are:

infile = 0;
outfile = "foo"

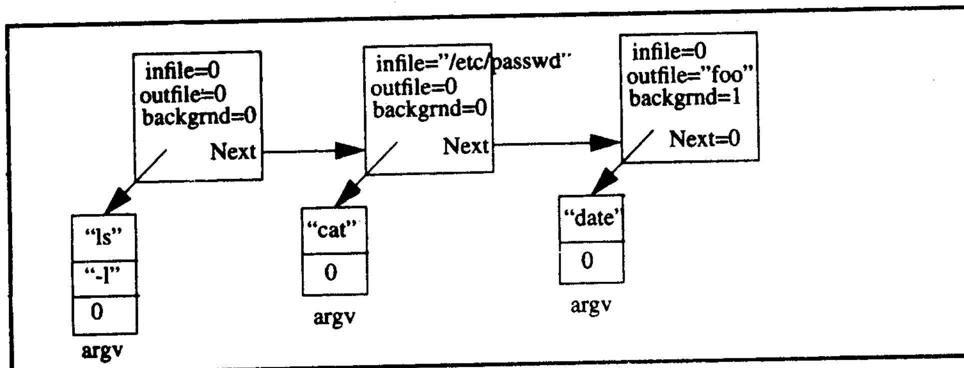
The *backgrnd* variable specifies whether a command is to be executed in foreground or background. The variable by default is 0 which means the corresponding command is to be executed in foreground. However, if "&" is specified in a command, the *backgrnd* variable of that command is set to 1. The *backgrnd* variable of the CMD_INFO object of the *sort* command is:

backgrnd = 1;

The *Next* variable is used to link another CMD_INFO object whose command is executed by the same shell process after the current object command. This allows users to execute multiple commands in a command line by delimiting commands with the ";" character. The command line is as follows:

% ls -l ; cat < /etc/passwd ; date > foo&

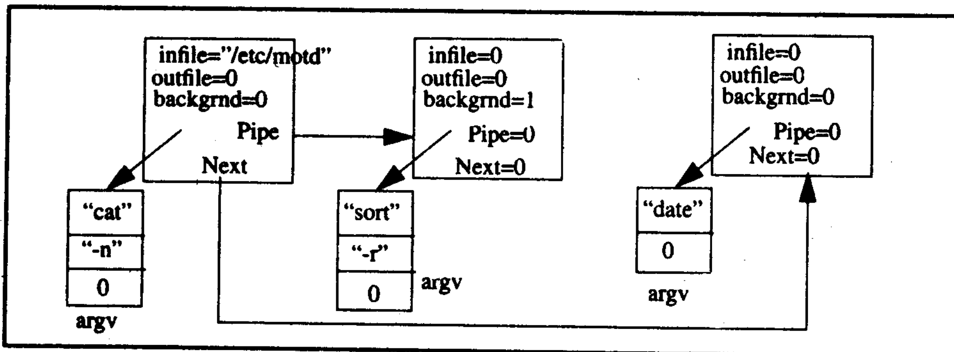
The CMD_INFO objects that execute the three UNIX commands are:



If the *Pipe* variable is not NULL, it specifies that when the current CMD_INFO object command is executed, its standard output data are piped to the standard input of the CMD_INFO object (pointed to by the *Pipe* variable). Furthermore, the commands of both the current object and the one pointed to by the *Pipe* variable will be executed concurrently by the same shell process. Thus, given the following command line:

```
%    cat -n < /etc/motd | sort -r &; date
```

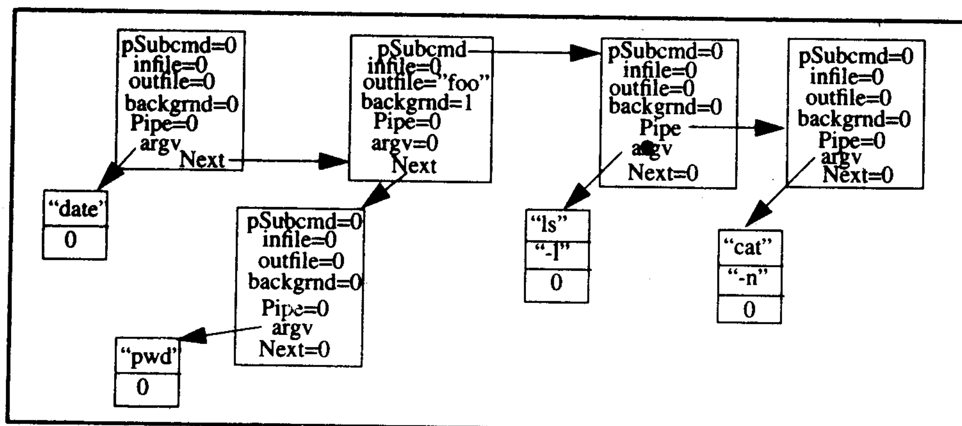
the CMD_INFO objects that execute these commands are:



Finally, the *pSubcmd* is used to point to a linked list of CMD_INFO objects whose commands are executed in a separate subshell process from the current shell process. If the *pSubcmd* is not NULL, the *argv* variable of the same object must be NULL, as the current shell's job for this command is to create a subshell to execute the objects pointed to by the *pSubcmd*. Thus, given the following command:

```
%    date;    (ls -l | cat -n) > foo &;        pwd
```

The CMD_INFO objects that execute these commands are:



The following *shell.h* header declares the CMD_INFO class and its member functions:

```
/* This is the shell.h header file, which declares the CMD_INFO class */
#ifndef    SHELL_H
#define    SHELL_H

#include <iostream.h>
#include <string.h>
#include <assert.h>
#include <malloc.h>

/* check IO redirection and command pipe conflict */
#define CHECK_IO(fn) if (fn) { \
      cerr << "Invalid re-direct: " << fn << endl; delete fn; fn = 0; }

class CMD_INFO
{
   public:
      char**         argv;       // command and argument list
      char*          infile;     // std input re-directed file
      char*          outfile;    // std. output re-directed file
      int            backgrnd;   // 1 if cmd to run in backgrnd
      CMD_INFO*      pSubcmd;    // cmds to be run in a sub-shell
      CMD_INFO*      Pipe;       // next command after '|'
      CMD_INFO*      Next;       // next command after ';'

      CMD_INFO()
      {
         argv = 0;
         infile = outfile = 0;
         backgrnd = 0;
         pSubcmd = Pipe = Next = 0;
      };

      ~CMD_INFO()
      {
         if (infile) delete infile;
         if (outfile) delete outfile;
         for (int i=0; argv && argv[i]; i++) free( argv[i] );
         free( argv );
      };

      // Add one argument string to argv list
      void add_arg( char* str )
      {
```

```
        int len = 1;
        if (!argv)
            argv = (char**)malloc(sizeof(char*)*2);
        else {
            while (argv[len]) len++;
            len++;
            argv = (char**)realloc(argv,sizeof(char*)*(len+1));
        }
        assert(argv[len-1] = strdup(str));
        argv[len] = 0;
    };

    // Add a standard input or output redirect file name
    void add_iofile( char*& iofile, char* fnm )
    {
        if (iofile)
            cerr << "Multiple in-direct: " << iofile << " vs " << fnm << endl;
        else iofile = fnm;
    };

    // Add a command pipe
    void add_pipe ( CMD_INFO* pCmd )
    {
        if (Pipe)
            Pipe->add_pipe(pCmd);
        else {
            CHECK_IO(outfile);
            CHECK_IO(pCmd->infile);
            Pipe = pCmd;
        }
    };

    // Add a next command stage
    void add_next( CMD_INFO* pCmd )
    {
        if (Next)
            Next->add_next(pCmd);
        else Next = pCmd;
    };
};  /* CMD_INFO */

/* function is defined in exec_cmd.C */
extern void exec_cmd ( CMD_INFO * cmdp );

#endif
```

The CMD_INFO::CMD_INFO constructor function initializes all variables of a newly created object to be zero.

The ~CMD_INFO::CMD_INFO destructor function deallocates any dynamic memory used by the *argv*, *infile*, and *outfile* variables of the object to be deleted.

The *CMD_INFO::add_arg* function is called to add word tokens that constitute a shell command to the *argv* variable of an object. This function uses the dynamic memory allocation functions *malloc* and *realloc* to adjust the size of the *argv* according to the actual number of word tokens present in a shell command.

The *CMD_INFO::add_iofile* function is called to add a file name (specified via the *fnm* argument) to either the *infile* or *outfile* variable (specified via the *iofile* argument) of an object. This file is used to redirect the standard input or output of a process that will be created to execute the object's command.

The *CMD_INFO::add_next* function adds a CMD_INFO object to the end of the *Next* linked list of an object. The *Next* linked list specifies a set of commands to be executed in the order of the records present in the list.

The *CMD_INFO::add_pipe* function adds a CMD_INFO object to the end of the *Pipe* linked list of an object. The *Pipe* linked list specifies a set of commands to be executed concurrently, with the standard output of one command piped to the standard input of the next command in the linked list. Moreover, the function checks that a CMD_INFO object cannot pipe data to a pipe and have its standard output redirected to a file at the same time. Similarly, a CMD_INFO object cannot receive data from a pipe and have its standard input redirected from a file at the same time.

With the CMD_INFO class defined, the minishell program will parse each command input line and build up one or more CMD_INFO object linked-lists to represent the corresponding shell commands of an input line. The parsing function of the minishell program is made up of a lexical analyzer and a parser that is created by using *lex* and yacc. Specifically, the lexical analyzer that recognizes command line tokens is constructed from a *lex* source file, as follows:

```
%{/* shell.l:minishell lexical analyzer lex source file */
#include <string.h>
#include "shell.h"
#include "y.tab.h"
%}
%%
;[ \t\v]*\n        return '\n';                /* skip ';' at end of line */
^[ \t\v]*\n        ;                           /* skip blank lines */
```

```
^#[^\n]\n          ;                                  /* skip comment lines */
#[^\n]*            ;                                  /* skip in-line comment s*/
[ \t\v]            ;                                  /* skip white space */
[A-Za-z_0-9/,.-]+{  yylval.s = strdup(yytext);
                    return NAME;                      /* return a chracter token */
                   }
                   |                                  /* a single char token */
\n                 return yytext[0];
%%

/* scanner wrap up routine */
int yywrap(){ return 1; }
```

The primary function of the lexical analyzer is to collect NAME tokens and special characters like "<," "<," "|," "(," ")," and ";" which constitute one or more shell commands in each command input line.

A NAME token consists of one or more alphanumeric characters,"-," ".," "_," ",," and "/" characters. It can be a shell command name, a command switch, or an argument to a shell command. When a NAME token is found, the lexical analyzer will return the token character string to the parser via the *yylval* global variable, and the NAME token ID which is defined in the y.tab.h (created from the yacc source file). Punctuation characters like "<," "<," "|," "(," and ")" are returned to the parser as they are.

In addition to the above, the lexical analyzer also ignores comments (a comment begins with a "#" character and is terminated by a newline character), white spaces, blank lines, and the optional ";" at the end of an input line.

Finally, the *yywrap* function is defined as required by *lex*. This function is called when the lexical analyzer encounters end-of-file in its input stream. The function instructs the scanner to stop scanning the input stream by returning one value, so that the parser and, hence, the minishell program, will halt.

