The Linux Programmer's Guide

Sven Goldt
Sven van der Meer
Scott Burkett
Matt Welsh

Version 0.4
March 1995

[0]...Our continuing mission: to seek out knowledge of C, to explore strange unix commands, and to boldly code where no one has man page 4.

Contents

- Preface
  This guide is far from being complete.
  The first release started at version 0.1 in September 1994. It concentrated on system calls because of lack of manpower and information. Planned are the description of library functions and major kernel changes as well as excursions into important areas like networking, sound, graphics and asynchronous I/O. Maybe some hints about how to build shared libraries and pointers to useful toolkits will later be included.
  This guide will only be a success with generous help in the form of information or perhaps even submission of whole chapters.

- Introduction
  Once upon a time I installed Linux on my PC to learn more about system administration. I tried to install a slip server but it didn't work with shadow and mgetty. I had to patch sliplogin and it worked until the new Linux 1.1 releases. No one could tell me what had happened. There was no documentation about changes since the 0.99 kernel except the kernel change summaries from Russ Nelson, but they didn't help me very much in solving problems.

  The Linux Programmer's Guide is meant to do what the name implies— It is to help Linux programmers understand the peculiarities of Linux. By its nature, this also means that it should be useful when porting programs from other operating systems to Linux. Therefore, this guide must describe the system calls and the major kernel changes which have effects on older programs like serial I/O and networking.

Chapter 1

The Linux operating system

In March 1991 Linus Benedict Torvalds bought the multitasking system Minix for his AT 386. He used it to develop his own multitasking system which he called Linux. In September 1991 he released the first prototype by e-mail to some other Minix users on the internet, thus beginning the Linux project. Many programmers from that point on have supported Linux. They have added device drivers, developed applications, and aimed for POSIX compliance. Today Linux is very powerful, but what is best is that it's free. Work is beeing done to port Linux to other platforms.

Chapter 2

The Linux kernel

The base of Linux is the kernel. You could replace each and every library, but as long as the Linux kernel remained, it would still be Linux. The kernel contains device drivers, memory management, process management and communication management. The kernel hacker gurus follow POSIX guidelines which sometimes makes programming easier and sometimes harder. If your program behaves differently on a new Linux kernel release, chances are that a new POSIX guideline has been implemented. For programming information about the Linux kernel, read the Linux Kernel Hacker's Guide.

Chapter 3

The Linux libc package

libc: ISO 8859.1, < *linux/param.h* >, YP functions, crypt functions,
some basic shadow routines (by default not included), ...
old routines for compatibility in libcompat (by default not activated),
english, french or german error messages,
bsd 4.4lite compatible screen handling routines in libcurses,
bsd compatible routines in libbsd, screen handling routines in libtermcap,
database management routines in libdbm,
mathematic routines in libm, entry to execute programs in crt0.o ???,
byte sex information in libieee ??? (could someone give some infos
instead of laughing ?), user space profiling in libgmon.
I wish someone of the Linux libc developers would write this chapter.
All i can say now that there is going to be a change from the
a.out executable format to the elf (executable and linkable format)
which also means a change in building shared libraries.
Currently both formats (a.out and elf) are supported.

Most parts of the Linux libc package are under the Library GNU Public License, though some are under a special exception copyright like crt0.o. For commercial binary distributions this means a restriction that forbids statically linked executables. Dynamically linked executables are again a special exception and Richard Stallman of the FSF said:

*[...] But it seems to me that we should unambiguously permit distribution of a dynamically linked executable *without* accompanying libraries, provided that the object files that make up the executable are themselves unrestricted according to section 5 [...] So I'll make the decision now to permit this. Actually updating the LGPL will have to wait for when I have time to make and check a new version.*

Chapter 4

System calls

A system call is usually a request to the operating system (kernel) to do a hardware/system-specific or privileged operation. As of Linux-1.2, 140 system calls have been defined. System calls like close() are implemented in the Linux libc. This implementation often involves calling a macro which eventually calls syscall(). Parameters passed to syscall() are the number of the system call followed by the needed arguments. The actual system call numbers can be found in $< linux/unistd.h >$ while $< sys/syscall.h >$ gets updated with a new libc. If new calls appear that don't have a stub in libc yet, you can use syscall(). As an example, you can close a file using syscall() like this (not advised):

```
#include <syscall.h>

extern int syscall(int, ...);

int my_close(int filedescriptor)
{
   return syscall(SYS_close, filedescriptor);
}
```

On the i386 architecture, system calls are limited to 5 arguments besides the system call number because of the number of hardware registers. If you use Linux on another architecture you can check $< asm/unistd.h >$ for the _syscall macros to see how many arguments your hardware supports or how many the developers chose to support. These _syscall macros can be used instead of syscall(), but this is not recommended since such a macro expands to a full function which might already exist in a library. Therefore, only kernel hackers should play with the _syscall macros. To demonstrate, here is the close() example using a _syscall macro.

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

_syscall1(int, close, int, filedescriptor);
```

The _syscall1 macro expands revealing the close() function. Thus we have close() twice–once in libc and once in our program. The return value of syscall()

or a _syscall macro is -1 if the system call failed and 0 or greater on success. Take a look at the global variable errno to see what happened if a system call failed.

The following system calls that are available on BSD and SYS V are not available on Linux:
audit(), auditon(), auditsvc(), fchroot(), getauid(), getdents(), getmsg(), mincore(), poll(), putmsg(), setaudit(), setauid().

Chapter 5

The "swiss army knife" ioctl

ioctl stands for input/output control and is used to manipulate a character device via a filedescriptor. The format of ioctl is
ioctl(unsigned int fd, unsigned int request, unsigned long argument).
The return value is -1 if an error occured and a value greater or equal than 0 if the request succeeded just like other system calls. The kernel distinguishes special and regular files. Special files are mainly found in /dev and /proc. They differ from regular files in that way that they hide an interface to a driver and not to a real (regular) file that contains text or binary data. This is the UNIX philosophy and allows to use normal read/write operations on every file. But if you need to do more with a special file or a regular file you can do it with ... yes, ioctl. You more often need ioctl for special files than for regular files, but it's possible to use ioctl on regular files as well.

Chapter 6

Linux Interprocess Communications

6.1 Introduction

The Linux IPC (Inter-process communication) facilities provide a method for multiple processes to communicate with one another. There are several methods of IPC available to Linux C programmers:

- Half-duplex UNIX pipes
- FIFOs (named pipes)
- SYSV style message queues
- SYSV style semaphore sets
- SYSV style shared memory segments
- Networking sockets (Berkeley style) (not covered in this paper)
- Full-duplex pipes (STREAMS pipes) (not covered in this paper)

These facilities, when used effectively, provide a solid framework for client/server development on any UNIX system (including Linux).

6.2 Half-duplex UNIX Pipes

6.2.1 Basic Concepts

Simply put, a *pipe* is a method of connecting the *standard output* of one process to the *standard input* of another. Pipes are the eldest of the IPC tools, having been around since the earliest incarnations of the UNIX operating system. They provide a method of one-way communications (hence the term half-duplex) between processes.

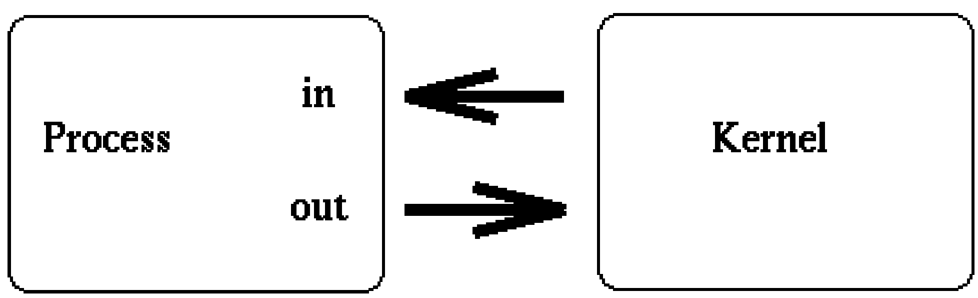
This feature is widely used, even on the UNIX command line (in the shell).

```
ls | sort | lp
```

The above sets up a pipeline, taking the output of ls as the input of sort, and the output of sort as the input of lp. The data is running through a half duplex pipe, traveling (visually) left to right through the pipeline.

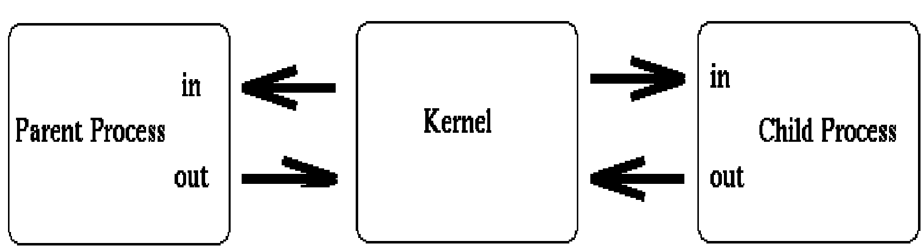
Although most of us use pipes quite religiously in shell script programming, we often do so without giving a second thought to what transpires at the kernel level.

When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe. One descriptor is used to allow a path of input into the pipe (write), while the other is used to obtain data from the pipe (read). At this point, the pipe is of little practical use, as the creating process can only use the pipe to communicate with itself. Consider this representation of a process and the kernel after a pipe has been created:

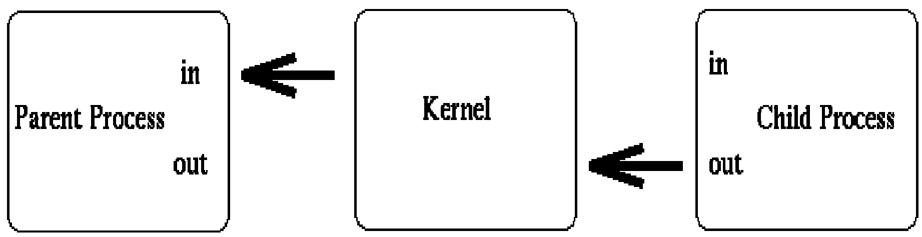


From the above diagram, it is easy to see how the descriptors are connected together. If the process sends data through the pipe (fd0), it has the ability to obtain (read) that information from fd1. However, there is a much larger objective of the simplistic sketch above. While a pipe initially connects a process to itself, data traveling through the pipe moves through the kernel. Under Linux, in particular, pipes are actually represented internally with a valid inode. Of course, this inode resides within the kernel itself, and not within the bounds of any physical file system. This particular point will open up some pretty handy I/O doors for us, as we will see a bit later on.

At this point, the pipe is fairly useless. After all, why go to the trouble of creating a pipe if we are only going to talk to ourself? At this point, the creating process typically forks a child process. Since a child process will inherit any open file descriptors from the parent, we now have the basis for multiprocess communication (between parent and child). Consider this updated version of our simple sketch:



Above, we see that both processes now have access to the file descriptors which constitute the pipeline. It is at this stage, that a critical decision must be made. In which direction do we desire data to travel? Does the child process send information to the parent, or vice-versa? The two processes mutually agree on this issue, and proceed to "close" the end of the pipe that they are not concerned with. For discussion purposes, let's say the child performs some processing, and sends information back through the pipe to the parent. Our newly revised sketch would appear as such:



Construction of the pipeline is now complete! The only thing left to do is make use of the pipe. To access a pipe directly, the same system calls that are used for low-level file I/O can be used (recall that pipes are actually represented internally as a valid inode).

To send data to the pipe, we use the write() system call, and to retrieve data from the pipe, we use the read() system call. Remember, low-level file I/O system calls work with file descriptors! However, keep in mind that certain system calls, such as lseek(), do not work with descriptors to pipes.

6.2.2 Creating Pipes in C

Creating "pipelines" with the C programming language can be a bit more involved than our simple shell example. To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline. After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).

```
SYSTEM CALL: pipe();

PROTOTYPE: int pipe( int fd[2] );
  RETURNS: 0 on success
           -1 on error: errno = EMFILE (no free descriptors)
                                EMFILE (system file table is full)
                                EFAULT (fd array is not valid)

NOTES: fd[0] is set up for reading, fd[1] is set up for writing
```

The first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing. Visually speaking, the output of fd1 becomes the input for fd0. Once again, all data traveling through the pipe moves through the kernel.

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

main()
{
        int     fd[2];

        pipe(fd);
        .
        .
}
```

Remember that an array name in C *decays* into a pointer to its first member. Above, `fd` is equivalent to `&fd[0]`. Once we have established the pipeline, we then fork our new child process:

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

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
}
```

If the parent wants to receive data from the child, it should close fd1, and the child should close fd0. If the parent wants to send data to the child, it should close fd0, and the child should close fd1. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with. On a technical note, the EOF will never be returned if the unnecessary ends of the pipe are not explicitly closed.

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

```
        main()
        {
                int     fd[2];
                pid_t   childpid;

                pipe(fd);

                if((childpid = fork()) == -1)
                {
                        perror("fork");
                        exit(1);
                }

                if(childpid == 0)
                {
                        /* Child process closes up input side of pipe */
                        close(fd[0]);
                }
                else
                {
                        /* Parent process closes up output side of pipe */
                        close(fd[1]);
                }
                .
                .
        }
```

As mentioned previously, once the pipeline has been established, the file descriptors may be treated like descriptors to normal files.

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
```

```
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, strlen(string));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

Often, the descriptors in the child are duplicated onto standard input or output. The child can then exec() another program, which inherits the standard streams. Let's look at the dup() system call:

```
  SYSTEM CALL: dup();

  PROTOTYPE: int dup( int oldfd );
    RETURNS: new descriptor on success
             -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                  EBADF (newfd is out of range)
                                  EMFILE (too many descriptors for the process)

  NOTES: the old descriptor is not closed!  Both may be used interchangeably
```

Although the old descriptor and the newly created descriptor can be used interchangeably, we will typically close one of the standard streams first. The dup() system call uses the lowest-numbered, unused descriptor for the new one.

Consider:

```
.
.
childpid = fork();

if(childpid == 0)
{
        /* Close up standard input of the child */
        close(0);

        /* Duplicate the input side of pipe to stdin */
        dup(fd[0]);
        execlp("sort", "sort", NULL);
        .
}
```

Since file descriptor 0 (stdin) was closed, the call to dup() duplicated the input descriptor of the pipe (fd0) onto its standard input. We then make a call to execlp(), to overlay the child's text segment (code) with that of the sort program. Since newly exec'd programs inherit standard streams from their spawners, it actually inherits the input side of the pipe as its standard input! Now, anything that the original parent process sends to the pipe, goes into the sort facility.

There is another system call, dup2(), which can be used as well. This particular call originated with Version 7 of UNIX, and was carried on through the BSD releases and is now required by the POSIX standard.

```
SYSTEM CALL: dup2();

PROTOTYPE: int dup2( int oldfd, int newfd );
  RETURNS: new descriptor on success
           -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                EBADF (newfd is out of range)
                                EMFILE (too many descriptors for the process)

NOTES: the old descriptor is closed with dup2()!
```

With this particular call, we have the close operation, and the actual descriptor duplication, wrapped up in one system call. In addition, it is guaranteed to be atomic, which essentially means that it will never be interrupted by an arriving signal. The entire operation will transpire before returning control to the kernel for signal dispatching. With the original dup() system call, programmers had to perform a close() operation before calling it. That resulted in two system calls, with a small degree of vulnerability in the brief amount of time which elapsed between them. If a signal arrived during that brief instance, the descriptor duplication would fail. Of course, dup2() solves this problem for us.

Consider:

```
.
.
childpid = fork();

if(childpid == 0)
{
        /* Close stdin, duplicate the input side of pipe to stdin */
        dup2(0, fd[0]);
        execlp("sort", "sort", NULL);
        .
        .
}
```

6.2.3 Pipes the Easy Way!

If all of the above ramblings seem like a very round-about way of creating and utilizing pipes, there is an alternative.

```
LIBRARY FUNCTION: popen();

PROTOTYPE: FILE *popen ( char *command, char *type);
  RETURNS: new file stream on success
           NULL on unsuccessful fork() or pipe() call

NOTES: creates a pipe, and performs fork/exec operations using "command"
```

This standard library function creates a half-duplex pipeline by calling pipe() internally. It then forks a child process, execs the Bourne shell, and executes the "command" argument within the shell. Direction of data flow is determined by the second argument, "type". It can be "r" or "w", for "read" or "write". It cannot be both! Under Linux, the pipe will be opened up in the mode specified by the first character of the "type" argument. So, if you try to pass "rw", it will only open it up in "read" mode.

While this library function performs quite a bit of the dirty work for you, there is a substantial tradeoff. You lose the fine control you once had by using the pipe() system call, and handling the fork/exec yourself. However, since the Bourne shell is used directly, shell metacharacter expansion (including wildcards) is permissible within the "command" argument.

Pipes which are created with popen() must be closed with pclose(). By now, you have probably realized that popen/pclose share a striking resemblance to the standard file stream I/O functions fopen() and fclose().

```
  LIBRARY FUNCTION: pclose();

  PROTOTYPE: int pclose( FILE *stream );
    RETURNS: exit status of wait4() call
             -1 if "stream" is not valid, or if wait4() fails

  NOTES: waits on the pipe process to terminate, then closes the stream.
```

The pclose() function performs a wait4() on the process forked by popen(). When it returns, it destroys the pipe and the file stream. Once again, it is synonymous with the fclose() function for normal stream-based file I/O.

Consider this example, which opens up a pipe to the sort command, and proceeds to sort an array of strings:

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: popen1.c
 *****************************************************************************/

#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
        int  cntr;
        FILE *pipe_fp;
        char *strings[MAXSTRS] = { "echo", "bravo", "alpha",
                                   "charlie", "delta"};

        /* Create one way pipe line with call to popen() */
        if (( pipe_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Processing loop */
        for(cntr=0; cntr<MAXSTRS; cntr++) {
                fputs(strings[cntr], pipe_fp);
                fputc('\n', pipe_fp);
        }

        /* Close the pipe */
```

```
        pclose(pipe_fp);

        return(0);
}
```

Since `popen()` uses the shell to do its bidding, all shell expansion characters and metacharacters are available for use! In addition, more advanced techniques such as redirection, and even output piping, can be utilized with `popen()`. Consider the following sample calls:

```
        popen("ls ~scottb", "r");
        popen("sort > /tmp/foo", "w");
        popen("sort | uniq | more", "w");
```

As another example of popen(), consider this small program, which opens up two pipes (one to the ls command, the other to sort):

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: popen2.c
 *****************************************************************************/

#include <stdio.h>

int main(void)
{
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];

        /* Create one way pipe line with call to popen() */
        if (( pipein_fp = popen("ls", "r")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Create one way pipe line with call to popen() */
        if (( pipeout_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Processing loop */
```

```
        while(fgets(readbuf, 80, pipein_fp))
                fputs(readbuf, pipeout_fp);

        /* Close the pipes */
        pclose(pipein_fp);
        pclose(pipeout_fp);

        return(0);
}
```

For our final demonstration of popen(), let's create a generic program that opens up a pipeline between a passed command and filename:

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: popen3.c
 *****************************************************************************/

#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *pipe_fp, *infile;
        char readbuf[80];

        if( argc != 3) {
                fprintf(stderr, "USAGE:  popen3 [command] [filename]\n");
                exit(1);
        }

        /* Open up input file */
        if (( infile = fopen(argv[2], "rt")) == NULL)
        {
                perror("fopen");
                exit(1);
        }

        /* Create one way pipe line with call to popen() */
        if (( pipe_fp = popen(argv[1], "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }
```

```
        /* Processing loop */
        do {
                fgets(readbuf, 80, infile);
                if(feof(infile)) break;

                fputs(readbuf, pipe_fp);
        } while(!feof(infile));

        fclose(infile);
        pclose(pipe_fp);

        return(0);
}
```

Try this program out, with the following invocations:

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

6.2.4 Atomic Operations with Pipes

In order for an operation to be considered "atomic", it must not be interrupted for any reason at all. The entire operation occurs at once. The POSIX standard dictates in /usr/include/posix1_lim.h that the maximum buffer size for an atomic operation on a pipe is:

```
#define _POSIX_PIPE_BUF         512
```

Up to 512 bytes can be written or retrieved from a pipe atomically. Anything that crosses this threshold will be split, and not atomic. Under Linux, however, the atomic operational limit is defined in "linux/limits.h" as:

```
#define PIPE_BUF        4096
```

As you can see, Linux accommodates the minimum number of bytes required by POSIX, quite considerably I might add. The atomicity of a pipe operation becomes important when more than one process is involved (FIFOS). For example, if the number of bytes written to a pipe exceeds the atomic limit for a single operation, and multiple processes are writing to the pipe, the data will be "interleaved" or "chunked". In other words, one process may insert data into the pipeline between the writes of another.

6.2.5 Notes on half-duplex pipes:

- Two way pipes can be created by opening up two pipes, and properly reassigning the file descriptors in the child process.

- The pipe() call must be made BEFORE a call to fork(), or the descriptors will not be inherited by the child! (same for popen()).
- With half-duplex pipes, any connected processes must share a related ancestry. Since the pipe resides within the confines of the kernel, any process that is not in the ancestry for the creator of the pipe has no way of addressing it. This is not the case with named pipes (FIFOS).

6.3 Named Pipes (FIFOs - First In First Out)

6.3.1 Basic Concepts

A named pipe works much like a regular pipe, but does have some noticeable differences.

- Named pipes exist as a device special file in the file system.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

6.3.2 Creating a FIFO

There are several ways of creating a named pipe. The first two can be done directly from the shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

The above two commands perform identical operations, with one exception. The mkfifo command provides a hook for altering the permissions on the FIFO file directly after creation. With mknod, a quick call to the chmod command will be necessary.

FIFO files can be quickly identified in a physical file system by the "p" indicator seen here in a long directory listing:

```
$ ls -l MYFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MYFIFO|
```

Also notice the vertical bar ("pipe sign") located directly after the file name. Another great reason to run Linux, eh?

To create a FIFO in C, we can make use of the mknod() system call:

```
LIBRARY FUNCTION: mknod();

PROTOTYPE: int mknod( char *pathname, mode_t mode, dev_t dev);
  RETURNS: 0 on success,
           -1 on error: errno = EFAULT (pathname invalid)
```

```
                                        EACCES (permission denied)
                                        ENAMETOOLONG (pathname too long)
                                        ENOENT (invalid pathname)
                                        ENOTDIR (invalid pathname)
                                        (see man page for mknod for others)

  NOTES: Creates a filesystem node (file, device file, or FIFO)
```

I will leave a more detailed discussion of mknod() to the man page, but let's consider a simple example of FIFO creation from C:

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In this case, the file "/tmp/MYFIFO" is created as a FIFO file. The requested permissions are "0666", although they are affected by the umask setting as follows:

```
final_umask = requested_permissions & ~original_umask
```

A common trick is to use the umask() system call to temporarily zap the umask value:

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In addition, the third argument to mknod() is ignored unless we are creating a device file. In that instance, it should specify the major and minor numbers of the device file.

6.3.3 FIFO Operations

I/O operations on a FIFO are essentially the same as for normal pipes, with once major exception. An "open" system call or library function should be used to physically open up a channel to the pipe. With half-duplex pipes, this is unnecessary, since the pipe resides in the kernel and not on a physical filesystem. In our examples, we will treat the pipe as a stream, opening it up with fopen(), and closing it with fclose().

Consider a simple server process:

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoserver.c
 *****************************************************************************/

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

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

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
        }

        return(0);
}
```

Since a FIFO blocks by default, run the server in the background after you compile it:

```
        $ fifoserver&
```

We will discuss a FIFO's blocking action in a moment. First, consider the following simple client frontend to our server:

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoclient.c
 *****************************************************************************/

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

#define FIFO_FILE       "MYFIFO"
```

```
int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

6.3.4 Blocking Actions on a FIFO

Normally, blocking occurs on a FIFO. In other words, if the FIFO is opened for reading, the process will "block" until some other process opens it for writing. This action works vice-versa as well. If this behavior is undesirable, the O_NONBLOCK flag can be used in an open() call to disable the default blocking action.

In the case with our simple server, we just shoved it into the background, and let it do its blocking there. The alternative would be to jump to another virtual console and run the client end, switching back and forth to see the resulting action.

6.3.5 The Infamous SIGPIPE Signal

On a last note, pipes must have a reader and a writer. If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline.

6.4 System V IPC

6.4.1 Fundamental Concepts

With System V, AT&T introduced three new forms of IPC facilities (message queues, semaphores, and shared memory). While the POSIX committee has not yet completed its standardization of these facilities, most implementations do support these. In addition, Berkeley (BSD) uses sockets as its primary form of IPC, rather than the System V elements. Linux has the ability to use both forms of IPC (BSD and System V), although we will not discuss sockets until a later chapter.

The Linux implementation of System V IPC was authored by *Krishna Balasubramanian*, at `balasub@cis.ohio-state.edu`.

IPC Identifiers

Each IPC *object* has a unique IPC identifier associated with it. When we say "IPC object", we are speaking of a single message queue, semaphore set, or shared memory segment. This identifier is used within the kernel to uniquely identify an IPC object. For example, to access a particular shared memory segment, the only item you need is the unique ID value which has been assigned to that segment.

The uniqueness of an identifier is relevant to the *type* of object in question. To illustrate this, assume a numeric identifier of "12345". While there can never be two message queues with this same identifier, there exists the distinct possibility of a message queue and, say, a shared memory segment, which have the same numeric identifier.

IPC Keys

To obtain a unique ID, a *key* must be used. The key must be mutually agreed upon by both client and server processes. This represents the first step in constructing a client/server framework for an application.

When you use a telephone to call someone, you must know their number. In addition, the phone company must know how to relay your outgoing call to its final destination. Once the other party responds by answering the telephone call, the connection is made.

In the case of System V IPC facilities, the "telephone" correllates directly with the type of object being used. The "phone company", or routing method, can be directly associated with an IPC key.

The key can be the same value every time, by hardcoding a key value into an application. This has the disadvantage of the key possibly being in use already. Often, the ftok() function is used to generate key values for both the client and the server.

```
LIBRARY FUNCTION: ftok();

PROTOTYPE: key_t ftok ( char *pathname, char proj );
  RETURNS: new IPC key value if successful
           -1 if unsuccessful, errno set to return of stat() call
```

The returned key value from ftok() is generated by combining the inode number and minor device number from the file in argument one, with the one character project indentifier in the second argument. This doesn't guarantee uniqueness, but an application can check for collisions and retry the key generation.

```
key_t   mykey;
mykey = ftok("/tmp/myapp", 'a');
```

In the above snippet, the directory `/tmp/myapp` is combined with the one letter identifier of `'a'`. Another common example is to use the current directory:

```
key_t   mykey;
mykey = ftok(".", 'a');
```

The key generation algorithm used is completely up to the discretion of the application programmer. As long as measures are in place to prevent race conditions, deadlocks, etc, any method is viable. For our demonstration purposes, we will use the ftok() approach. If we assume that each client process will be running from a unique "home" directory, the keys generated should suffice for our needs.

The key value, however it is obtained, is used in subsequent IPC system calls to create or gain access to IPC objects.

The `ipcs` Command

The `ipcs` command can be used to obtain the status of all System V IPC objects. The Linux version of this tool was also authored by *Krishna Balasubramanian*.

