Linux Kernel Notes

Pramode C.E

Gopakumar C.E

Linux Kernel Notes
by Pramode C.E and Gopakumar C.E

This document has grown out of random experiments conducted by the authors to understand the working of parts of the Linux Operating System Kernel. It may be used as part of an Operating Systems course to give students a feel of the way a real OS works.

Table of Contents

Chapter 1. Philosophy

It is difficult to talk about Linux without first understanding the 'Unix Philosophy'. Unix was designed to be an environment which is pleasant to the programmer. Linux, its GUI trappings not withstanding, is a 'Unix' at heart, and embraces its philosophy just like all other Unices. The Linux programming environment is replete with myriads of tools and utilities, many of which seem trivial in isolation. It is possible to combine these tools in creative ways (using stuff like redirection and piping) and solve problems with astounding ease. Linux is a toolsmith's dream-come-true.

1.1. Introduction

1.1.1. Copyright and License

1.1.2. Feedback and Corrections

Kindly forward feedback and corrections to *pramode_ce@yahoo.co.in*.

1.1.3. Acknowledgements

Gopakumar would like to thank the faculty and friends at the Government Engineering College, Trichur for introducing him to GNU/Linux and initiating a 'Free Software Drive' which ultimately resulted in the whole Computer Science curriculum being taught without the use of propreitary tools and platforms.

As kernel newbies, we were fortunate to lay our hands on a copy of Alessandro Rubini and Jonathan Corbet's great book on Linux Device Drivers - we would like to thank them for writing such a wonderful book.

We express our gratitude towards those countless individuals who answer our queries on Internet newsgroups and mailing lists, those people who maintain this infrastructure, the hackers who write cool code just for the fun of writing it and everyone else who is a part of the great Free Software movement.

1.2. A simple problem and its solution

The 'anagram' problem has proved to be quite effective in conveying the power of the 'toolkit' approach. The problem is discussed in Jon Bentley's book *Programming Pearls.* The idea is this - you have to discover all anagrams contained in the system dictionary (say, /usr/share/dict/words) - an anagram being a combination of words like this:

```
top opt pot
```

The dictionary is sure to contain lots of interesting anagrams. Our job is to write a program which helps us see all anagrams which contain, say 5 words, or 4 words and so on. The impatient programmer would right away start coding in C - but the Unix master waits a bit, reflects on the problem, and hits upon a simple and elegant solution. She first writes a program which reads in a word from the keyboard and prints out the same word, together with its sorted form. That is, if the user enters:

hello

The program would print

ehllo hello

The program should keep on reading from the input till an EOF appears. Here is the code:

```
main()
{
 char s[100], t[100];
 while(scanf("%s", s) != EOF) {
  strcpy(t, s);
  sort(s);
  printf("%s %s\n", s, t);
 }
}

```

The function ‘sort’ is a user defined function which simply sorts the contents of the array alphabetically in ascending order. Lets call this program ‘sign.c’ and compile it into a binary called ‘sign’. Any program which reads from the keyboard can be made to read from a pipe - so we can do:

cat /usr/share/dict/words | ./sign

We will see lines from the dictionary scrolling through the screen with their ‘signatures’ (let’s call the sorted form of a word its ‘signature’) to the left. The dictionary might contain certain words which begin with upper case characters - it’s better to treat upper case and lower case uniformly, so we might transform all words to lowercase - we do it using the ‘tr’ command.

cat /usr/share/dict/words | tr ’A-Z’ ’a-z’ | ./sign

The ‘sort’ command sorts lines read from the standard input in ascending order based on the first word of each line. Lets do:

cat /usr/share/dict/words | tr ’A-Z’ ’a-z’ | ./sign | sort

Now, all anagrams are sure to come together (because their signatures are the same). In the next stage, we eliminate the signatures and bring all words which have the same signature on to the same line. We do it using a program called ‘sameline.c’.

```
main()
{
  char prev_sign[100]="";
  char curr_sign[100], word[100];
  while(scanf("%s%s", curr_sign, word)!=EOF) {
    if(strcmp(prev_sign, curr_sign) == 0) {
      printf("%s ", word);
    } else { /* Signatures differ */
      printf("\n");
      printf("%s ", word);
      strcpy(prev_sign, curr_sign);
    }
  }
}
```

Now, all sets of words which form anagrams appear on the same line in the output of the pipeline:

cat /usr/share/dict/words | tr ’A-Z’ ’a-z’ | ./sign | sort | ./sameline

All that remains for us to do is extract all three word anagrams, or four word anagrams etc. We do this using the ‘awk’ program:

cat /usr/share/dict/words | tr ’A-Z’ ’a-z’ | ./sign | sort | ./sameline | awk ’{if(NF==3)print}’

Awk reads an input line, checks if the number of fields (NF) is equal to 3, and if so, prints that line. We change the expression to NF==4 and we get all four word anagrams.

A competent Unix programmer, once he hits upon this idea, would be able to produce perfectly working code in under fifteen minutes - try doing this with any other OS!

1.2.1. Exercise

1.2.1.1. Hashing

Try adopting the ‘Unix approach’ to solving the following problem. You are given a hash function:

```
#define NBUCKETS 1000
#define MAGIC 31
int hash(char *s)
{
  unsigned int sum = 0, i;
  for(i = 0; s[i] != 0; i++)
    sum = sum * MAGIC + s[i];
  return sum%NBUCKETS;
```

```
}

```

Can you check whether it is a ‘uniform’ hash function? You note that the function returns values in the range 0 to 999, both included. If you are applying the function on say 45000 strings (say, the words in the system dictionary), you will be getting lots of repetitions - your job is to find out, say, how many times the number ‘230’ appears in the output.

1.2.1.2. Picture Drawing

Operating Systems which call themselves ‘Unix’ have a habit of treating everything as programming - even drawing a picture is a ‘programming’ activity! Try reading some document on the ‘pic’ language. Create a file which contains the following lines:

```
.PS
box "Hello"
arrow
box "World"
.PE

```

Run the following pipeline:

```
(pic a.pic | groff -Tps) > a.ps
```

View the resulting Postscript file using a viewer like ‘gv’.

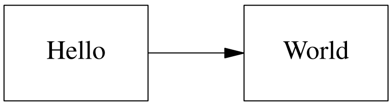

Figure 1-1. PIC in action

Chapter 2. Tools

It’s difficult to work on Linux without first getting to know the tools which make the environment so powerful. A thorough description of even a handful of tools would make up a mighty tome - so we have to really restrict ourselves.

2.1. The Unix Shell

The Unix Shell is undoubtedly the ‘Number One’ tool. Linux systems run ‘bash’ by default, but you can as well switch over to something like ‘csh’ - though there is little reason to do so. The inherent programmability of the shell is seductive - once you fall for it, there is no looking back. There are plenty of books which describe the environment which the shell provides - the best of them being ‘The Unix Programming Environment’, by ‘Kernighan&Pike’. You must ABSOLUTELY read at least the first three or four chapters of this book before you start doing something solid on Linux. Writing ‘throwaway’ scripts on the command line becomes second nature once you really start understanding the shell. Here is what we do when wish to put all our .jpg downloads whose size is greater than 15k onto a directory called ‘img’.

```
$ for i in `find . -name '*.jpg' -size +15k`
> do
> cp $i img
> done

```

The idea is that programming becomes so natural that you are not even aware of the fact that you are ‘programming’. What more can you ask for?

2.2. The C Compiler

C should be the last language a programmer thinks of when she plans to write an application program - there are far ‘safer’ languages available, our personal choice being Python. But once you decide that poking Operating Systems is going to be your favourite pasttime, there is only one way to go - you have to master the ‘Deep C Secrets’ (as Peter van der Linden puts it). Even though the language is very popular, there are very few good books - the first, and still the best is ‘The C Programming Language’ by Kernighan and Ritchie. It would be good if you could spend some time on it, especially the Appendix, which needs very careful reading. The ‘C FAQ’ and ‘C Traps and Pitfalls’, both of which, we believe, are available for download on the net should also be consulted.

The GNU Compiler Collection (GCC) is perhaps the most widely ported (and used) compiler toolkit outside the Windows world. Whatever be your CPU architecture, right from lowly 8 bit microcontrollers to high speed 64 bit processors, you may be assured of a GCC port.

2.2.1. From source code to machine code

It is essential that you have some idea of what really happens when you type ‘cc hello.c’.

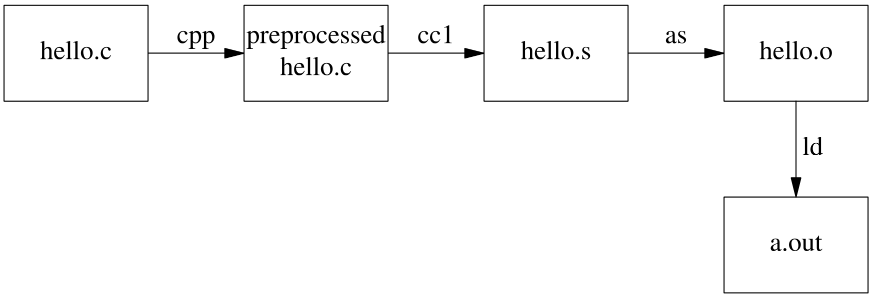

Figure 2-1. The four phases of compilation

The first phase of the compilation process is preprocessing; an independent program called ‘cpp’ reads your C code and ‘includes’ header files, replaces all occurrences of #defined symbols with their values, performs conditional filtering etc. The preprocessed C file is passed on to a program called ‘cc1’ which is the real C compiler - a complex program which converts the C source to assembly code. In the next phase, the assembler converts the assembly language program to machine code. The last phase is linking - a program called ‘ld’ combines the object code of your program with the object code of certain libraries to generate the executable ‘a.out’.

2.2.2. Options

The ‘cc’ command is merely a compiler ‘driver’ or ‘front end’. Its job is to collect command line arguments and pass them on to the four programs which do the actual compilation process.

The -E option makes ‘cc’ call only ‘cpp’. The output of the preprocessing phase is displayed on the screen. The -S option makes ‘cc’ invoke both ‘cpp’ and ‘cc1’. What you get would be a file with extension ‘.s’, an assembly language program. The -c option makes ‘cc’ invoke the first three phases - output would be an object file with extension ‘.o’. Typing

```
cc hello.c -o hello
```

Will result in output getting stored in a file called ‘hello’ instead of ‘a.out’.

The -Wall option enables all warnings. It is essential that you always compile your code with -Wall - you should let the compiler check your code as thoroughly as possible. The -pedantic-errors options checks your code for strict ISO compatibility. You must be aware that GCC implements certain extensions to the C language, if you wish your code to be strict ISO C, you must eliminate the possibility of such extensions creeping into it. Here is a small program which demonstrates the idea - we are using the named structure field initalization extension here, which gcc allows, unless -pedantic-errors is provided.

```
main()
{
 struct complex {int re, im;}
 struct complex c = {im:4, re:5};
}

```

Here is what gcc says when we use the -pedantic-errors option:

```
a.c: In function 'main':
a.c:4: ISO C89 forbids specifying structure member to initialize
a.c:4: ISO C89 forbids specifying structure member to initialize
```

As GCC is the dominant compiler in the free software world, using GCC extensions is not really a bad idea.

The compiler performs several levels of optimizations - which are enabled by the options -O, -O2 and -O3. Read the gcc man page and find out what all optimizations are enabled by each option.

The -I option is for the preprocessor - if you do

```
cc a.c -I/usr/proj/include
```

you are adding the directory /usr/proj/include to the standard preprocessor search path. The -D option is useful for defining symbols on the command line.

```
main()
{
 #ifdef DEBUG
 printf("hello");
 #endif
}

```

Try compiling the above program with the option -DDEBUG and without the option. It is also instructive to do:

```
cc -E -DDEBUG a.c
cc -E a.c
```

to see what the preprocessor really does. Note that the Linux kernel code makes heavy use of preprocessor tricks - so don't skip the part on the preprocessor in K&R.

The -L and -l options are for the linker. If you do

```
cc a.c -L/usr/X11R6/lib -lX11
```

the linker tries to combine the object code of your program with the object code contained in a file call 'libX11.so'; this file will be searched for in the directory /usr/X11R6/lib too, besides the standard directories like /lib and /usr/lib.

2.2.3. Exercise

Find out what the -fwritable-strings option does. Find out what the 'inline' keyword does - what is the effect of 'inline' together with optimization options like -O, -O2 and -O3? You will need to compile your code with the -S option and read the resulting assembly language program to solve this problem.

2.3. Make

Make is a program for automating the program compilation process - it is one of the most important components of the Unix programmer's toolkit. Kernighan and Pike describe 'make' in their book 'The Unix Programming Environment'. Make comes with a comprehensive manual, which might be found under /usr/info (or /usr/share/info) of your Linux system.

We are typing this document using the LyX wordprocessor. LyX exports the document we type as an SGML file. This SGML file is converted to the 'dvi' format by a program called 'db2dvi'. The resulting '.dvi' file is then converted to postscript using a program called 'dvips'. Postscripts files can be viewed using the program 'gv', which runs under X-Windows. We have created a file called 'Makefile' in the directory where we run LyX. The file contains the following lines:

```
module.ps: module.dvi
	dvips module.dvi -o module.ps; gv module.ps

module.dvi:module.sgml
	db2dvi module.sgml

```

After exporting the file as SGML from LyX, we simply type 'make' on another console. What does 'make' do? It first checks whether a file 'module.ps' (called a 'target') exists. Then it checks whether another file called 'module.dvi' exists - if not, this file is created by executing the action 'db2dvi module.sgml'. Once 'module.dvi' is built, make executes the actions

dvips module.dvi -o module.ps

gv module.ps

We see the file 'module.ps' displayed on a window.

Now what if we make some modifications to our LyX file and re-export it as an SGML document? We type 'make' once again. This time, the target 'module.ps' exists. The 'dependency' module.dvi also exists. Make checks the timestamps of both files to verify whether module.dvi is newer than module.ps. No. Now, make checks whether module.sgml is newer than module.dvi. It is. So make reexecutes the action and constructs a new module.dvi. Now module.dvi has become more recent than module.ps. So make calls dvips and constructs a new module.ps.

Linux programs distributed in source form always come with a Makefile. You will find the Makefile for the Linux kernel under /usr/src/linux. Try reading it.

2.4. Diff and Patch

The distributed development model, of which the Linux kernel is a good example, depends a good deal on two utilites - diff and patch. Diff takes two files as input and generates their 'difference'. If the original file is large, and if the modifications are minimal (which is usually the case in incremental software development), the ‘difference file’ would be quite small. Suppose two persons A and B are working on the same program. A makes some changes and sends the diff over to B; B then uses the ‘patch’ command to merge the changes to his copy of the original program.

2.4.1. Exercise

Find out what a ‘context diff’ is. Apply a context diff on two program files.

2.5. Grep

You know what it is - otherwise you won’t be reading this.

2.6. Vi, Ctags

The vi editor is a very powerful tool - it is adviced that you spend some time reading a book or some online docs and understand its capabilities.

When you are browsing through the source of large programs, you may wish to jump to the definition of certain function when you see them being invoked - these functions need not be defined in the file which you are currently reading. Suppose that you do

```
ctags *.c *.h
```

in the directory which holds the source files. Now you start reading one file, say, do_this.c. You see a function call

```
foo_baz(p, (int*)&m);
```

You want to see the definition of ‘foo_baz’. You simply switch over to command mode, place the cursor under foo_baz and type

```
Ctrl ]
```

That is, the Ctrl key and the close-square-brace key together. Vi immediately loads the file which contains the definition of foo_baz and takes you to the part which contains the body of the function. Now suppose you wish to go back. You type

```
Ctrl t
```

Very useful indeed!

Chapter 3. The System Call Interface

The ‘kernel’ is the heart of the Operating System. Your Linux system will most probably have a directory called /boot under which you will find a file whose name might look somewhat like ‘vmlinuz’. This file contains machine code (which is compiled from source files under /usr/src/linux) which gets loaded into memory when you boot your machine. Once the kernel is loaded into memory, it stays there until you reboot the machine, overseeing each and every activity going on in the system. The kernel is responsible for managing hardware resources, scheduling processes, controlling network communication etc. If a user program wants to, say, send data over the network, it has to interact with the TCP/IP code present within the kernel. This interaction takes place through special C functions which are called ‘System Calls’. Understanding a few elementary system calls is the first step towards understanding Linux.

The definitive book on the Unix system call interface is W.Richard Steven’s *Advanced Programming in the Unix Environment*. The reader may go through this book to get a deeper understanding of the topics discussed here. We have shamelessly copied a few of Steven’s diagrams in this document (well, we did learn PIC for drawing the figures - that was a great experience).

3.1. Files and Processes

3.1.1. File I/O

The Linux operating system, just like all Unices, takes the concept of a file to dizzying heights. A file is not merely a few bytes of data residing on disk - it is an abstraction for anything that can be read from or written to. Files are manipulated using three fundamental system calls - open, read and write. A system call is a C function which transfers control to a point within the operating system kernel. This needs to be elaborated a little bit.

The Linux source tree is rooted at /usr/src/linux. If you examine the file fs/open.c, you will see a function whose prototype looks like this:

```
asmlinkage long sys_open(const char* filename,
 int flags, int mode);

```

Now, this function is compiled into the kernel and is as such resident in memory. When the C program which you write calls ‘open’, control is getting transferred to this function within the operating system kernel. It is possible to make alterations to this function(or any other), recompile and install a new kernel - you just have to look through the ‘README’ file under /usr/src/linux. The availability of kernel source provides a multitude of opportunities to the student and researcher - students can ‘see’ how abstract operating system principles are implemented in practice and researchers can make their own enhancements.

Here is a small program which behaves like the copy command.

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

```

#define BUFLEN 1024

int main(int argc, char *argv[])
{
    int fdr, fdw, n;
    char buf[BUFLEN];

    assert(argc == 3);

    fdr = open(argv[1], O_RDONLY);
    assert(fdr >= 0);
    fdw = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0644);
    assert(fdw >= 0);
    while((n = read(fdr, buf, sizeof(buf))) > 0)
        if (write(fdw, buf, n) != n) {
            fprintf(stderr, "write error\n");
            exit(1);
        }

    if (n < 0) {
        fprintf(stderr, "read error\n");
        exit(1);
    }

    return 0;
}


```

Let us look at the important points. We see that ‘open’ returns an integer ‘file descriptor’ which is to be passed as argument to all other file manipulation functions. The first file is opened as read only. The second one is opened for writing - we are also specifying that we wish to truncate the file (to zero length) if it exists. We are going to create the file if it does not exist - and hence we pass a creation mode (octal 644 - user read/write, group and others read) as the last argument.

The read system call returns the actual number of bytes read, the return value is 0 if EOF is reached, it is -1 in case of errors.

The write system call returns the number of bytes written, which should be equal to the number of bytes which we have asked to write.

Note that there are subtleties with write. The write system call simply ‘schedules’ data to be written - it returns without verifying that the data has been actually transferred to the disk.

3.1.2. Process creation with ‘fork’

The fork system call creates an exact replica(in memory) of the process which executes the call.

```
main()
{
    fork();
    printf("hello\n");
}
```

You will see that the program prints hello twice. Why? After the call to ‘fork’, we will have two processes in memory - the original process which called the ‘fork’ (the parent process) and the clone which fork has created (the child process). Lines after the fork will be executed by both the parent and the child.

Fork is a peculiar function, it seems to return twice.

```
main()
{
    int pid;
    pid = fork();
    assert(pid >= 0);
    if (pid == 0) printf("I am child");
    else printf("I am parent");
}

```

This is quite an amazing program to anybody who is not familiar with the working of fork. Both the ‘if’ part as well as the ‘else’ part seems to be getting executed. The idea is that both parts are being executed by two different processes. Fork returns 0 in the child process and process id of the child in the parent process. It is important to note that the parent and the child are replicas - both the code and the data in the parent gets duplicated in the child - only thing is that parent takes the else branch and child takes the if branch.

3.1.3. Sharing files

It is important to understand how a fork affects open files. Let us play with some simple programs.

```
int main()
{
    char buf1[] = "hello", buf2[] = "world";
    int fd1, fd2;
    fd1 = open("dat", O_WRONLY|O_CREAT, 0644);
    assert(fd1 >= 0);
    fd2 = open("dat", O_WRONLY|O_CREAT, 0644);
    assert(fd2 >= 0);

    write(fd1, buf1, strlen(buf1));
    write(fd2, buf2, strlen(buf2));
}

```

After running the program, we note that the file ‘dat’ contains the string ‘world’. This demonstrates that calling open twice lets us manipulate the file independently through two descriptors. The behaviour is similar when we open and write to the file from two independent programs.

Every running process will have a per process file descriptor table associated with it - the value returned by open is simply an index to this table. Each per process file descriptor table slot will contain a pointer to a kernel file table entry which will contain:

1. the file status flags (read, write, append etc)
2. a pointer to the v-node table entry for the file
3. the current file offset

What does the v-node contain? It is a datastructure which contains, amongst other things, information using which it would be possible to locate the data blocks of the file on the disk. The diagram below shows the arrangement of these data structures for the code which we had right now written.

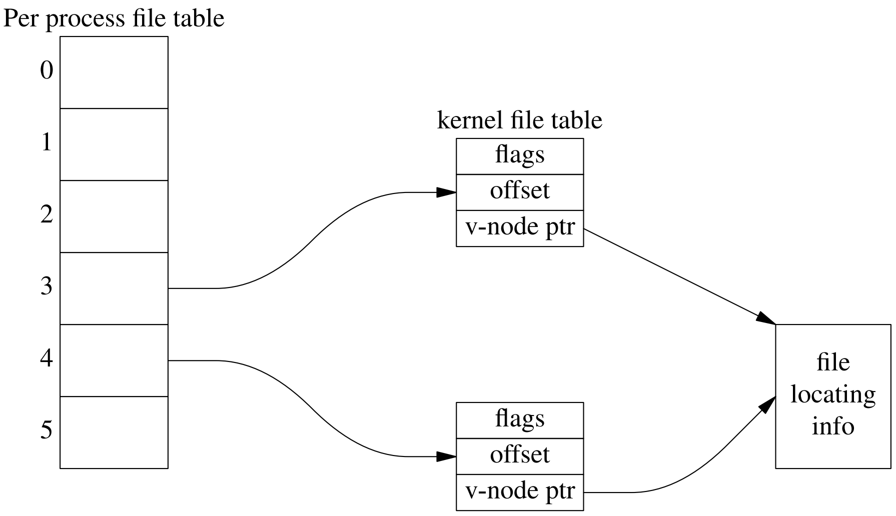

Figure 3-1. Opening a file twice

Note that the two descriptors point to two different kernel file table entries - but both the file table entries point to the same v-node structure. The consequence is that writes to both descriptors results in data getting written to the same file. Because the offset is maintained in the kernel file table entry, they are completely independent - the first write results in the offset field of the kernel file table entry pointed to by slot 3 of the file descriptor table getting changed to five (length of the string 'hello'). The second write again starts at offset 0, because slot 4 of the file descriptor table is pointing to a different kernel file table entry.

What happens to open file descriptors after a fork? Let us look at another program.

```
#include "myhdr.h"
main()
{
    char buf1[] = "hello";
    char buf2[] = "world";
    int fd;
    fd = open("dat", O_WRONLY|O_CREAT|O_TRUNC, 0644);
    assert(fd >= 0);

    write(fd, buf1, strlen(buf1));
    if(fork() == 0) write(fd, buf2, strlen(buf2));
}

```

We note that ‘open’ is being called only once. The parent process writes ‘hello’ to the file. The child process uses the same descriptor to write ‘world’. We examine the contents of the file after the program exits. We find that the file contains ‘helloworld’. The ‘open’ system call creates an entry in the kernel file table, stores the address of that entry in the process file descriptor table and returns the index. The ‘fork’ results in the child process inheriting the parent’s file descriptor table. The slot indexed by ‘fd’ in both the parent’s and child’s file descriptor table contains pointers to the same file table entry - which means the offsets are shared by both the process. This explains the behaviour of the program.