The minishell parser expects its entire input data to consist of one or more command lines. Each command line is terminated by a newline character, and the parser will invoke a child process to execute that command line. Furthermore, a command line may consist of one or more shell commands delimited by ";" characters. These shell commands will be executed by the child process sequentially, in the order that they are specified, and any of these commands may be executed in the background if the "&" character is specified at the end of the commands. These syntax rules may be specified as:

```
<input_stream> ::=   [ <cmd_line> '\n' ]+
<cmd_line>     ::=   <shell> ['&'] [';' <shell> ['&'] ]+
```

A shell command may be a basic command, a pipe command, or a complex shell command:

```
<shell>      ::=    <basic> | <pipe> | <complex>
```

A basic command consists of a command name, optional switches and arguments, and any input and/or output redirection. The formal syntax of a basic command is:

```
<basic>     ::= <cmd> [ <switch> ]* [<arg>]* ['<' <file>] ['>' <file>]
```

The following are two examples of basic commands:

```
ls -l /etc/passwd
cat -n < srcfile > destfile
```

A pipe command consists of a set of basic and/or complex commands linked together by command pipes ("|"). The formal syntax of a pipe command is:

```
<pipe>      ::= [ <basic> | <complex> ] [ '|' <basic> | <complex> ]+
```

The following is an example of a pipe command:

```
ls -l /etc/passwd    |    sort -r   |    cat -n > foo
```

A complex command is one or more shell commands (simple, pipe, and/or complex) enclosed in parentheses and with optional input and output redirection. The formal syntax of a complex command is:

```
<complex>  ::= '(' <shell> [';' <shell> ]* ')' ['<' <file>] ['>' <file>]
```

The following is an example of a complex command:

```
( cat < /etc/passwd | sort -r | wc; pwd ) > foo
```

Putting all the above syntax rules together, the *yacc* source file for the minishell parser is as follows:

```
%{      /* shell.y:The minishell program parser */
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include "shell.h"
static CMD_INFO *    pCmd = 0;
%}

%union
{
   char*            s;
   CMD_INFO *    c;
}
%token <s> NAME

%%
input_stream :  cmd_line '\n'
                    {  exec_cmd($<c>1); }
                |   input_stream cmd_line '\n'
                    {  exec_cmd($<c>2); }
                |   error '\n'
                    {  yyerrok; yyclearin; }
                ;
cmd_line     :  shell backgrnd
                    {  $<c>$ = $<c>1;    }
                |   cmd_line ';' shell backgrnd
                    {  $<c>1->add_next($<c>3);
                       $<c>$ = $<c>1;
                    }
                ;
shell        :  basic
                    {  $<c>$ = $<c>1;    }
                |   complex
                    {  $<c>$ = $<c>1;    }
                |   shell '|' basic
                    {  $<c>1->add_pipe($<c>3);
                       $<c>$ = $<c>1;
                    }
                |   shell '|' complex
                    {  $<c>1->add_pipe($<c>3);
                       $<c>$ = $<c>1;
                    }
                ;
basic        :  cmd_spec io_spec
                    {  $<c>$ = $<c>1;    }
                ;
```

```
complex   :   '(' cmd_line ')'
                 {  pCmd = new CMD_INFO;
                    pCmd->pSubcmd = $<c>2;
                 }
               io_spec
                 {  $<c>$ = pCmd;    }
          ;
cmd_spec :    NAME
                 {  $<c>$ = pCmd = add_vect(0,$<s>1); }
          |    cmd_spec NAME
                 {  $<c>$ = add_vect($<c>1,$<s>2); }
          ;
io_spec   :    /* empty */
          |    io_spec redir
          ;
redir     :    '<' NAME
                 {  pCmd->add_iofile(pCmd->infile, $<s>2); }
          |    '>' NAME
                 {  pCmd->add_iofile(pCmd->outfile,$<s>2); }
          ;
backgrnd :    /* empty */
          |    '&'
                 {  pCmd->backgrnd = 1; }
          ;
%%
/* parser error reporting routine */
void yyerror(char* s)  {cerr << s << endl; }

/* add a cmd or arg to vector list */
CMD_INFO *add_vect (CMD_INFO* pCmd, char* str)
{
   int len = 1;
   if (!pCmd) assert(pCmd = new CMD_INFO);
   pCmd->add_arg(str);
   return pCmd:
}
```

In the *shell.y* file, the *cmd_spec* grammar rule recognizes a command name and any optional switches and arguments. When this rule matches, it will create a CMD_INFO object to contain the command data via the *add_vect* function and set the global *pCmd* pointer to point to this newly created object.

The *io_spec* rule collects the file names for any input and/or output redirection and adds these data to the current CMD_INFO object (pointed to by the *pCmd* pointer) via the *CMD_INFO::add_iofile* function.

The *basic* rule is made up of the *cmd_spec* and *io_spec* rules, and it represents one basic shell command specified by a user. It passes the CMD_INFO object that was created by the *cmd_spec* rule to the *shell* rule.

A *shell* rule may be matched by a *basic* rule, a *complex* rule, or *pipe* rule, which consists of a set of *basic* and/or *complex* rules that are separated by the "|" token. For the *pipe* rule the parser will link the CMD_INFO objects, via their *CMD_INFO::Pipe* pointer, that were created by the basic and complex rules through the *CMD_INFO::add_pipe* function. The *shell* rule returns to the *cmd_line* rule either a CMD_INFO object created by the *basic* or *complex* rule or a *Pipe* linked list of CMD_INFO objects.

The *cmd_line* rule is matched by one or more *shell* rules, each of which may be terminated with an optional "&" character and delimited by the ";" character. The parser will link the CMD_INFO objects returned from the *shell* rules together, via their *CMD_INFO::Next* pointer, into a *Next* linked list through the *CMD_INFO::add_next* function. The *cmd_line* rule represents one command input line to the minishell program and returns the first CMD_INFO object in a *Next* linked list (which is constructed to the *input_stream* or *complex* rule).

The *complex* rule is made up of a *cmd_line* rule that is specified in between a matching pair of parentheses characters. Furthermore, input and or output redirection maybe specified after the ')' character. The *complex* rule represents a set of shell commands to be executed in a separate subshell process. The parser will create a special CMD_INFO object to capture this complex rule. In this CMD_INFO object, the *CMD_INFO::argv* argument is NULL, the *CMD_INFO::pSubcmd* pointer will point to the CMD_INFO object (which may be a *Next* linked list) returned from the *cmd_line* rule, and any input and/or output redirection is recorded in the object *CMD_INFO::infile* and *CMD_INFO::outfile* variables. The *complex* rule returns the CMD_INFO object it created to the shell rule.

Finally, the *input_stream* rule is made up of one or more *cmd_line* rules, each terminated by the "\n" character. For each *cmd_line* rule that is matched, the parser will call the *exec_cmd* function to execute the shell commands associated with the CMD_INFO objects returned by the *cmd_line* rule. The *exec_cmd* function is depicted in the following:

```
/* exec_cmd.C: Functions to execute one shell command input line */
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include "shell.h"
```

```
/* change the standard I/O port of a process */
void chg_io( char* fileName, mode_t mode, int fdesc )
{
    int fd= open(fileName,mode,0777);
    if (fd==-1)
        perror("open");
    else {
        if (dup2(fd,fdesc)==-1) perror("dup2");
        close(fd);
    }
}

/* execute one or more command pipe */
void exec_pipes( CMD_INFO *pCmd )
{
    CMD_INFO *ptr;
    int     fifo[2][2];
    int     bg=0, first=1, cur_pipe = 0;
    pid_t pid;
    while (ptr=pCmd)  {
        pCmd = ptr->Pipe;
        if (pipe(fifo[cur_pipe])==-1)     {
            perror("pipe"); return;
        }
        switch(fork())     {
            case -1:   perror("fork");
                       return;
            case 0:    if (!first)     {                // not the first cmd
                           dup2(fifo[1-cur_pipe][0],0);
                           close(fifo[1-cur_pipe][0]);
                       }
                       else if (ptr->infile)
                           chg_io(ptr->infile,O_RDONLY,0);
                       if (pCmd)                        // not the last cmd
                           dup2(fifo[cur_pipe][1],1);
                       else if (ptr->outfile)
                           chg_io(ptr->outfile,
                                 O_WRONLY|O_CREAT|O_TRUNC,1);
                       close(fifo[cur_pipe][0]);
                       close(fifo[cur_pipe][1]);
                       execvp(ptr->argv[0],ptr->argv);
                       cerr << "Execute '" << ptr->argv[0] << "' fails\n";
                       exit(4);
        }
        if (!first) close(fifo[1-cur_pipe][0]);
        close(fifo[cur_pipe][1]);
```

```
        cur_pipe = 1 - cur_pipe;
        bg = ptr->backgrnd;
        delete ptr;
        first = 0;
    }
    close(fifo[1-cur_pipe][0]);
    while (!bg && (pid=waitpid(-1,0,0))!=-1) ;
}

/* execute one shell command line */
void exec_cmd( CMD_INFO *pCmd )
{
    pid_t prim_pid, pid;
    CMD_INFO *ptr = pCmd;

    // create a sub-shell to process one command line
    switch( prim_pid = fork())  {
        case -1:   perror("fork"); return,
        case 0:    break;
        default:   if (waitpid(prim_pid,0,0)!=prim_pid) perror("waitpid");
                   return;
    }
    while (ptr=pCmd)  {         // execute each command stage
        pCmd = ptr->Next;
        if (ptr->Pipe)
            exec_pipes(ptr);    // execute one stage which has pipe
        else  {
            // sub-process to execute one command stage
            switch (pid=fork()) {
                case -1:   perror("fork"); return;
                case 0:    break;
                default:   if (!ptr->backgrnd && waitpid(pid,0,0)!=pid)
                               perror("waitpid");
                           delete ptr;
                           continue;
            }
            if (ptr->infile)
                chg_io(ptr->infile,O_RDONLY,0);
            if (ptr->outfile)
                chg_io(ptr->outfile,O_WRONLY|O_CREAT|O_TRUNC,1);
            if (ptr->argv)    {
                execvp(ptr->argv[0], ptr->argv);
                cerr << "Execute '" << ptr->argv[0] << "' fails\n";
                exit(255);
            } else {
                exec_cmd(ptr->pSubcmd);
```

```
                exit(0);
            }
        }
    }
    exit(0);
}
```

The *exec_cmd* function emulates the UNIX shell in that it forks a child process to execute each shell command input line. Specifically, the function is called with a pointer to a CMD_INFO object, which represents one or more shell commands to be executed. The function first calls *fork* to create a child process, and the parent will call *waitpid* to wait for that child process to terminate. The child is the subshell process, and it scans through the given CMD_INFO object linked-list and executes each command associated with a CMD_INFO object as follows: If the command is a pipe command, it will call the *exec_pipe* command to execute the commands specified via the *Pipe* linked list of that object. If the command is a complex command or basic command, the subshell process will call *fork* to create a new child process to execute the command in the following manner:

If standard input and/or output redirection is specified in the CMD_INFO object, the new child process will change its standard input and/or output to the specified files via the *chg_io* function. If the command is a basic command, the new child process will call *exec* to execute the command specified in the *CMD_INFO::argv* of the object. However, if the command is a complex command (the *CMD_INFO::pSubcmd* of the object is not NULL), the new child process will call the *exec_cmd* function recursively to create a separate child process to execute the complex commands in a different context.

The subshell process will wait for the new child process to terminate before it processes the next CMD_INFO object (unless the *CMD_INFO::backgrnd* flag is set in the current object). This means the new child process is to be run in the background, and the subshell process will not call *waitpid* for the new child process.

If the *CMD_INFO::Pipe* pointer of a CMD_INFO object is not NULL, the object carries a pipe command, and the subshell process will call the *exec_pipe* function to create unnamed pipes and new child processes to execute the corresponding pipe commands.