```
ipcs     -q:    Show only message queues
ipcs     -s:    Show only semaphores
ipcs     -m:    Show only shared memory
ipcs --help:    Additional arguments
```

By default, all three categories of objects are shown. Consider the following sample output of `ipcs`:

```
------ Shared Memory Segments --------
shmid     owner     perms     bytes     nattch    status

------ Semaphore Arrays --------
semid     owner     perms     nsems     status

------ Message Queues --------
msqid     owner     perms     used-bytes  messages
0         root      660       5           1
```

Here we see a single message queue which has an identifier of "0". It is owned by the user *root*, and has octal permissions of `660`, or `-rw-rw---`. There is one message in the queue, and that message has a total size of 5 bytes.

The `ipcs` command is a very powerful tool which provides a peek into the kernel's storage mechanisms for IPC objects. Learn it, use it, revere it.

The `ipcrm` Command

The **ipcrm** command can be used to remove an IPC object from the kernel. While IPC objects can be removed via system calls in user code (we'll see how in a moment), the need often arises, especially under development environments, to remove IPC objects manually. Its usage is simple:

```
ipcrm <msg | sem | shm>  <IPC ID>
```

Simply specify whether the object to be deleted is a message queue (*msg*), a semaphore set (*sem*), or a shared memory segment (*shm*). The IPC ID can be obtained by the **ipcs** command. You have to specify the type of object, since identifiers are unique among the same type (recall our discussion of this earlier).

6.4.2 Message Queues

Basic Concepts

Message queues can be best described as an internal linked list within the kernel's addressing space. Messages can be sent to the queue in order and retrieved from the queue in several different ways. Each message queue (of course) is uniquely identified by an IPC identifier.

Internal and User Data Structures

The key to fully understanding such complex topics as System V IPC is to become intimately familiar with the various internal data structures that reside within the confines of the kernel itself. Direct access to some of these structures is necessary for even the most primitive operations, while others reside at a much lower level.

Message buffer The first structure we'll visit is the **msgbuf** structure. This particular data structure can be thought of as a *template* for message data. While it is up to the programmer to define structures of this type, it is imperative that you understand that there **is** actually a structure of type **msgbuf**. It is declared in **linux/msg.h** as follows:

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
    long mtype;         /* type of message */
    char mtext[1];      /* message text */
};
```

There are two members in the **msgbuf** structure:

`mtype`

The message *type*, represented in a positive number. This *must* be a positive number!

`mtext`

The message data itself.

The ability to assign a given message a *type*, essentially gives you the capability to *multiplex* messages on a single queue. For instance, client processes could be assigned a magic number, which could be used as the message type for messages sent from a server process. The server itself could use some other number, which clients could use to send messages to it. In another scenario, an application could mark error messages as having a message type of 1, request messages could be type 2, etc. The possibilities are endless.

On another note, do not be misled by the almost too-descriptive name assigned to the message data element (`mtext`). This field is not restricted to holding only arrays of characters, but any data, in any form. The field itself is actually completely arbitrary, since this structure gets redefined by the application programmer. Consider this redefinition:

```
struct my_msgbuf {
        long    mtype;          /* Message type */
        long    request_id;     /* Request identifier */
        struct  client info;    /* Client information structure */
};
```

Here we see the message type, as before, but the remainder of the structure has been replaced by two other elements, one of which is another structure! This is the beauty of message queues. The kernel makes no translations of data whatsoever. Any information can be sent.

There does exist an internal limit, however, of the maximum size of a given message. In Linux, this is defined in `linux/msg.h` as follows:

```
#define MSGMAX  4056   /* <= 4056 */   /* max size of message (bytes) */
```

Messages can be no larger than 4,056 bytes in total size, including the `mtype` member, which is 4 bytes in length (`long`).

Kernel `msg` structure The kernel stores each message in the queue within the framework of the `msg` structure. It is defined for us in `linux/msg.h` as follows:

```
/* one msg structure for each message */
struct msg {
    struct msg *msg_next;   /* next message on queue */
    long  msg_type;
    char *msg_spot;         /* message text address */
    short msg_ts;           /* message text size */
};
```

`msg_next`

This is a pointer to the next message in the queue. They are stored as a singly linked list within kernel addressing space.

`msg_type`

This is the message type, as assigned in the user structure `msgbuf`.

`msg_spot`

A pointer to the beginning of the message body.

`msg_ts`

The length of the message text, or body.

Kernel `msqid_ds` structure Each of the three types of IPC objects has an internal data structure which is maintained by the kernel. For message queues, this is the `msqid_ds` structure. The kernel creates, stores, and maintains an instance of this structure for every message queue created on the system. It is defined in `linux/msg.h` as follows:

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue */
    struct msg *msg_last;   /* last message in queue */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* last receive pid */
};
```

While you will rarely have to concern yourself with most of the members of this structure, a brief description of each is in order to complete our tour:

`msg_perm`

An instance of the `ipc_perm` structure, which is defined for us in `linux/ipc.h`. This holds the permission information for the message queue, including the access permissions, and information about the creator of the queue (uid, etc).

`msg_first`

Link to the first message in the queue (the head of the list).

`msg_last`

Link to the last message in the queue (the tail of the list).

`msg_stime`

Timestamp (`time_t`) of the last message that was sent to the queue.

`msg_rtime`

Timestamp of the last message retrieved from the queue.

`msg_ctime`

Timestamp of the last "change" made to the queue (more on this later).

`wwait`

and

`rwait`

Pointers into the kernel's *wait queue*. They are used when an operation on a message queue deems the process go into a sleep state (i.e. queue is full and the process is waiting for an opening).

`msg_cbytes`

Total number of bytes residing on the queue (sum of the sizes of all messages).

`msg_qnum`

Number of messages currently in the queue.

`msg_qbytes`

Maximum number of bytes on the queue.

`msg_lspid`

The PID of the process who sent the last message.

`msg_lrpid`

The PID of the process who retrieved the last message.

Kernel `ipc_perm` structure The kernel stores permission information for IPC objects in a structure of type `ipc_perm`. For example, in the internal structure for a message queue described above, the `msg_perm` member is of this type. It is declared for us in `linux/ipc.h` as follows:

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
```

```
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* slot usage sequence number */
};
```

All of the above are fairly self-explanatory. Stored along with the IPC key of the object is information about both the creator and owner of the object (they may be different). The octal access modes are also stored here, as an `unsigned short`. Finally, the *slot usage sequence* number is stored at the end. Each time an IPC object is closed via a system call (destroyed), this value gets incremented by the maximum number of IPC objects that can reside in a system. Will you have to concern yourself with this value? No.

NOTE: *There is an excellent discussion on this topic, and the security reasons as to its existence and behavior, in Richard Stevens'* **UNIX Network Programming** *book, pp. 125.*

SYSTEM CALL: msgget()

In order to create a new message queue, or access an existing queue, the `msgget()` system call is used.

```
  SYSTEM CALL: msgget();

  PROTOTYPE: int msgget ( key_t key, int msgflg );
    RETURNS: message queue identifier on success
             -1 on error: errno = EACCESS (permission denied)
                                  EEXIST (Queue exists, cannot create)
                                  EIDRM (Queue is marked for deletion)
                                  ENOENT (Queue does not exist)
                                  ENOMEM (Not enough memory to create queue)
                                  ENOSPC (Maximum queue limit exceeded)
  NOTES:
```

The first argument to `msgget()` is the key value (in our case returned by a call to `ftok()`). This key value is then compared to existing key values that exist within the kernel for other message queues. At that point, the open or access operation is dependent upon the contents of the `msgflg` argument.

IPC_CREAT

Create the queue if it doesn't already exist in the kernel.

IPC_EXCL

When used with IPC_CREAT, fail if queue already exists.

If `IPC_CREAT` is used alone, `msgget()` either returns the message queue identifier for a newly created message queue, or returns the identifier for a queue which exists with the same key value. If `IPC_EXCL` is used along with `IPC_CREAT`, then either a new queue is created, or if the queue exists, the call fails with -1. `IPC_EXCL` is useless by itself, but when combined with `IPC_CREAT`, it can be used as a facility to guarantee that no existing queue is opened for access.

An optional octal mode may be OR'd into the mask, since each IPC object has permissions that are similar in functionality to file permissions on a UNIX file system!

Let's create a quick wrapper function for opening or creating message queue:

```
int open_queue( key_t keyval )
{
        int     qid;

        if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(qid);
}
```

Note the use of the explicit permissions of `0660`. This small function either returns a message queue identifier (`int`), or -1 on error. The key value must be passed to it as its only argument.

SYSTEM CALL: msgsnd()

Once we have the queue identifier, we can begin performing operations on it. To deliver a message to a queue, you use the `msgsnd` system call:

```
  SYSTEM CALL: msgsnd();

  PROTOTYPE: int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg );
    RETURNS: 0 on success
             -1 on error: errno = EAGAIN (queue is full, and IPC_NOWAIT was asserted)
                                  EACCES (permission denied, no write permission)
                                  EFAULT (msgp address isn't accessable - invalid)
                                  EIDRM  (The message queue has been removed)
                                  EINTR  (Received a signal while waiting to write)
                                  EINVAL (Invalid message queue identifier, nonpositive
                                          message type, or invalid message size)
                                  ENOMEM (Not enough memory to copy message buffer)
  NOTES:
```

The first argument to `msgsnd` is our queue identifier, returned by a previous call to `msgget`. The second argument, `msgp`, is a pointer to our redeclared and loaded message buffer. The `msgsz` argument contains the size of the message in bytes, excluding the length of the message type (4 byte long).

The `msgflg` argument can be set to 0 (ignored), or:

IPC_NOWAIT

If the message queue is full, then the message is not written to the queue, and control is returned to the calling process. If not specified, then the calling process will suspend (block) until the message can be written.

Let's create another wrapper function for sending messages:

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);

        if((result = msgsnd( qid, qbuf, length, 0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

This small function attempts to send the message residing at the passed address (`qbuf`) to the message queue designated by the passed queue identifier (`qid`). Here is a sample code snippet utilizing the two wrapper functions we have developed so far:

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

main()
{
        int    qid;
        key_t  msgkey;
        struct mymsgbuf {
                long    mtype;          /* Message type */
                int     request;        /* Work request number */
                double  salary;         /* Employee's salary */
        } msg;

        /* Generate our IPC key value */
        msgkey = ftok(".", 'm');

        /* Open/create the queue */
```

```
        if(( qid = open_queue( msgkey)) == -1) {
                perror("open_queue");
                exit(1);
        }

        /* Load up the message with arbitrary test data */
        msg.mtype   = 1;        /* Message type must be a positive number! */
        msg.request = 1;        /* Data element #1 */
        msg.salary  = 1000.00;  /* Data element #2 (my yearly salary!) */

        /* Bombs away! */
        if((send_message( qid, &msg )) == -1) {
                perror("send_message");
                exit(1);
        }
}
```

After creating/opening our message queue, we proceed to load up the message buffer with test data (*note the lack of character data to illustrate our point about sending binary information*). A quick call to `send_message` merrily distributes our message out to the message queue.

Now that we have a message on our queue, try the `ipcs` command to view the status of your queue. Now let's turn the discussion to actually retrieving the message from the queue. To do this, you use the `msgrcv()` system call:

```
  SYSTEM CALL: msgrcv();
  PROTOTYPE: int msgrcv ( int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg );
    RETURNS: Number of bytes copied into message buffer
             -1 on error: errno = E2BIG  (Message length is greater than msgsz, no MSG_NOERROR)
                                  EACCES (No read permission)
                                  EFAULT (Address pointed to by msgp is invalid)
                                  EIDRM  (Queue was removed during retrieval)
                                  EINTR  (Interrupted by arriving signal)
                                  EINVAL (msgqid invalid, or msgsz less than 0)
                                  ENOMSG (IPC_NOWAIT asserted, and no message exists
                                          in the queue to satisfy the request)
  NOTES:
```

Obviously, the first argument is used to specify the queue to be used during the message retrieval process (should have been returned by an earlier call to `msgget`). The second argument (`msgp`) represents the address of a message buffer variable to store the retrieved message at. The third argument (`msgsz`) represents the size of the message buffer structure, excluding the length of the `mtype` member. Once again, this can easily be calculated as:

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

The fourth argument (`mtype`) specifies the *type* of message to retrieve from the queue. The kernel will search the queue for the oldest message having a matching type, and will return a copy of it in the address pointed to by the `msgp` argument. One special case exists. If the `mtype` argument is passed with a value of zero, then the oldest message on the queue is returned, regardless of type.

If **IPC_NOWAIT** is passed as a flag, and no messages are available, the call returns ENOMSG to the calling process. Otherwise, the calling process blocks until a message arrives in the queue that satisfies the `msgrcv()` parameters. If the queue is deleted while a client is waiting on a message, EIDRM is returned. EINTR is returned if a signal is caught while the process is in the middle of blocking, and waiting for a message to arrive.

Let's examine a quick wrapper function for retrieving a message from our queue:

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);

        if((result = msgrcv( qid, qbuf, length, type,  0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

After successfully retrieving a message from the queue, the message entry within the queue is destroyed.

The **MSG_NOERROR** bit in the `msgflg` argument provides some additional capabilities. If the size of the physical message data is greater than `msgsz`, and **MSG_NOERROR** is asserted, then the message is truncated, and only `msgsz` bytes are returned. Normally, the `msgrcv()` system call returns -1 (**E2BIG**), and the message will remain on the queue for later retrieval. This behavior can used to create another wrapper function, which will allow us to "peek" inside the queue, to see if a message has arrived that satisfies our request:

```
int peek_message( int qid, long type )
{
        int     result, length;

        if((result = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
        {
```

```
                if(errno == E2BIG)
                        return(TRUE);
        }

        return(FALSE);
}
```

Above, you will notice the lack of a buffer address and a length. In this particular case, we *want* the call to fail. However, we check for the return of **E2BIG** which indicates that a message does exist which matches our requested type. The wrapper function returns **TRUE** on success, **FALSE** otherwise. Also note the use of the **IPC_NOWAIT** flag, which prevents the blocking behavior described earlier.

SYSTEM CALL: msgctl()

Through the development of the wrapper functions presented earlier, you now have a simple, somewhat elegant approach to creating and utilizing message queues in your applications. Now, we will turn the discussion to directly manipulating the internal structures associated with a given message queue.

To perform control operations on a message queue, you use the `msgctl()` system call.

```
  SYSTEM CALL: msgctl();
  PROTOTYPE: int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
    RETURNS: 0 on success
             -1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
                                  EFAULT (Address pointed to by buf is invalid with IPC_SET and
                                          IPC_STAT commands)
                                  EIDRM  (Queue was removed during retrieval)
                                  EINVAL (msgqid invalid, or msgsz less than 0)
                                  EPERM  (IPC_SET or IPC_RMID command was issued, but
                                          calling process does not have write (alter)
                                          access to the queue)
  NOTES:
```

Now, common sense dictates that direct manipulation of the internal kernel data structures could lead to some late night fun. Unfortunately, the resulting duties on the part of the programmer could only be classified as fun if you like trashing the IPC subsystem. By using `msgctl()` with a selective set of commands, you have the ability to manipulate those items which are less likely to cause grief. Let's look at these commands:

IPC_STAT

Retrieves the msqid_ds structure for a queue, and stores it in the address of the buf argument.

IPC_SET

Sets the value of the ipc_perm member of the msqid_ds structure for a queue. Takes the values from the buf argument.

IPC_RMID

Removes the queue from the kernel.

Recall our discussion about the internal data structure for message queues (`msqid_ds`). The kernel maintains an instance of this structure for each queue which exists in the system. By using the **IPC_STAT** command, we can retrieve a copy of this structure for examination. Let's look at a quick wrapper function that will retrieve the internal structure and copy it into a passed address:

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
        if( msgctl( qid, IPC_STAT, qbuf) == -1)
        {
                return(-1);
        }

        return(0);
}
```

If we are unable to copy the internal buffer, -1 is returned to the calling function. If all went well, a value of 0 (zero) is returned, and the passed buffer should contain a copy of the internal data structure for the message queue represented by the passed queue identifier (`qid`).

Now that we have a copy of the internal data structure for a queue, what attributes can be manipulated, and how can we alter them? The only modifiable item in the data structure is the `ipc_perm` member. This contains the permissions for the queue, as well as information about the owner and creator. However, the only members of the `ipc_perm` structure that are modifiable are `mode`, `uid`, and `gid`. You can change the owner's user id, the owner's group id, and the access permissions for the queue.

Let's create a wrapper function designed to change the mode of a queue. The mode must be passed in as a character array (i.e. *"660"*).

```
int change_queue_mode( int qid, char *mode )
{
        struct msqid_ds tmpbuf;

        /* Retrieve a current copy of the internal data structure */
        get_queue_ds( qid, &tmpbuf);

        /* Change the permissions using an old trick */
        sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);
```

```
        /* Update the internal data structure */
        if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
        {
                return(-1);
        }

        return(0);
}
```

We retrieve a current copy of the internal data structure by a quick call to our `get_queue_ds` wrapper function. We then make a call to `sscanf()` to alter the `mode` member of the associated `msg_perm` structure. No changes take place, however, until the new copy is used to update the internal version. This duty is performed by a call to `msgctl()` using the **IPC_SET** command.

BE CAREFUL! It is possible to alter the permissions on a queue, and in doing so, inadvertently lock yourself out! Remember, these IPC objects don't go away unless they are properly removed, or the system is rebooted. So, even if you can't see a queue with `ipcs` doesn't mean that it isn't there.

> *To illustrate this point, a somewhat humorous anecdote seems to be in order. While teaching a class on UNIX internals at the University of South Florida, I ran into a rather embarrassing stumbling block. I had dialed into their lab server the night before, in order to compile and test the labwork to be used in the week-long class. In the process of my testing, I realized that I had made a typo in the code used to alter the permissions on a message queue. I created a simple message queue, and tested the sending and receiving capabilities with no incident. However, when I attempted to change the mode of the queue from "660" to "600", the resulting action was that I was locked out of my own queue! As a result, I could not test the message queue labwork in the same area of my source directory. Since I used the ftok() function to create the IPC key, I was trying to access a queue that I did not have proper permissions for. I ended up contacting the local system administrator on the morning of the class, only to spend an hour explaining to him what a message queue was, and why I needed him to run the ipcrm command for me. grrrr.*

After successfully retrieving a message from a queue, the message is removed. However, as mentioned earlier, IPC objects remain in the system unless explicitly removed, or the system is rebooted. Therefore, our message queue still exists within the kernel, available for use long after a single message disappears. To complete the life cycle of a message queue, they should be removed with a call to `msgctl()`, using the **IPC_RMID** command:

```
int remove_queue( int qid )
{
```

```
        if( msgctl( qid, IPC_RMID, 0) == -1)
        {
                return(-1);
        }

        return(0);
}
```

This wrapper function returns 0 if the queue was removed without incident, else a value of -1. The removal of the queue is atomic in nature, and any subsequent accesses to the queue for whatever purpose will fail miserably.

msgtool: An interactive message queue manipulator

Few can deny the immediate benefit of having accurate technical information readily available. Such materials provide a tremendous mechanism for learning and exploring new areas. On the same note, having real world examples to accompany any technical information will speed up and reinforce the learning process.

Until now, the only useful examples which have been presented were the wrapper functions for manipulating message queues. While they are extremely useful, they have not been presented in a manner which would warrant further study and experimentation. To remedy this, you will be presented with *msgtool*, an interactive command line utility for manipulating IPC message queues. While it certainly functions as an adequate tool for education reinforcement, it can be applied directly into real world assignments, by providing message queue functionality via standard shell scripts.

Background The `msgtool` program relies on command line arguments to determine its behavior. This is what makes it especially useful when called from a shell script. All of the capabilities are provided, from creating, sending, and retrieving, to changing the permissions and finally removing a queue. Currently, it uses a character array for data, allowing you to send textual messages. Changing it to facilitate additional data types is left as an exercise to the reader.

Command Line Syntax

Sending Messages

```
msgtool s (type) "text"
```

Retrieving Messages

```
msgtool r (type)
```

Changing the Permissions (mode)

```
msgtool m (mode)
```

Deleting a Queue

```
msgtool d
```

Examples

```
msgtool  s   1 test
msgtool  s   5 test
msgtool  s   1 "This is a test"
msgtool  r   1
msgtool  d
msgtool  m   660
```

The Source The following is the source code for the `msgtool` facility. It should compile clean on any recent (decent) kernel revision that supports System V IPC. Be sure to enable System V IPC in your kernel when doing a rebuild!

On a side note, this utility will *create* a message queue if it does not exist, no matter what type of action is requested.

> **NOTE:** *Since this tool uses the ftok() function to generate IPC key values, you may encounter directory conflicts. If you change directories at any point in your script, it probably won't work. Another solution would be to hardcode a more complete path into msgtool, such as "/tmp/msgtool", or possibly even allow the path to be passed on the command line, along with the operational arguments.*

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: msgtool.c
 *****************************************************************************
 A command line tool for tinkering with SysV style Message Queues
 *****************************************************************************/

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

```
#define MAX_SEND_SIZE 80

struct mymsgbuf {
        long mtype;
        char mtext[MAX_SEND_SIZE];
};

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);


int main(int argc, char *argv[])
{
        key_t key;
        int   msgqueue_id;
        struct mymsgbuf qbuf;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 'm');

        /* Open the queue - create if necessary */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
                perror("msgget");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                                       atol(argv[2]), argv[3]);
                          break;
                case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                          break;
                case 'd': remove_queue(msgqueue_id);
                          break;
                case 'm': change_queue_mode(msgqueue_id, argv[2]);
                          break;

                 default: usage();

        }
```

```
        return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Send a message to the queue */
        printf("Sending a message ...\n");
        qbuf->mtype = type;
        strcpy(qbuf->mtext, text);

        if((msgsnd(qid, (struct msgbuf *)qbuf,
                strlen(qbuf->mtext)+1, 0)) ==-1)
        {
                perror("msgsnd");
                exit(1);
        }
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
        /* Read a message from the queue */
        printf("Reading a message ...\n");
        qbuf->mtype = type;
        msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);

        printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}

void remove_queue(int qid)
{
        /* Remove the queue */
        msgctl(qid, IPC_RMID, 0);
}

void change_queue_mode(int qid, char *mode)
{
        struct msqid_ds myqueue_ds;

        /* Get current info */
        msgctl(qid, IPC_STAT, &myqueue_ds);

        /* Convert and load the mode */
        sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);

        /* Update the mode */
        msgctl(qid, IPC_SET, &myqueue_ds);
```

```
}

void usage(void)
{
        fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
        fprintf(stderr, "\nUSAGE: msgtool (s)end <type> <messagetext>\n");
        fprintf(stderr, "                (r)ecv <type>\n");
        fprintf(stderr, "                (d)elete\n");
        fprintf(stderr, "                (m)ode <octal mode>\n");
        exit(1);
}
```

6.4.3 Semaphores

Basic Concepts

Semaphores can best be described as counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it. Semaphores are often dubbed the most difficult to grasp of the three types of System V IPC objects. In order to fully understand semaphores, we'll discuss them briefly before engaging any system calls and operational theory.

The name *semaphore* is actually an old railroad term, referring to the crossroad "arms" that prevent cars from crossing the tracks at intersections. The same can be said about a simple semaphore set. If the semaphore is *on* (the arms are up), then a resource is available (cars may cross the tracks). However, if the semaphore is *off* (the arms are down), then resources are not available (the cars must wait).

While this simple example may stand to introduce the concept, it is important to realize that semaphores are actually implemented as *sets*, rather than as single entities. Of course, a given semaphore set might only have one semaphore, as in our railroad example.

Perhaps another approach to the concept of semaphores, is to think of them as *resource counters*. Let's apply this concept to another real world scenario. Consider a print spooler, capable of handling multiple printers, with each printer handling multiple print requests. A hypothetical print spool manager will utilize semaphore sets to monitor access to each printer.

Assume that in our corporate print room, we have 5 printers online. Our print spool manager allocates a semaphore set with 5 semaphores in it, one for each printer on the system. Since each printer is only physically capable of printing one job at a time, each of our five semaphores in our set will be initialized to a value of 1 (one), meaning that they are all online, and accepting requests.

John sends a print request to the spooler. The print manager looks at the semaphore set, and finds the first semaphore which has a value of one. Before sending John's request to the physical device, the print manager *decrements* the semaphore for the corresponding printer by a value of negative one (-1). Now, that semaphore's value is zero. In the world of System V semaphores, a value of zero represents 100% resource utilization on that semaphore. In our example, no other request can be sent to that printer until it is no longer equal to zero.

When John's print job has completed, the print manager *increments* the value of the semaphore which corresponds to the printer. Its value is now back up to one (1), which means it is available again. Naturally, if all 5 semaphores had a value of zero, that would indicate that they are all busy printing requests, and that no printers are available.

Although this was a simple example, please do not be confused by the initial value of one (1) which was assigned to each semaphore in the set. Semaphores, when thought of as resource counters, may be initialized to *any positive* integer value, and are not limited to either being zero or one. If it were possible for each of our five printers to handle 10 print jobs at a time, we could initialize each of our semaphores to 10, decrementing by one for every new job, and incrementing by one whenever a print job was finished. As you will discover in the next chapter, semaphores have a close working relationship with shared memory segments, acting as a *watchdog* to prevent multiple writes to the same memory segment.

Before delving into the associated system calls, lets take a brief tour through the various internal data structures utilized during semaphore operations.

Internal Data Structures

Let's briefly look at data structures maintained by the kernel for semaphore sets.

Kernel `semid_ds` structure As with message queues, the kernel maintains a special internal data structure for each semaphore set which exists within its addressing space. This structure is of type `semid_ds`, and is defined in `linux/sem.h` as follows:

```
/* One semid data structure for each set of semaphores in the system. */
struct semid_ds {
        struct ipc_perm sem_perm;       /* permissions .. see ipc.h */
        time_t          sem_otime;      /* last semop time */
        time_t          sem_ctime;      /* last change time */
        struct sem      *sem_base;      /* ptr to first semaphore in array */
        struct wait_queue *eventn;
        struct wait_queue *eventz;
        struct sem_undo  *undo;         /* undo requests on this array */
        ushort          sem_nsems;      /* no. of semaphores in array */
};
```

As with message queues, operations on this structure are performed by a special system call, and should not be tinkered with directly. Here are descriptions of the more pertinent fields:

`sem_perm`

This is an instance of the `ipc_perm` structure, which is defined for us in `linux/ipc.h`. This holds the permission information for the semaphore set, including the access permissions, and information about the creator of the set (uid, etc).

`sem_otime`

Time of the last `semop()` operation (more on this in a moment)

`sem_ctime`

Time of the last change to this structure (mode change, etc)

`sem_base`

Pointer to the first semaphore in the array (see next structure)

`sem_undo`

Number of *undo* requests in this array (more on this in a moment)

`sem_nsems`

Number of semaphores in the semaphore set (the array)

Kernel `sem` structure In the `semid_ds` structure, there exists a pointer to the base of the semaphore array itself. Each array member is of the `sem` structure type. It is also defined in `linux/sem.h`:

```
/* One semaphore structure for each semaphore in the system. */
struct sem {
        short   sempid;         /* pid of last operation */
        ushort  semval;         /* current value */
        ushort  semncnt;        /* num procs awaiting increase in semval */
        ushort  semzcnt;        /* num procs awaiting semval = 0 */
};
```

`sem_pid`

The PID (process ID) that performed the last operation

`sem_semval`

The current value of the semaphore

`sem_semncnt`

Number of processes waiting for resources to become available

`sem_semzcnt`

Number of processes waiting for 100% resource utilization

SYSTEM CALL: semget()

In order to create a new semaphore set, or access an existing set, the `semget()` system call is used.

```
SYSTEM CALL: semget();

PROTOTYPE: int semget ( key_t key, int nsems, int semflg );
  RETURNS: semaphore set IPC identifier on success
           -1 on error: errno = EACCESS (permission denied)
                                EEXIST (set exists, cannot create (IPC_EXCL))
                                EIDRM (set is marked for deletion)
                                ENOENT (set does not exist, no IPC_CREAT was used)
                                ENOMEM (Not enough memory to create new set)
                                ENOSPC (Maximum set limit exceeded)
NOTES:
```

The first argument to `semget()` is the key value (in our case returned by a call to `ftok()`). This key value is then compared to existing key values that exist within the kernel for other semaphore sets. At that point, the open or access operation is dependent upon the contents of the `semflg` argument.

