Advanced UNIX Programming: lecture notes

Technical Report of the ISIS Group
at the University of Notre Dame
ISIS-97-006
August , 1997

Michael Lemmon
Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556

Interdisciplinary Studies of Intelligent Systems

Advanced UNIX Programming: lecture notes

Michael Lemmon *
Complex Systems Simulation Facility
Dept. of Electrical Engineering
University of Notre Dame

October 11, 1997

These are lecture notes an advanced UNIX programming taught in the summer of 1997 ath the University of Notre Dame's department of electrical engineering. These lectures were compiled to help prepare students for working in the complex simulation research facility. A list of the references used to prepare these lectures include

- Kernighan and Pike, The UNIX Programming Environment, Prentice-Hall, 1984
- Kernighan and Ritchie, The C Programming Language, Prentice Hall, 1988 2nd edition
- W.R. Stevens, UNIX Network Programming, Prentice-Hall, 1990
- W. R. Stevens, Advanced Programming in the UNIX Environment, Addison-Wesley, 1992
- B. Gallmeister, Programming for the Real World: POSIX.4, O'Reilly and Associates Inc. , 1995
- K. Robbins and S. Robbins, Practical UNIX Programming: a guide to concurrency, communication, and multithreading, Prentice-Hall, 1996

1 Programs and Processes

A process is an *instance* of a program. Several instances of the same program may run on the same computer. An *operating system* coordinates the access of these processes to the limited resources (CPU, memory, I/O) of the computer. In these lectures we will be interested in writing "C" programs which make UNIX system calls. The operating system, UNIX, is a pre-emptive multi-tasking operating system. The *kernel* of the operating system is written in the "C" programming language. It consists of a set of *kernel* processes which have special permissions. All other processes are run at a permission level which is more restrictive than the kernel processes. Figure 1 illustrates the relationship of the kernel processes to other user processes.

*The authors gratefully acknowledge the partial finanical support of the Army Research Office (DAAH04-96-1-0285)

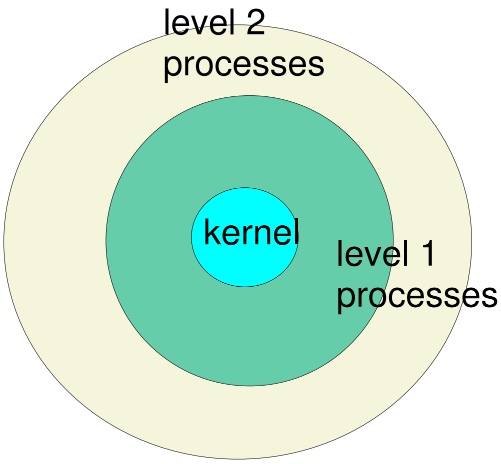


Figure 1: Kernel and other user processes

1.1 Error Handling

Error handling is extremely important in advanced UNIX programming. System calls can fail for a variety of reasons which cannot be forseen by the programmer. For example, a system call opening a file may fail if the file is not present or if access to the file has been locked by another program. There are, basically, two different ways of including error processing. The first way is illustrated by the code segment shown below.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>

int fd;

if ((fd = open(``my.file'', O_RDONLY))==-1)
perror(``unsuccessful open of my.file\n'');
```

In this particular program segment, the `open` system state return -1 when it fails. The failure of the call is tested and then routine `perror` writes out an error message along with the error message generated by the system call which caused `open` to fail.

A more sophisticated form of error processing is seen in the following code segment.

```
#include<string.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
```

```
#include<errno.h>

int fd;

if (( fd = open(argv[1], O_RDONLY))==-1)
fprintf(stderr, ``could not open file %s: %s\n),
argv[1], strerror(errno));
```

In this particular example, the failure of the system call writes to a global `errno` variable reprsenting an error number. The meaning of this error number is defined (via a macro) in `errno.h`. The system call `strerror(errno)` then returns a string with this defined constant indicating the precise nature of the error.

1.2 Command Line Processing

A UNIX command line consists of tokens (arguments) which are separated by blanks, tabs, or backslashes just before a newline `\n` character. Each token, itself, is a string of characters. When a user enters a command corresponding to a "C" executable program, the shell parses the command line and passes the result to the program in the form of an argument array. An argument array with declaration

```
char **argv
```

is a pointer to an array of strings. Recall that each string is a pointer to an array of 1byte characters. Consider, for example, the command line

```
ls -l
```

This command line has two tokens, `ls` and `-l`, with a blank space acting as the the delimiter. The `main` body of a "C" program reads this command line through the declaration

```
void main(int argc, char **argv)
```

where `argc` is an integer holding the number of tokens on the command line and `argv` is the argument array pointing to each token. As a matter of reviewing "C"-programming it is useful to review how one might write a program which transforms a command line into an argument array. The resulting program will be called `makeargv()`.

1.3 Example: `makeargv`

We design a function by first specifying its prototype, then characterizing what the variables are in the prototype declaration and how they should look upon calling and upon return from the program. Finally we discuss the program's implementation.

The pointer returned by the makeargv function is shown in figure 2. This figure shows the array of pointers in myargv which point to the first character in each token of a command line.

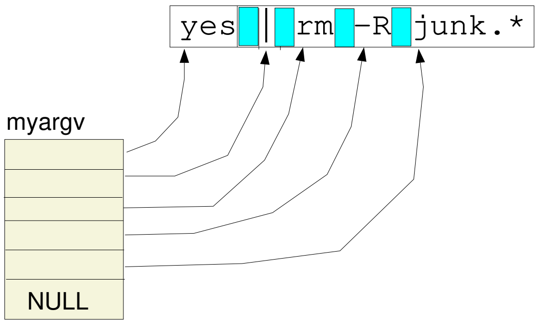


Figure 2: Data Structures in makeargv

1.4 Prototype makeargv

The prototype for our function can take the form

```
char **makeargv(char *s)
```

The string s is the string to be parsed and upon successful return the function gives a pointer to the variable array. If the function aborts on return, then the function returns a NULL pointer. A simple test program illustrating the use of makeargv is shown below.

```
int i;
char **myargv;
char mytest[]=''This is a test'';

if((myargv = makeargv(mytest))==NULL)
fprintf(stderr, ``could not construct an argument array\n'');
else
for(i=0; myargv[i] != NULL; i++)
printf(``%i, %s\n'', i, myargv[i]);
```

Sometimes it is better to use a slightly more sophisticated function prototype. The following prototype,

```
int makeargv(char *s, char *delims, char ***argvp)
```

returns an integer which can be used to indicate the error status of the routine if it fails. The function also allows us to modify the type of delimiters used between tokens.

The following program shows how this new prototype might be used. This program includes a specific "useage" section.

```
#include<stdio.h>
#include<stdlib.h>
int makeargv(char *s, char *delimiters, char ***argvp);

void main(int argc, char **argv)
{
    char **myargv;
    char delim[]=" \t";
    int i;
    int numtokens;

    if(argc != 2){
        fprintf(stderr, "useage: %s string\n", argv[0]);
        exit(1);
    }
    if((numtokens=makeargv(argv[1], delim, &myargv))<0){
        fprintf(stderr,
            "could not construct argument array for %s\n",argv[1]);
        exit(1);
    }else{
        printf("the argument aray contains: \n");
        for(i=0;i<numtokens;i++)
            printf("[%d]:%s\n",i,myargv[i]);
    }
    exit(0);
}
```

1.5 Implementation of `makeargv`

We now turn to look at the implementation of `makeargv`. This particular implementation will be very useful later on. In this particular implementation we make use of the C library routine

```
#include<stdio.h>

char *strtok(char *s, char *ct);
```

A sequence of calls to `strtok` can be used to split `s` into tokens delimited by characters in `ct`. This is done as follows,

1. The first call to `strtok(s,ct)` finds the first token in `s`, (the first continguous string of characters in `s` that have no characters form `ct` in it) and returns a pointer to that token. It then overwrites the next character (delimiter) with `'\0'` to mark the end of the token.
2. Subsequent calls with `s = NULL` return the next token in the original `s`.
3. `strtok` returns NULL when the end is reached.

Note that `strtok` is not a *safe* function in a concurrent environement. This is because `strtok` is not re-entrant. One call to `strtok` can modify the behaviour of future calls, because the function keeps track of its position between successive calls. This can create problems in a concurrent environment when an asynchronous signal generates an interrupt that context switches to a signal handler. If the signal is caught between two successive calls to `strtok`, the function may forget which version of `s` it is suppose to parse. In future POSIX standards, it is required that functions such as `strtok` be made *thread-safe* or *asynchronous-safe*.

The `makeargv` implemenation shown below uses the following strategy

- Allocate space for the tokens instead of modifying `s` itself. Rather than doing a `calloc` for each toekn, allocate a new string `t` which is the same size as the original string `s`.
- make a pss through string `t` using `strtok` t count the tokens.
- Use the token count (`numtokens`) to allocate an `argv` array.
- Use `strtok` to obtain pointers to the individual tokens, modifying `t` in the process.

```
#include<string.h>
#include<stdlib.h>

/*
 * make argv array (*arvp) for tokens in s which are separated by
 * delimiters.  return -1 on error or the number of tokens otherwise.
 */

int makeargv(char *s, char *delimiters, char ***argvp)
{
    char *t;
    char *snew;
    int numtokens;
    int i;
    /*snew is real start of string after skipping leading delimiters*/
    snew=s+strspn(s, delimiters);
        /*create space for a copy of snew in t*/
    if ((t = calloc(strlen(snew)+1, sizeof(char)) == NULL){
        *argvp = NULL;
        numtokens=-1;
    }else{      /*count the number of tokens in snew*/
        strcpy(t,snew);
        if(strtok(t,delimiters) == NULL)
            numtokens=0;
        else
            for(numtokens=1; strtok(NULL,delimiters) != NULL;
            numtokens++)
            ;
                /*create an argument array to contain ptrs to tokens */
```

```
        if((*argvp=calloc(numtokens+1,sizeof(char *)))==NULL){
            free(t);
            numtokens=-1;
        }else{          /*insert pointers to tokens into the array*/
            if (numtokens > 0){
                strkcpy(t,snew);
                **argvp = strtok(t,delimiters);
                for(i=1;i<numtokens+1;i++)
                    *((*argvp)+i) = strtok(NULL,delimiters);
            }else{
                **argvp = NULL;
                free(t);
            }
        }
    }
    return numtokens;
}
```

1.6 Exercises

1. Test out the `makeargv` as discussed above.
2. Read the man page for `strtok` to determine if there is a thread-safe version of the function. Rewrite `makeargv` using the thread-safe version of the function.
3. Write a function `freeargv` which has the prototype

   ```
   void freeargv(char ***argvp);
   ```

   This function frees the argument array pointed to by `*argvp` and set `*argvp` to `NULL`.

2 Process Creation

Recall that a process is an *instance* of a program which is characterized by its Process ID and Process State. A process can request the creation of a new process. The requesting process is called the `parent` and the created process is called the `child` process. In this section, we discuss the creation and elementary management of processes through `fork`, `wait`, and `exec` functions.

2.1 Process ID

The process ID is a (long) positive integer uniquely identifying the process to the operating system. It is possible to determine the process ID of a parent and its child through the following system calls.

```
#include<sys/type.h>
#inlcude<unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

UNIX associates a *user* or *owner* with each process. The user or owner has certain priviledges with regard to process management. There is a *user ID* which uniquely identifies the user associated with the process. There is also an *effective user ID* which identifies the priviledge of that process in accessing resources such as files, shared memory, and semaphores. The user ID and effective user ID's can be accessed through the following system calls.

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

uid_t getuid(void);
uid_t geteuid(void);
```

2.2 Process State

The process is run in one of a set of discrete states. The following table identifies a fundamental list of process states.

State	Meaning
`new`	process being created
`running`	process is being executed
`blocked`	process waiting for an event such as I/O
`ready`	process is waiting to be assigned to a processor
`done`	process has finished and its resources are deallocated

The operating system manipulates the process state in order to coordinate the operation of multiple processes on a single processor. A standard state diagram is shown in figure 3. In this case, we see that a process starts in the `new` state and after creation is placed in the `ready` state. At this point, the process is waiting for a *scheduler* to take the waiting process of the ready list and begin executing it. In many operating systems, such as Unix, the scheduler is also a process which decides at specified scheduling instant which process is to be executed during the next scheduling quantum or interval. At the point when the ready process is selected to run, it switches to the run state. Switching to the run state takes all of the information needed to restart the currently running process (i.e. its executable, stack, registers, program counters....) saves it and then loads the new process. The current program state is called the *context* and the act of switching to a new process is called a *context switch*. After the process begins running there are one of three operations that can be done. The operating system may decide for another context switch and thereby swap another processor into the CPU, the process may abort in a normal or abnormal manner, or the process may be blocked while it waits for an I/O request. Most operating systems switch the process state to `blocked` when a process is waiting on an I/O request and another process is switched to. Blocking occurs because I/O requests can take a very long time. After the I/O request is completed, the process goes back to the ready queue.

In the Solaris operating system, the process the following state table is used

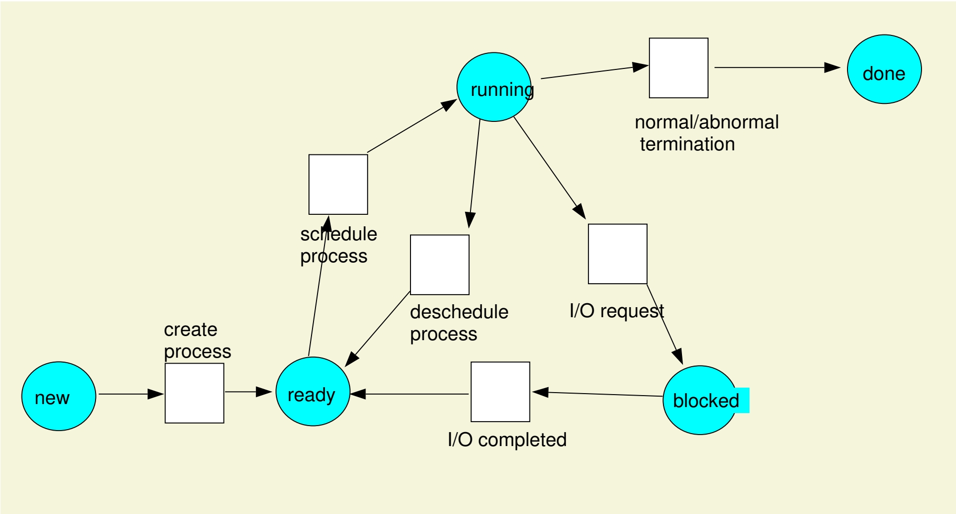


Figure 3: State Diagram for Process State Transition

state	meaning	related meaning
O	Process is running on a processor.	running
S	Sleeping: process is waiting for an event to complete.	blocked
R	Runnable: process is on run queue.	ready
Z	Zombie state: process terminated and parent not waiting.	done
T	Process is stopped, either by a job control signal or because it is being traced.	done

Note that this set of states is closely related to our earlier version. There is a notable exception, namely, that there are two different types of `done` states. The `zombie` state will be discussed in greater detail below.

2.3 Process Creation: `fork()`

A process can create a new process through a call to the system function `fork`. The creating process is called the *parent*. The new process is called the *child*. The basic UNIX system calls are

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

pid_t fork(void);
```

A call to the fork function returns 0 if the process making the call is the child. The return is a positive integer with the child's ID if the call is made by the parent. The following code segment provides a simple example illustrating the use of the `fork` function.

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

main(int argc, char **argv)
{
pid_t childpid;

if (( childpid = fork()) == 0 ){
printf(``I am the child, ID = %ld'', (long) getpid());

/* CHILD CODE GOES HERE*/

} else if (childpid > 0) {
printf(``I am the parent, ID = %ld'', (long) getppid());

/* PARENT CODE GOES HERE*/

}
}
```

The fork creates a new process by making a copy of the parent process' image and copying it into the child's address space. The child inherits some of the parent's attributes, but not all of them. For instance, the child gets a new process ID and the CPU useage is reset to zero. The child does not inherit any locks which may have been set by the adult, so the child is not notified when an alarm set by the parent expires. This basically means that the child starts with no pending signals. Other than these attributes, the child is a duplicate of the parent process and will compete as an equal with the parent for system resources.

Figure 4 illustrates two ways in which forks can be used. The first way construct a chain of processes. The second way constructs a fan of processes.

2.4 Waiting: `wait()`

What happens to the parent after the child is created, both processes proceed from the fork command. There may be times, of course, when we want the parent to wait for the child process to complete its execution before continuing. This waiting is accomplished using the `wait` or `waitpid` functins.

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

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

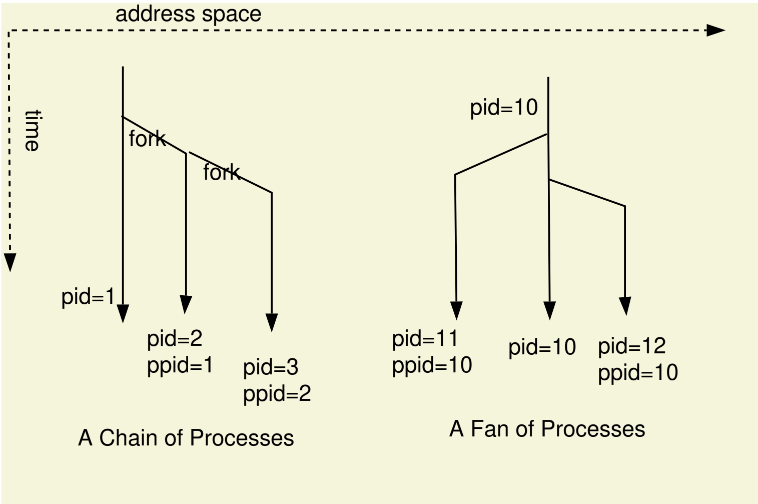


Figure 4: Process Chains and Fans

The `wait` and `waitpid` functions return -1 and sets `errno` if there is an abnormal return from the wait function. In particular if `errno=ECHILD`, then there is no child process to wait for and if `errno=EINTR`, then the call was interrupted by a signal (we'll cover signals later). The other argument in the `wait` command is an integer pointer `stat-loc`. This pointer can be passed to a macro to determine the status of the returning child process. The status is passed back to the parent when the child executes a `return` or `exit` function. The following program shows how `stat-loc` can be used to determine the status of the returning child.

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

main(int argc, char *argv){




    pid_t childpid;
   int status;

    if ((childpid = fork()) == 0) {
        printf("I am the child, ID = %ld", (long) getpid());
        /*child code goes here*/
    }else if (childpid > 0){
        printf("I am the parent, ID= %ld", (long) getppid());
```

```
        /*parent code goes here*/

        while ((( childpid= wait(&status))==-1) && (errno == EINTR));
        if(child==-1)
            perror("could not wait for child\n");
        else if (!status)
            printf("child %ld terminated normally", return status is zero\n",
                    (long) childpid);
        else if (WIFEXITED(status))
            printf("child %ld terminated normally, return status is %d\n",
                    (long) childpid, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf("child %ld terminated due to signal not caught\n",
                    (long) childpid);

    }
}
```

The `waitpid` system call provides a more flexible way of waiting for children processes by specifying which child process we are to wait for. It takes three parameters: a `pid`, a pointer to a location for returning status, and option flags. If `pid` is -1, `waitpid` waits for any process. If `pid` is positive, then *waitpid* waits for the specific child whose process ID is `pid`. The `WNOHANG` option causes `waitpid` to return even ef the status of a child is not immediately available.