The *main* function of the minishell program is:

```
/* shell.c:The minishell main program */
#include <iostream.h>
#include <stdio.h>
extern "C" int yyparse();
extern FILE* yyin;
```

```
int main(int argc, char* argv[])
{
   if (argc > 1)  {
      while (--argc > 0)
         if (yyin=fopen(*++argv,"r"))
            yyparse();
         else cerr << "Can't open file: " << *argv << endl;
   } else
      yyparse();
   return 0;
}
```

The main function will call the *yyparse* function to parse the input stream. The *yyparse* function is generated by *yacc* when the *shell.y* is translated by *yacc* to the *y.tab.h* file. The input stream to the minishell program may be the standard input if there is no command line argument to the program (hence, the *argc* value is 1) or it may one or more text files (shell scripts), which are named specifically by a user when the program is invoked. For the latter case, the *main* function will open each shell script file and direct the *yyparse* function to read from that file via the *yyin* global stream pointer.

The minishell program may be compiled as follows:

```
%    yacc -d    shell.y          # create y.tab.c and y.tab.h
%    lex   shell.l               # create lex.yy.c
%    CC -o shell shell.C exec_cmd.C lex.yy.c y.tab.c
```

The sample output of the program is depicted below. The user input commands are highlighted in italic to distinguish them from the minishell program outputs:

```
/export/home/terry/test1 7 > shell
date
Sat Aug 6 11:53:03 PDT 1994
cat -n /etc/motd
   Welcome to T.J. Systems
ls -l  |  cat -n   |  sort -r  |  wc
   10   93      635
pwd;   date;    ls |  wc ;      ps
/export/home/terry/test1
Sat Aug 6 11:54:21 PDT 1994
   9     9      69
PID    TTY     TIME        COMD
351    pts/2   0:00        shell
269    pts/2   0:00        csh
341    pts/2   0:00        shell
```

```
356      pts/2    0:01       ps
pwd &; (ls -l  |  cat -n   |  sort -r   |  wc) > xyz;    cat xyz
/export/home/terry/test1
   11   103     700
(ls -l  |  cat -n   |  sort -r   |  wc;  pwd) &;    date
Sat Aug 6 11:56:03 PDT 1994
   11   103        700
```

8.6 Summary

This chapter depicted the UNIX and POSIX APIs for process creation, control, communication between parent and child processes, and process attribute queries and changes. Furthermore, the methods used to change processes' standard input and output with files, establishing command pipelines, and executing shell commands in a user program are demonstrated. The final minishell example was a simplified UNIX shell program that put all the concepts described in the chapter together and illustrated the applications of those APIs.

One important aspect of process control is signal handling, which deals with the interaction between a process and an operating system kernel in handling asynchronous events. This is the main subject of the next chapter.

Signals

Signals are triggered by events and are posted on a process to notify it that something has happened and requires some action. An event can be generated from a process, a user, or the UNIX kernel. For example, if a process performs a divide-by-zero mathematical operation, or dereferences a NULL pointer, the kernel will send the process a signal to interrupt it. Furthermore, if a user hits the <Delete> or <Ctrl-C> key at the keyboard, the kernel will send the foreground process a signal to interrupt it. Finally, a parent and its child processes can send signals to each other for process synchronization. Thus, signals are the software version of hardware interrupts. Just as there are several levels of hardware interrupts on any given system, there are also different types of signals defined for different events that may occur in a UNIX system.

Signals are defined as integer flags, and the <signal.h> header depicts the list of signals defined for a UNIX system. The table below lists the POSIX-defined signals that are commonly found in most UNIX systems.

Signal name	Use	Core file generated at default
SIGALRM	Alarm timer time-outs. Can be generated by the alarm() API	No
SIGABRT	Abort process execution. Can be generated by the abort() API	Yes
SIGFPE	Illegal mathematical operation	Yes
SIGHUP	Controlling terminal hang-up	No
SIGILL	Execution of an illegal machine instruction	Yes
SIGINT	Process interruption. Commonly generated by the <Delete> or <ctrl-C> keys	No
SIGKILL	Sure kill a process. Can be generated by the *kill -9 <process_id>* command	Yes

Signal name	Use	Core file generated at default
SIGPIPE	Illegal write to a pipe	Yes
SIGQUIT	Process quit. Commonly generated by a control-\ keys	Yes
SIGSEGV	Segmentation fault. Can be generated by de-referencing a NULL pointer	Yes
SIGTERM	Process termination. Can be generated by the "kill <process_id>" command	Yes
SIGUSR1	Reserved to be defined by users	No
SIGUSR2	Reserved to be defined by users	No
SIGCHLD	Sent to a parent process when its child process has terminated	No
SIGCONT	Resume execution of a stopped process	No
SIGSTOP	Stop a process execution	No
SIGTTIN	Stop a background process when it tries to read from its controlling terminal	No
SIGTSTP	Stop a process execution by the control-Z keys	No
SIGTTOU	Stop a background process when it tries to write to its controlling terminal	No

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:

- Accepts 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 effect 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. If the function finishes its execution without terminating the process, the process will continue execution from the point it was interrupted by the signal

A process may set up per signal handling mechanisms, such that it ignores some signals, catches some other signals, and accepts the default action from the remaining signals. Furthermore, a process may change the handling of certain signals in its course of execution. For example, a signal may be ignored in the beginning, then set to be caught, and after being caught, set to accept the default action. A signal is said to have been *delivered* if it has been reacted to by the recipient process.

The default action for most signals is to terminate a recipient process (exceptions are the SIGCHLD and SIGPWR signals). Furthermore, some signals will generate a core file for the aborted process so that users can trace back the state of the process when it was aborted. These signals are usually generated when there is an implied program error in the aborted process. For example, the SIGSEGV signal is generated when a process tries to de-reference a NULL pointer. Thus if the process accepts the default action of SIGSEGV, a core file is generated when the process is aborted and the user can use the core file to debug the program.

Most signals can be ignored or caught except the SIGKILL and SIGSTOP signals. The SIGKILL signal can be generated by a user via the *kill -9 <process ID>* command in a UNIX shell. The SIGSTOP signal halts a process execution. For example, when you type <ctrl-Z> at the keyboard, the kernel will send the SIGSTOP signal to the foreground process to stop it. A companion signal to SIGSTOP is SIGCONT, which resumes a process execution after it has been stopped. SIGSTOP and SIGCONT signals are used for job control in UNIX.

A process is allowed to ignore certain signals so that it is not interrupted while doing certain mission-critical work. For example, when a database management process is updating a database file, it should not be interrupted until it is finished, otherwise, the database file will be corrupted. Thus, this process should specify that all common interrupt signals (e.g., SIGINT and SIGTERM) are to be ignored before it starts updating the database file. It should restore signal handling actions for these signals afterward.

Because most signals are generated asynchronously to a process, a process may specify a per signal handler function. These functions are called when their corresponding signals are caught. A common practice of a signal handler function is to clean up a process work environment, such as closing all input and output files, before terminating the process gracefully.

9.1 The UNIX Kernel Supports of Signals

In UNIX System V.3, each entry in the kernel Process Table slot has an array of signal flags, one for each signal defined in the system. 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. Furthermore, if the recipient process is asleep (for example, it is waiting for a child process to terminate or is executing the *pause* API), the kernel will awaken the process by scheduling it as well. When the recipient process runs, the kernel will check the process *U*-area that contains an array of signal handling specifications, where each entry of the array corresponds to a signal defined in the system. The kernel will consult the array to find out how the process will react to the pending signal. If the array entry for the signal contains a zero value, the process will accept the default action of the signal. If the array entry contains a 1 value, the process will ignore the signal, and the kernel will discard it. Finally, if the array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine. The kernel will set up the process to execute that function immediately, and the process will return to its current point of execution (or to someplace else if the signal handler does a long jump) if the signal handler function does not terminate the process.

If there are different signals pending on a process, the order in which they are sent to a recipient process is undefined. Furthermore, if multiple instances of a signal are pending on a process, it is implementation-dependent on whether a single instance or multiple instances of the signal will be delivered to the process. In UNIX System V.3, each signal flag in a Process Table slot records only whether a signal is pending, but not how many of them are present.

The way caught signals are handled by UNIX System V.2 and by earlier versions has been criticized as unreliable. Subsequently, BSD UNIX 4.2 (and later versions) and POSIX.1 use different mechanisms to handle caught signals.

Specifically, in UNIX System V.2 and earlier versions, when a signal is caught, the kernel will first reset the signal handler (for that signal) in the recipient process *U*-area, then call the user signal handling function specified for that signal. Thus, if there are multiple instances of a signal being sent to a process at different points, the process will catch only the first instance of the signal. All subsequent instances of the signal will be handled in the default manner.

For a process to continuously catch multiple occurrences of a signal, the process must reinstall the signal handler function every time the signal is caught. However, this is still not a guarantee that the process will catch the signal every time: between the time a signal handler is invoked for a caught signal *X* and the time the signal handler method is reestablished, the process is in a state of accepting the default action for signal *X*. If another instance of signal *X* is delivered to the process during that interval, the process will have to handle the signal in the default manner. This is a race condition, where two events occur simultaneously, and which event will take effect first is unpredictable.

To remedy the unreliability of signal handling in System V.2, BSD UNIX 4.2 (and later versions) and POSIX.1 use a different method: When a signal is caught, the kernel does not reset the signal handler, so there is no need for the process to reestablish the signal handling method. Furthermore, the kernel will block further delivery of the same signal to the process until the signal handler function has completed execution. This ensures that the signal handler function will not be invoked recursively for multiple instances of the same signal. System V.3 introduced the *sigset* API, which behaves in such a reliable manner also.

UNIX System V.4 has adopted the POSIX.1 signal handling method. However, users still have the option to instruct the kernel to use the System V.2 signal handling method on a per-signal basis. This is done via the *signal* APIs, as described next.

9.2 signal

All UNIX systems and ANSI-C support the *signal* API, which can be used to define the per-signal handling method. The function prototype of the *signal* API is:

```
#include <signal.h>

void (*signal ( int signal_num, void (*handler)(int) ) ) (int);
```

The formal arguments of the API are: *signal_num* is a signal identifier like SIGINT or SIGTERM, as defined in the <signal.h> header. The *handler* argument is the function pointer of a user-defined signal handler function. This function should take an integer formal argument and does not return any value.

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_nm << 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 and SIG_DFL are manifest constants defined in the <signal.h> header:

```
#define SIG_DFL      void (*)(int)0
#define SIG_IGN      void (*)(int)1
```

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.

The return value of the *signal* API is the previous signal handler for a signal. This can be used to restore the signal handler for a signal after it has been altered:

```
#include <signal.h>
int main()
{
```

```
        void (*old_handler)(int) = signal (SIGINT, SIG_IGN);
        /* do mission critical processing */
        signal (SIGINT, old_handler);  /* restore previous signal handling */
}
```

The *signal* API is not a POSIX.1 standard. However, it is defined by ANSI-C and is available on all UNIX systems. Because the behavior of the *signal* API in System V.2 and earlier versions is different than that in BSD and POSIX.1 systems, it is not recommended to be used by portable applications.The BSD UNIX and the POSIX.1 define a new set of APIs for signal manipulation. The API's behavior is consistent in all UNIX and POSIX.1 systems that support them and they are described in the next two sections.

Note that UNIX System V.3 and V.4 support the *sigset* API, which has the same prototype and similar use as *signal*:

```
#include <signal.h>

void (*sigset ( int signal_num, void (*handler)(int) ) ) (int);
```

The *sigset* arguments and return value are the same as that of *signal*. Both functions set signal handling methods for any named signal. However, whereas the *signal* API is unreliable (as explained in Section 9.1), the *sigset* API is reliable. This means that when a signal is set to be caught by a signal handler via *sigset*, when multiple instances of the signal arrive one of them is handled while the other instances are blocked. Furthermore, the signal handler is not reset to SIG_DFT when the it is invoked.

9.3 Signal Mask

Each process in a UNIX (BSD 4.2 and later, and System V.4) or POSIX.1 system has a signal mask that defines which signals are blocked when generated to a process. A blocked signal depends on the recipient process to unblock it and handle it accordingly. If a signal is specified to be ignored and blocked, it is implementation-dependent on whether such a signal will be discarded or left pending when it is sent to the process.

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 );
```

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.

The *old_mask* argument is the address of a *sigset_t* variable that will be assigned the calling process's original signal mask prior to a *sigprocmask* call. If the actual argument to *old_mask* is a NULL pointer, no previous signal mask will be returned.

The return value of a *sigprocmask* call is zero if it succeeds or -1 if it fails. Possible failure may occur because the *new_mask* and/or the *old_mask* actual arguments are invalid addresses.

The *sigset_t* is a data type defined in the <signal.h> header. It contains a collect of bit-flags, with each bit-flag representing one signal defined in a given system.

The BSD UNIX and POSIX.1 define a set of API known as *sigsetops* functions, which set, reset, and query the presence of signals in a *sigset_t*-typed variable:

```
#include <signal.h>

int sigemptyset ( sigset_t* sigmask );
int sigaddset ( sigset_t* sigmask, const int signal_num );
int sigdelset ( sigset_t* sigmask, const int signal_num);
int sigfillset ( sigset_t* sigmask );
int sigismember ( const sigset_t* sigmask, const int signal_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.

The return value of the *sigemptyset, sigaddset, sigdelset*, and *sigfillset* calls is zero if the calls succeed or -1 if they fail. Possible causes of failure may be that the *sigmask* and/or the *signal_num* arguments are invalid.

The *sigismember* API returns 1 if the flag corresponding to the *signal_num* signal in the *sigmask* argument is set, zero if it is 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. Then it clears the SIGSEGV signal from the process signal mask:

```
#include <stdio.h>
#include <signal.h>
int main ()
{
    sigset_t        sigmask;
    sigemptyset(&sigmask);                    /* initialize 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");                 /* set a new signal mask */
}
```

When one or more signals are pending for a process and are unblocked via the *sigprocmask* API, the signal handler methods for those signals that are in effect at the time of the *sigprocmask* call will be applied before the API is returned to the caller. If there are multiple

instances of the same signal pending for the process, it is implementation-dependent whether one or all of those instances will be delivered to the process.

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

```
#include <signal.h>
int sigpending ( sigset_t* sigmask );
```

The *sigmask* argument to the *sigpending* API is the address of a *sigset_t*-typed variable and is assigned the set of signals pending for the calling process by the API. The API returns a zero if it succeeds and a -1 value if it 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.

The following example reports to the console whether the SIGTERM signal is pending for the process:

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

int main()
{
      sigset_t          sigmask;
      sigemptyset(&sigmask);
      if (sigpending(&sigmask)==-1)
         perror( "sigpending");
      else cout << "SIGTERM signal is: "
         << (sigismember(&sigmask,SIGTERM) ? "Set" : "No Set" )
         << endl;
}
```

Note that, in addition to the above APIs, UNIX System V.3 and V.4 also support the following APIs as simplified means for signal mask manipulation:

```
#include <signal.h>

int sighold ( nt signal_num );
int sigrelse ( int signal_num );
int sigignore ( int signal_num);
int sigpause ( int signal_num );
```

The *sighold* API adds the named signal *signal_num* to the calling process signal mask. It is the same as using the *sigset* API with the SIG_HOLD action:

```
sigset ( <signal_num>, SIG_HOLD );
```

The *sigrelse* API removes the named signal *signal_num* for the calling process signal mask.

The *sigignore* API sets the signal handling method for the named signal *signal_num* to SIG_DFT.

Finally, the *sigpause* API removes the named *signal* signal_num from the calling process signal mask and suspends the process until it is interrupted by a signal.

9.4 sigaction

The *sigaction* API is a replacement for the signal API in the latest UNIX and POSIX. systems. Like the *signal* API, the *sigaction* API is called by a process to setup a signal handling method for each signal it wants to deal with. Both APIs pass back the previous signal handling method for a given signal. Furthermore, 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);
```

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 *sa_handler* field corresponds to the second argument of the *signal* API. It can be set to SIG_IGN, SIG_DFL, or a user-defined signal handler function. The *sa_mask* field specifies additional signals that a process wishes to block (besides those signals currently specified in the process's signal mask and the *signal_num* signal) when it is handling the *signal_num* signal.

Putting all these together, the *signal_num* argument designates which signal handling action is defined in the *action* argument. The previous signal handling method for *signal_num* will be returned via the *old_action* argument if it is not a NULL pointer. If the *action* argument is a NULL pointer, the calling process's existing signal handling method for *signal_num* will be unchanged.

The following *sigaction.C* program illustrates 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();                                    /* wait for signal interruption */

        cout << argv[0] << " exists\n";
        return 0;
}.
```

In the above example, the process signal mask is set with the SIGTERM signal. The process then defines a signal handler for the SIGINT signal and also specifies that the SIGSEGV signal is to be blocked when the process is handling the SIGINT signal. The process then suspends its execution via the *pause* API.

The sample output of the program is:

```
% CC sigaction.C -o sigaction
% sigaction &
[1] 495
% kill -INT 495
catch signal: 2
sigaction exits
[1]  Done            sigaction
```

If the SIGINT signal is generated to the process, the kernel first sets the process signal mask to block the SIGTERM, SIGINT, and SIGSEGV signals. It then arranges the process to execute the *callme* signal handler function. When the *callme* function returns, the process signal mask is restored to contain only the SIGTERM signal, and the process will continue to catch the SIGILL signal.

The *sa_flag* field of the *struct sigaction* is used to specify special handling for certain signals. POSIX.1 defines only two values for the *sa_flag*: zero or SA_NOCLDSTOP. The SA_NOCLDSTOP flag is an integer literal defined in the <signal.h> header and can be used when the *signal_num* is SIGCHLD. The effect of the SA_NOCHLDSTOP flag is that the kernel will generate the SIGCHLD signal to a process when its child process has terminated, but not when the child process has been stopped. On the other hand, if the *sa_flag* value is zero in a *sigaction* call for SIGCHLD, the kernel will send the SIGCHLD signal to the calling process whenever its child process is either terminated or stopped.

UNIX System V.4 defines additional flags for the *sa_flag* field of *struct sigaction*. These flags can be used to specify the UNIX System V.3 style of signal handling method:

***sa_flag* value**	**Effects on handling of *signal_num***
SA_RESETHAND	If *signal_num* is caught, the *sa_handler* is set to SIG_DFL before the signal handler function is called, and *signal_num* will not be added to the process signal mask when the signal handler function is executed
SA_RESTART	If a signal is caught while a process is executing a system call, the kernel will restart the system call after the signal handler returns. If this flag is not set in the *sa_flag*, after the signal handler returns, the system call will be aborted with a return value of -1 and will set *errno* to EINTR

9.5 The SIGCHLD Signal and the waitpid API

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:

1. Parent accepts the default action of the SIGCHLD signal: Unlike most signals, the SIGCHLD signal does not terminate the parent process. It affects only the parent process if it arrives at the same time the parent process is suspended by the *waitpid* system call. If that is the case, the parent process will be awakened, the API will return the child's exist status and process ID to the parent, and the kernel will clear up the Process Table slot allocated for the child process. Thus, with this setup, a parent process can call the *waitpid* API repeatedly to wait for each child it created.

2. Parent ignores the SIGCHLD signal: The SIGCHLD signal will be discarded, and the parent will not be disturbed, even if it is executing the *waitpid* system call. The effect of this setup is that if the parent calls the *waitpid* API, the API will suspend the parent until all its child processes have terminated. Furthermore, the child process table slots will be cleared up by the kernel, and the API will return a -1 value to the parent process.

3. Process catches the SIGCHLD signal: The signal handler function will be called in the parent process whenever a child process terminates. Furthermore, if the SIGCHLD signal arrives while the parent process is executing the *waitpid* system call, after the signal handler function returns, the *waitpid* API may be restarted to

collect the child exit status and clear its Process Table slot. On the other hand, the API may be aborted and the child Process Table slot not freed, depending on the parent setup of the signal action for the SIGCHLD signal.

The interaction between SIGCHLD and the *wait* API is the same as that between SIGCHLD and the *waitpid* API. Furthermore, earlier versions of UNIX use the SIGCLD signal instead of SIGCHLD. The SIGCLD signal is now obsolete, but most of the latest UNIX systems have defined SIGCLD to be the same as SIGCHLD for backward compatibility.

9.6 The sigsetjmp and siglongjmp APIs

The *sigsetjmp* and *siglongjmp* APIs have similar functions as their corresponding *setjmp* and *longjmp* APIs. Specifically, both *setjmp* and *sigsetjmp* mark one or more locations in a user program. Later on, the program may call the *longjmp* or *siglongjmp* API to return to any of those marked location. Thus, these APIs provide interfunction *goto* capability.

The *sigsetjmp* and *siglongjmp* APIs are defined in POSIX.1 and on most UNIX systems that support signal masks. The function prototypes of the APIs are:

```
#include <setjmp.h>

int sigsetjmp ( sigjmpbuf env, int save_sigmask );
int siglongjmp ( sigjmpbuf env, int ret_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 *sigsetjmp* API behaves similarly to the *setjmp* API, except that it has a second argument, *save_sigmask,* which allows a user to specify whether a calling process signal mask should be saved to the provided *env* argument. Specifically, if the *save_sigmask* argument is nonzero, the caller's signal mask is saved. Otherwise, the signal mask is not saved.

The *siglongjmp* API does all the operations as the *longjmp* API, but it also restores a calling process signal mask if the mask was saved in its *env* argument. The *ret_val* argument specifies the return value of the corresponding *sigsetjmp* API when it is called by *siglongjmp*. Its value should be a nonzero number, and if it is zero the *siglongjmp* API will reset it to 1.

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 (if a user does not want to resume execution at the code where the signal interruption occurred) to ensure the process signal mask is restored properly when "jumping out" from a signal handling function.

The following *sigsetjmp.C* program illustrates the uses of *sigsetjmp* and *siglongjmp* APIs. The program is modified from the *sigaction.C* program, as depicted in Section 9.4. Specifically, the program sets its signal mask to contain SIGTERM, then sets up a signal trap for the SIGINT signal. The program then calls *sigsetjmp* to store its code location in the *env* global variable. Note that the *sigsetjmp* call returns a zero value when it is called directly in user program and not via *siglongjmp*. The program suspends its execution via the *pause* API. When a user interrupts the process from the keyboard, the *callme* function is called. The *callme* function calls the *siglongjmp* API to transfer program flow back to the *sigsetjmp* function (in the *main* function), which now returns a 2 value.