IPC_CREAT

Create the semaphore set if it doesn't already exist in the kernel.

IPC_EXCL

When used with IPC_CREAT, fail if semaphore set already exists.

If `IPC_CREAT` is used alone, `semget()` either returns the semaphore set identifier for a newly created set, or returns the identifier for a set which exists with the same key value. If `IPC_EXCL` is used along with `IPC_CREAT`, then either a new set is created, or if the set exists, the call fails with -1. `IPC_EXCL` is useless by itself, but when combined with `IPC_CREAT`, it can be used as a facility to guarantee that no existing semaphore set is opened for access.

As with the other forms of System V IPC, an optional octal mode may be OR'd into the mask to form the permissions on the semaphore set.

The `nsems` argument specifies the number of semaphores that should be created in a new set. This represents the number of printers in our fictional print room described earlier. The maximum number of semaphores in a set is defined in "linux/sem.h" as:

```
#define SEMMSL  32      /* <=512 max num of semaphores per id */
```

Note that the `nsems` argument is ignored if you are explicitly opening an existing set.

Let's create a wrapper function for opening or creating semaphore sets:

```
int open_semaphore_set( key_t keyval, int numsems )
{
        int     sid;

        if ( ! numsems )
                return(-1);

        if((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(sid);
}
```

Note the use of the explicit permissions of `0660`. This small function either returns a semaphore set identifier (`int`), or -1 on error. The key value must be passed to it, as well as the number of semaphores to allocate space for if creating a new set. In the example presented at the end of this section, notice the use of the IPC_EXCL flag to determine whether or not the semaphore set exists or not.

SYSTEM CALL: semop()

```
  SYSTEM CALL: semop();
  PROTOTYPE: int semop ( int semid, struct sembuf *sops, unsigned nsops);
    RETURNS: 0 on success (all operations performed)
             -1 on error: errno = E2BIG (nsops greater than max number of ops allowed atomically)
                                  EACCESS (permission denied)
                                  EAGAIN (IPC_NOWAIT asserted, operation could not go through)
                                  EFAULT (invalid address pointed to by sops argument)
                                  EIDRM (semaphore set was removed)
                                  EINTR (Signal received while sleeping)
                                  EINVAL (set doesn't exist, or semid is invalid)
                                  ENOMEM (SEM_UNDO asserted, not enough memory to create the
                                          undo structure necessary)
                                  ERANGE (semaphore value out of range)
  NOTES:
```

The first argument to `semget()` is the key value (in our case returned by a call to `semget`). The second argument (`sops`) is a pointer to an array of *operations* to be performed on the semaphore set, while the third argument (`nsops`) is the number of operations in that array.

The `sops` argument points to an array of type `sembuf`. This structure is declared in `linux/sem.h` as follows:

```
        /* semop system call takes an array of these */
        struct sembuf {
```

```
                ushort  sem_num;        /* semaphore index in array */
                short   sem_op;         /* semaphore operation */
                short   sem_flg;        /* operation flags */
        };
```

`sem_num`

The number of the semaphore you wish to deal with

`sem_op`

The operation to perform (positive, negative, or zero)

`sem_flg`

Operational flags

If `sem_op` is negative, then its value is subtracted from the semaphore. This correlates with obtaining resources that the semaphore controls or monitors access of. If **IPC_NOWAIT** is not specified, then the calling process sleeps until the requested amount of resources are available in the semaphore (another process has released some).

If `sem_op` is positive, then it's value is added to the semaphore. This correlates with returning resources back to the application's semaphore set. Resources should always be returned to a semaphore set when they are no longer needed!

Finally, if `sem_op` is zero (0), then the calling process will sleep() until the semaphore's value is 0. This correlates to waiting for a semaphore to reach 100% utilization. A good example of this would be a daemon running with superuser permissions that could dynamically adjust the size of the semaphore set if it reaches full utilization.

In order to explain the `semop` call, let's revisit our print room scenario. Let's assume only one printer, capable of only one job at a time. We create a semaphore set with only one semaphore in it (only one printer), and initialize that one semaphore to a value of one (only one job at a time).

Each time we desire to send a job to this printer, we need to first make sure that the resource is available. We do this by attempting to obtain one *unit* from the semaphore. Let's load up a sembuf array to perform the operation:

```
        struct sembuf sem_lock = { 0, -1, IPC_NOWAIT };
```

Translation of the above initialized structure dictates that a value of "-1" will be added to semaphore number 0 in the semaphore set. In other words, one unit of resources will be obtained from the only semaphore in our set (0th member). **IPC_NOWAIT** is specified, so the call will either go through immediately, or fail if another print job is currently printing. Here is an example of using this initialized `sembuf` structure with the `semop` system call:

```
        if((semop(sid, &sem_lock, 1) == -1)
                perror("semop");
```

The third argument (`nsops`) says that we are only performing one (1) operation (there is only one `sembuf` structure in our array of operations). The `sid` argument is the IPC identifier for our semaphore set.

When our print job has completed, we must *return* the resources back to the semaphore set, so that others may use the printer.

```
struct sembuf sem_unlock = { 0, 1, IPC_NOWAIT };
```

Translation of the above initialized structure dictates that a value of "1" will be added to semaphore number 0 in the semaphore set. In other words, one unit of resources will be returned to the set.

SYSTEM CALL: semctl()

```
SYSTEM CALL: semctl();
PROTOTYPE: int semctl ( int semid, int semnum, int cmd, union semun arg );
  RETURNS: positive integer on success
           -1 on error: errno = EACCESS (permission denied)
                                EFAULT (invalid address pointed to by arg argument)
                                EIDRM (semaphore set was removed)
                                EINVAL (set doesn't exist, or semid is invalid)
                                EPERM (EUID has no privileges for cmd in arg)
                                ERANGE (semaphore value out of range)
NOTES: Performs control operations on a semaphore set
```

The *semctl* system call is used to perform control operations on a semaphore set. This call is analogous to the *msgctl* system call which is used for operations on message queues. If you compare the argument lists of the two system calls, you will notice that the list for *semctl* varies slightly from that of *msgctl*. Recall that semaphores are actually implemented as sets, rather than as single entities. With semaphore operations, not only does the IPC key need to be passed, but the target semaphore within the set as well.

Both system calls utilize a *cmd* argument, for specification of the command to be performed on the IPC object. The remaining difference lies in the final argument to both calls. In *msgctl*, the final argument represents a copy of the internal data structure used by the kernel. Recall that we used this structure to retrieve internal information about a message queue, as well as to set or change permissions and ownership of the queue. With semaphores, additional operational commands are supported, thus requiring a more complex data type as the final argument. The use of a *union* confuses many neophyte semaphore programmers to a substantial degree. We will dissect this structure carefully, in an effort to prevent any confusion.

The first argument to `semctl()` is the key value (in our case returned by a call to `semget`). The second argument (`semun`) is the semaphore number that an operation is targeted towards. In essence, this can be thought of as an *index* into the semaphore set, with the first semaphore (or only one) in the set being represented by a value of zero (0).

The `cmd` argument represents the command to be performed against the set. As you can see, the familiar IPC_STAT/IPC_SET commands are present, along with a wealth of additional commands specific to semaphore sets:

IPC_STAT

Retrieves the semid_ds structure for a set, and stores it in the address of the buf argument in the semun union.

IPC_SET

Sets the value of the ipc_perm member of the semid_ds structure for a set. Takes the values from the buf argument of the semun union.

IPC_RMID

Removes the set from the kernel.

GETALL

Used to obtain the values of all semaphores in a set. The integer values are stored in an array of unsigned short integers pointed to by the *array* member of the union.

GETNCNT

Returns the number of processes currently waiting for resources.

GETPID

Returns the PID of the process which performed the last *semop* call.

GETVAL

Returns the value of a single semaphore within the set.

GETZCNT

Returns the number of processes currently waiting for 100% resource utilization.

SETALL

Sets all semaphore values with a set to the matching values contained in the *array* member of the union.

SETVAL

Sets the value of an individual semaphore within the set to the *val* member of the union.

The `arg` argument represents an instance of type `semun`. This particular union is declared in `linux/sem.h` as follows:

```
        /* arg for semctl system calls. */
        union semun {
                int val;                /* value for SETVAL */
                struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
                ushort *array;          /* array for GETALL & SETALL */
                struct seminfo *__buf;  /* buffer for IPC_INFO */
                void *__pad;
        };
```

`val`

Used when the SETVAL command is performed. Specifies the value to set the semaphore to.

`buf`

Used in the IPC_STAT/IPC_SET commands. Represents a copy of the internal semaphore data structure used in the kernel.

`array`

A pointer used in the GETALL/SETALL commands. Should point to an array of integer values to be used in setting or retrieving all semaphore values in a set.

The remaining arguments *__buf* and *__pad* are used internally in the semaphore code within the kernel, and are of little or no use to the application developer. As a matter of fact, these two arguments are specific to the Linux operating system, and are not found in other UNIX implementations.

Since this particular system call is arguably the most difficult to grasp of all the System V IPC calls, we'll examine multiple examples of it in action.

The following snippet returns the value of the passed semaphore. The final argument (the union) is ignored when the GETVAL command is used:

```
int get_sem_val( int sid, int semnum )
{
        return( semctl(sid, semnum, GETVAL, 0));
}
```

To revisit the printer example, let's say the status of all five printers was required:

```
        #define MAX_PRINTERS 5

        printer_usage()
        {
                int x;

                for(x=0; x<MAX_PRINTERS; x++)
                        printf("Printer %d: %d\n\r", x, get_sem_val( sid, x ));
        }
```

Consider the following function, which could be used to initialize a new semaphore value:

```
void init_semaphore( int sid, int semnum, int initval)
{
        union semun semopts;

        semopts.val = initval;
        semctl( sid, semnum, SETVAL, semopts);
}
```

Note that the final argument of *semctl* is a copy of the union, rather than a pointer to it. While we're on the subject of the union as an argument, allow me to demonstrate a rather common mistake when using this system call.

Recall from the msgtool project that the IPC_STAT and IPC_SET commands were used to alter permissions on the queue. While these commands are supported in the semaphore implementation, their usage is a bit different, as the internal data structure is retrieved and copied from a member of the union, rather than as a single entity. Can you locate the bug in this code?

```
/* Required permissions should be passed in as text (ex: "660") */

void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
                perror("semctl");
                exit(1);
        }

        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");
}
```

The code is attempting to make a local copy of the internal data structure for the set, modify the permissions, and IPC_SET them back to the kernel. However, the first call to *semctl* promptly returns EFAULT, or bad address for the last argument (the union!). In addition, if we hadn't checked for errors from that call, we would have gotten a memory fault. Why?

Recall that the IPC_SET/IPC_STAT commands use the *buf* member of the union, which is a *pointer* to a type *semid_ds*. Pointers are pointers are pointers are pointers! The *buf* member must point to some valid storage location in order for our function to work properly. Consider this revamped version:

```
void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */

        /* Point to our local copy first! */
        semopts.buf = &mysemds;

        /* Let's try this again! */
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
                perror("semctl");
                exit(1);
        }

        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");
}
```

semtool: An interactive semaphore manipulator

Background The `semtool` program relies on command line arguments to determine its behavior. This is what makes it especially useful when called from a shell script. All of the capabilities are provided, from creating and manipulating, to changing the permissions and finally removing a semaphore set. It can be used to control shared resources via standard shell scripts.

Command Line Syntax

Creating a Semaphore Set

```
semtool c (number of semaphores in set)
```

Locking a Semaphore

```
semtool l (semaphore number to lock)
```

Unlocking a Semaphore

```
semtool u (semaphore number to unlock)
```

Changing the Permissions (mode)

```
semtool m (mode)
```

Deleting a Semaphore Set

```
semtool d
```

Examples

```
semtool  c 5
semtool  l
semtool  u
semtool  m 660
semtool  d
```

The Source

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: semtool.c
 *****************************************************************************
 A command line tool for tinkering with SysV style Semaphore Sets

 *****************************************************************************/

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

```
#define SEM_RESOURCE_MAX        1       /* Initial value of all semaphores */

void opensem(int *sid, key_t key);
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
unsigned short get_member_count(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
void changemode(int sid, char *mode);
void usage(void);

int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 's');

        switch(tolower(argv[1][0]))
        {
                case 'c': if(argc != 3)
                                usage();
                          createsem(&semset_id, key,  atoi(argv[2]));
                          break;
                case 'l': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
                          locksem(semset_id, atoi(argv[2]));
                          break;
                case 'u': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
                          unlocksem(semset_id, atoi(argv[2]));
                          break;
                case 'd': opensem(&semset_id, key);
                          removesem(semset_id);
                          break;
                case 'm': opensem(&semset_id, key);
                          changemode(semset_id, argv[2]);
                          break;
```

```
                 default: usage();

        }

        return(0);
}

void opensem(int *sid, key_t key)
{
        /* Open the semaphore set - do not create! */

        if((*sid = semget(key, 0, 0666)) == -1)
        {
                printf("Semaphore set does not exist!\n");
                exit(1);
        }

}

void createsem(int *sid, key_t key, int members)
{
        int cntr;
        union semun semopts;

        if(members > SEMMSL) {
                printf("Sorry, max number of semaphores in a set is %d\n",
                        SEMMSL);
                exit(1);
        }

        printf("Attempting to create new semaphore set with %d members\n",
                                members);

        if((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666))
                        == -1)
        {
                fprintf(stderr, "Semaphore set already exists!\n");
                exit(1);
        }

        semopts.val = SEM_RESOURCE_MAX;

        /* Initialize all members (could be done with SETALL) */
        for(cntr=0; cntr<members; cntr++)
                semctl(*sid, cntr, SETVAL, semopts);
}
```

```
void locksem(int sid, int member)
{
        struct sembuf sem_lock={ 0, -1, IPC_NOWAIT};

        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }

        /* Attempt to lock the semaphore set */
        if(!getval(sid, member))
        {
                fprintf(stderr, "Semaphore resources exhausted (no lock)!\n");
                exit(1);
        }

        sem_lock.sem_num = member;

        if((semop(sid, &sem_lock, 1)) == -1)
        {
                fprintf(stderr, "Lock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources decremented by one (locked)\n");

        dispval(sid, member);
}

void unlocksem(int sid, int member)
{
        struct sembuf sem_unlock={ member, 1, IPC_NOWAIT};
        int semval;

        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }

        /* Is the semaphore set locked? */
        semval = getval(sid, member);
        if(semval == SEM_RESOURCE_MAX) {
                fprintf(stderr, "Semaphore not locked!\n");
                exit(1);
        }
```

```
        sem_unlock.sem_num = member;

        /* Attempt to lock the semaphore set */
        if((semop(sid, &sem_unlock, 1)) == -1)
        {
                fprintf(stderr, "Unlock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources incremented by one (unlocked)\n");

        dispval(sid, member);
}

void removesem(int sid)
{
        semctl(sid, 0, IPC_RMID, 0);
        printf("Semaphore removed\n");
}

unsigned short get_member_count(int sid)
{
        union semun semopts;
        struct semid_ds mysemds;

        semopts.buf = &mysemds;

        /* Return number of members in the semaphore set */
        return(semopts.buf->sem_nsems);
}

int getval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        return(semval);
}

void changemode(int sid, char *mode)
{
        int rc;
        union semun semopts;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
```

```
        semopts.buf = &mysemds;

        rc = semctl(sid, 0, IPC_STAT, semopts);

        if (rc == -1) {
                perror("semctl");
                exit(1);
        }

        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%ho", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");

}

void dispval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        printf("semval for member %d is %d\n", member, semval);
}

void usage(void)
{
        fprintf(stderr, "semtool - A utility for tinkering with semaphores\n");
        fprintf(stderr, "\nUSAGE:  semtool4 (c)reate <semcount>\n");
        fprintf(stderr, "              (l)ock <sem #>\n");
        fprintf(stderr, "              (u)nlock <sem #>\n");
        fprintf(stderr, "              (d)elete\n");
        fprintf(stderr, "              (m)ode <mode>\n");
        exit(1);
}
```

---

semstat: A semtool companion program

As an added bonus, the source code to a companion program called `semstat` is provided next. The `semstat` program displays the values of each of the semaphores in the set created by `semtool`.

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: semstat.c
 *****************************************************************************
 A companion command line tool for the semtool package.  semstat displays
 the current value of all semaphores in the set created by semtool.
 *****************************************************************************/

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

int get_sem_count(int sid);
void show_sem_usage(int sid);
int get_sem_count(int sid);
void dispval(int sid);

int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;

        /* Create unique key via call to ftok() */
        key = ftok(".", 's');

        /* Open the semaphore set - do not create! */
        if((semset_id = semget(key, 1, 0666)) == -1)
        {
                printf("Semaphore set does not exist\n");
                exit(1);
        }

        show_sem_usage(semset_id);
        return(0);
}

void show_sem_usage(int sid)
{
        int cntr=0, maxsems, semval;

        maxsems = get_sem_count(sid);

        while(cntr < maxsems) {
```

```
                semval = semctl(sid, cntr, GETVAL, 0);
                printf("Semaphore #%d:  --> %d\n", cntr, semval);
                cntr++;
        }
}

int get_sem_count(int sid)
{
        int rc;
        struct semid_ds mysemds;
        union semun semopts;

        /* Get current values for internal data structure */
        semopts.buf = &mysemds;

        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1) {
                perror("semctl");
                exit(1);
        }

        /* return number of semaphores in set */
        return(semopts.buf->sem_nsems);
}

void dispval(int sid)
{
        int semval;

        semval = semctl(sid, 0, GETVAL, 0);
        printf("semval is %d\n", semval);
}
```

6.4.4 Shared Memory

Basic Concepts

Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process. This is by far the fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue, etc). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process, and subsequently written to and read from by any number of processes.

Internal and User Data Structures

Let's briefly look at data structures maintained by the kernel for shared memory segments.

Kernel shmid_ds structure As with message queues and semaphore sets, the kernel maintains a special internal data structure for each shared memory segment which exists within its addressing space. This structure is of type `shmid_ds`, and is defined in `linux/shm.h` as follows:

```
        /* One shmid data structure for each shared memory segment in the system. */
        struct shmid_ds {
                struct ipc_perm shm_perm;        /* operation perms */
                int     shm_segsz;               /* size of segment (bytes) */
                time_t  shm_atime;               /* last attach time */
                time_t  shm_dtime;               /* last detach time */
                time_t  shm_ctime;               /* last change time */
                unsigned short  shm_cpid;        /* pid of creator */
                unsigned short  shm_lpid;        /* pid of last operator */
                short   shm_nattch;              /* no. of current attaches */

                                                 /* the following are private */

                unsigned short   shm_npages;     /* size of segment (pages) */
                unsigned long   *shm_pages;      /* array of ptrs to frames -> SHMMAX */
                struct vm_area_struct *attaches; /* descriptors for attaches */
        };
```

Operations on this structure are performed by a special system call, and should not be tinkered with directly. Here are descriptions of the more pertinent fields:

`shm_perm`

This is an instance of the `ipc_perm` structure, which is defined for us in `linux/ipc.h`. This holds the permission information for the segment, including the access permissions, and information about the creator of the segment (uid, etc).

`shm_segsz`

Size of the segment (measured in bytes).

`shm_atime`

Time the last process attached the segment.

`shm_dtime`

Time the last process detached the segment.

`shm_ctime`

Time of the last change to this structure (mode change, etc).

`shm_cpid`

The PID of the creating process.

`shm_lpid`

The PID of the last process to operate on the segment.

`shm_nattch`

Number of processes currently attached to the segment.

SYSTEM CALL: shmget()

In order to create a new message queue, or access an existing queue, the `shmget()` system call is used.

```
SYSTEM CALL: shmget();

PROTOTYPE: int shmget ( key_t key, int size, int shmflg );
  RETURNS: shared memory segment identifier on success
           -1 on error: errno = EINVAL (Invalid segment size specified)
                                EEXIST (Segment exists, cannot create)
                                EIDRM (Segment is marked for deletion, or was removed)
                                ENOENT (Segment does not exist)
                                EACCES (Permission denied)
                                ENOMEM (Not enough memory to create segment)
NOTES:
```

This particular call should almost seem like old news at this point. It is strikingly similar to the corresponding `get` calls for message queues and semaphore sets.

The first argument to `shmget()` is the key value (in our case returned by a call to `ftok()`). This key value is then compared to existing key values that exist within the kernel for other shared memory segments. At that point, the open or access operation is dependent upon the contents of the `shmflg` argument.

IPC_CREAT

Create the segment if it doesn't already exist in the kernel.

IPC_EXCL

When used with IPC_CREAT, fail if segment already exists.

If `IPC_CREAT` is used alone, `shmget()` either returns the segment identifier for a newly created segment, or returns the identifier for a segment which exists with the same key value. If `IPC_EXCL` is used along with `IPC_CREAT`, then either a new segment is created, or if the segment exists, the call fails with -1. `IPC_EXCL` is useless by itself, but when combined with `IPC_CREAT`, it can be used as a facility to guarantee that no existing segment is opened for access.

Once again, an optional octal mode may be OR'd into the mask.

Let's create a wrapper function for locating or creating a shared memory segment :

```
int open_segment( key_t keyval, int segsize )
{
        int     shmid;

        if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(shmid);
}
```

Note the use of the explicit permissions of `0660`. This small function either returns a shared memory segment identifier (`int`), or -1 on error. The key value and requested segment size (in bytes) are passed as arguments.

Once a process has a valid IPC identifier for a given segment, the next step is for the process to `attach` or map the segment into its own addressing space.

SYSTEM CALL: shmat()

```
  SYSTEM CALL: shmat();

  PROTOTYPE: int shmat ( int shmid, char *shmaddr, int shmflg);
    RETURNS: address at which segment was attached to the process, or
             -1 on error: errno = EINVAL (Invalid IPC ID value or attach address passed)
                                  ENOMEM (Not enough memory to attach segment)
                                  EACCES (Permission denied)
  NOTES:
```

If the addr argument is zero (0), the kernel tries to find an unmapped region. This is the recommended method. An address can be specified, but is typically only used to facilitate proprietary hardware or to resolve conflicts with other apps. The SHM_RND flag can be OR'd into the flag argument to force a passed address to be page aligned (rounds down to the nearest page size).

In addition, if the SHM_RDONLY flag is OR'd in with the flag argument, then the shared memory segment will be mapped in, but marked as readonly.

This call is perhaps the simplest to use. Consider this wrapper function, which is passed a valid IPC identifier for a segment, and returns the address that the segment was attached to:

```
char *attach_segment( int shmid )
{
        return(shmat(shmid, 0, 0));
}
```

Once a segment has been properly attached, and a process has a pointer to the start of that segment, reading and writing to the segment become as easy as simply referencing or dereferencing the pointer! Be careful not to lose the value of the original pointer! If this happens, you will have no way of accessing the base (start) of the segment.

SYSTEM CALL: shmctl()

```
  SYSTEM CALL: shmctl();
  PROTOTYPE: int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
    RETURNS: 0 on success
             -1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
                                  EFAULT (Address pointed to by buf is invalid with IPC_SET and
                                          IPC_STAT commands)
                                  EIDRM  (Segment was removed during retrieval)
                                  EINVAL (shmqid invalid)
                                  EPERM  (IPC_SET or IPC_RMID command was issued, but
                                          calling process does not have write (alter)
                                          access to the segment)
      NOTES:
```

This particular call is modeled directly after the *msgctl* call for message queues. In light of this fact, it won't be discussed in too much detail. Valid command values are:

IPC_STAT

Retrieves the shmid_ds structure for a segment, and stores it in the address of the buf argument

IPC_SET

Sets the value of the ipc_perm member of the shmid_ds structure for a segment. Takes the values from the buf argument.

IPC_RMID

Marks a segment for removal.

The IPC_RMID command doesn't actually remove a segment from the kernel. Rather, it `marks` the segment for removal. The actual removal itself occurs when the last process currently attached to the segment has properly detached it. Of course, if no processes are currently attached to the segment, the removal seems immediate.

To properly detach a shared memory segment, a process calls the *shmdt* system call.

SYSTEM CALL: shmdt()

```
  SYSTEM CALL: shmdt();

  PROTOTYPE: int shmdt ( char *shmaddr );
    RETURNS: -1 on error: errno = EINVAL (Invalid attach address passed)
```

After a shared memory segment is no longer needed by a process, it should be detached by calling this system call. As mentioned earlier, this is not the same as removing the segment from the kernel! After a detach is successful, the shm_nattch member of the associates shmid_ds structure is decremented by one. When this value reaches zero (0), the kernel will physically remove the segment.

shmtool: An interactive shared memory manipulator

Background Our final example of System V IPC objects will be `shmtool`, which is a command line tool for creating, reading, writing, and deleting shared memory segments. Once again, like the previous examples, the segment is created during any operation, if it did not previously exist.

Command Line Syntax

Writing strings to the segment

```
shmtool w "text"
```

Retrieving strings from the segment

```
shmtool r
```

Changing the Permissions (mode)

```
shmtool m (mode)
```

Deleting the segment

```
shmtool d
```

Examples

```
shmtool  w   test
shmtool  w   "This is a test"
shmtool  r
shmtool  d
shmtool  m   660
```

The Source

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

#define SEGSIZE 100
```

```
main(int argc, char *argv[])
{
        key_t key;
        int   shmid, cntr;
        char  *segptr;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 'S');

        /* Open the shared memory segment - create if necessary */
        if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
        {
                printf("Shared memory segment exists - opening as client\n");

                /* Segment probably already exists - try as a client */
                if((shmid = shmget(key, SEGSIZE, 0)) == -1)
                {
                        perror("shmget");
                        exit(1);
                }
        }
        else
        {
                printf("Creating new shared memory segment\n");
        }

        /* Attach (map) the shared memory segment into the current process */
        if((segptr = shmat(shmid, 0, 0)) == -1)
        {
                perror("shmat");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 'w': writeshm(shmid, segptr, argv[2]);
                          break;
                case 'r': readshm(shmid, segptr);
                          break;
                case 'd': removeshm(shmid);
                          break;
                case 'm': changemode(shmid, argv[2]);
                          break;
```

```
                default: usage();

        }
}

writeshm(int shmid, char *segptr, char *text)
{
        strcpy(segptr, text);
        printf("Done...\n");
}

readshm(int shmid, char *segptr)
{
        printf("segptr: %s\n", segptr);
}

removeshm(int shmid)
{
        shmctl(shmid, IPC_RMID, 0);
        printf("Shared memory segment marked for deletion\n");
}

changemode(int shmid, char *mode)
{
        struct shmid_ds myshmds;

        /* Get current values for internal data structure */
        shmctl(shmid, IPC_STAT, &myshmds);

        /* Display old permissions */
        printf("Old permissions were: %o\n", myshmds.shm_perm.mode);

        /* Convert and load the mode */
        sscanf(mode, "%o", &myshmds.shm_perm.mode);

        /* Update the mode */
        shmctl(shmid, IPC_SET, &myshmds);

        printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}

usage()
{
        fprintf(stderr, "shmtool - A utility for tinkering with shared memory\n");
        fprintf(stderr, "\nUSAGE:  shmtool (w)rite <text>\n");
        fprintf(stderr, "                (r)ead\n");
        fprintf(stderr, "                (d)elete\n");
```

```
        fprintf(stderr, "                (m)ode change <octal mode>\n");
        exit(1);
}
```

Chapter 7

Sound Programming

A PC has at least one sound device: the internal speaker. But, you can also buy a sound card to plug into your PC to provide a more sophisticated sound device. Look at the Linux Sound User's Guide or the Sound-HOWTO for supported sound cards.

7.1 Programming the internal speaker

Believe it or not, your PC speaker is part of the Linux console and thus a character device. Therefore, ioctl() requests exist to manipulate it. For the internal speaker the following 2 requests exist:

1. KDMKTONE

   Generates a beep for a specified time using the kernel timer.

   Example: `ioctl (fd, KDMKTONE,(long) argument)`.

2. KIOCSOUND

   Generates an endless beep or stops a currently sounding beep.

   Example: `ioctl(fd,KIOCSOUND,(int) tone)`.

The `argument` consists of the `tone` value in the low word and the duration in the high word. The `tone` value is not the frequency. The PC mainboard timer 8254 is clocked at 1.19 MHz and so it's 1190000/frequency. The duration is measured in timer ticks. Both ioctl calls return immediately so you can this way produce beeps without blocking the program.
KDMKTONE should be used for warning signals because you don't have to worry about stopping the tone.
KIOCSOUND can be used to play melodies as demonstrated in the example program splay *(please send more .sng files to me).* To stop the beep you have to use the `tone` value 0.

7.2 Programming a sound card

For you as a programmer, it is important to know if the current Linux system has a sound card plugged in. One way to check is to examine /dev/sndstat. If opening /dev/sndstat fails and errno=ENODEV then no sound driver is activated which means you will get no help from the kernel sound driver. The same result might be achieved by trying to open /dev/dsp as long as it is not a link to the pcsnd driver in which case open() will not fail.

If you want to mess with a sound card at the hardware level you know that some combination of outb() and inb() calls will detect the sound card you are looking for.

By using the sound driver for your programs, chances are that they will work on other i386 systems as well, since some clever people decided to use the same driver for Linux, isc, FreeBSD and most other i386 based systems. It will aid in porting programs if Linux on other architectures offers the same sound device interface. A sound card is not part of the Linux console, but is a special device. A sound card mostly offers three main features:

- Digital sample input/output
- Frequency modulation output
- A midi interface

Each of these features have their own device driver interface. For digital samples it is /dev/dsp, for the frequency modulation it is /dev/sequencer and for the midi interface it is /dev/midi. The sound settings (like volume, balance or bass) can be controlled via the /dev/mixer interface. For compatibility reasons a /dev/audio device exists which can read SUN μ-law sound data, but it maps to the digital sample device.

You are right if you guessed that you use ioctl() to manipulate these devices. The ioctl() requests are defined in $< linux/soundcard.h >$ and begin with SNDCTL_.

Since I don't own a soundcard someone else has to continue here

Chapter 8

Character Cell Graphics

This chapter deals with screen input and output that is not pixel based, but character based. When we say character, we mean a composition of pixels that can be changed depending on a charset. Your graphic card already offers one or more charsets and operates by default in text (charset) mode because text can be processed much faster than pixel graphic. There is more to do with terminals than to use them as simple (dumb) and boring text displays. I will describe how to use the special features that your linux terminal, especially the linux console, offers.

- **printf, sprintf, fprintf, scanf, sscanf, fscanf**
  With these functions from *libc* you can output formatted strings to *stdout* (standard output), *stderr* (standard error) or other streams defined as `FILE *stream` (files, for example). **Scanf(...)** provides a similar way to read formatted input from *stdin*.

- **termcap**
  The TERMinal CAPabilitie database is a set of terminal description entries in the ASCII file /etc/termcap. Here you can find information about how to display special characters, how to perform operations (delete, insert characters or lines etc) and how to initialize a terminal. The database is used, for example, by the editor vi. There are view library functions to read and use the terminal capabilities (*see termcap(3x)*). With this database, programs can work with a variety of terminals with the same code. Using the termcap database and library functions provides only low level access to the terminal. Changing attributes or colors, parameterized output and optimization must be done by the programmer himself.

- **terminfo database**
  The TERMinal INFOrmation database is based on the termcap database and also describes terminal capabilities, but on a higher level than termcap. Using terminfo, the program can easily change screen attributes, use special keys such as function keys and more. The database can be found in */usr/lib/terminfo/[A-z,0-9]**. Every file describes one terminal.

- **curses**
  Terminfo is a good base to use for terminal handling in a program. The (BSD-)CURSES library gives you high level access to the terminal and is based on the terminfo database. Curses allows you to open and manipulate windows on the screen, provides a complete set of input and output functions, and can alter video attributes in a terminal independent manner on more than 150 terminals. The curses library can be found in */usr/lib/libcurses.a*. This is the BSD version of curses.

- **ncurses**
  Ncurses is the next improvement. In version 1.8.6it should be compatible with AT&T curses as defined in SYSVR4 and has some extensions such as color manipulation, special optimization for output, terminal specific optimizations, and more. It has been tested on a lot of systems such as Sun-OS, HP and Linux. I recommend using ncurses instead of the others. On SYSV Unix systems (such as Sun's Solaris) there should exist a curses library with the same functionality as ncurses (actually the solaris curses has some more functions and mouse support).

In the following sections I will describe how to use the different packages to access a terminal. With Linux we have the GNU-version of termcap and we can use ncurses instead of curses.

8.1 I/O Function in libc

8.1.1 Formatted Output

The **printf(...)** functions in *libc* provide formatted output and allow transformations of the arguments.

- `int fprintf(FILE *stream, const char *format, ...),`
  will transform the output (arguments to fill in `...`) and write it to `stream`. The format defined in `format` will be written, too. The function will return the number of written characters or a negative value on error.

  `format` contains two kinds of objects

  1. normal characters for the output and
  2. information how to transform or format the arguments.

  Format information must begin with % followed by values for the format followed by a character for the translation (to print % by itself use %%). Possible values for the format are:

  – Flags
    * `-`
      The formatted argument will be printed on the left margin (default is the right margin in the argument field).

    * `+`
      Every number will be printed with a sign, e.g. `+12` or `-2.32`.
  - Blank
    When the first character is not a sign, a blank will be inserted.
  - `0`
    For numeric transformation the field width will be filled up with `0`'s on the left side.
  - `#`
    Alternate output depending on the transformation for the argument
    * For *o* the first number is a *0*.
    * For *x* or *X* *0x* or *0X* will be printed in front of the argument.
    * For *e*, *E*, *f* or *F* the output has a decimal point.
    * For *g* or *G* zeroes on the end of the argument are printed.
  - A number for the minimal field width.
    The transformed argument is printed in a field which is at least as big as the argument itself. With a number you can make the field width bigger. If the formatted argument is smaller, then the field width will be filled with zeroes or blanks.
  - A point to separate the field width and the precision.
  - A number for the precision.

  Possible values for the transformation are in table 8.1 on page 82.

- `int printf(const char *format, ...)`
  Same as **fprintf(stdout, ...)**.

- `int sprintf(char *s, const char *format, ...)`
  Same as **printf(...)**, except that the output will be written to the the character pointer `s` (with a following *\0*).

  (**Note:** You must allocate enough memory for `s`.)

- `vprintf(const char *format, va_list arg)`
  `vfprintf(FILE *stream, const char *format, va_list arg)`
  `vsprintf(char *s, const char *format, va_list arg)`
  The same as the functions above, only the argument list is set to `arg`.

8.1.2 Formatted Input

Just as **printf(...)** is used for formatted output you can use **scanf(...)** for formatted input.

- `int fscanf(FILE *stream, const char *format, ...)`
  fscanf(...) reads from `stream` and will transform the input with the rules defined in `format`. The results will be placed in the arguments given by ...(**Note:** the arguments **must** be pointer.). The read ends, when no

Table 8.1: Libc - printf transformations

Character	**Formatted to**
d,i	*int* signed, decimal
o	*int* unsigned, octal, without leading 0
x,X	*int* unsigned, hexadecimal without leading 0x
u	*int* unsigned, decimal
c	*int* (unsigned) single character
s	*char ** up to \0
f	*double* as [-]mmm.ddd
e,E	*double* as [-]m.dddddde±xx
g,G	*double* using %e or %f as needed
p	*void **
n	*int **
%	*%*

more transformation rules are in `format`. **fscanf(...)** will return EOF when the first transformation reached the file end or some error occured. Otherwise it will return the number of transformed arguments.

`format` can include rules on how to format the input arguments (see table 8.2 on page 83). It can also include:

– Spaces or tabs, which are ignored.

– any normal character (except %). The characters must be in the input on the corresponding position.

– transformation rules, which assembled with a *%*, the optional character *** (this will permit **fscanf(...)** to assign to an argument), an optional number, an optional character *h, l or L* (this is for the length of the target) and the transformation character.

- `int scanf(const char *format, ...)`
  The same as **fscanf(stdin,...)**.

- `int sscanf(char *str, const char *format, ...)`
  As **scanf(...)**, but the input comes from `str`.

8.2 The Termcap Library

8.2.1 Introduction

The termcap library is an API to the termcap database which can be found in */etc/termcap/*. The library functions allow the following actions:

- Get a description of the current terminal: **tgetent(...)**.

Table 8.2: Libc - scanf transformations

Character	**Input - Argumenttyp**
d	decimal integer - *int **
i	integer - *int ** (input can be octal or hex)
o	octal integer - *int ** (with or without leading 0)
u	decimal, unsigned - *unsigned int **
x	hex integer - *int ** (with or without leading 0x)
c	one or more characters - *char ** (without \0)
s	characters(without space, tab ...) - *char ** (with \0)
e,f,gf	float - *float ** (e.g [-]m.dddddde±xx)
p	pointer - *void **
n	number of transformed arguments - *int **
[...]	nonempty characters in input - *char **
[^...]	exclude those nonempty characters - *char **
%	%

h can be before d,i,n,o,u and x when the pointer is *short*
l can be before d,i,n,o,u and x when the pointer is *long*
l can be before e,f and g when the pointer is *double*
L can be before e,f and g when the pointer is *long double*

- Search the description for information: **tgetnum(...)**, **tgetflag(...)**, **tgetstr(...)**.
- Encode numeric parameters in a terminal specific form: **tparam(...)**, **tgoto(...)**.
- Compute and perform padding **tputs(...)**.

Programs using the termcap library must include *termcap.h* and should be linked with:

```
gcc [flags] files -ltermcap
```

Termcap functions are terminal independent routines but only give the programmer low level access to the terminal. For a higher level package, curses or ncurses should be used.

8.2.2 Find a Terminal Description

- `int tgetent(void *buffer, const char *termtype)`
  On the Linux operating system the current terminal name is contained in the environment variable *TERM*. So, `termtype` is the result of a call to *getenv(3)*.

  For `buffer`, no memory has to be allocated when using the GNU version of termcap. This is what we can assume under Linux! Otherwise, you'll have to allocate 2048 Bytes. (Formerly, `buffer` only needed to be 1024 Bytes, but the size has doubled).

tgetent(...) returns 1 on success and 0 when the database is found but has no entry for TERM. Other errors will return different values.

The following example should explain how to use **tgetent(...)**:

```
#define buffer 0
char *termtype=getenv("TERM");
int ok;

ok=tgetent(buffer,termtype);
if(ok==1)
  /* all right, we have the entry */
else if(ok==0)
 /* uups, something wrong with TERM
  * check termtype first, then termcap database
  */
else
 /* huuu, fatal error */
```

By, default termcap uses */etc/termcap/* as the database. If the environment variable TERMCAP is set, with *$HOME/mytermcap* for instance, all functions will use *mytermcap* instead of */etc/termcap*. With no leading slash in TERMCAP, the defined value is used as a name for a terminal.

8.2.3 Look at a Terminal Description

Every piece of information is called a capability, every capability is a two letter code, and every two letter code is followed by the value for the capability. Possible types are:

- **Numeric**: For instance *co* – number of columns
- **Boolean** or **Flag**: For instance *hc* – hardcopy terminal
- **String**: For instance *st* – set tab stop

Each capability is associated with a single value type. (*co* is always numeric, *hc* is always a flag and *st* is always a string). There are three different types of values, so there are also three functions to interrogate them. `char *name` is the two letter code for the capability.

- `int tgetnum(char *name)`
  Get a capability value that is numeric, such as *co*. **tgetnum(...)** returns the numeric value if the capability is available, otherwise 1. (Note: the returned value is not negative.)
- `int tgetflag(char *name)`
  Get a capability value that is boolean (or flag). Returns 1 if the flag is present, 0 otherwise.

- `char *tgetstr(char *name, char **area)`
  Get a capability value that is a string. Returns a pointer to the string or NULL if not present. In the GNU version, if `area` is NULL, termcap will allocate memory by itself. Termcap will never refer to this pointer again, so don't forget to free `name` before leaving the program. This method is preferred, because you don't know how much space is needed for the pointer, so let termcap do this for you.

```
char *clstr, *cmstr;
int   lines,  cols;

void term_caps()
{
char *tmp;

 clstr=tgetstr("cl",0); /* clear screen */
 cmstr=tgetstr("cm",0); /* move y,x     */

 lines=tgetnum("li"); /* terminal lines   */
 cols=tgetnum("co");  /* terminal columns */

 tmp=tgetstr("pc",0); /* padding character */

 PC=tmp ? *tmp : 0;
 BC=tgetstr("le",0); /* cursor left one char */
 UP=tgetstr("up",0); /* cursor up one line   */
}
```

8.2.4 Termcap Capabilities

Boolean Capabilities

`5i` Printer will not echo on screen
`am` Automatic margins which means automatic line wrap
`bs` Control-H (8 dec.) performs a backspace
`bw` Backspace on left margin wraps to previous line and right margin
`da` Display retained above screen
`db` Display retained below screen
`eo` A space erases all characters at cursor position
`es` Escape sequences and special characters work in status line
`gn` Generic device
`hc` This is a hardcopy terminal
`HC` The cursor is hard to see when not on bottom line
`hs` Has a status line
`hz` Hazel tine bug, the terminal can not print tilde characters
`in` Terminal inserts nulls, not spaces, to fill whitespace
`km` Terminal has a meta key
`mi` Cursor movement works in insert mode
`ms` Cursor movement works in standout/underline mode
`NP` No pad character
`NR` ti does not reverse te
`nx` No padding, must use XON/XOFF

- `os` Terminal can overstrike
- `ul` Terminal underlines although it can not overstrike
- `xb` Beehive glitch, f1 sends ESCAPE, f2 sends ˆC
- `xn` Newline/wraparound glitch
- `xo` Terminal uses xon/xoff protocol
- `xs` Text typed over standout text will be displayed in standout
- `xt` Teleray glitch, destructive tabs and odd standout mode

Numeric Capabilities

- `co` Number of columns
- `dB` Delay in milliseconds for backspace on hardcopy terminals
- `dC` Delay in milliseconds for carriage return on hardcopy terminals
- `dF` Delay in milliseconds for form feed on hardcopy terminals
- `dN` Delay in milliseconds for new line on hardcopy terminals
- `dT` Delay in milliseconds for tab stop on hardcopy terminals
- `dV` Delay in milliseconds for vertical tab stop on hardcopy terminals
- `it` Difference between tab positions
- `lh` Height of soft labels
- `lm` Lines of memory
- `lw` Width of soft labels
- `li` Number of lines
- `Nl` Number of soft labels
- `pb` Lowest baud rate which needs padding
- `sg` Standout glitch
- `ug` Underline glitch
- `vt` virtual terminal number
- `ws` Width of status line if different from screen width

String Capabilities

- `!1` shifted save key
- `!2` shifted suspend key
- `!3` shifted undo key
- `#1` shifted help key
- `#2` shifted home key
- `#3` shifted input key
- `#4` shifted cursor left key
- `%0` redo key
- `%1` help key
- `%2` mark key
- `%3` message key
- `%4` move key
- `%5` next-object key
- `%6` open key
- `%7` options key
- `%8` previous-object key
- `%9` print key
- `%a` shifted message key
- `%b` shifted move key
- `%c` shifted next key
- `%d` shifted options key
- `%e` shifted previous key
- `%f` shifted print key
- `%g` shifted redo key
- `%h` shifted replace key
- `%i` shifted cursor right key
- `%j` shifted resume key
- `&0` shifted cancel key
- `&1` reference key
- `&2` refresh key
- `&3` replace key
- `&4` restart key
- `&5` resume key
- `&6` save key
- `&7` suspend key
- `&8` undo key
- `&9` shifted begin key
- `*0` shifted find key
- `*1` shifted command key
- `*2` shifted copy key
- `*3` shifted create key
- `*4` shifted delete character
- `*5` shifted delete line
- `*6` select key
- `*7` shifted end key
- `*8` shifted clear line key
- `*9` shifted exit key
- `0` find key

`1`	begin key
`2`	cancel key
`3`	close key
`4`	command key
`5`	copy key
`6`	create key
`7`	end key
`8`	enter/send key
`9`	exit key
`al`	Insert one line
`AL`	Insert %1 lines
`ac`	Pairs of block graphic characters to map alternate character set
`ae`	End alternative character set
`as`	Start alternative character set for block graphic characters
`bc`	Backspace, if not ^H
`bl`	Audio bell
`bt`	Move to previous tab stop
`cb`	Clear from beginning of line to cursor
`cc`	Dummy command character
`cd`	Clear to end of screen
`ce`	Clear to end of line
`ch`	Move cursor horizontally only to column %1
`cl`	Clear screen and cursor home
`cm`	Cursor move to row %1 and column %2 (on screen)
`CM`	Move cursor to row %1 and column %2 (in memory)
`cr`	Carriage return
`cs`	Scroll region from line %1 to %2
`ct`	Clear tabs
`cv`	Move cursor vertically only to line %1
`dc`	Delete one character
`DC`	Delete %1 characters
`dl`	Delete one line
`DL`	Delete %1 lines
`dm`	Begin delete mode
`do`	Cursor down one line
`DO`	Cursor down #1 lines
`ds`	Disable status line
`eA`	Enable alternate character set
`ec`	Erase %1 characters starting at cursor
`ed`	End delete mode
`ei`	End insert mode
`ff`	Form-feed character on hardcopy terminals
`fs`	Return character to its position before going to status line
`F1`	The string sent by function key f11
`F2`	The string sent by function key f12
`F3`	The string sent by function key f13
	... ...
`F9`	The string sent by function key f19
`FA`	The string sent by function key f20
`FB`	The string sent by function key f21
	... ...
`FZ`	The string sent by function key f45
`Fa`	The string sent by function key f46
`Fb`	The string sent by function key f47
	... ...
`Fr`	The string sent by function key f63
`hd`	Move cursor a half line down
`ho`	Cursor home
`hu`	Move cursor a half line up
`i1`	Initialization string 1 at login
`i3`	Initialization string 3 at login
`is`	Initialization string 2 at login
`ic`	Insert one character
`IC`	Insert %1 characters
`if`	Initialization file
`im`	Begin insert mode
`ip`	Insert pad time and needed special characters after insert
`iP`	Initialization program
`K1`	upper left key on keypad
`K2`	center key on keypad
`K3`	upper right key on keypad
`K4`	bottom left key on keypad
`K5`	bottom right key on keypad
`k0`	Function key 0
`k1`	Function key 1
`k2`	Function key 2
`k3`	Function key 3
`k4`	Function key 4
`k5`	Function key 5
`k6`	Function key 6
`k7`	Function key 7
`k8`	Function key 8
`k9`	Function key 9
`k;`	Function key 10
`ka`	Clear all tabs key
`kA`	Insert line key
`kb`	Backspace key
`kB`	Back tab stop
`kC`	Clear screen key
`kd`	Cursor down key
`kD`	Key for delete character under cursor
`ke`	turn keypad off
`kE`	Key for clear to end of line
`kF`	Key for scrolling forward/down

`kh`	Cursor home key
`kH`	Cursor home down key
`kI`	Insert character/Insert mode key
`kl`	Cursor left key
`kL`	Key for delete line
`kM`	Key for exit insert mode
`kN`	Key for next page
`kP`	Key for previous page
`kr`	Cursor right key
`kR`	Key for scrolling backward/up
`ks`	Turn keypad on
`kS`	Clear to end of screen key
`kt`	Clear this tab key
`kT`	Set tab here key
`ku`	Cursor up key
`l0`	Label of zeroth function key, if not f0
`l1`	Label of first function key, if not f1
`l2`	Label of first function key, if not f2
	... ...
`la`	Label of tenth function key, if not f10
`le`	Cursor left one character
`ll`	Move cursor to lower left corner
`LE`	Cursor left %1 characters
`LF`	Turn soft labels off
`LO`	Turn soft labels on
`mb`	Start blinking
`MC`	Clear soft margins
`md`	Start bold mode
`me`	End all mode like so, us, mb, md and mr
`mh`	Start half bright mode
`mk`	Dark mode (Characters invisible)
`ML`	Set left soft margin
`mm`	Put terminal in meta mode
`mo`	Put terminal out of meta mode
`mp`	Turn on protected attribute
`mr`	Start reverse mode
`MR`	Set right soft margin
`nd`	Cursor right one character
`nw`	Carriage return command
`pc`	Padding character
`pf`	Turn printer off
`pk`	Program key %1 to send string %2 as if typed by user
`pl`	Program key %1 to execute string %2 in local mode
`pn`	Program soft label %1 to to show string %2
`po`	Turn the printer on
`pO`	Turn the printer on for %1 (<256) bytes
`ps`	Print screen contents on printer
`px`	Program key %1 to send string %2 to computer
`r1`	Reset string 1, set sane modes
`r2`	Reset string 2, set sane modes
`r3`	Reset string 3, set sane modes
`RA`	disable automatic margins
`rc`	Restore saved cursor position
`rf`	Reset string file name
`RF`	Request for input from terminal
`RI`	Cursor right %1 characters
`rp`	Repeat character %1 for %2 times
`rP`	Padding after character sent in replace mode
`rs`	Reset string
`RX`	Turn off XON/XOFF flow control
`sa`	Set %1 %2 %3 %4 %5 %6 %7 %8 %9 attributes
`SA`	enable automatic margins
`sc`	Save cursor position
`se`	End standout mode
`sf`	Normal scroll one line
`SF`	Normal scroll %1 lines
`so`	Start standout mode
`sr`	Reverse scroll
`SR`	scroll back %1 lines
`st`	Set tabulator stop in all rows at current column
`SX`	Turn on XON/XOFF flow control
`ta`	move to next hardware tab
`tc`	Read in terminal description from another entry
`te`	End program that uses cursor motion
`ti`	Begin program that uses cursor motion
`ts`	Move cursor to column %1 of status line
`uc`	Underline character under cursor and move cursor right
`ue`	End underlining
`up`	Cursor up one line
`UP`	Cursor up %1 lines
`us`	Start underlining
`vb`	Visible bell
`ve`	Normal cursor visible
`vi`	Cursor invisible
`vs`	Standout cursor
`wi`	Set window from line %1 to %2 and column %3 to %4
`XF`	XOFF character if not ^S

8.3 Ncurses - Introduction

The following terminology will be used in this chapter:

- window - is an internal representation containing an image of a part of the screen. WINDOW is defined in *ncurses.h*.
- screen - is a window with the size of the entire screen (from the upper left to the lower right). *Stdscr* and *curscr* are screens.
- terminal - is a special screen with information about what the screen currently looks like.
- variables - the following variables and constants defined in *ncurses.h*
  - WINDOW *curscr - current screen
  - WINDOW *stdscr - standard screen
  - int LINES - lines on the terminal
  - int COLS - columns on the terminal
  - bool TRUE - true flag, 1
  - bool FALSE - false flag, 0
  - int ERR - error flag, -1
  - int OK - ok flag, 0
- functions - in the function description the arguments are of the following type:
  - win - WINDOW*
  - bf - bool
  - ch - chtype
  - str - char*
  - chstr - chtype*
  - fmt - char*
  - otherwise int

Usually a program using the ncurses library looks like this:

```
#include <ncurses.h>
...
main()
{
  ...
  initscr();
  /* ncurses function calls */
  endwin();
  ...
}
```

Including *ncurses.h* will define variables and types for ncurses, such as WINDOW and function prototypes. It automatically includes *stdio.h*, *stdarg.h*, *termios.h* and *unctrl.h*.

initscr() is used to initialize the ncurses data structures and to read the proper terminfo file. Memory for *stdscr* and *curscr* will be allocated. If an error occurs, initscr will return ERR, otherwise a pointer to *stdscr* will be returned. Additionally, the screen will be erased and *LINES* and *COLS* will be initialized.

endwin() will clean up all allocated resources from ncurses and restore the tty modes to the status they had before calling **initscr()**. It must be called before any other function from the ncurses library and **endwin()** must be called before your program exits. When you want to do output to more than one terminal, you can use **newterm(...)** instead of **initscr()**.

Compile the program with:

```
gcc [flags] files -lncurses
```

In flags you can include anything you like (*see gcc(1)*). Since the path for ncurses.h has changed you have to include the following line:

```
-I/usr/include/ncurses
```

Otherwise, ncurses.h, nterm.h, termcap.h and unctrl.h will not be found. Possible other flags for Linux are:

```
-O2 -ansi -Wall -m486
```

O2 tells gcc to do some optimization, *-ansi* is for ansi conformant c-code, *-Wall* will print out all warnings, *-m486* will use optimized code for an Intel 486 (the binary can be used on an Intel 386, too).

The ncurses library can be found in /usr/lib/. There are three versions of the ncurses library:

- **libncurses.a** the normal ncurses library.
- **libdcurses.a** ncurses for debugging.
- **libpcurses.a** ncurses for profiling (since 1.8.6libpcurses.a exists no longer ?).
- **libcurses.a** No fourth version, but the original BSD curses (in my slackware 2.1.0 it is the bsd package).

The data structures for the screen are called *windows* as defined in *ncurses.h*. A window is something like a character array in memory which the programmer can manipulate without output to the terminal. The default window is *stdscr* with the size of the terminal. You can create other windows with **newwin(...)**.

To update the physical terminal optimally, ncurses has another window declared, *curscr*. This is an image of how the terminal actually looks and *stdscr* is an image of how the terminal should look. The output will be done when you call **refresh()**. Ncurses will then update *curscr* and the physical terminal with the information in *stdscr*. The library functions will use internal optimization for the update process so you can change different windows and then update the screen at once in the most optimal way.

With the ncurses functions you can manipulate the data structure *window*. Functions beginning with *w* allow you to specify a *window*, while others will usually affect *stdscr*. Functions beginning with *mv* will move the cursor to the position *y,x* first.

A character has the type *chtype* which is *long unsigned int* to store additional information about it (attributes etc.).

Ncurses use the *terminfo* database. Normally the database is located in *usr/lib/terminfo/* and ncurses will look there for local terminal definitions. If you want to test some other definitions for a terminal without changing the original terminfo, set the environment variable *TERMINFO*. Ncurses will check this variable and use the definitions stored there instead of /usr/lib/terminfo/.

Current ncurses version is 1.8.6(.

At the end of this chapter you can find a table with an overview for the BSD-Curses, ncurses and the curses from Sun-OS 5.4. Refer to it when you want to look for a specific function and where it is implemented.

8.4 Initializing

- `WINDOW *initscr()`
  This is the first function usually called from a program using ncurses. In some cases it is useful to call **slk_init(int)**, **filter()**, **ripoffline(...)** or **use_env(bf)** before **initscr()**. When using multiple terminals (or perhaps testing capabilities), you can use **newterm(...)** instead of **initscr()**.

  initscr() will read the proper terminfo file and initialize the ncurses data structures, allocate memory for *curscr* and *stdscr* and set *LINES* and *COLS* to the values the terminal has. It will return a pointer to *stdscr* or ERR when an error has occured. You don't need to initialize the pointer with:

  ```
  stdscr=initscr();
  ```

  initscr() will do this for you. If the return value is ERR, your program should exit because no ncurses function will work:

  ```
  if(!(initscr())){
    fprintf(stderr,"type: initscr() failed\n\n");
    exit (1);
  }
  ```

- `SCREEN *newterm(char *type, FILE *outfd, FILE *infd)`
  For multiple terminal output call **newterm(...)** for every terminal you would access with ncurses instead of **initscr()**. `type` is the name of the terminal as contained in $TERM (ansi, xterm, vt100, for example), `outfd` is the output pointer and `infd` is the pointer used for input. Call **endwin()** for every terminal opened with **newterm(...)**.

- `SCREEN *set_term(SCREEN *new)`
  With **set_term(SCREEN)** you can switch the current terminal. All functions will affect the current terminal which is set with **set_term(SCREEN)**.

- `int endwin()`
  endwin() will do the cleanup, restore the terminal modes in the state they had before calling **initscr()** and move the cursor to the lower left corner. Don't forget to close all opened windows before you call **endwin()** to exit your program.

  An additional call to **refresh()** after **endwin()** will restore the terminal to the status it had before calling **initscr()** (visual-mode) otherwise it will be cleared (non-visual-mode).

- `int isendwin()`
  Returns TRUE if **endwin()** was called with a following **refresh()**, otherwise FALSE.

- `void delscreen(SCREEN* sp)`
  After **endwin()** call **delscreen(SCREEN)** to free up all occupied resources, when SCREEN is no longer needed. (**Note:** not implemented yet.)

8.5 Windows

Windows can be created, deleted, moved, copied, touched, duplicated and more.

- `WINDOW *newwin(nlines, ncols, begy, begx)`
  `begy` and `begx` are the window coordinates of the upper left corner. `nlines` is an integer with the number of lines and `ncols` is an integer with the number of columns.

  ```
  WINDOW *mywin;
  mywin=newwin(10,60,10,10);
  ```

  The upper left corner of our window is in line 10 and column 10 and the window has 10 lines and 60 columns. If `nlines` is zero, the window will have $LINES - begy$ rows. In the same way the, window will have $COLS - begx$ columns when `ncols` is zero.

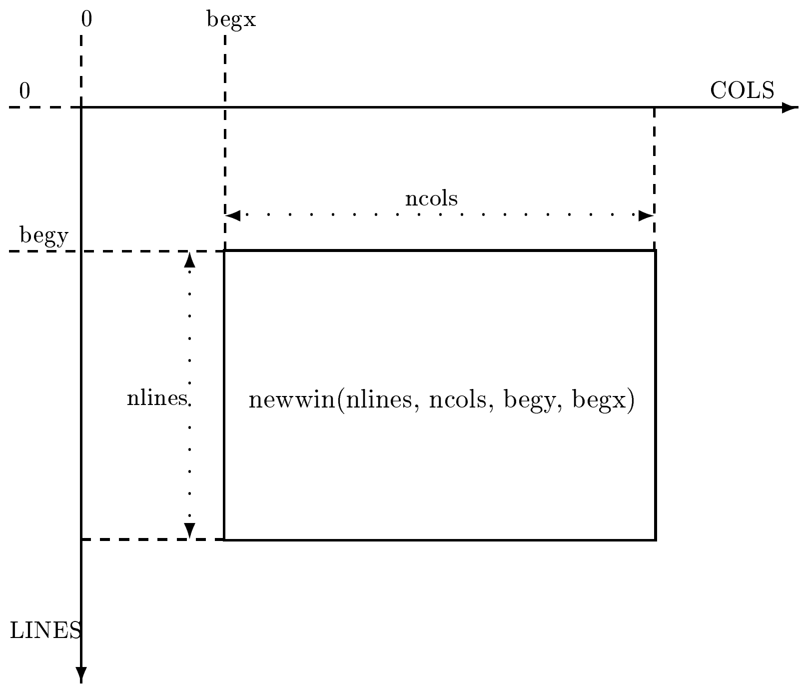
  When you call **newwin(...)** with all argument zero:

  ```
  WINDOW *mywin;
  mywin=newwin(0,0,0,0);
  ```

  the opened window will have the size of the screen.

  With *LINES* and *COLS* we can open windows in the middle of the screen, whatever dimension it has:

Figure 8.1: Ncurses - scheme for newwin




```
#define MYLINE (int) ((LINES-22)/2)
#define MYCOL ((COLS-70)/2)
#define MYLINES 22
#define MYCOLS  70
...
WINDOW *win;
...
if(!(initscr())){
  fprintf(stderr,"type: initscr() failed\n\n");
  exit(1);
}
...
if ((LINES<22)||(COLS<70)){
  fprintf(stderr,"screen to small\n\n");
  endwin(); exit (1);
}
win=newwin(MYLINES,MYCOLS,MYLINE,MYCOL);
...
```

This will open a window with 22 lines and 70 rows in the middle of the screen. Check the screen size before opening windows. In the Linux console we have 25 or more lines and 80 or more columns, but in xterms this may not be the case (they're resizable).

Alternatively, use *LINES* and *COLS* to adapt two windows to the screen size:

```
#define MYROWS   (int) (LINES/2+LINES/4)
#define MYCOLS   (int) (COLS/2+COLS/4)
#define LEFTROW  (int) ((LINES-MYROWS)/2)
#define LEFTCOL  (int) (((COLS-2)-MYCOLS)/2)
#define RIGHTROW (int) (LEFTROW)
#define RIGHTCOL (int) (LEFTROW+2+MYCOLS)
#define WCOLS    (int) (MYCOLS/2)
...
WINDOW *leftwin, *rightwin;
...
leftwin=newwin(MYROWS, WCOLS, LEFTROW, LEFTCOL);
rightwin=newwin(MYROWS, WCOLS, RIGHTROW, RIGHTCOL);
...
```

See *screen.c* in the example directory for more explanations.

- `int delwin(win)`
  Delete the window `win`. When there are subwindows delete them before `win`. It will free up all resources occupied by `win`. Delete all windows you have opened before calling **endwin()**.

- `int mvwin(win, by, bx)`
  Will move a window to the coordinates `by,bx`. If this means moving the window beyond the edges of the screen, nothing is done, and ERR is returned.

- `WINDOW *subwin(origwin, nlines, ncols, begy, begx)`
  Returns a subwindow in the middle of `origwin`. When you change one of the two windows (`origwin` or the new one) this change will be reflected in both windows. Call **touchwin(origwin)** before the next **refresh()**.

  `begx` and `begy` are relative to the screen, not to `origwin`.

- `WINDOW *derwin(origwin, nlines, ncols, begy, begx)`
  The same as **subwin(...)** except that `begx` and `begy` are relative to the window `origwin` than to the screen.

- `int mvderwin(win, y, x)`
  Will move `win` inside its parent window. (**Note:** not implemented yet.)

- `WINDOW *dupwin(win)`
  Duplicate the window `win`.

- `int syncok(win, bf)`
  `void wsyncup(win)`
  `void wcursyncup(win)`
  `void wsyncdown(win)`
  (**Note:** not implemented yet.)

- `int overlay(win1, win2)`
  `int overwrite(win1, win2)`
  overlay(...) will copy all text from `win1` to `win2` without copying blanks. **overwrite(...)** does the same, but copies blanks, too.

- `int copywin(win1, win2, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)`
  Similar to **overlay(...)** and **overwrite(...)**, but provides control over what region of the window to copy.

8.6 Output

- `int addch(ch)`
  `int waddch(win, ch)`
  `int mvaddch(y, x, ch)`
  `int mvwaddch(win, y, x, ch)`
  These functions are used for character output to a window. They will manipulate the window and you will have to call **refresh()** to put it on screen. **addch(...)** and **waddch(...)** put the character `ch` in the window *stdscr* or `win`. **mvaddch(...)** and **mvwaddch(...)** do the same except that they move the cursor to position y,x first.

- `int addstr(str)`
  `int addnstr(str, n)`
  `int waddstr(win, str)`
  `int waddnstr(win, str, n)`
  `int mvaddstr(y, x, str)`
  `int mvaddnstr(y, x, str, n)`
  `int mvwaddstr(win, y, x, str)`
  `int mvwaddnstr(win, y, x, str, n)`
  These functions write a string to a window and are equivalent to series of calls to **addch(...)**. `str` is a null terminated string (*"blafoo\0"*). Functions with *w* write the string `str` to the window `win`, while other funcions write to *stdscr*. Functions with `n` write n characters of `str`. If `n` is -1, the entire string `str` is written.

- `int addchstr(chstr)`
  `int addchnstr(chstr, n)`
  `int waddchstr(win, chstr)`
  `int waddchnstr(win, chstr, n)`
  `int mvaddchstr(y, x, chstr)`
  `int mvaddchnstr(y, x, chstr, n)`
  `int mvwaddchstr(win, y, x, chstr)`
  `int mvwaddchnstr(win, y, x, chstr, n)`
  These functions copy `chstr` to the window image (*stdscr* or `win`). The starting position is the current cursor position. Functions with `n` write n characters of `chstr`. If `n` is -1, the entire string `chstr` is written. The cursor is not moved and no control character check is done. These functions are faster than the **addstr(...)** routines. `chstr` is a pointer to an array of chtype.

- `int echochar(ch)`
  `int wechochar(win, ch)`

The same as call **addch(...)** (**waddch(...)** followed by **refresh()** (**wrefresh(win)**.

8.6.1 Formatted Output

- `int printw(fmt, ...)`
  `int wprintw(win, fmt, ...)`
  `int mvprintw(y, x, fmt, ...)`
  `int mvwprintw(win, y, x, fmt, ...)`
  `int vwprintw(win, fmt, va_list)`
  These functions correspond to **printf(...)** and its counterparts from *libc*.

  In the *libc* package **printf(...)** is used for formatted output. You can define an output string and include variables of different types in it. See section 8.1.1 on page 80 for more.

  For the use of **vwprintw(...)** you have to include also *varargs.h*.

8.6.2 Insert Characters/Lines

- `int insch(c)`
  `int winsch(win, c)`
  `int mvinsch(y,x,c)`
  `int mvwinsch(win,y,x,c)`
  Character `ch` is inserted to the left of the cursor and all characters are moved one position to the right. The character on the right end of the line may be lost).

- `int insertln()`
  `int winsertln(win)`
  Insert a blank line above the current one. (The bottom line will be lost).

- `int insdelln(n)`
  `int winsdelln(win, n)`
  For positive `n` these functions will insert n lines above the cursor in the appropriate window (so the n bottom lines will be lost). When `n` is negative, n lines under the cursor will be deleted and the rest will moved up.

- `int insstr(str)`
  `int insnstr(str, n)`
  `int winsstr(win, str)`
  `int winsnstr(win, str, n)`
  `int mvinsstr(y, x, str)`
  `int mvinsnstr(y, x, str, n)`
  `int mvwinsstr(win, y, x, str)`
  `int mvwinsnstr(win, y, x, str, n)`
  These functions will insert `str` in the current line left from the cursor (as many characters as fit to the line). The characters on the right of

the cursor are moved right and will be lost when the end of the line is reached. The cursor position is not changed.

`y` and `x` are the coordinates to which the cursor is moved before `str` will be inserted, `n` is the number of characters to insert (with `n`=0 the entire string is inserted).

8.6.3 Delete Characters/Lines

- `int delch()`
  `int wdelch(win)`
  `int mvdelch(y, x)`
  `int mvwdelch(win, y, x)`
  Delete the character under the cursor and move the remaining characters to the right of the cursor one position to the left.

  `y` and `x` are the coordinates to which the cursor will be moved to before deleting.

- `int deleteln()`
  `int wdeleteln(win)`
  Delete the line under the cursor and move all other lines below one position up. Additionally, the bottom line of the window will be erased.

8.6.4 Boxes and Lines

- `int border(ls, rs, ts, bs, tl, tr, bl, br)`
  `int wborder(win, ls, rs, ts, bs, tl, tr, bl, br)`
  `int box(win, vert, hor)`
  Draw a border around the edges of a window (*stdscr* or `win`). In the following table you see the characters and their default values when zero in a call to **box(...)**. In the picture you can see the position from the characters in a box.

- `int vline(ch, n)`
  `int wvline(win, ch, n)`
  `int hline(ch, n)`
  `int whline(win, ch, n)`
  These functions draw a vertical or horizontal line starting at the current cursor position. `ch` is the character to use and `n` is the number of characters to draw. The cursor position is not advanced.

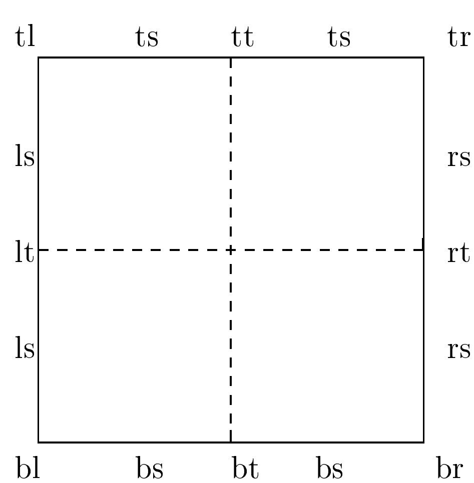
8.6.5 Background Character

- `void bkgdset(ch)`
  `void wbkgdset(win, ch)`
  Set the background character and attribute for the screen or a window. The attribute in `ch` will be ORed with every non blank character in the window. The background is then part of the window and will not be changed with scrolling and in- or output.

Table 8.3: Ncurses - border characters

Character	**Position**	**Default**
tl	top left	ACS_ULCORNER
ts	top side	ACS_HLINE
tr	top right	ACS_URCORNER
ls	left side	ACS_VLINE
rs	right side	ACS_VLINE
bl	bottom left	ACS_LLCORNER
bs	bottom side	ACS_HLINE
br	bottom right	ACS_LRCORNER
rt	right tee	ACS_RTEE
lt	left tee	ACS_LTEE
tt	top tee	ACS_TTEE
bt	bottom tee	ACS_BTEE

Figure 8.2: Ncurses - box characters



- `int bkgd(ch)`
  `int wbkgd(win, ch)`
  Will change the background character and attribute to `ch`.

8.7 Input

- `int getch()`
  `int wgetch(win)`
  `int mvgetch(y, x)`
  `int mvwgetch(win, y, x)`
  getch() will read input from the terminal in a manner depending on whether delay mode is set or not. If delay is on, **getch()** will wait until a key is pressed, otherwise it will return the key in the input buffer or ERR if this buffer is empty. **mvgetch(...)** and **mvwgetch(...)** will move the cursor to position `y,x` first. The `w` functions read input from the terminal related to the window `win`, **getch()** and **mvgetch(...)** from the terminal related to *stdscr*.

  With **keypad(...)** enabled, **getch()** will return a code defined in *ncurses.h* as KEY_* macros when a function key is pressed. When ESCAPE is pressed (which can be the beginning of a function key) ncurses will start a one second timer. If the remainder of the keystroke is not finished in this second, the key is returned. Otherwise, the function key value is returned. (If necessary, use **notimeout()** to disable the second timer).

- `int ungetch(ch)`
  Will put the character `ch` back to the input buffer.

- `int getstr(str)`
  `int wgetstr(win, str)`
  `int mvgetstr(y, x, str)`
  `int mvwgetstr(win, y, x, str)`
  `int wgetnstr(win, str, n)`
  These functions will do a series of calls to **getch()** until a newline is received. The characters are placed in `str` (so don't forget to allocate memory for your character pointer before calling **getstr(...)**). If echo is enabled the string is echoed (use **noecho()** to disable echo) and the user's kill and delete characters will be interpreted.

- `chtype inch()`
  `chtype winch(win)`
  `chtype mvinch(y, x)`
  `chtype mvwinch(win, y, x)`
  These functions return a character from the screen or window. Because the type of the return value is `chtype` attribute information is included. This information can be extracted from the character using the A_* constants (see table 8.4 on page 109).

- `int instr(str)`
  `int innstr(str, n)`
  `int winstr(win, str)`
  `int winnstr(win, str, n)`
  `int mvinstr(y, x, str)`
  `int mvinnstr(y, x, str, n)`
  `int mvwinstr(win, y, x, str)`
  `int mvwinnstr(win, y, x, str, n)`
  Return a character string from the screen or a window. (**Note:** not implemented yet.)

- `int inchstr(chstr)`
  `int inchnstr(chstr, n)`
  `int winchstr(win, chstr)`
  `int winchnstr(win, chstr, n)`
  `int mvinchstr(y, x, chstr)`
  `int mvinchnstr(y, x, chstr, n)`
  `int mvwinchstr(win, y, x, chstr)`
  `int mvwinchnstr(win, y, x, chstr, n)`
  Return a chtype string from the screen or window. In the string, attribute information is included for every character. (**Note:** not implemented yet, lib_inchstr not included in the ncurses lib.)

8.7.1 Formated Input

- `int scanw(fmt, ...)`
  `int wscanw(win, fmt, ...)`
  `int mvscanw(y, x, fmt, ...)`
  `int mvwscanw(win, y, x, fmt, ...)`
  `int vwscanw(win, fmt, va_list)`
  These are similar to **scanf(...)** from *libc*(see section 8.1.2 on page 81). **wgetstr(...)** is called and the results will be used as input for the scan.

8.8 Options

Output Options

- `int idlok(win, bf)`
  `void idcok(win, bf)`
  Enable or disable terminal's insert/delete features for the window (**idlok(...)** for lines and **idcok(...)** for characters). (**Note: idcok(...)** not implemented yet.)

- `void immedok(win, bf)`
  If set TRUE, every change to the window `win` will cause a refresh to the physical screen. This can decrease the performance of a program, so the default value is FALSE. (**Note:** not implemented yet.)

???	KEY_HOME	KEY_PPAGE
CTRL+D	KEY_END	KEY_NPAGE

	KEY_UP	
KEY_LEFT	KEY_DOWN	KEY_RIGHT

<table>
<tr><td>NUM</td><td>/</td><td>*</td><td>-</td></tr>
<tr><td>KEY_HOME</td><td>KEY_UP</td><td>KEY_PPAGE</td><td rowspan="2">+</td></tr>
<tr><td>KEY_LEFT</td><td>???</td><td>KEY_RIGHT</td></tr>
<tr><td>KEY_END</td><td>KEY_DOWN</td><td>KEY_NPAGE</td><td rowspan="2">CTRL+M</td></tr>
<tr><td colspan="2">???</td><td>KEY_DC</td></tr>
</table>

- `int clearok(win, bf)`
  If `bf` is TRUE the next call to **wrefresh(win)** will clear the screen and redraw it completely. (as in the editor vi when you press CTRL+L).

- `int leaveok(win, bf)`
  The default behavior is for ncurses to leave the physical cursor in the same place it was on the last refresh of the window. Programs which don't use the cursor can set **leaveok(...)** TRUE and save the time normally required for cursor motion. In addition, ncurses will try to make the terminal cursor invisible.

- `int nl()`
  `int nonl()`
  Control the translation for newline. Turned on with **nl()** will translate a newline in carriage return and line feed on output. **nonl()** will turn translation off. With disabled translation ncurses can do faster cursor motion.

8.8.1 Input Options

- `int keypad(win, bf)`
  If TRUE, it enables the keypad on the keyboard of the user's terminal when waiting for input. Ncurses will then return a key code defined in *ncurses.h* as KEY_* for the function and arrow keys on the keypad. This is very useful for a PC keyboard because you can enable the numerical block and the cursor keys.

- `int meta(win, bf)`
  If TRUE, the key codes returned from **getch()** are 8-bit-clean (the highest bit will not be stripped).

- `int cbreak()`
  `int nocbreak()`
  `int crmode()`
  `int nocrmode()`

cbreak() and **nocbreak()** will turn the terminal CBREAK mode on or off. When CBREAK is on, input from a read will be immediately available to the program, when off the input will be buffered until newline occurs. (**Note: crmode()** and **nocrmode()** are for upward compatibility, don't use them.)

- `int raw()`
  `int noraw()`
  Turn RAW mode on or off. RAW is the same as CBREAK, except that in RAW mode no special character processing will be done.

- `int echo()`
  `int noecho()`
  Set **echo()** to echo input typed by the user and **noecho()** to be silent about it.

- `int halfdelay(t)`
  As **cbreak()** with a delay of `t` seconds.

- `int nodelay(win, bf)`
  Terminal will be set to no blocking mode. **cetch()** will return ERR if no input is ready. If set to FALSE, **getch()** will wait until a key is pressed.

- `int timeout(t)`
  `int wtimeout(win, t)`
  It is recommended to use these functions instead of **halfdelay(t)** and **nodelay(win,bf)**. The result of **getch()** depends on the value of `t`. If `t` is positive, the read is blocked for `t` milliseconds, if `t` is zero, no blocking is done, and when `t` is negative the program blocks until input is available.

- `int notimeout(win, bf)`
  If `bf` is TRUE, **getch()** will use a special timer (of one second length) to interpret and input sequence beginning with keys as ESCAPE etc.

- `int typeahead(fd)`
  If `fd` is -1 no typeahead check will be done, else ncurses will use the file descriptor `fd` instead of *stdin* for these checks.

- `int intrflush(win, bf)`
  When enabled with `bf` TRUE an interrupt key pressed on the terminal (quit, break ...) will flush all output in the tty driver queue.

- `void noqiflush()`
  `void qiflush()`
  (**Note:** not implemented yet.)

8.8.2 Terminal Attributes

- `int baudrate()`
  Returns the terminal speed in bps.

- `char erasechar()`
  Returns the current erase character.

- `char killchar()`
  Returns the current kill character.

- `int has_ic()`
  `int has_il()`
  has_ic() returns TRUE if the terminal has insert/delete character capability, **has_il()** returns TRUE when the terminal has insert/delete line capability. Otherwise the functions return ERR. (**Note:** not implemented yet.)

- `char *longname()`
  The returned pointer gives access to the description of the current terminal.

- `chtype termattrs()`
  (**Note:** not implemented yet.)

- `char *termname()`
  Returns the contents of TERM from the users environment. (**Note:** not implemented yet.)

8.8.3 Use Options

Now we have seen the window options and terminal modes it is time to describe their use.

First, on Linux you should enable the keypad. This will allow use of the cursor keys and the numeric block on the PC keyboard.

```
keypad(stdscr,TRUE);
```

Now, there are two main types of input.

1. The program wants the user to enter a key and then will call a function depend on this key. (For example, something like "press 'q' for quit" and wait for *q*)

2. The program wants a string of characters typed by the user in a mask on the screen. For example: a directory or an address in a database.

For the first we use the following options and modes and the while loop will work correctly.

```
char c;

noecho();
timeout(-1);
nonl();
cbreak();
```

```
    keypad(stdscr,TRUE);
    while(c=getch()){
      switch(c){
        case 'q': your_quit_function();
        default:  break;
      }
   }
```

The program will hang until a key is pressed. If the key was *q* we call our quit function else we wait for other input.

The switch statement can be expanded until we have an input function that fits our wishes. Use the KEY_* macros to check special keys, for instance

```
    KEY_UP      KEY_RIGHT    KEY_A1    KEY_B2    KEY_C1
    KEY_DOWN    KEY_LEFT     KEY_A3              KEY_C3
```

for the cursor keys on the keyboard. For a file viewer the loop can look like this:

```
    int loop=TRUE;
    char c;
    enum{UP,DOWN,RIGHT,LEFT};

    noecho();
    timeout(-1);
    nonl();
    cbreak();
    keypad(stdscr,TRUE);
    while(loop==TRUE){
      c=getch();
      switch(c){
        case KEY_UP:
        case 'u':
        case 'U': scroll_s(UP);
                  break;
        case KEY_DOWN:
        case 'd':
        case 'D': scroll_s(DOWN);
                  break;
        case KEY_LEFT:
        case 'l':
        case 'L': scroll_s(LEFT);
                  break;
        case KEY_RIGHT
        case 'r':
        case 'R': scroll_s(RIGHT);
                  break;
        case 'q':
        case 'Q': loop=FALSE;
        default:  break;
      }
   }
```

For the second, we only need to set **echo()** and the characters typed by the user will be printed to the screen. To have the characters printed on the position you want, use **move(...)** or **wmove(...)**.

Or, we could open a window with a mask in it (some other colors than those of the window will do this) and ask the user to input a string:

```
WINDOW *maskwin;
WINDOW *mainwin;
char *ptr=(char *)malloc(255);
...
  mainwin=newwin(3,37,9,21);
  maskwin=newwin(1,21,10,35);
  ...
  werase(mainwin);
  werase(maskwin);
  ...
  box(mainwin,0,0);
  mvwaddstr(mainwin,1,2,"Inputstring: ");
  ...
  wnoutrefresh(mainwin);
  wnoutrefresh(maskwin);
  doupdate();
  ...
  mvwgetstr(maskwin,0,0,ptr);
  ...
  delwin(maskwin);
  delwin(mainwin);
  endwin();
  free(ptr);
```

See *input.c* in the example directory for more explanation.

8.9 Clear Window and Lines

- `int erase()`
  `int werase(win)`
  werase(...) and **erase()** will copy blanks to every position on the window `win` or *stdscr*. For instance, when you set color attributes to a window and call **werase()** the window would be colored. I had some problems with COLOR_PAIRS when I defined other attributes then black on white so I wrote my own erase function (this is a low level access to the WINDOW structure):

```
void NewClear(WINDOW *win)
{
int  y,x;

 for ( y = 0 ; y <= win -> _maxy ; y++ )
   for ( x = 0 ; x <= win -> _maxx ; x++ )
     (chtype *) win-> _line[y][x] = ' '|win-> _attrs;
 win -> _curx = win -> _cury = 0;
```

```
    touchwin(win);
  }
```

The problem is, that ncurses sometimes makes no use of the window attributes when blanking the screen. For instance, in *lib_clrtoeol.c*, is BLANK defined as

```
#define BLANK ' '|A_NORMAL
```

so that the other window attributes get lost while the line is erased.

- `int clear()`
  `int wclear(win)`
  The same as **erase()**, but will also set **clearok()** (the screen will be cleared with the next refresh).

- `int clrtobot()`
  `int wclrtobot(win)`
  Clearing the current cursor line (start is one character right from the cursor) and the line below the cursor.

- `int clrtoeol()`
  `int wclrtoeol(win)`
  Clear the current line right from the cursor up to its end.

8.10 Updating the Terminal

As written in the overview, ncurses windows are images in memory. This means that any change to a window is not printed to the physical screen until a refresh is done. This optimizes the output to the screen because you can do a lot of manipulations and then, once, call refresh to print it to screen. Otherwise, every change would be printed to the terminal and decrease the performance of your programs.

- `int refresh()`
  `int wrefresh(win)`
  refresh() copies *stdscr* to the terminal and **wrefresh(win)** copies the window image to *stdscr* and then makes *curscr* looks like *stdscr*.

- `int wnoutrefresh(win)`
  `int doupdate()`
  wnoutrefresh(win) copies the window `win` to *stdscr* only. This means that no output to the terminal is done but the virtual screen *stdscr* actually looks like the programmer wanted. **doupdate()** will do the output to the terminal. A program can change various windows, call **wnoutrefresh(win)** for every window and then call **doupdate()** to update the physical screen only once.

  For instance, we have the following program with two windows. We change both windows by altering some lines of text. We can write *changewin(win)* with **wrefresh(win)**.

```
main()                    changewin(WINDOW *win)
{                         {
WINDOW *win1,*win2;           ... /* here we change */
    ...                       ... /* the lines */
    changewin(win1);          wrefresh(win);
    changewin(win2);      return;
    ...                   }
}
```

This will cause ncurses to update the terminal twice and slow down our execution. With **doupdate()** we change *changewin(win)* and our main function and will get better a performance.

```
main()                    changewin(WINDOW *win)
{                         {
WINDOW *win1,*win2;           ... /* here we change */
    ...                       ... /* the lines */
    changewin(win1);          wnoutrefresh(win);
    changewin(win2);      return;
    doupdate();           }
    ...
}
```

- `int redrawwin(win)`
  `int wredrawln(win, bline, nlines)`
  Use these functions when some lines or the entire screen should thrown away before writing anything new in it (may be when the lines are trashed or so).

- `int touchwin(win)`
  `int touchline(win, start, count)`
  `int wtouchln(win, y, n, changed)`
  `int untouchwin(win)`
  Tells ncurses that the whole window `win` or the lines from `start` up to `start+count` have been manipulated. For instance, when you have some overlapping windows (as in the example *type.c*) a change to one window will not affect the image from the other.

  wtouchln(...) will touch `n` lines starting at `y`. If `change` is TRUE the lines are touched, otherwise untouched (changed or unchanged).

  untouchwin(win) will mark the window `win` as unchanged since the last call to **refresh()**.

- `int is_linetouched(win, line)`
  `int is_wintouched(win)`
  With these functions you can check if the line `line` or the window `win` has been touched since the last call to **refresh()**.

Table 8.4: Ncurses - attributes

Definition	**Attribute**
A_ATTRIBUTES	mask for attributes (chtype)
A_NORMAL	normal, reset all other
A_STANDOUT	best highlighting mode
A_UNDERLINE	underline
A_REVERSE	reverse video
A_BLINK	blinking
A_DIM	dim or half bright
A_BOLD	bold or extra bright
A_ALTCHARSET	use alternate character set
A_INVIS	invisible
A_PROTECT	???
A_CHARTEXT	mask for actual character (chtype)
A_COLOR	mask for color
COLOR_PAIR(n)	set color-pair to that stored in n
PAIR_NUMBER(a)	get color-pair stored in attribute a

8.11 Video Attributes and Color

Attributes are special terminal capabilities used when printing characters to the screen. Characters can be printed bold, underlined, blinking, etc. In ncurses you have the ability to turn attributes on or off to get better looking output. Possible attributes are listed in the following table.

Ncurses defines eight colors you can use on a terminal with color support. First, initialize the color data structures with **start_color()**, then check the terminal capabilities with **has_colors()**. **start_color()** will initialize *COLORS*, the maximum colors the terminal supports, and *COLOR_PAIR*, the maximum number of color pairs you can define.

The attributes can be combined with the OR operator '|' so that you can

Table 8.5: Ncurses - colors

Definition	**Color**
COLOR_BLACK	black
COLOR_RED	red
COLOR_GREEN	green
COLOR_YELLOW	yellow
COLOR_BLUE	blue
COLOR_MAGENTA	magenta
COLOR_CYAN	cyan
COLOR_WHITE	white

produce bold blinking output with

```
A_BOLD|A_BLINK
```

When you set a window to attribute `attr`, all characters printed to this window will get this property until you change the window attribute. It will not get lost when you scroll or move the window or anything else.

When you write programs for ncurses and BSD curses be careful with colors because BSD curses has no color support. (Also, old SYS V versions of curses do not have color support). So you have to use *#ifdef* operations when you compile for both libraries.

- `int attroff(attr)`
  `int wattroff(win, attr)`
  `int attron(attr)`
  `int wattron(win, attr)`
  Turn on or off the specified attribute `attr` without reflecting the other attributes in a window (*stdscr* or `win`).

- `int attrset(attr)`
  `int wattrset(win, attr)`
  Set the attribute on *stdscr* or `win` to `attr`.

- `int standout()`
  `int standend()`
  `int wstandout(win)`
  `int wstandend(win)`
  Turn on standout attribute for the window (*stdscr* or `win`).

- `chtype getattrs(win)`
  Return the current attributes for window `win`.

- `bool has_colors()`
  Returns TRUE if the terminal has colors. Before you use colors check the terminal with **has_colors()**, and before this initialize colors with **start_color())**.

- `bool can_change_color()`
  TRUE if the terminal can redefine colors.

- `int start_color()`
  Color initializing. This function has to be called before using colors!

- `int init_pair(pair, fg, bg)`
  When you use colors as attributes for windows you have first to define a color pair with **init_pair(...)**. `fg` is the foreground color and `bg` the background color for `pair`. This is a value from 1 to $COLORPAIRS-1$ (No fault, but 0 is reserved for black on white). Once defined you can use `pair` like an attribute. For instance when you want to have red characters on a blue screen do:

```
init_pair(1,COLOR_RED,COLOR_BLUE);
```

Now use **wattr(...)** to set `win` to our new color pair:

```
wattr(win,COLOR_PAIR(1));
```

Or, combine color pairs with other attributes, such as:

```
wattr(win ,A_BOLD|COLOR_PAIR(1));
wattr(win1,A_STANDOUT|COLOR_PAIR(1));
```

The first will invoke the color pair and set the attribute BOLD and the second will turn on standout mode, so that you get highlighted red on a blue screen.

- `int pair_content(pair, f, b)`
  Will return the foreground and background color from `pair`.
- `int init_color(color, r, g, b)`
  Change the color components `r`, `g` and `b` for `color`. `r`, `g` and `b` can have values from 1 to $COLORS - 1$.
- `int color_content(color, r, g, b)`
  Get the color components `r`, `g` and `b` for `color`.

And how to combine attributes and colors? Some terminals, as the console in Linux, have colors and some not (xterm, vs100 etc). The following code should solve the problem:

```
void CheckColor(WINDOW *win1, WINDOW *win2)
{
 start_color();
 if (has_colors()){
   /* fine, we have colors, define color_pairs for foreground
    * and background colors
    */
   init_pair(1,COLOR_BLUE,COLOR_WHITE);
   init_pair(2,COLOR_WHITE,COLOR_RED);
   /* now use the defined color_pairs for the windows */
   wattrset(win1,COLOR_PAIR(2));
   wattrset(win2,COLOR_PAIR(1));
 }
 else{
   /* Ohh, no color (maybe a vt100 or xterm). OK, we'll
    * use black/white attributes instead.
    */
   wattrset(win1,A_REVERSE);
   wattrset(win2,A_BOLD);
 }
return;
}
```

First, the function *CheckColor* initializes the colors with **start_color()**, then the function **has_colors()** will return TRUE if the current terminal has colors. We check this and call **init_pair(...)** to combine foreground and background colors and **wattrset(...)** to set these pairs for the specified window. Alternatively, we can use **wattrset(...)** alone to set attributes if we have a black and white terminal.

To get colors in an xterm the best way I found out is to use the ansi_xterm with the patched terminfo entries from the Midnight Commander. Just get the sources of ansi_xterm and Midnight Commander (mc-x.x.tar.gz). Then compile the ansi_xterm and use tic with xterm.ti and vt100.ti from the mc-x.x.tar.gz archive. Execute ansi_xterm and test it out.

8.12 Cursor and Window Coordinates

- `int move(y, x)`
  `int wmove(win, y, x)`
  move() moves the cursor from *stdscr*, **wmove(win)** the cursor from window `win`. For input/output functions, additional macros are defined which move the cursor before the specified function is called.

- `int curs_set(bf)`
  This will turn the cursor visibility on or off, if the terminal has this capability.

- `void getyx(win, y, x)`
  getyx(...) will return the current cursor position. (**Note:** this is a macro.)

- `void getparyx(win, y, x)`
  When `win` is a sub window, **getparyx(...)** will store the window coordinates relative to the parent window in `y` and `x`. Otherwise `y` and `x` are -1. (**Note:** not implemented yet.)

- `void getbegyx(win, y, x)`
  `void getmaxyx(win, y, x)`
  `int getmaxx(win)`
  `int getmaxy(win)`
  Store the begin and size coordinates for `win` in `y` and `x`.

- `int getsyx(int y, int x)`
  `int setsyx(int y, int x)`
  Store the virtual screen cursor in `y` and `x` or set this cursor. When `y` and `x` are -1 and you call **getsyx(...)** *leaveok* will be set.

8.13 Scrolling

- `int scrollok(win, bf)`
  If TRUE, the text in the window `win` will be scrolled up one line when the

cursor is on the lower right corner and a character is typed (or newline). If FALSE, the cursor is left in the same position.

When turned on the contents of a window can be scrolled with the following functions. (**Note:** It would be also scrolled, if you print a new line in the last line of the window. So, be careful with **scrollok(...)** or you will get unreasonable results.)

- `int scroll(win)`
  This function will scroll up the window (and the lines in the data structure) one line.

- `int scrl(n)`
  `int wscrl(win, n)`
  These functions will scroll the window *stdscr* or `win` up or down depending on the value of the integer `n`. If `n` is positive the window will be scrolled up `n` lines, otherwise if `n` is negative the window will be scrolled down `n` lines.

- `int setscrreg(t, b)`
  `int wsetscrreg(win, t, b)`
  Set a software scrolling region.

The following code should explain how to get the effect of scrolling a text on the screen. Look also in *type.c* in the example directory.

We have a window with 18 lines and 66 columns and want to scroll a text in it. *S*[] is a character array with the text. *Max_s* is the number of the last line in *s*[]. *Clear_line* will print blank characters from the current cursor position up to the end of the line using the current attributes from the window (not A_NORMAL as clrtoeol does). *Beg* is the last line from *s*[] currently shown on the screen. *Scroll* is an enumerate to tell the function what to do, show the NEXT or PREVious line from the text.

```
enum{PREV,NEXT)};

void scroll_s(WINDOW *win, int scroll)
{
 /* test if we should scroll down and if there is
  * anything to scroll down
  */
 if((scroll==NEXT)&&(beg<=(max_s-18))){
 /* one line down */
     beg++;
 /* give permissions to scroll */
     scrollok(win, TRUE);
 /* scroll */
     wscrl(win, +1);
 /* deny permission to scroll */
     scrollok(win, FALSE);
 /* set the new string in the last line */
     mvwaddnstr(win,17,0,s[beg+17],66);
```

```
 /* clear the last line from the last character up to end
  * of line. Otherwise the attributes will be garbaged.
  */
     clear_line(66,win);
 }
 else if((scroll==PREV)&&(beg>0)){
     beg--;
     scrollok(win, TRUE);
     wscrl(win, -1);
     scrollok(win, FALSE);
     mvwaddnstr(win,0,0,s[beg],66);
     clear_line(66,win);
 }
 wrefresh(win);
return;
}
```

8.14 Pads

- `WINDOW *newpad(nlines, ncols)`
- `WINDOW *subpad(orig, nlines, ncols, begy, begx)`
- `int prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)`
- `int pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)`
- `int pechochar(pad, ch)`

8.15 Soft-labels

- `int slk_init(int fmt)`
- `int slk_set(int labnum, char *label, int fmt)`
- `int slk_refresh()`
- `int slk_noutrefresh()`

- `char *slk_label(int labnum)`

- `int slk_clear()`

- `int slk_restore()`

- `int slk_touch()`

- `int slk_attron(chtype attr)`
  `int slk_attrset(chtype attr)`
  `int slk_attroff(chtype attr)`
  These functions correspond to **attron(attr)**, **attrset(attr)** and **attroff(attr)**. Not implemented yet.

8.16 Miscellaneous

- `int beep()`

- `int flash()`

- `char *unctrl(chtype c)`

- `char *keyname(int c)`

- `int filter()`
  (**Note:** not implemented yet.)

- `void use_env(bf)`

- `int putwin(WINDOW *win, FILE *filep)`
  (**Note:** not implemented yet.)

- `WINDOW *getwin(FILE *filep)`
  (**Note:** not implemented yet.)

- `int delay_output(int ms)`

- `int flushinp()`

8.17 Low-level Access

- `int def_prog_mode()`

- `int def_shell_mode()`

- `int reset_prog_mode()`

- `int reset_shell_mode()`

- `int resetty()`

- `int savetty()`

- `int ripoffline(int line, int (*init)(WINDOW *, int))`

- `int napms(int ms)`

8.18 Screen Dump

- `int scr_dump(char *filename)`
  (**Note:** not implemented yet.)
- `int scr_restore(char *filename)`
  (**Note:** not implemented yet.)
- `int scr_init(char *filename)`
  (**Note:** not implemented yet.)
- `int scr_set(char *filename)`
  (**Note:** not implemented yet.)

8.19 Termcap Emulation

- `int tgetent(char *bp, char *name)`

- `int tgetflag(char id[2])`

- `int tgetnum(char id[2])`

- `char *tgetstr(char id[2], char **area)`
- `char *tgoto(char *cap, int col, int row)`
- `int tputs(char *str, int affcnt, int (*putc)())`

8.20 Terminfo Functions

- `int setupterm(char *term, int fildes, int *errret)`
- `int setterm(char *term)`
- `int set_curterm(TERMINAL *nterm)`
- `int del_curterm(TERMINAL *oterm)`
- `int restartterm(char *term, int fildes, int *errret)`
  (**Note:** not implemented yet.)
- `char *tparm(char *str, p1, p2, p3, p4, p5, p6, p7, p8, p9)`
  p1 - p9 long int.
- `int tputs(char *str, int affcnt, int (*putc)(char))`
- `int putp(char *str)`
- `int vidputs(chtype attr, int (*putc)(char))`
- `int vidattr(chtype attr)`
- `int mvcur(int oldrow, int oldcol, int newrow, int newcol)`
- `int tigetflag(char *capname)`
- `int tigetnum(char *capname)`
- `int tigetstr(char *capname)`

8.21 Debug Function

- `void _init_trace()`
- `void _tracef(char *, ...)`
- `char *_traceattr(mode)`
- `void traceon()`
- `void traceoff()`

8.22 Terminfo Capabilities

8.22.1 Boolean Capabilities

Variable	Cap. Name	Int. Code	Description
auto_left_margin	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
back_color_erase	bce	ut	screen erased with background color
can_change	ccc	cc	terminal can re-define exiting colors
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting (hp)
col_addr_glitch	xhpa	YA	only positive motion for hpa/mhpa caps
cpi_changes_res	cpix	YF	changing character pitch changes resolution
cr_cancels_micro_mode	crxm	YB	using cr turns off micro mode
eat_newline_glitch	xenl	xn	newline ignored after 80 cols (Concept)
erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type (e.g.,, dialup, switch).
hard_copy	hc	hc	Hardcopy terminal
hard_cursor	chts	HC	cursor is hard to see
has_meta_key	km	km	Has a meta key (shift, sets parity bit)
has_print_wheel	daisy	YC	printer needs operator to change character set
has_status_line	hs	hs	Has extra "status line"
hue_lightness_saturation	hls	hl	terminal uses only HLS color notation (Tektronix)
insert_null_glitch	in	in	Insert mode distinguishes nulls
lpi_changes_res	lpix	YG	changing line pitch changes resolution
memory_above	da	da	Display may be retained above the screen
memory_below	db	db	Display may be retained below the screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
needs_xon_xoff	nxon	nx	padding won't work, xon/xoff required
no_esc_ctl_c	xsb	xb	Beehive (f1=escape, f2=ctrl C)
non_rev_rmcup	nrrmc	NR	smcup does not reverse rmcup
no_pad_char	npc	NP	pad character does not exist
non_dest_scroll_region	ndscr	ND	scrolling region is non-destructive

over_strike	`os`	`os`	Terminal overstrikes
prtr_silent	`mc5i`	`5i`	printer won't echo on screen
row_addr_glitch	`xvpa`	`YD`	only positive motion for vhp/mvpa caps
semi_auto_right_margin	`sam`	`YE`	printing in last column causes cr
status_line_esc_ok	`eslok`	`es`	Escape can be used on the status line
dest_tabs_magic_smso	`xt`	`xt`	Tabs ruin, magic so char (Teleray 1061)
tilde_glitch	`hz`	`hz`	Hazel-tine; can not print ~'s
transparent_underline	`ul`	`ul`	underline character overstrikes
xon_xoff	`xon`	`xo`	Terminal uses xon/xoff handshaking

8.22.2 Numbers

Variable	**Cap. Name**	**Int. Code**	**Description**
bit_image_entwining	`bitwin`	`Yo`	Undocumented in SYSV
buffer_capacity	`bufsz`	`Ya`	numbers of bytes buffered before printing
columns	`cols`	`co`	Number of columns in a line
dot_vert_spacing	`spinv`	`Yb`	spacing of dots horizontally in dots per inch
dot_horz_spacing	`spinh`	`Yc`	spacing of pins vertically in pins per inch
init_tabs	`it`	`it`	Tabs initially every # spaces
label_height	`lh`	`lh`	rows in each label
label_width	`lw`	`lw`	columns in each label
lines	`lines`	`li`	Number of lines on screen or page
lines_of_memory	`lm`	`lm`	Lines of memory if ¿ lines. 0 means varies
magic_cookie_glitch	`xmc`	`sg`	Number of blank chars left by smso or rmso
max_colors	`colors`	`Co`	maximum numbers of colors on screen
max_micro_address	`maddr`	`Yd`	maximum value in micro_..._address
max_micro_jump	`mjump`	`Ye`	maximum value in parm_..._micro
max_pairs	`pairs`	`pa`	maximum number of color-pairs on the screen
micro_col_size	`mcs`	`Yf`	Character step size when in micro mode
micro_line_size	`mls`	`Yg`	Line step size when in micro mode
no_color_video	`ncv`	`NC`	video attributes that can't be used with colors
number_of_pins	`npins`	`Yh`	numbers of pins in print-head
num_labels	`nlab`	`Nl`	number of labels on screen
output_res_char	`orc`	`Yi`	horizontal resolution in units per line
output_res_line	`orl`	`Yj`	vertical resolution in units per line
output_res_horz_inch	`orhi`	`Yk`	horizontal resolution in units per inch
output_res_vert_inch	`orvi`	`Yl`	vertical resolution in units per inch
padding_baud_rate	`pb`	`pb`	Lowest baud where cr/nl padding is needed
virtual_terminal	`vt`	`vt`	Virtual terminal number (UNIX system)
width_status_line	`wsl`	`ws`	No. columns in status line

(The following numeric capabilities are present in the SYSV term structure, but are not yet documented in the man page. Comments are from the term structure header.)

bit_image_type	`bitype`	`Yp`	Type of bit-image device
buttons	`btns`	`BT`	Number of mouse buttons
max_attributes	`ma`	`ma`	Max combined attributes terminal can handle
maximum_windows	`wnum`	`MW`	Max number of definable windows
print_rate	`cps`	`Ym`	Print rate in chars per second
wide_char_size	`widcs`	`Yn`	Char step size in double wide mode

8.22.3 Strings

Variable	Cap. Name	Int. Code	Description
acs_chars	acsc	ac	Graphics charset pairs - def=vt100
alt_scancode_esc	scesa	S8	Alternate esc for scancode emulation (default is vt100)
back_tab	cbt	bt	Back tab (P)
bell	bel	bl	Audible signal (bell) (P)
bit_image_repeat	birep	Xy	Repeat bit image cell #1 #2 times (use tparm)
bit_image_newline	binel	Zz	Move to next row of the bit image (use tparm)
bit_image_carriage_return	bicr	Yv	Move to beginning of same row (use tparm)
carriage_return	cr	cr	Carriage return (P*)
change_char_pitch	cpi	ZA	Change # chars per inch
change_line_pitch	lpi	ZB	Change # lines per inch
change_res_horz	chr	ZC	Change horizontal resolution
change_res_vert	cvr	ZD	Change vertical resolution
change_scroll_region	csr	cs	Change to lines #1 through #2 (vt100) (PG)
char_padding	rmp	rP	Like ip but when in insert mode
char_set_names	csnm	Zy	List of character set names
clear_all_tabs	tbc	ct	Clear all tab stops (P)
clear_margins	mgc	MC	Clear all margins (top, bottom, and sides)
clear_screen	clear	cl	Clear screen and home cursor (P*)
clr_bol	el1	cb	Clear to beginning of line
clr_eol	el	ce	Clear to end of line (P)
clr_eos	ed	cd	Clear to end of display (P*)
code_set_init	csin	ci	Init sequence for multiple code sets
color_names	colornm	Yw	Give name for color #1
column_address	hpa	ch	Set cursor column (PG)
command_character	cmdch	CC	Term. settable cmd char in prototype
cursor_address	cup	cm	Screen rel. cursor motion row #1 col #2 (PG)
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no cup)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move cursor left one space
cursor_mem_address	mrcup	CM	Memory relative cursor addressing
cursor_normal	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll	ll	ll	Last line, first column (if no cup)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
define_bit_image_region	defbi	Yx	Define rectangular bit image region (use tparm)
define_char	defc	ZE	Define character in a character set
delete_character	dch1	dc	Delete character (P*)

delete_line	dl1	dl	Delete line (P*)
device_type	devt	dv	Indicate language/codeset support
dis_status_line	dsl	ds	Disable status line
display_pc_char	dispc	S1	Display PC character
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	enable alternate char set
end_bit_image_region	endbi	Yy	End bit image region (use tparm)
enter_alt_charset_mode	smacs	as	Start alternate character set (P)
enter_am_mode	smam	SA	turn on automatic margins
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use cup
enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode
enter_doublewide_mode	swidm	ZF	Enable double-wide mode
enter_draft_quality	sdrfq	ZG	Set draft-quality printing
enter_insert_mode	smir	im	Insert mode (enter);
enter_italics_mode	sitm	ZH	Enable italics mode
enter_leftward_mode	slm	ZI	Enable leftward carriage motion
enter_micro_mode	smicm	ZJ	Enable micro-motion capabilities
enter_near_letter_quality	snlq	ZK	Set NLQ printing
enter_normal_quality	snrmq	ZL	Set normal quality printing
enter_pc_charset_mode	smpch	S2	Enter PC character display mode
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_scancode_mode	smsc	S4	Enter PC scancode mode
enter_secure_mode	invis	mk	Turn on blank mode (chars invisible)
enter_shadow_mode	sshm	ZM	Enable shadow-mode printing
enter_standout_mode	smso	so	Begin stand out mode
enter_subscript_mode	ssubm	ZN	Enable subscript printing
enter_superscript_mode	ssupm	ZO	Enable superscript printing
enter_underline_mode	smul	us	Start underscore mode
enter_upward_mode	sum	ZP	Enable upward carriage motion
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking
erase_chars	ech	ec	Erase #1 characters (PG)
exit_alt_charset_mode	rmacs	ae	End alternate character set (P)
exit_am_mode	rmam	RA	Turn off automatic margins
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use cup
exit_delete_mode	rmdc	ed	End delete mode
exit_doublewide_mode	rwidm	ZQ	Disable doublewide printing
exit_insert_mode	rmir	ei	End insert mode
exit_italics_mode	ritm	ZR	Disable italic printing
exit_leftward_mode	rlm	ZS	Enable rightward (normal) carriage motion
exit_micro_mode	rmicm	ZT	Disable micro motion capabilities
exit_pc_charset_mode	rmpch	S3	Disable PC character display
exit_scancode_mode	rmsc	S5	Disable PC scancode mode
exit_shadow_mode	rshm	ZU	Disable shadow printing
exit_standout_mode	rmso	se	End stand out mode
exit_subscript_mode	rsubm	ZV	Disable subscript printing
exit_superscript_mode	rsupm	ZW	Disable superscript printing
exit_underline_mode	rmul	ue	End underscore mode

exit_upward_mode	`rum`	`ZX`	Enable downward (normal) carriage motion
exit_xon_mode	`rmxon`	`RX`	turn off xon/xoff handshaking
flash_screen	`flash`	`vb`	Visible bell (may not move cursor)
form_feed	`ff`	`ff`	Hardcopy terminal page eject (P*)
from_status_line	`fsl`	`fs`	Return from status line
init_1string	`is1`	`i1`	Terminal initialization string
init_2string	`is2`	`i2`	Terminal initialization string
init_3string	`is3`	`i3`	Terminal initialization string
init_file	`if`	`if`	Name of file containing is
init_prog	`iprog`	`iP`	Path name of program for init
initialize_color	`initc`	`Ic`	Initialize the definition of color
initialize_pair	`initp`	`Ip`	Initialize color-pair
insert_character	`ich1`	`ic`	Insert character (P)
insert_line	`il1`	`al`	Add new blank line (P*)
insert_padding	`ip`	`ip`	Insert pad after character inserted (p*)
key_a1	`ka1`	`K1`	Upper left of keypad
key_a3	`ka3`	`K3`	Upper right of keypad
key_b2	`kb2`	`K2`	Center of keypad
key_backspace	`kbs`	`kb`	Sent by backspace key
key_beg	`kbeg`	`1`	begin key
key_btab	`kcbt`	`kB`	back-tab key
key_c1	`kc1`	`K4`	Lower left of keypad
key_c3	`kc3`	`K5`	Lower right of keypad
key_cancel	`kcan`	`2`	cancel key
key_catab	`ktbc`	`ka`	Sent by clear-all-tabs key
key_clear	`kclr`	`kC`	Sent by clear screen or erase key
key_close	`kclo`	`3`	close key
key_command	`kcmd`	`4`	command key
key_copy	`kcpy`	`5`	copy key
key_create	`kcrt`	`6`	create key
key_ctab	`kctab`	`kt`	Sent by clear-tab key
key_dc	`kdch1`	`kD`	Sent by delete character key
key_dl	`kdl1`	`kL`	Sent by delete line key
key_down	`kcud1`	`kd`	Sent by terminal down arrow key
key_eic	`krmir`	`kM`	Sent by rmir or smir in insert mode
key_end	`kend`	`7`	end key
key_enter	`kent`	`8`	enter/send key
key_eol	`kel`	`kE`	Sent by clear-to-end-of-line key
key_eos	`ked`	`kS`	Sent by clear-to-end-of-screen key
key_exit	`kext`	`9`	exit key

key_f0	`kf0`	`k0`	F00 function key	key_f10	`kf10`	`k;`	F10 function key
key_f1	`kf1`	`k1`	F01 function key	key_f11	`kf11`	`F1`	F11 function key
key_f2	`kf2`	`k2`	F02 function key	key_f12	`kf12`	`F2`	F12 function key
key_f3	`kf3`	`k3`	F03 function key	key_f13	`kf13`	`F3`	F13 function key
key_f4	`kf4`	`k4`	F04 function key	key_f14	`kf14`	`F4`	F14 function key
key_f5	`kf5`	`k5`	F05 function key	key_f15	`kf15`	`F5`	F15 function key
key_f6	`kf6`	`k6`	F06 function key	key_f16	`kf16`	`F6`	F16 function key
key_f7	`kf7`	`k7`	F07 function key	key_f17	`kf17`	`F7`	F17 function key
key_f8	`kf8`	`k8`	F08 function key	key_f18	`kf18`	`F8`	F18 function key
key_f9	`kf9`	`k9`	F09 function key	key_f19	`kf19`	`F9`	F19 function key

key_f20	kf20	FA	F20 function key	key_f42	kf42	FW	F42 function key
key_f21	kf21	FB	F21 function key	key_f43	kf43	FX	F43 function key
key_f22	kf22	FC	F22 function key	key_f44	kf44	FY	F44 function key
key_f23	kf23	FD	F23 function key	key_f45	kf45	FZ	F45 function key
key_f24	kf24	FE	F24 function key	key_f46	kf46	Fa	F46 function key
key_f25	kf25	FF	F25 function key	key_f47	kf47	Fb	F47 function key
key_f26	kf26	FG	F26 function key	key_f48	kf48	Fc	F48 function key
key_f27	kf27	FH	F27 function key	key_f49	kf49	Fd	F49 function key
key_f28	kf28	FI	F28 function key	key_f50	kf50	Fe	F50 function key
key_f29	kf29	FJ	F29 function key	key_f51	kf51	Ff	F51 function key
key_f30	kf30	FK	F30 function key	key_f52	kf52	Fg	F52 function key
key_f31	kf31	FL	F31 function key	key_f53	kf53	Fh	F53 function key
key_f32	kf32	FM	F32 function key	key_f54	kf54	Fi	F54 function key
key_f33	kf33	FN	F33 function key	key_f55	kf55	Fj	F55 function key
key_f34	kf34	FO	F34 function key	key_f56	kf56	Fk	F56 function key
key_f35	kf35	FP	F35 function key	key_f57	kf57	Fl	F57 function key
key_f36	kf36	FQ	F36 function key	key_f58	kf58	Fm	F58 function key
key_f37	kf37	FR	F37 function key	key_f59	kf59	Fn	F59 function key
key_f38	kf38	FS	F38 function key	key_f60	kf60	Fo	F60 function key
key_f39	kf39	FT	F39 function key	key_f61	kf61	Fp	F61 function key
key_f40	kf40	FU	F40 function key	key_f62	kf62	Fq	F62 function key
key_f41	kf41	FV	F41 function key	key_f63	kf63	Fr	F63 function key

key_find	kfnd	0	find key
key_help	khlp	%1	help key
key_home	khome	kh	Sent by home key
key_ic	kich1	kI	Sent by ins char/enter ins mode key
key_il	kil1	kA	Sent by insert line
key_left	kcub1	kl	Sent by terminal left arrow key
key_ll	kll	kH	Sent by home-down key
key_mark	kmrk	%2	mark key
key_message	kmsg	%3	message key
key_move	kmov	%4	move key
key_next	knxt	%5	next key
key_npage	knp	kN	Sent by next-page key
key_open	kopn	%6	open key
key_options	kopt	%7	options key
key_ppage	kpp	kP	Sent by previous-page key
key_previous	kprv	%8	previous key
key_print	kprt	%9	print key
key_redo	krdo	%0	redo key
key_reference	kref	&1	reference key
key_refresh	krfr	&2	refresh key
key_replace	krpl	&3	replace key
key_restart	krst	&4	restart key
key_resume	kres	&5	resume key
key_right	kcuf1	kr	Sent by terminal right arrow key
key_save	ksav	&6	save key
key_sbeg	kBEG	&9	shifted begin key
key_scancel	kCAN	&0	shifted cancel key
key_scommand	kCMD	*1	shifted command key
key_scopy	kCPY	*2	shifted copy key
key_screate	kCRT	*3	shifted create key

key_sdc	kDC	*4	shifted delete char key
key_sdl	kDL	*5	shifted delete line key
key_select	kslt	*6	select key
key_send	kEND	*7	shifted end key
key_seol	kEOL	*8	shifted end of line key
key_sexit	kEXT	*9	shifted exit key
key_sf	kind	kF	Sent by scroll-forward/down key
key_sfind	kFND	*0	shifted find key
key_shelp	kHLP	#1	shifted help key
key_shome	kHOM	#2	shifted home key
key_sic	kIC	#3	shifted insert char key
key_sleft	kLFT	#4	shifted left key
key_smessage	kMSG	%a	shifted message key
key_smove	kMOV	%b	shifted move key
key_snext	kNXT	%c	shifted next key
key_soptions	kOPT	%d	shifted options key
key_sprevious	kPRV	%e	shifted previous key
key_sprint	kPRT	%f	shifted print key
key_sr	kri	kR	Sent by scroll-backward/up key
key_sredo	kRDO	%g	shifted redo key
key_sreplace	kRPL	%h	shifted replace key
key_sright	kRIT	%i	shifted right key
key_srsume	kRES	%j	shifted resume key
key_ssave	kSAV	!1	shifted save key
key_ssuspend	kSPD	!2	shifted suspend key
key_stab	khts	kT	Sent by set-tab key
key_sundo	kUND	!3	shifted undo key
key_suspend	kspd	&7	suspend key
key_undo	kund	&8	undo key
key_up	kcuu1	ku	Sent by terminal up arrow key
keypad_local	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit	smkx	ks	Put terminal in "keypad transmit" mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_on	smln	LO	turn on soft labels
label_off	rmln	LF	turn off soft labels
meta_off	rmm	mo	Turn off "meta mode"
meta_on	smm	mm	Turn on "meta mode" (8th bit)
micro_column_address	mhpa	ZY	Like column_address for micro adjustment
micro_down	mcud1	ZZ	Like cursor_down for micro adjustment
micro_left	mcub1	Za	Like cursor_left for micro adjustment
micro_right	mcuf1	Zb	Like cursor_right for micro adjustment
micro_row_address	mvpa	Zc	Like row_address for micro adjustment
micro_up	mcuu1	Zd	Like cursor_up for micro adjustment

newline	nel	nw	Newline (behaves like cr followed by lf)
order_of_pins	porder	Ze	Matches software buts to print-head pins
orig_colors	oc	oc	Reset all color pairs
orig_pair	op	op	Set default color-pair to original one
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars (PG*)
parm_delete_line	dl	DL	Delete #1 lines (PG*)
parm_down_cursor	cud	DO	Move cursor down #1 lines (PG*)
parm_down_micro	mcud	Zf	Like cud for micro adjust
parm_ich	ich	IC	Insert #1 blank chars (PG*)
parm_index	indn	SF	Scroll forward #1 lines (PG)
parm_insert_line	il	AL	Add #1 new blank lines (PG*)
parm_left_cursor	cub	LE	Move cursor left #1 spaces (PG)
parm_left_micro	mcub	Zg	Like cul for micro adjust
parm_right_cursor	cuf	RI	Move cursor right #1 spaces (PG*)
parm_right_micro	mcuf	Zh	Like cuf for micro adjust
parm_rindex	rin	SR	Scroll backward #1 lines (PG)
parm_up_cursor	cuu	UP	Move cursor up #1 lines (PG*)
parm_up_micro	mcuu	Zi	Like cuu for micro adjust
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
pkey_plab	pfxl	xl	Program key #1 to xmit #2 and show #3
plab_norm	pln	pn	program label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer
repeat_char	rep	rp	Repeat char #1 #2 times. (PG*)
req_for_input	rfi	RF	request for input
reset_1string	rs1	r1	Reset terminal completely to sane modes.
reset_2string	rs2	r2	Reset terminal completely to sane modes.
reset_3string	rs3	r3	Reset terminal completely to sane modes.
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last sc
row_address	vpa	cv	Vertical position absolute (set row) (PG)
save_cursor	sc	sc	Save cursor position (P)
scancode_escape	scesc	S7	Escape for scancode emulation
scroll_forward	ind	sf	Scroll text up (P)
scroll_reverse	ri	sr	Scroll text down (P)
select_char_set	scs	Zj	Select character set
set0_des_seq	s0ds	s0	Shift to codeset 0 (EUC set 0, ASCII)
set1_des_seq	s1ds	s1	Shift to codeset 1
set2_des_seq	s2ds	s2	Shift to codeset 2
set3_des_seq	s3ds	s3	Shift to codeset 3
set_a_background	setab	AB	Set background color using ANSI escape
set_a_foreground	setaf	AF	Set foreground color using ANSI escape
set_attributes	sgr	sa	Define the video attributes (PG9)
set_background	setb	Sb	Set current background color
set_bottom_margin	smgb	Zk	Set bottom margin at current line
set_bottom_margin_parm	smgbp	Zl	Set bottom line at line #1 or #2 lines from bottom
set_color_band	setcolor	Yz	Change to ribbon color #1

set_color_pair	`scp`	`sp`	Set current color pair
set_foreground	`setf`	`Sf`	Set current foreground color
set_left_margin	`smgl`	`ML`	Set left margin at current line
set_left_margin_parm	`smglp`	`Zm`	Set left (right) margin at #1 (#2)
set_lr_margin	`smglr`	`ML`	Set both left and right margins
set_page_length	`slines`	`YZ`	Set page length to #1 lines (use tparm)
set_right_margin	`smgr`	`MR`	Set right margin at current column
set_right_margin_parm	`smgrp`	`Zn`	Set right margin at column #1
set_tab	`hts`	`st`	Set a tab in all rows, current column
set_tb_margin	`smgtb`	`MT`	Sets both top and bottom margins
set_top_margin	`smgt`	`Zo`	Set top margin at current line
set_top_margin_parm	`smgtp`	`Zp`	Set top margin at line #1
set_window	`wind`	`wi`	Current window is lines #1-#2 cols #3-#4
start_bit_image	`sbim`	`Zq`	Start printing bit image graphics
start_char_set_def	`scsd`	`Zr`	Start definition of a character set
stop_bit_image	`rbim`	`Zs`	End printing bit image graphics
stop_char_set_def	`rcsd`	`Zt`	End definition of character set
subscript_characters	`subcs`	`Zu`	List of subscriptable chars
superscript_characters	`supcs`	`Zv`	List of superscriptable chars
tab	`ht`	`ta`	Tab to next 8 space hardware tab stop
these_cause_cr	`docr`	`Zw`	These characters cause a CR
to_status_line	`tsl`	`ts`	Go to status line, column #1
underline_char	`uc`	`uc`	Underscore one char and move past it
up_half_line	`hu`	`hu`	Half-line up (reverse 1/2 linefeed)
xoff_character	`xoffc`	`XF`	XON character
xon_character	`xonc`	`XN`	XOFF character

(The following string capabilities are present in the SYSVr term structure, but are not documented in the man page. Comments are from the term structure header.)

label_format	`fln`	`Lf`	??
set_clock	`sclk`	`SC`	Set time-of-day clock
display_clock	`dclk`	`DK`	Display time-of-day clock
remove_clock	`rmclk`	`RC`	Remove time-of-day clock??
create_window	`cwin`	`CW`	Define win #1 to go from #2,#3 to #4,#5
goto_window	`wingo`	`WG`	Goto window #1
hangup	`hup`	`HU`	Hang up phone
dial_phone	`dial`	`DI`	Dial phone number #1
quick_dial	`qdial`	`QD`	Dial phone number #1, without progress detection
tone	`tone`	`TO`	Select touch tone dialing
pulse	`pulse`	`PU`	Select pulse dialing
flash_hook	`hook`	`fh`	Flash the switch hook
fixed_pause	`pause`	`PA`	Pause for 2-3 seconds
wait_tone	`wait`	`WA`	Wait for dial tone
user0	`u0`	`u0`	User string # 0
user1	`u1`	`u1`	User string # 1
user2	`u2`	`u2`	User string # 2
user3	`u3`	`u3`	User string # 3
user4	`u4`	`u4`	User string # 4
user5	`u5`	`u5`	User string # 5
user6	`u6`	`u6`	User string # 6
user7	`u7`	`u7`	User string # 7

user8	`u8`	`u8`	User string # 8
user9	`u9`	`u9`	User string # 9
get_mouse	`getm`	`Gm`	Curses should get button events
key_mouse	`kmous`	`Km`	??
mouse_info	`minfo`	`Mi`	Mouse status information
pc_term_options	`pctrm`	`S6`	PC terminal options
req_mouse_pos	`reqmp`	`RQ`	Request mouse position report
zero_motion	`zerom`	`Zx`	No motion for the subsequent character

8.23 [N]Curses Function Overview

In the following text you will find an overview over the different (n)curses packages. In the first column is the bsd-curses (as it is in slackware 2.1.0 and in Sun-OS 4.x), in the second is the sysv-curses (in Sun-OS 5.4 / Solaris 2) and in the third is the ncurses (version 1.8.6).

In the fourth column is a reference to the page in the text where the function is described (if it is actually described).

x package has this function

n function not yet implemented

Function	BSD	SYSV	Nc.	Page
`_init_trace()`			x	118
`_traceattr(mode)`			x	118
`_tracef(char *, ...)`			x	118
`addbytes(...)`	x			
`addch(ch)`	x	x	x	96
`addchnstr(...)`		x	x	96
`addchstr(chstr)`		x	x	96
`addnstr(...)`		x	x	96
`addnwstr(...)`		x		
`addstr(str)`	x	x	x	96
`addwch(...)`		x		
`addwchnstr(...)`		x		
`addwchstr(...)`		x		
`addwstr(...)`		x		
`adjcurspos()`		x		
`attroff(attr)`		x	x	110
`attron(attr)`		x	x	110
`attrset(attr)`		x	x	110
`baudrate()`	x	x	x	103
`beep()`		x	x	115
`bkgd(ch)`		x	x	100
`bkgdset(ch)`		x	x	98
`border(...)`		x	x	98
`box(...)`	x	x	x	98
`can_change_color()`		x	x	110
`cbreak()`	x	x	x	102
`clear()`	x	x	x	107
`clearok(...)`	x	x	x	102
`clrtobot()`	x	x	x	107
`clrtoeol()`	x	x	x	107
`color_content(...)`		x	x	111
`copywin(...)`		x	x	96
`crmode()`	x	x	x	102
`curs_set(bf)`		x	x	112
`curserr()`		x		
`def_prog_mode()`		x	x	116
`def_shell_mode()`		x	x	116
`del_curterm(...)`		x	x	117
`delay_output(ms)`		x	x	115
`delch()`	x	x	x	98
`deleteln()`	x	x	x	98
`delscreen(...)`		x	x,n	93
`delwin(win)`	x	x	x	95
`derwin(...)`		x	x	95
`doupdate()`		x	x	107
`drainio(int)`		x		
`dupwin(win)`		x	x	95
`echo()`	x	x	x	103
`echochar(ch)`		x	x	96
`echowchar(ch)`		x		
`endwin()`	x	x	x	93
`erase()`	x	x	x	106
`erasechar()`	x	x	x	104
`filter()`		x	x	115
`flash()`		x	x	115
`flushinp()`		x	x	115
`flushok(...)`	x			
`garbagedlines(...)`		x		
`garbagedwin(win)`		x		
`getattrs(win)`		x	x	110
`getbegyx(...)`		x	x	112
`getbkgd(win)`		x		
`getbmap()`		x		
`getcap(str)`	x			
`getch()`	x	x	x	100
`getmaxx(win)`		x	x	112
`getmaxy(win)`		x	x	112
`getmaxyx(...)`		x	x	112
`getmouse()`		x		
`getnwstr(...)`		x		
`getparyx(...)`		x	x	112

getstr(str)	x	x	x	100
getsyx(...)		x	x	112
gettmode()	x	x		
getwch(...)		x		
getwin(...)		x		
getwin(FILE *)		x	x,n	115
getwstr(...)		x		
getyx(...)	x	x	x	112
halfdelay(t)		x	x	103
has_colors()		x	x	110
has_ic()		x	x,n	104
has_il()		x	x,n	104
hline(...)		x	x	98
idcok(...)		x	x,n	101
idlok(...)	x	x	x	101
immedok(...)		x	x	101
inch()	x	x	x	100
inchnstr(...)		x	x,n	101
inchstr(...)		x	x,n	101
init_color(...)		x	x	111
init_pair(...)		x	x	110
initscr()	x	x	x	92
innstr(...)		x	x,n	101
innwstr(...)		x		
insch(c)	x	x	x	97
insdelln(n)		x	x	97
insertln()	x	x	x	97
insnstr(...)		x	x	97
insstr(str)		x	x	97
instr(str)		x	x,n	101
inswch(...)		x		
inswstr(...)		x		
intrflush(...)		x	x	103
inwch(...)		x		
inwchnstr(...)		x		
inwchstr(...)		x		
inwchstr(...)		x		
inwstr(...)		x		
is_linetouched(...)		x	x	108
is_wintouched(win)		x	x	108
isendwin()		x	x	93
keyname(c)		x	x	115
keypad(...)		x	x	102
killchar()	x	x	x	104
leaveok(...)	x	x	x	102
longname()	x	x	x	104
map_button(long)		x		
meta(...)		x	x	102
mouse_off(long)		x		
mouse_on(long)		x		
mouse_set(long)		x		
move(...)	x	x	x	112
movenextch()		x		
moveprevch()		x		
mvaddbytes(...)	x			
mvaddch(...)	x	x	x	96
mvaddchnstr(...)		x	x	96
mvaddchstr(...)		x	x	96
mvaddnstr(...)		x	x	96
mvaddnwstr(...)		x		
mvaddstr(...)	x	x	x	96
mvaddwch(...)		x		
mvaddwchnstr(...)		x		
mvaddwchstr(...)		x		
mvaddwstr(...)		x		
mvcur(...)	x	x	x	117
mvdelch(...)	x	x	x	98
mvderwin(...)		x	x,n	95
mvgetch(...)	x	x	x	100
mvgetnwstr(...)		x		
mvgetstr(...)	x	x	x	100
mvgetwch(...)		x		
mvgetwstr(...)		x		
mvhline(...)		x		
mvinch(...)	x	x	x	100
mvinchnstr(...)		x	x,n	101
mvinchstr(...)		x	x,n	101
mvinnstr(...)		x	x,n	101
mvinnwstr(...)		x		
mvinsch(...)	x	x	x	97
mvinsnstr(...)		x	x	97
mvinsnwstr(...)		x		
mvinsstr(...)		x	x	97
mvinstr(...)		x	x,n	101
mvinswch(...)		x		
mvinswstr(...)		x		
mvinwch(...)		x		
mvinwchnstr(...)		x		
mvinwchstr(...)		x		
mvinwstr(...)		x		
mvprintw(...)	x	x	x	97
mvscanw(...)	x	x	x	101
mvvline(...)		x		
mvwaddbytes(...)	x			
mvwaddch(...)	x	x	x	96
mvwaddchnstr(...)		x	x	96
mvwaddchstr(...)		x	x	96
mvwaddnstr(...)		x	x	96
mvwaddnwstr(...)		x		
mvwaddstr(...)	x	x	x	96
mvwaddwch(...)		x		
mvwaddwchnstr(...)		x		
mvwaddwchstr(...)		x		
mvwaddwstr(...)		x		
mvwdelch(...)	x	x	x	98
mvwgetch(...)	x	x	x	100
mvwgetnwstr(...)		x		
mvwgetstr(...)	x	x	x	100
mvwgetwch(...)		x		
mvwgetwstr(...)		x		
mvwhline(...)		x		
mvwin(...)	x	x	x	95
mvwinch(...)	x	x	x	100
mvwinchnstr(...)		x	x,n	101
mvwinchstr(...)		x	x,n	101
mvwinnstr(...)		x	x,n	101
mvwinnwstr(...)		x		
mvwinsch(...)	x	x	x	97
mvwinsnstr(...)		x	x	97
mvwinsstr(...)		x	x	97
mvwinstr(...)		x	x,n	101
mvwinswch(...)		x		
mvwinswstr(...)		x		
mvwinwch(...)		x		
mvwinwchnstr(...)		x		
mvwinwchstr(...)		x		
mvwinwstr(...)		x		
mvwprintw(...)	x	x	x	97
mvwscanw(...)	x	x	x	101
mvwvline(...)		x		
napms(ms)		x	x	116
newkey(...)		x		

newpad(...)		x	x	114
newscreen(...)		x		
newterm(...)		x	x	92
newwin(...)	x	x	x	93
nl()	x	x	x	102
nocbreak()	x	x	x	102
nocrmode()	x	x	x	102
nodelay(...)		x	x	103
noecho()	x	x	x	103
nonl()	x	x	x	102
noqiflush()		x	x,n	103
noraw()	x	x	x	103
notimeout(...)		x	x	103
overlay(...)	x	x	x	95
overwrite(...)	x	x	x	95
pair_content(...)		x	x	111
pechochar(...)		x	x	114
pechowchar(...)		x		
pnoutrefresh(...)		x	x	114
prefresh(...)		x	x	114
printw(...)	x	x	x	97
putp(char *)		x	x	117
putwin(...)		x	x,n	115
qiflush()		x	x,n	103
raw()	x	x	x	103
redrawwin(win)		x	x	108
refresh()	x	x	x	107
request_mouse_pos()		x		
reset_prog_mode()		x	x	116
reset_shell_mode()		x	x	116
resetty()	x	x	x	116
restartterm(...)		x	x,n	117
ripoffline(...)		x	x	116
savetty()	x	x	x	116
scanw(...)	x	x	x	101
scr_dump(char *)		x	x,n	116
scr_init(char *)		x	x,n	116
scr_restore(char *)		x	x,n	116
scr_set(char *)		x	x,n	116
scrl(n)		x	x	113
scroll(win)	x	x	x	113
scrollok(...)	x	x	x	112
set_curterm(...)		x	x	117
set_term(...)		x	x	93
setcurscreen(SCREEN *)		x		
setscrreg(...)		x	x	113
setsyx(...)		x	x	112
setterm(char *)	x	x	x	117
setupterm(...)		x	x	117
slk_attroff(attr)		x	x,n	115
slk_attron(attr)		x	x,n	115
slk_attrset(attr)		x	x,n	115
slk_clear()		x	x	115
slk_init(fmt)		x	x	114
slk_label(labnum)		x	x	115
slk_noutrefresh()		x	x	114
slk_refresh()		x	x	114
slk_restore()		x	x	115
slk_set(...)		x	x	114
slk_touch()		x	x	115
standend()	x	x	x	110
standout()	x	x	x	110
start_color()		x	x	110
subpad(...)		x	x	114
subwin(...)	x	x	x	95
syncok(...)		x	x,n	95
termattrs()		x	x,n	104
termname()		x	x,n	104
tgetent(...)		x	x	116
tgetflag(char [2])		x	x	116
tgetnum(char [2])		x	x	116
tgetstr(...)		x	x	117
tgoto(...)		x	x	117
tigetflag(...)		x	x	117
tigetnum(...)		x	x	117
tigetstr(...)		x	x	117
timeout(t)		x	x	103
touchline(...)	x	x	x	108
touchwin(win)	x	x	x	108
tparm(...)		x	x	117
tputs(...)			x	117
traceoff()		x	x	118
traceon()		x	x	118
typeahead(fd)		x	x	103
unctrl(chtype c)		x	x	115
ungetch(ch)		x	x	100
ungetwch(c)		x		
untouchwin(win)		x	x	108
use_env(bf)		x	x	115
vidattr(...)		x	x	117
vidputs(...)		x	x	117
vidupdate(...)		x		
vline(...)		x	x	98
vwprintw(...)		x	x	97
vwscanw(...)		x	x	101
waddbytes(...)	x			
waddch(...)	x	x	x	96
waddchnstr(...)		x	x	96
waddchstr(...)		x	x	96
waddnstr(...)		x	x	96
waddnwstr(...)		x		
waddstr(...)	x	x	x	96
waddwch(...)		x		
waddwchnstr(...)		x		
waddwchstr(...)		x		
waddwstr(...)		x		
wadjcurspos(win)		x		
wattroff(...)		x	x	110
wattron(...)		x	x	110
wattrset(...)		x	x	110
wbkgd(...)		x	x	100
wbkgdset(...)		x	x	98
wborder(...)		x	x	98
wclear(win)	x	x	x	107
wclrtobot(win)	x	x	x	107
wclrtoeol(win)	x	x	x	107
wcursyncup(win)		x	x,n	95
wdelch(win)	x	x	x	98
wdeleteln(win)	x	x	x	98
wechochar(...)		x	x	96
wechowchar(...)		x		
werase(win)	x	x	x	106
wgetch(win)	x	x	x	100
wgetnstr(...)		x	x	100
wgetnwstr(...)		x		
wgetstr(...)	x	x	x	100
wgetwch(...)		x		
wgetwstr(...)		x		
whline()		x		
whline(...)		x		
whline(...)		x	x	98
winch(win)	x	x	x	100

winchnstr(...)		x	x,n	101
winchstr(...)		x	x,n	101
winnstr(...)		x	x,n	101
winnwstr(...)		x		
winsch(...)	x	x	x	97
winsdelln(...)	x	x	x	97
winsertln(win)		x	x	97
winsnstr(...)		x	x	97
winsnwstr(...)		x		
winsstr(...)		x	x	97
winstr(...)		x	x,n	101
winswch(...)		x		
winswstr(...)		x		
winwch(...)		x		
winwchnstr(...)		x		
winwchstr(...)		x		
winwstr(...)		x		
wmouse_position(...)		x		
wmove(...)	x	x	x	112
wmovenextch(win)		x		
wmoveprevch(win)		x		
wnoutrefresh(win)		x	x	107
wprintw(...)	x	x	x	97
wredrawln(...)		x	x	108
wrefresh(win)	x	x	x	107
wscanw(...)	x	x	x	101
wscrl(...)		x	x	113
wsetscrreg(...)		x	x	113
wstandend(win)	x	x	x	110
wstandout(win)	x	x	x	110
wsyncdown(win)		x	x,n	95
wsyncup(win)		x	x,n	95
wtimeout(...)		x	x	103
wtouchln(...)		x	x	108
wvline()		x		
wvline(...)		x		
wvline(...)		x	x	98

To be continued...

Chapter 9

Programming I/O ports

Usually a PC at least has 2 serial and 1 parallel interfaces. These interfaces are special devices and are mapped as follows:

- $/dev/ttyS0 - /dev/ttyS\mathbf{n}$
  these are the RS232 serial devices 0-**n** where **n** depends on your hardware.

- $/dev/cua0 - /dev/cua\mathbf{n}$
  these are the RS232 serial devices 0-**n** where **n** depends on your hardware.

- $/dev/lp0 - /dev/lp\mathbf{n}$
  these are the parallel devices 0-**n** where **n** depends on your hardware.

- $/dev/js0 - /dev/js\mathbf{n}$
  these are the joystick devices 0-**n** where 0 <= **n** <= 1.

The difference between the $/dev/ttyS*$ and $/dev/cua*$ devices is how a call to open() is handled. The $/dev/cua*$ devices are supposed to be used as call-out devices and thus get other default settings by a call to open() than the $/dev/ttyS*$ devices which will be initalized for incoming and outgoing calls. By default the devices are controlling devices for the process that opened them. Normally ioctl() requests should handle all these special devices, but POSIX preferred to define new functions to handle asynchronous terminals heavily depending on the struct termios. Both methods require including $< termios.h >$.

1. method ioctl:
   TCSBRK, TCSBRKP, TCGETA (get attributes), TCSETA (set attributes)
   Terminal I/O control (TIOC) requests:
   TIOCGSOFTCAR (set soft carrier), TIOCSSOFTCAR (get soft carrier), TIOCSCTTY (set controlling tty), TIOCMGET (get modemlines), TIOCMSET (set modemlines), TIOCGSERIAL, TIOCSSERIAL, TIOCSERCONFIG, TIOCSERGWILD, TIOCSERSWILD, TIOCSERGSTRUCT, TIOCMBIS, TIOCMBIC, ...

2. method POSIX:
   tcgetattr(), tcsetattr(), tcsendbreak(), tcdrain(), tcflush(), tcflow(), tcgetpgrp(), tcsetpgrp()
   cfsetispeed(), cfgetispeed(), cfsetospeed(), cfgetospeed()

3. other methods:
   outb,inb for hardware near programming like using the printer port not for a printer.

9.1 Mouse Programming

A mouse is either connected to a serial port or directly to the AT bus and different types of mouse send distinct kinds of data, which makes mouse programming a bit harder. But, Andrew Haylett was so kind as to put a generous copyright on his selection program which means you can use his mouse routines for your own programs. Included in this guide you can find the pre-release of selection-1.8 with the COPYRIGHT notice. X11 already offers a comfortable mouse API, so Andrew's routines should be used for non-X11 applications only. You only need the modules mouse.c and mouse.h from the selection package. To get mouse events you basically have to call ms_init() and get_ms_event(). ms_init needs the following 10 arguments:

1. *int acceleration*
   is the acceleration factor. If you move the mouse more than *delta* pixels, motion becomes faster depending on this value.

2. *int baud*
   is the bps rate your mouse uses (normally 1200).

3. *int delta*
   this is the number of pixels that you have to move the mouse before the acceleration starts.

4. *char *device*
   is the name of your mouse device (e.g. /dev/mouse).

5. *int toggle*
   toggle the DTR, RTS or both DTR and RTS mouse modem lines on initialization (normally 0).

6. *int sample*
   the resolution (dpi) of your mouse (normally 100).

7. *mouse_type mouse*
   the identifier of the connected mouse like P_MSC (Mouse Systems Corp.) for my mouse ;).

8. *int slack*
   amount of slack for wraparound which means if slack is -1 a try to move the mouse over the screen border will leave the mouse at the border. Values >= 0 mean that the mouse cursor will wrap to the other end after moving the mouse *slack* pixels against the border.

9. *int maxx*
   the resolution of your current terminal in x direction. With the default font, a char is 10 pixels wide, and therefore the overall x screen resolution is 10*80-1.

10. *int maxy*
    the resolution of your current terminal in y direction. With the default font, a char is 12 pixels high and therefore the overall y screen resolution is 12*25-1.

get_ms_event() just needs a pointer to a struct ms_event. If get_ms_event() returns -1, an error occured. On success, it returns 0, and the struct ms_event will contain the actual mouse state.

9.2 Modem Programming

See example miniterm.c
Use termios to control rs232 port.
Use Hayes Commands to control modem.

9.3 Printer Programming

See example checklp.c
Don't use termios to control printer port. Use ioctl and inb/outb if necessary.
Use Epson, Postscript, PCL, etc. commands to control printer.
$< linux/lp.h >$
ioctl calls: LPCHAR, LPTIME, LPABORT, LPSETIRQ, LPGETIRQ, LPWAIT
inb/outb for status and control port.

9.4 Joystick Programming

See example js.c in the joystick loadable kernel module package.
$< linux/joystick.h >$
ioctl
calls: JS_SET_CAL, JS_GET_CAL, JS_SET_TIMEOUT, JS_GET_TIMEOUT, JS_SET_TIMELIMIT, JS_GET_TIMELIMIT, JS_GET_ALL, JS_SET_ALL. A read operation on /dev/js**n** will return the struct JS_DATA_TYPE.

Chapter 10

Porting Applications to Linux

10.1 Introduction

Porting UNIX applications to the Linux operating system is remarkably easy. Linux, and the GNU C library used by it, have been designed with applications portability in mind, meaning that many applications will compile simply by issuing `make`. Those which don't generally use some obscure feature of a particular implementation, or rely strongly on undocumented or undefined behavior of, say, a particular system call.

Linux is mostly compliant with IEEE Std 1003.1-1988 (POSIX.1), but has not actually been certified as such. Similarly, Linux also implements many features found in the SVID and BSD strains of UNIX, but again does not necessarily adhere to them in all cases. In general, Linux has been designed to be compatible with other UNIX implementations, to make applications porting easier, and in a number of instances has improved upon or corrected behavior found in those implementations.

As an example, the *timeout* argument passed to the *select* system call is actually decremented during the poll operation by Linux. Other implementations don't modify this value at all, and applications which aren't expecting this could break when compiled under Linux. The BSD and SunOS man pages for *select* warn that in a "future implementation", the system call may modify the timeout pointer. Unfortunately, many applications still assume that the value will be untouched.

The goal of this paper is to provide an overview of the major issues associated with porting applications to Linux, highlighting the differences between Linux, POSIX.1, SVID, and BSD in the following areas: signal handling, terminal I/O, process control and information gathering, and portable conditional compilation.

10.2 Signal handling

Over the years, the definition and semantics of signals have been modified in various ways by different implementations of UNIX. Today, there are two major classes of symbols: *unreliable* and *reliable*. Unreliable signals are those for which the signal handler does not remain installed once called. These "one-shot" signals must re-install the signal handler within the signal handler itself, if the program wishes the signal to remain installed. Because of this, there is a race condition in which the signal can arrive again before the handler is re-installed, which can cause the signal to either be lost or for the original behavior of the signal to be triggered (such as killing the process). Therefore, these signals are "unreliable" because the signal catching and handler re-installation operations are nonatomic.

Under unreliable signal semantics, system calls are not restarted automatically when interrupted by a signal. Therefore, in order for a program to account for all cases, the program would need to check the value of *errno* after every system call, and reissue the system call if its value is *EINTR*.

Along similar lines, unreliable signal semantics don't provide an easy way to get an atomic pause operation (put the process to sleep until a signal arrives). Because of the unreliable nature of reinstalling signal handlers, there are cases in which a signal can arrive without the program realizing this.

Under reliable signal semantics, on the other hand, the signal handler remains installed when called, and the race condition for reinstallation is avoided. Also, certain system calls can be restarted, and an atomic pause operation is available via the POSIX *sigsuspend* function.

10.2.1 Signals under SVR4, BSD, and POSIX.1

The SVR4 implementation of signals incorporates the functions *signal*, *sigset*, *sighold*, *sigrelse*, *sigignore*, and *sigpause*. The *signal* function under SVR4 is identical to the classic UNIX V7 signals, providing only unreliable signals. The other functions do provide signals with automatic reinstallation of the signal handler, but no system call restarting is supported.

Under BSD, the functions *signal*, *sigvec*, *sigblock*, *sigsetmask*, and *sigpause* are supported. All of the functions provide reliable signals with system call restarting by default, but that behavior can be disabled if the programmer wishes.

Under POSIX.1, *sigaction*, *sigprocmask*, *sigpending*, and *sigsuspend* are provided. Note that there is no *signal* function, and according to POSIX.1 it is depreciated. These functions provide reliable signals, but system call restart behavior is not defined by POSIX. If *sigaction* is used under SVR4 and BSD, system call restarting is disabled by default, but it can be turned on if the signal flag `SA_RESTART` is specified.

Therefore, the "best" way to use signals in a program is to use *sigaction*, which allows you to explicitly specify the behavior of signal handlers. However, *signal* is still used in many applications, and as we can see above *signal* provides different semantics under SVR4 and BSD.

10.2.2 Linux signal options

The following values for the `sa_flags` member of the `sigaction` structure are defined for Linux.

- `SA_NOCLDSTOP`: Don't send `SIGCHLD` when a child process is stopped.
- `SA_RESTART`: Force restart of certain system calls when interrupted by a signal handler.
- `SA_NOMASK`: Disable signal mask (which blocks signals during execution of a signal handler).
- `SA_ONESHOT`: Clear signal handler after execution. Note that SVR4 uses `SA_RESETHAND` to mean the same thing.
- `SA_INTERRUPT`: Defined under Linux, but unused. Under SunOS, system calls were automatically restarted, and this flag disabled that behavior.
- `SA_STACK`: Currently a no-op, to be used for signal stacks.

Note that POSIX.1 defines only `SA_NOCLDSTOP`, and there are several other options defined by SVR4 not available under Linux. When porting applications which use *sigaction*, you may have to modify the values of `sa_flags` to get the appropriate behavior.

10.2.3 *signal* under Linux

Under Linux, the *signal* function is equivalent to using *sigaction* with the `SA_ONESHOT` and `SA_NOMASK` options; that is, it corresponds to the classic, unreliable signal semantics as used under SVR4.

If you wish *signal* to use BSD semantics, most Linux systems provide a BSD compatibility library which can be linked with. To use this library, you could add the options

```
-I/usr/include/bsd -lbsd
```

to the compilation command line. When porting applications using *signal*, pay close attention to what assumptions the program makes about use of signal handlers, and modify the code (or compile with the appropriate definitions) to get the right behavior.

10.2.4 Signals supported by Linux

Linux supports nearly every signal provided by SVR4, BSD, and POSIX, with few exceptions:

- `SIGEMT` is not supported; it corresponds to a hardware fault under SVR4 and BSD.
- `SIGINFO` is not supported; it is used for keyboard information requests under SVR4.
- `SIGSYS` is not supported; it refers to an invalid system call in SVR4 and BSD. If you link with `libbsd`, this signal is redefined to `SIGUNUSED`.
- `SIGABRT` and `SIGIOT` are identical.

- SIGIO, SIGPOLL, and SIGURG are identical.
- SIGBUS is defined as SIGUNUSED. Technically there is no "bus error" in the Linux environment.

10.3 Terminal I/O

As with signals, terminal I/O control has three different implementations under SVR4, BSD, and POSIX.1.

SVR4 uses the termio structure, and various *ioctl* calls (such as TCSETA, TCGETA, and so forth) on a terminal device to obtain and set parameters with the termio structure. This structure looks like:

```
struct termio {
  unsigned short c_iflag;  /* Input modes */
  unsigned short c_oflag;  /* Output modes */
  unsigned short c_cflag;  /* Control modes */
  unsigned short c_lflag;  /* Line discipline modes */
  char c_line;             /* Line discipline */
  unsigned char c_cc[NCC]; /* Control characters */
};
```

Under BSD, the sgtty structure is used with various *ioctl* calls, such as TIOCGETP, TIOCSETP, and so forth.

Under POSIX, the termios struct is used, along with various functions defined by POSIX.1, such as *tcsetattr* and tcgetattr. The termios structure is identical to struct termio used by SVR4, but the types are renamed (such as tcflag_t instead of unsigned short), and NCCS is used for the size of the c_cc array.

Under Linux, both POSIX.1 *termios* and SVR4 *termio* are supported directly by the kernel. This means that if your program uses either of these methods for accessing terminal I/O, it should compile directly under Linux. If you're ever in doubt, it's easy to modify code using termio to use termios, using a bit of knowledge of both methods. Hopefully, this shouldn't ever be necessary. But, do pay attention if a program attempts to use the c_line field in the termio structure. For nearly all applications, this should be N_TTY, and if the program assumes that some other line discipline is available you might have trouble.

If your program uses the BSD *sgtty* implementation, you can link against libbsd.a as described above. This will provide a replacement for *ioctl* which will resubmit the terminal I/O requests in terms of the POSIX termios calls used by the kernel. When compiling such a program, if symbols such as TIOCGETP are undefined, you will need to link against libbsd.

10.4 Process information and control

Programs such as *ps*, *top*, and free must have some way to obtain information from the kernel about a processes and system resources. Similarly, debuggers and other like tools need the ability to control and inspect a running process. These features have been provided by a number of interfaces by different versions of UNIX, and nearly all of them are either machine-specific or tied to a particular kernel design. So far, there has been no universally-accepted interface for this kind of process-kernel interaction.

10.4.1 *kvm* routines

Many systems use routines such as *kvm_open*, *kvm_nlist*, and *kvm_read* to access kernel data structures directly via the */dev/kmem* device. In general, these programs will open */dev/kmem*, read the kernel's symbol table, locate data in the running kernel with this table, and read the appropriate addresses in the kernel address space with these routines. Because this requires the user program and the kernel to agree upon the size and format of data structures read in this fashion, such programs often have to be rebuilt for each kernel revision, CPU type, and so forth.

10.4.2 *ptrace* and the */proc* filesystem

The *ptrace* system call is used in 4.3BSD and SVID to control a process and read information from it. It is classically used by debuggers to, say, trap execution of a running process or examine its state. Under SVR4, *ptrace* is superseded by the */proc* filesystem, which appears as a directory containing a single file entry for each running process, named by process ID. The user program can open the file corresponding to the process of interest and issue various *ioctl* calls on it to control its execution or obtain information from the kernel on the process. Similarly, the program can read or write data directly in the process's address space through the file descriptor into the */proc* filesystem.

10.4.3 Process control under Linux

Under Linux, the *ptrace* system call is supported for process control, and it works as in 4.3BSD. To obtain process and system information, Linux also provides a */proc* filesystem, but with very different semantics. Under Linux, */proc* consists of a number of files providing general system information, such as memory usage, load average, loaded module statistics, and network statistics. These files are generally accessed using *read* and *write* and their contents can be parsed using *scanf*. The */proc* filesystem under Linux also provides a directory entry for each running process, named by process ID, which contains file entries for information such as the command line, links to the current working directory and executable file, open file descriptors, and so forth. The kernel provides all of this information on the fly in response to *read* requests. This implementation is not unlike the */proc* filesystem found in Plan 9, but it does have its drawbacks—for example, for a tool such as *ps* to list a table of information on all running processes, many directories must be traversed and many files opened and read. By comparison, the *kvm* routines used on other UNIX systems read kernel data structures directly with only a few system calls.

Obviously, each implementation is so vastly different that porting applications which use them can prove to be a real task. It should be pointed out that the SVR4 */proc* filesystem is a very different beast than that found in Linux, and they may not be used in the same context. Arguably, any program which uses the *kvm* routines or SVR4 */proc* filesystem is not really portable, and those sections of code should be rewritten for each operating system.

The Linux *ptrace* call is nearly identical to that found in BSD, but there are a few differences:

- The requests `PTRACE_PEEKUSER` and `PTRACE_POKEUSER` under BSD are named `PTRACE_PEEKUSR` and `PTRACE_POKEUSR`, respectively, under Linux.
- Process registers can be set using the `PTRACE_POKEUSR` request with offsets found in `/usr/include/linux/ptrace.h`.
- The SunOS requests `PTRACE_{READ,WRITE}{TEXT,DATA}` are not supported, nor are `PTRACE_SETACBKPT`, `PTRACE_SETWRBKPT`, `PTRACE_CLRBKPT`, or `PTRACE_DUMPCORE`. These missing requests should only affect a small number of existing programs.

Linux does *not* provide the *kvm* routines for reading the kernel address space from a user program, but some programs (most notably *kmem_ps*) implement their own versions of these routines. In general, these are not portable, and any code which uses the *kvm* routines is probably depending upon the availability of certain symbols or data structures in the kernel—not a safe assumption to make. Use of *kvm* routines should be considered architecture-specific.

10.5 Portable conditional compilation

If you need to make modifications to existing code in order to port it to Linux, you may need to use `ifdef`...`endif` pairs to surround parts of Linux-specific code—or, for that matter, code corresponding to other implementations. No real standard for selecting portions of code to be compiled based on the operating system exists, but many programs use a convention such as defining `SVR4` for System V code, `BSD` for BSD code, and `linux` for Linux-specific code.

The GNU C library used by Linux allows you to turn on various features of the library by defining various macros at compile time. These are:

- `__STRICT_ANSI__`: For ANSI C features only
- `_POSIX_SOURCE`: For POSIX.1 features
- `_POSIX_C_SOURCE`: If defined as 1, POSIX.1 features; if defined as 2, POSIX.2 features.
- `_BSD_SOURCE`: ANSI, POSIX, and BSD features.
- `_SVID_SOURCE`: ANSI, POSIX, and System V features.
- `_GNU_SOURCE`: ANSI, POSIX, BSD, SVID, and GNU extensions. This is the default if none of the above are defined.

If you define `_BSD_SOURCE` yourself, the additional definition `_FAVOR_BSD` will be defined for the library. This will cause BSD behavior for certain things to be selected over POSIX or SVR4. For example, if `_FAVOR_BSD` is defined, *setjmp* and *longjmp* will save and restore the signal mask, and *getpgrp* will accept a PID argument. Note that you must still link against `libbsd` to get BSD-like behavior for the features mentioned earlier in this paper.

Under Linux, `gcc` defines a number of macros automatically which you can use in your program. These are:

- `__GNUC__` (major GNU C version, e.g., 2)
- `__GNUC_MINOR__` (minor GNU C version, e.g., 5)
- `unix`
- `i386`
- `linux`
- `__unix__`
- `__i386__`
- `__linux__`
- `__unix`
- `__i386`
- `__linux`

Many programs use

```
#ifdef linux
```

to surround Linux-specific code. Using these compile-time macros you can easily adapt existing code to include or exclude changes necessary to port the program to Linux, Note that because Linux supports more System V-like features in general, the best code base to start from with a program written for both System V and BSD is probably the System V version. Alternately, you can start from the BSD base and link against `libbsd`.

10.6 Additional Comments

[1] This chapter covers most of the porting issues except the missing system calls that are named in the system calls chapter and the yet missing streams (rumors say a loadable stream module should exist at ftp.uni-stuttgart.de in pub/systems/linux/isdn).

[1] Added by Sven Goldt

Chapter 11

Systemcalls in alphabetical order

_exit	-	like exit but with fewer actions (m+c)
accept	-	accept a connection on a socket (m+c!)
access	-	check user's permissions for a file (m+c)
acct	-	not yet implemented (mc)
adjtimex	-	set/get kernel time variables (-c)
afs_syscall	-	reserved andrew filesystem call (-)
alarm	-	send SIGALRM at a specified time (m+c)
bdflush	-	flush dirty buffers to disk (-c)
bind	-	name a socket for interprocess communication (m!c)
break	-	not yet implemented (-)
brk	-	change data segment size (mc)
chdir	-	change working directory (m+c)
chmod	-	change file attributes (m+c)
chown	-	change ownership of a file (m+c)
chroot	-	set a new root directory (mc)
clone	-	see fork (m-)
close	-	close a file by reference (m+c)
connect	-	link 2 sockets (m!c)
creat	-	create a file (m+c)
create_module	-	allocate space for a loadable kernel module (-)
delete_module	-	unload a kernel module (-)
dup	-	create a file descriptor duplicate (m+c)
dup2	-	duplicate a file descriptor (m+c)
execl, execlp, execle, ...	-	see execve (m+!c)
execve	-	execute a file (m+c)
exit	-	terminate a program (m+c)
fchdir	-	change working directory by reference ()
fchmod	-	see chmod (mc)
fchown	-	change ownership of a file (mc)
fclose	-	close a file by reference (m+!c)
fcntl	-	file/filedescriptor control (m+c)

flock	-	change file locking (m!c)
fork	-	create a child process (m+c)
fpathconf	-	get info about a file by reference (m+!c)
fread	-	read array of binary data from stream (m+!c)
fstat	-	get file status (m+c)
fstatfs	-	get filesystem status by reference (mc)
fsync	-	write file cache to disk (mc)
ftime	-	get timezone+seconds since 1.1.1970 (m!c)
ftruncate	-	change file size (mc)
fwrite	-	write array of binary datas to stream (m+!c)
get_kernel_syms	-	get kernel symbol table or its size (-)
getdomainname	-	get system's domainname (m!c)
getdtablesize	-	get filedescriptor table size (m!c)
getegid	-	get effective group id (m+c)
geteuid	-	get effective user id (m+c)
getgid	-	get real group id (m+c)
getgroups	-	get supplemental groups (m+c)
gethostid	-	get unique host identifier (m!c)
gethostname	-	get system's hostname (m!c)
getitimer	-	get value of interval timer (mc)
getpagesize	-	get size of a system page (m-!c)
getpeername	-	get address of a connected peer socket (m!c)
getpgid	-	get parent group id of a process (+c)
getpgrp	-	get parent group id of current process (m+c)
getpid	-	get process id of current process (m+c)
getppid	-	get process id of the parent process (m+c)
getpriority	-	get a process/group/user priority (mc)
getrlimit	-	get resource limits (mc)
getrusage	-	get usage of resources (m)
getsockname	-	get the adress of a socket (m!c)
getsockopt	-	get option settings of a socket (m!c)
gettimeofday	-	get timezone+seconds since 1.1.1970 (mc)
getuid	-	get real uid (m+c)
gtty	-	not yet implemented ()
idle	-	make a process a candidate for swap (mc)
init_module	-	insert a loadable kernel module (-)
ioctl	-	manipulate a character device (mc)
ioperm	-	set some i/o port's permissions (m-c)
iopl	-	set all i/o port's permissions (m-c)
ipc	-	interprocess communication (-c)
kill	-	send a signal to a process (m+c)
killpg	-	send a signal to a process group (mc!)
klog	-	see syslog (-!)
link	-	create a hardlink for an existing file (m+c)
listen	-	listen for socket connections (m!c)
llseek	-	lseek for large files (-)

lock	-	not implemented yet ()
lseek	-	change the position ptr of a file descriptor (m+c)
lstat	-	get file status (mc)
mkdir	-	create a directory (m+c)
mknod	-	create a device (mc)
mmap	-	map a file into memory (mc)
modify_ldt	-	read or write local descriptor table (-)
mount	-	mount a filesystem (mc)
mprotect	-	read, write or execute protect memory (-)
mpx	-	not implemented yet ()
msgctl	-	ipc message control (m!c)
msgget	-	get an ipc message queue id (m!c)
msgrcv	-	receive an ipc message (m!c)
msgsnd	-	send an ipc message (m!c)
munmap	-	unmap a file from memory (mc)
nice	-	change process priority (mc)
oldfstat	-	no longer existing
oldlstat	-	no longer existing
oldolduname	-	no longer existing
oldstat	-	no longer existing
olduname	-	no longer existing
open	-	open a file (m+c)
pathconf	-	get information about a file (m+!c)
pause	-	sleep until signal (m+c)
personality	-	change current execution domain for ibcs (-)
phys	-	not implemented yet (m)
pipe	-	create a pipe (m+c)
prof	-	not yet implemented ()
profil	-	execution time profile (m!c)
ptrace	-	trace a child process (mc)
quotactl	-	not implemented yet ()
read	-	read data from a file (m+c)
readv	-	read datablocks from a file (m!c)
readdir	-	read a directory (m+c)
readlink	-	get content of a symbolic link (mc)
reboot	-	reboot or toggle vulcan death grip (-mc)
recv	-	receive a message from a connected socket (m!c)
recvfrom	-	receive a message from a socket (m!c)
rename	-	move/rename a file (m+c)
rmdir	-	delete an empty directory (m+c)
sbrk	-	see brk (mc!)
select	-	sleep until action on a filedescriptor (mc)
semctl	-	ipc semaphore control (m!c)
semget	-	ipc get a semaphore set identifier (m!c)
semop	-	ipc operation on semaphore set members (m!c)
send	-	send a message to a connected socket (m!c)

sendto	-	send a message to a socket (m!c)
setdomainname	-	set system's domainname (mc)
setfsgid	-	set filesystem group id ()
setfsuid	-	set filesystem user id ()
setgid	-	set real group id (m+c)
setgroups	-	set supplemental groups (mc)
sethostid	-	set unique host identifier (mc)
sethostname	-	set the system's hostname (mc)
setitimer	-	set interval timer (mc)
setpgid	-	set process group id (m+c)
setpgrp	-	has no effect (mc!)
setpriority	-	set a process/group/user priority (mc)
setregid	-	set real and effective group id (mc)
setreuid	-	set real and effective user id (mc)
setrlimit	-	set resource limit (mc)
setsid	-	create a session (+c)
setsockopt	-	change options of a socket (mc)
settimeofday	-	set timezone+seconds since 1.1.1970 (mc)
setuid	-	set real user id (m+c)
setup	-	initialize devices and mount root (-)
sgetmask	-	see siggetmask (m)
shmat	-	attach shared memory to data segment (m!c)
shmctl	-	ipc manipulate shared memory (m!c)
shmdt	-	detach shared memory from data segment (m!c)
shmget	-	get/create shared memory segment (m!c)
shutdown	-	shutdown a socket (m!c)
sigaction	-	set/get signal handler (m+c)
sigblock	-	block signals (m!c)
siggetmask	-	get signal blocking of current process (!c)
signal	-	setup a signal handler (mc)
sigpause	-	use a new signal mask until a signal (mc)
sigpending	-	get pending, but blocked signals (m+c)
sigprocmask	-	set/get signal blocking of current process (+c)
sigreturn	-	not yet used ()
sigsetmask	-	set signal blocking of current process (c!)
sigsuspend	-	replacement for sigpause (m+c)
sigvec	-	see sigaction (m!)
socket	-	create a socket communication endpoint (m!c)
socketcall	-	socket call multiplexer (-)
socketpair	-	create 2 connected sockets (m!c)
ssetmask	-	see sigsetmask (m)
stat	-	get file status (m+c)
statfs	-	get filesystem status (mc)
stime	-	set seconds since 1.1.1970 (mc)
stty	-	not yet implemented ()
swapoff	-	stop swapping to a file/device (m-c)

swapon	-	start swapping to a file/device (m-c)
symlink	-	create a symbolic link to a file (m+c)
sync	-	sync memory and disk buffers (mc)
syscall	-	execute a systemcall by number (-!c)
sysconf	-	get value of a system variable (m+!c)
sysfs	-	get infos about configured filesystems ()
sysinfo	-	get Linux system infos (m-)
syslog	-	manipulate system logging (m-c)
system	-	execute a shell command (m!c)
time	-	get seconds since 1.1.1970 (m+c)
times	-	get process times (m+c)
truncate	-	change file size (mc)
ulimit	-	get/set file limits (c!)
umask	-	set file creation mask (m+c)
umount	-	unmount a filesystem (mc)
uname	-	get system information (m+c)
unlink	-	remove a file when not busy (m+c)
uselib	-	use a shared library (m-c)
ustat	-	not yet implemented (c)
utime	-	modify inode time entries (m+c)
utimes	-	see utime (m!c)
vfork	-	see fork (m!c)
vhangup	-	virtually hang up current tty (m-c)
vm86	-	enter virtual 8086 mode (m-c)
wait	-	wait for process termination (m+!c)
wait3	-	bsd wait for a specified process (m!c)
wait4	-	bsd wait for a specified process (mc)
waitpid	-	wait for a specified process (m+c)
write	-	write data to a file (m+c)
writev	-	write datablocks to a file (m!c)

(m) manual page exists.
(+) POSIX compliant.
(-) Linux specific.
(c) in libc.
(!) not a sole system call.uses a different system call.

Chapter 12

Abbreviations

ANSI	American National Standard for Information Systems
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AT 386	Advanced Technology Intel 80386 based PC
FIPS	Federal Information Processing Standard
FSF	Free Software Foundation
IEEE	Institute of Electrical and Electronics Engineers, Inc.
IPC	Inter Process Communication
ISO	International Organization for Standards
POSIX	Portable Operating System Interface for uniX
POSIX.1	IEEE Std. 1003.1-1990 Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API)