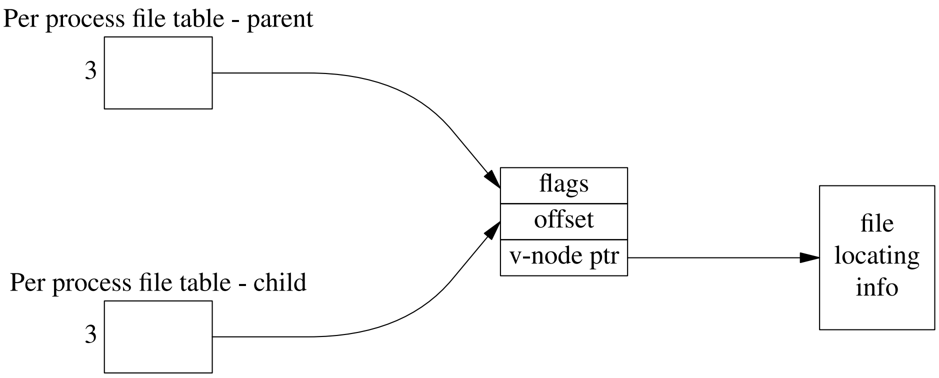

Figure 3-2. Sharing across a fork

3.1.4. The ‘exec’ system call

Let’s look at a small program:

```
int main()
{
    execlp("ls", "ls", 0);
    printf("Hello\n");
    return 0;
}

```

The program executes the ‘ls’ command - but we see no trace of a ‘Hello’ anywhere on the screen. What’s up?

The ‘exec’ family of functions perform ‘program loading’. If exec succeeds, it replaces the memory image of the currently executing process with the memory image of ‘ls’ - ie, exec has no place to return to if it succeeds! The first argument to execlp is the name of the command to execute. The subsequent arguments form the command line arguments of the execed program (ie, they will be available as argv[0], argv[1] etc in the execed program). The list should be terminated by a null pointer.

What happens to an open file descriptor after an exec? That is what the following program tries to find out. We first create a program called ‘t.c’ and compile it into a file called ‘t’.

```

main(int argc, char *argv[])
{
    char buf[] = "world";
    int fd;

    assert(argc == 2);
    fd = atoi(argv[1]);
    printf("got descriptor %d\n", fd);
    write(fd, buf, strlen(buf));
}

```

The program receives a file descriptor as a command line argument - it then executes a write on that descriptor. We will now write another program 'forkexec.c', which will fork and exec this program.

```
int main()
{
    int fd;
    char buf[] = "hello";
    char s[10];

    fd = open("dat", O_WRONLY|O_CREAT|O_TRUNC, 0644);
    assert(fd >= 0);
    sprintf(s, "%d", fd);
    write(fd, buf, strlen(buf));
    if(fork() == 0) {
        execl("./t", "t", s, 0);
        fprintf(stderr, "exec failed\n");
    }
}


```

What would be the contents of file 'dat' after this program is executed? We note that it is 'helloworld'. This demonstrates the fact that the file descriptor is not closed during the exec.

3.1.5. The 'dup' system call

You might have observed that the value of the file descriptor returned by 'open' is minimum 3. Why? The Unix shell, before forking and exec'ing your program, had opened the console thrice - on descriptors 0, 1 and 2. Standard library functions which write to 'stdout' are guaranteed to invoke the 'write' system call with a descriptor value of 1 while those functions which write to 'stderr' and read from 'stdin' invoke 'write' and 'read' with descriptor values 2 and 0. This behaviour is vital for the proper working of standard I/O redirection.

The 'dup' system call 'duplicates' the descriptor which it gets as the argument on the lowest unused descriptor in the per process file descriptor table.

```
#include "myhdr.h"

main()
{
    int fd;
```

```

    fd = open("dat", O_WRONLY|O_CREAT|O_TRUNC, 0644);
    close(1);
    dup(fd);
    printf("hello\n");
}

```

Note that after the dup, file descriptor 1 refers to whatever 'fd' is referring to. The 'printf' function invokes the write system call with descriptor value equal to 1, with the result that the message gets 'redirected' to the file 'dat' and does not appear on the screen.

3.2. The 'process' file system

The /proc directory of your Linux system is very interesting. The files (and directories) present under /proc are not really disk files. Here is what you will see if you do a 'cat /proc/interrupts':

```
           CPU0
  0:     296077          XT-PIC  timer
  1:       3514          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  4:       6385          XT-PIC  serial
  5:         15          XT-PIC  usb-ohci, usb-ohci
  8:          1          XT-PIC  rtc
 11:     337670          XT-PIC  nvidia, NVIDIA nForce Audio
 14:      11765          XT-PIC  ide0
 15:     272508          XT-PIC  ide1
NMI:          0
LOC:          0
ERR:          0
MIS:          0

```

By reading from (or writing to) files under /proc, you are in fact accessing data structures present within the Linux kernel - /proc exposes a part of the kernel to manipulation using standard text processing tools. You can try 'man proc' and learn more about the process information pseudo file system.

3.2.1. Exercises

1. You should attempt to design a simple Unix shell. You need to look up the man pages for certain other syscalls which we have not covered here - especially 'pipe' and 'wait'.
2. The Linux OS kernel contains support for TCP/IP networking. It is possible to plug in multiple network interfaces (say two ethernet cards) onto a Linux box and make it act as a gateway. When your machine acts as a 'gateway', it should be able to forward packets - ie, it should read a packet from one network interface and transfer it onto

another interface. It is possible to enable and disable IP forwarding by manipulating kernel data structures through the /proc file system. Try finding out how this could be done.

3. Read the manual page of the ‘mknod’ command and find out its use.

Chapter 4. Defining New System Calls

This will be our first kernel hack - mostly because it is extremely simple to implement. We shall examine the processing of adding new system calls to the Linux kernel - in the process, we will learn something about building new kernels - and one or two things about the very nature of the Linux kernel itself. Note that we are dealing with Linux kernel version 2.4.

Please note that making modifications to the kernel and installing modified kernels can lead to system hangs and data corruption and should not be attempted on production systems.

4.1. What happens during a system call?

In one word - Magic. It is difficult to understand the actual sequence of events which take place during a system call without having an intimate understanding of the processor on which the kernel is running - say the Intel 386+ family of CPU's. CPU's with built in memory management units (MMU's) implement various levels of 'protection' in hardware. The body of code which interacts intimately with the machine hardware forms the OS kernel - it runs at a very high privilege level. The code which runs as part of the kernel has permissions to do anything - read from and write to I/O ports, manage interrupts, control Direct Memory Access (DMA) transfers, execute 'privileged' CPU instructions etc. User programs run at a very low privilege level - and are not really capable of doing any 'low-level' stuff other than reading and writing I/O ports. User programs have to 'enter' into the kernel whenever they want service from hardware devices (say read from disk, keyboard etc). System calls form well defined 'entry points' through which user programs can get into the kernel. Whenever a user program invokes a system call, a few lines of assembly code executes - which takes care of switching from low privileged user mode to high privileged kernel mode.

4.2. A simple system call

Let's go to the /usr/src/linux/fs subdirectory and create a file called 'mycall.c'.

```
/* /usr/src/linux/fs/mycall.c */
#include <linux/linkage.h>

asmlinkage void sys_zap(void)
{
    printk("This is Zap from kernel...\n");
}

```

The Linux kernel convention is that system calls be prefixed with a sys_. The 'asmlinkage' is some kind of preprocessor macro which is present in /usr/src/linux/include/linux/linkage.h and seems to be essential for defining system calls. The system call simply prints a message using the kernel function 'printk' which is somewhat similar to the C library function 'printf' (Note that the kernel can't make use of the standard C library - it has its own implementation of most simple C library functions).

It is essential that this file gets compiled into the kernel - so you have to make some alterations to the 'Makefile'.

```

# Some lines deleted...
```

```

obj-y:=open.o read_write.o devices.o file_table.o buffer.o \
    super.o block_dev.o char_dev.o stat.o exec.o pipe.o namei.o \
    fcntl.o ioctl.o readdir.o select.o fifo.o locks.o \
    dcache.o inode.o attr.o bad_inode.o file.o iobuf.o dnotify.o \
    filesystems.o namespace.o seq_file.o mycall.o

ifeq ($(CONFIG_QUOTA),y)
obj-y += dquot.o
else

# More lines deleted ...

```

Note the line containing ‘mycall.o’.

Once this change is made, we have to examine the file /usr/src/linux/arch/i386/kernel/entry.S. This file defines a table of system calls - we add our own syscall at the end. Each system call has a number of its own, which is basically an index into this table - ours is numbered 239.

```
    .long SYMBOL_NAME(sys_ni_syscall)
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    .long SYMBOL_NAME(sys_read)
    .long SYMBOL_NAME(sys_write)
    .long SYMBOL_NAME(sys_open)

/* Lots of lines deleted */
    .long SYMBOL_NAME(sys_ni_syscall)
    .long SYMBOL_NAME(sys_tkill)
    .long SYMBOL_NAME(sys_zap)

    .rept NR_syscalls-(.-sys_call_table)/4
    .long SYMBOL_NAME(sys_ni_syscall)
    .endr

```

We will also add a line

```
#define __NR_zap 239

```

to /usr/src/linux/include/asm/unistd.h. We are now ready to go.

We have made all necessary modifications to our kernel. We now have to rebuild it. This can be done by typing, in sequence:

1. make menuconfig
2. make dep
3. make bzImage

A new kernel called ‘bzImage’ will be available under /usr/src/linux/arch/i386/boot. You have to copy this to a directory called, say, /boot - remember not to overwrite the kernel which you are currently running - if there is some problem with your modified kernel, you should be able to fall back to your functional kernel. You will have to add the name of this kernel to a boot loader configuration file (if you are using lilo, then /etc/lilo.conf) and run some command like 'lilo'. Here is the /etc/lilo.conf which we are using:

```
prompt
timeout=50
default=linux
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
message=/boot/message
lba32
vga=0xa
image=/boot/vmlinuz-2.4.18-3
        label=linux
        read-only
        append="hdd=ide-scsi"
        root=/dev/hda3

image=/boot/nov22-ker
        label=syscall-hack
        read-only
        root=/dev/hda3


other=/dev/hda1
        optional
        label=DOS

other=/dev/hda2
        optional
        label=FreeBSD


```

The default kernel is /boot/vmlinuz-2.4.18-3. The modified kernel is called /boot/nov22-ker. Note that you have to type 'lilo' after modifying /etc/lilo.conf. If you are using something like 'Grub', consult the man pages and make the necessary modifications.

You can now reboot the system and load the new Linux kernel. You then write a C program:

```
main()
{
    syscall(239);
}

```

And you will see a message 'This is Zap from kernel...' on the screen (Note that if you are running something like an xterm, you may not see the message on the screen - you can then use the 'dmesg' command. We will explore printk and message logging in detail later).

You should try one experiment if you don't mind your machine hanging. Place an infinite loop in the body of sys_zap - a 'while(1);' would do. What happens when you invoke sys_zap? Is the Linux kernel capable of preempting itself?

Chapter 5. Module Programming Basics

The next few chapters will cover the basics of writing kernel modules. Our discussion will be centred around the Linux kernel version 2.4. As this is an ‘introductory’ look at Linux systems programming, we shall skip those material which might confuse a novice reader - especially those related to portability between various kernel versions and machine architectures, SMP issues and error handling. Please understand that these are very vital issues, and should be dealt with when writing professional code. The reader who gets motivated to learn more should refer the excellent book ‘Linux Device Drivers’ by Alessandro Rubini and Jonathan Corbet.

5.1. What is a kernel module?

A kernel module is simply an object file which can be inserted into the running Linux kernel - perhaps to support a particular piece of hardware or to implement new functionality. The ability to dynamically add code to the kernel is very important - it helps the driver writer to skip the install-new-kernel-and-reboot cycle; it also helps to make the kernel lean and mean. You can add a module to the kernel whenever you want certain functionality - once that is over, you can remove the module from kernel space, freeing up memory.

5.2. Our First Module

```
#include <linux/module.h>
int init_module(void)
{
    printk("Module Initializing...\n");
    return 0;
}
void cleanup_module(void)
{
    printk("Cleaning up...\n");
}

```

Compile the program using the commandline:

```
cc -c -O -DMODULE -D__KERNEL__ module.c -I/usr/src/linux/include
```

You will get a file called ‘module.o’. You can now type:

```
insmod ./module.o
```

and your module gets loaded into kernel address space. You can see that your module has been added, either by typing

```
lsmod
```

or by examining /proc/modules. The ‘init_module’ function is called after the module has been loaded - you can use it for performing whatever initializations you want. The ‘cleanup_module’ function is called when you type:

```
rmmod module
```

That is, when you attempt to remove the module from kernel space.

5.3. Accessing kernel data structures

The code which you write as a module is running as part of the Linux kernel, and is capable of manipulating data structures defined in the kernel. Here is a simple program which demonstrates the idea.

```

#include <linux/module.h>
#include <linux/sched.h>

int init_module(void)
{
    printk("hello\n");
    printk("name = %s\n", current->comm);
    printk("pid = %d\n", current->pid);
    return 0;
}

void cleanup_module(void) { printk("world\n"); }

/* Look at /usr/src/linux/include/asm/current.h,
 * especially, the macro implementation of current
 */


```

The init_module function is called by the ‘insmod’ command after the module is loaded into the kernel. You can think of ‘current’ as a globally visible pointer to structure - the ‘comm’ and ‘pid’ fields of this structure give you the command name as well as the process id of the ‘currently executing’ process (which, in this case, is ‘insmod’ itself).

Every now and then, it would be good to browse through the header files which you are including in your program and look for ‘creative’ uses of preprocessor macros. Here is /usr/src/linux/include/asm/current.h for your reading pleasure!

```
#ifndef _I386_CURRENT_H
#define _I386_CURRENT_H

struct task_struct;

static inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
    return current;
 }
```

```
#define current get_current()
#endif /* !(_I386_CURRENT_H) */
```

'current' is infact a function which, using some inline assembly magic, retrieves the address of an object of type 'task struct' and returns it to the caller.

5.4. Symbol Export

The global variables defined in your module are accessible from other parts of the kernel. Lets compile and load the following module:

```
#include <linux/module.h>
int foo_baz = 101;
int init_module(void) { printk("hello\n"); return 0;}
void cleanup_module(void) { printk("world\n"); }
```

Now, either run the 'ksyms' command or look into the file /proc/ksysms - this file will contain all symbols which are 'exported' in the Linux kernel - you should find 'foo_baz' in the list. Once we take off the module, recompile and reload it with foo_baz declared as a 'static' variable, we wont be able to see foo_baz in the kernel symbol listing.

Modules may sometimes 'stack over' each other - ie, one module will make use of the functions and variables defined in another module. Let's check whether this works. We compile and load another module, in which we try to print the value of the variable foo_baz.

```
#include <linux/module.h>
extern int foo_baz;
int init_module(void)
{
    printk("foo_baz=%d\n", foo_baz);
    return 0;
}
void cleanup_module(void) { printk("world\n"); }
```

The module gets loaded and the init_module function prints 101. It would be interesting to try and delete the module in which foo_baz was defined.

The 'modprobe' command is used for automatically locating and loading all modules on which a particular module depends - it simplifies the job of the system administrator. You may like to go through the file /lib/modules/2.4.18-3/modules.dep (note that your kernel version number may be different).

5.5. Usage Count

```
#include <linux/module.h>
int init_module(void)
{
```

```
    MOD_INC_USE_COUNT;
    printk("hello\n"); return 0;
}

void cleanup_module(void) { printk("world\n"); }

```

After loading the program as a module, what if you try to ‘rmmod’ it? We get an error message. The output of ‘lsmod’ shows the used count to be 1. A module should not be accidentally removed when it is being used by a process. Modern kernels can automatically track the usage count, but it will be sometimes necessary to adjust the count manually.

5.6. User defined names to initialization and cleanup functions

The initialization and cleanup functions need not be called init_module() and cleanup_module().

```

#include <linux/module.h>
#include <linux/init.h>
int foo_init(void) { printk("hello\n"); return 0;}
void foo_exit(void) { printk("world\n"); }

module_init(foo_init);
module_exit(foo_exit);


```

Note that the macro’s placed at the end of the source file, module_init() and module_exit(), perform the ‘magic’ required to make foo_init and foo_exit act as the initialization and cleanup functions.

5.7. Reserving I/O Ports

A driver needs some way to tell the kernel that it is manipulating some I/O ports - and well behaved drivers need to check whether some other driver is using the I/O ports which it intends to use. Note that what we are looking at is a pure software solution - there is no way that you can reserve a range of I/O ports for a particular module in hardware.

Here is the content of the file /file/ioports on my machine running Linux kernel 2.4.18:

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : serial(auto)
```

```
0376-0376 : ide1
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(auto)
0cf8-0cff : PCI conf1
5000-500f : PCI device 10de:01b4 (nVidia Corporation)
5100-511f : PCI device 10de:01b4 (nVidia Corporation)
5500-550f : PCI device 10de:01b4 (nVidia Corporation)
b800-b80f : PCI device 10de:01bc (nVidia Corporation)
  b800-b807 : ide0
  b808-b80f : ide1
e000-e07f : PCI device 10de:01b1 (nVidia Corporation)
e100-e1ff : PCI device 10de:01b1 (nVidia Corporation)
```

The content can be interpreted in this way - the serial driver is using ports in the range 0x2f8 to 0x2ff, hard disk driver is using 0x376 and 0x3f6 etc.

Here is a program which checks whether a particular range of I/O ports is being used by any other module, and if not reserves that range for itself.

```
#include <linux/module.h>
#include <linux/ioport.h>

int init_module(void)
{
    int err;
    if((err = check_region(0x300, 5)) < 0) return err;
    request_region(0x300,5, "foobaz");
    return 0;
}

void cleanup_module(void)
{
    release_region(0x300, 5);
    printk("world\n");
}
```

You should examine /proc/ioports once again after loading this module.

5.8. Passing parameters at module load time

It may sometimes be necessary to set the value of certain variables within the module at load time. Take the case of an old ISA network card - the module has to be told the I/O base of the network card. We do it by typing:

```
insmod ne.o io=0x300
```

Here is an example module where we pass the value of the variable foo_dat at module load time.

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

int foo_dat = 0;
MODULE_PARM(foo_dat, "i");

int init_module(void)
{
        printk("hello\n");
        printk("foo_dat = %d\n", foo_dat);
        return 0;
}

void cleanup_module(void) { printk("world\n"); }

/* Type insmod ./k.o foo_dat=10. If
 * misspelled, we get an error message.
 *
 */
```

The MODULE_PARM macro announces that foo_dat is of type integer and can be provided a value at module load time, on the command line. Five types are currently supported, b for one byte; h for two bytes; i for integer; l for long and s for string.

Chapter 6. Character Drivers

Device drivers are classified into character, block and network drivers. The simplest to write and understand is the character driver - we shall start with that. Note that we will not attempt any kind of actual hardware interfacing at this stage - we will do it later.

Before we proceed any further, you have to once again refresh whatever you have learnt about the file handling system calls - open, read, write etc and the way file descriptors are shared between parent and child processes.

6.1. Special Files

Go to the /dev directory and try ‘ls -l’. Here is the output on our machine:

```
total 170
crw-------    1 root     root       10,  10 Apr 11  2002 adbmouse
crw-r--r--    1 root     root       10, 175 Apr 11  2002 agpgart
crw-------    1 root     root       10,   4 Apr 11  2002 amigamouse
crw-------    1 root     root       10,   7 Apr 11  2002 amigamouse1
crw-------    1 root     root       10, 134 Apr 11  2002 apm_bios
drwxr-xr-x    2 root     root          4096 Oct 14 20:16 ataraid
crw-------    1 root     root       10,   5 Apr 11  2002 atarimouse
crw-------    1 root     root       10,   3 Apr 11  2002 atibm
crw-------    1 root     root       10,   3 Apr 11  2002 atimouse
crw-------    1 root     root       14,   4 Apr 11  2002 audio
crw-------    1 root     root       14,  20 Apr 11  2002 audio1
crw-------    1 root     root       14,   7 Apr 11  2002 audioctl
brw-rw----    1 root     disk       29,   0 Apr 11  2002 aztcd
crw-------    1 root     root       10, 128 Apr 11  2002 beep

```

You note that the permissions field begins with, in most cases, the character ‘c’. We have a ‘d’ against one name and a ‘b’ against another. A file whose permission field starts with a ‘c’ is called a character special file and one which starts with ‘b’ is a block special file. These files dont have sizes, instead they have what are called major and minor numbers. They are not files in the sense they don’t represent streams of data on a disk - they are mostly abstractions of peripheral devices.

Let’s suppose that you execute the command

```
echo hello > /dev/lp0
```

Had lp0 been an ordinar file, the string ‘hello’ would have appeared within it. But you observe that if you have a printer connected to your machine and if it is turned on, ‘hello’ gets printed on the paper. Thus, lp0 is acting as some kind of ‘access point’ through which you can talk to your printer. The choice of the file as a mechanism to define access points to peripheral devices is perhaps one of the most significant (and powerful) ideas popularized by Unix.

How is it that a ‘write’ to /dev/lp0 results in characters getting printed on paper? Let’s think of it this way. The kernel contains some routines (loaded as a module) for initializing a printer, writing data to it, reading back error messages etc. These routines form the ‘printer device driver’. Let’s suppose that these routines are called:

```
printer_open
```

printer_read

printer_write

Now, the device driver programmer loads these routines into kernel memory either statically linked with the kernel or dynamically as a module. Let's suppose that the driver programmer stores the address of these routines in some kind of a structure (which has fields of type 'pointer to function', whose names are, say, 'open', 'read' and 'write') - let's also suppose that the address of this structure is 'registered' in a table within the kernel, say at index 254. Now, the driver writer creates a 'special file' using the command:

mknod c printer 253 0

An 'ls -l printer' displays:

crw-r--r-- 1 root root 253, 0 Nov 26 08:15 printer

What happens when you attempt to write to this file? The 'write' system call understands that 'printer' is a special file - so it extracts the major number (which is 254) and indexes a table in kernel memory(the very same table into which the driver programmer has stored the address of the structure containing pointers to driver routines) from where it gets the address of a structure. Write then simply calls the function whose address is stored in the 'write' field of this structure, thereby invoking 'printer_write'. That's all there is to it, conceptually.

Before we write to a file, we will have to 'open' it - the 'open' system call also behaves in a similar manner - ultimately executing 'printer_open'.

Let's put these ideas to test. Look at the following program:

```

#include <linux/module.h>
#include <linux/fs.h>

static struct file_operations fops = {
    open: NULL,
    read: NULL,
    write: NULL,
};

static char *name = "foo";
static int major;

int init_module(void)
{
    major = register_chrdev(0, name, &fops);
    printk("Registered, got major = %d\n", major);
    return 0;
}

void cleanup_module(void)
{
    printk("Cleaning up...\n");
    unregister_chrdev(major, name);
}
```

We are not defining any device manipulation functions at this stage - we simply create a variable of type ‘struct file_operations’ and initialize some of its fields to NULL Note that we are using the GCC structure initialization extension to the C language. We then call a function

```
register_chrdev(0, name, &fops);
```

The first argument to register_chrdev is a Major Number (ie, the slot of a table in kernel memory where we are going to put the address of the structure) - we are using the special number ‘0’ here - by using which we are asking register_chrdev to identify an unused slot and put the address of our structure there - the slot index will be returned by register_chrdev. During cleanup, we ‘unregister’ our driver.