A terminating child process whose parent does not wait for it is often called a *zombie*. Essentially, this means that the operating system maintains the exit status (state) of the child process, but since the child process is really doing nothing it is a "zombie". Zombies stay in the system until they are waited for. If the parent process terminates without waiting for a child, the child becomes an *orphan* and is adopted by the system *init* process. This kernel process periodically waits for orphans so they can be removed from the system.

3 Executing New Code :`exec`

The `fork` creates a child which executes another copy of the parent. Often, however, we need to execute different code. This is done by using an `exec` function to overlay the parent code inherited by the child with a new executable. There are six variations of the *exec* system call that are distinguished by the way the command line arguments and environment are passed and by whether a full pathname has to be given for the executable. The `execl` calls (`exec`, `execlp`, and `execle`) pass the command-line arguments as a list and are useful if the number of command line arguments are known at compile time. The `execv` calls (`execv`, `execvp`, and `execve`) pass the command-line arguments as an argument array. The six variations on the exec system call are shown below

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

```
int execl(const char *path, const char *arg0, ...,
          const char *argn, char * /*NULL*/);
int execle (const char *path, const char *arg0[], ... ,
           const char *argn, char * /*NULL*/,
           char *const envp[]);
int execlp(const char *file, const char *arg0, ... ,
          const char *argn, char * /*NULL*/);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve (const char *path, char *const argv[],
            char *const envp[]);
```

The following program illustrates the use of the `exec` system call in executing a command passed as command-line arguments. This program uses `execvp` to execute a command passed on the command line. The original process forks a child and then waits for the child to terminate. If the wait is interrupted by a signal, `errno` is equal to `EINTR` and the parent restarts its wait. The child process calls `execvp` with the argument array formed by the command-line arguments for the original program.

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

void main(int argc, char **argv){

    pid_t childpid;
    int status;

    if((childpid= fork()) == -1) {
        perror("fork failed\n");
        exit(1);
    } else if (childpid == 0) { /*child code*/
        if (execvp(argv[1], &argv[1]) < 0) {
            perror("the exec of command failed \n");
            exit(1);
        }
    } else                      /*parent code*/
        while(childpid != wait(&status))
            if((childpid == -1) && (errno != EINTR))   break;

    exit(0);
}
```

The program shown below calls `makeargv` to create an argument array from the string passed as its first command-line argument. It then does an `execvp` of the comand represented by that string. Pass a string containing multiple token by placing the string within double quotes (i.e. `myexec "ls -l"`)

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

int makeargv(char *s, char *delimiters, char ***argvp);

void main(int argc, char *argv)
{
    char **myargv;
    char delim[]=" \t";
    pid_t childpid;
    int status;

    if(argc != 2) {
        fprintf(stderr, "useage: %s string\n", argv[0]);
        exit(1);
    }
    if ((childpid=fork( ))==-1) {       /*child code*/
        perror("the fork failed\n");
        exit(1);
    }else if (childpid == 0) {
        if(makeargv(argv[1],delim, &myargv) < 0) {
            fprintf(stderr,"argument array cound not be constructed\n");
            exit(1);
        } else if (execvp(myargv[0], &myargv[0]) < 0) {
            perror("the exec of command failed\n");
            exit(1);
        }
    }else                               /*parent code*/
        while(childpid != wait(&status))
            if((childpid == -1) && (errno != EINTR))
                break;
    exit(0);
}
```

The exec copies a new executable into the process image. Exactly what is preserved from the original process is not obvious. The program text, variables, stack, and heap are overwritten. The new process inherits the environent, unless the original process caled execle or execve. Files that are open before the exec are usually still open afterwards. There can be varying effects on locks and alarms (which we'll discuss later). The following table summarizes the attributes inhered by the exec'd process. The first column gives the attribute and the second column gives the associated system call. The ID's associated with the process are intact after exec. If a process sets an alarm before calling exec then the alarm still generates a signal when it expires. PEnding signals are also carried over with an exec.

Attribute	Relevant System Call
process ID	getpid()
parent process ID	getppid()
process group ID	get pgid()
session membership	getsid()
real user ID	getuid()
real group ID	getgid()
time left on an alarm signal	alarm()
current working directory	getcwd()
root directory	
file mode creation mask	umask()
process signal mask	sigprocmask()
pending signals	sigpending()
time used so far	times()

3.1 Exercises

This section contains two exercises, one of process chains and one on process fans.

3.1.1 Process Chains

The following code creates a chain of n processes

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

int i;
int n;
pid_t childpid;
for(i=1; i<n; ++i)
    if (childpid = fork())
        break;
printf("This is process %ld with parent %ld\n",
    (long) getpid(), (long)getppid());
```

In this exercise you are to write a program that creates a chain of n processes. The program should take a single command-line argument that specifies the number of processes to create. Before exiting, each process otuputs its i value (i.e. which process in the chain), its process ID, its parent process ID, and its child's process ID. There will be no `wait`, so a parent process may exit before its child, thereby causing the child process to be adopted by the `init` process. In this case, some of the child process will have a parent process ID of 1. A sample version of this program is shown below.

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

void main(int argc, char **argv){

int i;
pid_t childpid;
int n;

if(argc!=2){
    fprintf(stderr, "Useage: %s processes \n", argv[0]);
    exit(1);
}
n=atoi(argv[1]);

childpid=0;
for(i=1; i<n; ++i)
    if(childpid=fork())
        break;
    if(childpid==-1){
        perror("the fork failed\n");
        exit(1);
    }

    fprintf(stderr
        "i:%d process ID: %ld parent ID:%ld child ID:%ld\n",
        i, (long)getpid(), (long)getppid(), (long)childpid);
    exit(0);


}
```

Do the following exercises with this program.

- Run the program and observe the results for different nmbers of processes.
- Find the largest number of processes that the program can generate.
- Place a `sleep(10)` directly before the final `fprintf` statement. Now what is the maximum number of processes generated.
- Put a loop around the final `fprintf` command. Have the loop execute k times. BUt a `sleep(m)` inside this loop. Pass k and m on the command line. RUn the program for several values of k and m and observe the results.
- Modify the program by putting `wait` system call before the final `fprintf` statement. How does this affect the system output.

3.1.2 Process Fans

The following code segment creates a fan of processes

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

int i;
int n;
pid_t childpid;

for(i=1;i<n;++i)
    if((childpid=fork())<=0)
        break;
    fprintf(stderr, "this is process %ld with parent %ld\n",
            (long)getpid(), (long)getppid());
```

You are now going to modify this code to develop a simple batch processing facility. The batch processing executable is called `runsim`. Write a test proram called `testsim` to test the facility.

The `runsim` program runs up to `prlimit` processes at at time. The program takes exactly one command-line argument specifying the maximum number of simultaneous executions. Follow the outline below for implementing `runsim`. Suggested library functions and system calls appear in parantheses.

- Check for the appropriate command-line argument and output a useage meesage if the command line is incorrect.
- Initialize `prlimit` from the command line. The `prlimit`variable specifies the maximum number of children to run at a time.
- The `prcount` variable holds the number of active children. Initialize it to zero.

- Execute the following main loop until the end-of-file is reach on standard input:
  - If `prcount` is `prlimit`, wait for a child to finish (`wait`) and decrement `prcount`.
  - Read a line from standard input (`fgets`) of up to `MAXCANON` characters and execute a program corresponding to that command line by forking a child (`fork`, `makeargv`, `execvp`).
  - Increment `prcount` t track the number of active children.
  - Check to see if any of the children have finished (`waitpid` with the `WNOHANG` option). Decrement `prcount` for each child that has completed.
- After encountering an EOF on standard input, wait for all of the remaining children to finish (`wait`) and then exit.

Now write a test program `testsim` that takes two command-line arguments: the sleep time and the repeate factor. The `testsim` program loops repeat factor times. In the loop, `testsim` sleeps for the specified sleep time and then outputs a message with its process ID to standard error. Use `runsim` to run multiple coppies of the `testsim` program.

4 UNIX File I/O

This lecture discusses the UNIX filesystem and methods for accessing the file system through a C program. This lecture also discusses methods for redirecting the input and output of processes through pipes and filters.

4.1 UNIX File System

Disk formatting divides the physical disk into a number of **partitions**. Each partition has a **file system** associated with it. A file system consists of a directory tree. A typical directory tree found in a UNIX file system is shown in figure 5.

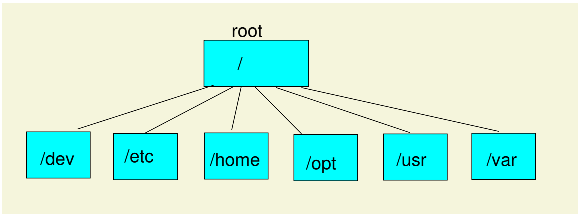


Figure 5: Structuree of a typical UNIX filesystem

There are some standard diretories that appear in the UNIX filesystem. The `dev` directory consists of special device files (to be discussed later), `etc` a directory consisting of files that are required for the specific system, `home` a collection of user directories, `opt` application files, `usr`, files that can be shared by all users, `var` contains files of varying size that such as mail (before it's read) and print spool files.

4.2 UNIX File Representation

A UNIX file has a description which is stored in a stucture called an **inode**. The inode contains information about file size, location, owner, time of creation, time of last access, etc. Most user files are **ordinary files**. Directories are also represented as files and have an associated **inode**. Devices are represented by **special files**. Character special files are used to represent devices such as terminals while block special files are used for disk devices. The FIFO special files are used for interprocess communication.

Figure 6 shows the inode structure of a typical file. In addition to descriptive information, the inode contains pinters to the first few data blocks of the file. If the file is large, the indirect pointer contains a pointer to a block of pointers to the file fragments.

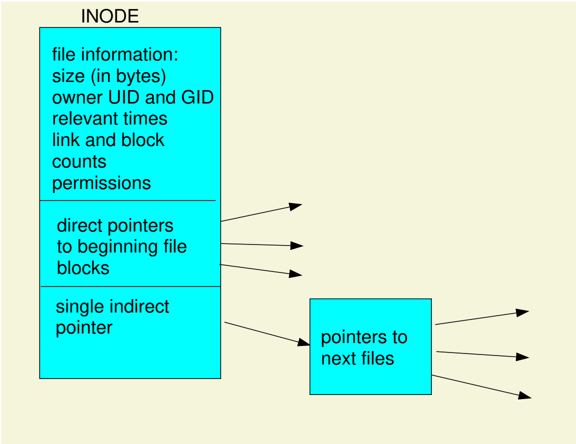


Figure 6: schematic structure of an Inode

Retrieving information contained in the inode of a file uses the `stat` system call

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

int stat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

The structure `stat` that is returned by the function call contains the following information

```
mode_t st_mode; /*file mode */
ino_t st_ino; /*inode number  */
dev_t st_dev; /*ID of device containing a*/
/*directory entry for this file*/
dev_t  st_rdev; /*ID of device*/
/*This entry is defined only for */
```

```
/*char special or block special files*/
nlink_t st_nlink; /*number of links*/
uid_t st_uid; /*user ID of the file's owner*/
gid_t st_gid; /*group ID of the file's group*/
off_t st_size; /*file size in bytes*/
time_t st_atime; /*last access time*/
time_t st_mtime; /*last modificaiton time*/
time_t st_ctime; /*time of last file status change*/
long st_blksize; /*preferred I/O block size*/
long st_blocks; /*number of st_blksize blocks allocated*/
```

As an example, of the use of `stat`, the following program determines whether a file name passed on the command line is a directory. If the filename is a directory, the program outputs the time at whichthe directory was last modified.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<time.h>
#include<sys/types.h>
#include<sys/stat.h>

void main(int argc, char **argv){
struct stat statbuf;

if(argc!=2){
fprintf(stderr,''useage: %s filename\n'', argv[0]);
exit(1);
}
if (stat(argv[1], &statbuf) == -1){
  fprintf(stderr, ``could not get stat of file %s: %s \n'',
argv[1], strerror(errno));
exit(1);
}
if(statbuf.st_mode & S_IFDIR) {
printf(``%s is a directory: ``, argv[1]);
printf(``last modified at %s\n'', ctime(&statbuf.st_mtime));
}else{
printf(``%s is not a directory\n'',argv[1]);
}
exit(0);
}
```

4.3 File Handler Representation

Files are designated within C programs by file pointers or file descriptors. The standard I/O library for ANSI C (fopen, fscanf, fprintf, fread, fwrite, fclose)) use file pointers, while the UNIX I/O library (open, read, write, close, ioctl use file descriptors. These pointers or descriptors provide logical names or **handles** for performing device independent input and output. The file pointer handles for standard input, output, and error are stdin, stdout, and stderr and are defined in stdio.h. The file descriptor handles for standard input, output, and error are are STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO and are defined in unistd.h.

FILE DESCRIPTORS

The open system call associates a file descriptor with a file or physical device.

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

int open(const char *path, int oflag, ...);
```

The values for oflag include O_RDONLY, O_WRONLY,O_RDWR , O_CREAT, O_EXCL, O_NOCTTY, O_NONBLOCK, and O_TRUNC. Bitwise OR (|) flags to get a combined effect. The following code segment opens the file /home/lemmon/my.dat for reading.

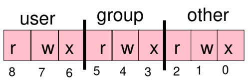
```
#include< sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>

int myfd;
myfd = open (``/home/lemmon/my.dat'', O_RDONLY);
```

The open system call sets the file's status flags according to the value of the second parameter. In this example, the flag O_RDONLY indicates that the file is to be opened read only. If we wanted to create a new file, then the flag O_CREAT would have to be used. In ths case, the third parameter would have to be specified of type mode_t. This third parameter specifies the permissions assocaited with the newly created file.

Each file has three classes associated with it: a user (owner), a group, and everyone else (others). The possible permissions or priviledges are read(r), write(w), and execute(x). These priviledges are specified separately for the user, group, and others. Figure 7 shows the layout of the permission mask. A 1 in the designated bit position indicates that the corresponding priviledge has been accorded to that file. The following code segment, shows how to create a file that can be read or written by the user and only read by everyone else.

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

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

int fd;

mode_t fd_mode= S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

if (( fd= open(‘‘/home/lemmon/my.dat’’, O_RDWR | O_CREAT, fd_mode))
== -1)
perror(‘‘could not open /home/lemmon/my.dat\n’’);
```



Symbol	Meaning
S_IRUSR	read bit (owner)
S_IWUSR	write bit(owner)
S_IXUSR	exec bit (owner)
S_IRWXU	r,w,x (owner)
S_IRGRP	read bit (group)
S_IWGRP	write bit (group)
S_IXGRP	exec bit (group)
S_IRWXG	r,w,x (group)
S_IROTH	read bit (other)
S_IWOTH	write bit(other)
S_IXOTH	exec bit(other)
S_IRWXO	r,w,x (other)
S_ISUID execution	set user ID on
S_ISGID execution	set group ID on

Figure 7: Typical layout of the permissions mask

The file descriptor is just an integer specifying the index into the file descriptor table. The file descriptor table is specific to a process. It contains an entry for each open file in the process and is therefore located in the process user area. An entry in the file descriptor table points to an entry in the system file table. The system file table contains an entry for each active open and is shared by all the processes in the ssytem. Several entries in the system may correspond to the same physical file. Each of the system file table entries corresponds to the same entry in the in-memory inode table. This table contains a copy of the inode for each active file in the system.

File Pointers and Buffering

The ANSI C standard I/O library uses file pointers rather than file descriptors. A file pointer is a pointer to a data structure called a file structure in the user area of the process. This data structure contains a buffer and a file descriptor. The following code segment opens the file /home/lemmon/my.dat for output and then writes a string to the file using the standard ANSI C library calls

```
#include<stdio.h>
FILE *myfp

if ((myfp=fopen(‘‘/home/lemmon/my.dat’’, ‘‘w’’))==NULL)
fprintf(stderr, ‘‘Could not fopen file \n’’);
```

```
else
fprintf(myfp,''this is a test'');
```

What happens when the program calls fprintf? The result depends on the file type. Disk files are fully buffered so that fprintf does not directly write to the file, but instead writes to the buffer in the file structure. When the buffer is full, fprintf calls the system function write to write the file. Buffering has an undesirable effect that output may be lost if the program crashes during an fprintf. The fflush command can then be used to flush the buffer. Terminal I/O works differently than disk I/O. In this case, the files associated with a terminal are line buffered. Line buffering means that the line is not written out until the buffer is full or until a newline symbol is encountered.

4.4 Filters, Redirection

A *filter* reads from standard input, performs a transformation, and outputs the result to standard output. Filters write their error messages to standard error. All of the parameters of a filter are communicated as command-line arguments. The filter input data should have no headers or trailers, and a filter should not require interaction with the user.

Recall that a file descriptor is an index into the process file descriptor table. Each entry in this table points to an entry in the system file table. A program can modify the file descriptor table entry so that it points to a different entry in the system file table. This action is called **redirection**. Most shells use the great than character (>) as redirection of standard otuput and the less than character (<) as rediretion of standard input.

To use redirection in a C program, one first opens the file (my.file) where the output is to be redirected and then the pointer to my.file into the entry for standard output by executing the dup2 system call. dup2 takes to parameters fildes and fildes2. It closes entry fildes2 of the file descriptor table and then copies the pointer of entry fildes into entry fildes2. The use of dup2 is illustrated in the following code segment.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>

int fd;

mode_t fd_mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

if (( fd=open(``my.file'', OWRONLY | O_CREAT, fd_mode))==-1)
perror(``could not open file my.file'');
else {
```

```
if (dup2(fd, STDOUT_FILENO)==-1)
   perror(``could not redirect standard output'');
   close(fd);
}
```

4.5 Pipes

A programmer can build complicated transformations from simple filters by feeding the standard output of one filter into the standard input of the next. This can be done in the shell through the following commands

```
ls -l > my.file
sort -n +4 < my.file
```

It can also be done using the special **piping** command

```
ls -l | sort -n +4
```

In this later case, the output of ls is connected to the standard input of sort through a communication buffer called a **pipe**. The system call used for piping in a C program has the following prototype

```
#include<unistd.h>

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

The pipe call creates a communication buffer which the caller can access through file descriptors fildes[0] and fildes[1]. The data written to fildes[1] is rad from fildes[0] on a first-in-first-out (FIFO) basis. A pipe has no external or permanent name, so a program can only access it through the two file descriptors. For this reason a pipe can only be used by the process that created it and by descendents that inherit the descriptors on fork. The following program impelmeents redirection of the ls and sort system calls.

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

void main(void)
{

int fd[2];
```

```
pid_t childpid;

pipe(fd);
if (( childpid=fork())==0) { /* ls is the child */
dup2(fd[1],STDOUT_FILENO);
close(fd[0]);
close(fd[1]);
execl(``/usr/bin/ls'',''ls'',''-l'',NULL);
perror(``The exec of ls failed'');
} else { /*sort is the parent*/
dup2(fd[0], STDIN_FILENO)
close(fd[0]);
close(fd[1]);
execl(``/usr/bin/sort'', ``sort'',''-n'',''+4'',NULL);
perror(``the exec of sort failed'');
}
exit(0);
}
```

Figure 8, 9, and 10 illustrate the state of the file descriptor table in the above code seigment. In figure 8, the child process inherits a copy of the file descriptor table of the parent. Figure 9 shows the file descriptor table after the child redirects its standard output and the parent redirects its standard input, but before either process closes unneeded file descriptors. Figure 10 shows the final configuration.

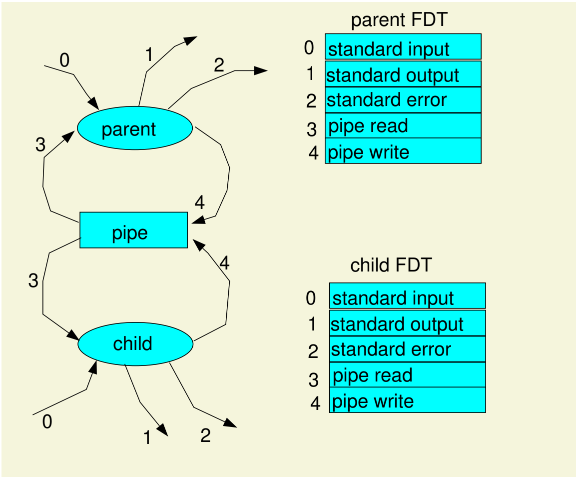


Figure 8: Status of file descriptor table after fork

4.6 Reading and Writing

UNIX provides sequential access to files through the read and write system calls. These calls read or write a block of data starting from the current file offset. They update

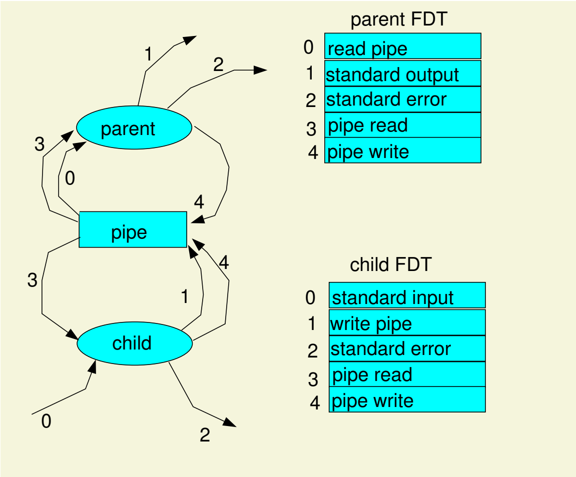


Figure 9: Status of FDT after dup2

the offset so that the next operation occurs where the last one left off.

```
#include<unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

The following code segment reads from from_fd and writes to to_fd.

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

#define BLKSIZE 1024

int from_fd, to_fd;
char buf[BLKSIZE];
int bytesread;

while ((bytesread= read(from_fd,buf,BLKSIZE))>0)
if (write(to_fd, buf, bytesread)<=0)
break;
```

The following program shows how these system calls can be used to copy a file. The names of the source and destination files are passed as command line arguments. Because the open for the destination file has O_CREAT | O_EXCL, the file copy fails if that file already exists. The loop for doing the copy handles partial writes and continues copying even if there is an interruption by a signal.

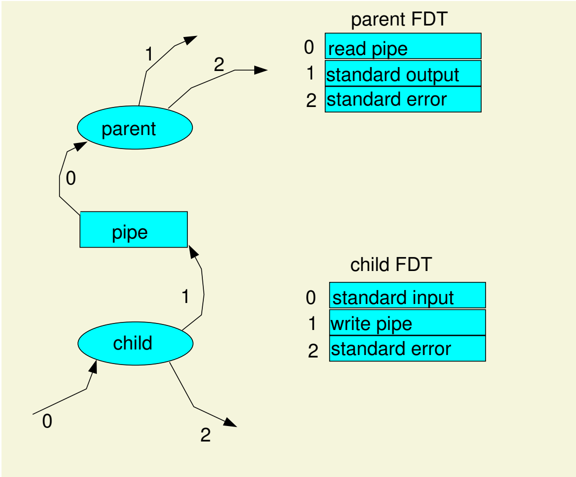


Figure 10: status of FDT after execl

```
#include <sys/types.h>
#include <sys/uio.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BLKSIZE 1024

void main(int argc, char *argv[])
{
   int from_fd, to_fd;
   int bytesread, byteswritten;
   char buf[BLKSIZE];
   char *bp;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);
      exit(1);
   }

   if ((from_fd = open(argv[1], O_RDONLY)) == -1) {
      fprintf(stderr, "Could not open %s: %s\n",
              argv[1], strerror(errno));
      exit(1);
   }
```

```
   if ((to_fd = open(argv[2], O_WRONLY | O_CREAT | O_EXCL,
              S_IRUSR | S_IWUSR)) == -1) {
      fprintf(stderr, "Could not create %s: %s\n",
              argv[2], strerror(errno));
      exit(1);
   }

   while (bytesread = read(from_fd, buf, BLKSIZE)) {
      if ((bytesread == -1) && (errno != EINTR))
         break;             /* real error occurred on the descriptor */

      else if (bytesread > 0) {
         bp = buf;
         while(byteswritten = write(to_fd, bp, bytesread)) {
             if ((byteswritten == -1) && (errno != EINTR))
                break;
             else if (byteswritten == bytesread)
                break;
             else if (byteswritten > 0) {
                bp += byteswritten;
                bytesread -= byteswritten;
             }
         }
         if (byteswritten == -1)
             break;
      }
   }
   close(from_fd);
   close(to_fd);
   exit(0);
}
```

4.7 `fcntl` and `select`

Communication buffers such as pipes can be empty if all of the information previosuly written to them has been read. The empty buffer is not an EOF indication. Rather it reflects the asynchronous nature of the communication between processes. Normally, when a process attempts to read such a buffer, it blocks (waits in the read) until the input is available.

Blocking I/O is a problem for a process that is monitoring several communication buffers. It can sometimes lead to a process ‘‘hanging’’ indefinitely, if it is waiting to read a buffer that is written to by another process that is also blocked.

One way to handle this problem is to use nonblocking I/O. By setting the appropriate control falgs, a program can cause reads to return immediately if there is no input available. The process then continuously attempts to read the input FDT. This is sometimes called **polling** and can be an inefficient way of using system resources.

A program does nonblocking I/O by setting the O_NONBLOCK flag associated with the file descriptor. The fcntl system call modifies the flag

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

int fcntl(int fildes, int cmd, /* arg */ ...);
```

The following code segimetn sets an already opened file descriptor, fd, for nonblocking I/O. The code segment reads the current value of the flags associated with fd, performs a bitwise OR with O_NONBLOCK, and installs the modified flags. After this segment executes, read system calls asociated with fd return immediately if there is no input available.

```
#include <fcntl.h>
#include <stdio.h>
int fd_flags;

if ((fd_flags = fcntl(fd, F_GETFL, 0)) == -1)
   perror("Could not get flags for fd");
else {
   fd_flags |= O_NONBLOCK;
   if (fcntl(fd, F_SETFL, fd_flags) == -1)
      perror("Could not set flags for fd");
}
```

An alternative to polling is for the process to block until I/O is available from any of a set of file descriptors. Blocking until at least one memeber of a set of conditions becomes true is called OR synchronization.

The select call provides a method of monitoring file descriptors for one of three conditions

- a read can be done without blocking
- a write can be done without blocking
- or an exceptional condition is pending (

The prototype for the select call is

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

int select(int nfds, fd_set *readfds, fd_set *writefds,
fd_set *execeptfds, struct timeval *timeout);
```

The first parameter of select is the number of bits to be checked in the file descriptor sets. This value must be at least one greater than the largest file descriptors to be checked. The readfd parameter specifies the set of descriptors to be monitored for reading. Similarly, writefds specifies the set of descriptors to be monitored for writing. and execptfds specifies the exceptional condition descriptors to be monitored.

On return select clears all of the descriptors in each readfds, writefds, and exceptfds except those descriptors that are ready. The return value for select is the number of file descriptors that are ready or -1 if there was an error. The last parameter is a timeout value used to force ar eturn from the select call. If select is interrupted by a signal, it returns -1 and sets errno to EINTR.

Macros are used to manipulate the descriptor sets.

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

void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
void FD_ISSET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

The FD_SET macro sets the bits in fdset corresponding to the fd file descriptor, while FD_CLR clears this bit. FD_ZERO macros clears all bits in fdset.

The following code segment continuously monitors two file descriptors pipe1 and pipe1 for input.

```
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <errno.h>

fd_set readset;
int maxfd;
int  pipe1;
int  pipe2;

    /* setup for pipe1 and pipe2 goes here */
maxfd = pipe1;                /* find the biggest fd for select */
if (pipe2 > maxfd)
   maxfd = pipe2;

for( ; ; ) {
   FD_ZERO(&readset);
   FD_SET(pipe1, &readset);
   FD_SET(pipe2, &readset);
```

```
   if ( (select(maxfd+1, &readset, NULL, NULL, NULL) == -1) &&
      (errno != EINTR) )
      perror("Select failed");
   else {
      if (FD_ISSET(pipe1, &readset)) {
          /*  read and process pipe1 input */
      }
      if (FD_ISSET(pipe2, &readset)) {
         /*  read and process pipe2 input */
      }
   }
}
```

4.8 FIFOs

FIFOs or *named pipes* are represented by special files and persist even after all processes have closed them. Any process with the appropriate permissions can access a FIFO. FIFOs can be created with the mkfifo command from the shell or with the mkfifo system call.

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

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

The following code segment creates a named pipe called myfifo that can be read by everybody but is only writable by the owner.

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

mode_t fifo_perms = S_IRUSR | S_IWUSR | S_IRGRP| S_IROTH;

if (mkfifo("myfifo", fifo_perms) == -1)
   perror("Could not create myfifo");
```

The following program creates a neamed pipe from a path specified on the command line. It then forks a child. The child process writes to a named pipe and the parent reads what the child has written.

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

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#define BUFSIZE 256
void main (int argc, char *argv[])
{
   mode_t fifo_mode = S_IRUSR | S_IWUSR;
   int fd;
   int status;
   char buf[BUFSIZE];
   unsigned strsize;
   int mychild;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s pipename\n", argv[0]);
      exit(1);
   }
                          /* create a named pipe with r/w for user */
   if ((mkfifo(argv[1], fifo_mode) == -1) && (errno != EEXIST)) {
      fprintf(stderr, "Could not create a named pipe: %s\n", argv[1]);
      exit(1);
   }

   if ((mychild = fork()) == -1){
      perror("Could not fork");
      exit(1);
   } else if (mychild == 0) {                 /* The child writes */
      fprintf(stderr, "Child[%ld] about to open FIFO %s\n",
                     (long)getpid(), argv[1]);
      if ((fd = open(argv[1], O_WRONLY))== -1) {
         perror("Child cannot open FIFO");
         exit(1);
      }
      sprintf(buf,
         "This was written by the child[%ld]\n", (long)getpid());
      strsize = strlen(buf) + 1;
      if (write(fd, buf, strsize) != strsize) {
         fprintf(stderr, "Child write to FIFO failed\n");
         exit(1);
      }
      fprintf(stderr, "Child[%ld] is done\n", (long)getpid());
   } else {                             /* The parent does a read */
      fprintf(stderr, "Parent[%ld] about to open FIFO %s\n",
                       (long)getpid(), argv[1]);
      if ((fd = open(argv[1], O_RDONLY | O_NONBLOCK)) == -1) {
         perror("Parent cannot open FIFO");
         exit(1);
      }
```

```
      fprintf(stderr,"Parent[%ld] about to read\n", (long)getpid());
      while ((wait(&status)== -1) && (errno == EINTR))
           ;
      if (read(fd, buf, BUFSIZE) <= 0) {
         perror("Parent read from FIFO failed\n");
         exit(1);
      }
      fprintf(stderr, "Parent[%ld] got: %s\n", (long)getpid(), buf);
   }
   exit(0);
}
```

5 Token Ring Example

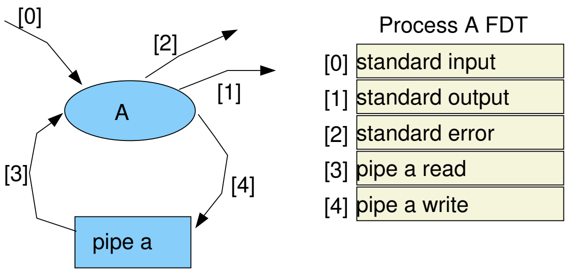
The ring topology is the simplest configuration for connecting communicating processes. Several network standards such as the *token ring*, *token bus*, and FDDI are based on ring connectivity. In these lectures we develop an implementation of a ring of processes communicating via pipes. The processes represent the nodes on the ring. Each process reads from standard input and writes to standard output. Process *n-1* redirects its standard output to the standard input of process n through a pipe. Once the ring structure is set up, the project can be extended to simulate networks or implement algorithms for mutual exclusion.

5.1 Single Process Ring

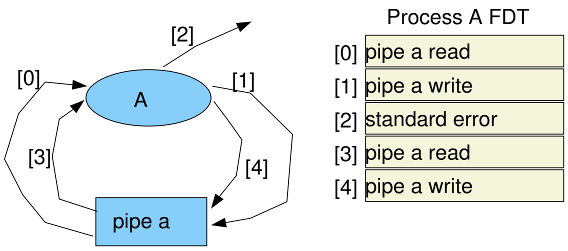
The following code segment connects the standard output of a process to its standard input through a pipe. Error checking has been omitted for simplicity.

```
#include<unistd.h>

int fd[2];

pipe(fd);
dup2(fd[0],STDIN_FILENO);
dup2(fd[1],STDOUT_FILENO);
close(fd[0]);
close(fd[1]);
```

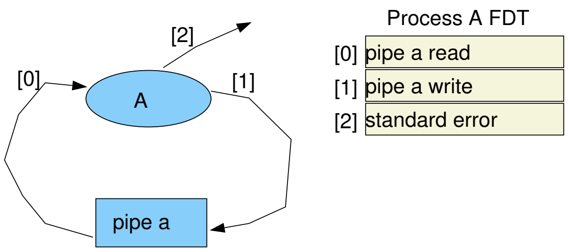
The following figure shows the file descriptor table after pipe A has been created. File descriptor entries [3] and [4] point to system file table entries that were created by a pipe call. A program can write to the pipe at this point by using a file descriptor value of 4 in a write.



This figure shows the status of the file descriptor table after the dup2 commands. At this point hte program can write to the pipe using either 1 or 4 as the file descriptor values.



This figure shows the status of the file descriptor table after descriptors 3 and 4 are closed



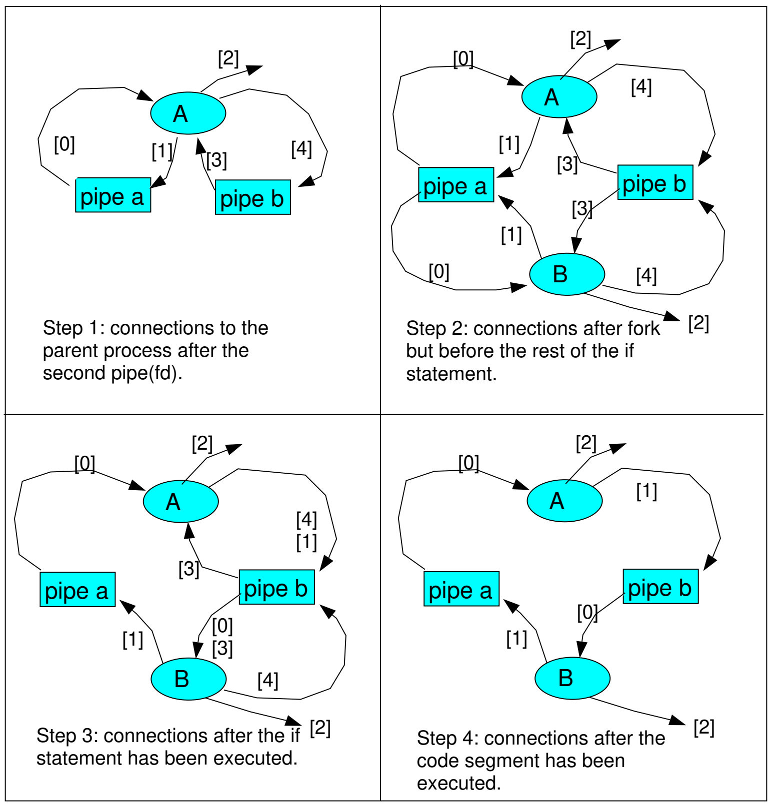
5.2 2 Process Token Ring

The following code segment creates a ring of two processes

```
#include <unistd.h>
int fd[2];
pid_t haschild;

pipe(fd);
dup2(fd[0], STDIN_FILENO);
```

```
dup2(fd[1], STDOUT_FILENO);
close(fd[0]);
close(fd[1]);
pipe(fd);
if (haschild = fork())
   dup2(fd[1], STDOUT_FILENO); /* parent redirects std output */
else
   dup2(fd[0], STDIN_FILENO);  /* child redirects std input */
close(fd[0]);
close(fd[1]);
```

The parent process redirects standard input through the first pipe to the standard output of the child and redirects standard output through the second pipe to the standard input of the child. The attached sequence of figures illustrate the connection mechanism.



5.3 Exercise: 2 Process Token Ring

- What happens if after connecting standard output to standard input via a pipe, the 2-process token ring example executes the following code segment

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

```
#include<stdio.h>

int i;
int myint;

for (i=0;i<10;i++){
write(STDOUT_FILENO, &i, sizeof(i));
read(STDIN_FILENO, &myint, sizeof(myint));
fprintf(stderr, ``%d\n'', myint);
}
```

- What happends if the code in the preceding exercises is replaced by

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

int i;
int myint;

for (i=0;i<10;i++){
read(STDIN_FILENO, &myint, sizeof(myint));
write(STDOUT_FILENO, &i, sizeof(i));
fprintf(stderr, ``%d\n'', myint);
}
```

- What happens in the example if the following code is used?

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

int i;
int myint;

for (i=0;i<10;i++){
printf(``%d'', i);
scanf(``%d'', &myint);
fprintf(stderr, ``%d\n'', myint);
}
```

5.4 Exercise: n Process Token Ring

The source code for a program that creates a ring of n processes is attached below.

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

```
#include <errno.h>
/*
 * Sample C program for generating a unidirectional ring of processes.
 * Invoke this program with a command-line argument indicating the
 * number of processes on the ring.  Communication is done via pipes
 * that connect the standard output of a process to the standard input
 * of its successor on the ring.  After the ring is created, each
 * process identifies itself with its process ID and the process ID
 * of its parent.  Each process then exits.
 */

void main(int argc,  char *argv[ ])
{
   int   i;             /* number of this process (starting with 1)   */
   int   childpid;      /* indicates process should spawn another     */
   int   nprocs;        /* total number of processes in ring          */
   int   fd[2];         /* file descriptors returned by pipe          */
   int   error;         /* return value from dup2 call                */

    /* check command line for a valid number of processes to generate */
   if ( (argc != 2) || ((nprocs = atoi (argv[1])) <= 0) ) {
       fprintf (stderr, "Usage: %s nprocs\n", argv[0]);
       exit(1);
   }
                        /* connect std input to std output via a pipe */
   if (pipe (fd) == -1) {
      perror("Could not create pipe");
      exit(1);
   }
   if ((dup2(fd[0], STDIN_FILENO) == -1) ||
       (dup2(fd[1], STDOUT_FILENO) == -1)) {
      perror("Could not dup pipes");
      exit(1);
   }
   if ((close(fd[0]) == -1) || (close(fd[1]) == -1)) {
      perror("Could not close extra descriptors");
      exit(1);
   }
        /* create the remaining processes with their connecting pipes */
   for (i = 1; i < nprocs;  i++) {
      if (pipe (fd) == -1) {
         fprintf(stderr,"Could not create pipe %d: %s\n",
            i, strerror(errno));
         exit(1);
      }
      if ((childpid = fork()) == -1) {
         fprintf(stderr, "Could not create child %d: %s\n",
            i, strerror(errno));
```

```
        exit(1);
      }
      if (childpid > 0)        /* for parent process, reassign stdout */
          error = dup2(fd[1], STDOUT_FILENO);
      else
          error = dup2(fd[0], STDIN_FILENO);
      if (error == -1) {
         fprintf(stderr, "Could not dup pipes for iteration %d: %s\n",
                 i, strerror(errno));
         exit(1);
      }
      if ((close(fd[0]) == -1) || (close(fd[1]) == -1)) {
         fprintf(stderr, "Could not close extra descriptors %d: %s\n",
                i, strerror(errno));
         exit(1);
      }
      if (childpid)
         break;
   }
                               /* say hello to the world */
   fprintf(stderr,"This is process %d with ID %d and parent id %d\n",
      i, (int)getpid(), (int)getppid());
   exit (0);
}     /* end of main program here */
```

- Run ring several times for various n between 1 and 20 and observe what happens.
- Modify your program so it puts a wait before the final fprintf command. What happens?
- Modify the program so it places the wait after the fprintf. What happens?
- Modify the program so it can be a bidirectional ring.

5.5 Signals and Timers

A **signal** is a software notification to a process of an event. A signal is **generated** when the event causes that signal to occur. A signal is **delivered** when the process takes action based on that signal. The **lifetime** of a signal is the interval between its generation and its delivery. A signal that has been generated but not yet delivered is **pending**. A process **catches** a signal if it executes a **signal handler** when the signal is delivered. A program installs a **signal handler** using the system call sigaction and the user provides the functionality for the handling function.

5.6 Sending Signals

Every signal has a symbolic name. The signal names are defined in signal.h. The following table lists the signals which an operating system must have to be POSIX.1 compliant.

In all of the following signals the default action is process termination.

Symbol	Meaning
SIGABRT	abnormal termination as initiated by abort
SIGALRM	timeout signal as initiated by alarm
SIGFPE	error in arithmetic operation as in division by zero
SIGHUP	hang up on controlling terminal
SIGILL	invalid hardware instruction
SIGINT	interactive attention signal
SIGKILL	terminate (cannot be caught or ignored)
SIGPIPE	write on a pipe with no readers
SIGQUIT	interactive termination
SIGSEGV	invalid memory reference
SIGTERM	termination
SIGUSR1	user defined signal 1
SIGUSR2	user defined signal 2

The following set of optional signals is also defined in POSIX.1 for job control. Shells use these signals to control the interaction of background and foreground processes. These signals are used if _POSIX_JOB_CONTROL is defined.

Symbol	Meaning	Default Action
SIGCHLD	Child died or stopped	signal ignored
SIGCONT	continue process	continues stopped process
SIGSTOP	stop the process	stops process (uncatchable)
SIGTSTP	stop the process	stops process (from terminal)
SIGTTIN	stop the process that tries to read form terminal	stop the background process
SIGRTMIN-SIGRTMAX	termination	real-time signals (discussed later)

Signals can be generated from within the shell using the kill command. This may seem strange, but actually most signals serve the purpose of terminating processes, so this is not really that unusual. The prototypes for the kill command are

```
kill -s signal pid ...
kill -l [exit_status]
kill [-signal] pid...
```

for example the following command sends the SIGUSR1 signal to process 3423.

```
kill -USR1 3423
```

The following command lists the symbolic signal names for the system.

```
kill -l
```

Use the kill system call from within a C program to send a signal to a process. This call takes a process ID and a signal number as parameters

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

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

If the process id is positive, kill sends the specified signal to the process with that pid. If pid is negative,, then kill sends the sginal to the process group with group ID equal to |pid|. If pid=0, then kill sends the message to all processes of the caller's process group.

A process can send a signal to itself with the raise function. This function has only one argument, the signal parameter

```
#include < signal.h>

int raise(int sig);
```

The alarm function causes a SIGALRM to be sent to the calling process after a specified number of real time seconds have passed.

```
#include < unistd.h>

unsigned int alarm(unsigned int seconds);
```

Requests to alarm are not stacked, so that if a program calls alarm before the previous one expres, the alarm is reset to the new value. Since the default action for SIGALRM is to terminate the process, the following program runs for approximately ten seconds of wall-clock time.

```
#include <unistd.h>

void main(void){
alarm(10);
for( ; ; ){
}
}
```

5.7 Signal Masks and Signal Sets

A process can temporarily prevent a signal from being delivered by blocking it. Blocked signals do not affect the behaviour of the process until they are delivered. The process

signal mask gives the set of signals that are currently being blocked. The signal mask is of type sigset_t.

Blocking a signal is not the same as ignoring the signal. In blocking a signal, the process simply waits until it has time to process it (i.e. after removing the block). In ignoring the signal, the process catches the signal and then discards it.

The following signal set functions are used to specify blocking or unblocking on groups of signals.

```
#include <signal.h>

sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

The sigemptyset function initializes a signal set to contain no signals, while sigfillset initializes the signal set to contain all signals. These calls are used for initialization of the signal snet. The sigaddset and sigdelset calls are or delete signals to the signal set. The function call sigismember queries if a given signal is in the signal set.

The following code segment initializes signal set twosigs to contain exactly two signal, SIGINT and SIGQUIT.

```
#include<signal.h>
sigset_t twosigs;

sigemptyset(&twosigs);
sigaddset(&twosigs, SIGINT);
sigaddset(&twosigs, SIGQUIT);
```

A process can examine or modify its process signal mask with the sigprocmask function call.

```
#include<signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

The how parameter is an integer indicating the manner in which the signal mask is to be modified. The three possibilities for this parameter are

- SIG_BLOCK: add a collection of signals to those currently blocked.
- SIG_UNBLOCK: delete a collection of signals from those currently blocked.

- SIG_SETMASK: set the collection of signals being blocked to the collection given.

The set parameter of sigprocmask points to the set of signals to be used for the modification and the oset parameter is the address of a sigset_t variable for holding the set of signals taht were blocked before the call to sigprocmask. The second or third parameters may be NULL. sigprocmask returns -1 if there is an error and sets errno, 0 is returned for a successful call. The following code segment adds SIGINT to the list of blocked signals

```
#include<stdio.h>
#include<signal.h>
sigset_t newsigset;

sigemptyset(&newsigset)
sigaddset(&newsigset, SIGINT);
if(sigprocmask(SIG_BLOCK, &newsigset, NULL) < 0)
perror(``could not block the signal'');
```

The following example shows the use of these function calls in a program. In this program, a message is displayed, blocks the SIGINT signal while doing some useless work, unblocks the signal and does more useless work. The program repeats this sequence continuously.

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

void main(int argc, char **argv){
double y;
sigset_t intmask;
int i, repeat_factor;

if(argc!=2){
fprintf(stderr,''useage: %s repeat_factor\n'',argv[0]);
exit(1);
}
repeat_factor=atoi(argv[1]);
sigemptyset(&intmask);
sigaddset(&intmask,SIGINT);
for( ; ; ){
sigprocmask(SIG_BLOCK, &intmask, NULL);
fprintf(stderr, ``SIGINT signal blocked\n'');
for(i=0;i<repeat_factor; i++) y= sin((double) i);
sigprocmask(SIG_UNBLOCK, &intmask, NULL);
fprintf(stderr,''SIGINT signal unblocked\n'');
for(i=0;i<repeat_factor;i++) y=sin((double) i);
```

```
fprintf(stderr,''unblocked calculation is finished'');
}
}
```

The following code segment is used to illustrate what happens to signal masks under forking and exec system calls. In this code segment, both the parent and child attempt to restore the signal mask to its original value after the fork.

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>

sigset_t oldmask, blockmask;
pid_t mychild;

sigfillset(&blockmask);
if(sigprocmask(SIG_SETMASK, &blockmask, &oldmask)==-1){
perror(``could not block all signals'');
exit(1);
}
if((mychild==fork())==-1){
perror(``could not fork child'');
exit(1);
} else if (mychild==0) { /*child code here*/
if(sigprocmask(SIG_SETMASK, &oldmask, NULL)==-1){
  perror(``Child could not restore signal mask'');
  exit(1);
}
/*rest of child code*/
} else { /*parent code here*/
if(sigprocmask(SIG_SETMASK, &oldmask,NULL)==-1){
  perror(``parent could not restore signal mask'');
  exit(1);
}
/* rest of parent code*/
}
}
```

The processes inherit the signal mask after both fork and exec commands. The child created by the fork in the above example has a copy of the original signal mask saved in oldmask. Note, however, that if we then did an exec that the original code of the child will be overwritten and the old mask saved in oldmask will be written over. In this case then, the child will be unable to restore the original mask.

5.8 Catching and Ignoring Signals

The sigaction function installs signal handlers for a process. A data structure of the type struct sigaction holds the handler information. The sigaction system call has three parameters: the signal number, a pointer to the new handler structure, and a pointer to the old structure. The prototype for sigaction is

```
#include<signal.h>

int sigaction(int signo, const struct sigaction *act,
 struct sigaction *oact);

struct sigaction{
void (*sa_handler)();   /*pointer to function or SIG_DFL or SIG_IGN*/
sigset_t sa_mask/ /*additional signal to be blocked during
  execution of hander*/
int sa_flags; /*special flags and options*/
}
```

The following code segment illustrates how to use sigaction to install a handler for SIGINT.

```
#include<signal.h>
#include<stdio.h>
struct sigaction newact;

newact.sa_handler = mysighand; /*set the new handler*/
sigemptyset(&newact.sa_mask); /*no other signals blocked*/
newact.sa_flags=0; /*no special options*/
if(sigaction(SIGINT, &newact, NULL) ==-1)
perror(``could not install SIGINT signal handler'');
```

A signal handler is an ordinary function that returns void and has one integer parameter. When the operating system delivers the signal, it sets this parameter to the number of the signal that was delivered. Most signal handlers ignore this value, but it is possible to have a single signal handler for many signals. The usefulness of handlers is limited by the inability to pass values to them. However, the realtime (POSIX.1b) extensions to signals have changed this and will be discussed below.

POSIX.1 guarantees that write is async-signal safe (meaning it can be called safely inside a signal handler. There are no such guarantees for fprintf. So use the write function within a handler.

5.9 Waiting for Signals

One reason for using signals is to avoid **busy waiting**. Busy waiting means that that CPU cycles are continuously used to wait for an event (i.e. a big while loop). A more efficient method is to suspend the process until the waited for event occurs. UNIX provides two functions that allow a process to suspend itself until a signal occurs; pause and sigsuspend.

pause suspends the calling process until a signal that is not being ignored is delivered to the process. If a signal is caught by the process, the pause returns after the signal handler returns.

```
#include<unistd.h>

int pause(void);
```

The pause function always returns -1.

The following code segment causes a process to wait for a particular signal with pause by having the signal handler set the signal_received variable to 1. Initially, this variable is zero. This example uses a loop because the pause returns when any signal is delivered to the process. If the signal was not the right one, then signal_received is still 0 and the loop calls pause again.

```
#include<unistd.h>
int signal_received=0;

while(signal_received==0) pause();
```

Note that if a signal is delivered between the testing of signal_received and pause, then pause will not catch that signal and return from it.

The following example is a negative example showing an incorrect approach to solving the preceding problem found in our example above.

```
#include<unistd.h>
#include<signal.h>
int signal_received=0;

sig_set_t sigset;
int signum;

sigemptyset(&sigset);
sigaddset(&sigset,signum);
sigprocmask(SIG_BLOCK,&sigset, NULL);
while(signal_received==0) pause();
```

The preceding program executes the pause while the signal is blocked. As a result the program never receives the signal and pause never returns. The only way to fix this is to have the program unblock the signal before executing pause. However, we might still receive a signal between the unblocking and pause command, so that our problem persists

Basically, there is no way around this problem using pause. The actions of unblocking the signal and pausing must be done **atomically**, i.e. in one logically indivisible unit. This requires a new system function sigsuspend.

```
#include<signal.h>

int sigsuspend(const sigset_t *sigmask);
```

The sigsuspend functions sets the signal mask to the one pointed to by sigmask and suspends until a signal is delivered to the process. The sigmask can be used to unblock the signal the program is looking for. When sigsuspend returns, the signal mask is reset to the value it had before the sigsuspend. The following code segment suspends a process until the signal given by the value of signum occurs. It then restores the original signal mask.

```
#include<signal.h>
int signal_received=0;

sigset_t sigset;
sigset_t sigoldmak;
int signum;

sigprocmask(SIG_SETMASK, NULL, &sigoldmask);
sigprocmask(SIG_SETMASK, NULL, &sigset);
sigaddset(&sigset, signum);
sicprocmask(SIG_BLOCK, &sigset, NULL);

sigdelset(&sigset, signum);

while(signal_received==0) sigsuspend(&sigset);
sigprocmask(SIG_SETMASK, &sigoldmask, NULL);
```

5.10 Realtime Signals

One deficiency with the original POSIX standard was the lack of user defined signals. There are, basically, only two SIGUSR1 and SIGUSR2. In many applications we'll need more than this. To address this problem, POSIX.1b defines a new set of signals. The new signal numbers begin with SIGRTMIN and go to SIGRTMAX. There should be at least RTSIG_MAX real-time signal between these two bounds and the standard set a minimum value at 8. The new features described below only apply to these new real-time signals.

POSIX.1b introduces enhancements to traditional POSIX.1 signal handling that are relevant to realtime operation. The revised standard extends the sigaction structure to allow for additional parameters to the signal handler. An implementation (such as Solaris) supports these expanded capabilities if _POSIX_REALTIME_SIGNALS is defined.

```
#include<signal.h>

struct sigaction{
void (*sa_handler)(); /*SIG_DFL, SIG_IGN, or
  pointer to function*/
void (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t sa_mask; /*additinal signals to be blocked
  curring execution of handler*/
int sa_flags; /*special flags and options*/
};
```

The new sa_sigaction member provides an alternative type of signal handler. The new type of handler is used if sa_flags && SA_SIGINFO is true. The form of the new handler is

```
void func(int signo, siginfo_t *info, void *context);
```

The signo is the same as before and is set to the number of the caught signal. The context is current undefined by the standard. The *info structure contains at least the following members;

```
int si_signo; /*signal number*/
int si_code; /* cause of the signal*/
union sigval si_value; /* signal value*/
```

The si_signo contains the signal number and this must be the same as signo. The si_code inducates the cause of the signal, its possible values are

SI_USER generated by kill, raise, or abort
SI_QUEUE sigqueue
SI_TIMER timer generated signal
SI_ASYNCHIO completion of asynch I/O
SI_MESGQ indicates arrival of amessage on an empty message queue.

The sigval union has the members sival_int and *sival_ptr to allow integers or pointers to be transmitted to the signal handler.

In the original POSIX standard, there is no guarantee as to what happens if there are multiple instances of a signal pending. One of these instances is guaranteed delivery, but the others may be lost. The sigqueue function is an extension to kill which permits signals to be queued.

```
#include<signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);
```

The additional parameter specifies the value to be received by the signal handler in the si_value member of its info structure.

To guarantee that multiple instances of a particular signal are queued, a program must use the SA_SIGINFO flag in the sa_flags when setting up the handler with sigaction. Multiple signals of the same signal number generated by sigqueue are queued up to a maximum of SIGQUEUE_MAX which is typically 32.

Whether signal are guaranteed to be queued is determined by the method used to generate the signals, not by how the signal handler is set up. Multiple instances of a signal generated with kill may not be queued, even if instances of the signal generated by sigqueue will be queued.

The following program shows a program that sends queued signals to a process. The process behaves like kill, but uses the siqueue command. It allows the sending of a queued signal and an integer value for each one.

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

void main(int argc, char *argv[])
{
   union sigval qval;
   int val;
   int pid;
   int signo;

   if (argc != 4) {
      fprintf(stderr, "Usage: %s pid signal value\n", argv[0]);
      exit(1);
   }

   pid = atoi(argv[1]);
   signo = atoi(argv[2]);
   val = atoi(argv[3]);

   fprintf(stderr, "Sending signal %d with value %d to process %d\n",
      signo, val, pid);
   qval.sival_int = val;
   sigqueue(pid, signo, qval);
}
```

This program shows a program that prints its process ID, sets up a signal handler for

SIGUSR1, blocks the signal, and waits for input before exiting. The signal handler just displays the values it receives from its parameters.

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

void my_handler(int signo, siginfo_t* info, void *context)
{
   fprintf(stderr, "Signal handler entered for signal number %d\n",
      signo);
   fprintf(stderr, "   si_signo = %3d\n", info->si_signo);
   fprintf(stderr, "   si_code  = ");

   if (info->si_code == SI_USER) fprintf(stderr, "USER\n");
   else if (info->si_code == SI_QUEUE) fprintf(stderr, "QUEUE\n");
   else if (info->si_code == SI_TIMER) fprintf(stderr, "TIMER\n");
   else if (info->si_code == SI_ASYNCIO) fprintf(stderr, "ASYNCIO\n");
   else if (info->si_code == SI_MESGQ) fprintf(stderr, "MESGQ\n");
   else fprintf(stderr, "%d\n", info->si_code);
   fprintf(stderr,"   si_value = %3d\n", info->si_value.sival_int);
}

void main(void)
{
   struct sigaction act;
   sigset_t sigset;

   fprintf(stderr, "Process ID is %ld\n", (long)getpid());

   sigemptyset(&sigset);
   sigaddset(&sigset, SIGALRM);
   sigprocmask(SIG_BLOCK, &sigset, NULL);
   sigemptyset(&act.sa_mask);
   act.sa_flags = SA_SIGINFO;
   act.sa_sigaction = my_handler;
   if (sigaction(SIGUSR1, &act, NULL) < 0) {
      perror("Sigaction failed");
      exit(1);
   }
   fprintf(stderr, "Signal SIGUSR1 = %d ready\n", SIGUSR1);

   for( ; ; )
      pause();
}
```

5.11 Basic UNIX Clocks

Clocks and timers have been part of UNIX systems for sometime. Standard UNIX (POSIX.1) has three clocks. One tells the time and is directly accessible to the user. The other two have other purposes which don't often concern the application programmer. There are two types of timers: one-shot timers and repeating. The one-shot timer only expires once and is used to signal a special event that needs to occur only once. Repeating timers go off at regular intervals and are used to schedule periodic events.

There is, basically, only one clock which application programs use in standard UNIX. This clock is accessed through the system call time and it returns a number representing the number of seconds since Jan 1, 1970 which is called the **epoch** The prototype for the time call is

```
#include< time.h>

time_t time(time_t *tloc);
```

The value returned is the number of seconds since the epoch. The parameter in the system call is a pointer to a time_t structure where the system call will store the number of seconds for you.

The time_t is convenient for calculating differences between times, but is cumbersome for printing dates. The C library function ctime converts a time to an ASCII string suitable for printing

```
#include<time.h>

char *ctime(const time_t *clock)
```

The time function measures **real time** or wall-clock time. In a multiprogramming environment, we may be more interested in the amount of time an individual process uses. This is called **virtual time**. The system call times returns information about the execution times of a process and its children.

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

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

The clock_t type holds a number of clock ticks. The structure tms contains at least the following members

```
clock_t  tms_utime; /*user CPU time*/
clock_t tms_stime; /*system CPU time*/
clock_t tms_cutime; /*user CPU time of terminated children*/
clock_t tms_sutime; /*system CPU time of terminated children*/
```

The function returns the amount of elapsed time in clock ticks since an arbitrary point in the past. The following code segment caculates the fraction of time that a process is running on a processor while performing a computation.

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

clock_t real_start;
clock_t real_end;
clock_t ticks_used;
struct tms process_start;
struct tms process_end;

if ((real_start = times(&process_start)) == -1)
   perror("Could not get starting times");
else {
   /* perform calculation to be timed */
   if ((real_end = times(&process_end)) == -1)
      perror("Could not get ending times");
   else {
      ticks_used = process_end.tms_utime + process_end.tms_cstime -
         process_start.tms_utime - process_start.tms_cstime;
      printf("Fraction of time running = %f\n",
         (double)(ticks_used)/(real_end - real_start));
   }
}
```

5.12 One Shot Timers

The most well known example of a one-shot timer is the sleep call.

```
#include<unistd.h>

unsigned sleep(unsigned seconds);
```

When a process executes sleep, it blocks for the number of seconds specified. The sleep function can be used as a sort of crude interval timer. For example, let's examine the following code segment

```
while (1) {
sleep(nsec);
/*do something*/
}
```

This code attempts to execute the ‘‘do something’’ code segment every nsec seconds. The problem, however, is that nsec is in ‘‘seconds’’. In many situations, this sort of timing is much too coarse for actual control applications. In addition to this, the actual period for the while loop will be nsec plus some ‘‘drift’’ associated with the amount of time required to ‘‘do something’’. In a multitasking environment where signals can asynchronously interrupt processes, it is impossible to determine precisely how long it takes to ‘‘do something’’. To get around this difficulty, the standard UNIX interval timers were created.

5.13 Standard UNIX Interval Timers

The standard UNIX interval timers were defined in the Berkeley and SR4(1170) versions of UNIX. A conforming Spec 1170 implementation must provide each process with the following three interval timers.

- ITIMER_REAL: decrements in real time and generates a SIGALRM signal when it expires.
- ITIMER_VIRTUAL: decrements in virtual time (time used by the process) and generates a SIGVTALRM signal when it expires
- ITIMER_PROF: decrements in virtual time and system time for the process and generates a SIGPROF signal when it expires.

Interval timers use a struct itimerval structure to contain the following members:

```
struct timeval it_value; /*time until next expiration*/
struct timeval it_interval; /*value to reload into timer*/
```

The system call setitimer is provided for starting and stopping interval timers.

```
#include<sys/time.h>
int setitimer(int which, const struct itimerval *value;
       struct itimerval *ovalue);
```

The setitimer function returns 0 on success. It returns -1 and sets errno on failure. The which parameter specifies the timer. The caller specifies the time intevale for setting the timer in *value. The system call fills the members of the structure point to by ovalue with the current timer values. This pointer may be NULL if we don’t need to see the information.

Use getitimer to determine the amount of time remaining on a Spec 1170 timer

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

int getitimer(int which,struct itimerval *value);
```

The system call sets *value to the time remaining on timer which and returns 0 on success. It returns -1 and sets errno on failure.

The following program causes the process to print out an asterisk for each two seconds of CPU time used. The ITIMER_PROF timer generates a SIGPROF signal after every two seconds of CPU time used by the process. The process catches the SIGPROF signal and handles it with myhandler.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>

char astbuf[] = "*";
static void myhandler(int s)
{
   write(STDERR_FILENO, astbuf, sizeof(char));
}

/* set up the myhandler handler for signal SIGPROF */
void init_timer_interrupt(void)
{
   struct sigaction newact;
   newact.sa_handler = myhandler;
   newact.sa_flags = SA_RESTART;
   sigemptyset(&newact.sa_mask);
   sigaction(SIGPROF, &newact, NULL);
}

/* set the ITIMER_PROF interval timer for 2-second intervals */
void setup_interval_timer(void)
{
   struct itimerval value;
   value.it_interval.tv_sec = 2;
   value.it_interval.tv_usec = 0;
   value.it_value = value.it_interval;
   setitimer(ITIMER_PROF, &value, NULL);
}

void main(int argc, char *argv[])
{
   init_timer_interrupt();
   setup_interval_timer();
   /* execute rest of main program here */
   exit(0);
}
```

5.14 POSIX Interval Timers

The 1170 specification do not provide sufficient flexiblity to deal with real time tasks. To be useful in a realtime programming environment, we need the following

- support of additional clocks beyond ITIMER_REAL, ITIMER_VIRTUAL, and ITIMER_PROF.
- allowance for greater time resolution (i.e. nanosecond resolution)
- ability to determine timer overruns.
- ability to use something other than SIGALRM to indicate timer expiration.

As a result, POSIX.1b defines a new set of timer functions that meet the additional requirements stated above. Note that in compiling POSIX.1b programs, the compiler options must include -lposix4. (POSIX.4 is an older name for the POSIX.1b standard).

In POSIX.1, each process has only three timers. In POSIX.1b, there are a small number of clocks such as CLOCK_REALTIME and a process can create many independent timers based on these hardware clocks.

POSIX.1b timers are based on the structure itimerspec which has the following members

```
struct timespec it_interval; /*timer period*/
struct timespec it_value; /*timer expiration*/
```

In addition to this, POSIX.1b realtime extensions define a fine-grained facility for telling time. The structure timespec provides the additional resolution and has the following members

```
struct timespec{
time_t tv_sec; /*seconds*/
long tv_nsec; /*nanoseconds*/
};
```

A process can create specific timers by using the system call timer_create. The timers are not inherited on fork. The prototype for the function call is

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

int timer_create(clockid_t clock_id, struct sigevent *evp,
 timer_t *timerid);

struct sigevent{
int sigev_notify; /*notification type*/
int sigev_signo; /*signal number*/
```

```
union sigval sigev_value; /*signal value*/
};

union sigval{
int sival_int; /*integer value*/
void *sival_ptr; /*pointer value*/
};
```

The clock_id specifies which clock the timer is based on and *timerid holds the ID of the created timer. The call timer_create returns 0 on success. It returns -1 and sets errno on failure.

The members of the sigevent structure and the sigval union define what ‘‘event’’ occurs upon timer expiration. The *evp parameter specifies which siganl should be sent to the process when the timer expires. If evp is NULL, the timer generates the default. For CLOCK_REALTIME the default signal is SIGALRM. In order for the timer expiration to generate a different signal, the structure member evp->sigev_signo must be set to the desired signal number. The evp->sigev_notify member indicates the action to be taken when the timer expires. Normally, this member is SIGEV_SIGNAL to indicate that the signal is generated. The program can prevent the timer expiration from generating a signal by setting evp->sigev_notify to SIGEV_NONE.

If several timers generate the same signal, the handler can use evp->sigev_value to distinguish which timer generated the signal. In order to do this, th eprogram must use the SA_SIGINFO falg in the sa_flags member of the structure sigaction when it installs the handeler for the signal.

The following three functions manipulate the process’ POSIX.1b timers.

```
#include<time.h>

int timer_settime(timer_t timerid, int flags);
const struct itimerspec *value,
struct itimerspec *ovalue);

int timer_gettime(timer_t timerid, struct itimerspec *value);

int timer_getoverrun(timer_t timerid);
```

The timer_settime function starts or stops a timer that was created by calling timer_create. The flags parameter indicates whether th timer uses relative or absolute time. Relative time is similar to Spec 1170 timers while absolute time allows for greater accuracy and control of timer drift (this is what we use for realtime programming) The last two parameters have the same meaning as for setitimer. The timer_settime and timer_gettime functions return 0 on success while timer_getoverrun returns the number of timer overruns. On failure all of these routines return -1 and set errno.

The following code segment shows an example of the simplest and most common way in which POSIX.1b facilities are used.

```
#define _POSIX_C_SOURCE 199506L

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

#define OUR_CLOCK CLOCK_REALTIME

char *prog_name;

void timer_intr(int sig, siginfo_t *extra, void *cruft)
{
int noverflow;
if(noverflow=timer_getoverrun(*(timer_t *)
extra->si_value.sival_ptr)){
/*timer has overflowed -- error !*/
}
return;
}

timer_t mytimer;

main(int argc, char **argv){
int c;
struct itimerspec i;
struct timespec resolution;
struct sigaction sa;
sigset_t allsigs;
struct sigevent timer_event;

prog_name=argv[0];

sigemptyset(&sa.sa_mask);
sa.sa_flags=SA_SIGINFO; /*real-time signal*/
sa.sa_sigaction=timer_intr; /*pointer to action*/

if(sigaction(SIGRTMIN, &sa, NULL) < 0){
perror(``sigaction error'');
exit(1);
}

/*
 * first determine whether the desired clock exists
 */

if (clock_getres(OUR_CLOCK, &resolution) < 0){
  perror(``clock_getres error'');
```

```
  exit(1);
}

printf(``clock resolution %d sec %d nsec \n'',
resolution.tv_sec, resolution.tv_nsec);

/*
 * create a timer based upon the CLOCK_REALTIME clock
 */

 i.it_interval.tv_sec=0;

/* set resolution to one-tenth of the maximum allowed*/

 i.it_interval.tv_nsec=resolution.tv_nsec*10;
 i.it_value=i.it_interval;

/*
 * this describes the asynchronous notification to be posted
 * upon this timer's expiration:
 *
 * - use signals
 * - send SIGRTMIN
 * - send extra data consisting of a pointer back to the timer ID
 *   cannot pass the timer ID itself because we haven't created it
 *   yet.
 */

 timer_event.sigev_notify=SIGEV_SIGNAL;
 timer_event.sigev_signo= SIGRTMIN;
 timer_event.sigev_value.sival_ptr = (void *)&mytimer;

 if (timer_create(OUR_CLOCK, &timer_event, &mytimer) < 0){
perror(``timer create error'');
exit(1);
 }

 /* relative timer, go off at the end of the interval*/

 if(timer_settime(mytimer, 0 , &i, NULL) < 0){
perror(``settimer'');
exit(3);
}

sigemptyset(&allsigs);
while(1){
sigsuspend(&allsigs);
}
```

```
exit(4);
}
```

5.15 Exercise

The exercise in the section is actually an extended project associated with the ARO-DURIP lab (http://www.nd.edu/~lemmon/aro-durip). The results from the project will eventually be posted on the above website. The objective of this exercise is to write a program which monitors the memory locations in two dSpace miniboxes in a simultaneous manner. The extra difficult introduced by this project is that our communication with the dSpace boxes will have to go through DSP_DAEMON that was provided by dSpace.

What we need you to do is write a program which

- spawns two child processes that read from a specified location in dspace box memory every 10 milliseconds,
- The child processes signal to the parent process that the memory location has changed.
- The parent process signals to the children that it has been notified of the change. Upon notification, the children then reset the memory location in the dspace box.

5.16 Communicating with dSpace Boxes

Connecting a process to a dSpace box requires that a dsp_daemon already be connected to the desired minibox. Such a daemon can be started and registered by using the BoxGUI interface described at website http://www.nd.edu/~lemmon/aro-durip.

Once the appropriate daemons are running, then the procedure connecting to the dSpace box consists of the following steps.

- DSP_net_connect(client_name);

  This call begins to connect the process to the client.
- DSP_register_host_app(prog_name);

  This call registers the host application to the central daemon running on rottweiler. It is important that your process has a distinct name associated with it for this to work correctly.
- DSP_board_index(DS1102, &board_index);

  This call returns a unique index number for the Dspace board (type DS1102) that is being connected to. In all future calls, the board_index pointer will be used to access the board.
- DSP_lock_board(board_index);

  This call locks the board so it can only be accessed by a single process.

- DSP_read_dual_port_memory(board_index, MQUEUE_IN, &new_value);

  This call reads the memory location specified on MQUEUE_IN on board_index and places it in new_value. The format of the data is in a special TI (Texas Instrument) format that has to be converted into IEEE standard floating point number. This is accomplished with the last call, DSP_cvt_ti_to_ieee(new_value);

- DSP_unlock_board(board_index);
  DSP_unregister_host_app();

  These calls must be called upon exit to release the daemon and unlock the board.

5.17 Sample Dspace Program

Details on the preceding function calls will be found in the lab reference manuals on the CLIB software environment. A simple sample program illustrating the use of these function calls is provided below.

```
/* hdem_2.c ***********************************************************
*                                                                     *
* Step-Signal Generator application demo example                      *
*                                                                     *
*                                                                     *
*  Instruction to compile on a SUN Workstation under Solaris 2.x:     *
*     cc -Xa hdem_2.c -DUNIX -DNET -I$DSPACE_ROOT/clib -o hdem_2 \    *
*          -L$DSPACE_ROOT/clib -lclib -lsocket -lnsl                  *
*                                                                     *
*  Instruction to compile on an HP Workstation:                       *
*     cc -Aa hdem_2.c -DUNIX -DNET -I$DSPACE_ROOT/clib -o hdem_2 \    *
*          -L$DSPACE_ROOT/clib -lclib                                 *
*                                                                     *
* Copyright (C) 1995 by dSPACE GmbH, Germany, Paderborn               *
**********************************************************************/

/* $RCSfile: hdem_2.c $ $Revision: 1.2 $ $Date: 1995/05/04 11:19:16 $ */

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

#include <clib.h>           /* Host-DSP interface library include file */


#if defined __STDC__            /* distinct ANSI-C and K&R-C compilers */
#define ARG_1(t1,v1) (t1 v1)
#define ARG_2(t1,v1,t2,v2) (t1 v1, t2 v2)
#else
#define ARG_1(t1,v1) (v1) \
  t1 v1;
```

```
#define ARG_2(t1,v1,t2,v2) (v1, v2) \
  t1 v1; t2 v2;
#endif


#define DP_MEM_OFFS 0        /* use first dual-port memory address */
#define STEP_SIZE 1.0

unsigned int board_index;


void close_and_exit ARG_1(int, error_code)
{
    DSP_unregister_host_app();
    exit(error_code);
}


void write_dual_port_memory ARG_2(UInt32, address, UInt32, value)
{
    int error;
    error = DSP_lock_board(board_index);
    if(error != DSP_NO_ERROR)
    {
        printf("Error: can't lock board  error = %d.\n\n",error);
        close_and_exit(5);
    }
    error = DSP_write_dual_port_memory(board_index, address, value);
    DSP_unlock_board(board_index);
    if(error != DSP_NO_ERROR)
    {
        printf("Error %d writing the DSP board's dual-port memory !\n\n",
               error);
        close_and_exit(5);
    }
}


void main ARG_2(int, argc, char **, argv)
{
    int error;

    printf("\nHDEM_2 - Step-Signal Generator, (C) dSPACE 1995\n\n");

#ifdef UNIX

    if(argc != 3)
    {
```

```
        printf("Usage: hdem_2 board client\n\n");
        exit(1);
    }
    error = DSP_net_connect(argv[2]);
    if(error)
    {
        printf("Error connecting to client %s\n\n", argv[2]);
        exit(1);
    }
#else

    if(argc != 2)
    {
        printf("Usage: HDEM_2 board\n\n");
        exit(1);
    }

#endif

    error = DSP_register_host_app("HDEM_2");
    if(error != DSP_NO_ERROR){
        switch(error){
#ifdef UNIX
  case DSP_NOT_CONNECTED_TO_DAEMON:
    printf("\nCan't connect to DSP daemon for %s.\n",argv[2]);
            break;
#else
          case DSP_DEVICE_DRIVER_NOT_FOUND:
            printf("\nDevice Driver not installed.\n");
            break;
          case DSP_VXD_NOT_LOADED:
            printf("\nVirtual device driver not installed.\n");
            break;
#endif
          case DSP_NO_FREE_HOST_APP_IDX:
            printf("\nNo free host application index.");
            break;
#ifdef NET
          case DSP_NET_ERROR:
            printf("\nNetwork error.");
            break;
#endif
          default:
            printf("\nDSP_register_host_app: error code %d\n",error);
            break;
        }
        exit(1);
    }
```

```
    error = DSP_board_index(argv[1],&board_index);
    if(error != DSP_NO_ERROR)
    {
        printf("\nBoard %s not registered, error = %d.\n",
               argv[1],error);
        close_and_exit(2);
    }

    /* initialize output signal with 0.0 */
    write_dual_port_memory(DP_MEM_OFFS,
                           DSP_cvt_ieee_to_ti((Float32) 0.0));

    /* set output signal to step-size */
    write_dual_port_memory(DP_MEM_OFFS,
                           DSP_cvt_ieee_to_ti((Float32) STEP_SIZE));

    printf("Press RETURN to abort ...\n");
    rewind(stdin);
    getchar();

    /* reset output signal to 0.0 */
    write_dual_port_memory(DP_MEM_OFFS,
                           DSP_cvt_ieee_to_ti((Float32) 0.0));

    close_and_exit(0);
}
```

6 System V Semaphores

Semaphores represent data structures used by the operating system kernel to synchronize processes. They are particularly useful in synchronizing the access of different processes to shared resources in a mutually exclusive manner. Semaphores are implemented in UNIX operating systems in a variety of ways. The following lectures discuss the System V implementation of semaphores and a introduce a simplified interface to these System V semaphores which was developed by A. Stevens. The use of both implementations will be demonstrated on a mutually exclusive file access application.

7 Synchronization using Semaphores

A critical section is a code segment that must be executed in a mutually exclusive manner. Code that modifies a shared variable usually has the following parts.

- Entry Section: The code that requests permission to modify the shared variable.

- Critical Section: The code that modifies the shared variable.
- Exit Section: the code that releases access.
- remainder section: the rest of the program.

The critical-section (CS) problem refers to the problem of executing critical sections in a fair, symmetric manner. Solutions to the critical section problem must satisfy

- mutual exclusion: At most one process is in its critical section at any time;
- progress: If no process is executing its critical section, a process that wishes to enter can get in. Only those processes that are not in their remainder section can participate in the decision to determine which process enters its critical section.
- Bounded Waiting: No process is postponed indefinitely. In addition there must be a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section.

The following program generates a process chain. After falling out of the forking loop, each process outputs an informative message to standard error, one character at a time. Since standard error is shared by all processes in the chain, that part of the code is a *critical section* and should be executed in a mutually exclusive manner. As written, the critical section in the following program is not protected. This means that the output of different processes can be interleaved. In other words, a process' access to standard error must be done in a mutually exclusive manner.

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

void main  (int argc, char *argv[])
{
    char buffer[MAX_CANON];
    char *c;
    int i;
    int n;
    pid_t childpid;

    if ( (argc != 2) || ((n = atoi(argv[1])) <= 0) ) {
       fprintf (stderr, "Usage: %s number_of_processes\n", argv[0]);
       exit(1);
    }

    for (i = 1; i < n;  ++i)
       if (childpid = fork())
```

```
        break;

    sprintf(buffer,
       "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
       i, (long)getpid(), (long)getppid(), (long)childpid);

    c = buffer;
       /****************start of critical section ********************/
    while (*c != '\0') {
       fputc(*c, stderr);
       c++;
    }
    fputc('\n', stderr);
       /*******************end of critical section ********************/

    exit(0);
}
```

The problem with the preceding example is that the actions in the critical section are not done in an **atomic** manner. An atomic operation is one that, once started, completes in a logically indivisable manner (i.e. without any other related instructions interleaved). Solutions to the critical section problem all require atomic operations. Atomic operations, however, cannot be coded from the user's program. The only way to ensure that a set of operations on a shared resource are carried out atomically is to write this structure into the kernel of the operating system. This is precisely what the special **semaphore** data structures do.

8 System V Semaphores

The semaphore described in the preceding section is an example of a binary semaphore. The System V implementation extends the binary semaphore in two directions.

1. A semaphore is a set of nonnegative integers.
2. The semaphore value can assume any nonnegative value.

For every set of semaphores in the system, th kernel maintains the following structure.

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

struct semid_ds{
    struct ipc_perm sem_perm;
    struct sem      *sem_base;
    ushort          sem_nsems;
```

```
    time_t          sem_otime;
    time_t          sem_ctive;
};
```

The ipc_perm structure contains the permissions for the semaphore. It has a structure similar to other permission structures.

```
struct ipc_perm{
    ushort  uid;        /*owner's user id*/
    ushort  gid;        /*owner's group id*/
    ushort cuid;        /*creator's user id*/
    ushort cgid;        /*creator's group id*/
    ushort mode;        /*access modes*/
    ushort  seq;        /*slot usage sequence number*/
    key_t   key;        /*key*/
}
```

The sem structure is a data structure used by the kernel to maintain the set of values for a given semaphore. Every member of a semaphore set is described by the following structure.

```
struct sem {
    ushort  semval;     /*semaphore value, nonnegative*/
    short   sempid;     /*pid of last operation*/
    ushort  semncnt;    /*number of waiting semval > cval*/
    ushort  semzcnt;    /*number of waiting semval = 0*/
};
```

The value of the semaphore is semval and semid is the pid of the last process to successfully modify the semaphore. The other two integers in the structure keep track of the number of processes waiting to increment the semaphore (semncnt) and the number of processes waiting for the semaphore to go to zero (semznt).

A semaphore is created on an existing semaphore is accessed with the semget system call.

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

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

The value returned by semget is the semaphore identifier, semid or -1 if an error occurred. The nsems argument specifies the number of semaphores in the set. We set this to zero if we are accessing a semaphore that has already been created. The semflag value is a combination of the constants shown below

Numeric	Symbolic	Description
0400	SEM_R	read by owner
0200	SEM_A	alter by owner
0040	SEM_R >> 3	read by group
0020	SEM_A >> 3	alter by group
0004	SEM_R >> 6	read by world
0002	SEM_A >> 6	alter by world
	IPC_CREAT	create semaphore
	IPC_EXCL	see details below

Once a semaphore set is opened with semget, operations are performed on one or more of the semaphore values in the set by using the semop system call. The prototype for this system call is shown below.

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

int semop(int semid, struct sembuf *opsptr, unsigned int nops);
```

The pointer opsptr points to an array of the following structure,

```
struct sembuf{
    ushort  sem_num;        /*semaphore number*/
    short   sem_op;         /* semaphore operation*/
    short   sem_flg;        /* operation flags*/
};
```

The number of elements in the array of sembuf structures point to by opsptr is specified by the nops argument. Each element in this array specifies an operation for one particular semaphore value in the set. The particular semaphore value is specified by the sem_num value, which is zero for the first element, one or the secon and so on, up to nsems, the value that was used to access or create the semaphore set.

The array of operations passed to the semop system call are guaranteed to be done atomically by the kernle. In other words, the kernel does either all of the operations or none of them. Each particular operation is specified by a sem_op value which can be negative, zero, or positive.

1. If sem_op is positive, the value of sem_op is added to the semaphore's current value. This corresponds to the relase of resources that the semaphore controls.
2. If sem_op is zero, then the caller wants to wait util the semaphore's value becomes zero.
3. If sem_op is negative, then the caller wants to wait until the semaphore's value becomes thereat than or equal to the absolute value of sem_op. Then the absolute value of sem_op is subtracted from the semaphore's current value. This corresponds to the allocation of resources.

The return value form semop is zero if all is okay, or -1 if an error occurred.

There are many options available to the semop system call. We can specify, for example, the IPC_NOWAIT flag for the sem_flg value associated with an operation. THis tells the system we don't have to wait if that operation can't be completed.

The system call semctl provides various control operations on the semaphore. The prototype declaration for this system call is

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

int semctl(int semid, int semnm, int cmd, union semun arg);

union semun{
    int                 val;    /*used for SETVAL only*/
    struct  semid_ds    *buff;  /*used for IPC_STAT and IPC_SET*/
    ushort              *array; /*used for GETALL and SETALL*/
} arg;
```

We'll use a *cmd* of IPC_RMID to remove a semaphore from the system. We'll also use the GETVAL and SETVAL to fetch and set a specific semaphore value. These two commands use the semnum argument to speicify one member of the semaphore set.

8.1 File Locking using System V Semaphores

There are occasions when multiple processes want to share some resource. It is essential that some for of mutual exclusion be provided so that only one process at a time accesses the resource. Consider the following scenario. Consider a line printer daemon. The process that places a job on the print queue has to assign a unique sequence number to each print job. We cannot use the process ID as the sequence nmber as it is possible for a print job to exist long enough for a given pid to be reused. The technique used by the UNIX printer spoolers is to have a file for each printer that contains the next sequence number to be used. The file is just a single line containing the sequence number in ASCII. Each process that needs to assign a sequence number goes through three steps.

1. it reads the sequence number file
2. it uses the number
3. it increments the number and writes it back.

The problem is that in the time it takes a single process to execute these three steps, another process an perform the same three steps. What is needed is for a process to be able to set a lock so that no other process can access the file until the first process has finished with the file. In other words we need to synchronize the access of the processes to the sequence number file.

Since semaphores are synchronization tools, we create a binary semaphore (it has the value zero or one) to control the access to the file. To lock the semaphore we call semop to do two operations atomically. First, we wait for the semaphore value to become zero, and then increment the value by one. This is an example where multiple semaphore operations must be done atomically by the kernel. If it took two system calls to do this (one to test and the other to set the semaphore), then the operation might be interrupted between the two system calls.

To unlock the resource we call semop to decrement the semaphore value. Since we have the lock on the resource, we know that the semaphore value is one before the call, so the call cannot wait. We explicitly set sem_flg to IPC_NOWAIT so that if this impossible condition does happen, then an error return occurs and the process aborts.

The following code segment shows the implementation of our semaphore synchronization,

```
/*
 *Locking routines using semaphores
 */

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

#define SEMKEY 123456L      /* key value for semget() */
#define PERMS  0666

static struct sembuf    op_lock[2]={
    0 , 0, 0        /*wait for sem#0 to become 0*/
    0 , 1, 0        /*increment sem#0 to 1 */
};

static struct sembuf op_unlock[1]={
    0, -1 , IPC_NOWAIT  /*decrement sem0 by 1 */
};

int semid = -1      /* semaphore id */

void my_lock(int fd){
    if (semid < 0) {
        if( (semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
            err_sys("semget error");
    }
    if (semop(semid, &op_lock[0],2)<0)
        err_sys("semop lock error");
}
```

```
void my_unlock(int fd){
    if(semop(semid, &op_unlock[0], 1) < 0)
        err_sys("semop unlock error");
}


void main(int argc, char **argv){
    int fd,i,n,pid,seqno;
    char buff[MAXBUFF+1];

    pid=getpid();
    if( (fd=open(SEQFILE,2)) < 0)
        perror("can't open %s",SEQFILE);

    for(i=0;i<20;i++){
        my_lock(fd);
        lseek(fd,0L,0);
        if( (n=read(fd,buff,MAXBUFF))<=0)
            perror("read error");

        buff[n]='\0';
        if( (n=sscanf(buff, "%d\n", &seqno))!=1)
            perror("sscanf error");

        printf("pid=%d, seqno=%d\n",pid,seqno);

        seqno++;

        sprintf(buff, "%03d\n",seqno);
        n=strlen(buff);
        lseek(fd,0L,0);

        if(write(fd,buff,n)!=n)
            perror("write error");
    my_unlock(fd);
    }

}
```

8.2 Simplified Semaphore Interface

A simpler set of functions have been devised for using system V semaphores. The basic routines and the function are given below.

```
int sem_create( key_t key, int intval);
    This creates an initial value or open
```

```
int sem_open(key_t key);
    opens an existing semaphore

void sem_wait(int id);
    wait = P = down by 1

void sem_signal(int id);
    signal = V = up by 1

void sem_op(int id, int amount);
    wait if (amount < 0)
    signal if (amount > 0)

void sem_close(int id);
    close

void sem_rm(int id);
    remove (delete)
```

We create and use a 3 member set for the requested semaphore. The first member, [0], is the actual semaphore value and the second member, [1], is a counter used to know when all processes have finished with the semaphore. The counter is initialized to a large number, decremented on every create or open and incremented on every close. This way we can use the "adjust" feature provided by system V so that any process that exits without calling sem_close can be accounted for. It doesn't help us if the last process does this (as we have have no way of getting control to remove the semaphore) but it will work if any other process does an exit The third member, [2], of the semaphore set is used as a lock variable to avoid any race conditions in sem_create() and sem_close() functions. The source code is attached in subsection 8.3.

8.3 Code for Simplified Interface

```
/*
 * Provide an simpler and easier to understand interface to the System V
 * semaphore system calls.
 */

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

#include <errno.h>
extern int errno;

#define BIGCOUNT 10000 /* initial value of process counter */
```

```
/*
 * Define the semaphore operation arrays for the semop() calls.
 */

static struct sembuf op_lock[2] = {
2, 0, 0, /* wait for [2] (lock) to equal 0 */
2, 1, SEM_UNDO /* then increment [2] to 1 - this locks it */
/* UNDO to release the lock if processes exits
   before explicitly unlocking */
};

static struct sembuf op_endcreate[2] = {
1, -1, SEM_UNDO,/* decrement [1] (proc counter) with undo on exit */
/* UNDO to adjust proc counter if process exits
   before explicitly calling sem_close() */
2, -1, SEM_UNDO /* then decrement [2] (lock) back to 0 */
};

static struct sembuf op_open[1] = {
1, -1, SEM_UNDO /* decrement [1] (proc counter) with undo on exit */
};

static struct sembuf op_close[3] = {
2, 0, 0, /* wait for [2] (lock) to equal 0 */
2, 1, SEM_UNDO, /* then increment [2] to 1 - this locks it */
1, 1, SEM_UNDO /* then increment [1] (proc counter) */
};

static struct sembuf op_unlock[1] = {
2, -1, SEM_UNDO /* decrement [2] (lock) back to 0 */
};

static struct sembuf op_op[1] = {
0, 99, SEM_UNDO /* decrement or increment [0] with undo on exit */
/* the 99 is set to the actual amount to add
   or subtract (positive or negative) */
};

/****************************************************************************
 * Create a semaphore with a specified initial value.
 * If the semaphore already exists, we don't initialize it (of course).
 * We return the semaphore ID if all OK, else -1.
 */

int
sem_create(key, initval)
key_t key;
```

```
int initval; /* used if we create the semaphore */
{
register int id, semval;
union semun {
int val;
struct semid_ds *buf;
ushort *array;
} semctl_arg;

if (key == IPC_PRIVATE)
return(-1); /* not intended for private semaphores */

else if (key == (key_t) -1)
return(-1); /* probably an ftok() error by caller */

again:
if ( (id = semget(key, 3, 0666 | IPC_CREAT)) < 0)
return(-1); /* permission problem or tables full */

/*
 * When the semaphore is created, we know that the value of all
 * 3 members is 0.
 * Get a lock on the semaphore by waiting for [2] to equal 0,
 * then increment it.
 *
 * There is a race condition here.  There is a possibility that
 * between the semget() above and the semop() below, another
 * process can call our sem_close() function which can remove
 * the semaphore if that process is the last one using it.
 * Therefore, we handle the error condition of an invalid
 * semaphore ID specially below, and if it does happen, we just
 * go back and create it again.
 */

if (semop(id, &op_lock[0], 2) < 0) {
if (errno == EINVAL)
goto again;
err_sys("can't lock");
}

/*
 * Get the value of the process counter.  If it equals 0,
 * then no one has initialized the semaphore yet.
 */

if ( (semval = semctl(id, 1, GETVAL, 0)) < 0)
err_sys("can't GETVAL");
```

```
if (semval == 0) {
/*
 * We could initialize by doing a SETALL, but that
 * would clear the adjust value that we set when we
 * locked the semaphore above.  Instead, we'll do 2
 * system calls to initialize [0] and [1].
 */

semctl_arg.val = initval;
if (semctl(id, 0, SETVAL, semctl_arg) < 0)
err_sys("can SETVAL[0]");

semctl_arg.val = BIGCOUNT;
if (semctl(id, 1, SETVAL, semctl_arg) < 0)
err_sys("can SETVAL[1]");
}

/*
 * Decrement the process counter and then release the lock.
 */

if (semop(id, &op_endcreate[0], 2) < 0)
err_sys("can't end create");

return(id);
}

/****************************************************************************
 * Open a semaphore that must already exist.
 * This function should be used, instead of sem_create(), if the caller
 * knows that the semaphore must already exist.  For example a client
 * from a client-server pair would use this, if its the server's
 * responsibility to create the semaphore.
 * We return the semaphore ID if all OK, else -1.
 */

int
sem_open(key)
key_t key;
{
register int id;

if (key == IPC_PRIVATE)
return(-1); /* not intended for private semaphores */

else if (key == (key_t) -1)
return(-1); /* probably an ftok() error by caller */
```

```
if ( (id = semget(key, 3, 0)) < 0)
return(-1); /* doesn't exist, or tables full */

/*
 * Decrement the process counter.  We don't need a lock
 * to do this.
 */

if (semop(id, &op_open[0], 1) < 0)
err_sys("can't open");

return(id);
}

/****************************************************************************
 * Remove a semaphore.
 * This call is intended to be called by a server, for example,
 * when it is being shut down, as we do an IPC_RMID on the semaphore,
 * regardless whether other processes may be using it or not.
 * Most other processes should use sem_close() below.
 */

sem_rm(id)
int id;
{
if (semctl(id, 0, IPC_RMID, 0) < 0)
err_sys("can't IPC_RMID");
}

/****************************************************************************
 * Close a semaphore.
 * Unlike the remove function above, this function is for a process
 * to call before it exits, when it is done with the semaphore.
 * We "decrement" the counter of processes using the semaphore, and
 * if this was the last one, we can remove the semaphore.
 */

sem_close(id)
int id;
{
register int semval;

/*
 * The following semop() first gets a lock on the semaphore,
 * then increments [1] - the process counter.
 */

if (semop(id, &op_close[0], 3) < 0)
```

```
err_sys("can't semop");

/*
 * Now that we have a lock, read the value of the process
 * counter to see if this is the last reference to the
 * semaphore.
 * There is a race condition here - see the comments in
 * sem_create().
 */

if ( (semval = semctl(id, 1, GETVAL, 0)) < 0)
err_sys("can't GETVAL");

if (semval > BIGCOUNT)
err_dump("sem[1] > BIGCOUNT");
else if (semval == BIGCOUNT)
sem_rm(id);
else
if (semop(id, &op_unlock[0], 1) < 0)
err_sys("can't unlock"); /* unlock */
}

/****************************************************************************
 * Wait until a semaphore's value is greater than 0, then decrement
 * it by 1 and return.
 * Dijkstra's P operation.  Tanenbaum's DOWN operation.
 */

sem_wait(id)
int id;
{
sem_op(id, -1);
}

/****************************************************************************
 * Increment a semaphore by 1.
 * Dijkstra's V operation.  Tanenbaum's UP operation.
 */

sem_signal(id)
int id;
{
sem_op(id, 1);
}

/****************************************************************************
 * General semaphore operation.  Increment or decrement by a user-specified
 * amount (positive or negative; amount can't be zero).
```

```
 */

sem_op(id, value)
int id;
int value;
{
if ( (op_op[0].sem_op = value) == 0)
err_sys("can't have value == 0");

if (semop(id, &op_op[0], 1) < 0)
err_sys("sem_op error");
}
```

8.4 File Locking using Simplified Semaphore Interface

We can use these simpler semaphore functions in our locking example. HEre we have changed the main function from the version that was used in the previous examples so that the sem_create and sem_close are outside the main loop.

```
/*
 * Locking example using the simpler semaphore operations
 */

#include < sys/types.h>

#define SEQFILE "seqno"
#define SEMKEY  ((key_t) 23456L)
#define MAXBUFF 100

main(){
    int     fd,i,n,pid,seqno,semid;
    char    buff[MAXBUFF];

    pid=getpid();
    if( (fd=open(SEQFILE, 2)) < 0)
        perror("can't open%s",SEQFILE);
    if( (semid=sem_createSEMKEY,1)) < 0)
        perror("can't open semaphore");

    for (i=0;i<20;i++){
        sem_wait(semid);
        lseek(fd, 0L, 0);       /*rewind before read*/
        if( (n=read(fd, buff, MAXBUFF)) <=0)
            perror("read error");
        buff[n]='\0';
```

```
        if( (n=sscanf(buf, "%d\n", &seqno)) != 1)
            perror("sscanf error");

        printf("pid=%d, seqno=%d\n",pid,seqno);

        seqno++;

        sprintf(buff, "%03d\n", seqno);
        n=strlen(buff);
        lseek(fd,0L,0);
        if (write(fd,buff,n)!=n)
            perror("write_error");

        sem_signal(semid);
    }
    sem_close(semid);
}
```

8.5 Exercise

The exercise for this lecture builds upon the timer project. In the timer project, you spawned two child processes that require listen for changes in a special location in the dSpace boxes' memory, then notify the parent process of the change, and then reset the memory location. In this project, we will assume that the miniboxes are used to coordinate the access of 2 robots to a shared resource.

Consider two robots who must obtain components from a parts bin and move the components to a work area where an assembly operation takes place. The task of fetching the workpiece, transporting it to the work area, and then returning to get another workpiece is performed in a periodic manner. As illustrated in figure 11, however, it is assumed that the parts bin is common to both robots. This shared resource immediately introduces a mutual exclusion requirement on the entire system since both robots cannot enter the parts bin at the same time. The problem that is involved here, therefore, is concerned with coordination of the individual robot's motions so that the task is completed (in a timely manner) while respecting the mutual exclusion condition on the parts bin.

To coordinate the action of both systems it is essential that information be passed back and forth between the robots over their LAN connection. Binary semaphores are the most common synchronization structures used in such situations. The LAN connection, however, takes about 5 milliseconds to establish over a standard TCP/IP network and during this period it is possible for both robots to obtain a lock on the semaphore. As a result, both robots will attempt to access the parts bin at the same time.

One solution to this problem, is to use a semaphore which is controlled by one of the supervisory workstations. The workstation controls the access of the dSpace boxes to the semaphore in a mutually exclusive manner.

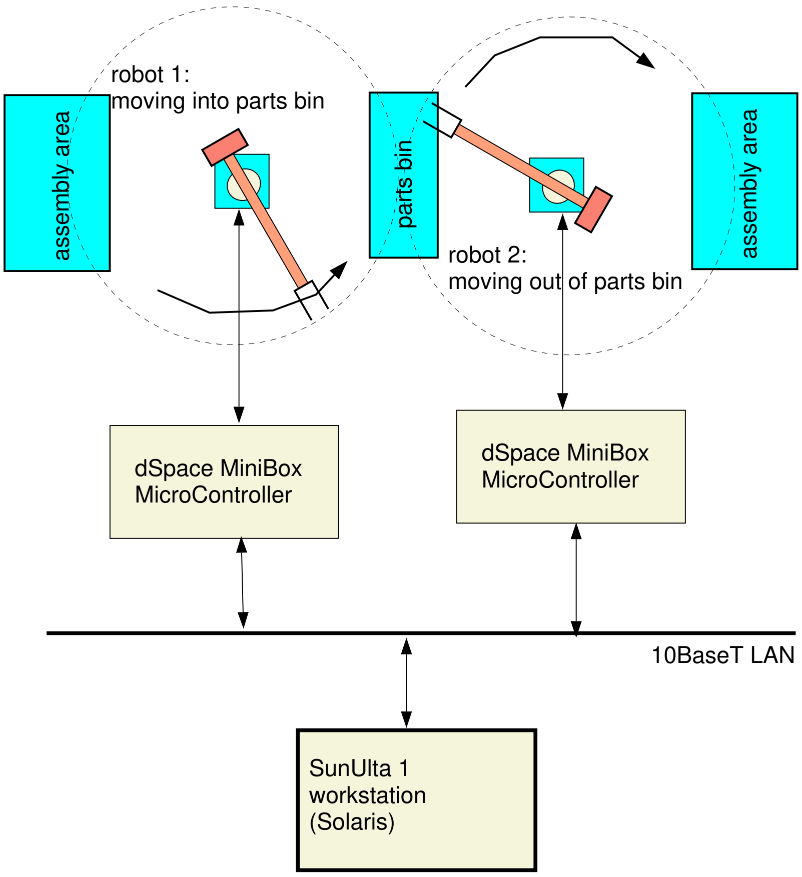


Figure 11: Manufacturing System with a Mutual Exclusion Constraint

In this project, you are asked to add the code to your previous timer project, that control access of the dSpace boxes to the parts bin.

9 Sockets and Network Programming

These lectures discuss network programming using the Berkeley sockets.

9.1 Network Primer

A **computer network** is a communication system for connecting computer systems. We often refer to each of these systems as **hosts**. A **local area network** (LAN) connects computer systems that are close together. Popular LAN's are ethernet and token rings. LAN's operate at high speeds. An ethernet operates at 10-100 Mbps while IBM's token ring operates at 4 and 16 Mbps. The fiber distributed data interface (FDDI) operates at 100 Mbps. Each computer on a LAN has an interface card that connects to computer to the network hardware. An **internet** is a connection of two or more distinct networks (LAN). A **gateway** or **router** is used to pass information between networks.

The standard model for network applications is the **client-server** model. A **server** is a proticess that is waiting to be contacted by a **client** process so that the server can do something for the client. A typical (but not mandatory) scenario is

- The server process is started on some computer system. It initializes itself and then goes to sleep waiting for the client to contact it.
- A client process is started. The client sends a request across the network to the serve requesting service of some form.
- The server process has finished providing the service to the lcient and the server goes back to sleep, waiting for the next client request to arrive.

Computers use **protocols** (systematic rules) to communicate. These protocols can be complex and they are designed in layers to make it easier to maintain and design. The typical Open System Interconnect (OSI) model is shown below in figure 12. This model consists of 7 layers of protocols which a message needs to go through before a message at the computer can reach the network hardware. In a similar way, messages received at the network level, must go through these same 7 layers before being delivered to the computer.

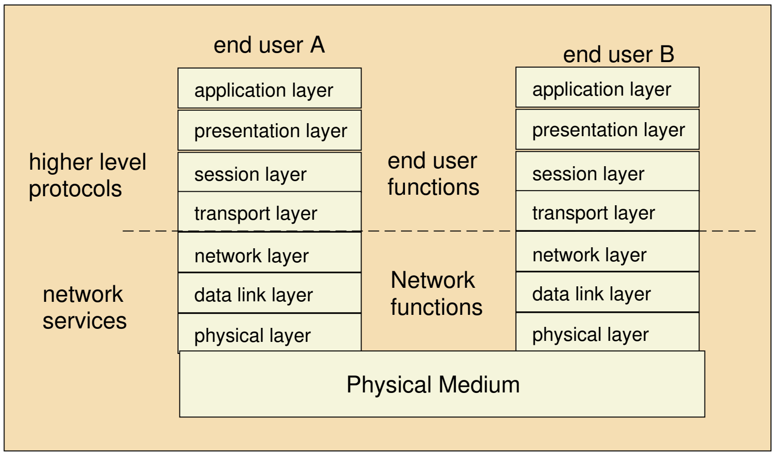


Figure 12: OSI model

These layers can be divided into two groups; protocols providing network services and higher level protocols. The network protocols deal with the specific associated with transmitting data over a particular type of network (ethernet, myrinet, FDDI). The higher-level protocols provide for services that ensure that messages are transmitted error free and in the proper format for the receiving computer. The bottom three layers (physical, data link, and network) provide the network services. The top four levels (transport, session, presentation, and application) form the user services which are generally resident on the host computer.

Layering provides well-defined interfaces between the layers, so that a change in one layer does not effect an adjacent layer. There are a variety of protocol suites available. Some of these are

- the TCP/IP protocol suite
- Xerox Network Systems (XNS)
- IBM's Systems Network Architecture (SNA)
- IBM's NetBIOS
- the OSI protocols
- UUCP

In our work we will be using the TCP/IP protocol suite.

9.2 TCP/IP Protocol Suite

During the late 1960's and 1970's the Advanced Research Projects Agency (ARPA) sponsored the development of ARPANET, which has since become known as the Internet. The DARPA set of internet protocols become commonly referred to as TCP/IP.

The TCP/IP protocol suite consists of the following

- TCP (Transmission Control Protocol): A connection-oriented protocol that provides a reliable, full-duplex, byte stream for a user process.
- UDP (User Datagram Protocol): A connectionless protocol for user processes. (unlike TCP, there is no guarantee of delivery with this).
- ICMP (Internet Control Message Protocol): The protocol to handle error and control information between gateways and hosts.
- IP (Internet Protocol): IP is the protocol that provides the packet delivery service for TCP, UDP, and ICMP.
- ARP (Address Resolution Protocol): The protocol that maps an internet address into a hardware address.
- RARP (Reverse Address REsolution Protocol): The protocol that maps a hardware address into an internet address.

We can simplify the 7 layer OSI standard to 4 different levels. These are the data link layer, ntework layer, transport layer and application layer. Each of these is defined below.

Data-link Layer

The datalink layer is the physical network. It consists of a collections of computers connected over several leased telephone lines.

Network Layer

The IP layer provides a connectionless and unreliable delivery system. It is connectionless because all datagrams are treated independently of the others. It is unreliable because it does not guarantee that the packets are delivered. Reliability must be provided by upper layers.

The IP layer handles routing through the internet. It is also responsible for fragmenting long messages into constant length datagrams. The IP layer also provides an elementary form of flow control. When IP packets arrive at a host or gateway so fst that they are discarded the IP module sends a message telling the source to slow down.

Transport Layer

User processes interact with the TCP/IP protocol suite by sending and receiving either TCP or UDP data. The transport layer handles these services. TCP provides a connection oriented reliable full-duplex, byte stream service to an application. UDP provides a connectionless unreliable datagram service.

Application Layer

There are several application programs that are provided with almost every TCP/IP implementation. Some of these well known services are listed below.

- FTP - File Transfer Protocol
- TFTP - Trivial File Transfer Protocol
- TELNET - Remote Login
- SMTP - Simple Mail Transfer Protocol

9.3 Berkeley Sockets

This section describes an **application program interface** (API) to the communication protocols known as **sockets**. The API is an interface available to the programmer.

Figure 13 shows a time line of the typical scenario that takes place for a connection-oriented transfer. First the server is started, then sometime later a client is started that connects to the server.

For a client-server using a connectionless protocol, the system calls are different. Figure 14 shows these system calls. The client does not establish a connection to the server. Instead, the client sends a datagram to the server using the sendto system call which requires the address of the server as a parameter. Similar, the server does

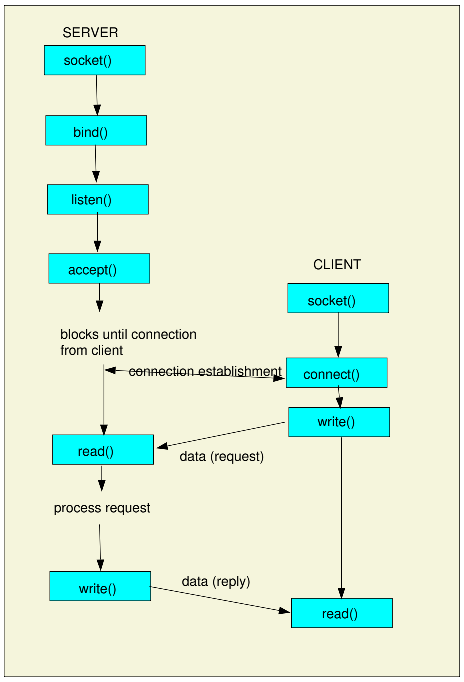


Figure 13: Socket Calls for Connection Oriented Protocol

not have to accept a connection from a client. Instead, the server issues a recvfrom system call that waits until data arrives form some client.

The use of the original sockets is somewhat involved. In the next section we'll use a simplified API which is called by Robbins the Universal Internet Communication Interface (UICI).

9.4 Socket System Calls

The elementary socket system calls are itemized below.

9.4.1 socket()

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

int socket(int family, int type, int protocol);
```

The **int family** takes on values of

- AF_UNIX Unix internal protocols

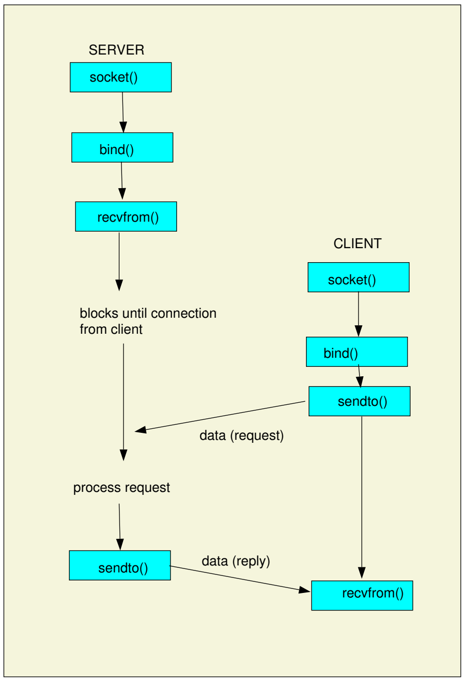


Figure 14: Socket Calls for Connectionless Oriented Protocol

- AF_INET internet protocols
- AF_NS Xerox NS protocols
- AF_IMPLINK IMP link layer

The **int socket** type takes value of

- SOCK_STREAM stream socket
- SOCK_DGRAM datagram socket
- SOCK_RAW raw socket
- SOCK_SEQPACKET sequence packet socket

The **protocol** argument is usually set to zero.

The socket system call returns an integer value called the socket descriptor sockfd which is similar to a file descriptor. This call is the first thing done to do network I/O.

9.4.2 socektpair()

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

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

int socketpair(int family, int type, int protocol, int sockvec[2]);
```

This call returns two socket descriptors, sockvec[0] and sockvec[1], that are unnamed and connected. This system call is similar to the pipe system call. The structure created by this call is called a **stream pipe**.

9.4.3 bind()

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

int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

This system call assigns a name to an unnamed socket. The second argument myaddr is ap ointer to a protocol-specific address and the third argument is the size of this address structure. There are three uses of bind.

1. servers register their well-known address with the system.
2. a client regesters a specific address for itself
3. A connectionless client needs to assure that the system assigns it some unique address.

9.4.4 connect()

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

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

This routine connects a socket descriptor follwing the socket system call to establish a connection with a server. The sockfd descriptor is the socket descriptor returned by the socket system call. The second and third arguments are pointer to the server socket address and its associated length.

9.4.5 listen()

This system call is used by a connection-oriented server to
indicate that it is willing to receive connections.

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

```
int listen(int sockfd, int backlog)
```

This system call is usually executed after both the socket and bind system calls and immediately before the accept system call. The backlog arguement specifies how many connection requests can be queued by the system while it waits to execute the accept system call. This argument is often set to 5.

9.4.6 accept()

After a connection oriented server exectutes the listen system call, an actual connection for a client process is waited for by having the server do the accept system call

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

int accept(int sockfd, struct sockaddr *peer, int *addrlen);
```

accept takes the first connection request on the queue and creates another socket with the same properties as sockfd. If there are no connection requests pending, then this call blocks the caller until one arrives.

The peer and addrlen arguments are used to return the address of the connected client. addrlen is called a value-result argument. The caller sets its value before the system call and the system call stores a result in the variable.

9.4.7 send, sendto, recv, recvfrom

These are similar to the standard read and write system calls.

```
int send(int sockfd, char *buff, int nbytes, int flags);

int sendto(int sockfd, char *buff, int nbytes, int flags,
struct sockaddr *to, int addrlen);

int recv(int sockfd, char *buff, int nbytes, int flags);

int recvfrom(int sockfd, char *buff, int nbytes, int flags,
struct sockaddr *from, int addrlen);
```

These calls require additional arguemnts. The flags arument is either zero or is formed by Ors of the following constants

MSG_OOB send or receive out of band data

MSG_PEEK peek at incoming message
MSG_DONTROUTE bypass routing

9.4.8 close()

```
int close(int fd);
```

The usual Unix close command is used to close a socket.

9.5 Example

This sections provides 4 examples illustrating the use of sockets. These examples are

9.5.1 read n bytes from a descriptor

```
/*
*read n bytes from a descriptor
*use in place of read() when fd is a stream socket
*/

int readn(register int fd,
  register char *ptr,
  register int nbytes)
{
int nleft, nread;

nleft=nbytes;
while (nleft > 0){
nread=read(fd,ptr,nleft);
if(nread < 0)
return(nread);
else if (nread==0)
break;
nleft-=nread;
ptr  +=nread;
}
return(nbytes - nleft);
}
```

9.5.2 write n bytes to a descriptor

```
/*
*write ``n'' bytes to a descriptor
```

```
*use in place of write() when fd is a stream socket
*/

int writen(register int fd,
   register char *ptr,
           register int nbytes)
{
int nleft, nwritten;

nleft=nbytes;
while(nleft>0){
nwritten=write(fd,ptr,nleft);
if(nwritten<=0)
return(nwritten);
nleft -= nwritten;
ptr   += nwritten;
}
return(nbytes-nleft);
}
```

9.5.3 TCP example

This TCP example follows the flow of system calls diagrammed earlier for a connection oriented server and client. The header file required for both programs (server and client) is

```
/*
 * Definitions for TCP and UDP client/server programs.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERV_UDP_PORT 6000
#define SERV_TCP_PORT 6000
#define SERV_HOST_ADDR "192.43.235.6" /* host addr for server */

char *pname;
```

the server code is

```
/*
 * Example of server using TCP protocol.
```

```
 */

#include "inet.h"

main(argc, argv)
int argc;
char *argv[];
{
int sockfd, newsockfd, clilen, childpid;
struct sockaddr_in cli_addr, serv_addr;

pname = argv[0];

/*
 * Open a TCP socket (an Internet stream socket).
 */

if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
err_dump("server: can't open stream socket");

/*
 * Bind our local address so that the client can send to us.
 */

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port        = htons(SERV_TCP_PORT);

if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
err_dump("server: can't bind local address");

listen(sockfd, 5);

for ( ; ; ) {
/*
 * Wait for a connection from a client process.
 * This is an example of a concurrent server.
 */

clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
     &clilen);
if (newsockfd < 0)
err_dump("server: accept error");

if ( (childpid = fork()) < 0)
err_dump("server: fork error");
```

```
else if (childpid == 0) { /* child process */
close(sockfd); /* close original socket */
str_echo(newsockfd); /* process the request */
exit(0);
}

close(newsockfd); /* parent process */
}
}
```

The client code is:

```
/*
 * Example of client using TCP protocol.
 */

#include "inet.h"

main(argc, argv)
int argc;
char *argv[];
{
int sockfd;
struct sockaddr_in serv_addr;

pname = argv[0];

/*
 * Fill in the structure "serv_addr" with the address of the
 * server that we want to connect with.
 */

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port        = htons(SERV_TCP_PORT);

/*
 * Open a TCP socket (an Internet stream socket).
 */

if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
err_sys("client: can't open stream socket");

/*
 * Connect to the server.
 */
```

```
if (connect(sockfd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr)) < 0)
err_sys("client: can't connect to server");

str_cli(stdin, sockfd); /* do it all */

close(sockfd);
exit(0);
}
```

9.5.4 UDP example

The UDP client server programs are shown below and follow the function calls diagrammed earlier.

The server code is:

```
/*
 * Example of server using UDP protocol.
 */

#include "inet.h"

main(argc, argv)
int argc;
char *argv[];
{
int sockfd;
struct sockaddr_in serv_addr, cli_addr;

pname = argv[0];

/*
 * Open a UDP socket (an Internet datagram socket).
 */

if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
err_dump("server: can't open datagram socket");

/*
 * Bind our local address so that the client can send to us.
 */

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
```

```
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port        = htons(SERV_UDP_PORT);

if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
err_dump("server: can't bind local address");

dg_echo(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));

/* NOTREACHED */
}
```

The client code is:

```
/*
 * Example of client using UDP protocol.
 */

#include "inet.h"

main(argc, argv)
int argc;
char *argv[];
{
int sockfd;
struct sockaddr_in cli_addr, serv_addr;

pname = argv[0];

/*
 * Fill in the structure "serv_addr" with the address of the
 * server that we want to send to.
 */

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port        = htons(SERV_UDP_PORT);

/*
 * Open a UDP socket (an Internet datagram socket).
 */

if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
err_dump("client: can't open datagram socket");

/*
 * Bind any local address for us.
```

```
 */

bzero((char *) &cli_addr, sizeof(cli_addr)); /* zero out */
cli_addr.sin_family      = AF_INET;
cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
cli_addr.sin_port        = htons(0);
if (bind(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0)
err_dump("client: can't bind local address");

dg_cli(stdin, sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

close(sockfd);
exit(0);
}
```

9.6 Sockets and Signals

There are three signals that can be generatd for a socket.

(SIGIO) This signal indicates that a socket is ready for asynchronous I/O. The signal is sent to the process group of the socket. This progress group is established by calling ioctl with a command of either FIOSETOWN or SIOCSPGRP, or by calling fcntl with a command of F_SETOWN. The signal is sent ot the process group only if the process has enabled asynchronous I/O on the socket by calling ioctl with a command of FIOASYNC or by calling fcntl with an argument of FASYNC.

(SIGURG) This signal indicates the presence of an urgent condition. An urgent condition is either the arrival of out of band data on the socket. Out of band data is information that arrives at the user process outside of the I/O buffer. Examples are of such data include interrupt signals.

(SIGPIPE) This signal indicates that we can no longer write to a socket, pipe, or FIFO.

9.7 Asynchronous I/O

Asynchronous I/O allows the process to tell the kernel to notify it when a specified descriptor is ready for I/O. It is also called signal driven I/O. The notification from the kernel to the user process takes place with a signal, SIGIO.

To do asynchronous I/O a process must

1. establish a handler for the SIGIO signal.
2. set the process ID or the process group ID to recieve the SIGIO signals. This is done with the fcntl command.

3. The process must enable asynchronous I/O using fcntl system call.

9.8 Universal Interface Communication Interface (UICI)

The sockets provide a complex API for the network programmer. A simplified API was presented by Robbins and Robbins in their textbook on advanced UNIX programming. This interface consists of the following primitives.

1. int u_open(u_port_t, port)

   Opens file descriptor bound to port. Returns listening file descriptor.
2. int u_listen(int fd, char *hostn)

   listens for connection request on fd
3. int u_close(fd)

   closes file descriptor fd.
4. ssize_t u_read(int fd, char *buf, size_t nbyte)

   reads up to nbyte bytes from fd into buf. Returns number of bytes actually read.
5. ssize_t u_write(int fd, char *buf, size_t nbyte)

   writes nbyte bytes from buf to fd. Returns number of bytes actually written.
6. void u_error(char *errmsg)

   outputs errmsg followed by a UICI error message.
7. int u_sync(int fd)

   updates relevant kernel information after calls to exec

The following program copies data from a network communication file descriptor to a file.

```
/* Program 12.1 */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include "uici.h"
#define BLKSIZE 1024
int copy_from_network_to_file(int communfd, int filefd)
{
   int bytes_read;
   int bytes_written;
   int bytes_to_write;
   int total_bytes = 0;
   char buf[BLKSIZE];
   char *bufp;
```

```
   for ( ; ; ) {
      if ((bytes_read = u_read(communfd, buf, BLKSIZE)) < 0) {
         u_error("Server read error");
         break;
      } else if (bytes_read == 0) {
         fprintf(stderr, "Network end-of-file\n");
         break;
      } else {    /* allow for interruption of write by signal */
         for (bufp = buf, bytes_to_write = bytes_read;
            bytes_to_write > 0;
            bufp += bytes_written, bytes_to_write -= bytes_written){
            bytes_written = write(filefd, bufp, bytes_to_write);
            if ((bytes_written) == -1 && (errno != EINTR)){
               perror("Server write error");
               break;
            } else if (bytes_written == -1)
               bytes_written = 0;
            total_bytes += bytes_written;
         }
         if (bytes_written == -1)
            break;
      }
   }
   return total_bytes;
}
```

A UICI server is organized as follows:

- open a well-known listening port
- listen for a connection request
- communicate directly or through a surrogate (child or thread) with the client using the comm file descriptor
- close the commm file descriptor

The following program shows a serial-server program for copying files from a client using the UICI library.

```
/* Program 12.2 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include "uici.h"
int copy_from_network_to_file(int communfd, int filefd);
```

```
/*
 *                        UICI Server
 *  Open a UICI port specified as a command-line argument
 *  and listen for a request.  When a request
 *  arrives, use the provided communication file descriptor to
 *  read from the UICI connection and echo to standard output
 *  until the connection is terminated.  Then the server resumes
 *  listening for additional requests.
 */
void main(int argc, char *argv[])
{
   u_port_t portnumber;
   int listenfd;
   int communfd;
   char client[MAX_CANON];
   int bytes_copied;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s port\n", argv[0]);
      exit(1);
   }
   portnumber = (u_port_t) atoi(argv[1]);
   if ((listenfd = u_open(portnumber)) == -1) {
      u_error("Unable to establish a port connection");
      exit(1);
   }
   while ((communfd = u_listen(listenfd, client)) != -1) {
      fprintf(stderr, "A connection has been made to %s\n",client);
      bytes_copied = copy_from_network_to_file(communfd, STDOUT_FILENO);
      fprintf(stderr, "Bytes transferred = %d\n", bytes_copied);
      u_close(communfd);
   }
   exit(0);
}
```

The UICI clients are organized as follows:

- connect to a specified host and port.
- communicate with the server using the comm file descriptor.
- close the comm file descriptor.

The following program shows the client side of the file copy program.

```
/* Program 12.4 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include "uici.h"

#define BLKSIZE 1024

/*
 *                      UICI Client
 *  Make a UICI connection request to the host and port specified
 *  as command-line arguments.  Read from standard input and
 *  write to the UICI communication file descriptor until
 *  end-of-file.
 */
void main(int argc, char *argv[])
{
   u_port_t portnumber;
   int communfd;
   ssize_t bytesread;
   char buf[BLKSIZE];

   if (argc != 3) {
      fprintf(stderr, "Usage: %s host port\n", argv[0]);
      exit(1);
   }
   portnumber = (u_port_t)atoi(argv[2]);
   if ((communfd = u_connect(portnumber, argv[1])) < 0) {
      u_error("Unable to establish an Internet connection");
      exit(1);
   }
   fprintf(stderr, "A connection has been made to %s\n",argv[1]);
   for ( ; ; ) {
      if ((bytesread = read(STDIN_FILENO, buf, BLKSIZE)) < 0) {
         perror("Client read error");
         break;
      } else if (bytesread == 0) {
         fprintf(stderr, "Client detected end-of-file on input\n");
         break;
      } else if (bytesread !=
                  u_write(communfd, buf, (size_t)bytesread)) {
         u_error("Client write_error");
         break;
      }
   }
   u_close(communfd);
   exit(0);
```

```
}
```

9.8.1 UICI Header files

```
/* Program B.1 */
/********************************* uici.h *************************/
/*   various definitions required for the UICI functions          */
/******************************************************************/
typedef unsigned short u_port_t;
int u_open(u_port_t port);
int u_listen(int fd, char *hostn);
int u_connect(u_port_t port, char *inetp);
int u_close(int fd);
ssize_t u_read(int fd, void *buf, size_t nbyte);
ssize_t u_write(int fd, void *buf, size_t nbyte);
void u_error(char *s);
int u_sync(int fd);
```

9.8.2 A client program for testing UICI

```
/* Program B.2 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include "uici.h"

#define BLKSIZE  1024

void usr1handler(int s)
{
   fprintf(stderr, "SIGUSR1 signal caught\n");
}

void usr2handler(int s)
{
   fprintf(stderr, "SIGUSR2 signal caught\n");
}


void installusrhandlers()
{
    struct  sigaction  newact;
    newact.sa_handler = usr1handler;      /* set the new usr1 handler */
```

```
    sigemptyset(&newact.sa_mask);    /* no additional signals blocked */
    newact.sa_flags = 0;            /* nothing special on the options */
    if (sigaction(SIGUSR1, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR1 signal handler");
       return;
    }
    newact.sa_handler = usr2handler;      /* set the new usr2 handler */
    if (sigaction(SIGUSR2, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR2 signal handler");
       return;
    }
    fprintf(stderr,
       "Client process %ld set to use SIGUSR1 and SIGUSR2\n",
       (long)getpid());
}

void main(int argc, char *argv[])
/*
 *  This is a client test of UICI. It opens a connection
 *  to a machine specified by host and a port number.
 *  It reads a file from stdin in blocks of size BLKSIZE
 *  and outputs it to the connection.
 */
{
    u_port_t portnumber;
    int outfd;
    ssize_t bytesread;
    ssize_t byteswritten;
    char buf[BLKSIZE];

    if (argc != 3) {
        fprintf(stderr, "Usage: %s host port\n", argv[0]);
        exit(1);
    }

    installusrhandlers();

    portnumber = (u_port_t)atoi(argv[2]);

    if ((outfd = u_connect(portnumber, argv[1])) < 0) {
       u_error("Unable to establish an Internet connection");
       exit(1);
    }

    fprintf(stderr, "Connection has been made to %s\n", argv[1]);

    for ( ; ; ) {
       bytesread = read(STDIN_FILENO, buf, BLKSIZE);
```

```
      if ( (bytesread == -1) && (errno == EINTR) )
         fprintf(stderr, "Client restarting read\n");
      else if (bytesread <= 0) break;
      else {
         byteswritten = u_write(outfd, buf, bytesread);
         if (byteswritten != bytesread) {
            fprintf(stderr,
               "Error writing %ld bytes, %ld bytes written\n",
               (long)bytesread, (long)byteswritten);
            break;
         }
      }
   }
   u_close(outfd);
   exit(0);
}
```

9.8.3 A server program for testing UICI

```
/* Program B.3 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/uio.h>
#include "uici.h"

#define BLKSIZE  1024
void usr1handler(int s)
{
   fprintf(stderr, "SIGUSR1 signal caught by server\n");
}

void usr2handler(int s)
{
   fprintf(stderr, "SIGUSR2 signal caught by server\n");
}

void installusrhandlers()
{
    struct  sigaction  newact;
    newact.sa_handler = usr1handler;      /* set the new usr1 handler */
    sigemptyset(&newact.sa_mask);    /* no additional signals blocked */
    newact.sa_flags = 0;            /* nothing special on the options */
```

```
    if (sigaction(SIGUSR1, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR1 signal handler");
       return;
    }
    newact.sa_handler = usr2handler;      /* set the new usr2 handler */
    if (sigaction(SIGUSR2, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR2 signal handler");
       return;
    }
    fprintf(stderr,
       "Server process %ld set to use SIGUSR1 and SIGUSR2\n",
       (long)getpid());
}

void main(int argc, char *argv[])
/*
 *  This is a server which opens a connection to a
 *  port number and listens for a request.  It then
 *  opens a communication port and reads from the connection
 *  until the connection is terminated.   Each block that is
 *  read is echoed to standard out.
 */
{
    u_port_t portnumber;
    int listenfd;
    int communfd;
    ssize_t bytesread;
    ssize_t byteswritten;
    char buf[BLKSIZE];
    char remote[MAX_CANON];

    if (argc != 2) {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(1);
    }

    installusrhandlers();

    portnumber = (u_port_t)atoi(argv[1]);

    if ((listenfd = u_open(portnumber)) < 0) {
       u_error("Unable to establish a port connection");
       exit(1);
    }

    if ((communfd = u_listen(listenfd, remote)) < 0) {
       u_error("Failure to listen on server");
       exit(1);
```

```
    }
    fprintf(stderr, "Connection has been made to %s\n", remote);

    while( (bytesread = u_read(communfd, buf, BLKSIZE)) > 0) {
       byteswritten = write(STDOUT_FILENO, buf, bytesread);
       if (bytesread != byteswritten) {
          fprintf(stderr,
             "Error writing %ld bytes, %ld bytes written\n",
             (long)bytesread, (long)byteswritten);
          break;
       }
    }
    u_close(listenfd);
    u_close(communfd);
    exit(0);
}
```

9.8.4 A complete implementation of UICI in terms of sockets

```
/* Program B.4 */
/* uici.c  sockets implementation */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <errno.h>
#include "uici.h"

#define MAXBACKLOG 5

/* return 1 if error, 0 if OK */
int u_ignore_sigpipe()
{
   struct sigaction act;

   if (sigaction(SIGPIPE, (struct sigaction *)NULL, &act) < 0)
      return 1;
   if (act.sa_handler == SIG_DFL) {
      act.sa_handler = SIG_IGN;
      if (sigaction(SIGPIPE, &act, (struct sigaction *)NULL) < 0)
         return 1;
```

```
   }
   return 0;
}

/*
 *                           u_open
 * Return a file descriptor which is bound to the given port.
 *
 * parameter:
 *       s = number of port to bind to
 * returns:  file descriptor if successful and -1 on error
 */
int u_open(u_port_t port)
{
   int sock;
   struct sockaddr_in server;

   if ( (u_ignore_sigpipe() != 0) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
      return -1;

   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = htons((short)port);

   if ( (bind(sock, (struct sockaddr *)&server, sizeof(server)) < 0) ||
        (listen(sock, MAXBACKLOG) < 0) )
      return -1;
   return sock;
}

/*
 *                           u_listen
 * Listen for a request from a particular host on a specified port.
 *
 * parameters:
 *      fd = file descriptor previously bound to listening port
 *      hostn = name of host to listen for
 * returns:  communication file descriptor or -1 on error
 *
 * comments: This function is used by the server to listen for
 * communication.  It blocks until a remote request is received
 * from the port bound to the given file descriptor.
 * hostn is filled with an ASCII string containing the remote
 * host name.  It must point to s string which is large enough to
 * hold this name.
 */
int u_listen(int fd, char *hostn)
```

```
{
   struct sockaddr_in net_client;
   int len = sizeof(struct sockaddr);
   int retval;
   struct hostent *hostptr;

   while ( ((retval =
           accept(fd, (struct sockaddr *)(&net_client), &len)) == -1) &&
          (errno == EINTR) )
      ;
   if (retval == -1)
      return retval;
   hostptr =
      gethostbyaddr((char *)&(net_client.sin_addr.s_addr), 4, AF_INET);
   if (hostptr == NULL)
      strcpy(hostn, "unknown");
   else
      strcpy(hostn, (*hostptr).h_name);
   return retval;
}

/*
 *                           u_connect
 * Initiate communication with a remote server.
 *
 * parameters:
 *     port  = well-known port on remote server
 *     inept = character string giving the Internet name of the
 *             remote machine
 * returns: the file descriptor used for communication or -1 if error
 */
int u_connect(u_port_t port, char *hostn)
{
   struct sockaddr_in server;
   struct hostent *hp;
   int sock;
   int retval;

   if ( (u_ignore_sigpipe() != 0) ||
        !(hp = gethostbyname(hostn)) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
      return -1;

   memcpy((char *)&server.sin_addr, hp->h_addr_list[0], hp->h_length);

   server.sin_port = htons((short)port);
   server.sin_family = AF_INET;
```

```
   while ( ((retval =
       connect(sock, (struct sockaddr *)&server, sizeof(server))) == -1)
        && (errno == EINTR) )
      ;
   if (retval == -1) {
        close(sock);
        return -1;
   }
   return sock;
}

/*
 *                           u_close
 * Close communication for the given file descriptor.
 * parameter:
 *       fd = file descriptor of socket connection to be closed
 * returns:
 *       a negative value indicates an error occurred
 */

int u_close(int fd)
{
   return close(fd);
}

/*
 *                           u_read
 *
 * Retrieve information from a file descriptor opened by u_open.
 *
 * parameters:
 *       fd = file descriptor
 *       buf = buffer to be output
 *       nbyte = number of bytes to retrieve
 * returns:
 *      a negative value indicates an error occurred
 *      otherwise the number of bytes read is returned
 */

ssize_t u_read(int fd, void *buf, size_t size)
{
   ssize_t retval;

   while (retval = read(fd, buf, size), retval == -1 && errno == EINTR)
     ;
   return retval;
}
```

```
/*
 *                            u_write
 *
 * Send information on a file descriptor opened by u_open.
 *
 * parameters:
 *       fd = file descriptor
 *       buf = buffer to be output
 *       nbyte = number of bytes to send
 * returns:
 *      a negative value indicates an error occurred
 *      otherwise the number of bytes written is returned
 */

ssize_t u_write(int fd, void *buf, size_t size)
{
   ssize_t retval;

   while (retval = write(fd, buf, size), retval == -1 && errno == EINTR)
      ;
   return retval;
}

/*
 *                            u_error
 *  Display an error message in the manner of perror or t_error.
 *
 *  parameter:
 *          s =  string to be prepended to system error message
 *  returns:    0
 *
 *  algorithm:  Since the only type of error comes from a system call,
 *              we only need to call perror.  This implementation should
 *              be as MT-safe as the underlying sockets.
 */

void u_error(char *s)
{
   perror(s);
}

/*
 *                            u_sync
 *
 * This function must be called after an exec or a dup to attach an
 * opened file descriptor in simple UICI implementations.  It is not
 * needed for sockets.
 *
```

```
*/
/*ARGSUSED*/
int u_sync(int fd)
{
/* Not needed for sockets, retained for compatibility */
   return 0;
}
```