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

sigjmp_buf      env;

void callme( int sig_num )
{
    cout << "catch signal: " << sig_num << endl;
    siglongjmp( env, 2 );
}

int main()
{
    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 = (void (*)())callme;
    action.sa_flags = 0;

    if (sigaction(SIGINT,&action,&old_action)==-1)
       perror( "sigaction");
```

```
        if (sigsetjmp( env, 1 ) != 0 )        {
                cerr << "Return from signal interruption\n";
                return 0;
        }
        else    cerr << "Return from first time sigsetjmp is called\n";

        pause();            // wait for signal interruption (e.g., from keyboard)
}
```

The sample output of the above program is:

```
% CC sigsetjmp.C
% a.out &
[1] 377
Return from first time sigsetjmp is called
% kill -INT 377
catch signal: 2
Return from signal interruption
[1]  Done         a.out
%
```

9.7 kill

A process can send a signal to a related process via the *kill* API. This is a simple means of interprocess communication or control. The sender and recipient processes must be related such that either the sender process real or effective user ID matches that of the recipient process, or the sender process has superuser privileges. For example, a parent and a child process can send signals to each other via the *kill* API.

The *kill* API is defined in most UNIX systems and is a POSIX.1 standard. The function prototype of the API is:

```
#include <signal.h>

int kill ( pid_t pid, int signal_num );
```

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 value	Effects on the *kill* API
a positive value	*pid* is a process ID. Sends *signal_num* to that process
0	Sends *signal_num* to all processes whose process group ID is the same as the calling process
-1	Sends *signal_num* to all processes whose real user ID is the same as the effective user ID of the calling process. If the calling process effective user ID is the superuser user ID, *signal_num* will be sent to all processes in the system (except processes-0 and 1). The latter case is used when the system is shutting down -- the kernel calls the *kill* API to terminate all processes except 0 and 1. Note that POSIX.1 does not specify the behavior of the *kill* API when the *pid* value is -1. The above effects are for UNIX systems only
a negative value	Sends *signal_num* to all processes whose process group ID matches the absolute value of *pid*

The return value of *kill* is zero if it succeeds or -1 if it fails.

The following *kill.C* 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)  {            /* get signal number */
            cerr << "Invalid number: " << argv[1] << endl;
            return -1;
         }
         argv++, argc--;
      }
      while (--argc > 0)
```

```
        if (sscanf(*++argv,"%d",&pid)==1) {          /* get process ID */
            if (kill (pid, sig)==-1)
                perror("kill");
        }
        else 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, as defined in the <signal.h> header. The *<Pid>* is the integer number of a process ID. There can be one or more *<Pid>* specified, and the *kill* command will send the signal *<signal_num>* to each process that corresponds to a *<Pid>*.

To simplify the above program, any signal specification at the command line must be a signal's integer value. It does not support signal symbolic names. If no signal number is specified, the program will use the default signal SIGTERM, which is the same for the UNIX *kill* command. The program calls the *kill* API to send a signal to each process whose process ID is specified at the command line. If a process ID is invalid or if the *kill* API fails, the program will flag an error message.

9.8 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. This is like setting an alarm clock to remind someone to do something after a specified period of time.

The *alarm* API is defined in most UNIX systems and is a POSIX.1 standard. The function prototype of the API is:

```
#include <signal.h>

unsigned int alarm ( unsigned int time_interval );
```

The *time_interval* argument is the number of CPU seconds elapse time, after which the kernel will send the SIGALRM signal to the calling process. If a *time_interval* value is zero, it turns off the alarm clock.

The return value of the *alarm* API is the number of CPU seconds left in the process timer, as set by a previous *alarm* system call. The effect of the previous *alarm* API call is canceled, and the process timer is reset with the new *alarm* call. A process alarm clock is not passed on to its forked child process, but an *exec*'ed process retains the same alarm clock value as was prior to the *exec* API call.

The *alarm* API can be used to implement the *sleep* API:

```
/* sleep.C */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void wakeup()              {};

unsigned int sleep ( unsigned int timer )
{
        struct sigaction action;
        action.sa_handler = wakeup;
        action.sa_flags = 0;
        sigemptyset(&action.sa_mask);

        if (sigaction(SIGALRM, &action,0)==-1)  {
                perror("sigaction");
                return -1;
        }
        (void)alarm( timer );
        (void)pause();
        return 0;
}
```

The *sleep* API suspends a calling process for the specified number of CPU seconds The process will be awakened by either the elapse time exceeding the *timer* value or when the process is interrupted by a signal.

In the above example, the *sleep* function sets up a signal handler for tne SIGALRM, calls the *alarm* API to request the kernel to send the SIGALRM signal (after the *timer* interval), and finally, suspends its execution via the *pause* system call. The *wakeup* signal handler function is called when the SIGALRM signal is sent to the process. When it returns, the *pause* system call will be aborted, and the calling process will return from the *sleep* function.

BSD UNIX defines the *ualarm* function, which is the same function as the *alarm* API, except that the argument and return value of the *ualarm* function are in microsecond units. This is useful for some time-critical applications where the resolution of time must be in microsecond levels.

The *ualarm* function can be used to implement the BSD-specific *usleep* function, which is like the *sleep* function, except its argument is in microsecond units.

9.9 Interval Timers

The *sleep* function that suspends a process for a fixed amount of time is only one use of the *alarm* API. The more general use of the *alarm* API is to set up an interval timer in a process. 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, *timer.C*, 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( SIGALRM,&action,0 )==-1 )  {
        perror( "sigaction");
        return 1;
     }
     if (alarm( INTERVAL ) == -1)
        perror("alarm" );
     else while( 1 )  {
        /* do normal operation */
     }
     return 0;
  }
```

In the above program, the *sigaction* API is called to set up *callme* as the signal handling function for the SIGALRM signal. The program then invokes the *alarm* API to send itself the

SIGALRM after 5 real clock seconds. The program then goes off to perform its normal operation in an infinite loop. When the timer expires, the *callme* function is invoked, which restarts the alarm clock for another 5 seconds and then does the scheduled tasks. When the *callme* function returns, the program continues its "normal" operation until another timer expiration.

The above sample program may be useful in creating a clock synchronization program: Every time the *callme* function is invoked, it polls a remote host for current time, then calls the *stime* API to set the local system clock to be the same as the reference host.

In addition to using the *alarm* API to set up an interval timer in a process, BSD UNIX invented the *setitimer* API, which provides capabilities additional to those of the *alarm* API:

* The *setitimer* resolution time is in microseconds, whereas the resolution time for *alarm* is in seconds

* The *alarm* API can be used to set up one real-time clock timer per process. The *setitimer* API can be used to define up to three different types of timers in a process:

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

The *setitimer* API is also available in UNIX System V.3 and V.4. However, it is not specified by POSIX. POSIX.1b defines a new set of APIs for interval timer manipulation. These APIs are described in the next section.

The *getitimer* API is also defined in BSD and System V UNIX for users to query the timer values that are set by the *setitimer* API.

The *setitimer* and *getitimer* function prototypes are:

```
#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:

which **argument value**	**Timer type**
ITIMER_REAL	Timer based on real-time clock. Generates a SIGALRM signal when it expires
ITIMER_VIRTUAL	Timer based on user-time spent by a process. Generates a SIGVTALRM signal when it expires
ITIMER_PROF	Timer based on total user and system times spent by a process. Generates a SIGPROF signal when it expires

The *struct itimerval* data type is defined in the <sys/time.h> header as:

```
struct itimeval
{
        struct timeval      it_interval;            // timer interval
        struct timeval      it_value;               // current value
};
```

For the *setitimer* API, the *val.it_value* is the time to set the named timer, and the *val.it_interval* is the time to reload the timer when it expires. The *val.it_interval* may be set to zero if the timer is to run once only. Furthermore, if the *val.it_value* value is set to zero, it stops the named timer if it is running.

For the *getitimer* API, the *old.it_value* and the *old.t_interval* return the named timer's remaining time (to expiration) and the reload time, respectively.

The *old* argument of the *setitimer* API is like the *old* argument of the *getitimer* API. If this is an address of a *struct itimeval*-typed variable, it returns the previous timer value. If the *old* argument is set to NULL, the old timer value will not be returned.

The ITIMER_VIRTUAL and ITIMER_PROF timers are primary useful in timing the total execution time of selected user functions, as the timer runs only while the user process is running (or the kernel is executing system functions on behalf of the user process for the ITIMER_PROF timer).

The *setitimer* and *getitimer* APIs return a zero value if they succeed or a -1 value if they fail. Moreover, timers set by the *setitimer* API in a parent process are not inherited by its child processes, but these timers are retained when a process *exec*'s a new program.

The following example program, *timer2.C*, is the same as the *timer.C* program, except that it uses the *setitimer* API instead of the *alarm* API. Also, there is no need to call the *setitimer* API inside the signal handling function, as the timer is specified to be reloaded automatically:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <signal.h>
#define  INTERVAL  2

void callme( int sig_no )
{
     /* do some schedule tasks */
}

int main()
{
     struct itimerval val;
     struct sigaction  action;

     sigemptyset(&action.sa_mask);
     action.sa_handler = (void (*)())callme;
     action.sa_flags = SA_RESTART;
     if (sigactionSIGALRM,&action,0)==-1) {
        perror( "sigaction");
        return 1;
     }

     val.it_interval.tv_sec        = INTERVAL;
     va' it_interval.tv_usec       = 0;
     val.it_value.tv_sec           = INTERVAL;
     val.it_value.tv_usec          = 0;

     if (setitimer( ITIMER_REAL, &val, 0 ) == -1)
        perror("alarm" );
     else while( 1 )  {
        /* do normal operation */
     }
     return 0;
}
```

Note that the real time clock timer set by the *setitimer* API is different from that set by the *alarm* API. Thus, a process may set up two real-time clock timers using the two APIs. Furthermore, since the *alarm* and *setitimer* APIs require that users set up signal handling to catch timer expiration, they (when used to set up real-time clock timer) should not be used in conjunction with the *sleep* API. This is because the *sleep* API may modify the signal handling function for the SIGALRM signal.

9.10 POSIX.1b Timers

POSIX.1b defines a set of APIs for interval timer manipulation. The POSIX.1b timers are more flexible and powerful than are the UNIX timers in the following ways:

- Users may define multiple independent timers per system clock.
- The timer resolution is in nanoseconds.
- Users may specify, on a per-timer basis, the signal to be raised when a timer expires.
- The timer interval may be specified as either an absolute or a relative time.

There is a limit on how many POSIX timers can be created per process. This maximum limit is the TIMER_MAX constant, as defined in the <limits.h> header. Moreover, POSIX timers created by a process are not inherited by its child processes, but are retained across the *exec* system call. However, unlike the UNIX timers, if a POSIX.1 timer does not use the SIGALRM signal when it expires, it can be used safely with the *sleep* API in the same program.

The POSIX.1b APIs for timer manipulation are:

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

int timer_create ( clockid_t clock, struct sigevent* spec, timer_t* timer_hdrp);
int timer_settime ( timer_t timer_hdr, int flag, struct itimrspec*val,
                                        struct itimerspec* old );
int timer_gettime ( timer_t timer_hdr, struct itimerspec*old );

int timer_getoverrun ( timer_t timer_hdr );
int timer_delete ( timer_t timer_hdr );
```

The *timer_create* API is used to dynamically create a timer and returns its handler. The *clock* argument specifies which system clock the new timer should be based on. The *clock* argument value may be CLOCK_REALTIME for creating a real time clock timer. This value is defined by POSIX.1b. Other values for the *clock* argument are system-dependent.

The *spec* argument defines what action to take when the timer expires. The *struct sigevent* data type is defined as:

```
struct sigevent
{
```

```
        int             sigev_notify;
        int             sigev_signo;
        union sigval    sigev_value;
};
```

The *sigev_signo* field specifies a signal number to be raised at the timer expiration. It is valid only when the *sigev_notify* field is set to SIGEV_SIGNAL. If the *sigev_notify* field is set to SIGEV_NONE, no signal is raised by the timer when it expires. Because multiple timers may generate the same signal, the *sigev_value* field is used to contain any user-defined data to identify that a signal is raised by a specific timer. The data structure of the *sigev_value* field is:

```
union sigval {
        int             sival_int;
        void            *sival_ptr;
};
```

For example, a process may assign each timer an unique integer ID number. This number may then be assigned to the *spec->sigev_value.sival_int* field. Furthermore, to pass this data along with the signal (*spec->sigev_signo*) when it is raised, the SA_SIGINFO flag should be set in an *sigaction* call, which sets up the handling for the signal, and the signal handling function prototype should be:

void ***<signal_handler>*** (**int** signo, **siginfo_t*** evp, **void*** ucontext);

The data structure of the *siginfo_t* is defined in the <siginfo.h> header. When the signal handler is called, the *evp->si_value* contains the data of the *spec->sigev_value*.

If the *spec* argument is set to NULL and the timer is based on CLOCK_REALTIME, the SIGALRM signal is raised when the timer expires.

Finally, the *timer_hdrp* argument of the *timer_create* API is an address of a *timer_t*-typed variable to hold the handler of the newly generated timer. This argument should not be set to NULL, as the handler is used to call other POSIX.1b timer APIs.

The *timer_create* API, as well as all the following POSIX1.b timer APIs, return zero if they succeed and -1 if they fail.

The *timer_settime* starts or stops a timer running. The *timer_gettime* API is used to query the current values of a timer. Specifically, the *struct itimerpsec* data type is defined as:

```
struct itimerspec {
        struct timespec     it_interval;
```

```
                struct timespec         it_value;
        };
```

and the *struct timespec* data structure is defined as:

```
        struct timespec {
                time_t                  tv_sec;
                long                    tv_nsec;
        };
```

The *itimerspec::it_value* specifies the time remaining in the timer, and the *itimerspec::it_interval* specifies the new time to reload the timer after it expires. All times are specified in seconds (via the *timespec::tv_sec* field) and in nanoseconds (via the *timespec::tv_nsec* field).

In the *timer_settime* API, the *flag* argument value may be 0 or TIMER_RELTIME if the timer start time (as contained in the *val* argument) is relative to the current time. If the *flag* argument value is TIMER_ABSTIME, the timer start time is an absolute time. Note that the ANSI C *mktime* function may be used to generate the absolute time for setting a timer. Note that if the *val.it_value* is zero, it stops the timer from running. Furthermore, if the *val.it_interval* is zero, the timer will not restart after it expires. Finally, the *old* argument of the *timer_settime* API is used to obtain the previous timer values. The *old* argument value may be set to NULL, and no timer values are returned.

The *old* argument of the *timer_gettime* API returns the current values of the named timer.

The *timer_getoverrun* API returns the number of signals generated by a timer but was lost (overrun). Specifically, timer signals are not queued by the kernel if they are raised but are not being handled by their target processes (maybe they were busy handling other signals). Instead, the kernel records the number of these overrun signals per timer. The *timer_getoverrun* API can be used to determine the amount of time elapsed (between the timer started or handled to the present time), based on the overrun count of a named timer. Note that the overrun count in a timer is reset whenever a process handles the timer signal.

The *timer_destroy* API is used to destroy a timer created by the *timer_create* API.

The following program., *posix_timer_abs.C*, illustrates how to set up an absolute-time timer that will go off at 10:27 AM, April 20, 1996:

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

```
#include <time.h>
#define     TIMER_TAG        12

void callme( int signo, siginfo_t* evp, void* ucontext )
{
      time_t tim = time(0);
      cerr  << "callme: " << evp->si_value.sival_int
            << ", signo: " << signo << ", " << ctime(&tim);
}
int main()
{
      struct sigaction        sigv;
      struct sigevent         sigx;
      struct itimerspec       val;
      struct tm               do_time;
      timer_t                 t_id;

      sigemptyset( &sigv.sa_mask );
      sigv.sa_flags = SA_SIGINFO;
      sigv.sa_sigaction = callme;

      if (sigaction( SIGUSR1, &sigv, 0) == -1) {
            perror("sigaction");
            return 1;
      }

      sigx.sigev_notify = SIGEV_SIGNAL;
      sigx.sigev_signo  = SIGUSR1;
      sigx.sigev_value.sival_int = TIMER_TAG;

      if ( timer_create( CLOCK_REALTIME, &sigx, &t_id ) == -1) {
         perror("timer_create");
         return 1;
      }

      /* Set timer to go off at April 20, 1996, 10:27am */
      do_time.tm_hour       = 10;
      do_time.tm_min        = 27;
      do_time.tm_sec        = 30;
      do_time.tm_mon        = 3;
      do_time.tm_year       = 96;
      do_time.tm_mday       = 20;

      val.it_value.tv_sec     = mktime( &do_time );
      val.it_value.tv_nsec    = 0;
      val.it_interval.tv_sec  = 15;
```

```
        val.it_interval.tv_nsec = 0;

        cerr << "timer will go off at: " << ctime(&val.it_value.tv_sec);

    if (timer_settime( t_id, TIMER_ABSTIME, &val, 0 ) == -1 ) {
        perror("timer_settime");
        return 2;
    }

    /* do something then wait for the timer to expire twice*/
    for (int i=0; i < 2; i++ )
        pause();

    if (timer_delete( t_id ) ==-1) {
        perror( "timer_delete" );
        return 3;
    }

    return 0;
}
```

The above program first sets up the *callme* function as the signal handler for the SIGUSR1 signal. It then creates a timer based on the system real-time clock. The program specifies that the timer should raise the SIGUSR1 signal whenever it expires, and the timer-specific data that should be sent along with the signal is TIMER_TAG. The timer handler returned by the *timer_create* API is stored in the *t_id* variable.

The next step is to set the timer to go off on April 20, 1996, at 10:27 AM and 30 seconds, and the timer should rerun for every 30 seconds thereafter. The absolute expiration data/time is specified in the *do_time* variable (of type *struct tm*) and is being converted to a *time_t*-type value via the *mktime* function. After these are all done, the *timer_settime* function is called to start the timer running. The program then waits for the timer to expire at the said date/time and expires again 30 seconds later. Finally, before the program terminates, it calls the *timer_delete* to free all system resources allocated for the timer.

The example output of the program is:

```
%   CC posix_timer_sbs.C  -o posix_timer_abs
%   posix_timer_abs
timer will go off at: Sat Apr 20 10:27:30 1996
callme: 12, signo: 16, Sat Apr 20 10:27:30 1996
callme: 12, signo: 16, Sat Apr 20 10:27:45 1996
```

Note that the above program can be modified to use a relative-time timer instead. For

example, to set the timer to go off 60 minutes from now and repeat every 120 seconds thereafter, the *main* function will be modified as in the following:

```
int main()
{
    /* set up sigaction for SIGUSR1 */
    ...

    /* Create a timer using timer_create */
    ...

    struct itimerspec      val;

    val.it_value.tv_sec    = 60;            /* expire 60 sec. from now */
    val.it_value.tv_nsec   = 0;
    val.it_interval.tv_sec  = 120;          /* repeat every 120 sec */
    val.it_interval.tv_nsec = 0;

    if (timer_settime( t_id, 0, &val, 0 ) == -1 ) {
        perror("timer_settime");
        return 2;
    }

    /* wait for timer expires */
    ...
}
```

The only differences in the modified *main* function from that in the *posix_timer_abc.C* are: (1) the *do_time* variable and the *mktime* API are not being used; (2) the *val.it_value* is set directly with the relative time (from the present) when the timer will first expire; and (3) the second argument to the *timer_settime* call is set to 0 instead of to TIMER_ABSTIME.

9.11 *timer* Class

Along with the improved flexibility and accuracy offered by the POSIX.1b timers, there is also considerably more code needed to create, use, and deallocate these timers. However, the APIs map nicely to a C++ timer class, and this class offers the following advantages to users:

- It provides a high-level interface for manipulation of timers. This reduces the time in learning how to use timer, and in the programming and debug effort
- It encapsulates the interface codes to the APIs. These codes are being reused when multiple timers are created
- The class member functions may be altered to use the *setitimer* API on systems that

are not POSIX.1b-compliant. This reduces the porting efforts of user applications

- The timer class can be incorporated into other user classes that require built-in timers for their operations. For example, a bank ATM software may cancel a transaction when no user inputs are detected for 5 minutes

The timer class functions map to the POSIX.1b timer APIs in the following manner:

Timer class function	POSIX.1b API
constructor	timer_create
destructor	timer_delete
start or stop timer	timer_settime
get overrun statistics	timer_getoverrun
query timer values	timer_gettime

In addition to the above, the timer class constructor also sets up signal handling for the timer (via the *sigaction* API). The timer class declaration is specified in the *timer.h* and depicted below:

```
#ifndef TIMER_H
#define TIMER_H

#include <signal.h>
#include <time.h>
#include <errno.h>
typedef void (*SIGFUNC)(int, siginfo*, void*);

class timer
{
        timer_t             timer_id;
        int                 status;
        struct itimerspec   val;
    public:
        /* constructor: setup a timer */
        timer( int signo, SIGFUNC action, int timer_tag,
                            clock_t sys_clock = CLOCK_REALTIME)
        {
          status = 0;

          struct sigaction sigv;;
          sigemptyset( &sigv.sa_mask );
          sigv.sa_flags = SA_SIGINFO;
          sigv.sa_sigaction = action;
```

```
    if (sigaction( signo, &sigv, 0 ) == -1) {
          perror("sigaction");
          status = errno;
    }
    else {
          struct sigevent sigx;
          sigx.sigev_notify  = SIGEV_SIGNAL;
          sigx.sigev_signo   = signo;
          sigx.sigev_value.sival_int = timer_tag;

          if (timer_create( sys_clock, &sigx, &timer_id ) == -1) {
                    perror("timer_create");
                    status = errno;
          }
    }
};

/* destructor: discard a timer */
~timer()
{
    if (status == 0)  {
          stop();
          if (timer_delete( timer_id ) == -1)
                perror( "timer_delete" );
    }
};

/* Check timer status */
int operator!()
{
    return status ? 1 : 0;
};

/* setup a relative time timer */
int run( long start_sec, long start_nsec, long reload_sec, long
        reload_nsec )
{
    if (status) return -1;
    val.it_value.tv_sec         = start_sec;
    val.it_value.tv_nsec        = start_nsec;
    val.it_interval.tv_sec      = reload_sec;
    val.it_interval.tv_nsec     = reload_nsec;
    if (timer_settime( timer_id, 0, &val, 0 ) == -1 ) {
          perror("timer_settime");
          status = errno;
          return -1;
```

```
        }
        return 0;
    };

    /* setup an absolute time timer */
    int run( time_t start_time, long reload_sec, long reload_nsec )
    {
        if (status) return -1;
        val.it_value.tv_sec          = start_time;
        val.it_value.tv_nsec         = 0·
        val.it_interval.tv_sec       = reload_sec;
        val.it_interval.tv_nsec      = reload_nsec;
        if (timer_settime( timer_id, TIMER_ABSTIME, &val, 0 ) == -1 ) {
                perror("timer_settime");
                status = errno;
                return -1;
        }
        return 0;
};

/* Stop a timer from running */
int stop()
{
    if (status) return -1;
    val.it_value.tv_sec      = 0;
    val.it_value.tv_nsec     = 0;
    val.it_interval.tv_sec   = 0;
    val.it_interval.tv_nsec  = 0;
    if (timer_settime( timer_id, 0, &val, 0 ) == -1 ) {
            perror("timer_settime");
            status = errno;
            return -1;
    }
    return 0;
};

/* Get timer overrun statistic */
int  overrun()
{
    if (status) return -1;
    return timer_getoverrun( timer_id );
};

/* Get timer remaining time to expiration */
int values( long& sec, long& nsec )
{
```

```
            if (status) return -1;
            if (timer_gettime( timer_id, &val ) == -1) {
               perror(" timer_gettime" );
               status = errno;
               return -1;
            }
            sec  = val.it_value.tv_sec;
            nsec = val.it_value.tv_nsec;
            return 0;
        };
        /* Overload << operator for timer objects */
        friend ostream& operator<<( ostream& os, timer& obj)
        {
            long sec, nsec;
            obj.values( sec, nsec );
            double tval = sec + ((double)nsec/1000000000.0);
            os << " time left: " << tval ;
            return os;
        };
};
#endif
```

In the above class, the *timer::timer* constructor takes as argument the signal number of a signal to be raised when the new timer expires, the signal handler for the timer, and a timer tag. The last argument, *sys_clock*, is optional. It specifies that the new timer be based on a certain system clock. The constructor internally sets up signal handling for the named signal and creates a new timer, based on the *sys_clock*, and uses the named signal and *timer_tag*.

The *timer::run* and *timer::stop* starts and stops a timer running, respectively. They internally set up the *struct itimerspec* data to call the *timer_settime* API. Specifically, the *timer::run* function is overloaded, so that users may specify relative or absolute time for the initial run of the timer. If an absolute time is specified, the *start_time* argument value may be obtained via the *mktime* function, as shown in the *posix_timer.C* program (see last section).

In addition to the above, the *timer::overrun* function returns the overrun statistic of a timer object, and the *timer::values* function returns the remaining time until the next expiration of the timer. Finally, the "<<" operator is overloaded to print the *timer::values* results of a timer object.

Overall, the *timer* class member functions provide an abstract view of a POSIX.1b timer. Their interfaces are all the basic data required to set up and manipulate a timer. All the low-level code that interfaces with the POSIX.1b APIs are encapsulated and readily reusable for multiple timer objects.

The following sample program, *posix_timer2.C*, illustrates the advantages and ease of use of the timer class:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "timer.h"

void callme( int signo, siginfo_t* evp, void* ucontext )
{
      long   sec, nsec;
      time_t tim = time(0);
      cerr  << "timer Id: " << evp->si_value.sival_int
            << ", signo: " << signo << ", " << ctime(&tim);
}
int main()
{
      timer t1 ( SIGINT,  callme,  1 );
      timer t2 ( SIGUSR1, callme,  2 );
      timer t3 ( SIGUSR2, callme,  3 );

      if (!t1 || !t2 || !t3 ) return 1;

      t1.run( 2, 0,  2, 0 );
      t2.run( 3, 500000000,  3, 500000000 );
      t3.run( 5, 0,  5, 0 );

      /* wait for timers to expire 10 times */
      for ( int i =0 ; i < 10; i++) {
         /* do some work and before timers expire */
         pause();

         /* show timers remaining time to expiration */
         cerr << "  t1: " << t1 << endl;
         cerr << "  t2: " << t2 << endl;
         cerr << "  t3: " << t3 << endl;
      }

      /* show timers overrun statistics */
      cerr << "t1 overrun: " << t1.overrun() << endl;
      cerr << "t2 overrun: " << t2.overrun() << endl;
      cerr << "t3 overrun: " << t3.overrun() << endl;

      return 0;
}
```

In the program, three timers are set up, such that the first timer expires every 2 seconds and raises the SIGINT signal. The second timer expires every 3.5 seconds and raises the SIGUSR1 signal. The third timer expires every 5 seconds and raises the SIGUSR2 signal. The signal handler for all the three signals is *callme* (users may use a different signal handling function per signal if they wish).

After the timer objects are created, the program goes into a loop and waits for 10 interruptions by any timer. For each interruption, it prints out (in the *callme* function) the timer that expires and the current date and time. Furthermore, it prints out (in the *main* function) the time remaining for all timer objects. After the 10 timer interruptions, the program prints out the timer object overrun statistics and then quits. The timer objects are destroyed implicitly via the *timer::~timer* function when the program terminates.

The sample output of the program is:

```
%    CC timer.C
%    a.out
timer Id: 1, signo: 2, Sat Apr 20 13:00:29 1996
      t1: time left: 1.99944
      t2: time left: 1.49698
      t3: time left: 2.99601
timer Id: 2, signo: 16, Sat Apr 20 13:00:31 1996
      t1: time left: 0.504464
      t2: time left: 3.50374
      t3: time left: 1.50304
timer Id: 1, signo: 2, Sat Apr 20 13:00:31 1996
      t1: time left: 2.0047
      t2: time left: 3.00398
      t3: time left: 1.00327
timer Id: 3, signo: 17, Sat Apr 20 13:00:32 1996
      t1: time left: 1.00468
      t2: time left: 2.00397
      t3: time left: 5.00326
timer Id: 1, signo: 2, Sat Apr 20 13:00:33 1996
      t1: time left: 2.0047
      t2: time left: 1.00398
      t3: time left: 4.00251
timer Id: 2, signo: 16, Sat Apr 20 13:00:34 1996
      t1: time left: 1.00467
      t2: time left: 3.50385
      t3: time left: 3.00313
timer Id: 1, signo: 2, Sat Apr 20 13:00:35 1996
      t1: time left: 2.0047
      t2: time left: 2.50399
      t3: time left: 2.00328
```

```
timer Id: 3, signo: 17, Sat Apr 20 13:00:37 1996
timer Id: 1, signo: 2, Sat Apr 20 13:00:37 1996
        t1:  time left: 2.00309
        t2:  time left: 0.502374
        t3:  time left: 5.00143
timer Id: 2, signo: 16, Sat Apr 20 13:00:38 1996
        t1:  time left: 1.50468
        t2:  time left: 3.50396
        t3:  time left: 4.50325
timer Id: 1, signo: 2, Sat Apr 20 13:00:39 1996
        t1:  time left: 2.00468
        t2:  time left: 2.00385
        t3:  time left: 3.00313
t1 overrun: 0
t2 overrun: 0
t3 overrun: 0
```

9.12 Summary

This chapter described the signal handling methods in UNIX and POSIX.1 systems and the various means where a process could generate signals to other processes or to itself. The primary use of signals is for process controls, such that users, kernel, or processes can interrupt runaway processes via signals.

Furthermore, signals may be used to implement some simple means of interprocess communication. For an example, two processes can install signal handlers for the SIGUSR1 signal and synchronize their execution by sending each other the SIGUSR1 signal. In the next chapter, more elaborate methods for interprocess communication in the UNIX and POSIX.1 systems will be depicted.

Finally, signals can also be used to support the implementation of interval timers. Interval timers are useful in setting up scheduled tasks to be performed by processes, when those processes require timing or limiting of execution times of certain operations by processes. This chapter described the UNIX and POSIX.1b methods for implementing interval timers. A timer class is depicted to facilitate the use of timers in user applications. The specific advantages of the timer class are that it provides a simplified interface to timer creation and manipulation, promotes code reuse, and reduces porting efforts. Moreover, the timer class can be incorporated easily into other user classes for adding functionality to those classes.

CHAPTER 10

Interprocess Communication

Interprocess communication (IPC) is a mechanism whereby two or more processes communicate with each other to perform tasks. These processes may interact in a client/server manner (that is, one or more "client" processes send data to a central server process and the server process responds to each client) or in a peer-to-peer fashion (that is, any process may exchange data with others). Examples of applications that use interprocess communication are database servers and their associated client programs (using the client/server model) and electronic mail systems (using the peer-to-peer model), where a mailer process communicates with other remote mailer processes to send and receive electronic mails over the Internet.

Interprocess communication is supported by all UNIX systems. However, different UNIX systems implement different methods for IPC. Specifically, BSD UNIX provides sockets for processes running on different machines to communicate. UNIX System V.3 and V.4 support messages, semaphores, and shared memory for processes running on the same machine to communicate, and they provide Transport Level Interface (TLI) for intermachine communication. Furthermore, UNIX System V.4 supports sockets to facilitate porting of socket-based applications on to their system. Finally, both BSD and UNIX System V support memory map as an intramachine communication method.

This chapter examines the message, shared memory, memory map, and semaphore IPC methods. The next chapter will describe the socket and Transport Level Interface IPC methods.

10.1 POSIX.1b IPC Methods

IPC methods are not defined in POSIX.1 but are defined in POSIX.1b (the standard for a portable real-time operating system). The IPC methods defined in POSIX.1b are messages, shared memory, and semaphores. Although these POSIX.1b IPC methods have the same names as those of UNIX System V, their syntax is totally different from that of System V. This is done intentionally, due to the following drawbacks of the System V methods:

- The System V messages, shared memory, and semaphores use integer keys as identifiers (names). This creates a different name space from that of files that an operating system needs to support
- The integer identifiers of messages, shared memory, and semaphores are not unique across machines. Thus, these IPC methods inherently cannot be used by network-based applications for intermachine communication
- The System V messages, shared memory, and semaphores are implemented in the kernel space. This means that every operation on these IPC objects requires a process to do context switches from user-mode to kernel-mode to be able to access data kept in these IPC objects. Process performance is taxed using these methods

To overcome the above drawbacks, the POSIX.1b messages, shared memory, and semaphores are implemented differently, as follows:

- The POSIX.1b messages, shared memory, and semaphores use file name-like identifiers (e.g., */psx4_message*), which means that IPC objects can be referenced like any file object and no separate name space requires support by a kernel
- By defining network-wide unique textual names, IPC objects may support intermachine communication. POSIX.1b does not specify the naming convention for such purposes
- POSIX.1b IPC methods do not mandate kernel-level supports; thus, vendors may implement these IPC methods using library functions. Furthermore, IPC objects are created and manipulated in the process address space. All these minimize kernel involvement and improve the efficiency of these methods

Note that not many commercial UNIX systems currently support these IPC methods yet, but they will in future operating system releases.

This chapter describes both the UNIX System V and POSIX.1b message, shared memory, and semaphore IPC methods.

10.2 The UNIX System V IPC Methods

The IPC methods supported by UNIX System V are:

- *Messages*: allow processes on the same machine to exchange formatted data
- *Semaphores*: provide a set of system-wide variables that can be modified and used by processes on the same machine to synchronize their execution. Semaphores are commonly used with a shared memory to control the access of data in each shared memory region
- *Shared memory*: allows multiple processes on the same machine to share a common region of virtual memory, such that data written to a shared memory can be directly read and modified by other processes
- *Transport Level Interface:* allows two processes on different machines to set up a direct, two-way communication channel. This method uses STREAMS as the underlying data transport interface

In addition to the above, System V.4 also supports BSD sockets. This is to facilitate socket-based applications ported to that system, with minimum modification.

10.3 UNIX System V Messages

Messages allow multiple processes on the same UNIX machine to communicate by sending and receiving messages among themselves. This is like setting up a central mail box in a building, allowing people to deposit and retrieve mail from the mail box. Furthermore, just as all mail has a recipient address, each message has an integer message type assigned to it by a sender process, so that a recipient process can selectively receive messages based on a message type.

Messages were invented to overcome some of the deficiencies of pipes (named and unnamed). One problem with pipes is that multiple processes can attach to the read and write ends of a pipe, but there are no mechanisms provided to promote selective communication between a reader and a writer process. For example, suppose there are processes, *A* and *B*, attached to the write end of a pipe and processes *C* and *D* attached to the read end of a pipe. If both *A* and *B* write data to the pipe, such that *A*'s data is read by *C* and *B*'s data is read by *D*, there is no easy way for *C* and *D* to selectively read data that are destined for them only. Another problem arises if data stored in a pipe is destroyed when both ends of the pipe have no process attached. Thus, pipes and their stored data are transient objects. They cannot be used by processes to exchange data reliably if they are not running in the same period of time.

Messages also allow multiple processes to access a central message queue. However, every process that deposits a message in the queue needs to specify an integer message type for the message. Thus, a recipient process can retrieve that message by specifying that same message type. Furthermore, messages stored in a message queue are persistent, even when

there is no process referencing the queue. Messages are removed from a queue only when processes explicitly retrieve them. Thus, messages are more flexible for multiprocesses communication.

When multiple message queues exist in a UNIX system, each can be used by a set of applications for interprocess communication.

10.3.1 UNIX Kernel Support for Messages

The implementation of message queues in UNIX System V.3 and V.4 is analogous to the implementation of UNIX files. Specifically, there is a message queue table in a kernel address space that keeps track of all message queues created in a system. Each entry of the message tables stores the following data for one message queue:

- A name, which is an integer ID key assigned by the process that created the queue. Other processes may specify this key to "open" the queue and gets a descriptor for future access of the queue
- The creator user ID and group ID. A process whose effective user ID matches a message queue creator user ID may delete the queue and also change the queue control data
- The assigned owner user ID and group ID. These are normally the same as those of the creator, but a creator can set these values to reassign the queue owner and group membership
- Read-write access permission of the queue for owner, group members, or others. A process that has read permission to the queue may retrieve messages from the queue and query the assigned user and group IDs of the queue. A process that has write permission to a queue may send messages to the queue
- The time and process ID of the last process that sent a message to the queue
- The time and process ID of the last process that read a message from the queue
- The pointer to a linked list of message records in the queue. Each message record stores one message of data and its assigned message type

Figure 10.1 depicts the kernel data structure for messages.

When a process sends a message to a queue, the kernel creates a new message record and puts it at the end of the message record linked list for the specified queue. The message record stores the message type, the number of bytes of the message data, and the pointer to another kernel data region, where the actual message data is stored. The kernel copies the message data from the sender process's virtual address into this kernel data region, so that the sender process is free to terminate, and the message can still be read by another process in the future.

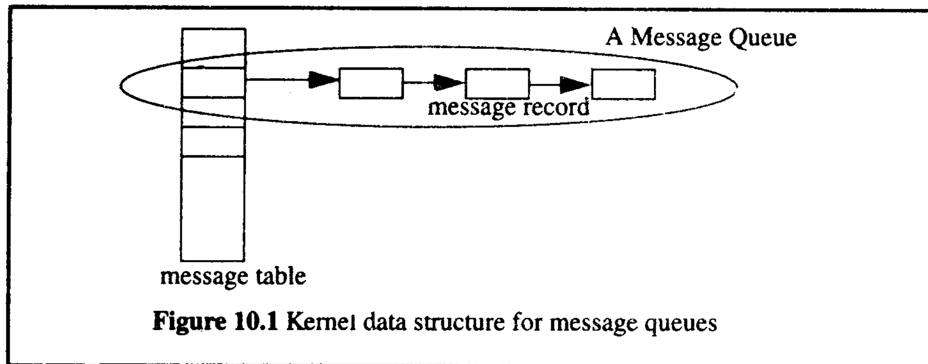


Figure 10.1 Kernel data structure for message queues

When a process retrieves a message from a queue, the kernel copies the message data from a message record to the process's virtual address and then discards the message record. The process can retrieve a message in a queue in the following manners:

- Retrieve the oldest message in the queue, regardless of its message type
- Retrieve a message whose message ID matches the one specified by the process. If there are multiple messages with the given message type existing in the queue, retrieve the oldest one among them
- Retrieve a message whose message type is the lowest among those that are less than or equal to the one specified by the process. If there are multiple messages that satisfy the same criteria, retrieve the oldest one among them

If a process attempts to read a message from a queue but no messages in the queue satisfy the retrieval criteria, then by default the process will be put to sleep by the kernel (until a message arrives in the queue that can be read by that process). However, the process can specify a nonblocking flag to the message receive system call that causes it to return a failure status instead of blocking the process.

Finally, there are several system-imposed limits on the manipulation of messages. These limits are defined in the <sys/msg.h> header:

System limit	**Meaning**
MSGMNI	The maximum number of message queues that may exist at any given time in a system
MSGMAX	The maximum number of bytes of data allowed for a message
MSGMNB	The maximum number of bytes of all messages allowed in a queue
MSGTQL	The maximum number of messages in all queues allowed in a system

The effects of these system-imposed limits on processes are:

- If the current number of message queues exist in the system is MSGMNI, any attempt to create a new message queue by a process will fail, until an existing queue is deleted by a process
- If a process attempts to send a message whose size is larger than MSGMAX, the system call will fail
- If a process attempts to send a message to a queue that will cause either the MSGMNB or MSGTQL limit to be exceeded, then the process will be blocked until one or more messages are retrieved from the queue and the message can be inserted into the queue without violating both the MSGMNB and MSGTQL limits

10.3.2 The UNIX APIs for Messages

The <sys/ipc.h> header defines a *struct ipc_perm* data type that stores the owner and creator user and group IDs, the assigned name key, and the read-write permission of a message queue. This data type is also used by the UNIX System V semaphore and shared memory IPC methods.

The message table entry data type is *struct msqid_ds*, that is defined in the <sys/message.h> header. The data fields of the structure and the corresponding data it stores are:

Data field	Data stored
msg_perm	Data stored in a *struct ipc_perm* record
msg_first	Pointer to the first (oldest) message in the queue
msg_last	Pointer to the last (newest) message in the queue
msg_cbyte	Total number of bytes of all messages currently in the queue
msg_qnum	Total number of messages currently in the queue
msg_qbyte	Maximum number of bytes of all messages allowed in the queue. This is normally MSGMNB, but it can be set to a lower value by either the creator or the assigned owner of the queue
msg_lspid	Process ID of last process that sent a message to the queue
msg_lrpid	Process ID of the last process that read a message from the queue
msg_stime	Time when the latest message was sent to the queue
msg_rtime	Time when the latest message was read from the queue

Data field	Data stored
msg_ctime	Time when the message queue control data (the access permission and owner user ID and group ID) was last modified

The *struct msg* as defined in the <sys/msg.h> header is the data type of a message record. The data fields and the corresponding data stored are:

Data field	Data stored
msg_type	Assigned integer message type
msg_ts	Number of bytes of the message text
msg_spot	Pointer to the message text that is stored in a different kernel data region
msg_next	Pointer to the next message record, or NULL if this is the last record in a message queue

Figure 10.2 illustrates the uses of the aforementioned structures in the message table and message records.

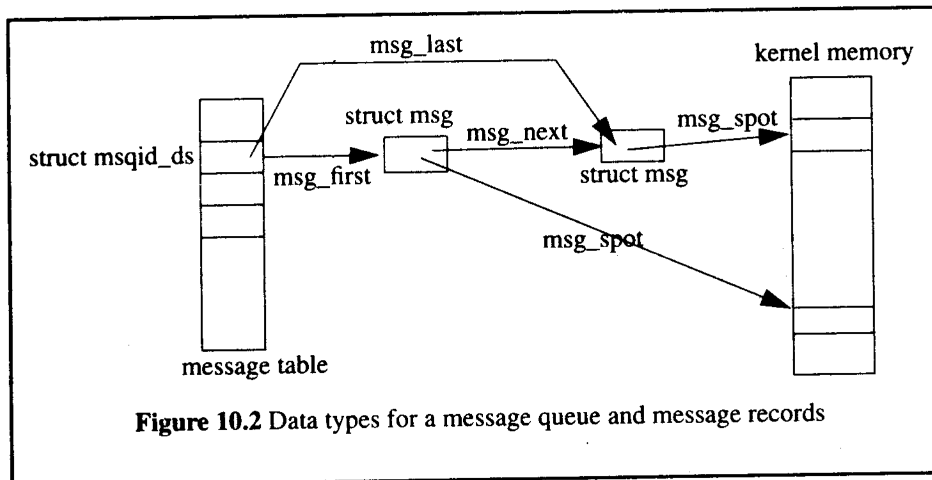


Figure 10.2 Data types for a message queue and message records

There are four APIs for message manipulation:

Messages API	Uses
msgget	Open and create if needed, a message queue for access
msgsnd	Send a message to a message queue
msgrcv	Receive a message from a message queue
msgctl	Manipulate the control data of a message queue

As stated earlier, message implementation is analogous to that of UNIX files, thus, the analogous APIs between messages and files are:

Messages API	Files API
msgget	open
msgsnd	write
msgrcv	read
msgctl	stat, unlink, chmod, chown

The header files needed for the messages APIs are:

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

10.3.3 msgget

The function prototype of the *msgget* API is:

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

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

This function "opens" a message queue whose key ID matches the *key* actual value and returns a positive integer descriptor. This can be used in other message APIs to send and receive messages and to query and/or set control data for the queue.

If the value of the *key* argument is a positive integer, the API attempts to open a message queue whose key ID matches that value. However, if the *key* value is the manifested constant IPC_PRIVATE, the API allocates a new message queue to be used exclusively by the calling process.

If the *flag* argument is 0, the API aborts if there is no message queue whose key ID matches the given *key* value; otherwise, it returns a descriptor for that queue. If a process wishes to create a new queue (if none exists) with the given *key* ID, the *flag* value should contain the manifested constant IPC_CREAT and the read-write access permission (for owner, group and others) for the new queue.

For example, the following system call creates a message queue with the key ID of 15 and access permission of 0644 (that is, read-write for owner and read-only for group members and others), if such a queue does not preexist. The call also returns an integer descriptor for future queue references:

```
int   msgfdesc = msgget ( 15, IPC_CREAT|0644 );
```

If a process wishes to guarantee the creation of a new message queue, it can specify the IPC_EXCL flag with the IPC_CREAT flag, and the API will succeed only if it creates a new queue with the given *key*.

The API returns -1 it if fails. Some possible causes of failure are:

***Errno* value**	**Cause of error**
ENOSPC	The system-imposed limit MSGMNI has been reached
ENOENT	The *flag* value does not contain the IPC_CREAT flag, and no queue exists with the specified *key*
EEXIST	The IPC_EXCL and IPC_CREAT flags are set in the *flag* value, and a message queue with the specified *key* already exists
EACCESS	The queue with the specified *key* exists, but the calling process has no access permission to the queue

10.3.4 msgsnd

The function prototype of the *msgsnd* API is:

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

int msgsnd ( int msgfd, const void* msgPtr, int len, int flag );
```

This API sends a message (pointed to by *msgPt)* to a message queue designated by the *msgfd* descriptor.

The *msgfd* value is obtained from a *msgget* function call.

The actual value of the *msgPtr* argument is the pointer of an object that contains the actual message text and a message type to be sent. The following data type can be used to define an object for such purpose:

```
struct msgbuf
{
      long  mtype;              // message type
      char  mtext[MSGMAX];      // buffer to hold the message text
};
```

The *len* value is the size, in bytes, of the *mtext* field of the object pointed to by the *msgPtr* argument.

The *flag* value may be 0, which means the process can be blocked, if needed, until the function call completes successfully. If it is the IPC_NOWAIT flag, the function aborts if the process is to be blocked.

The return value of the API is 0 if it succeeds or -1 if it fails.

The following *test_msgsnd.C* program creates a new message queue with the key ID of 100 and sets the access permission of the queue to be read-write for owner, read-only for group members and write-only for others. If the *msgget* call succeeds, the process sends the message *Hello* of type 15 to the queue and specifies the call to be nonblocking. If either the *msgget* or the *msgsnd* call fails, the process calls *perror* to print out a diagnostic message.

```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msgbuf
{
     long    mtype;
     char    mtext[MSGMAX];
} mobj = { 15, "Hello" };

int main()
{
   int fd = msgget (100, IPC_CREAT|IPC_EXCL|0642);
   if (fd==-1 || msgsnd(fd,&mobj,strlen(mobj.mtext)+1, IPC_NOWAIT))
      perror("message");
```