We compile this program into a file called ‘a.o’ and load it. Here is what /proc/devices looks like after loading this module:

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
-----Many Lines Deleted----
140 pts
141 pts
142 pts
143 pts
162 raw
180 usb
195 nvidia
254 foo

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md
 12 unnamed
 14 unnamed
 22 ide1
 38 unnamed
 39 unnamed

```

Note that our driver has been registered with the name ‘foo’, major number is 254. We will now create a special file called, say, ‘foo’ (the name can be anything, what matters is the major number).

```
mknod foo c 254 0
```

Let’s now write a small program to test our dummy driver.

```
#include "myhdr.h"

```

```
main()
{
    int fd, retval;
    char buf[] = "hello";

    fd = open("foo", O_RDWR);
    if (fd < 0) {
        perror("");
        exit(1);
    }
    printf("fd = %d\n", fd);
    retval=write(fd, buf, sizeof(buf));
    printf("write retval=%d\n", retval);
    if(retval < 0) perror("");
    retval=read(fd, buf, sizeof(buf));
    printf("read retval=%d\n", retval);
    if (retval < 0) perror("");
}
```

Here is the output of running the above program(Note that we are not showing the messages coming from the kernel).

```
fd = 3
write retval=-1
Invalid argument
read retval=-1
Invalid argument
```

Lets try to interpret the output. The ‘open’ system call, upon realizing that our file is a special file, looks up the table in which we have registered our driver routines(using the major number as an index). It gets the address of a structure and sees that the ‘open’ field of the structure is NULL. Open assumes that the device does not require any initialization sequence - so it simply returns to the caller.

Open performs some other tricks too. It builds up a structure (of type ‘file’) and stores certain information (like the current offset into the file, which would be zero initially) in it. A field of this structure will be initialized with the address of the structure which holds pointers to driver routines. Open stores the address of this object (of type file) in a slot in the per process file descriptor table and returns the index of this slot as a ‘file descriptor’ back to the calling program.

Now what happens during

write(fd, buf, sizeof(buf));

The write system call uses the value in fd to index the file descriptor table - from there it gets the address of an object of type ‘file’ - one field of this object will contain the address of a structure which contains pointers to driver routines - write examines this structure and realizes that the ‘write’ field of the structure is NULL - so it immediately goes back to the caller with a negative return value - the logic being that a driver which does not define a ‘write’ can’t be written to. The application program gets -1 as the return value - calling perror() helps it find out the nature of the error (there is a little bit of ‘magic’ here which we intentionally leave out from our discussion). Similar is the case with read.

We will now change our module a little bit.

```
#include <linux/module.h>
#include <linux/fs.h>

static char *name = "foo";
static int major;

static int
foo_open(struct inode* inode, struct file *filp)
{
    printk("Major=%d, Minor=%d\n", MAJOR(inode->i_rdev),
            MINOR(inode->i_rdev));
    /* Perform whatever actions are
     * need to physically open the
     * hardware device
     */
    printk("Offset=%d\n", filp->f_pos);
    printk("filp->f_op->open=%x\n", filp->f_op->open);
    printk("address of foo_open=\n", foo_open);
    return 0; /* Success */
}

static int
foo_read(struct file *filp, char *buf,
 size_t count, loff_t *offp)
{
    printk("&filp->f_pos=%x\n", &filp->f_pos);
    printk("offp=%x\n", offp);
    /* As of now, dummy */
    return 0;
}

static int
foo_write(struct file *filp, const char *buf,
 size_t count, loff_t *offp)
{
    /* As of now, dummy */
    return 0;
}

static struct file_operations fops = {
    open: foo_open,
    read: foo_read,
    write: foo_write
};

int init_module(void)
{
    major = register_chrdev(0, name, &fops);
    printk("Registered, got major = %d\n", major);
    return 0;
}
```

```
void cleanup_module(void)
{
    printk("Cleaning up...\n");
    unregister_chrdev(major, name);
}
```

We are now filling up the structure with address of three functions, foo_open, foo_read and foo_write. What are the arguments to foo_open? When the ‘open’ system call ultimately gets to call foo_open after several layers of indirection, it always passes two arguments, both of which are pointers. Our foo_open should be prepared to access these arguments. The first argument is a pointer to an object of type ‘struct inode’. An inode is a disk data structure which stores information about a file like its permissions, ownership, date, size, location of data blocks (if it is a real disk file) and major and minor numbers (in case of special files). An object of type ‘struct inode’ mirrors this information in kernel memory space. Our foo_open function, by accessing the field i_rdev through certain macros, is capable of finding out what the major and minor numbers of the file on which the ‘open’ system call is acting.

The next argument is of type ‘pointer to struct file’. We had mentioned earlier that the per process file descriptor table contains addresses of structures which store information like current file offset etc. The second argument to open is the address of this structure. Note that this structure in turn contains the address of the structure which holds the address of the driver routines(the field is called f_op), including foo_open! Does this make you crazy?

It should not. When you read the kernel source, you will realize that most of the complexity of the code is in the way the data structures are organized. The code which acts on these data structures would be fairly straightforward. This is the way large programs are (or should be) written, most of the complexity should be confined to (or captured in) the data structures - the algorithms should be made as simple as possible. It is comparitively easier for us to decode complex data structures than complex algorithms. Of courses, there will be places in the code where you will be forced to use complex algorithms - if you are writing numerical programs, algorithmic complexity is almost unavoidable; same is the case with optimizing compilers, many optimization techniques have strong mathematical (read graph theoretic) foundations and they are inherently complex. Operating systems are fortunately not riddled with such algorithmic complexitites.

What about the arguments to foo_read and foo_write. We have a buffer and count, together with a field called ‘offp’, which we may interpret as the address of the f_pos field in the structure pointed to by ‘filep’ (Wonder why we need this field? Why dont we straightaway access filp->f_pos?).

Here is what gets printed on the screen when we run the test program (which calls open, read and write). Again, note that we are not printing the kernel’s response.

```
fd = 3
write retval=0
read retval=0
```

The response from the kernel is interesting. We note that the address of foo_open does not change. That is because the module stays in kernel memory - every time we are running our test program, we are calling the same foo_open. But note that the ‘&filp->f_pos’ and ‘offp’ values, though they are equal, may keep on changing. This is because every time we are calling ‘open’, the kernel creates a new object of type ‘struct file’.

6.2. Use of the ‘release’ method

The driver open method should be composed of initializations. It is also preferable that the ‘open’ method increments the usage count. If an application program calls open, it is necessary that the driver code stays in memory till it calls ‘close’. When there is a close on a file descriptor (either explicit or implicit - when your program terminates, ‘close’ is invoked on all open file descriptors automatically) - the ‘release’ driver method gets called - you can think of decrementing the usage count in the body of ‘release’.

```
#include <linux/module.h>
#include <linux/fs.h>

static char *name = "foo";
static int major;

static int
foo_open(struct inode* inode, struct file *filp)
{
    MOD_INC_USE_COUNT;
    return 0; /* Success */
}

static int foo_close(struct inode *inode,
 struct file *filp)
{
    printk("Closing device...\n");
    MOD_DEC_USE_COUNT;
    return 0;
}
static struct file_operations fops = {
    open: foo_open,
    release: foo_close

};

int init_module(void)
{
    major = register_chrdev(0, name, &fops);
    printk("Registered, got major = %d\n", major);
    return 0;
}

void cleanup_module(void)
{
    printk("Cleaning up...\n");
    unregister_chrdev(major, name);
}
```

Lets load this module and test it out with the following program:

```
#include "myhdr.h"

main()
{
    int fd, retval;
    char buf[] = "hello";

    fd = open("foo", O_RDWR);
    if (fd < 0) {
      perror("");
      exit(1);
    }
    while(1);
}
```

We see that as long as the program is running, the use count of the module would be 1 and rmmod would fail. Once the program terminates, the use count becomes zero.

A file descriptor may be shared among many processes - the release method does not get invoked every time a process calls close() on its copy of the shared descriptor. Only when the last descriptor gets closed (that is, no more descriptors point to the ‘struct file’ type object which has been allocated by open) does the release method get invoked. Here is a small program which will make the idea clear:

```
#include "myhdr.h"

main()
{
    int fd, retval;
    char buf[] = "hello";

    fd = open("foo", O_RDWR);
    if (fd  < 0) {
        perror("");
        exit(1);
    }
    if(fork() == 0) {
        sleep(1);
        close(fd); /* Explicit close by child */
    } else {
        close(fd); /* Explicit close by parent */
    }
}
```

6.3. Use of the ‘read’ method

Transferring data from kernel address space to user address space is the main job of the read function:

ssize_t read(struct file* filep, char *buf, size_t count, loff_t *offp);

Say we are defining the read method of a scanner device. Using various hardware tricks, we acquire image data from the scanner device and store it in an array. We now have to copy this array to user address space. It is not possible to do this using standard functions like 'memcpy' due to various reasons. We have to make use of the functions:

```
unsigned long copy_to_user(void *to, const void* from, unsigned long count);
```

and

```
unsigned long copy_from_user(void *to, const void* from, unsigned long count);
```

These functions return 0 on success (ie, all bytes have been transferred, 0 more bytes to transfer).

Before we try to implement read (we shall try out the simplest implementation - the device supports only read - and we shall not pay attention to details of concurrency. This is a bad approach. We shall examine concurrency issues later on) we should once again examine how an application program uses the read syscall. Read is invoked with a file descriptor, a buffer and a count. Suppose that an application program is attempting to read a file in full, till EOF is reached, trying to read N bytes at a time. Read can return a value less than or equal to N. The application program should keep on reading till read returns 0. This way, it will be able to read the file in full.

Here is a simple driver read method - trying to see the contents of this device by using a standard command like cat should give us the output 'Hello, World\n'. Also, we should be able to get the same output from programs which attempt to read from the file in several different block sizes.

```
static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    static char msg[] = "Hello, world\n";
    int data_len = strlen(msg);
    int curr_off = *f_pos, remaining;

    if(curr_off >= data_len) return 0;
    remaining = data_len - curr_off;
    if (count <= remaining) {
        if(copy_to_user(buf, msg+curr_off, count))
            return -EFAULT;
        *f_pos = *f_pos + count;
        return count;
    } else {
        if(copy_to_user(buf, msg+curr_off, remaining))
            return -EFAULT;
        *f_pos = *f_pos + remaining;
        return remaining;
    }
}
```

Here is a small application program which exercises the driver read function with different read counts:

```
#include "myhdr.h"
#define MAX 1024

int
main()
{
    char buf[MAX];
    int fd, n, ret;

    fd = open("foo", O_RDONLY);
    assert(fd >= 0);
    printf("Enter read quantum: ");
    scanf("%d", &n);

    while((ret=read(fd, buf, n))>0)
       write(1, buf, ret); /* Write to stdout */
    if (ret < 0) {
       fprintf(stderr, "Error in read\n");
       exit(1);
    }
    exit(0);
}

```

6.4. A simple ‘ram disk’

Here is a simple ram disk device which behaves like this - initially, the device is empty. If you write, say 5 bytes and then perform a read

echo -n hello > foo

cat foo

You should be able to see ‘hello’. If you now do

echo -n abc > foo

cat foo

you should be able to see only ‘abc’. If you attempt to write more than MAXSIZE characters, you should get a ‘no space’ error - but as many characters as possible should be written. Here is the full source code:

```
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

#define MAXSIZE 512

static char *name = "foo";
```

```
static int major;
static char msg[MAXSIZE];
static int curr_size = 0;

static int
foo_open(struct inode* inode, struct file *filp)
{
    MOD_INC_USE_COUNT;
    return 0; /* Success */
}


static int
foo_write(struct file* filp, const char *buf,
 size_t count, loff_t *f_pos)
{
    int curr_off = *f_pos;
    int remaining = MAXSIZE - curr_off;

    if(curr_off >= MAXSIZE) return -ENOSPC;
    if (count <= remaining) {
        if(copy_from_user(msg+curr_off, buf, count))
                return -EFAULT;
        *f_pos = *f_pos + count;
        curr_size = *f_pos;
        return count;
    } else {
        if(copy_from_user(msg+curr_off, buf, remaining))
                return -EFAULT;
        *f_pos = *f_pos + remaining;
        curr_size = *f_pos;
        return remaining;
    }
}

static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    int data_len = curr_size;
    int curr_off = *f_pos, remaining;

    if(curr_off >= data_len) return 0;
    remaining = data_len - curr_off;
    if (count <= remaining) {
        if(copy_to_user(buf, msg+curr_off, count))
                return -EFAULT;
        *f_pos = *f_pos + count;
        return count;
    } else {
        if(copy_to_user(buf, msg+curr_off, remaining))
                return -EFAULT;
        *f_pos = *f_pos + remaining;
        return remaining;
    }
}

```

```
static int foo_close(struct inode *inode,
 struct file *filp)
{
    MOD_DEC_USE_COUNT;
    printk("Closing device...\n");
    return 0;
}

static struct file_operations fops = {
    open: foo_open,
    read: foo_read,
    write: foo_write,
    release: foo_close

};

int init_module(void)
{
    major = register_chrdev(0, name, &fops);
    printk("Registered, got major = %d\n", major);
    return 0;
}

void cleanup_module(void)
{
    printk("Cleaning up...\n");
    unregister_chrdev(major, name);
}


```

After compiling and loading the module and creating the necessary device file, try redirecting the output of Unix commands. See whether you get the ‘no space’ error (try ls -l > foo). Write C programs and verify the behaviour of the module.

6.5. A simple pid retriever

A process opens the device file, ‘foo’, performs a read, and magically, it gets its own process id.

```

static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    static char msg[MAXSIZE];
    int data_len;
    int curr_off = *f_pos, remaining;

    sprintf(msg, "%u", current->pid);
    data_len = strlen(msg);
    if(curr_off >= data_len) return 0;
    remaining = data_len - curr_off;
    if (count <= remaining) {
        if(copy_to_user(buf, msg+curr_off, count))
```

```
                return -EFAULT;
        *f_pos = *f_pos + count;
        return count;
    } else {
        if(copy_to_user(buf, msg+curr_off, remaining))
                return -EFAULT;
        *f_pos = *f_pos + remaining;
        return remaining;
    }
}


```

Chapter 7. Ioctl and Blocking I/O

We discuss some more advanced character driver operations in this chapter.

7.1. Ioctl

It may sometimes be necessary to send ‘commands’ to your device - especially when you are controlling a real physical device, say a serial port. Lets say that you wish to set the baud rate (data transfer rate) of the device to 9600 bits per second. One way to do this is to embed control sequences in the input stream of the device. Let’s send a string ‘set baud: 9600’. The difficulty with this approach is that the input stream of the device should now never contain a string of the form ‘set baud: 9600’ during normal operations. Imposing special ‘meaning’ to symbols on the input stream is most often an ugly solution. A better way is to use the ‘ioctl’ system call.

```
ioctl(int fd, int cmd, ...);
```

Associated with which we have a driver method:

```
foo_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
```

Here is a simple module which demonstrates the idea.

Lets first define a header file which will be included both by the module and by the application program.

```
#define FOO_IOCTL1 0xab01
#define FOO_IOCTL2 0xab02

```

We now create the module:

```
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

#include "foo.h"

static int major;
char *name = "foo";

static int
foo_ioctl(struct inode *inode, struct file *filp,
 unsigned int cmd, unsigned long arg)
{
    printk("received ioctl number %x\n", cmd);
    return 0;
}


static struct file_operations fops = {
    ioctl: foo_ioctl,

```

```
};

int init_module(void)
{
    major = register_chrdev(0, name, &fops);
    printk("Registered, got major = %d\n", major);
    return 0;
}

void cleanup_module(void)
{
    printk("Cleaning up...\n");
    unregister_chrdev(major, name);
}


```

And a simple application program which exercises the ioctl:

```
#include "myhdr.h"
#include "foo.h"


main()
{
    int r;
    int fd = open("foo", O_RDWR);
    assert(fd >= 0);

    r = ioctl(fd, FOO_IOCTL1);
    assert(r == 0);
    r = ioctl(fd, FOO_IOCTL2);
    assert(r == 0);
}

```

The kernel should respond with

received ioctl number ab01

received ioctl number ab02

The general form of the driver ioctl function could be somewhat like this:

```
static int
foo_ioctl(struct inode *inode, struct file *filp,
 unsigned int cmd, unsigned long arg)
{
    switch(cmd) {
        case FOO_IOCTL1: /* Do some action */
            break;
        case FOO_IOCTL2: /* Do some action */
            break;
        default: return -ENOTTY;
    }
    /* Do something else */
```

```
    return 0;
}
```

We note that the driver ioctl function has a final argument called ‘arg’. Also, the ioctl syscall is defined as:

```
ioctl(int fd, int cmd, ...);
```

This does not mean that ioctl accepts variable number of arguments - but only that type checking is disabled on the last argument. Sometimes, it may be necessary to pass data to the ioctl routine (ie, set the data transfer rate on a communication port) and sometimes it may be necessary to receive back data (get the current data transfer rate). If your intention is to pass finite amount of data to the driver as part of the ioctl, you can pass the last argument as an integer. If you wish to get back some data, you may think of passing a pointer to integer. Whatever be the type which you are passing, the driver routine sees it as an unsigned long - proper type casts should be done in the driver code.

```
static int
foo_ioctl(struct inode *inode, struct file *filp,
 unsigned int cmd, unsigned long arg)
{
    printk("cmd=%x, arg=%x\n", cmd, arg);
    switch(cmd) {
        case FOO_GETSPEED:
            put_user(speed, (int*)arg);
            break;
        case FOO_SETSPEED:
            speed = arg;
            break;
        default: return -ENOTTY; /* Failure */
    }
    return 0; /* Succes */
}
```

Here is the application program which tests this ioctl:

```

main()
{
    int r, speed;
    int fd = open("foo", O_RDWR);
    assert(fd >= 0);

    r = ioctl(fd, FOO_SETSPEED, 9600);
    assert(r == 0);
    r = ioctl(fd, FOO_GETSPEED, &speed);
    assert(r == 0);
    printf("current speed = %d\n", speed);
}
```

When writing production code, it is necessary to use certain macros to generate the ioctl command numbers. The reader should refer *Linux Device Drivers* by Rubini for more information.

7.2. Blocking I/O

A user process which attempts to read from a device should ‘block’ till data becomes ready. A blocked process is said to be in a ‘sleeping’ state - it does not consume CPU cycles. Take the case of the ‘scanf’ function - if you dont type anything on the keyboard, the program which calls it just keeps on sleeping (this can be observed by running ‘ps ax’ on another console). The terminal driver, when it receives an ‘enter’ (or as and when it receives a single character, if the terminal is in raw mode), wakes up all processes which were deep in sleep waiting for input.

Let us see some of the functions used to implement sleep/wakeup mechanisms in Linux. A fundamental datastructure on which all these functions operate on is a wait queue. A wait que is declared as:

```
wait_queue_head_t foo_queue;
```

We have to do some kind of initialization before we use foo_queue. If it is a static(global) variable, we can invoke a macro:

```
DECLARE_WAIT_QUEUE_HEAD(foo_queue);
```

Otherwise, we may call:

```
init_waitqueue_head(&foo_queue);
```

Now, if the process wants to go to sleep, it can call one of many functions, we shall use:

```
interruptible_sleep_on(&foo_queue);
```

Let’s look at an example module.

```
DECLARE_WAIT_QUEUE_HEAD(foo_queue);

static int
foo_open(struct inode* inode, struct file *filp)
{
    if(filp->f_flags == O_RDONLY) {
        printk("Reader going to sleep...\n");
        interruptible_sleep_on(&foo_queue);
    } else if(filp->f_flags == O_WRONLY){
        printk("Writer waking up readers...\n");
        wake_up_interruptible(&foo_queue);
    }
```

```
    return 0; /* Success */
}



```

What happens to a process which tries to open the file 'foo' in read only mode? It immediately goes to sleep. When does it wake up? Only when another process tries to open the file in write only mode. You should experiment with this code by writing two C programs, one which calls open with the O_RDONLY flag and another which calls open with O_WRONLY flag (don't try to use 'cat' - seems that cat opens the file in O_RDONLY|O_LARGEFILE mode).

You should be able to take the first program out of its sleep either by hitting Ctrl-C or by running the second program. What if you change 'interruptible_sleep_on' to 'sleep_on' and 'wake_up_interruptible' to 'wake_up' (wake_up_interruptible wakes up only those processes which have gone to sleep using interruptible_sleep_on whereas wake_up shall wake up all processes). You note that the first program goes to sleep, but you are not able to 'interrupt' it by typing Ctrl-C. Only when you run the program which opens the file 'foo' in writeonly mode does the first program come out of its sleep. Signals are not delivered to processes which are not in interruptible sleep. This is somewhat dangerous, as there is a possibility of creating unkillable processes. Driver writers most often use 'interruptible' sleeps.

7.2.1. wait_event_interruptible

This function is interesting. Let's see what it does through an example.

```
/* Template for a simple driver */

#include <glinux/module.h>
#include <glinux/fs.h>
#include <gasm/uaccess.h>

#define BUFSIZE 1024

static char *name = "foo";
static int major;

static int foo_count = 0;

DECLARE_WAIT_QUEUE_HEAD(foo_queue);


static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{

    wait_event_interruptible(foo_queue, (foo_count == 0));
    printk("Out of read-wait...\n");
    return count;
}

static int
foo_write(struct file* filp, const char *buf,
 size_t count, loff_t *f_pos)
```

```
{
    if(buf[0] == 'I') foo_count++;
    else if(buf[0] == 'D') foo_count--;
    wake_up_interruptible(&foo_queue);
    return count;
}
```

The foo_read method calls wait_event_interruptible, a macro whose second parameter is a C boolean expression. If the expression is true, nothing happens - control comes to the next line. Otherwise, the process is put to sleep on a wait queue. Upon receiving a wakeup signal, the expression is evaluated once again - if found to be true, control comes to the next line, otherwise, the process is again put to sleep. This continues till the expression becomes true.

We write two application programs, one which simply opens ‘foo’ and calls ‘read’. The other program reads a string from the keyboard and calls ‘write’ with that string as argument. If the first character of the string is an upper case ‘I’, the driver routine increments foo_count, if it is a ‘D’, foo_count is decremented. Here are the two programs:

```
main()
{
    int fd;
    char buf[100];
    fd = open("foo", O_RDONLY);
    assert(fd >= 0);
    read(fd, buf, sizeof(buf));
}

/*------Here comes the writer----*/
main()
{
    int fd;
    char buf[100];

    fd = open("foo", O_WRONLY);
    assert(fd >= 0);
    scanf("%s", buf);
    write(fd, buf, strlen(buf));
}
```

Load the module and experiment with the programs. It’s real fun!

7.2.2. A pipe lookalike

Synchronizing the execution of multiple reader and writer processes is no trivial job - our experience in this area is very limited. Here is a small ‘pipe like’ application which is sure to be full of race conditions. The idea is that one process should be able to write to the device - if the buffer is full, the write should block (until the whole buffer becomes free). Another process keeps reading from the device - if the buffer is empty, the read should block till some data is available.

```
#define BUFSIZE 1024

```

```
static char *name = "foo";
static int major;

static char msg[BUFSIZE];
static int readptr = 0, writeptr = 0;

DECLARE_WAIT_QUEUE_HEAD(foo_readq);
DECLARE_WAIT_QUEUE_HEAD(foo_writeq);

static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    int  remaining;

    wait_event_interruptible(foo_readq, (readptr < writeptr));

    remaining = writeptr - readptr;
    if (count <= remaining) {
        if(copy_to_user(buf, msg+readptr, count))
            return -EFAULT;
        readptr = readptr + count;
        wake_up_interruptible(&foo_writeq);
        return count;
    } else {
        if(copy_to_user(buf, msg+readptr, remaining))
                return -EFAULT;
        readptr = readptr + remaining;
        wake_up_interruptible(&foo_writeq);
        return remaining;
    }
}

static int
foo_write(struct file* filp, const char *buf,
 size_t count, loff_t *f_pos)

    int  remaining;

    if(writeptr == BUFSIZE-1) {
        wait_event_interruptible(foo_writeq, (readptr == writeptr));
        readptr = writeptr = 0;
    }
    remaining = BUFSIZE-1-writeptr;
    if (count <= remaining) {
        if(copy_from_user(msg+writeptr, buf, count))
            return -EFAULT;
        writeptr = writeptr + count;
        wake_up_interruptible(&foo_readq);
        return count;
    } else {
        if(copy_from_user(msg+writeptr, buf, remaining))
            return -EFAULT;
        writeptr = writeptr + remaining;
        wake_up_interruptible(&foo_readq);
        return remaining;
    }
```

```
}
```

Chapter 8. Keeping Time

Drivers need to be aware of the flow of time. This chapter looks at the kernel mechanisms available for timekeeping.

8.1. The timer interrupt

Try

```
cat /proc/interrupts
```

This is what we see on our system:

```
           CPU0
  0:     314000          XT-PIC  timer
  1:      12324          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  4:      15155          XT-PIC  serial
  5:         15          XT-PIC  usb-ohci, usb-ohci
  8:          1          XT-PIC  rtc
 11:     212598          XT-PIC  nvidia
 14:       9717          XT-PIC  ide0
 15:         22          XT-PIC  ide1
NMI:          0
LOC:          0
ERR:          0
MIS:          0
```

The first line shows that the ‘timer’ has generated 314000 interrupts from system boot up. The ‘uptime’ command shows us that the system has been alive for around 52 minutes. Which means the timer has interrupted at a rate of almost 100 per second. A constant called ‘HZ’ defined in /usr/src/linux/include/asm/params.h defines this rate.

Every time a timer interrupt occurs, value of a globally visible kernel variable called ‘jiffies’ gets printed(jiffies is initialized to zero during bootup). You should write a simple module which prints the value of this variable. Device drivers are most often satisfied with the granularity which ‘jiffies’ provides.

Drivers seldom need to know the absolute time (that is, the number of seconds elapsed since the ‘epoch’, which is supposed to be 0:0:0 Jan 1 UTC 1970). If you so desire, you can think of calling the

```
void do_gettimeofday(struct timeval *tv);
```

function from your module - which behaves like the ‘gettimeofday’ syscall.

Trying grepping the kernel source for a variable called ‘jiffies’. Why is it declared ‘volatile’?

8.1.1. The perils of optimization

Let's move off track a little bit - we shall try to understand the meaning of the keyword 'volatile'. Let' write a program:

```
#include <signal.h>

int jiffies = 0;
void handler(int n)
{
    printf("called handler...\n");
    jiffies++;
}

main()
{
    signal(SIGINT, handler);
    while(jiffies < 3);
}


```

We define a variable called 'jiffies' and increment it in the handler of the 'interrupt signal'. So, every time you press Ctrl-C, the handler function gets called and jiffies is incremented. Ultimately, jiffies becomes equal to 3 and the loop terminates. This is the behaviour which we observe when we compile and run the program without optimization.

Now what if we compile the program like this:

```
cc a.c -O2
```

we are enabling optimization. If we run the program, we observe that the while loop does not terminate. Why? The compiler has optimized the access to 'jiffies'. The compiler sees that within the loop, the value of 'jiffies' does not change (the compiler is not smart enough to understand that jiffies will change asynchronously) - so it stores the value of jiffies in a CPU register before it starts the loop - within the loop, this CPU register is constantly checked - the memory area associated with jiffies is not at all accessed - which means the loop is completely unaware of jiffies becoming equal to 3 (you should compile the above program with the -S option and look at the generated assembly language code).

What is the solution to this problem? We want the compiler to produce optimized code, but we don't want to mess up things. The idea is to tell the compiler that 'jiffies' should not be involved in any optimization attempts. You can achieve this result by declaring jiffies as:

```
volatile int jiffies = 0;
```

The volatile keyword instructs the compiler to leave alone jiffies during optimization.

8.1.2. Busy Looping

Let's test out this module:

```
static int end;
```

```

static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    static int nseconds = 2;
    char c  = 'A';

    end = jiffies + nseconds*HZ;
    while(jiffies < end)
        ;
    copy_to_user(buf, &c, 1);
    return 1;
}
```

We shall test out this module with the following program:

```
#include "myhdr.h"

main()
{
    char buf[10];
    int fd = open("foo", O_RDONLY);
    assert(fd>=0);
    while(1) {
        read(fd, buf, 1);
        write(1, buf, 1);
    }
}
```

When you run the program, you will see a sequence of 'A's getting printed at about 2 second intervals. What about the response time of your system? It appears as if your whole system has been stuck during the two second delay. This is because the OS is unable to schedule any other job when one process is executing a tight loop in kernel context. Increase the delay and see what effect it has - this exercise should be pretty illuminating. Contrast this behaviour with that of a program which simply executes a tight infinite loop in user mode.

Try timing the above program; run it as

```
time ./a.out
```

how do you interpret the three times shown by the command?

8.2. interruptible_sleep_on_timeout

```
DECLARE_WAIT_QUEUE_HEAD(foo_queue);

static int
```

```
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    static int nseconds = 2;
    char c  = 'A';
    interruptible_sleep_on_timeout(&foo_queue, nseconds*HZ);
    copy_to_user(buf, &c, 1);
    return 1;
}


```

We observe that the process which calls read sleeps for 2 seconds, then prints 'A', again sleeps for 2 seconds and so on. The kernel wakes up the process either when somebody executes an explicit wakeup function on foo_queue or when the specified timeout is over.

8.3. udelay, mdelay

These are busy waiting functions which can be called to implement delays lesser than one timer tick. Eventhough udelay can be used to generate delays upto 1 second, the recommended maximum is 1 milli second. Here are the function prototypes:

#include <linux.h>

void udelay(unsigned long usescs);

void mdelay(unsigned long msecs);

8.4. Kernel Timers

It is possible to 'register' a function so that it is called after a certain time interval. This is made possible through a mechanism called 'kernel timers'. The idea is simple. You create a variable of type 'struct timer_list'

```
struct timer_list{
  struct timer_list *next;
  struct timer_list *prev;
  unsigned long expires; /* Absolute timeout in jiffies */
  void (*fn) (unsigned long); /* timeout function */
  unsigned long data; /* argument to handler function */
  volatile int running;
}

```

The variable is initialized by calling timer_init(). The expires, data and timeout function fields are set. The timer_list object is then added to a global list of timers. The kernel keeps scanning this list 100 times a second, if the current value of 'jiffies' is equal to the expiry time specified in any of the timer objects, the corresponding timeout function is invoked. Here is an example program.

```
DECLARE_WAIT_QUEUE_HEAD(foo_queue);

```

```
void
timeout_handler(unsigned long data)
{
    wake_up_interruptible(&foo_queue);
}

static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    struct timer_list foo_timer;
    char c='B';

    init_timer(&foo_timer);
    foo_timer.function = timeout_handler;
    foo_timer.data = 10;
    foo_timer.expires = jiffies + 2*HZ; /* 2 secs */
    add_timer(&foo_timer);
    interruptible_sleep_on(&foo_queue);
    del_timer_sync(&foo_timer); /* Take timer off the list*/
    copy_to_user(buf, &c, 1);
    return count;
}


```

As usual, you have to test the working of the module by writing a simple application program. Note that the time out function may execute long after the process which caused it to be scheduled vanished. The timeout function is then supposed to be working in ‘interrupt mode’ and there are many restrictions on its behaviour (shouldn’t sleep, shouldn’t access any user space memory etc). It is very easy to lock up the system when you play with such functions (we are speaking from experience!)

8.5. Timing with special CPU Instructions

Modern CPU’s have special purpose Machine Specific Registers associated with them for performance measurement, timing and debugging purposes. There are macro’s for accessing these MSR’s, but let’s take this opportunity to learn a bit of GCC Inline Assembly Language.

8.5.1. GCC Inline Assembly

It may sometimes be convenient (and necessary) to mix assembly code with C. We are not talking of C callable assembly language functions or assembly callable C functions - but we are talking of C code woven around assembly. An example would make the idea clear.

8.5.1.1. The CPUID Instruction

Modern Intel CPU’s (as well as Intel clones) have an instruction called CPUID which is used for gathering information regarding the processor, like, say the vendor id (GenuineIntel or AuthenticAMD). Let’s think of writing a functtion:

char* vendor_id();

which uses the CPUID instruction to retrieve the vendor id. We will obviously have to call the CPUID instruction and transfer the values which it stores in registers to C variables. Let's first look at what Intel has to say about CPUID:

> If the EAX register contains an input value of 0, CPUID returns the vendor identification string in EBX, EDX and ECX registers. These registers will contain the ASCII string 'GenuineIntel'.

Here is a function which returns the vendor id:

```
#include <stdlib.h>

char* vendor_id()
{
    unsigned int p, q, r;
    int i, j;
    char *result = malloc(13*sizeof(char));

    asm("movl $0, %%eax;
        cpuid"
        :"=b"(p), "=c"(q), "=d"(r)
        :
        :"%eax");

    for(i = 0, j = 0; i < 4; i++, j++)
        result[j] = *((char*)&p+i);
    for(i = 0; i < 4; i++, j++)
        result[j] = *((char*)&r+i);
    for(i = 0; i < 4; i++, j++)
        result[j] = *((char*)&q+i);
    result[j] = 0;
    return result;
}

```

How does it work? The template of an inline assembler sequence is:

```
asm(instructions
:output operands
:input operands
:clobbered register list)
```

Except the first (ie, instructions), everything is optional. The real power of inline assembly lies in its ability to operate directly on C variables and expressions. Lets take each line and understand what it does.

The first line is the instruction

```
movl $0, %eax
```

which means copy the immediate value 0 into register eax. The $ and % are merely part of the syntax. Note that we have to write %%eax in the instruction part - it gets translated to %eax (again, there is a reason for this, which we conveniently ignore).

The output operands specify a mapping between C variables (l-values) and CPU registers. "=b"(p) means the C variable ‘p’ is bound to the ebx register. "=c"(q) means variable ‘q’ is bound to the ecx register and "=d"(r) means that the variable ‘r’ is bound to register edx.

We leave the input operands section empty. The clobber list specifies those registers, other than those specified in the output list, which the execution of this sequence of instructions would alter. If the compiler is storing some variable in register eax, it should not assume that that value remains unchanged after execution of the instructions given within the ‘asm’ - the clobberlist thus acts as a warning to the compiler.

So, after the execution of CPUID, the ebx, edx, and ecx registers (each 4 bytes long) would contain the ASCII values of each character of the string AuthenticAMD (our system is an AMD Athlon). Because the variables p, r, q are mapped to these registers, we can easily transfer the ASCII values into a proper null terminated char array.

8.5.2. The Time Stamp Counter

The Intel Time Stamp Counter gets incremented every CPU clock cycle. It’s a 64 bit register and can be read using the ‘rdtsc’ assembly instruction which stores the result in eax (low) and edx (high).

```


main()
{
    unsigned int low, high;

    asm("rdtsc"
        :"=a" (low), "=d"(high));

    printf("%u, %u\n", high, low);
}

```

You can look into */usr/src/linux/include/asm/msr.h* to learn about the macros which manipulate MSR’s.

Chapter 9. Interrupt Handling

We examine how to use the PC parallel port to interface to real world devices. The basics of interrupt handling too will be introduced.

9.1. User level access

The PC printer port is usually located at I/O Port address 0x378. Using instructions like *outb* and *inb* it is possible to write/read data to/from the port.

```
#include <asm/io.h>

#define LPT_DATA 0x378
#define LPT_STATUS 0x379
#define LPT_CONTROL 0x37a

main()
{
    unsigned char c;

    iopl(3);
    outb(0xff, LPT_DATA);
    c = inb(LPT_DATA);
    printf("%x\n", c);
}
```

Before we call *outb/inb* on a port, we must set some kind of *privilege level* by calling the *iopl* instruction. Only the superuser can execute *iopl,* so this program can be executed only by *root*. We are writing hex *ff* to the data port of the parallel interface (there is a status as well as control port associated with the parallel interface). Pin numbers 2 to 9 of the parallel interface are output pins - the result of executing this program will be 'visible' if you connect some LED's between these pins and pin 25 (ground) through a 1KOhm current limiting resistor. All the LED's will light up! (the pattern which we are writing is, in binary 11111111, each bit controls one pin of the port - D0th bit controls pin 2, D1th bit pin 3 and so on). Note that it may sometimes be necessary to compile the program with the *-O* flag to *gcc.*

9.2. Access through a driver

Here is simple driver program which helps us play with the parallel port using Unix commands like *cat, echo, dd* etc.

```
#define LPT_DATA 0x378
#define BUFLEN 1024

static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    unsigned char c;

```

```
    if(count == 0) return 0;
    if(*f_pos == 1) return 0;
    c = inb(LPT_DATA);
    copy_to_user(buf, &c, 1);
    *f_pos = *f_pos + 1;
    return 1;

}

static int
foo_write(struct file* filp, const char *buf,
 size_t count, loff_t *f_pos)
{
    unsigned char s[BUFLEN];
    int i;

    /* Ignore extra data */
    if (count>BUFLEN) count = BUFLEN;
    copy_from_user(s, buf, count);
    for(i = 0; i < count; i++)
        outb(s[i], LPT_DATA);
    return count;
}
```

We load the module and create a device file called ‘led’. Now, if we try:

```
echo -n abcd > led
```

All the characters (ie, ASCII values) will be written to the port, one after the other. If we read back, we should be able to see the effect of the last write, ie, the character ‘d’.

9.3. Elementary interrupt handling

Pin 10 of the PC parallel port is an interrupt intput pin. A low to high transition on this pin will generate Interrupt number 7. But first, we have to enable interrupt processing by writing a 1 to bit 4 of the parallel port control register (which is at BASE+2). Our ‘hardware’ will consist of a piece of wire between pin 2 (output pin) and pin 10 (interrupt input). It is easy for us to trigger a hardware interrupt by making pin 2 go from low to high.

```

#define LPT1_IRQ 7
#define LPT1_BASE 0x378

static char *name = "foo";
static int major;

DECLARE_WAIT_QUEUE_HEAD(foo_queue);

static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
```

```
    static char c = 'a';
    if (count == 0) return 0;
    interruptible_sleep_on(&foo_queue);
    copy_to_user(buf, &c, 1);
    if (c == 'z') c = 'a';
    else c++;
    return 1;
}

void lpt1_irq_handler(int irq, void* data,
 struct pt_regs *regs)
{
        printk("irq: %d triggerred\n", irq);
        wake_up_interruptible(&foo_queue);
}

int init_module(void)
{
    int result;
    major = register_chrdev(0, name, &fops);
    printk("Registered, got major = %d\n", major);
    /* Enable parallel port interrupt */
    outb(0x10, LPT1_BASE+2);
    result = request_irq(LPT1_IRQ, lpt1_irq_handler,
                    SA_INTERRUPT, "foo", 0);
    if (result) {
        printk("Interrupt registration failed\n");
        return result;
    }
    return 0;
}

void cleanup_module(void)
{
    printk("Freeing irq...\n");
    free_irq(LPT1_IRQ, 0);
    printk("Freed...\n");
    unregister_chrdev(major, name);
}
```

Note the arguments to 'request_handler'. The first one is an IRQ number, second is the address of a handler function, third is a flag (SA_INTERRUPT stands for *fast interrupt.* We shall not go into the details), third argument is a name and fourth argument, 0. The function basically *registers* a handler for IRQ 7. When the handler gets called, its first argument would be the IRQ number of the interrupt which caused the handler to be called. We are not using the second and third arguments. In *cleanup_module,* we tell the kernel that we are no longer interested in IRQ 7. The registration of the interrupt handler should really be done only in the foo_open function - and freeing up done when the last process which had the device file open closes it.

It is instructive to examine /proc/interrupts while the module is loaded. You have to write a small application program to trigger the interrupt (make pin 2 low, then high).

```
#include <asm/io.h>
```

```
#define LPT1_BASE 0x378

void enable_int()
{
    outb(0x10, LPT1_BASE+2);
}

void low()
{
    outb(0x0, LPT1_BASE);
}

void high()
{
    outb(0x1, LPT1_BASE);
}

void trigger()
{
    low();
    usleep(1);
    high();
}


main()
{
    iopl(3);
    enable_int();
    while(1) {
            trigger();
            getchar();
    }
}
```

9.3.1. Tasklets and Bottom Halves

The interrupt handler runs with interrupts disabled - if the handler takes too much time to execute, it would affect the performance of the system as a whole. Linux solves the problem in this way - the interrupt routine responds as fast as possible - say it copies data from a network card to a buffer in kernel memory - it then schedules a job to be done later on - this job would take care of processing the data - it runs with interrupts enabled. Task queues and kernel timers can be used for scheduling jobs to be done at a later time - but the preferred mechanism is a *tasklet.*

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>
#include <asm/irq.h>
#include <asm/io.h>

#define LPT1_IRQ 7
```

```
#define LPT1_BASE 0x378

static char *name = "foo";
static int major;
static void foo_tasklet_handler(unsigned long data);

DECLARE_WAIT_QUEUE_HEAD(foo_queue);
DECLARE_TASKLET(foo_tasklet, foo_tasklet_handler, 0);

static int
foo_read(struct file* filp, char *buf,
 size_t count, loff_t *f_pos)
{
    static char c = 'a';
    if (count == 0) return 0;
    interruptible_sleep_on(&foo_queue);
    copy_to_user(buf, &c, 1);
    if (c == 'z') c = 'a';
    else c++;
    return 1;
}

static void foo_tasklet_handler(unsigned long data)
{
    printk("In tasklet...\n");
    wake_up_interruptible(&foo_queue);
}

void lpt1_irq_handler(int irq, void* data,
 struct pt_regs *regs)
{
    printk("irq: %d triggerred, scheduling tasklet\n", irq);
    tasklet_schedule(&foo_tasklet);

}

int init_module(void)
{
    int result;
    major = register_chrdev(0, name, &fops);
    printk("Registered, got major = %d\n", major);
    /* Enable parallel port interrupt */
    outb(0x10, LPT1_BASE+2);
    result = request_irq(LPT1_IRQ, lpt1_irq_handler,
                    SA_INTERRUPT, "foo", 0);
    if (result) {
        printk("Interrupt registration failed\n");
        return result;
    }
    return 0;
}

void cleanup_module(void)
{
    printk("Freeing irq...\n");
    free_irq(LPT1_IRQ, 0);
    printk("Freed...\n");
```

```
    unregister_chrdev(major, name);
}
```

The DECLARE_TASKLET macro takes a tasklet name, a tasklet function and a data value as argument. The *tasklet_schedule* function schedules the tasklet for future execution.

Chapter 10. Accessing the Performance Counters

10.1. Introduction

Modern CPU's employ a variety of dazzling architectural techniques like pipeling, branch prediction etc to achieve great throughput. CPU's from the Intel Pentium onwards (and also the AMD Athlon - not sure about some of the other variants) have some *Machine Specific Registers* associated with them with the help of which we can count architectural events like instruction/data cache hits and misses, pipeline stalls etc. These registers might help us to fine tune our application to exploit architectural quirks to the greatest possible extend (which is not always a good idea).

In this chapter, we develop a simple device driver to retrieve values from certains MSR's called Performance Counters. The code presented will work only on an AMD AthlonXP CPU - but the basic idea is so simple that with the help of the manufacturer's manual, it should be possible to make it work with any other microprocessor (586 and above only).

> **Note:** AMD brings out an *x86 code optimization guide* which was used for writing the programs in this chapter. The *Intel Architecture Software Developer's manual - volume 3* contains detailed description of Intel MSR's as well as code optimization tricks

If you have an interest in computer architecture, you can make use of the code developed here to gain a better understanding of some of the clever engineering tricks which the circuit guys (as well as the compiler designers) employ to get applications running real fast on modern microprocessors.

10.2. The Athlon Performance Counters

The AMD Athlon has four 64 bit performance counters which can be accessed at addresses 0xc0010004 to 0xc0010007 (using two special instructions *rdmsr* and *wrmsr*). Each of these counters can be configured to count a variety of architectural events like data cache access, data cache miss etc using four event select registers at locations 0xc0010000 to 0xc0010003 (one event select register for one event count register).

- Bits D0 to D7 of the event select register select the event to be monitored. For example, if these bits of the event select register at 0xc0010000 is 0x40, the count register at 0xc0010004 will monitor the number of data cache accesses taking place.
- Bit 16, if set, will result in the corresponding count register monitoring events only when the processor is in privilege levels 1, 2 or 3.
- Bit 17, if set, will result in the corresponding count register monitoring events only when the processor is operating at the highest privilege level (level 0).
- Bit 22, when set, will start the event counting process in the corresponding count register.

Let's first look at the header file:

Example 10-1. The perf.h header file

```
/*
 * perf.h
 * A Performance counter library for Linux
 */

#ifdef ATHLON

/* Some IOCTL's */

#define EVSEL 0x10 /* Choose Event Select Register */
#define EVCNT 0x20 /* Choose Event Counter Register */

/* Base address of event select register */
#define EVSEL_BASE 0xc0010000
/* Base address of event count register */
#define EVCNT_BASE 0xc0010004

/* Now, some events to be monitored */

#define DCACHE_ACCESS 0x40
#define DCACHE_MISS 0x41

/* Other selection bits */

#define ENABLE (1U << 22) /* Enable the counter */
#define USR (1U << 16) /* Count user mode event */
#define OS (1U << 17) /* Count OS mode events */


#endif /* ATHLON */


```

Here is the kernel module:

Example 10-2. perfmod.c

```
/*
 * perfmod.c
 * A performance counting module for Linux
 */

#include <linux/module.h>
#include <asm/uaccess.h>
#include <asm/msr.h>
#include <linux/fs.h>

#define ATHLON
#include "perf.h"


char *name = "perfmod";
int major, reg;

```

```
int
perf_ioctl(struct inode* inode, struct file* filp,
     unsigned int cmd, unsigned long val)
{
 switch(cmd){
  case EVSEL:
   reg = EVSEL_BASE + val;
   break;
  case EVCNT:
   reg = EVCNT_BASE + val;
   break;
  }
  return 0;
}

ssize_t
perf_write(struct file *filp, const char *buf,
   size_t len, loff_t *offp)
{
 unsigned int *p = (unsigned int*)buf;
 unsigned int low, high;

 if(len != 2*sizeof(int)) return -EIO;
 get_user(low, p);
 get_user(high, p+1);
 printk("write:low=%x,high=%x. reg=%x\n", low, high, reg);
 wrmsr(reg, low, high);
 return len;
}

ssize_t
perf_read(struct file *filp,  char *buf,
   size_t len, loff_t *offp)
{
 unsigned int *p = (unsigned int*)buf;
 unsigned int low, high;

 if(len != 2*sizeof(int)) return -EIO;
 rdmsr(reg, low, high);
 printk("read:low=%x,high=%x. reg=%x\n", low, high, reg);
 put_user(low, p);
 put_user(high, p+1);
 return len;
}

struct file_operations fops = {
ioctl:perf_ioctl,
read:perf_read,
write:perf_write,
};

int
init_module(void)
{
 major = register_chrdev(0, name, &fops);
 if(major < 0) {
  printk("Error registering device...\n");
```

```
  return major;
 }
 printk("Major = %d\n", major);
 return 0;
}

void
cleanup_module(void)
{
 unregister_chrdev(major, name);
}
```

And here is an application program which makes use of the module to compute data cache misses when reading from a square matrix.

Example 10-3. An application program

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

#define ATHLON
#include "perf.h"
#define SIZE 10000

unsigned char a[SIZE][SIZE];

void initialize()
{
 int i, j, k;

 for(i = 0; i < SIZE; i++)
  for(j = 0; j < SIZE; j++)
   a[i][j] = 0;
}

void action()
{
 int i, j, k;

 for(j = 0; j < SIZE; j++)
  for(i = 0; i < SIZE; i++)
   k = a[i][j];
}


main()
{
 unsigned int count[2] = {0,0}, ev[2];
 int fd = open("perf", O_RDWR);
 int r;
```

```
    assert(fd >= 0);

    /* First, select the event to be
     * monitored
     */

    r = ioctl(fd, EVSEL, 0); /* Event Select 0 */
    assert(r >= 0);

    ev[0] = DCACHE_MISS | USR | ENABLE;
    ev[1] = 0;
    r = write(fd, ev, sizeof(ev));
    assert(r >= 0);

    r = ioctl(fd, EVCNT, 0); /* Select Event Counter 0 */
    assert(r >= 0);

    initialize();

    r = read(fd, count, sizeof(count));
    assert(r >= 0);
    printf("lsb = %x, msb = %x\n", count[0], count[1]);
    printf("Press any key to proceed");
    getchar();
    action();
    r = read(fd, count, sizeof(count));
    assert(r >= 0);
    printf("lsb = %x, msb = %x\n", count[0], count[1]);
}

```

The first *ioctl* chooses event select register 0 as the target of the next read or write. We wish to count data cache misses in user mode, so we set ev[0] properly and invoke a *write*. The next ioctl chooes the event counter register 0 to be the target of subsequent reads or writes. We now initialize the two dimensional array, print the value of event counter register 0, read from the array and then once again display the event counter register.

Note the way in which we are reading the array - we read column by column. This is to generate the maximum number of cache misses. Try the experiment once again with the usual order of array access. You will see a very significant reduction in cache misses.

> **Note:** Caches are there to exploit locality of reference. When we read the very first element of the array (row 0, column 0), that byte, as well as the subsequent 64 bytes are read and stored into the cache. So, if we read the next adjacent 63 bytes, we get cache hits. Instead we are skipping the whole row and are starting at the first element of the next row, which won't be there in the cache.

Chapter 11. A Simple Real Time Clock Driver

11.1. Introduction

How does the PC "remember" the date and time even when you power it off? There is a small amount of battery powered RAM together with a simple oscillator circuit which keeps on ticking always. The oscillator is called a real time clock (RTC) and the battery powered RAM is called the CMOS RAM. Other than storing the date and time, the CMOS RAM also stores the configuration details of your computer (for example, which device to boot from).

The CMOS RAM as well as the RTC control and status registers are accessed via two ports, an address port (0x70) and a data port (0x71). Suppose we wish to access the 0th byte of the 64 byte CMOS RAM (RTC control and status registers included in this range) - we write the address 0 to the address port(only the lower 5 bits should be used) and read a byte from the data port. The 0th byte stores the seconds part of system time in BCD format. Here is an example program which does this.

Example 11-1. Reading from CMOS RAM

```

#include <asm/io.h>

#define ADDRESS_REG 0x70
#define DATA_REG 0x71
#define ADDRESS_REG_MASK 0xe0

#define SECOND 0x00

main()
{
  unsigned char i, j;
  iopl(3);

  i = inb(ADDRESS_REG);
  i = i & ADDRESS_REG_MASK;
  i = i | SECOND;
  outb(i, ADDRESS_REG);
  j = inb(DATA_REG);
  printf("j=%x\n", j);
}
```

11.2. Enabling periodic interrupts

The RTC is capable of generating periodic interrupts at rates from 2Hz to 8192Hz. This is done by setting the PI bit of the RTC Status Register B (which is at address 0xb). The frequency is selected by writing a 4 bit "rate" value to Status Register A (address 0xa) - the rate can vary from 0011 to 1111 (binary). Frequency is derived from rate using the formula *f = 65536/2^rate*. RTC interrupts are reported via IRQ 8.

Here is a program which puts the RTC in periodic interrupt generation mode.

Example 11-2. rtc.c - generate periodic interrupts

```


#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define ADDRESS_REG 0x70
#define DATA_REG 0x71
#define ADDRESS_REG_MASK 0xe0
#define STATUS_A 0x0a
#define STATUS_B 0x0b
#define STATUS_C 0x0c

#define SECOND 0x00

#include "rtc.h"
#define RTC_IRQ 8
#define MODULE_NAME "rtc"

unsigned char
rtc_inb(unsigned char addr)
{
 unsigned char i, j;
 i = inb(ADDRESS_REG);
 /* Clear lower 5 bits */
 i = i & ADDRESS_REG_MASK;
 i = i | addr;
 outb(i, ADDRESS_REG);
 j = inb(DATA_REG);
 return j;
}

void
rtc_outb(unsigned char data, unsigned char addr)
{
 unsigned char i;
 i = inb(ADDRESS_REG);
 /* Clear lower 5 bits */
 i = i & ADDRESS_REG_MASK;
 i = i | addr;
 outb(i, ADDRESS_REG);
 outb(data, DATA_REG);
}

void
enable_periodic_interrupt(void)
{
 unsigned char c;
 c = rtc_inb(STATUS_B);
 /* set Periodic Interrupt enable bit */
```

```
 c = c | (1 << 6);
 rtc_outb(c, STATUS_B);
 /* It seems that we have to simply read
  * this register to get interrupts started.
  * We do it in the ISR also.
  */
 rtc_inb(STATUS_C);
}

void
disable_periodic_interrupt(void)
{
 unsigned char c;
 c = rtc_inb(STATUS_B);
 /* set Periodic Interrupt enable bit */
 c = c & ~(1 << 6);
 rtc_outb(c, STATUS_B);
}

int
set_periodic_interrupt_rate(unsigned char rate)
{
 unsigned char c;
 if((rate < 3) && (rate > 15)) return -EINVAL;
 printk("setting rate %d\n", rate);
 c = rtc_inb(STATUS_A);
 c = c & ~0xf; /* Clear 4 bits LSB */
 c = c | rate;
 rtc_outb(c, STATUS_A);
 printk("new rate = %d\n", rtc_inb(STATUS_A) & 0xf);
 return 0;
}

void
rtc_int_handler(int irq, void *devid, struct pt_regs *regs)
{
 printk("Handler called...\n");
 rtc_inb(STATUS_C);
}

int rtc_init_module(void)
{
 int result;
 result = request_irq(RTC_IRQ, rtc_int_handler,
  SA_INTERRUPT, MODULE_NAME, 0);
 if(result < 0) {
  printk("Unable to get IRQ %d\n", RTC_IRQ);
  return result;
 }
 disable_periodic_interrupt();
 set_periodic_interrupt_rate(15);
 enable_periodic_interrupt();
 return result;
}

void rtc_cleanup(void)
{
```

```
    free_irq(RTC_IRQ, 0);
    return;
}

module_init(rtc_init_module);
module_exit(rtc_cleanup)
```

Your Linux kernel may already have an RTC driver compiled in - in that case you will have to compile a new kernel without the RTC driver - otherwise, the above program may fail to acquire the interrupt line.

11.3. Implementing a blocking read

The RTC helps us play with interrupts without using any external circuits. Suppose we invoke "read" on a device driver - the read method of the driver will transfer data to user space only if some data is available - otherwise, our process should be put to sleep and woken up later (when data arrives). Most peripheral devices generate interrupts when data is available - the interrupt service routine can be given the job of waking up processes which were put to sleep in the read method. We try to simulate this situation using the RTC.

Our read method does not transfer any data - it simply goes to sleep - and gets woken up when an interrupt arrives.

Example 11-3. Implementing blocking read

```

#define ADDRESS_REG 0x70
#define DATA_REG 0x71
#define ADDRESS_REG_MASK 0xe0
#define STATUS_A 0x0a
#define STATUS_B 0x0b
#define STATUS_C 0x0c
#define SECOND 0x00

#define RTC_PIE_ON 0x10 /* Enable Periodic Interrupt */
#define RTC_IRQP_SET 0x20 /* Set periodic interrupt rate */
#define RTC_PIE_OFF 0x30 /* Disable Periodic Interrupt */

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#include "rtc.h"
#define RTC_IRQ 8
#define MODULE_NAME "rtc"
static int major;

DECLARE_WAIT_QUEUE_HEAD(rtc_queue);

```

```
unsigned char
rtc_inb(unsigned char addr)
{
 unsigned char i, j;
 i = inb(ADDRESS_REG);
 /* Clear lower 5 bits */
 i = i & ADDRESS_REG_MASK;
 i = i | addr;
 outb(i, ADDRESS_REG);
 j = inb(DATA_REG);
 return j;
}

void
rtc_outb(unsigned char data, unsigned char addr)
{
 unsigned char i;
 i = inb(ADDRESS_REG);
 /* Clear lower 5 bits */
 i = i & ADDRESS_REG_MASK;
 i = i | addr;
 outb(i, ADDRESS_REG);
 outb(data, DATA_REG);
}

void
enable_periodic_interrupt(void)
{
 unsigned char c;
 c = rtc_inb(STATUS_B);
 /* set Periodic Interrupt enable bit */
 c = c | (1 << 6);
 rtc_outb(c, STATUS_B);
 rtc_inb(STATUS_C); /* Start interrupts! */
}
void
disable_periodic_interrupt(void)
{
 unsigned char c;
 c = rtc_inb(STATUS_B);
 /* set Periodic Interrupt enable bit */
 c = c & ~(1 << 6);
 rtc_outb(c, STATUS_B);
}

int
set_periodic_interrupt_rate(unsigned char rate)
{
 unsigned char c;
 if((rate < 3) && (rate > 15)) return -EINVAL;
 printk("setting rate %d\n", rate);
 c = rtc_inb(STATUS_A);
 c = c & ~0xf; /* Clear 4 bits LSB */
 c = c | rate;
 rtc_outb(c, STATUS_A);
 printk("new rate = %d\n", rtc_inb(STATUS_A) & 0xf);
 return 0;
```

```
}

void
rtc_int_handler(int irq, void *devid, struct pt_regs *regs)
{
 wake_up_interruptible(&rtc_queue);
 rtc_inb(STATUS_C);
}

int
rtc_open(struct inode* inode, struct file *filp)
{
 int result;
 result = request_irq(RTC_IRQ,
  rtc_int_handler, SA_INTERRUPT, MODULE_NAME, 0);
 if(result < 0) {
  printk("Unable to get IRQ %d\n", RTC_IRQ);
  return result;
 }
 return result;
}

int
rtc_close(struct inode* inode, struct file *filp)
{
    free_irq(RTC_IRQ, 0);
 return 0;
}

int
rtc_ioctl(struct inode* inode, struct file* filp,
     unsigned int cmd, unsigned long val)
{
 int result = 0;
 switch(cmd){
  case RTC_PIE_ON:
   enable_periodic_interrupt();
   break;
  case RTC_PIE_OFF:
   disable_periodic_interrupt();
   break;
  case RTC_IRQP_SET:
   result = set_periodic_interrupt_rate(val);
   break;
 }
 return result;
}

ssize_t
rtc_read(struct file *filp,  char *buf,
   size_t len, loff_t *offp)
{
 interruptible_sleep_on(&rtc_queue);
 return 0;
}

struct file_operations fops = {
```

```
 open:rtc_open,
 release:rtc_close,
 ioctl:rtc_ioctl,
 read:rtc_read,
};

int rtc_init_module(void)
{
 major=register_chrdev(0, MODULE_NAME, &fops);
 if(major < 0) {
  printk("Error register char device\n");
  return major;
 }
 printk("major = %d\n", major);
 return 0;
}

void rtc_cleanup(void)
{
    unregister_chrdev(major, MODULE_NAME);
}


module_init(rtc_init_module);
module_exit(rtc_cleanup)
```

Here is a user space program which tests the working of this driver.

Example 11-4. User space test program

```

#include "rtc.h"
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
 int fd, dat, i, r;
 fd = open("rtc", O_RDONLY);
 assert(fd >= 0);

 r = ioctl(fd, RTC_PIE_ON, 0);
 assert(r == 0);
 r = ioctl(fd, RTC_IRQP_SET, 15); /* Freq = 2Hz */
 assert(r == 0);

 for(i = 0; i < 20; i++) {
  read(fd, &dat, sizeof(dat)); /* Blocks for .5 seconds */
  printf("i = %d\n", i);
 }
}
```

11.4. Generating Alarm Interrupts

The RTC can be instructed to generate an interrupt after a specified period. The idea is simple. Locations 0x1, 0x3 and 0x5 should store the second, minute and hour at which the alarm should occur. If the Alarm Interrupt (AI) bit of Status Register B is set, then the RTC will compare the current time (second, minute and hour) with the alarm time each instant the time gets updated. If they match, an interrupt is raised on IRQ 8.

Example 11-5. Generating Alarm Interrupts

```
#define ADDRESS_REG 0x70
#define DATA_REG 0x71
#define ADDRESS_REG_MASK 0xe0
#define STATUS_A 0x0a
#define STATUS_B 0x0b
#define STATUS_C 0x0c

#define SECOND 0x00
#define ALRM_SECOND 0x01
#define MINUTE 0x02
#define ALRM_MINUTE 0x03
#define HOUR 0x04
#define ALRM_HOUR 0x05

#define RTC_PIE_ON 0x10 /* Enable Periodic Interrupt */
#define RTC_IRQP_SET 0x20 /* Set periodic interrupt rate */
#define RTC_PIE_OFF 0x30 /* Disable Periodic Interrupt */
#define RTC_AIE_ON 0x40 /* Enable Alarm Interrupt */
#define RTC_AIE_OFF 0x50 /* Disable Alarm Interrupt */

/* Set seconds after which alarm should be raised */
#define RTC_ALRMSECOND_SET 0x60

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#include "rtc.h"
#define RTC_IRQ 8
#define MODULE_NAME "rtc"
static int major;

DECLARE_WAIT_QUEUE_HEAD(rtc_queue);

int
bin_to_bcd(unsigned char c)
{
 return ((c/10) << 4) | (c % 10);
}
```

```
int
bcd_to_bin(unsigned char c)
{
 return (c >>4)*10 + (c & 0xf);
}

void
enable_alarm_interrupt(void)
{
 unsigned char c;

 printk("Enabling alarm interrupts\n");
 c = rtc_inb(STATUS_B);
 c = c | (1 << 5);
 rtc_outb(c, STATUS_B);
 printk("STATUS_B = %x\n", rtc_inb(STATUS_B));
 rtc_inb(STATUS_C);
}

void
disable_alarm_interrupt(void)
{
 unsigned char c;
 c = rtc_inb(STATUS_B);
 c = c & ~(1 << 5);
 rtc_outb(c, STATUS_B);
}

/* Raise an alarm after nseconds (nseconds <= 59) */
void
alarm_after_nseconds(int nseconds)
{
 unsigned char second, minute, hour;

 second = rtc_inb(SECOND);
 minute = rtc_inb(MINUTE);
 hour = rtc_inb(HOUR);

 second = bin_to_bcd((bcd_to_bin(second) + nseconds) % 60);
 if(second == 0)
  minute = bin_to_bcd((bcd_to_bin(minute)+1) % 60);
 if(minute == 0)
  hour = bin_to_bcd((bcd_to_bin(hour)+1) % 24);

 rtc_outb(second, ALRM_SECOND);
 rtc_outb(minute, ALRM_MINUTE);
 rtc_outb(hour, ALRM_HOUR);
}

rtc_ioctl(struct inode* inode, struct file* filp,
     unsigned int cmd, unsigned long val)
{
 int result = 0;
 switch(cmd){
  case RTC_PIE_ON:
   enable_periodic_interrupt();
   break;
```

```
    case RTC_PIE_OFF:
     disable_periodic_interrupt();
     break;
    case RTC_IRQP_SET:
     result = set_periodic_interrupt_rate(val);
     break;
    case RTC_AIE_ON:
     enable_alarm_interrupt();
     break;
    case RTC_AIE_OFF:
     disable_alarm_interrupt();
     break;
    case RTC_ALRMSECOND_SET:
     alarm_after_nseconds(val);
     break;
  }
  return result;
}
```

Chapter 12. Executing Python Byte Code

12.1. Introduction

> **Note:** The reader is supposed to have a clear idea of the use of the *exec* family of system calls - including the way command line arguments are handled.

Loading and executing a binary file is an activity which requires understanding of the format of the binary file. Binary files generated by compiling a C program on modern Unix systems are stored in what is called ELF format. The binary file header, which is laid out in a particular manner, informs the loader the size of the text and data regions, the points at which they begin, the shared libraries on which the program depends etc. Besides ELF, there can be other binary formats - and there should be a simple mechanism by which the kernel can be extended so that the *exec* function is able to load *any* kind of binary file.

The *exec* system call, which acts as the loader, does not make any attempt to decipher the structure of the binary file - it simply performs some checks on the file (whether the file has execute permission or not), opens it, stores the command line arguments passed to the executable somewhere in memory, reads the first 128 bytes of the file and stores it an a buffer, packages all this information in a structure and passes a pointer to that structure in turn to a series of functions registered with the kernel - each of these functions are responsible for recognizing and loading a particular binary format. A programmer who wants to support a new binary format simply has to write a function which can identify whether the file belongs to the particular format which he wishes to support by examining the first 128 bytes of the file (which the kernel has alread read and stored into a buffer to make our job simpler).

Note that this mechanism is very useful for the execution of scripts. A simple Python script looks like this:

```
#!/usr/bin/python
print 'Hello, World'
```

We can make this file executable and run it by simply typing its name. The exec system call hands over this file to a function registered with the kernel whose job it is to load ELF format binaries - that function examines the first 128 bytes of the file and sees that it is not an ELF file. The kernel then hands over the file to a function defined in `fs/binfmt_script.c`. This function checks the first two bytes of the file and sees the # and the ! symbols. It then extracts the pathname and redoes the program loading process with /usr/bin/python as the file to be loaded and the name of the script file as its argument. Now, because /usr/bin/python is an ELF file, the function registerd with the kernel for handling ELF files will load it successfully.

12.2. Registering a binary format

Let's look at a small program:

Example 12-1. Registering a binary format

```
#include <linux/module.h>
#include <linux/string.h>
#include <linux/stat.h>
#include <linux/slab.h>
#include <linux/binfmts.h>
#include <linux/init.h>
#include <linux/file.h>
#include <linux/smp_lock.h>

static int load_py(struct linux_binprm *bprm,
 struct pt_regs *regs)
{
 printk("pybin load script invoked\n");
 return -ENOEXEC;
}


static struct linux_binfmt py_format = {
 NULL, THIS_MODULE, load_py, NULL, NULL, 0
};

int pybin_init_module(void)
{
 return register_binfmt(&py_format);
}

void pybin_cleanup(void)
{
 unregister_binfmt(&py_format);
 return;
}


module_init(pybin_init_module);
module_exit(pybin_cleanup);
```

Here is the declaration of *struct linux_binfmt*

```
struct linux_binfmt {
 struct linux_binfmt * next;
 struct module *module;
 int (*load_binary)(struct linux_binprm *,
  struct  pt_regs * regs);
 int (*load_shlib)(struct file *);
 int (*core_dump)(long signr,
  struct pt_regs * regs, struct file * file);
 unsigned long min_coredump; /* minimal dump size */
};
```

And here comes *struct linux_binprm*

```
struct linux_binprm{
 char buf[BINPRM_BUF_SIZE];
 struct page *page[MAX_ARG_PAGES];
 unsigned long p; /* current top of mem */
 int sh_bang;
 struct file * file;
```

```
 int e_uid, e_gid;
 kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
 int argc, envc;
 char * filename; /* Name of binary */
 unsigned long loader, exec;
};
```

We initialize the *load_binary* field of *py_format* with the address of the function *load_py*. Once the module is compiled and loaded, we might see the kernel invoking this function when we try to execute programs - which might be because when the kernel scans through the list of registered binary formats, it might encounter *py_format* before it sees the other candidates (like the ELF loader and the #! script loader).

12.3. linux_binprm in detail

Let's first look at the field *buf*. Towards the end of this chapter, we will develop a module which when loaded into the kernel lets us run Python byte code like native code - so we will first look at how a Python program can be compiled into byte code.

If you are using say Python 2.2, you will find a script called `compileall.py` under `/usr/lib/python2.2/`. This script, when run with the name of a directory as argument, compiles all the Python files in it to byte code. We will run this script and compile a simple Python 'hello world' program to byte code. If we examine the first 4 bytes of the byte code file, we will see that they are 45, 237, 13 and 10. We will compile one or two other Python programs and just assume that all Python byte code files start with this signature.

Caution

We are definitely wrong here - consult a Python expert to get the real picture.

Let's modify our module a little bit:

```

int is_python_binary(struct linux_binprm *bprm)
{
 char py_magic[] = {45, 237, 13, 10};
 int i;
 for(i = 0; i < 4; i++)
  if(bprm->buf[i] != py_magic[i]) return 0;
 return 1;
}

static int load_py(struct linux_binprm *bprm,
 struct pt_regs *regs)
{
 int i;
 if(is_python_binary(bprm)) printk("Is Python\n");
 return -ENOEXEC;
}

```

Load this module and try to execute the Python byte code file (first make it executable, then just type its name, preceded by ./). We will see our *load_py* function getting executed. It's obvious that the field *buf* points to a buffer which contains the first few bytes of our file.

We shall now examine the fields *argc* and *filename*. Again, a small modification to our module:

```

static int load_py(struct linux_binprm *bprm,
 struct pt_regs *regs)
{
 int i;
 if(is_python_binary(bprm)) printk("Is Python\n");
 printk("argc = %d, filename = %s\n",
  bprm->argc, bprm->filename);
 return -ENOEXEC;
}

```

It's easy to see that *argc* will contain the number of command line arguments to our executable (including the name of the executable) and *filename* is the file name of the executable. You should be getting messages to that effect when you type any command after loading this module.

12.4. Executing Python Bytecode

We will now make the Linux kernel execute Python byte code. The general idea is this - our load_py function will recognize a Python byte code file - it will then attempt to load the Python interpreter (/usr/bin/python) with the name of the byte code file as argument. The loading of the Python interpreter, which is an ELF file, will of course be done by the kernel module responsible for loading ELF files (`fs/binfmt_elf.c`).

Example 12-2. Executing Python Byte Code

```

static int load_py(struct linux_binprm *bprm,
 struct pt_regs *regs)
{
 int i, retval;
 char *i_name = PY_INTERPRETER;
 struct file *file;
 if(is_python_binary(bprm)) {
  remove_arg_zero(bprm);
  retval = copy_strings_kernel(1, &bprm->filename, bprm);
  if(retval < 0) return retval;
  bprm->argc++;
  retval = copy_strings_kernel(1, &i_name, bprm);
  if(retval < 0) return retval;
  bprm->argc++;
  file = open_exec(i_name);
  if (IS_ERR(file)) return PTR_ERR(file);
  bprm->file = file;
  retval = prepare_binprm(bprm);
  if(retval < 0) return retval;
```

```
  return search_binary_handler(bprm, regs);
 }
 return -ENOEXEC;
}

```

> **Note:** The author's understanding of the code is not very clear - enjoy exploring on your own!

The parameter *bprm*, besides holding pointer to a buffer containing the first few bytes of the executable file, also contains pointers to memory areas where the command line arguments to the program are stored. Lets visualize the command line arguments as being stored one above the other, with the zeroth command line argument (which is the name of the executable) coming last. The function *remove_arg_zero* takes off this argument and decrements the argument count. We then place the name of the byte code executable file (say a.pyc) at this position and the name of the Python interpreter (/usr/bin/python) above it - effectively making the name of the interpreter the new zeroth command line argument and the name of the byte code file the first command line argument (this is the combined effect of the two invocations of *copy_strings_kernel*).

After this, we open /usr/bin/python for execution (open_exec). The *prepare_binprm* function modifies several fields of the structure pointed to by *bprm*, like *buf* to reflect the fact that we are attempting to execute a different file (*prepare_binprm* in fact reads in the first few bytes of the new file and stores it in buf - you should read the actual code for this function). The last step is the invocation of *search_binary_handler* which will once again cycle through all the registered binary formats attempting to load /usr/bin/python. The ELF loader registered with the kernel will succeed in loading and executing the Python interpreter with the name of the byte code file as the first command line argument.

Chapter 13. A simple keyboard trick

13.1. Introduction

All the low level stuff involved in handling the PC keyboard is implemented in `drivers/char/pc_keyb.c`. The keyboard interrupt service routine *keyboard_interrupt* invokes *handle_kbd_event*, which calls *handle_keyboard_event* which in turn invokes *handle_scancode*. By the time handle_scancode is invoked, the scan code (each key will have a scancode, which is distinct from the ASCII code) will be read and all the low level handling completed. We might say that *handle_scancode* forms the interface between the low level keyboard device handling code and the complex upper tty layer.

13.2. An interesting problem

> **Note:** There should surely be an easier way to do this - but let's do it the hard way.

It might sometimes be necessary for us to log in on a lot of virtual consoles as the same user. What if it is possible to automate this process - that is, you log in once, run a program and presto, you are logged in on all consoles. You need to be able to do two things:

- Switch consoles using a program. This is simple. You can apply an *ioctl* on */dev/tty* and switch over to any console. Read the *console_ioctl* manual page to learn more about this.
- Your program should simulate a keyboard and generate some keystrokes (login name and password). This too shouldn't be difficult - we can design a simple driver whose read method will invoke *handle_scancode*

13.2.1. A keyboard simulating module

Here is a program which can be used to simulate keystrokes:

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define MODULE_NAME "skel"
#define MAX 30
#define ENTER 28

/* scancodes of characters a-z */
```

```
static unsigned char scan_codes[] = {
 30, 48, 46, 32, 18, 33, 34, 35, 23, 36, 37,
 38, 50, 49, 24, 25, 16, 19, 31, 20, 22, 47,
 17, 45, 21, 44
};

static char login[MAX], passwd[MAX];
static char login_passwd[2*MAX];

static int major;

/*
 * Split login:passwd into login and passwd
 */
int split(void)
{
 int i;
 char *c, *p, *q;
 c = strchr(login_passwd, ':');
 if (c == NULL) return 0;
 for(p = login_passwd, q = login; p != c; p++, q++)
  *q = *p;
 *q = '\0';
 for(p++, q = passwd; *p ; p++, q++)
  *q = *p;
 *q = '\0';
 return 1;
}


unsigned char
get_scancode(unsigned char ascii)
{
 if((ascii - 'a') >=
 sizeof(scan_codes)/sizeof(scan_codes[0])) {
  printk("Trouble in converting %c\n", ascii);
  return 0;
 }
 return scan_codes[ascii - 'a'];
}


ssize_t
skel_write(struct file *filp, const char *buf,
   size_t len, loff_t *offp)
{
 if(len > 2*MAX) return -ENOSPC;
 copy_from_user(login_passwd, buf, len);
 login_passwd[len] = '\0';
 if(!split()) return -EINVAL;
 printk("login = %s, passwd = %s\n", login, passwd);
 return len;
}

ssize_t
skel_read(struct file *filp,  char *buf,
   size_t len, loff_t *offp)
```

```
{
 int i;
 unsigned char c;
 if(*offp == 0) {
  for(i = 0; login[i]; i++) {
   c = get_scancode(login[i]);
   if(c == 0) return 0;
   handle_scancode(c, 1);
   handle_scancode(c, 0);
  }
  handle_scancode(ENTER, 1);
  handle_scancode(ENTER, 0);
  *offp = 1;
  return 0;
 }

 for(i = 0; passwd[i]; i++) {
  c = get_scancode(passwd[i]);
  if(c == 0) return 0;
  handle_scancode(c, 1);
  handle_scancode(c, 0);
 }
 handle_scancode(ENTER, 1);
 handle_scancode(ENTER, 0);
 *offp = 0;
 return 0;
}

struct file_operations fops = {
 read:skel_read,
 write:skel_write,
};

int skel_init_module(void)
{
 major=register_chrdev(0, MODULE_NAME, &fops);
 printk("major=%d\n", major);
 return 0;
}

void skel_cleanup(void)
{
    unregister_chrdev(major, MODULE_NAME);
    return;
}

module_init(skel_init_module);
module_exit(skel_cleanup)
```

The working of the module is fairly straightforward. We first invoke the *write* method and give it a string of the form *login:passwd*. Now, suppose we invoke *read*. The method will simply generate scancodes corresponding to the characters in the login name and deliver those scancodes to the upper tty layer via *handle_scancode* (we call it twice for each character - once to simulate a key depression and the other to simulate a key release). Another read will deliver scancodes corresponding to the password. Whatever program is running on the currently active console will receive these simulated keystrokes.

Once we compile and load this module, we can create a character special file. We might then run:

```
echo -n 'luser:secret' > foo
```

so that a login name and password is registered within the module. The next step is to run a program of the form:

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

void login(void);

main(int argc, char **argv)
{
 int fd, start, end;

 assert(argc == 3);
 start = atoi(argv[1]);
 end = atoi(argv[2]);

 fd = open("/dev/tty", O_RDWR);
 assert(fd >= 0);
 for(; start <= end; start++) {
  ioctl(fd, VT_ACTIVATE, start);
  usleep(10000);
  login();
 }

}

void login(void)
{
 int  fd, i;

 fd = open("foo", O_RDONLY);
 assert(fd >= 0);
 read(fd, &i, sizeof(i));
 usleep(10000);
 read(fd, &i, sizeof(i));
 close(fd);
}
```

The program simply cycles through the virtual consoles (start and end numbers supplied from the commandline) every time invoking the *login* function which results in the driver *read* method getting triggerred.

Chapter 14. Network Drivers

14.1. Introduction

This chapter presents the facilities which the Linux kernel offers to Network Driver writer's. As usual, we see that developing a toy driver is simplicity itself; if you are looking to write a professional quality driver, you will soon have to start digging into the kernel source.

Alessandro Rubini and Jonathan Corbet present a lucid explanation of Network Driver design in their *Linux Device Drivers (2nd Edition)* . Our machine independent driver is a somewhat simplified form of the *snull* interface presented in the book.

You miss a lot of fun (or frustration) when you leave out real hardware from the discussion - those of you who are prepared to handle a soldering iron would sure love to make up a simple serial link and test out the "silly" SLIP implementation of this chapter.

It is expected that the reader is familiar with the basics of TCP/IP networking - *TCP/IP Illustrated* and *Unix Network Programming* by W.Richard Stevens are two standard references which you should consult (the first two or three chapters would be sufficient) before reading this document.

14.2. Linux TCP/IP implementation

The Linux kernel implements the TCP/IP protocol stack - the source can be found under `/usr/src/linux/net/ipv4`. It is possible to divide the networking code into two parts - one which implements the actual protocols (the `net/ipv4` directory) and the other which implements device drivers for a bewildering array of networking hardware - mostly various kinds of ethernet cards (found under `drivers/net`)

The kernel TCP/IP code is written in such a way that it is very simple to "slide in" drivers for any kind of real (or virtual) communication channel without bothering too much about the functioning of the network or transport layer code. The "layering" which all TCP/IP text books talk of has very real practical benefits as it makes it possible for us to enhance the functionality of a part of the protocol stack without disturbing large areas of code.

14.3. Configuring an Interface

The *ifconfig* command is used for manipulating network interfaces. Here is what the command displays on my machine:

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

This machine does not have any real networking hardware installed - but we do have a pure software interface - a so called "loopback interface". The interface is assigned an IP address of 127.0.0.1.

It is possible to bring the interface down by running *ifconfig lo down*. Once the interface is down, *ifconfig* will not display it in it's output. But it is possible to obtain information about inactive interfaces by running *ifconfig -a* . It is possible make the interface active once again (you guessed it - *ifconfig lo up*) - it's also possible to assign a different IP address - *ifconfig lo 127.0.0.2*.

Before an interface can be manipulated with *ifconfig*, it is necessary that the driver code for the interface is loaded into the kernel. In the case of the loopback interface, the code is compiled into the kernel. Usually, it would be stored as a module and inserted into the kernel whenever required by running commands like *modprobe*. Here is what I do to get the driver code for an old NE2000 ISA card into the kernel:

```
ifconfig ne.o io=0x300
```

Writing a network driver and thus creating your own interface requires that you have some idea of:

- Kernel data structures and functions which form the interface between the device driver and the protocol layer on top.
- The hardware of the device which you wish to control. Networking interfaces like the Ethernet make use of interrupts and DMA to perform data transfer and are as such not suited for newbies to cut their teeth on. A simple device like the serial port should do the job.

14.4. Driver writing basics

Our first attempt would be to design a hardware independent driver - this will help us to examine the kernel data structures and functions involved in the interaction between the driver and the upper layer of the protocol stack. Once we get the "big picture", we can look into the nitty-gritty involved in the design of a real hardware-based driver.

14.4.1. Registering a new driver

When we write character drivers, we begin by "registering" an object of type *struct file_operations*. A similar procedure is followed by network drivers also - but there is one major difference - a character driver is accessible from user space through a special device file entry which is not the case with network drivers. We shall examine this difference in detail, but first, a small program.

Example 14-1. Registering a network driver

```
#include <linux/config.h>
#include <linux/module.h>

#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
```

```
#include <linux/types.h>
#include <linux/string.h>
#include <linux/socket.h>
#include <linux/errno.h>
#include <linux/fcntl.h>
#include <linux/in.h>
#include <linux/init.h>
#include <linux/ip.h>
#include <asm/system.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/in6.h>
#include <asm/checksum.h>

#include <linux/inet.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/skbuff.h>
#include <net/sock.h>
#include <linux/if_ether.h> /* For the statistics structure. */
#include <linux/if_arp.h> /* For ARPHRD_SLIP */

int mydev_init(struct net_device *dev)
{
 printk("mydev_init...\n");
 return(0);
}

struct net_device mydev = {init: mydev_init};

int mydev_init_module(void)
{
 int result, i, device_present = 0;
 strcpy(mydev.name, "mydev");
 if ((result = register_netdev(&mydev))) {
  printk("mydev: error %d registering device %s\n",
   result, mydev.name);
  return result;
 }
 return 0;
}

void mydev_cleanup(void)
{
 unregister_netdev(&mydev) ;
 return;
}

module_init(mydev_init_module);
module_exit(mydev_cleanup);

```

The *net_device*structure has a role to play similar to the *file_operations* structure for character drivers. Note that we are filling up only two entries, *init* and *name*. We then "register" this object with the kernel by calling *register_netdev*, which will, besides doing a lot of other things, call the function pointed to by *mydev.init*, passing it as argument the address of *mydev*. Our *mydev_init* simply prints a message.

Here is part of the output from *ifconfig -a* once this module is loaded:

```
mydev     Link encap:AMPR NET/ROM  HWaddr
          [NO FLAGS]  MTU:0  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

ifconfig is getting some information about our device through members of the *struct net_device* object which we have registered with the kernel - most of the members are left uninitialized, we will see the effect of initialization when we run the next example.

Example 14-2. Initalizing the net_device object

```
int mydev_open(struct net_device *dev)
{
 MOD_INC_USE_COUNT;
  printk("Open called\n");
 netif_start_queue(dev);
 return 0;
}

int mydev_release(struct net_device *dev)
{
 printk("Release called\n");
 netif_stop_queue(dev); /* can't transmit any more */
 MOD_DEC_USE_COUNT;
 return 0;
}

static int mydev_xmit(struct sk_buff *skb, struct net_device *dev)
{
 printk("dummy xmit function called...\n");
 dev_kfree_skb(skb);
 return 0;
}

int mydev_init(struct net_device *dev)
{
 printk("loop_init...\n");
 dev->open = mydev_open;
 dev->stop = mydev_release;
 dev->mtu = 1000;
 dev->hard_start_xmit = mydev_xmit;
 dev->type = ARPHRD_SLIP;
 dev->flags = IFF_NOARP;
 return(0);
}

```

In the case of character drivers, we perform a static, compile time initialization of the *file_operations* object. The *net_device* object is used for holding function pointers as well as device specific data associated with the interface devices, say the hardware address in the case of Ethernet cards. It would be possible to fill in this information only by calling probe routines when the driver is loaded into memory and not when it is compiled.

We initialize the *open* field with the address of a routine which gets invoked when we activate the interface using the *ifconfig* command - the routine announces the readiness of the driver to accept data by calling *netif_start_queue*. The release routine is invoked when the interface is brought down. The Maximum Transmission Unit (MTU) associated with the device is the largest chunk of data which the interface is capable of transmitting as a whole - this information may be used by the higher level protocol layer to break up large data packets. The device type should be initialized to one of the many standard types defined in `include/linux/if_arp.h`.

The *hard_start_xmit* field requires special mention - it holds the address of the routine which is central to our program. We shall come to it after we load this module and play with it a bit.

```
[root@localhost stage1]# insmod  -f ./mydev.o
Warning: loading ./mydev.o will taint the kernel: no license
Warning: loading ./mydev.o will taint the kernel: forced load
loop_init...
[root@localhost stage1]# ifconfig mydev 192.9.200.1
Open called
[root@localhost stage1]# ifconfig
mydev     Link encap:Serial Line IP
          inet addr:192.9.200.1  Mask:255.255.255.0
          UP RUNNING NOARP  MTU:1000  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

[root@localhost stage1]# ifconfig mydev down
Release called
[root@localhost stage1]#
```

We see the effect of initializing the MTU, device type etc in the output of *ifconfig*. We use ifconfig to attach an IP address to our interface, at which time the *mydev_open* function gets called.

Now, for an interesting experiment. We write a small Python script:

Example 14-3. A Python program to send a "hello" to a remote machine

```
from socket import *
fd = socket(AF_INET, SOCK_DGRAM)
fd.sendto("hello", ("192.9.200.2", 7000))
```

You need not be a Python expert to understand that the program simply opens a UDP socket and tries to send a "hello" to a process running at port number 7000 on the machine 192.9.200.2. Needless to say, the "hello" won't go very far because such a machine does not exist! But we observe something interesting - the *mydev_xmit* function has been triggerred, and it has printed the message *dummy xmit function called...*!

How has this happened? The application program tells the UDP layer that it wants to send a "hello". UDP is happy to service the request - our message gets a UDP header attached to it and is driven down the protocol stack to the next lower layer - which is IP. The IP layer attaches its own header and then checks the destination address, which is 192.9.200.2.

There should be some registered interface on our machine the network id portion of whose IP address matches the net id portion of the address 192.9.200.2 (The network id portion is the first three bytes, that is 192.9.200 - the reader should look up some text book on networking and get to know the different IP addressing schemes). Our *mydev* interface, whose address is 192.9.200.1 is chosen to be the one to transmit the data to 192.9.200.2. The kernel simply calls the *mydev_xmit* function of the interface through the *mydev.start_hard_xmit* pointer, passing it as argument the data to be transmitted.

But what's that *struct sk_buff *skb* stuff which is passed as the first argument to *mydev_xmit*? The "socket buffer" is one of the most important data structures in the whole of the TCP/IP networking code in the Linux kernel. Simply put, it holds lots of control information plus the data being shuttled to and fro between the protocol layers - the data can be accessed as *skb->data*. Note that when we say "data", we refer to the actual data (which is the message "hello") plus the headers introduced by each protocol layer. In the next section, we examine sk_buff's a bit more in detail.

14.4.2. The sk_buff structure

We examine only one field of the sk_buff structure, which is *data*. The network layer code calls the *mydev_xmit* routine with the address of an sk_buff object as argument. The *data* field of the structure will point to a buffer whose initial few bytes would be the IP header, the next few bytes the UDP header and the remaining bytes, the actual data (the string "hello").

Example 14-4. Examining the IP header attached to skb->data

```
static int mydev_xmit(struct sk_buff *skb, struct net_device *dev)
{
 struct iphdr *iph;
 printk("dummy xmit function called...\n");
 iph = (struct iphdr*)skb->data;
 printk("saddr = %x, daddr = %x\n", ntohl(iph->saddr), ntohl(iph->daddr));
 dev_kfree_skb(skb);
 return 0;
}
```

The *iphdr* structure is defined in the file `include/linux/ip.h`. It contains two unsigned 32 bit fields called *saddr* and *daddr* which are the source and destination IP addresses respectively. Because the header stores these in big endian format, we convert that to the host format by calling *ntohl*. Once the module with this modified mydev_xmit is loaded and the interface is assigned an IP address, we can run the Python script once again. We will see the message:

```
saddr = c009c801, daddr = c009c802
```

The sk_buff object is created at the top of the protocol stack - it then journey's downward, gathering control information and data as it passes from layer to layer. Ultimately, it reaches the hands of the driver whose responsibility it is to despatch the data through the physical communication channel. Our transmit function has chosen not to send the data anywhere. But it has the responsibility of freeing up space consumed by the object as its prescence is no longer required in the system. That's what *dev_free_skb* does.

14.4.3. Towards a meaningful driver

It should be possible for us to transmit as well as receive data through a network interface. What we have seen till now is the transmission part - we have seen how data journey’s from the application layer (our Python program) and ultimately reaches the hands of the device driver packaged within an sk_buff. The driver can send the data out through some kind of communication hardware. The device driver program sitting at the other end receives the data (using some hardware tricks which we are not yet ready to examine) - but it’s job is not finished. It has to make sure that whatever application program is waiting for the data actually gets it. How is this done?

Let’s first look at an application program running on a machine with an interface bound to 192.9.200.2.

Example 14-5. Python program waiting for data

```
from socket import *
fd = socket(AF_INET, SOCK_DGRAM)
fd.bind((’192.9.200.2’, 7000))
s = fd.recvfrom(100)
```

The program is waiting for data packets with destination ip address equal to 192.9.200.2 and destination port number equal to 7000. Imagine the transport layer and the network layer being a pair of consumer - producer processes with a "shared queue" in between them. Think of the same relation as holding true between the network layer and the physical layer also. Now, the *recvfrom* system call scans the queue connecting the transport/network layer checking for data packets with destination port number equal to 7000. If it doesn’t see any such packet, it goes to sleep, at the same time notifying the kernel that it should be woken up in case some such packet arrives.

Let’s see what the device driver can do now. The driver has received a sequence of bytes over the "wire". The first step is to create an sk_buff structure and copy the data bytes to *skb->data*. Now the address of this sk_buff object can be given to the network layer (say, by putting it on a queue and passing a message that the que has got to be scanned). The network layer code gets the data bytes, removes the IP header, does plenty of "magic" and once convinced that the data is actually addressed to this machine (as opposed to simply stopping over during a long journey) puts it on the queue between itself and the transport layer - at the same time notifying the transport layer code that some data has arrived. The transport layer code knows which all processes are waiting for data to arrive on which all ports - so if it sees a packet with destination port number equal to 7000, it wakes up our Python program and gives it that packet.

Let’s think of applying this idea to a situation where we don’t really have a hardware communication channel. We register two interfaces - one called *mydev0* and the other one called *mydev1*. The interfaces are exactly identical. We assign the address 192.9.200.1 to mydev0 and 192.9.201.2 to mydev1. Now let’s suppose that we are trying to send a string "hello" to 192.9.200.2. The kernel will choose the interface with IP address 192.9.200.1 for transmitting the message - the data packet (including actual data + UDP/IP headers) will ultimately be given to the *mydev_xmit* routine of interface *mydev0*. Now here comes a nifty trick (thanks to Rubini and Corbet!). The transmit routine will toggle the least significant bit of the 3rd byte of both source and destination IP addresses on the data packet and will simply place it on the upward-bound queue linking the physical and network layer! The IP layer is fooled into believing that a packet has arrived from 192.9.201.1 to 192.9.201.2. An application program which is waiting for data over the 192.9.201.2 interface will soon come out of its sleep and receive this data. Similar is the case if you try to transmit data to say 192.9.201.1. The network layer will believe that data has arrived from 192.9.200.2 to 192.9.200.1.

Let's look at the code for this little driver.

Example 14-6. mydev0 and mydev1

```
static int mydev_xmit(struct sk_buff *skb, struct net_device *dev)
{
 struct iphdr *iph;
 struct sk_buff *skb2;
 unsigned char *saddr, *daddr;
 int len;
 short int protocol;

 len = skb->len;
 protocol = skb->protocol;

 skb2 = dev_alloc_skb(len+2);
 if(!skb2) {
  printk("low on memory...\n");
  return 0;
 }
 memcpy(skb_put(skb2, len), skb->data, skb->len);
 skb2->dev = dev;
 skb2->protocol = protocol;
 skb2->ip_summed = CHECKSUM_UNNECESSARY;

 dev_kfree_skb(skb);

 iph = (struct iphdr*)skb2->data;
 if(!iph){
  printk("data corrupt...\n");
  return 0;
 }
 saddr = (unsigned char *)(&(iph->saddr));
 daddr = (unsigned char *)(&(iph->daddr));
 saddr[2] = saddr[2] ^ 0x1;
 daddr[2] = daddr[2] ^ 0x1;

 iph->check = 0;
 iph->check = ip_fast_csum((unsigned char*)iph, iph->ihl);

 netif_rx(skb2);

 return 0;
}

int mydev_init(struct net_device *dev)
{
 printk("mydev_init...\n");
 dev->open = mydev_open;
 dev->stop = mydev_release;
 dev->mtu = 1000;
 dev->hard_start_xmit = mydev_xmit;
 dev->type = ARPHRD_SLIP;
 dev->flags = IFF_NOARP;
```

```
 return(0);
}

struct net_device mydev[2]= {{init: mydev_init}, {init:mydev_init}};

int mydev_init_module(void)
{

    int result, i, device_present = 0;

    strcpy(mydev[0].name, "mydev0");
    strcpy(mydev[1].name, "mydev1");
    if ((result = register_netdev(&mydev[0]))) {
            printk("mydev: error %d registering device %s\n",
                   result, mydev[0].name);
     return result;
    }
    if ((result = register_netdev(&mydev[1]))) {
            printk("mydev: error %d registering device %s\n",
                   result, mydev[1].name);
     return result;
    }
    return 0;
}

void mydev_cleanup(void)
{
    unregister_netdev(&mydev[0]) ;
    unregister_netdev(&mydev[1]) ;
    return;
}

module_init(mydev_init_module);
module_exit(mydev_cleanup)
```

Here are some hints for understanding the transmit routine:

- The *skb->len* field contains total length of the packet (including actual data + the headers).
- *dev_alloc_skb(len)*will create an sk_buff object and allocate enough space in it to hold a packet of size *len.*
- The sk_buff object gets shuttled up and down the protocol stack. During this journey, it may be necessary to add to the already existing data area either in the beginning or in the end. The *dev_alloc_skb* function, when called with an argument say "M", will create an sk_buff object with M bytes buffer space. When we call *skb_put(skb, L)*, the function will mark the first L bytes of the buffer as being used - it will also return the address of the first byte of this L byte block. Now suppose we are calling *skb_reserve(skb, N)* before we call skb_put. The function will mark the first N bytes of the M byte buffer as being "reserved". After this, skb_put(skb, L) will mark L bytes starting from the the N'th byte as being used; the starting address of this block will also be returned. Another skb_put(skb, P) will mark the P byte block after this L byte block as being reserved. An skb_push(skb, R) will mark off an R byte block aligned at the end of the first N byte block as being in use.

- We are creating a new sk_buff object and copying the data in the first sk_buff object to the second. Besides copying the data, certain control information should also be copied (for use by the upper protocol layers). For example, when the sk_buff object is handed over to the network layer, we let the layer know that the data is IP encapsulated by copying skb->protocol.
- We recompute the checksum because the source/destination IP addresses have changed.
- The *netif_rx* function does the job of passing the sk_buff object up to the higher layer.

14.4.4. Statistical Information

You have observed that *ifconfig*displays the number of received/transmitted packets, total number of bytes received/transmitted etc. For our interface, these numbers have remained constant at zero - we haven't been tracking these things. Let's do it now.

The *net_device* structure contains a "private" pointer field, which can be used for holding information. We will allocate an object of type *struct net_device_stats* and store it address in the private data area. As and when we receive/transmit data, we will update certain fields of this structure. When ifconfig wants to get statistical information about the interface, it will call a function whose address is stored in the *get_stats* field of the net_device object. This function should simply return the address of the net_device_stats object which holds the statistical information.

Example 14-7. Getting Statistical information

```
static int mydev_xmit(struct sk_buff *skb, struct net_device *dev)
{
 struct net_device_stats *stats;

 /* Transmission code deleted */

 stats = (struct net_device_stats*)dev->priv;
 stats->tx_bytes += len;
 stats->rx_bytes += len;
 stats->tx_packets++;
 stats->rx_packets++;

 netif_rx(skb2);
 return 0;
}

struct net_device_stats *get_stats(struct net_device *dev)
{
 return (struct net_device_stats*)dev->priv;
}

int mydev_init(struct net_device *dev)
{
 /* Code deleted */
 dev->priv = kmalloc(sizeof(struct net_device_stats), GFP_KERNEL);
 if(dev->priv == 0) return -ENOMEM;
 memset(dev->priv, 0, sizeof(struct net_device_stats));
```

```
 dev->get_stats = get_stats;
 return(0);
}
```

14.5. Take out that soldering iron

> **Caution**
>
> Linus talks of the days when men were men and wrote their own device drivers. To get real thrill out of this section, you have to go back to those days when real men made their own serial cables (even if one could be purchased from the hardware store)!
>
> That said, we are not to be held responsible for personal injuries arising out of amateurish use of soldering irons - or damages to your computer arising out of incorrect hardware connections.

We have seen how to build a sort of "loopback" network interface where no communication hardware actually exists and data transfer is done purely through software. With some very simple modifications, we would be to make our code transmit data through a serial cable. We choose the serial port as our communication hardware because it is the simplest interface available.

14.5.1. Setting up the hardware

Get yourself two 9 pin connectors and some cable. The pins on the serial connector are numbered. Pin 2 is receive, 3 is transmit and 5 is ground. Join Pin 5 of both connectors with a cable (this is our common ground). Pin 2 of one connector should be joined with Pin 3 of the other and vice versa (this forms our RxT and TxR connections). Thats all!

14.5.2. Testing the connection

Two simple user space C programs can be used to test the connections:

Example 14-8. Program to test the serial link - transmitter

```

#define COM_BASE 0x3F8 /* Base address for COM1 */
main()
{
 /* This program is the transmitter */
 int i;
 iopl(3); /* User space code needs this
    * to gain access to I/O space.
    */
 while(1) {
```

```
  for(i = 0; i < 10; i++) {
   outb(i, COM_BASE);
   sleep(1);
  }
 }
}
```

The program should be compiled with the -O option and should be executed as the superuser.

Example 14-9. Program to test the serial link - receiver

```

#define COM_BASE 0x3F8 /* Base address for COM1 */
#define STATUS COM_BASE+5

main()
{
 /* This program is the transmitter */
 int c;
 iopl(3); /* User space code needs this
    * to gain access to I/O space.
    */
 while(1) {
  while(!(inb(STATUS)&0x1));
  c = inb(COM_BASE);
  printf("%d\n", i);
 }
}
```

The LSB of the STATUS register becomes 1 when a new data byte is received. Our program will keep on looping till this bit becomes 1.

> **Note:** This example might not work always. The section below tells you why.

14.5.3. Programming the serial UART

PC serial communication is done with the help of a hardware device called the UART. Before we start sending data, we have to initialize the UART telling it the number of data bits which we are using, number of parity/stop bits, speed in bits per second etc. In the above example, we assume that the operating system would initialize the serial port and that the parameters would be same at both the receiver and the transmitter.

Let’s first look `uart.h`

Example 14-10. Header file containing UART specific stuff

```

#ifndef __UART_H
#define __UART_H

#define COM_BASE 0x3f8
#define COM_IRQ 4

#define LCR (COM_BASE+3) /* Line Control Register */
#define DLR_LOW COM_BASE /* Divisor Latch Register */
#define DLR_HIGH (COM_BASE+1)
#define SSR (COM_BASE+5) /* Serialization status register */
#define IER (COM_BASE+1) /* Interrupt enable register */
#define MCR (COM_BASE+4) /* Modem Control Register */
#define OUT2 3
#define TXE 6 /* Transmitter hold register empty */
#define BAUD 9600

#include <asm/io.h>

static inline unsigned char recv_char(void)
{
 return inb(COM_BASE);
}

static inline void send_char(unsigned char c)
{
 outb(c, COM_BASE);
 /* Wait till byte is transmitted */
 while(!(inb(SSR) & (1 << TXE)));
}
#endif
```

The *recv_char* routine would be called from within an interrupt handler - so we are sure that data is ready - we need to just take it off the UART. But our *send_char* method has been coded without using interrupts (which is NOT a good thing). So we have to write the data and then wait till we are sure that a particular bit in the status register, which indicates the fact that transmission is complete, is set.

Before we do any of these things, we have to initialize the UART.

Example 14-11. uart.c - initializing the UART

```
#include "uart.h"
#include <asm/io.h>

void uart_init(void)
{
 unsigned char c;
 outb(0x83, LCR); /* DLAB set, 8N1 format */
 outb(0xc, DLR_LOW);
 outb(0x0, DLR_HIGH); /* We set baud rate = 9600 */
 outb(0x3, LCR); /* We clear DLAB bit */
 c = inb(IER);
 c = c | 0x1;
```

```
 outb(c, IER); /* Receive interrupt set */

 c = inb(MCR);
 c = c | (1 << OUT2);
 outb(c, MCR);
 inb(COM_BASE); /* Clear any interrupt pending flag */
}
```

We are initializing the UART in 8N1 format (8 data bits, no parity and 1 stop bit). We set the baud rate by writing a divisor value of decimal 12 (the divisor "x" is computed using the expression 115200/x = baud rate) to a 16 bit Divisor Latch Register accessed as two independent 8 bit registers. Then we enable interrupts by setting specific bits of the Interrupt Enable Register and the Modem Control Register. The reader may refer a book on PC hardware to learn more about UART programming. As of now, it would no harm to consider *uart_init* to be a "black box" which initializes the UART in 8N1 format, 9600 baud and enables serial port interrupts.

14.5.4. Serial Line IP

We now examine a simple "framing" method for serial data. As the serial hardware is very simple and does not impose any kind of "packet structure" on data, it is the responsibility of the transmitting program to let the receiver know where a chunk of data begins and where it ends. The simplest way would be to place two "marker" bytes at the beginning and end. Let's call these marker bytes END. But what if the data stream itself contains a marker byte? The receiver might interpret that as an end-of-packet marker. To prevent this, we encode a literal END byte as two bytes, an ESC followed by an ESC_END. Now what if the data stream contains an ESC byte? We encode it as two bytes, ESC followed by another special byte, ESC_ESC. This simple encoding scheme is explained in *RFC 1055: A nonstandard for transmission of IP datagrams over serial lines* which the reader should read before proceeding any further with this section.

Example 14-12. slip.c - SLIP encoding and decoding

```
#include "uart.h"
#include "slip.h"

void send_packet(unsigned char *p, int len)
{
 send_char(END);
 while(len--) {
  switch(*p) {
   case END: send_char(ESC);
      send_char(ESC_END);
      break;
   case ESC: send_char(ESC);
      send_char(ESC_ESC);
      break;
   default:
      send_char(*p);
      break;
  }
  p++;
 }
```

```
 send_char(END);
 #ifdef DEBUG
 printk("at end of send_packet...\n");
 #endif
}

/* rev_packet is called only from an interrupt. We
 * structure it as a simple state machine.
 */
void recv_packet(void)
{
 unsigned char c;

 c = recv_char();
 #ifdef DEBUG
 printk("in recv_packet...\n");
 #endif
 if (c == END) {
  state = DONE;
  return;
 }

 if (c == ESC) {
  state = IN_ESC;
  return;
 }

 if (state == IN_ESC) {
  if (c == ESC_ESC) {
   state = OUT_ESC;
   slip_buffer[tail++] = ESC;
   return;
  }
  if (c == ESC_END) {
   state = OUT_ESC;
   slip_buffer[tail++] = END;
   return;
  }
 }

 slip_buffer[tail++] = c;
 state = OUT_ESC;
}
```

The *send_packet* function simply performs SLIP encoding and transmits the resulting sequence over the serial line (without using interrupts). *recv_packet* is more interesting. It is called from within the serial interrupt service routine and its job is to read and decode individual bytes of SLIP encoded data and let the interrupt service routine know when a full packet has been decoded.

Example 14-13. slip.h - contains SLIP byte definitions

```
#ifndef __SLIP_H
#define __SLIP_H

#define END 0300
```

```
#define ESC 0333
#define ESC_END 0334
#define ESC_ESC 0335
#define SLIP_MTU 1006

enum {DONE, IN_ESC, OUT_ESC};

void send_packet(unsigned char*, int);
void recv_packet(void);

extern unsigned char slip_buffer[];
extern int state;
extern int tail;

#endif
```

14.5.5. Putting it all together

The design of our network driver is very simple - the tranmit routine will simply call *send_packet*. The serial port interrupt service routine will decode and assemble a packet from the wire by invoking *receive_packet*. The decoded packet will be handed over to the upper protocol layers by calling netif_rx.

Example 14-14. mydev.c - the actual network driver

```

#include "uart.h"
#include "slip.h"

int state = DONE; /* Initial state of the UART receive machine */
unsigned char slip_buffer[SLIP_MTU];
int tail = 0; /* Index into slip_buffer */

int mydev_open(struct net_device *dev)
{
 MOD_INC_USE_COUNT;
  printk("Open called\n");
 netif_start_queue(dev);
 return 0;
}

int mydev_release(struct net_device *dev)
{
 printk("Release called\n");
 netif_stop_queue(dev); /* can't transmit any more */
 MOD_DEC_USE_COUNT;
 return 0;
}

static int mydev_xmit(struct sk_buff *skb, struct net_device *dev)
{
 #ifdef DEBUG
```

```
 printk("mydev_xmit called, len = %d...\n", skb->len);
 #endif
 send_packet(skb->data, skb->len);
 dev_kfree_skb(skb);
 return 0;
}

void uart_int_handler(int irq, void *devid, struct pt_regs *regs)
{
 struct sk_buff *skb;
 struct iphdr *iph;

 recv_packet();
 #ifdef DEBUG
 printk("after receive packet...\n");
 #endif
 if((state == DONE) && (tail != 0)) {
  #ifdef DEBUG
  printk("within if: tail = %d...\n", tail);
  #endif
  skb = dev_alloc_skb(tail+2);
  if(skb == 0) {
   printk("Out of memory in dev_alloc_skb...\n");
   return;
  }
  skb->protocol = 8;
  skb->dev = (struct net_device*)devid;
  skb->ip_summed = CHECKSUM_UNNECESSARY;
  memcpy(skb_put(skb, tail), slip_buffer, tail);
  tail = 0;
  #ifdef DEBUG
  iph = (struct iphdr*)skb->data;
  printk("before netif_rx:saddr = %x, daddr = %x...\n",
   ntohl(iph->saddr), ntohl(iph->daddr));
  #endif
  netif_rx(skb);
 }
 #ifdef DEBUG
 printk("leaving isr...\n");
 #endif
}

int mydev_init(struct net_device *dev)
{
 printk("mydev_init...\n");
 dev->open = mydev_open;
 dev->stop = mydev_release;
 dev->mtu = SLIP_MTU;
 dev->hard_start_xmit = mydev_xmit;
 dev->type = ARPHRD_SLIP;
 dev->flags = IFF_NOARP;
 return(0);
}

struct net_device mydev = {init: mydev_init};

int mydev_init_module(void)
```

```
{

    int result, i, device_present = 0;

    strcpy(mydev.name, "mydev");
    if ((result = register_netdev(&mydev))) {
            printk("mydev: error %d registering device %s\n",
                    result, mydev.name);
     return result;
    }
    result = request_irq(COM_IRQ, uart_int_handler,
   SA_INTERRUPT, "myserial", (void*)&mydev);
    if(result) {
     printk("mydev: error %d could not register irq %d\n",
  result, COM_IRQ);
     return result;
    }
    uart_init();
    return 0;
}

void mydev_cleanup(void)
{
    unregister_netdev(&mydev) ;
    free_irq(COM_IRQ, 0);
    return;
}


module_init(mydev_init_module);
module_exit(mydev_cleanup)
```

Note: The use of *printk* statements within interrupt service routines can result in the code going haywire - may be because they take up lots of time to execute (we are running with interrupts disabled) - and we might miss a few interrupts - especially if we are communicating at a very fast rate.

Chapter 15. The VFS Interface

15.1. Introduction

Modern Unix like operating systems have evolved very sophisticated mechanisms to support myriads of file systems - the so called VFS or the *Virtual File System Switch* is at the heart of Unix file management. We will try our best to get some idea of how the VFS layer can be used to implement file systems in this chapter.

Note: The reader is expected to have some idea of how Operating Systems store data on disks - general concepts about MS-DOS FAT or Linux Ext2 (things like super block, inode table etc) together with an understanding of file/directory handling system calls should be sufficient.

The Design of the Unix Operating System by *Maurice J Bach* is a good place to start. *Understanding the Linux Kernel* by *Daniel P. Bovet* and *Marco Cesati* would be the next logical step - just spend four or five hours reading the chapter on the VFS again and again and again... Then look at the implementations of *ramfs* and *procfs*. The *Documentation/filesystems* directory under the Linux kernel source tree root contains a file *vfs.txt* which provides useful information.

15.1.1. Need for a VFS layer

Different Operating Systems have evolved different strategies for laying out data on the tracks and sectors of a physical storage device - say a floppy, hard disk, CD ROM, flash memory etc. Linux is capable of reading a floppy which stores data in say the MS-DOS FAT format. Once the floppy is mounted, user programs need not bother about whether the device is DOS formatted or not - they can carry on with reading and writing - with the full assurance that whatever they write would be ultimately laid out on the floppy in such a way that MS-DOS would be able to read it. The important point here is that the operating system is designed in such a way that file handling system calls like *read, write* are coded so as to be completely independent of the data structures residing on the disk. These system calls basically interact with a large and complex body of code nicknamed the VFS - the VFS maintains a list of "registered" file systems - each filesystem in its simplest sense being a set of routines whose job it is to translate the data handed over by the system calls to its ultimate representation on the physical storage device. A programmer can think up a custom file format of his own and hook it up with the VFS - he can then mount this filesystem and use it just like the native ext2 format.

15.1.2. In-core and on-disk data structures

The VFS layer mostly manipulates in-core (ie, stored in RAM) representations of on-disk data structures. This has got some very interesting implications. The Unix system call *stat* is used for retrieving information like size, date, ownership, permissions etc of the file. *stat* assumes that these informations are stored in an in-core data structure called the *inode*. Now, some file systems like Linux's native *ext2* have the concept of a disk resident inode which stores administrative information regarding files. Simpler systems, like the MS-DOS FAT have no equivalent disk resident "inode", nor does it have any concept of "ownership" or "permissions" associated with files or directories (DOS does have a very minor idea of "permissions" which is not at all comparable to that of modern multiuser operating systems - so we can ignore that). Now, the VFS layer, upon receiving a stat call from userland, invokes some routines loaded into the kernel as part of registering the DOS filesystem - these routines on the fly generate an inode data structure mostly filled with "bogus" information - and a bit of real information (say size, date - the real information can be retreived only from the storage media - which the DOS specific routines do). With a little bit of imagination, it shouldn't be difficult to visualize the VFS magician fooling the rest of the kernel and userland programs into believing that random data, which need not even be stored on any secondary storage device, does in fact look like a directory tree. Look at `fs/proc/` for a good example.

The major in-core data structures associated with the VFS are:

- The super block structure - holds an in memory image of certain fields of the file system superblock. A file system like the ext2 which physically resides on a disk will have a few blocks of data in the beginning itself dedicated to storing statistics global to the file system as a whole.
- The inode structure - this is the in-memory copy of the inode, which contains information pertaining to files and directories (like size, permissions etc).
- The dentry (directory entry) structure. Directory entries are cached by the operating system (in the dentry cache) to speed up all operations involving path lookup. A file system which does not reside on a secondary storage device (like the *ramfs*) needs only to create a dentry structure and an inode structure, store the inode pointer in the dentry structure, increment a usage count associated with the dentry structure and add it to the dentry cache to get the effect of "creating" a directory entry. We shall examine this a bit more in detail when we look at the ramfs code.
- The *file* structure. This basically relates a process with an open file. As an example, a process may open the same file multiple times and read from (or write to) it. The process will be using multiple file descriptors (say fd1 and fd2). We visualize fd1 and fd2 as pointing to two different file structures - with both the file structures having the same inode pointer. Each of the file structures will have its own offset field, which indicates the offset in the file to which a write (or read) should take effect.

15.1.3. The Big Picture

- The application program invokes a system call with the pathname of a file (or directory) as argument.
- The kernel internally associates each mount point with a valid, registered filesystem. Certain file manipulation system calls satisfy themselves purely by manipulating VFS data structures (like the in-core inode or the in-core directory entry structure) - if no valid instance of such a data structure is found, the VFS layer invokes a routine specific to the filesystem which fills in the in-core data structures. Certain other system calls result in functions registered with the filesystem getting called immediately.

15.2. Experiments

We shall try to understand the working of the VFS by carrying out some simple experiments.

15.2.1. Registering a file system

Example 15-1. Registering a file system

```

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/pagemap.h>
#include <linux/init.h>
#include <linux/string.h>
#include <linux/locks.h>
#include <asm/uaccess.h>

#define MYFS_MAGIC 0xabcd12
#define MYFS_BLKSIZE 1024
#define MYFS_BLKBITS 10

struct inode *
myfs_get_inode(struct super_block *sb, int mode, int dev)
{
 struct inode * inode = new_inode(sb);

 printk("myfs_get_inode called...\n");
 if (inode) {
  inode->i_mode = mode;
  inode->i_uid = current->fsuid;
  inode->i_gid = current->fsgid;
  inode->i_blksize = MYFS_BLKSIZE;
  inode->i_blocks = 0;
  inode->i_rdev = NODEV;
  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
 }
 return inode;
}

static struct super_block *
myfs_read_super(struct super_block * sb, void * data, int silent)
{
 struct inode * inode;
 struct dentry * root;

 printk("myfs_read_super called...\n");
 sb->s_blocksize = MYFS_BLKSIZE;
 sb->s_blocksize_bits = MYFS_BLKBITS;
 sb->s_magic = MYFS_MAGIC;
 inode = myfs_get_inode(sb, S_IFDIR | 0755, 0);
 if (!inode)
  return NULL;
 root = d_alloc_root(inode);
 if (!root) {
  iput(inode);
```

```
  return NULL;
 }
 sb->s_root = root;
 return sb;
}

static
DECLARE_FSTYPE(myfs_fs_type, "myfs", myfs_read_super, FS_LITTER);

static int  init_myfs_fs(void)
{
 return register_filesystem(&myfs_fs_type);
}

static void exit_myfs_fs(void)
{
 unregister_filesystem(&myfs_fs_type);
}

module_init(init_myfs_fs)
module_exit(exit_myfs_fs)
MODULE_LICENSE("GPL");

```

- The macro DECLARE_FSTYPE creates a variable myfs_fs_type of type *struct file_system_type* and initializes a few fields. Of these, the *read_super* field is perhaps the most important. It is initialized to *myfs_read_super* which is a function that gets called when this filesystem is mounted - the job of this function is to fill up an object of type *struct super_block* (which would be partly filled by the VFS itself) either by reading an actual super block residing on the disk, or by simply assigning some values.
- myfs_read_super gets invoked at mount time - it gets as argument a partially filled super_block object. It's job is to fill up some other important fields.
  - The file system block size is filled up in number of bytes as well as number of bits required for addressing
  - An inode structure is allocated and filled up. The inode number (which is a field within the inode structure) will be some arbitrary value - which is not a problem as our inode does not map on to a real inode on the disk.
  - A dentry structure (which is used for caching directory entries to speed up path lookups) is created and the inode pointer is stored in it (a dentry object should contain an inode pointer, if it is to represent a real directory entry - dentry objects which do not have an inode pointer assigned to them are called "negative" dentries.) The super block structure is made to hold a pointer to the dentry object.
- The myfs_read_super function returns the address of the filled up super_block object.

How do we "mount" this filesystem? First, we compile and insert this module into the kernel (say as myfs.o). Then,

```
#mount -t myfs none foo
```

The mount command accepts a -t argument which specifies the file system type to mount, then an argument which indicates the device on which the file system is stored (because we have no such device here, this argument can be some random string) and the last argument, the directory on which to mount.

Try changing over to the directory *foo*. Also, run the *ls* command on *foo*. These don't work - our attempt would be to make them work.

15.2.2. Associating inode operations with a directory inode

We have been able to mount our file system onto a directory - but we have not been able to change over to the directory - we get an error message "Not a directory". We wish to find out why this error message is coming. A bit of searching around the VFS source leads us to line number 621 in `fs/namei.c`

```
if (lookup_flags & LOOKUP_DIRECTORY) {
 err = -ENOTDIR;
 if (!inode->i_op || !inode->i_op->lookup)
  break;
}
```

Aha - that's the case. Our root directory inode (remember, we had created an inode as well as a dentry and registered it with the file system superblock - that is the "root inode" of our file system) needs a set of inode operations associated with it - the set should contain at least the *lookup* function. Now, what is this *inode operation*?

System calls like *create, link, unlink, mkdir, rmdir* etc which act on a directory allways invoke a registered *inode operation function* - these are the functions which do file system specific work related to creating, deleting and manipulating directory entries. Once we associate a set of inode operations with our root directory inode, we would be able to make the kernel accept it as a "valid" directory. This is what we proceed to do in the next program.

Example 15-2. Associating inode operations

```

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/pagemap.h>
#include <linux/init.h>
#include <linux/string.h>
#include <linux/locks.h>
#include <asm/uaccess.h>

#define MYFS_MAGIC 0xabcd12
#define MYFS_BLKSIZE 1024
#define MYFS_BLKBITS 10

static struct dentry*
myfs_lookup(struct inode* dir, struct dentry *dentry)
{
  printk("lookup called...\n");
  return NULL;
}

```

```
struct inode_operations
myfs_dir_inode_operations = {lookup:myfs_lookup};

struct inode *myfs_get_inode(struct super_block *sb, int mode, int dev)
{
 struct inode * inode = new_inode(sb);

 printk("myfs_get_inode called...\n");
 if (inode) {
  inode->i_mode = mode;
  inode->i_uid = current->fsuid;
  inode->i_gid = current->fsgid;
  inode->i_blksize = MYFS_BLKSIZE;
  inode->i_blocks = 0;
  inode->i_rdev = NODEV;
  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
 }
 switch(mode & S_IFMT) {
  case S_IFDIR: /* Directory inode */
   inode->i_op = &myfs_dir_inode_operations;
   break;
 }
 return inode;
}

static struct super_block *
myfs_read_super(struct super_block * sb, void * data, int silent)
{
 struct inode * inode;
 struct dentry * root;

 printk("myfs_read_super called...\n");
 sb->s_blocksize = MYFS_BLKSIZE;
 sb->s_blocksize_bits = MYFS_BLKBITS;
 sb->s_magic = MYFS_MAGIC;
 inode = myfs_get_inode(sb, S_IFDIR | 0755, 0);
 if (!inode)
  return NULL;
 root = d_alloc_root(inode);
 if (!root) {
  iput(inode);
  return NULL;
 }
 sb->s_root = root;
 return sb;
}

static DECLARE_FSTYPE(myfs_fs_type, "myfs", myfs_read_super, FS_LITTER);

static int  init_myfs_fs(void)
{
 return register_filesystem(&myfs_fs_type);
}

static void exit_myfs_fs(void)
{
 unregister_filesystem(&myfs_fs_type);
```

```
}

module_init(init_myfs_fs)
module_exit(exit_myfs_fs)
MODULE_LICENSE("GPL");
```

It should be possible for us to mount the filesystem onto a directory and change over to it. An *ls* would not generate any error, but it will report no directory entries. We will rectify the situation - but before that, we will examine the role of the *myfs_lookup* function a little bit in detail.

15.2.3. The lookup function

Let's modify the lookup function a little bit.

Example 15-3. A slightly modified lookup

```

static struct dentry*
myfs_lookup(struct inode* dir, struct dentry *dentry)
{
  printk("lookup called...");
  printk("searching for file %s ", dentry->d_name.name);
  printk("under directory whose inode is %d\n", dir->i_ino);
  return NULL;
}

```

As usual, build and load the module and mount the "myfs" filesystem on a directory say *foo*. If we now type *ls foo* , nothing happens. But if we type *ls foo/abc*, we see the following message getting printed on the screen:

```
lookup called...searching for file abc under directory
whose inode is 3619
```

If we run the *strace* command to find out the system calls which the two different invocations of ls produce, we will see that:

- *ls tmp* basically calls *getdents* which is a sytem call for reading the directory contents as a whole.
- *ls tmp/abc* invokes the *stat* system call, which is used for exploring the contents of the inode of a file.

The *getdents* call is mapped to a particular function in the file system which has not been implemented - so it does not yield any output. But the *stat* system call tries to identify the inode associated with the file `tmp/abc`. In the process, it first searches the directory entry cache (dentry cache). A dentry will contain the name of a directory entry, a pointer to its associated inode and lots of other info. If the file name is not found in the dentry cache, the system call will invoke an inode operation function associated with the root inode of our filesystem (in our case, the myfs_lookup function) passing it as argument the inode pointer associated with the directory under which the search is to be performed together with a partially filled dentry which will contain the name of the file to be searched (in our case, `abc`). The job of the lookup function is to search the directory (the directory may be physically stored on a disk) and if the file exists, store its inode pointer in the required field of the partially filled dentry structure. The dentry structure may then be added to the dentry cache so that future lookups are satisfied from the cache itself.

In the next section, we will modify lookup further - our objective is to make it cooperate with some other inode operation functions.

15.2.4. Creating a file

We move on to more interesting stuff. We wish to be able to create zero byte files under our mount point.

Example 15-4. Adding a "create" routine

```

struct inode *
myfs_get_inode(struct super_block *sb, int mode, int dev);

static struct dentry*
myfs_lookup(struct inode* dir, struct dentry *dentry)
{
  printk("lookup called...\n");
  d_add(dentry, NULL);
  return NULL;
}

static int
myfs_mknod(struct inode *dir,
  struct dentry *dentry, int mode, int dev)
{
 struct inode * inode = myfs_get_inode(dir->i_sb, mode, dev);
 int error = -ENOSPC;
 printk("myfs_mknod called...\n");

 if (inode) {
  d_instantiate(dentry, inode);
  dget(dentry);
  error = 0;
 }
 return error;
}

static int
myfs_create(struct inode *dir, struct dentry *dentry, int mode)
{
 printk("myfs_create called...\n");
 return myfs_mknod(dir, dentry, mode | S_IFREG, 0);
}

static struct inode_operations
myfs_dir_inode_operations = {
 lookup:myfs_lookup,
```

```
 create:myfs_create,
};

static struct file_operations
myfs_dir_operations = {
 readdir:dcache_readdir
};

struct inode *
myfs_get_inode(struct super_block *sb, int mode, int dev)
{
 struct inode * inode = new_inode(sb);

 printk("myfs_get_inode called...\n");
 if (inode) {
  inode->i_mode = mode;
  inode->i_uid = current->fsuid;
  inode->i_gid = current->fsgid;
  inode->i_blksize = MYFS_BLKSIZE;
  inode->i_blocks = 0;
  inode->i_rdev = NODEV;
  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
 }
 switch(mode & S_IFMT) {
  case S_IFDIR: /* Directory inode */
   inode->i_op = &myfs_dir_inode_operations;
   inode->i_fop = &myfs_dir_operations;
   break;
 }
 return inode;
}
```

The *creat*system call ultimately invokes a file system specific create routine. Before that, it searches the dentry cache for the file which is being created - if the file is not found, the lookup routine *myfs_lookup* is invoked(as explained earlier) - it simply stores the value of zero in the inode field of the dentry object and adds it to the dentry cache (this is what d_add does). Because lookup has not been able to associate a valid inode with the dentry, it is assumed that the file does not exist and hence, a file system specific create routine, *myfs_create* is invoked. This routine, by calling *myfs_mknod*, first creates an inode, then associates the inode with the dentry object and increments a "usage count" associated with the dentry object (this is what dget does). The net effect is that:

- We have a dentry object which holds the name of the new file.
- We have an inode, and this inode is associated with the dentry object
- The dentry object is on the dcache
- We are associating an object of type *struct file_operations* through the *i_fop* field of the inode. The *readdir* field of this structure contains a pointer to a standard function called *dcache_readdir*
- Whenever a user program invokes the *readdir* or *getdents* syscall to read the contents of a directory, the VFS layer invokes the function whose address is stored in the *readdir* field of the structure pointed to by the *i_fop* field of the inode. The standard func-

tion *dcache_readdir* prints out all the directory entries corresponding to the root directory present in the dentry cache. Because an invocation of*myfs_create* always results in the file-name being added to the dentry and the dentry getting stored in the dcache, we have a sort of "pseudo directory" which is maintained by the VFS data structures alone.

We are now able to create zero byte files, either by using commands like *touch* or by writing a C program which calls the *open* or *creat* system call. We are also able to list the files. But what if we try to read from or write to the files? We see that we are not able to do so. The next section rectifies this problem.

15.2.5. Implementing read and write

Example 15-5. Implementing read and write

```
static ssize_t
myfs_read(struct file* filp, char *buf, size_t count,
    loff_t *offp)
{
  printk("myfs_read called...");
  printk("but not reading anything...\n");
  return 0;
}

static ssize_t
myfs_write(struct file *fip, const char *buf,
    size_t count, loff_t *offp)
{
  printk("myfs_write called...");
  printk("but not writing anything...\n");
  return count;
}

static struct file_operations
myfs_file_operations = {
 read:myfs_read,
 write:myfs_write
};

struct inode *myfs_get_inode(struct super_block *sb, int mode, int dev)
{
 struct inode * inode = new_inode(sb);

 printk("myfs_get_inode called...\n");
 if (inode) {
  inode->i_mode = mode;
  inode->i_uid = current->fsuid;
  inode->i_gid = current->fsgid;
  inode->i_blksize = MYFS_BLKSIZE;
  inode->i_blocks = 0;
  inode->i_rdev = NODEV;
  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
 }
```

```
 switch(mode & S_IFMT) {
  case S_IFDIR: /* Directory */
   inode->i_op = &myfs_dir_inode_operations;
   inode->i_fop = &myfs_dir_operations;
   break;
  case S_IFREG: /* Regular file */
   inode->i_fop = &myfs_file_operations;
   break;
 }
 return inode;
}
```

The important additions are:

- We are associating an object *myfs_file_operations* with the inode for a regular file. This object contains two methods, *read* and *write*. When we apply a read system call on an ordinary file, the read method of the file operations object associated with the inode of that file gets invoked. The prototype of the read and write methods are the same as what we have seen for character device drivers. Our read method simply prints a message and returns zero, the application program which attempts to read the file thinks that it has seen end of file and terminates. Similarly, the write method simply returns the count which it gets as argument, the program invoking the writing being fooled into believing that it has written all the data.

We are now able to run commands like *echo hello > a* and *cat a* on our file system without errors - eventhough we are not reading or writing anything.

15.2.6. Modifying read and write

We create a 1024 byte buffer in our module. A write to any file would write to this buffer. A read from any file would read from this buffer.

Example 15-6. Modified read and write

```
static char data_buf[MYFS_BLKSIZE];
static int data_len;

static ssize_t
myfs_read(struct file* filp, char *buf, size_t count,
    loff_t *offp)
{
  int remaining = data_len - *offp;
  printk("myfs_read called...");
  if(remaining <= 0) return 0;
  if(count > remaining) {
   copy_to_user(buf, data_buf + *offp, remaining);
   *offp += remaining;
   return remaining;
  }else{
   copy_to_user(buf, data_buf + *offp, count);
   *offp += count;
```

```
   return count;
  }
}

static ssize_t
myfs_write(struct file *fip, const char *buf,
    size_t count, loff_t *offp)
{
  printk("myfs_write called...\n");
  if(count > MYFS_BLKSIZE) {
   return -ENOSPC;
  } else {
   copy_from_user(data_buf, buf, count);
   data_len = count;
   return count;
  }
}
```

Note that the write always overwrites the file - with a little more effort, we could have made it better - but the idea is to demonstrate the core idea with a minimum of complexity.

Try running commands like *echo hello > a* and *cat a*. What would be the result of running:

```
dd if=/dev/zero of=abc bs=1025 count=1
```

15.2.7. A better read and write

It would be nice if read and write would work as they normally would - each file should have its own private data storage area. Thats what we aim to do with the following program. The inode structure has a filed called "u" which contains a void* field called *generic_ip*. This field can be used to store info private to each file system. We make this field store a pointer to our file's data block.

Example 15-7. A better read and write

```

static ssize_t
myfs_read(struct file* filp, char *buf, size_t count,
    loff_t *offp)
{
  char *data_buf = filp->f_dentry->d_inode->u.generic_ip;
  int data_len = filp->f_dentry->d_inode->i_size;
  int remaining = data_len - *offp;
  printk("myfs_read called...");
  if(remaining <= 0) return 0;
  if(count > remaining) {
   copy_to_user(buf, data_buf + *offp, remaining);
   *offp += remaining;
   return remaining;
  }else{
   copy_to_user(buf, data_buf + *offp, count);
```

```
        *offp += count;
        return count;
    }
}

static ssize_t
myfs_write(struct file *filp, const char *buf,
        size_t count, loff_t *offp)
{
    char *data_buf =
    filp->f_dentry->d_inode->u.generic_ip;

    printk("myfs_write called...\n");
    if(count > MYFS_BLKSIZE) {
     return -ENOSPC;
    } else {
     copy_from_user(data_buf, buf, count);
     filp->f_dentry->d_inode->i_size = count;
     return count;
    }
}

struct inode *
myfs_get_inode(struct super_block *sb, int mode, int dev)
{
 struct inode * inode = new_inode(sb);

 printk("myfs_get_inode called...\n");
 if (inode) {
  inode->i_mode = mode;
  inode->i_uid = current->fsuid;
  inode->i_gid = current->fsgid;
  inode->i_blksize = MYFS_BLKSIZE;
  inode->i_blocks = 0;
  inode->i_rdev = NODEV;
  inode->i_atime =
  inode->i_mtime = inode->i_ctime = CURRENT_TIME;
 }
 switch(mode & S_IFMT) {
  case S_IFDIR:
   inode->i_op = &myfs_dir_inode_operations;
   inode->i_fop = &myfs_dir_operations;
   break;
  case S_IFREG:
   inode->i_fop = &myfs_file_operations;
   inode->i_size = 0;
   /* Have to check return value of kmalloc, lazy  */
   inode->u.generic_ip = kmalloc(MYFS_BLKSIZE, GFP_KERNEL);
   break;
 }
 return inode;
}

```

15.2.8. Creating a directory

The Unix system call *mkdir* is used for creating directories. This in turn calls the inode operation *mkdir*.

Example 15-8. Implementing mkdir

```
static int
myfs_mkdir(struct inode* dir, struct dentry *dentry,
  int mode)
{
 return myfs_mknod(dir, dentry, mode|S_IFDIR, 0);
}

struct inode_operations
myfs_dir_inode_operations = {
 lookup:myfs_lookup,
 create:myfs_create,
 mkdir:myfs_mkdir
};
```

15.2.9. A look at how the dcache entries are chained together

Each dentry contains two fields of type *list_head*, one called *d_subdirs* and the other one called *d_child*. If the dentry is that of a directory, its d_subdirs field will be linked to the d_child field of one of the files (or directories) under it. The d_child field of that file (or directory) will be linked to the d_child field of a sibling (files or directories whose parent is the same) and so on.

Here is a program which prints all the siblings of a file when that file is read:

Example 15-9. Examining the way dentries are chained together

```
void
print_string(const char *str, int len)
{
 int i;
 printk("print_string called, len = %d\n", len);
 for(i = 0; str[i]; i++)
   printk("%c", str[i]);
 printk("\n");
}

void
print_siblings(struct dentry *dentry)

{
 struct dentry *parent = dentry->d_parent;
 struct list_head *start = &parent->d_subdirs, *head;
 struct dentry *sibling;

 for(head=start; start->next != head; start = start->next) {
  sibling = list_entry(start->next, struct dentry, d_child);
```

```
  print_string(sibling->d_name.name, sibling->d_name.len);
 }
}

static ssize_t
myfs_read(struct file* filp, char *buf, size_t count,
    loff_t *offp)
{
  char *data_buf = filp->f_dentry->d_inode->u.generic_ip;
  int data_len = filp->f_dentry->d_inode->i_size;
  int remaining = data_len - *offp;
  printk("myfs_read called...");
  print_siblings(filp->f_dentry);
  if(remaining <= 0) return 0;
  if(count > remaining) {
   copy_to_user(buf, data_buf + *offp, remaining);
   *offp += remaining;
   return remaining;
  }else{
   copy_to_user(buf, data_buf + *offp, count);
   *offp += count;
   return count;
  }
}
```

15.2.10. Implementing deletion

The *unlink* and *rmidr* syscalls are used for deleting files and directories - this in turn results in a file system specific unlink or rmdir getting invoked.

Example 15-10. Deleting files and directories

```
static inline int myfs_positive(struct dentry *dentry)
{
 printk("myfs_positive called...\n");
 return dentry->d_inode && !d_unhashed(dentry);
}

/*
 * Check that a directory is empty (this works
 * for regular files too, they'll just always be
 * considered empty..).
 *
 * Note that an empty directory can still have
 * children, they just all have to be negative..
 */
static int myfs_empty(struct dentry *dentry)
{
 struct list_head *list;

 printk("myfs_empty called...\n");

 spin_lock(&dcache_lock);
```

```
 list = dentry->d_subdirs.next;

 while (list != &dentry->d_subdirs) {
  struct dentry *de = list_entry(list, struct dentry, d_child);

  if (myfs_positive(de)) {
   spin_unlock(&dcache_lock);
   return 0;
  }
  list = list->next;
 }
 spin_unlock(&dcache_lock);
 return 1;
}

/*
 * This works for both directories and regular files.
 * (non-directories will always have empty subdirs)
 */
static int myfs_unlink(struct inode * dir, struct dentry *dentry)
{
 int retval = -ENOTEMPTY;
 printk("myfs_unlink called...\n");

 if (myfs_empty(dentry)) {
  struct inode *inode = dentry->d_inode;

  inode->i_nlink--;
  if(inode->i_nlink == 0) {
   printk("Freeing space...\n");
   if((inode->i_mode & S_IFMT) == S_IFREG)
    kfree(inode->u.generic_ip);
  }
  dput(dentry);
  /* Undo the count from "create" - this does all the work */
  retval = 0;
 }
 return retval;
}

#define myfs_rmdir myfs_unlink

static struct inode_operations
myfs_dir_inode_operations = {
 lookup:myfs_lookup,
 create:myfs_create,
 mkdir:myfs_mkdir,
 rmdir:myfs_rmdir,
 unlink:myfs_unlink

};
```

Removing a file involves the following operations:

- Remove the dentry object - the name should vanish from the directory. The *dput* function releases the dentry object.
- Many files can have the same inode (hard links). Removing a file necessitates decrementing the link count of the associated inode. When the link count becomes zero, the space allocated to the file should be reclaimed.
- Removing a directory requires that we first check whether it is empty or not.

Chapter 16. Dynamic Kernel Probes

16.1. Introduction

Dynamic Probes (dprobes) is an interesting facility developed by IBM programmers which helps us to place debugging ‘probes’ at arbitrary points within kernel code (and also user programs). This chapter presents a tutorial introduction.

16.2. Overview

A ‘probe’ is a program written in a simple stack based Reverse Polish Notation language and looks similar to assembly code. It is written in such a way that it gets triggerred when control flow within the program being debugged (the kernel, a kernel module or an ordinary user program) reaches a particular address. The probe program can access any kernel location, read from CPU registers, manipulate I/O ports, perform arithmetic and logical operations, execute loops and do many of the things which an assembly language program can do. The major advantage of the dprobes mechanism is that it helps us to debug the kernel ‘dynamically’ - suppose you wish to debug an interrupt service routine that is compiled into the kernel (you might wish to place certain print statements within the routine and check some values) - you will have to recompile the kernel and reboot the system. This is no longer necessary. With the help of dprobes, it is possible to register probe programs with the running kernel; these programs will get executed when kernel control flow reaches addresses specified in the programs themselves.

16.3. Installing dprobes

A Google search for ‘dprobes’ will take you to the home page of the project. You can download the latest package (ver 3.6.4 as of writing) and try to build it. The two major components of the package are

- Kernel patches for both kernel version 2.4.19 and 2.4.20
- The user level ‘dprobes’ program

Trying to patch the kernels supplied with Red Hat might fail - a ‘patch -p1’ on a 2.4.19 kernel downloaded from a kernel.org mirror worked fine. When configuring the patched kernel, the ‘kernel hooks’ and ‘dynamic probes’ options under ‘kernel hacking’ should be enabled. Now build the patched kernel.

The next step is to build the ‘dprobes’ command - the sources are found under the ‘cmd’ subdirectory of the distribution. Once you have ‘dprobes’, you can reboot the machine with the patched kernel. Assuming that the dprobes driver is compiled into the kernel (and not made into a module) a ‘cat /proc/devices’ will show you a device called ‘dprobes’. Note down its major number and build a device file /dev/dprobes with that particular major number and minor equal to zero. You are ready to start experimenting with dprobes!

16.4. A simple experiment

We write a C program:

```
fun()
{
}
main()
{
 int i;
 scanf("%d", &i);
 if(i == 1) fun();
}
```

We compile the program into ‘a.out’. Now, we will place a probe on this program - the probe should get triggerred when the function ‘fun’ is executed. We create a file called, say, ‘a.rpn’ which looks like this:

```
name = "a.out"
modtype = user
offset = fun
opcode = 0x55

push u, cs
push u, ds
log 2
exit
```

A few things about the probe program. First, we specifiy the name of the file on which the probe is to be attached. Then, we mention what kind of code we are attaching to; in this case, a user program. Next, we specify the point within the program upon reaching which the probe is to be triggerred - this can be done as either a name or a numeric address - here, we specify the name ‘fun’. Now, the ‘opcode’ field is some kind of double check - the dprobes mechanism, when it sees that control has reached the address specified by ‘fun’, checks whether the first byte of the opcode at that location is 0x55 itself - if not the probe wont be triggerred. We can discover the opcode at a particular address by running the ‘objdump’ program like this:

```
objdump --disassemble-all ./a.out
```

Now, the remaining lines specify the actions which the probe should execute. The first line says ‘push u,cs’. This means "push the user context cs register on to the RPN interpreter stack". When we are debugging kernel code, we might require the value of the CS register at the instant the probe was triggerred as well as the value of the register just before the kernel context was entered from user mode. If we want to push the ‘current’ context CS register, we might say ‘push r,cs’. When debugging user programs, both contexts are the same.

After pushing two 4 byte values on to the stack, we execute ‘log 2’. This will retrieve 2 four byte values from top of stack and they will be logged using the kernel logging mechanism (the log output may be viewed by running ‘dmesg’)

We now have to compile and register this probe program. The RPN program is compiled into a ‘ppdf’ file by running:

```
dprobes --build-ppdf file.rpn
```

We get a new file called file.rpn.ppdf. Now, the ppdf file should be registered with the kernel. This is done by:

```
dprobes --apply-ppdf file.rpn.ppdf
```

Now, we can run our C program and observe the probe getting triggerred. The applied probes can be removed by running ‘dprobes -r -a’.

16.5. Running a kernel probe

Let’s do something more interesting. We want a probe to get triggerred at the time when the keyboard interrupt gets raised. The keyboard interrupt handler is a function called ‘keyboard_interrupt’ defined in the file`drivers/char/pc_keyb.c`.

```
name = "/usr/src/linux/vmlinux"
modtype = kernel
offset = keyboard_interrupt
opcode = 0x8b
push task
log 1
exit
```

Note that we are putting the probe on "vmlinux", which should be the file from which the currently running dprobes-enabled kernel image has been extracted. We define module type to be ‘kernel’. We discover the opcode by running ‘objdump’ on vmlinux. The name ‘task’ referes to the address of the task structure of the currently executing process - we push it on to the stack and log it just to get some output. When this file is compiled, an extra option should be supplied:

```
dprobes --build-ppdf file.rpn --sym "/usr/src/linux/System.map"
```

Dprobes consults this ‘map file’ to get the address of the kernel symbol ‘keyboard_interrupt’.

16.6. Specifying address numerically

Here is the same probe routine as above rewritten to use numerical address:

```
name = "/usr/src/linux/vmlinux"
modtype = kernel
address = 0xc019b4f0
opcode = 0x8b
push task
log
exit
```

The address has been discovered by checking with System.map

16.7. Disabling after a specified number of 'hits'

The probe can be disabled after a specified number of hits by using a special variable called 'maxhits'.

```
name = "/usr/src/linux/vmlinux"
modtype = kernel
address = 0xc019b4f0
opcode = 0x8b
maxhits = 10
push task
log
exit
```

16.8. Setting a kernel watchpoint

It is possible to trigger a probe when certain kernel addresses are read from/written to or executed or when I/O instructions take place to/from particular addresses. In the example below, our probe is triggerred whenever the variable 'jiffies' is accessed (we know this takes place during every timer interrupt, ie, 100 times a second). The address is specified as a range - the watchpoint probe is triggerred whenever any byte in the given range is written to. We limit the number of hits to 100 (we don't want to be flooded with log messages).

```
name = "/usr/src/linux/vmlinux"
modtype = kernel
address = jiffies:jiffies+3
watchpoint = w
maxhits = 100
push 10
log 1
exit
```

Chapter 17. Running Embedded Linux on a StrongARM based hand held

17.1. The Simputer

The Simputer is a StrongArm CPU based handheld device running Linux. Originally developed by Professors at the Indian Institute of Science, Bangalore, the device has a social objective of bringing computing and connectivity within the reach of rural communities. This articles provides a tutorial introduction to programming the Simputer (and similar ARM based handheld devices - there are lots of them in the market). The reader is expected to have some experience programming on Linux. Disclaimer - I try to describe things which I had done on my Simputer without any problem - if following my instructions leads to your handheld going up in smoke - I should not be held responsible!

Note: Pramode had published this as an article in the Feb 2003 issue of Linux Gazette.

17.2. