## UNIX System Calls

**UNIX System Calls**

**A system call is just what its name implies -- a request for the**

**operating system to do something on behalf of the user's program. The**

**system calls are functions used in the kernel itself. To the**

**programmer, the system call appears as a normal C function call.**

**However since a system call executes code in the kernel, there must be a**

**mechanism to change the mode of a process from user mode to kernel mode.**

**The C compiler uses a predefined library of functions (the C library)**

**that have the names of the system calls. The library functions**

**typically invoke an instruction that changes the process execution mode**

**to kernel mode and causes the kernel to start executing code for system**

**calls. The instruction that causes the mode change is often referred to**

**as an "operating system trap" which is a software generated interrupt.**

**The library routines execute in user mode, but the system call interface**

**is a special case of an interrupt handler. The library functions pass**

**the kernel a unique number per system call in a machine dependent way --**

**either as a parameter to the operating system trap, in a particular**

**register, or on the stack -- and the kernel thus determines the specific**

**system call the user is invoking. In handling the operating system**

**trap, the kernel looks up the system call number in a table to find the**

**address of the appropriate kernel routine that is the entry point for**

**the system call and to find the number of parameters the system call**

**expects. The kernel calculates the (user) address of the first**

**parameter to the system call by adding (or subtracting, depending on the**

**direction of stack growth) an offset to the user stack pointer,**

**corresponding to the number of the parameters to the system call.**

**Finally, it copies the user parameters to the "u area" and call the**

**appropriate system call routine. After executing the code for the**

**system call, the kernel determines whether there was an error. If so,**

**it adjusts register locations in the saved user register context,**

**typically setting the "carry" bit for the PS (processor status) register**

**and copying the error number into register 0 location. If there were no**

**errors in the execution of the system call, the kernel clears the**

**"carry" bit in the PS register and copies the appropriate return values**

**from the system call into the locations for registers 0 and 1 in the**

**saved user register context. When the kernel returns from the operating**

**system trap to user mode, it returns to the library instruction after**

**the trap instruction. The library interprets the return values from the**

**kernel and returns a value to the user program.**

**UNIX system calls are used to manage the file system, control processes,**

**and to provide interprocess communication. The UNIX system interface**

**consists of about 80 system calls (as UNIX evolves this number will**

**increase). The following table lists about 40 of the more important**

**system call:**

**GENERAL CLASS SPECIFIC CLASS SYSTEM CALL**

**---------------------------------------------------------------------**

**File Structure Creating a Channel creat()**

**Related Calls open()**

**close()**

**Input/Output read()**

**write()**

**Random Access lseek()**

**Channel Duplication dup()**

**Aliasing and Removing link()**

**unlink()**

**Files**

**stat()**

**File Status fstat()**

**Access Control access()**

**chmod()**

**chown()**

**umask()**

**Device Control ioctl()**

**---------------------------------------------------------------------**

**Process Related Process Creation and exec()**

**Calls Termination fork()**

**wait()**

**exit()**

**Process Owner and Group getuid()**

**geteuid()**

**getgid()**

**getegid()**

**Process Identity getpid()**

**getppid()**

**Process Control signal()**

**kill()**

**alarm()**

**Change Working Directory chdir()**

**----------------------------------------------------------------------**

**Interprocess Pipelines pipe()**

**Communication Messages msgget()**

**msgsnd()**

**msgrcv()**

**msgctl()**

**Semaphores semget()**

**semop()**

**Shared Memory shmget()**

**shmat()**

**shmdt()**

**----------------------------------------------------------------------**

**[NOTE: The system call interface is that aspect of UNIX that has**

**changed the most since the inception of the UNIX system. Therefore,**

**when you write a software tool, you should protect that tool by putting**

**system calls in other subroutines within your program and then calling**

**only those subroutines. Should the next version of the UNIX system**

**change the syntax and semantics of the system calls you've used, you**

**need only change your interface routines.]**

**When a system call discovers and error, it returns -1 and stores the**

**reason the called failed in an external variable named "errno". The**

**"/usr/include/errno.h" file maps these error numbers to manifest**

**constants, and it these constants that you should use in your programs.**

**When a system call returns successfully, it returns something other than**

**-1, but it does not clear "errno". "errno" only has meaning directly**

**after a system call that returns an error.**

**When you use system calls in your programs, you should check the value**

**returned by those system calls. Furthermore, when a system call**

**discovers an error, you should use the "perror()" subroutine to print a**

**diagnostic message on the standard error file that describes why the**

**system call failed. The syntax for "perror()" is:**

**void perror(string)**

**char string;**

**"perror()" displays the argument string, a colon, and then the error**

**message, as directed by "errno", followed by a newline. The output of**

**"perror()" is displayed on "standard error". Typically, the argument**

**give to "perror()" is the name of the program that incurred the error,**

**argv[0]. However, when using subroutines and system calls on files, the**

**related file name might be passed to "perror()".**

**There are occasions where you the programmer might wish to maintain more**

**control over the printing of error messages than "perror()" provides --**

**such as with a formatted screen where the newline printed by "perror()"**

**would destroy the formatting. In this case, you can directly access the**

**same system external (global) variables that "perror()" uses. They are:**

**extern int errno;**

**extern char \*sys\_errlist[];**

**extern int sys\_nerr;**

**"errno" has been described above. "sys\_errlist" is an array (table) of**

**pointers to the error message strings. Each message string is null**

**terminated and does not contain a newline. "sys\_nerr" is the number of**

**messages in the error message table and is the maximum value "errno" can**

**assume. "errno" is used as the index into the table of error messages.**

**Following are two sample programs that display all of the system error**

**messages on standard error.**

**/\* errmsg1.c**

**print all system error messages using "perror()"**

**\*/**

**#include <stdio.h>**

**int main()**

**{**

**int i;**

**extern int errno, sys\_nerr;**

**for (i = 0; i < sys\_nerr; ++i)**

**{**

**fprintf(stderr, "%3d",i);**

**errno = i;**

**perror(" ");**

**}**

**exit (0);**

**}**

**/\* errmsg2.c**

**print all system error messages using the global error message table.**

**\*/**

**#include <stdio.h>**

**int main()**

**{**

**int i;**

**extern int sys\_nerr;**

**extern char \*sys\_errlist[];**

**fprintf(stderr,"Here are the current %d error messages:\n\n",sys\_nerr);**

**for (i = 0; i < sys\_nerr; ++i)**

**fprintf(stderr,"%3d: %s\n", i, sys\_errlist[i]);**

**}**

**Following are some examples in the use of the most often used system**

**calls.**

**File Structure Related System Calls**

**The file structure related system calls available in the UNIX system let**

**you create, open, and close files, read and write files, randomly access**

**files, alias and remove files, get information about files, check the**

**accessibility of files, change protections, owner, and group of files,**

**and control devices. These operations either use a character string**

**that defines the absolute or relative path name of a file, or a small**

**integer called a file descriptor that identifies the I/O channel. A**

**channel is a connection between a process and a file that appears to the**

**process as an unformatted stream of bytes. The kernel presents and**

**accepts data from the channel as a process reads and writes that**

**channel. To a process then, all input and output operations are**

**synchronous and unbuffered.**

**When doing I/O, a process specifies the file descriptor for an I/O**

**channel, a buffer to be filled or emptied, and the maximum size of data**

**to be transferred. An I/O channel may allow input, output, or both.**

**Furthermore, each channel has a read/write pointer. Each I/O operation**

**starts where the last operation finished and advances the pointer by the**

**number of bytes transferred. A process can access a channel's data**

**randomly by changing the read/write pointer.**

**All input and output operations start by opening a file using either the**

**"creat()" or "open()" system calls. These calls return a file**

**descriptor that identifies the I/O channel. Recall that file**

**descriptors 0, 1, and 2 refer to standard input, standard output, and**

**standard error files respectively, and that file descriptor 0 is a**

**channel to your terminal's keyboard and file descriptors 1 and 2 are**

**channels to your terminal's display screen.**

**creat()**

**The prototype for the creat() system call is:**

**int creat(file\_name, mode)**

**char \*file\_name;**

**int mode;**

**where file\_name is pointer to a null terminated character string that**

**names the file and mode defines the file's access permissions. The mode**

**is usually specified as an octal number such as 0666 that would mean**

**read/write permission for owner, group, and others or the mode may also**

**be entered using manifest constants defined in the**

**"/usr/include/sys/stat.h" file. If the file named by file\_name does not**

**exist, the UNIX system creates it with the specified mode permissions.**

**However, if the file does exist, its contents are discarded and the mode**

**value is ignored. The permissions of the existing file are retained.**

**Following is an example of how to use creat():**

**/\* creat.c \*/**

**#include <stdio.h>**

**#include <sys/types.h> /\* defines types used by sys/stat.h \*/**

**#include <sys/stat.h> /\* defines S\_IREAD & S\_IWRITE \*/**

**int main()**

**{**

**int fd;**

**fd = creat("datafile.dat", S\_IREAD | S\_IWRITE);**

**if (fd == -1)**

**printf("Error in opening datafile.dat\n");**

**else**

**{**

**printf("datafile.dat opened for read/write access\n");**

**printf("datafile.dat is currently empty\n");**

**}**

**close(fd);**

**exit (0);**

**}**

**The following is a sample of the manifest constants for the mode**

**argument as defined in /usr/include/sys/stat.h:**

**#define S\_IRWXU 0000700 /\* -rwx------ \*/**

**#define S\_IREAD 0000400 /\* read permission, owner \*/**

**#define S\_IRUSR S\_IREAD**

**#define S\_IWRITE 0000200 /\* write permission, owner \*/**

**#define S\_IWUSR S\_IWRITE**

**#define S\_IEXEC 0000100 /\* execute/search permission, owner \*/**

**#define S\_IXUSR S\_IEXEC**

**#define S\_IRWXG 0000070 /\* ----rwx--- \*/**

**#define S\_IRGRP 0000040 /\* read permission, group \*/**

**#define S\_IWGRP 0000020 /\* write " " \*/**

**#define S\_IXGRP 0000010 /\* execute/search " " \*/**

**#define S\_IRWXO 0000007 /\* -------rwx \*/**

**#define S\_IROTH 0000004 /\* read permission, other \*/**

**#define S\_IWOTH 0000002 /\* write " " \*/**

**#define S\_IXOTH 0000001 /\* execute/search " " \*/**

**Multiple mode values may be combined by or'ing (using the | operator)**

**the values together as demonstrated in the above sample program.**

**open()**

**Next is the open() system call. open() lets you open a file for**

**reading, writing, or reading and writing.**

**The prototype for the open() system call is:**

**#include <fcntl.h>**

**int open(file\_name, option\_flags [, mode])**

**char \*file\_name;**

**int option\_flags, mode;**

**where file\_name is a pointer to the character string that names the**

**file, option\_flags represent the type of channel, and mode defines the**

**file's access permissions if the file is being created.**

**The allowable option\_flags as defined in "/usr/include/fcntl.h" are:**

**#define O\_RDONLY 0 /\* Open the file for reading only \*/**

**#define O\_WRONLY 1 /\* Open the file for writing only \*/**

**#define O\_RDWR 2 /\* Open the file for both reading and writing\*/**

**#define O\_NDELAY 04 /\* Non-blocking I/O \*/**

**#define O\_APPEND 010 /\* append (writes guaranteed at the end) \*/**

**#define O\_CREAT 00400 /\*open with file create (uses third open arg) \*/**

**#define O\_TRUNC 01000 /\* open with truncation \*/**

**#define O\_EXCL 02000 /\* exclusive open \*/**

**Multiple values are combined using the | operator (i.e. bitwise OR).**

**Note: some combinations are mutually exclusive such as: O\_RDONLY |**

**O\_WRONLY and will cause open() to fail. If the O\_CREAT flag is used,**

**then a mode argument is required. The mode argument may be specified in**

**the same manner as in the creat() system call.**

**Following is an example of how to use open():**

**/\* open.c \*/**

**#include <fcntl.h> /\* defines options flags \*/**

**#include <sys/types.h> /\* defines types used by sys/stat.h \*/**

**#include <sys/stat.h> /\* defines S\_IREAD & S\_IWRITE \*/**

**static char message[] = "Hello, world";**

**int main()**

**{**

**int fd;**

**char buffer[80];**

**/\* open datafile.dat for read/write access (O\_RDWR)**

**create datafile.dat if it does not exist (O\_CREAT)**

**return error if datafile already exists (O\_EXCL)**

**permit read/write access to file (S\_IWRITE | S\_IREAD)**

**\*/**

**fd = open("datafile.dat",O\_RDWR | O\_CREAT | O\_EXCL, S\_IREAD | S\_IWRITE);**

**if (fd != -1)**

**{**

**printf("datafile.dat opened for read/write access\n");**

**write(fd, message, sizeof(message));**

**lseek(fd, 0L, 0); /\* go back to the beginning of the file \*/**

**if (read(fd, buffer, sizeof(message)) == sizeof(message))**

**printf("\"%s\" was written to datafile.dat\n", buffer);**

**else**

**printf("\*\*\* error reading datafile.dat \*\*\*\n");**

**close (fd);**

**}**

**else**

**printf("\*\*\* datafile.dat already exists \*\*\*\n");**

**exit (0);**

**}**

**close()**

**To close a channel, use the close() system call. The prototype for the**

**close() system call is:**

**int close(file\_descriptor)**

**int file\_descriptor;**

**where file\_descriptor identifies a currently open channel. close()**

**fails if file\_descriptor does not identify a currently open channel.**

**read() write()**

**The read() system call does all input and the write() system call does**

**all output. When used together, they provide all the tools necessary to**

**do input and output sequentially. When used with the lseek() system**

**call, they provide all the tools necessary to do input and output**

**randomly.**

**Both read() and write() take three arguments. Their prototypes are:**

**int read(file\_descriptor, buffer\_pointer, transfer\_size)**

**int file\_descriptor;**

**char \*buffer\_pointer;**

**unsigned transfer\_size;**

**int write(file\_descriptor, buffer\_pointer, transfer\_size)**

**int file\_descriptor;**

**char \*buffer\_pointer;**

**unsigned transfer\_size;**

**where file\_descriptor identifies the I/O channel, buffer\_pointer points**

**to the area in memory where the data is stored for a read() or where**

**the data is taken for a write(), and transfer\_size defines the maximum**

**number of characters transferred between the file and the buffer.**

**read() and write() return the number of bytes transferred.**

**There is no limit on transfer\_size, but you must make sure it's safe to**

**copy transfer\_size bytes to or from the memory pointed to by**

**buffer\_pointer. A transfer\_size of 1 is used to transfer a byte at a**

**time for so-called "unbuffered" input/output. The most efficient value**

**for transfer\_size is the size of the largest physical record the I/O**

**channel is likely to have to handle. Therefore, 1K bytes -- the disk**

**block size -- is the most efficient general-purpose buffer size for a**

**standard file. However, if you are writing to a terminal, the transfer**

**is best handled in lines ending with a newline.**

**For an example using read() and write(), see the above example of**

**open().**

**lseek()**

**The UNIX system file system treats an ordinary file as a sequence of**

**bytes. No internal structure is imposed on a file by the operating**

**system. Generally, a file is read or written sequentially -- that is,**

**from beginning to the end of the file. Sometimes sequential reading and**

**writing is not appropriate. It may be inefficient, for instance, to**

**read an entire file just to move to the end of the file to add**

**characters. Fortunately, the UNIX system lets you read and write**

**anywhere in the file. Known as "random access", this capability is made**

**possible with the lseek() system call. During file I/O, the UNIX system**

**uses a long integer, also called a File Pointer, to keep track of the**

**next byte to read or write. This long integer represents the number of**

**bytes from the beginning of the file to that next character. Random**

**access I/O is achieved by changing the value of this file pointer using**

**the lseek() system call.**

**The prototype for lseek() is:**

**long lseek(file\_descriptor, offset, whence)**

**int file\_descriptor;**

**long offset;**

**int whence;**

**where file\_descriptor identifies the I/O channel and offset and whence**

**work together to describe how to change the file pointer according to**

**the following table:**

**whence new position**

**------------------------------**

**0 offset bytes into the file**

**1 current position in the file plus offset**

**2 current end-of-file position plus offset**

**If successful, lseek() returns a long integer that defines the new file**

**pointer value measured in bytes from the beginning of the file. If**

**unsuccessful, the file position does not change.**

**Certain devices are incapable of seeking, namely terminals and the**

**character interface to a tape drive. lseek() does not change the file**

**pointer to these devices.**

**Following is an example using lseek():**

**/\* lseek.c \*/**

**#include <stdio.h>**

**#include <fcntl.h>**

**int main()**

**{**

**int fd;**

**long position;**

**fd = open("datafile.dat", O\_RDONLY);**

**if ( fd != -1)**

**{**

**position = lseek(fd, 0L, 2); /\* seek 0 bytes from end-of-file \*/**

**if (position != -1)**

**printf("The length of datafile.dat is %ld bytes.\n", position);**

**else**

**perror("lseek error");**

**}**

**else**

**printf("can't open datafile.dat\n");**

**close(fd);**

**}**

**Many UNIX systems have defined manifest constants for use as the**

**"whence" argument of lseek(). The definitions can be found in the**

**"file.h" and/or "unistd.h" include files. For example, the University**

**of Maryland's HP-9000 UNIX system has the following definitions:**

**from file.h we have:**

**#define L\_SET 0 /\* absolute offset \*/**

**#define L\_INCR 1 /\* relative to current offset \*/**

**#define L\_XTND 2 /\* relative to end of file \*/**

**and from unistd.h we have:**

**#define SEEK\_SET 0 /\* Set file pointer to "offset" \*/**

**#define SEEK\_CUR 1 /\* Set file pointer to current plus "offset" \*/**

**#define SEEK\_END 2 /\* Set file pointer to EOF plus "offset" \*/**

**The definitions from unistd.h are the most "portable" across UNIX and**

**MS-DOS C compilers.**

**dup()**

**The dup() system call duplicates an open file descriptor and returns the**

**new file descriptor. The new file descriptor has the following**

**properties in common with the original file descriptor:**

**refers to the same open file or pipe.**

**has the same file pointer -- that is, both file descriptors share one**

**file pointer.**

**has the same access mode, whether read, write, or read and write.**

**The prototype for dup() is:**

**int dup(file\_descriptor)**

**int file\_descriptor;**

**where file\_descriptor is the file descriptor describing the original I/O**

**channel returned by creat(), open(), pipe(), or dup() system calls.**

**dup() is guaranteed to return a file descriptor with the lowest integer**

**value available. It is because of this feature of returning the lowest**

**unused file descriptor available that processes accomplish I/O**

**redirection. The following example shows standard output redirected to**

**a file through the use of the dup() system call:**

**/\* dup.c**

**demonstrate redirection of standard output to a file.**

**\*/**

**#include <stdio.h>**

**#include <fcntl.h>**

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

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

**int main()**

**{**

**int fd;**

**fd = open("foo.bar",O\_WRONLY | O\_CREAT, S\_IREAD | S\_IWRITE );**

**if (fd == -1)**

**{**

**perror("foo.bar");**

**exit (1);**

**}**

**close(1); /\* close standard output \*/**

**dup(fd); /\* fd will be duplicated into standard out's slot \*/**

**close(fd); /\* close the extra slot \*/**

**printf("Hello, world!\n"); /\* should go to file foo.bar \*/**

**exit (0); /\* exit() will close the files \*/**

**}**

**link()**

**The UNIX system file structure allows more than one named reference to a**

**given file, a feature called "aliasing". Making an alias to a file**

**means that the file has more than one name, but all names of the file**

**refer to the same data. Since all names refer to the same data,**

**changing the contents of one file changes the contents of all aliases to**

**that file. Aliasing a file in the UNIX system amounts to the system**

**creating a new directory entry that contains the alias file name and**

**then copying the i-number of a existing file to the i-number position of**

**this new directory entry. This action is accomplished by the link()**

**system call. The link() system call links an existing file to a new**

**file.**

**The prototype for link() is:**

**int link(original\_name, alias\_name)**

**char \*original\_name, \*alias\_name;**

**where both original\_name and alias\_name are character strings that name**

**the existing and new files respectively. link() will fail and no link**

**will be created if any of the following conditions holds:**

**a path name component is not a directory.**

**a path name component does not exist.**

**a path name component is off-limits.**

**original\_name does not exist.**

**alias\_name does exist.**

**original\_name is a directory and you are not the superuser.**

**a link is attempted across file systems.**

**the destination directory for alias\_name is not writable.**

**the destination directory is on a mounted read-only file system.**

**Following is a short example:**

**/\* link.c**

**\*/**

**#include <stdio.h>**

**int main()**

**{**

**if ((link("foo.old", "foo.new")) == -1)**

**{**

**perror(" ");**

**exit (1); /\* return a non-zero exit code on error \*/**

**}**

**exit(0);**

**}**

**unlink()**

**The opposite of the link() system call is the unlink() system call.**

**unlink() removes a file by zeroing the i-number part of the file's**

**directory entry, reducing the link count field in the file's inode by 1,**

**and releasing the data blocks and the inode if the link count field**

**becomes zero. unlink() is the only system call for removing a file in**

**the UNIX system.**

**The prototype for unlink() is:**

**int unlink(file\_name)**

**char \*file\_name;**

**where file\_name names the file to be unlinked. unlink() fails if any of**

**the following conditions holds:**

**a path name component is not a directory.**

**a path name component does not exist.**

**a path name component is off-limits.**

**file\_name does not exist.**

**file\_name is a directory and you are not the superuser.**

**the directory for the file named by file\_name is not writable.**

**the directory is contained in a file system mounted read-only.**

**It is important to understand that a file's contents and its inode are**

**not discarded until all processes close the unlinked file.**

**Following is a short example:**

**/\* unlink.c**

**\*/**

**#include <stdio.h>**

**int main()**

**{**

**if ((unlink("foo.bar")) == -1)**

**{**

**perror(" ");**

**exit (1); /\* return a non-zero exit code on error \*/**

**}**

**exit (0);**

**}**

**Process Related System Calls**

**exec**

**The UNIX system provides several system calls to create and end program, to**

**send and receive software interrupts, to allocate memory, and to do other**

**useful jobs for a process. Four system calls are provided for creating a**

**process, ending a process, and waiting for a process to complete. These**

**system calls are fork(), the "exec" family, wait(), and exit().**

**The UNIX system calls that transform a executable binary file into a process**

**are the "exec" family of system calls. The prototypes for these calls are:**

**int execl(file\_name, arg0 [, arg1, ..., argn], NULL)**

**char \*file\_name, \*arg0, \*arg1, ..., \*argn;**

**int execv(file\_name, argv)**

**char \*file\_name, \*argv[];**

**int execle(file\_name, arg0 [, arg1, ..., argn], NULL, envp)**

**char \*file\_name, \*arg0, \*arg1, ..., \*argn, \*envp[];**

**int execve(file\_name, argv, envp)**

**char \*file\_name, \*argv[], \*envp[];**

**int execlp(file\_name, arg0 [, arg1, ..., argn], NULL)**

**char \*file\_name, \*arg0, \*arg1, ..., \*argn;**

**int execvp(file\_name, argv)**

**char \*file\_name, \*argv[];**

**where file\_name names the executable binary file to be transformed into a**

**process, arg0 through argn and argv define the arguments to be passed to the**

**process, and envp defines the environment, also to be passed to the process.**

**By convention, arg0 and argv[0] name the last path name component of the**

**executable binary file named by file\_name. For execl(), execv(), execle(),**

**and execve(), file\_name must be the fully qualified path name of the**

**executable binary file. However for execlp() and execvp(), the PATH variable**

**is used to find the executable binary file. When the environment is not**

**explicitly given as an argument to an exec system call, the environment of**

**the current process is used. Furthermore, the last array element of both**

**argv and envp must be null to signify the end of the array.**

**Unlike the other system calls and subroutines, a successful exec system call**

**does not return. Instead, control is given to the executable binary file**

**named as the first argument. When that file is made into a process, that**

**process replaces the process that executed the exec system call -- a new**

**process is not created. If an exec call should fail, it will return a -1.**

**Letters added to the end of exec indicate the type of arguments:**

**l argn is specified as a list of arguments.**

**v argv is specified as a vector (array of character pointers).**

**e environment is specified as an array of character pointers.**

**p user's PATH is searched for command, and command can be a shell program**

**Following is a brief description of the six routines that make up the**

**collective family of exec routines:**

**execl Takes the path name of an executable program (binary file) as its**

**first argument. The rest of the arguments are a list of command**

**line arguments to the new program (argv[]). The list is**

**terminated with a null pointer:**

**execl("/bin/cat", "cat", "f1", "f2", (char \*) 0);**

**execl("a.out", "a.out", (char \*) 0);**

**Note that, by convention, the argument listed after the program**

**is the name of the command being executed (argv[0]).**

**execle Same as execl(), except that the end of the argument list is**

**followed by a pointer to a null-terminated list of character**

**pointers that is passed a the environment of the new program**

**(i.e., the place that getenv() searches for exported shell**

**variables):**

**static char \*env[] = {**

**"TERM=vt100",**

**"PATH=/bin:/usr/bin",**

**(char \*) 0 };**

**execle("/bin/cat", "cat", "f1", "f2", (char \*) 0, env);**

**execv Takes the path name of an executable program (binary file) as it**

**first argument. The second argument is a pointer to a list of**

**character pointers (like argv[]) that is passed as command line**

**arguments to the new program:**

**static char \*args[] = {**

**"cat",**

**"f1",**

**"f2",**

**(char \*) 0 };**

**execv("/bin/cat", args);**

**execve Same as execv(), except that a third argument is given as a**

**pointer to a list of character pointers (like argv[]) that is**

**passed as the environment of the new program:**

**static char \*env[] = {**

**"TERM=vt100",**

**"PATH=/bin:/usr/bin",**

**(char \*) 0 };**

**static char \*args[] = {**

**"cat",**

**"f1",**

**"f2",**

**(char \*) 0 };**

**execve("/bin/cat", args, env);**

**execlp Same as execl(), except that the program name doesn't have to be**

**a full path name, and it can be a shell program instead of an**

**executable module:**

**execlp("ls", "ls", "-l", "/usr", (char \*) 0);**

**execlp() searches the PATH environment variable to find the**

**specified program.**

**execvp Same as execv(), except that the program name doesn't have to be**

**a full path name, and it can be a shell program instead of an**

**executable module:**

**static char \*args[] = {**

**"cat",**

**"f1",**

**"f2",**

**(char \*) 0 };**

**execvp("cat", args);**

**When transforming an executable binary file into a process, the UNIX system**

**preserves some characteristics of the replaced process. Among the items**

**saved by the exec system call are:**

**The "nice" value for scheduling.**

**The process ID and the parent process ID.**

**The time left until an alarm clock signal.**

**The current working directory and the root directory.**

**The file creation mask as established with umask().**

**All open files.**

**The last of these is the most interesting because the shell uses this feature**

**to handle input/output redirection.**

**fork()**

**The exec family of system calls transforms an executable binary file into a**

**process that overlays the process that made the exec system call. The UNIX**

**system does not create a new process in response to an exec system call. To**

**create a new process, you must use the fork() system call. The prototype for**

**the fork() system call is:**

**int fork()**

**fork() causes the UNIX system to create a new process, called the "child**

**process", with a new process ID. The contents of the child process are**

**identical to the contents of the parent process.**

**The new process inherits several characteristics of the old process. Among**

**the characteristics inherited are:**

**The environment.**

**All signal settings.**

**The set user ID and set group ID status.**

**The time left until an alarm clock signal.**

**The current working directory and the root directory.**

**The file creation mask as established with umask().**

**The child process begins executing and the parent process continues executing**

**at the return from the fork() system call. This is difficult to understand**

**at first because you only call fork() once, yet it returns twice -- once per**

**process. To differentiate which process is which, fork() returns zero in the**

**child process and non-zero (the child's process ID) in the parent process.**

**exec routines are usually called after a call to fork(). This combination,**

**known as a fork/exec, allows a process to create a child to execute a**

**command, so that the parent doesn't destroy itself through an exec. Most**

**command interpreters (e.g. the shell) on UNIX use fork and exec.**

**wait()**

**You can control the execution of child processes by calling wait() in the**

**parent. wait() forces the parent to suspend execution until the child is**

**finished. wait() returns the process ID of a child process that finished.**

**If the child finishes before the parent gets around to calling wait(), then**

**when wait() is called by the parent, it will return immediately with the**

**child's process ID. (It is possible to have more that one child process by**

**simply calling fork() more than once.). The prototype for the wait() system**

**call is:**

**int wait(status)**

**int \*status;**

**where status is a pointer to an integer where the UNIX system stores the**

**value returned by the child process. wait() returns the process ID of the**

**process that ended. wait() fails if any of the following conditions hold:**

**The process has no children to wait for.**

**status points to an invalid address.**

**The format of the information returned by wait() is as follows:**

**If the process ended by calling the exit() system call, the second lowest**

**byte of status is set to the argument given to exit() and the lowest byte**

**of status is set to zeroes.**

**If the process ended because of a signal, the second lowest byte of status**

**is set to zeroes and the lowest byte of status contains the signal number**

**that ended the process. If the seventh bit of the lowest byte of status**

**is set (i.e. status & 0200 == 0200) then the UNIX system produced a core**

**dump of the process.**

**exit()**

**The exit() system call ends a process and returns a value to it parent. The**

**prototype for the exit() system call is:**

**void exit(status)**

**int status;**

**where status is an integer between 0 and 255. This number is returned to the**

**parent via wait() as the exit status of the process. By convention, when a**

**process exits with a status of zero that means it didn't encounter any**

**problems; when a process exit with a non-zero status that means it did have**

**problems.**

**exit() is actually not a system routine; it is a library routine that call**

**the system routine \_exit(). exit() cleans up the standard I/O streams before**

**calling \_exit(), so any output that has been buffered but not yet actually**

**written out is flushed. Calling \_exit() instead of exit() will bypass this**

**cleanup procedure. exit() does not return.**

**Following are some example programs that demonstrate the use of fork(),**

**exec(), wait(), and exit():**

**/\* status.c**

**demonstrates exit() returning a status to wait().**

**\*/**

**int main()**

**{**

**unsigned int status;**

**if ( fork () == 0 ) { /\* == 0 means in child \*/**

**scanf ("%d", &status);**

**exit (status);**

**}**

**else { /\* != 0 means in parent \*/**

**wait (&status);**

**printf("child exit status = %d\n", status >> 8);**

**}**

**}**

**Note: since wait() returns the exit status multiplied by 256 (contained in**

**the upper 8 bits), the status value is shifted right 8 bits (divided by 256)**

**to obtain the correct value.**

**/\* myshell.c**

**This program is a simple command interpreter that uses execlp() to**

**execute commands typed in by the user.**

**\*/**

**#include <stdio.h>**

**#define EVER ;;**

**int main()**

**{**

**int process;**

**char line[81];**

**for (EVER)**

**{**

**fprintf(stderr, "cmd: ");**

**if ( gets (line) == (char \*) NULL) /\* blank line input \*/**

**exit (0);**

**/\* create a new process \*/**

**process = fork ();**

**if (process > 0) /\* parent \*/**

**wait ((int \*) 0); /\* null pointer - return value not saved \*/**

**else if (process == 0) /\* child \*/**

**{ /\* execute program \*/**

**execlp (line, line, (char \*) NULL);**

**/\* some problem if exec returns \*/**

**fprintf (stderr, "Can't execute %s\n", line);**

**exit (1);**

**}**

**else if ( process == -1) /\* can't create a new process \*/**

**{**

**fprintf (stderr, "Can't fork!\n");**

**exit (2);**

**}**

**}**

**}**

**The following program demonstrates a practical use of fork() and exec() to**

**create a new directory. Only the superuser has the permission to use the**

**mknod() system call to create a new directory -- an ordinary user cannot use**

**mknod() to create a directory. So, we use fork/exec to call upon the UNIX**

**system's mkdir command that anyone can use to create a directory.**

**/\* newdir.c**

**create a new directory, called newdir, using fork() and exec().**

**\*/**

**#include <stdio.h>**

**int main()**

**{**

**int fd;**

**if ( fork() != 0)**

**wait ((int \*) 0);**

**else**

**{**

**execl ("/bin/mkdir", "mkdir", "newdir", (char \*) NULL);**

**fprintf (stderr, "exec failed!\n");**

**exit (1);**

**}**

**/\* now use newdir \*/**

**if ( (fd = open("newdir/foo.bar", O\_RDWR | O\_CREAT, 0644)) == -1)**

**{**

**fprintf (stderr, "open failed!\n");**

**exit (2);**

**}**

**write (fd, "Hello, world\n", 14);**

**close (fd);**

**exit (0);**

**}**

**Software Interrupts**

**signal()**

**The UNIX system provides a facility for sending and receiving software**

**interrupts, also called SIGNALS. Signals are sent to a process when a**

**predefined condition happens. The number of signals available is system**

**dependent. For example, the University's HP-9000 has 31 signals defined.**

**The signal name is defined in /usr/include/sys/signal.h as a manifest**

**constant.**

**Programs can respond to signals three different ways. These are:**

**1. Ignore the signal. This means that the program will never be informed of**

**the signal no matter how many times it occurs. The only exception to this**

**is the SIGKILL signal which can neither be ignored nor caught.**

**2. A signal can be set to its default state, which means that the process**

**will be ended when it receives that signal. In addition, if the process**

**receives any of SIGQUIT, SIGILL, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, or**

**SIGSYS, the UNIX system will produce a core image (core dump), if possible,**

**in the directory where the process was executing when it received the**

**program-ending signal.**

**3. Catch the signal. When the signal occurs, the UNIX system will transfer**

**control to a previously defined subroutine where it can respond to the signal**

**as is appropriate for the program.**

**You define how you want to respond to a signal with the signal() system call.**

**The prototype is:**

**#include <sys/signal.h>**

**int (\* signal ( signal\_name, function ))**

**int signal\_name;**

**int (\* function)();**

**where signal\_name is the name of the signal from signal.h and function is any**

**of SIG\_IGN, meaning that you wish to ignore the signal when it occurs;**

**SIG\_DFL, meaning that you wish the UNIX system to take the default action**

**when your program receives the signal; or a pointer to a function that**

**returns an integer. The function is given control when your program receives**

**the signal, and the signal number is passed as an argument. signal() returns**

**the previous value of function, and signal() fails if any of the following**

**conditions hold:**

**signal\_name is an illegal name or SIGKILL.**

**function points to an invalid memory address.**

**Once a signal is caught, the UNIX system resets it to its initial state (the**

**default condition). In general, if you intend for your program to be able to**

**catch a signal repeatedly, you need to re-arm the signal handling mechanism.**

**You must do this as soon after receipt of the signal as possible, namely just**

**after entering the signal handling routine.**

**You should use signals in your programs to isolate critical sections from**

**interruption.**

**The state of all signals is preserved across a fork() system call, but all**

**caught signals are set to SIG\_DFL across an exec system call.**

**kill()**

**The UNIX system sends a signal to a process when something happens, such as**

**typing the interrupt key on a terminal, or attempting to execute an illegal**

**instruction. Signals are also sent to a process with the kill() system call.**

**Its prototype is:**

**int kill (process\_id, signal\_name )**

**int process\_it, signal\_name;**

**where process\_id is the ID of the process to be signaled and signal\_name is**

**the signal to be sent to that process. If process\_id has a positive value,**

**that value is assumed to be the process ID of the process to whom signal\_name**

**signal is to be sent. If process\_id has the value 0, then signal\_name signal**

**is sent to all processes in the sending process' process group, that is all**

**processes that have been started from the same terminal. If process\_id has**

**the value -1 and the process executing the kill() system call is the**

**superuser, then signal\_name is sent to all processes excluding process 0 and**

**process 1 that have the same user ID as the process executing the kill().**

**kill() fails if any of the following conditions hold:**

**signal\_name is not a valid signal.**

**there is not a process in the system with process ID process\_id.**

**even though the process named by process\_id is in the system, you cannot**

**send it a signal because your effective user ID does not match either the**

**real or effective user ID of process\_id.**

**alarm()**

**Every process has an alarm clock stored in its system-data segment. When the**

**alarm goes off, signal SIGALRM is sent to the calling process. A child**

**inherits its parent's alarm clock value, but the actual clock isn't shared.**

**The alarm clock remains set across an exec.**

**The prototype for alarm() is:**

**unsigned int alarm(seconds)**

**unsigned int seconds;**

**where seconds defines the time after which the UNIX system sends the SIGALRM**

**signal to the calling process. Each successive call to alarm() nullifies the**

**previous call, and alarm() returns the number of seconds until that alarm**

**would have gone off. If seconds has the value 0, the alarm is canceled.**

**alarm() has no error conditions.**

**The following is an example program that demonstrates the use of the signal()**

**and alarm() system calls:**

**/\* timesup.c \*/**

**#include <stdio.h>**

**#include <sys/signal.h>**

**#define EVER ;;**

**void main();**

**int times\_up();**

**void main()**

**{**

**signal (SIGALRM, times\_up); /\* go to the times\_up function \*/**

**/\* when the alarm goes off. \*/**

**alarm (10); /\* set the alarm for 10 seconds \*/**

**for (EVER) /\* endless loop. \*/**

**; /\* hope the alarm works. \*/**

**}**

**int times\_up(sig)**

**int sig; /\* value of signal \*/**

**{**

**printf("Caught signal #< %d >n", sig);**

**printf("Time's up! I'm outta here!!\n");**

**exit(sig); /\* return the signal number \*/**

**}**

**Interprocess Communication**

**UNIX System V allows processes to communicate with one another using pipes,**

**messages, semaphores, and shared memory. This sections describes how to**

**communicate using pipes.**

**One way to communicate between two processes is to create a pipeline with the**

**pipe() system call. pipe() builds the channel, but it is up to you to**

**connect the standard input of one process to the standard output of the other**

**process.**

**The prototype for pipe() is:**

**int pipe (file\_descriptors)**

**int file\_descriptors[2];**

**where file\_descriptors[2] is an array that pipe() fills with a file**

**descriptor opened for reading, file\_descriptor[0], and a file\_descriptor**

**opened for writing, file\_descriptor[1]. pipe() fails for the following**

**condition:**

**there are too many open I/O channels.**

**Some I/O system calls act differently on pipe file descriptors from the way**

**they do on ordinary files, and some do nothing at all. Following is a**

**summary of these actions:**

**write Data written to a pipe is sequenced in order of arrival.**

**Normally, is the pipe becomes full, write() will block until**

**enough old data is removed by read(). There are no partial**

**writes; the entire write() will be completed. The capacity of a**

**pipe varies with the UNIX implementation, but it is always at**

**least 4096 bytes (4K). If fcntl() is called to set the O\_NDELAY**

**flag, write() will not block on a full pipe and will return a**

**count of 0. The only way to put an end-of-file on a pipe is to**

**close the writing file descriptor.**

**read Data is read from a pipe in order of arrival, just as it was**

**written. Once read, data can't be reread or put back. Normally,**

**if the pipe is empty, read will block until at least one byte of**

**data is available, unless the writing file descriptor is closed,**

**in which case the read will return a 0 count (the usual**

**end-of-file indication). But the byte count given as the third**

**argument to read will not necessarily be satisfied - only as many**

**bytes as are present at that instant will be read, and an**

**appropriate count will be returned. The byte count will never be**

**exceeded, of course; unread bytes will remain for the next**

**read(). If the O\_NDELAY flag is set, a read() on an empty pipe**

**will return with a 0 count. This suffers from the same ambiguity**

**as reads on communication lines. A 0 count also means**

**end-of-file.**

**close Means more on a pipe than it does on a file. Not only does it**

**free up the file descriptor for reuse, but when the writing file**

**descriptor is closed it acts as an end-of-file for the reader.**

**If the read end file descriptor is closed, a write() on the other**

**file descriptor will cause and error. A fatal signal is also**

**normally generated (SIGPIPE - #13).**

**fcntl This system call sets or clears the O\_NDELAY flag, whose effect**

**is described under write and read above.**

**fstat Not very useful on pipes. The size returned is the number of**

**bytes in the pipe, but this fact is seldom useful. A pipe may be**

**distinguished by a link count of 0, since a pipe is the only**

**source of a file descriptor associated with something not linked**

**into a directory. This distinction might be useful to I/O**

**routines that want to treat pipes specially.**

**open Not used with pipes.**

**creat Not used with pipes.**

**lseek Not used with pipes. This means that if a pipe contains a**

**sequence of messages, it isn't possible to look through them for**

**the message to read next. Like toothpaste in a tube, you have to**

**get it out to examine it, and then there is no way to put it**

**back.**

**Pipes use the buffer cache just as ordinary files do. Therefore, the**

**benefits of writing and reading pipes in units of a block (usually 512 or**

**1024 bytes) are just as great. A single write() execution is atomic, so if**

**512 bytes are written with a single system call, the corresponding read()**

**will return with 512 bytes (if it requests that many). It will not return**

**with less than the full block. However, if the writer is not writing**

**complete blocks, but the reader is trying to read complete blocks, the reader**

**may keep getting partial blocks anyway.**

**The following example program demonstrates how to set up a one-way pipe**

**between two related processes. Note that the processes MUST be related**

**(parent, child, grandchild, etc.) since the pipe mechanism is based on the**

**fact that file descriptors are inherited when a process is created. Error**

**checking in the following program has been minimized in order to keep the**

**code uncluttered and readable. In a "real" program more error checking on**

**the system calls should be done.**

**/\* who\_wc.c \*/**

**/\* demonstrates a one-way pipe between two processes.**

**This program implements the equivalent of the shell command:**

**who | wc -l**

**which will count the number of users logged in.**

**\*/**

**#include <stdio.h>**

**/\* Define some manifest constants to make the code more understandable \*/**

**#define ERR (-1) /\* indicates an error condition \*/**

**#define READ 0 /\* read end of a pipe \*/**

**#define WRITE 1 /\* write end of a pipe \*/**

**#define STDIN 0 /\* file descriptor of standard in \*/**

**#define STDOUT 1 /\* file descriptor of standard out \*/**

**int main()**

**{**

**int pid\_1, /\* will be process id of first child - who \*/**

**pid\_2, /\* will be process id of second child - wc \*/**

**pfd[2]; /\* pipe file descriptor table. \*/**

**if ( pipe ( pfd ) == ERR ) /\* create a pipe \*/**

**{ /\* must do before a fork \*/**

**perror (" ");**

**exit (ERR);**

**}**

**if (( pid\_1 = fork () ) == ERR) /\* create 1st child \*/**

**{**

**perror (" ");**

**exit (ERR);**

**}**

**if ( pid\_1 != 0 ) /\* in parent \*/**

**{**

**if (( pid\_2 = fork () ) == ERR) /\* create 2nd child \*/**

**{**

**perror (" ");**

**exit (ERR);**

**}**

**if ( pid\_2 != 0 ) /\* still in parent \*/**

**{**

**close ( pfd [READ] ); /\* close pipe in parent \*/**

**close ( pfd [WRITE] ); /\* conserve file descriptors \*/**

**wait (( int \* ) 0); /\* wait for children to die \*/**

**wait (( int \* ) 0);**

**}**

**else /\* in 2nd child \*/**

**{**

**close (STDIN); /\* close standard input \*/**

**dup ( pfd [READ] ); /\* read end of pipe becomes stdin \*/**

**close ( pfd [READ] ); /\* close unneeded I/O \*/**

**close ( pfd [WRITE] ); /\* close unneeded I/O \*/**

**execl ("/bin/wc", "wc", "-l", (char \*) NULL);**

**}**

**}**

**else /\* in 1st child \*/**

**{**

**close (STDOUT); /\* close standard out \*/**

**dup ( pfd [WRITE] ); /\* write end of pipes becomes stdout \*/**

**close ( pfd [READ] ); /\* close unneeded I/O \*/**

**close ( pfd [WRITE] ); /\* close unneeded I/O \*/**

**execl ("/bin/who", "who", (char \*) NULL);**

**}**

**exit (0);**

**}**

**The following is a diagram of the processes created by who\_wc.**

**IMMMMMMMMMMMMMMMMMM;**

**: :**

**ZDDDDDDDDDDDDDDD: who\_wc :DDDDDDDDDDDDD?**

**3 : : 3**

**3 HMMMMMMMMMMMMMMMMMM< 3**

**3 3**

**IMMMMMMMOMMMMMM; IMMMMMMOMMMMMMM;**

**: : : :**

**: who :DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD> wc -l :**

**: : pipe channel : :**

**HMMMMMMMMMMMMMM< HMMMMMMMQMMMMMM<**

**3**

**3**

**ZDDDDDDADDDDDD?**

**3 3**

**3 terminal 3**

**@DDDDDDDDDDDDDY**

**File Status**

**stat() - fstat()**

**The i-node data structure holds all the information about a file except the**

**file's name and its contents. Sometimes your programs need to use the**

**information in the i-node structure to do some job. You can access this**

**information with the stat() and fstat() system calls. stat() and fstat()**

**return the information in the i-node for the file named by a string and by a**

**file descriptor, respectively. The format for the i-node struct returned by**

**these system calls is defined in /usr/include/sys/stat.h. stat.h uses types**

**built with the C language typedef construct and defined in the file**

**/usr/include/sys/types.h, so it too must be included and must be included**

**before the inclusion of the stat.h file.**

**The prototypes for stat() and fstat() are:**

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

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

**int stat(file\_name, stat\_buf)**

**char \*file\_name;**

**struct stat \*stat\_buf;**

**int fstat(file\_descriptor, stat\_buf)**

**int file\_descriptor;**

**struct stat \*stat\_buf;**

**where file\_name names the file as an ASCII string and file\_descriptor names**

**the I/O channel and therefore the file. Both calls returns the file's**

**specifics in stat\_buf. stat() and fstat() fail if any of the following**

**conditions hold:**

**a path name component is not a directory (stat() only).**

**file\_name does not exit (stat() only).**

**a path name component is off-limits (stat() only).**

**file\_descriptor does not identify an open I/O channel (fstat() only).**

**stat\_buf points to an invalid address.**

**Following is an extract of the stat.h file from the University's HP-9000. It**

**shows the definition of the stat structure and some manifest constants used**

**to access the st\_mode field of the structure.**

**/\* stat.h \*/**

**struct stat**

**{**

**dev\_t st\_dev; /\* The device number containing the i-node \*/**

**ino\_t st\_ino; /\* The i-number \*/**

**unsigned short st\_mode; /\* The 16 bit mode \*/**

**short st\_nlink; /\* The link count; 0 for pipes \*/**

**ushort st\_uid; /\* The owner user-ID \*/**

**ushort st\_gid; /\* The group-ID \*/**

**dev\_t st\_rdev; /\* For a special file, the device number \*/**

**off\_t st\_size; /\* The size of the file; 0 for special files \*/**

**time\_t st\_atime; /\* The access time. \*/**

**int st\_spare1;**

**time\_t st\_mtime; /\* The modification time. \*/**

**int st\_spare2;**

**time\_t st\_ctime; /\* The status-change time. \*/**

**int st\_spare3;**

**long st\_blksize;**

**long st\_blocks;**

**uint st\_remote:1; /\* Set if file is remote \*/**

**dev\_t st\_netdev; /\* ID of device containing \*/**

**/\* network special file \*/**

**ino\_t st\_netino; /\* Inode number of network special file \*/**

**long st\_spare4[9];**

**};**

**#define S\_IFMT 0170000 /\* type of file \*/**

**#define S\_IFDIR 0040000 /\* directory \*/**

**#define S\_IFCHR 0020000 /\* character special \*/**

**#define S\_IFBLK 0060000 /\* block special \*/**

**#define S\_IFREG 0100000 /\* regular (ordinary) \*/**

**#define S\_IFIFO 0010000 /\* fifo \*/**

**#define S\_IFNWK 0110000 /\* network special \*/**

**#define S\_IFLNK 0120000 /\* symbolic link \*/**

**#define S\_IFSOCK 0140000 /\* socket \*/**

**#define S\_ISUID 0004000 /\* set user id on execution \*/**

**#define S\_ISGID 0002000 /\* set group id on execution \*/**

**#define S\_ENFMT 0002000 /\* enforced file locking (shared with S\_ISGID)\*/**

**#define S\_ISVTX 0001000 /\* save swapped text even after use \*/**

**Following is an example program demonstrating the use of the stat() system**

**call to determine the status of a file:**

**/\* status.c \*/**

**/\* demonstrates the use of the stat() system call to determine the**

**status of a file.**

**\*/**

**#include <stdio.h>**

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

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

**#define ERR (-1)**

**#define TRUE 1**

**#define FALSE 0**

**int main();**

**int main(argc, argv)**

**int argc;**

**char \*argv[];**

**{**

**int isdevice = FALSE;**

**struct stat stat\_buf;**

**if (argc != 2)**

**{**

**printf("Usage: %s filename\n", argv[0]);**

**exit (1);**

**}**

**if ( stat( argv[1], &stat\_buf) == ERR)**

**{**

**perror("stat");**

**exit (1);**

**}**

**printf("\nFile: %s status:\n\n",argv[1]);**

**if ((stat\_buf.st\_mode & S\_IFMT) == S\_IFDIR)**

**printf("Directory\n");**

**else if ((stat\_buf.st\_mode & S\_IFMT) == S\_IFBLK)**

**{**

**printf("Block special file\n");**

**isdevice = TRUE;**

**}**

**else if ((stat\_buf.st\_mode & S\_IFMT) == S\_IFCHR)**

**{**

**printf("Character special file\n");**

**isdevice = TRUE;**

**}**

**else if ((stat\_buf.st\_mode & S\_IFMT) == S\_IFREG)**

**printf("Ordinary file\n");**

**else if ((stat\_buf.st\_mode & S\_IFMT) == S\_IFIFO)**

**printf("FIFO\n");**

**if (isdevice)**

**printf("Device number:%d, %d\n", (stat\_buf.st\_rdev > 8) & 0377,**

**stat\_buf.st\_rdev & 0377);**

**printf("Resides on device:%d, %d\n", (stat\_buf.st\_dev > 8) & 0377,**

**stat\_buf.st\_dev & 0377);**

**printf("I-node: %d; Links: %d; Size: %ld\n", stat\_buf.st\_ino,**

**stat\_buf.st\_nlink, stat\_buf.st\_size);**

**if ((stat\_buf.st\_mode & S\_ISUID) == S\_ISUID)**

**printf("Set-user-ID\n");**

**if ((stat\_buf.st\_mode & S\_ISGID) == S\_ISGID)**

**printf("Set-group-ID\n");**

**if ((stat\_buf.st\_mode & S\_ISVTX) == S\_ISVTX)**

**printf("Sticky-bit set -- save swapped text after use\n");**

**printf("Permissions: %o\n", stat\_buf.st\_mode & 0777);**

**exit (0);**

**}**

**access()**

**To determine if a file is accessible to a program, the access() system call**

**may be used. Unlike any other system call that deals with permissions,**

**access() checks the real user-ID or group-ID, not the effective ones.**

**The prototype for the access() system call is:**

**int access(file\_name, access\_mode)**

**char \*file\_name;**

**int access\_mode;**

**where file\_name is the name of the file to which access permissions given in**

**access\_mode are to be applied. Access modes are often defined as manifest**

**constants in /usr/include/sys/file.h. The available modes are:**

**Value Meaning file.h constant**

**----- ------ ------**

**00 existence F\_OK**

**01 execute X\_OK**

**02 write W\_OK**

**04 read R\_OK**

**These values may be ORed together to check for mone than one access**

**permission. The call to access() returns 0 if the program has the given**

**access permissions, otherwise -1 is returned and errno is set to the reason**

**for failure. This call is somewhat useful in that it makes checking for a**

**specific permission easy. However, it only answers the question "do I have**

**this permission?" It cannot answer the question "what permissions do I**

**have?"**

**The following example program demonstrates the use of the access() system**

**call to remove a file. Before removing the file, a check is made to make**

**sure that the file exits and that it is writable (it will not remove a**

**read-only file).**

**/\* remove.c \*/**

**#include <stdio.h>**

**#include <sys/file.h>**

**#define ERR (-1)**

**int main();**

**int main(argc, argv)**

**int argc;**

**char \*argv[];**

**{**

**if (argc != 2)**

**{**

**printf("Usage: %s filename\n", argv[0]);**

**exit (1);**

**}**

**if (access (argv[1], F\_OK) == ERR) /\* check that file exists \*/**

**{**

**perror(argv[1]);**

**exit (1);**

**}**

**if (access (argv[1], W\_OK) == ERR) /\* check for write permission \*/**

**{**

**fprintf(stderr,"File: %s is write protected!\n", argv[1]);**

**exit (1);**

**}**

**if (unlink (argv[1]) == ERR)**

**{**

**perror(argv[1]);**

**exit (1);**

**}**

**exit (0);**

**}**

**Miscellaneous System Calls / Examples**

**Directories**

**A directory is simply a special file that contains (among other information)**

**i-number/filename pairs. With the exception of 4.2 and 4.3 BSD, all versions**

**of the UNIX system limit filenames to 14 characters. These short filenames**

**make for a simple fixed size directory format on System V.**

**System V Directories**

**A directory contains structures of type direct, defined in the include file**

**/usr/include/sys/dir.h. The include file /usr/include/sys/types.h must also**

**be included to define the types used by the structure. The directory**

**structure is:**

**#define DIRSIZ 14**

**struct direct {**

**ino\_t d\_ino;**

**char d\_name[DIRSIZ];**

**};**

**It should be noted that the name of the file, d\_name is NOT guaranteed to be**

**null-terminated; programs should always be careful of this. Files which have**

**been deleted will have i-numbers (d\_ino) equal to zero; these should in**

**general be skipped over when reading the directory. A directory is read**

**simply by opening it (in read-only mode) and reading structures either one at**

**a time or all at once. The following example program simply opens the**

**current directory and prints the names of all the files it contains. The**

**program simulates the ls -a command. Note that the file names are not sorted**

**like the real ls command would do.**

**/\* my\_ls.c**

**This program simulates the System V style ls -a command. Filenames**

**are printed as they occur in the directory -- no sorting is done.**

**\*/**

**#include <stdio.h>**

**#include <fcntl.h>**

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

**#include <sys/dir.h>**

**#define ERR (-1)**

**int main()**

**{**

**int fd;**

**struct direct dir;**

**if (( fd = open (".", O\_RDONLY)) == ERR) /\* open current directory \*/**

**{**

**perror("open");**

**exit (1);**

**}**

**while (( read (fd, &dir, sizeof (struct direct)) > 0 )**

**{**

**if ( dir.d\_ino == 0 ) /\* is it a deleted file? \*/**

**continue; /\* yes, so go read another \*/**

**/\* make sure we print no more than DIRSIZ characters \*/**

**printf ("%.\*s\n", DIRSIZ, dir.d\_name);**

**}**

**close (fd);**

**exit (0);**

**}**

**If you need more information about the file such as size or permissions, you**

**would use the stat() system call to obtain it.**

**Berkeley Style Directories**

**A directory contains structures of type direct, defined in the include file**

**/usr/include/sys/ndir.h. The include file /usr/include/sys/types.h must also**

**be included to define the types used by the structure. The directory**

**structure is:**

**#define MAXNAMLEN 255**

**#define DIRSIZ\_CONSTANT 14 /\* equivalent to DIRSIZ \*/**

**struct direct {**

**long d\_fileno; /\* file number of entry \*/**

**short d\_reclen; /\* length of this record \*/**

**short d\_namlen; /\* length of string in d\_name \*/**

**char d\_name[MAXNAMLEN + 1]; /\* name (up to MAXNAMLEN + 1) \*/**

**};**

**#if !(defined KERNEL) && !(defined ATT3B2)**

**#define d\_ino d\_fileno /\* compatibility \*/**

**Unlike on System V, filenames can be longer than 14 characters and the size**

**of a directory structure can be variable. Therefore, the read() call can not**

**be used to read the directory. Instead, Berkely style systems provide a set**

**of library functions to read directories. These functions are also declared**

**in the ndir.h include file. They are:**

**extern DIR \*opendir();**

**extern struct direct \*readdir();**

**extern long telldir();**

**extern void seekdir();**

**#define rewinddir(dirp) seekdir((dirp), (long)0)**

**extern void closedir();**

**The following example shows how to perform a Berkeley (or HP) style ls -a**

**read of a directory. One important note: filenames in the directory**

**structure are null-terminated in Berkeley style systems -- on System V they**

**are not.**

**#include <stdio.h>**

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

**#include <ndir.h>**

**main()**

**{**

**DIR \*dirp;**

**struct direct \*dp;**

**dirp = opendir("."); /\* open the current directory \*/**

**while ((dp = readdir(dirp)) != NULL)**

**{**

**if (dp->d\_ino == 0) /\* ignore deleted files \*/**

**continue;**

**else**

**printf("%s\n",dp->d\_name); /\* the name is null-terminated \*/**

**}**

**}**

**For more information, type: man directory while logged onto the**

**University's HP system.**

**Time**

**The UNIX operating system keeps track of the current date and time by storing**

**the number of seconds that have elasped since midnight January 1, 1970 UTC**

**(Coordinated Universal Time, also known as Greenwich Mean Time (GMT)). This**

**date is considered the informal "birthday" of the UNIX operating system. The**

**time is stored in a signed long integer. (For the curious, assuming a 32 bit**

**signed long integer, UNIX time will break at 03:14:08 January 19, 2038 UTC.)**

**In all versions of UNIX, the time() system call may be used to obtain the**

**time of day. This call is peculiar in that if given the address of a long**

**integer as an argument, it places the time in that integer and returns it.**

**If, however, a null pointer is passed, the time of day is just returned.**

**Several routines are available to convert the long integer returned by time()**

**into an ASCII date string. With the UNIX operating system, an ASCII date**

**string is a string as shown below:**

**Day Mon dd hh:mm:ss yyyy**

**For example: Sat Mar 24 11:03:36 1990**

**The ctime() library function can be used to do the above conversion. An**

**example is:**

**/\* my\_date.c**

**print the current date and time in a format similar to the output**

**of the date command.**

**\*/**

**#include <stdio.h>**

**#include <time.h> /\* may need to be #include <sys/time.h> instead \*/**

**int main()**

**{**

**long now, time();**

**char \*ctime();**

**time (&now);**

**printf("It is now %s\n", ctime (&now));**

**exit (0);**

**}**

**Often you need access to specific information about the current date and**

**time. The localtime() and gmtime() functions will provide it. They do this**

**by converting the long integer returned by time() into a data structure**

**called tm, which is defined in the time.h header file. In fact, this is what**

**the header file looks like:**

**struct tm {**

**int tm\_sec; /\* seconds after the minute - [0,59] \*/**

**int tm\_min; /\* minutes after the hour - [0,59] \*/**

**int tm\_hour; /\* hours since midnight - [0,23] \*/**

**int tm\_mday; /\* day of the month - [1,31] \*/**

**int tm\_mon; /\* months since January - [0,11] \*/**

**int tm\_year; /\* years since 1900 \*/**

**int tm\_wday; /\* days since Sunday - [0,6] \*/**

**int tm\_yday; /\* days since January 1 - [0,365] \*/**

**int tm\_isdst; /\* daylight savings time flag \*/**

**};**

**As you can see, there is quite a bit of information you can access. The**

**tm\_isdst member is non-zero if Daylight Savings Time is in effect. The**

**localtime() function returns the time in the local time zone, whereas the**

**gmtime() function returns the time in the UTC (or GMT) time zone. Both**

**localtime() and gmtime() take as their argument a pointer to a long integer**

**that represents the date and time as the number of seconds since January 1,**

**1970 (such a returned by time() ). The return pointers to a tm structure,**

**where the converted data is placed. The following example prints the local**

**date in the familiar mm/dd/yy format:**

**/\* day.c**

**print date in mm/dd/yy format**

**\*/**

**#include <stdio.h>**

**#include <time.h> /\* may need to be #include <sys/time.h> instead \*/**

**int main()**

**{**

**long now, time();**

**struct tm \*today, \*localtime();**

**time (&now);**

**today = localtime (&now);**

**printf("Today is: %d/%d/%d\n", today->tm\_mon + 1, today->tm\_mday,**

**today->tm\_year);**

**exit (0);**

**}**

**Parsing Input**

**When dealing with input from a command line, the first step is to parse**

**(break up) the input line into tokens, which are groups of characters that**

**form syntactic units; examples are words, strings, and special symbols.**

**Following are some sample programs and functions that demonstrate various**

**ways to parse an input line:**

**/\* parse.c**

**Split the input buffer into individual tokens. Tokens are**

**assumed to be separated by space or tab characters.**

**A pointer to each token is stored in an array of pointers.**

**This method is very similar to the argv argument to main().**

**\*/**

**#include <stdio.h>**

**#define EVER ;;**

**#define MAXARG 64**

**int main()**

**{**

**char buf[256];**

**char \*args[MAXARG]; /\* accept MAXARG number of tokens \*/**

**int num\_arg,**

**lcv;**

**for (EVER)**

**{**

**printf("Enter line: ");**

**if ((gets(buf)) == (char \*) NULL)**

**{**

**putchar('\n');**

**exit(0);**

**}**

**num\_arg = parse\_cmd(buf, args);**

**printf("Number of tokens = %d\n",num\_arg);**

**for (lcv = 0; lcv < num\_arg; lcv++)**

**puts(args[lcv]);**

**}**

**}**

**int parse\_cmd(buf, args)**

**char \*buf;**

**char \*\*args;**

**{**

**int count = 0;**

**while (\*buf != '\0' && count < MAXARG)**

**{**

**while ((\*buf == ' ') || (\*buf == '\t'))**

**\*buf++ = '\0';**

**\*args++ = buf;**

**++count;**

**while ((\*buf != '\0') && (\*buf != ' ') && (\*buf != '\t'))**

**buf++;**

**}**

**\*args = (char \*) NULL; /\* make the last element of the array null \*/**

**return(count); /\* return the number of tokens parsed \*/**

**}**

**There is a C library function available that makes parsing a string into**

**tokens very easy; it is strtok(). Following is the above function (parse\_cmd**

**() ) as implemented using strtok():**

**int parse\_cmd(line, args)**

**char \*line;**

**char \*args[];**

**{**

**int count = 0;**

**char \*str, \*strtok();**

**static char delimiters[] = " \t\n";**

**while (( str = strtok(line, delimiters)) != (char \*) NULL)**

**{**

**line = (char \*) NULL;**

**args[count++] = str;**

**}**

**args[count] = (char \*) NULL;**

**return(count);**

**}**

**strtok() takes as arguments a pointer to the input string and a pointer to a**

**string containing the character or characters that delimit the token. In the**

**above example, the delimiters were defined to be a space, tab, or newline.**

**You are free to change the delimiter at any time. If you wish strtok() to**

**parse the complete line, you must pass a null-pointer on the second and**

**subsequent calls to strtok() (note that "line" was set to null inside the**

**body of the while loop). strtok() returns a null-pointer when the end of the**

**input string is reached.**

**strtok() is very useful in parsing the individual path elements as defined in**

**the PATH environment variable (set the delimiter to ":" ).**

**CURSES**

**What is curses? curses is a terminal-independent library of C routines and**

**macros that you use to write "window-based" screen management programs on the**

**UNIX system. curses is designed to let programmers control terminal I/O in**

**an easy fashion. Providing an easy-to-use "human interface" for users is an**

**increasingly important requirement for operating systems. Such a connection**

**between the machine and the humans that use it plays an important role in the**

**overall productivity of the system. curses gets its name from what it does:**

**cursor manipulation.**

**What can curses do? Among the functions to be found in curses are those**

**that:**

**- Move the cursor to any point on the screen**

**- Insert text anywhere on the screen, doing it even in highlight mode**

**- Divide the screen into rectangular areas called windows**

**- Manage each window independently, so you can be scrolling one window**

**while erasing a line in another**

**- Draw a box around a window using a character of your choice**

**- Write output to and read input from a terminal screen**

**- Control the data output and input -- for example, to print output in**

**bold type or prevent it from echoing (printing back on a screen)**

**- Draw simple graphics**

**If these features leave you unimpressed, remember that they are only tools.**

**When you use these tools in your programs, the results can be spectacular.**

**The point is -- curses is easy to use and ready to go -- so that you can**

**concentrate on what you want your program to do. curses will make you**

**program look sharp.**

**Where did curses come from? The author of curses in Ken Arnold who wrote the**

**package while a student at the University of California, Berkeley. At the**

**same time, Bill Joy was writing his editor program, vi. Ken Arnold credits**

**Bill Joy with providing the ideas (as well as code) for creating the**

**capability to generally describe terminals, writing routines to read the**

**terminal database, and implementing routines for optimal cursor movement.**

**The original source of information about curses is Ken Arnold's paper**

**entitled "Screen Updating and Cursor Movement Optimization: A Library**

**Package".**

**What makes curses tick? The original version of curses developed by Ken**

**Arnold incorporated a database known as termcap, or the terminal capabilities**

**database. In System V Release 2, the termcap database was replaced by the**

**terminfo data base, and curses was rewritten to incorporate it. Both of**

**these versions of curses can be used with more than one hundred terminals.**

**The information in the termcap or terminfo database is used by the curses**

**routines to determine what sequence of special characters must be sent to a**

**particular terminal to cause it to clear the screen, move the cursor up one**

**line, delete a line, etc. It is these databases that make curses truly**

**terminal independent, since any terminal not already in the database can be**

**added by a system administrator, and since the structure of both databases**

**allows users to add their own local additions or modifications for a**

**particular terminal.**

**How to use curses -- the basics: There are a couple of things you have to**

**know before you can start using the curses library. First, when you compile**

**a C program that call curses routines, you must specify to the cc command**

**that the curses library is to be linked in with the program. This is done**

**with the -lcurses option, which must be specified after all the C program**

**files. The following is an example cc command line for use on systems that**

**support the terminfo database:**

**cc myprog.c -lcurses**

**Next is an example cc command line for use on systems that support the**

**termcap database:**

**cc myprog.c -lcurses -ltermlib**

**Second, all program files that reference curses routines must include the**

**header file <curses.h> <curses.h> will include the header <stdio.h> so it**

**is not necessary for your program to include it. It won't hurt anything if**

**you do -- it just slows down the compilation. Lastly, before you run a**

**program that uses curses, you must inform curses what type of terminal you**

**have. You do this by setting the shell variable TERM to the type of terminal**

**you are using (e.g. a DEC VT100) and exporting the TERM variable into the**

**environment. This is done in the following manner:**

**$ TERM=vt100**

**$ export TERM**

**This action is usually done for you by your .profile when you log it.**

**The <curses.h> header file contains declarations for variables, constants,**

**data structures and macros. Among the variables are two integer variables**

**that prove to be very useful: LINES and COLS. LINES is automatically set to**

**the number of lines on your terminal; COLS is set to the number of columns.**

**Many of the curses routines address the terminal's screen, in that they move**

**the cursor to a specific place, or address. This address is specified as a**

**particular row and column (specified as arguments to the routine), where the**

**address of the upper left-hand corner is row LINES-1 and column COLS-1**

**(LINES-1, COLS-1). Following is a layout of the terminal screen:**

**IMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;**

**:(0,0) (0, COLS-1):**

**: :**

**: :**

**R : :**

**O : :**

**W : . :**

**: (row,col) :**

**: :**

**: :**

**: :**

**:(LINES-1,0) (LINES-1, COLS-1):**

**HMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<**

**COLUMN**

**-43-**

**All programs using the curses library must have the following basic**

**structure:**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**/\* main program \*/**

**endwin();**

**}**

**The initscr() function must be called before any other curses routines. Its**

**function is to determine the terminal type from the TERM environment variable**

**and to initialize certain data structures and variables (e.g. LINES and**

**COLS). The endwin() function should be called prior to program exit to**

**restore the terminal's original state. Some curses routines change the**

**terminal's characteristics (e.g. go into raw mode and turn off echoing) and**

**must be undone before the program exits; otherwise the terminal is left in an**

**odd state and the user may not know how to change it back. Here is how to**

**fix the terminal if a curses program leaves it in a "funny" state:**

**on System III and System V UNIX systems (including XENIX) type:**

**stty sane ctrl-j**

**note that you must type a control-j and not the return key, since most likely**

**NEWLINE mapping will be off and the RETURN key will not work.**

**on Berkeley UNIX systems type:**

**stty -raw -cbreak -nl echo ek ctrl-j**

**Windows and screens: Conceptually, a window is an independent rectangular**

**area of characters displayed on the screen. Physically, a window is a**

**WINDOW, that is, a C data structure that holds all the information about a**

**window.**

**The Standard Screen - stdscr: The traditional definition of the "standard**

**screen" is a window or a set of windows that fills the entire screen of a**

**video display terminal. The structure that describes stdscr is a WINDOW, or**

**more precisely, a pointer to a WINDOW. A WINDOW is a character array that**

**maintains an image of the terminal screen, known as the screen image. The**

**screen image array in stdscr is automatically made the length and width of**

**the terminal screen. Thus, there is one character in that array for every**

**place on the screen.**

**The Current Screen - curscr: curses does not know directly what the terminal**

**is displaying; it would be even slower to have to query the terminal to find**

**out what character is being displayed at each location. Instead, curses**

**keeps and image of what it thinks the screen looks like in a window called**

**curscr. curscr, like stdscr, is created automatically when you initialize**

**curses with initscr(). curscr is a WINDOW, and has a screen image the size**

**of the physical screen. When refresh() is called, it writes the characters**

**that it is sending to the terminal into their corresponding location in the**

**screen image of curscr. curscr contains the image of the screen as curses**

**thinks it was made to look by the last refresh(). refresh() uses the screen**

**image in curscr to minimize its work. When it goes to refresh a window, it**

**compares the contents of that window to curscr. refresh() assumes that the**

**physical screen looks like curscr so it does not output characters that are**

**the same in curscr and the window that is being refreshed. In this way,**

**refresh() minimizes the number of characters that it sends to the screen and**

**save a great deal of time.**

**The following are a few of the more commonly used curses routines. The list**

**is not comprehensive:**

**Terminal Modes: the terminal modes for I/O are usually set after the call to**

**initscr(). None of the mode setting routines accept parameters.**

**echo() / noecho() These functions allow programmers to turn on or**

**off the terminal driver's echoing to the terminal.**

**The default state is echo on. The function**

**noecho() disables the automatic echoing.**

**nl() / nonl() These functions allow programmers to enable or**

**disable carriage return/newline mappings. When**

**enabled, carriage return is mapped on input to**

**newline and newline is mapped on output to**

**newline/carriage return. The default state is**

**mapping enabled., and nonl() is used to turn this**

**mapping off. It is interesting to note that while**

**mapping is disabled, cursor movement is optimized.**

**cbreak() / nocbreak() Canonical processing (line at a time character**

**processing) is disabled within the terminal driver**

**when calling cbreak(), allowing a break for each**

**character. Interrupt and flow control keys are**

**unaffected. The default state is nocbreak, which**

**enables canonical processing.**

**raw() / noraw() These functions are similar to the cbreak() /**

**nocbreak() functions, except that interrupt and**

**flow control key are also disabled or enabled.**

**savetty() / resetty() The current state of the terminal can be saved**

**into a buffer reserved by curses when calling**

**savetty() function. The last save state can be**

**restored via the resetty() function.**

**gettmode() This function is used to establish the current tty**

**mode while in curses. It reads the baud rate of**

**the terminal, turns off the mapping of carriage**

**returns to line feeds on output, and the expansion**

**of tabs into spaces by the system.**

**I/O Function:**

**addch() This function adds a character to a window at the current**

**cursor position.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**addch('e');**

**refresh();**

**endwin();**

**}**

**mvaddch() This function moves a character into a window at the**

**position specified by the x and y coordinates.**

**#include <curses.h>**

**main()**

**{**

**int x,y;**

**x = 3; y = 10;**

**initscr();**

**mvaddch(x, y, 'e');**

**refresh();**

**endwin();**

**}**

**addstr() This function adds the specified string to a window at the**

**current cursor position.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**addstr("This is a string example.");**

**refresh();**

**endwin();**

**}**

**mvaddstr() This function moves the specified string into a window**

**located at the position specified by the x and y**

**coordinates.**

**#include <curses.h>**

**main()**

**{**

**int x,y;**

**x = 3; y = 10;**

**initscr();**

**mvaddstr(x, y, "This is the string example.");**

**refresh();**

**endwin();**

**}**

**printw() This function outputs formatted strings at the current**

**cursor position and is similar to the printf() function of**

**C, in that multiple arguments may be specified.**

**#include <curses.h>**

**main()**

**{**

**static char \*word = "example";**

**int number = 1;**

**initscr();**

**printw("this is just %d %s of a formatted string!\n",number,word);**

**refresh();**

**endwin();**

**}**

**mvprintw() This function outputs formatted strings at the line**

**specified in y and the column specifed in x. Multiple**

**arguments may be given.**

**#include <curses.h>**

**main()**

**{**

**static char \*word = "example";**

**int number = 1;**

**int x = 3, y = 10;**

**initscr();**

**mvprintw(x ,y, "this is just %d %s of a formatted string!\n",**

**number,word);**

**refresh();**

**endwin();**

**}**

**move() This function moves the cursor to the line/column**

**coordinates given.**

**#include <curses.h>**

**main()**

**{**

**int line = 3, column = 10;**

**initscr();**

**move(line, column);**

**refresh();**

**endwin();**

**}**

**getyx() This function is used to determine and return the current**

**line/column location of the cursor.**

**#include <curses.h>**

**main()**

**{**

**WINDOW \*win;**

**int y, x;**

**initscr();**

**win = newwin(10,5,12,39);**

**getyx(win, y, x)**

**refresh();**

**endwin();**

**}**

**getch() This function is used to read a single character from the**

**keyboard, and returns an integer value. It is similar to**

**the the C standard I/O function getc();**

**#include <curses.h>**

**main()**

**{**

**int in\_char;**

**initscr();**

**in\_char = getch();**

**refresh();**

**endwin();**

**}**

**inch() This function returns the character from under the current**

**cursor position of the terminals screen, in an integer.**

**#include <curses.h>**

**main()**

**{**

**int in\_char;**

**initscr();**

**in\_char = inch();**

**refresh();**

**endwin();**

**}**

**mvinch() This function is used to get the character under the cursor**

**location specified as x and y coordinates. The value**

**returned is an integer.**

**#include <curses.h>**

**main()**

**{**

**int in\_char;**

**initscr();**

**in\_char = mvinch(3, 10);**

**refresh();**

**endwin();**

**}**

**clear() This function completely clear the terminal screen by**

**writing blank spaces to all physical screen locations via**

**calls to erase() and clearok(), and is completed by the**

**next call to refresh().**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**clear()**

**refresh();**

**endwin();**

**}**

**erase() This function is used to insert blank spaces in the**

**physical screen and, like clear(), erases all data on the**

**terminal screen, but does not require a call to refresh().**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**erase()**

**endwin();**

**}**

**clrtobot() This function is used to clear the physical screen from the**

**current cursor position to the bottom of the screen,**

**filling it with blank spaces.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**clrtobot();**

**refresh();**

**endwin();**

**}**

**clrtoeol() This function is used to clear the physical screen from the**

**current cursor position to the end of the physical screen**

**line by filling it with blank spaces.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**clrtoeol();**

**refresh();**

**endwin();**

**}**

**delch() This function deletes the character under the current**

**cursor position, moving all characters on that line**

**(located to the right of the deleted character) one**

**position to the left, and fills the last character position**

**(on that line) with a blank space. The current cursor**

**position remains unchanged.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**delch();**

**refresh();**

**endwin();**

**}**

**mvdelch() This function deletes the character under the cursor**

**position at the line/column specified in y/x. In all other**

**respects, it works the same as the delch() function,**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**mvdelch(3, 10);**

**refresh();**

**endwin();**

**}**

**insch() This function is used to insert the character named in 'c'**

**to be inserted at the current cursor position, causing all**

**characters to the right of the cursor (on that line, only)**

**to shift one space to the right, losing the last character**

**of that line. The cursor is moved one position to the**

**right of the inserted character.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**insch('c');**

**refresh();**

**endwin();**

**}**

**mvinsch() This function inserts the character named in 'c' to the**

**line/column position named in y/x, and otherwise works**

**identically to the insch() function.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**mvinsch(3, 10, 'c');**

**refresh();**

**endwin();**

**}**

**deleteln() This function allows the deletion of the current cursor**

**line, moving all lines located below up one line and**

**filling the last line with blank spaces. The cursor**

**position remains unchanged.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**deleteln();**

**refresh();**

**endwin();**

**}**

**insertln() This function inserts a blank filled line at the current**

**cursor line, moving all lines located below down one line.**

**The bottom line is lost, and the current cursor position is**

**unaffected.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**insertln();**

**refresh();**

**endwin();**

**}**

**refresh() This function is used to update the physical terminal**

**screen from the window buffer and all changes made to that**

**buffer (via curses functions) will be written. If the**

**buffer size is smaller than the physical screen, then only**

**that part of the screen is refreshed, leaving everything**

**else unchanged.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**/\* curses function call(s) here \*/**

**refresh();**

**endwin();**

**}**

**wrefresh() This function is identical to the refresh() function,**

**except that the refresh operation is performed on the named**

**window.**

**#include <curses.h>**

**main()**

**{**

**WINDOW \*win;**

**initscr();**

**/\* curses function call(s) here \*/**

**wrefresh(win);**

**endwin();**

**}**

**initscr() This function call must be present in all programs calling**

**the curses functions. It clears the physical terminal**

**screen and sets up the default modes. It should be the**

**first call to the curses functions when using the library**

**to initialize the terminal.**

**endwin() This function call should be present in any program using**

**the curses functions, and should also be the last function**

**call of that program. It restores all terminal settings to**

**their original state prior to using the initscr() function**

**call and it places the cursor to the lower left hand**

**portion of the screen and terminates a curses program.**

**attrset() This function allows the programmer to set single or**

**multiple terminal attributes. The call attrset(0) resets**

**all attributes to their default state.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**attrset(A\_BOLD);**

**/\* sets character attributes to bold \*/**

**...**

**/\* curses function call(s) here \*/**

**attrset(0);**

**/\* resets all attributes to default \*/**

**refresh();**

**endwin();**

**}**

**attron() This function is used to set the named attribute of a**

**terminal to an on state.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**attron(A\_BOLD);**

**/\* sets character attribute to bold \*/**

**...**

**/\* curses function call(s) here \*/**

**refresh();**

**endwin();**

**}**

**attroff() This function is the opposite of the attron() function and**

**will turn off the named attribute of a terminal.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**attron(A\_BOLD);**

**/\* sets character attribute to bold \*/**

**...**

**/\* curses function call(s) here \*/**

**attroff(A\_BOLD);**

**/\* turns off the bold character attribute \*/**

**refresh();**

**endwin();**

**}**

**standout() This function sets the attribute A\_STANDOUT to an on state,**

**and is nothing more than a convenient way of saying**

**attron(A\_STANDOUT).**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**standout();**

**...**

**/\* curses function call(s) here \*/**

**refresh();**

**endwin();**

**}**

**standend() This function, like standout(), is just a convenient way of**

**saying attroff(A\_STANDOUT), meaning that the A\_STANDOUT**

**attribute is set to an off state. Actually, this function**

**resets all attributes to the off state.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**standout();**

**...**

**/\* curses function call(s) here \*/**

**standend();**

**/\* end of attribute settings \*/**

**refresh();**

**endwin();**

**}**

**box() This function draws a box around the edge of the window.**

**One of its arguments is the horizontal character and the**

**other is the vertical character.**

**#include <curses.h>**

**main()**

**{**

**initscr();**

**box(stdscr, '-', '\*');**

**/\* draws a box around the stdscr \*/**

**/\* horizontal characters are '-' and vertical characters are '\*' \*/**

**refresh();**

**endwin();**

**}**

**Attribute Values: the following is a list of the terminal attributes that**

**may be set on or off using the curses library. It is important to note that**

**all of these attributes may not be available to the physical terminal,**

**depending upon the given terminal's characteristics.**

**A\_STANDOUT - this attribute allows the terminal to display characters in**

**highlight, bold, or some other fashion (depending upon the terminal's**

**characteristics).**

**A\_REVERSE - this attribute allows the terminal to display its characters**

**in reverse video.**

**A\_BOLD - this attribute allows the terminal to display its characters in**

**bold lettering.**

**A\_DIM - this attribute allows the terminal to display its characters at**

**less intensity than normal.**

**A\_UNDERLINE - this attribute allows the terminal to display characters**

**with a horizontal line beneath them (underlined).**

**A\_BLINE - this attribute allows the terminal to display blinking**

**characters that will appear and disappear at a rate depending upon the**

**terminal characteristics.**

**Creating and Removing Windows:**

**WINDOW \*newwin(lines, cols, y1, x1) will create a new window. The new**

**window will have lines lines and cols columns, with the upper left corner**

**located at (y1,x1). newwin() returns a pointer to WINDOW that points at**

**the new window structure. The screen image in the new window is filled**

**with blanks.**

**WINDOW \*subwin(win, lines, cols, y1, x1) will create a sub-window. win is**

**a pointer to the parent window. The other arguments are the same as in**

**newwin(), except lines and cols are interpreted relative to the parent's**

**window and not the terminal screen. A sub-window is a real WINDOW and may**

**have sub-windows just as easily as the parent window.**

**delwin(win) will delete the specified window. delwin() calls the system**

**utility free() to return the space to the pool of available memory. If**

**the window is a sub-window, delwin() does not free() the space because**

**that space is still being used by the parent. Deletint a parent does not**

**free the space occupied by sub-windows. The sub-windows will continue to**

**occupy space, but their screen images will be undefined. You should take**

**care to delete windows in the proper order and when needed in order to**

**maintain good housekeeping of the available memory.**

**Window Specific Functions: these functions are some of the functions above**

**applied to a window. A 'w' is placed before the function name, and the first**

**argument is a pointer to the window.**

**waddch(win, ch) winch(win)**

**waddstr(win,str) winsch(win, c)**

**wclear(win) winsertln(win)**

**wclrtobot(win) wmove(win,y,x)**

**wclrtoeol(win) wprintw(win, fmt, arg1,arg2,...)**

**wdelch(win,c) wrefresh(win)**

**wdeleteln(win) wscanw(win, fmt, agr1, arg2, ...)**

**werase(win) wstandout(win)**

**wgetch(win) wstandend(win)**

**wgetstr(win,str)**

**Move and Act Function: these functions first move the cursor, then perform**

**their action. The function names have a 'mv' placed before the corresponding**

**function above.**

**mvaddch(y,x,ch) mvwaddch(win,y,x,ch)**

**mvaddstr(y,x,str) mvwaddstr(win,y,x,ch)**

**mvdelch(y,x) mvwdelch(win,y,x)**

**mvdeleteln(y,x) mvwdeleteln(win,y,x)**

**mvinch(y,x) mvwinch(win,y,x)**

**mvinsch(y,x,ch) mvwinsch(win,y,x,ch)**

**mvinsertln(y,x) mvwinsertln(win,y,x)**

**The following example program demonstrates a few of the curses funcitons:**

**/\* disptime.c**

**this program displays the time and refreshes the screen once**

**every second, so that the screen resembles a digital clock.**

**\*/**

**#include <curses.h>**

**#include <time.h>**

**#include <signal.h>**

**#define EVER ;;**

**main()**

**{**

**void sig\_catch();**

**long seconds;**

**static char \*title = "The current time is", \*convtime, \*ctime();**

**/\* call sig\_catch if the user hits DELETE/BREAK key \*/**

**signal (SIGINT, sig\_catch);**

**/\* intial setup of curses \*/**

**initscr();**

**/\* output title centered \*/**

**mvaddstr (LINES / 2-1, (COLS - strlen (title)) / 2, title);**

**for (EVER)**

**{**

**/\* get time and convert to ASCII \*/**

**time (&seconds);**

**convtime = ctime (&seconds);**

**/\* display time centered under the title \*/**

**mvaddstr (LINES / 2 , (COLS - strlen (convtime)) /2, convtime);**

**refresh ();**

**sleep (1);**

**}**

**}**

**/\* signal handling routine, call endwin() and exit \*/**

**void sig\_catch()**

**{**

**endwin ();**

**exit (1);**

**}**

**What is awk?**

**awk is one of the more unusual UNIX commands. Named after and by its**

**creators: Aho, Weinberger, and Kernighan, awk combines pattern matching,**

**comparison making, line decomposition, numberical operations, and C-like**

**programming features into one program.**

**awk is a "small" language, in that it lacks some of the more complicated**

**features found in traditional languages like C, Pascal, and Ada. In general,**

**awk omits many mechanisms that support the development of large applications,**

**such as modules and user defined types.**

**Nonetheless, awk is a powerful and general-purpose language, capable of**

**nearly anything you would want a programming language to do. The language**

**omissions foster the development of small applications, as do the robust**

**string manipulation capabilities and the powerful table facility. awk's**

**automatic storage management frees the programmer from having to explicitly**

**keep track of memory -- this alone can cut programming and debugging in half.**

**The Structure of an awk Program**

**Many applications consist of simple collections of patterns and actions.**

**Each time a pattern is recognized in the input, the corresponding action is**

**executed. C code to do this would resemble the following:**

**while ( getRecord() != EOF) {**

**if ( pattern1) { action1 }**

**if ( pattern2) { action2 }**

**...**

**if ( patternN) { actionN }**

**}**

**Because this approach is suitable to so many applications, awk's syntax was**

**specifically streamlined to support it. The corresponding awk program**

**eliminates the outermost control-flow syntax:**

**pattern1 { action1 }**

**pattern2 { action2 }**

**...**

**patternN { actionN }**

**awk read each line in the input file(s) one at a time. When a line is read,**

**each pattern is tested in sequence. Whenever a pattern matches the current**

**line, the corresponding action is executed. This continues until all the**

**input has been processed.**

**awk breaks each input record into fields separated by whitespace (or a**

**specified delimiter). Within the awk program, fields are designated $1, $2,**

**etc., and the variable NF is set to the total number of fields in the current**

**record. The variable $0 stands for the entire record not broken into fields.**

**Patterns resemble boolean expressions, with the addition of regular**

**expression operators, ~, and a syntax for regular expressions contained in**

**slashes added e.g. the pattern $2 ~ /foo/ is true if the field $2 contains a**

**substring "foo". A regular expression without an explicit range is matched**

**against the input record ($0).**

**Several patterns are special. Action connected to BEGIN is executed once**

**before any records are read. END action is executed once after all input**

**records are processed. Omitted patterns match every record.**

**Example Application - Create an Index**

**An index might be appended to the end of a report, or it might be used to**

**extract specific cross-reference information from source code. An awk script**

**to generate an index is characteristically simple:**

**BEGIN { while (getline < "keywords" > 0)**

**KEY[$0] = ""**

**}**

**{ for (k in KEY)**

**if( $0 ~ k)**

**KEY[k] = KEY[k] " " NR**

**}**

**END { for ( k in KEY)**

**print k, KEY[k]**

**}**

**The action associated with BEGIN reads a list of keywords to be indexed from**

**a file called keywords. Each keyword is used as an index into a table called**

**KEY, and the corresponding entries in the table are nulled.**

**The middle action (with no pattern) executes for every input file line. Each**

**line is checked for the presence of each keyword in the KEY table. The line**

**number (NR) is appended to the KEY entry for each match.**

**After all lines are processed, the END action prints each keyword followed by**

**the lines where it appeared.**

**Rapid Prototyping**

**awk is suitable for prototyping applications -- quickly implementing ideas to**

**test feasibility before making a major investment in implementation. If the**

**idea is a bad one, this can be discovered after 50 lines of awk rather than**

**5000 lines of C. Rapid prototyping provides prospective users with a program**

**to "play with" -- the most effective way to find out what the users really**

**want in the final product. Typical awk prototypes are often less that 10% of**

**the length of the equivalent C program. Once you have a suitable prototype**

**program written in awk, you can decide whether to recode the program in a**

**conventional language like C, or just stick with the awk program itself.**

**References**

**1. UNIX System V Programmer's Guide**

**Prentice-Hall, 1987**

**2. Using C on the UNIX System**

**David C. Curry**

**O'Reilly & Associates, Inc. 1989**

**3. UNIX for MS-DOS Programmers**

**Steven Mikes**

**Addison-Wesley, 1989**

**4. Advanced Programmer's Guide to UNIX System V**

**Rebecca Thomas, Lawrence Rogers, Jean Yates**

**Osborne/McGraw-Hill, 1986**

**5. The Design of the UNIX Operating System**

**Maurice Bach**

**Prentice-Hall, 1986**

**6. Topics in C Programming**

**Stephen Kochan and Patrick Wood**

**Hayden Books, 1987**

**7. Advanced UNIX Programming**

**Marc Rochkind**

**Prentice-Hall, 1985**

**8. Programming with Curses**

**John Strang**

**O'Reilly & Associates, 1986**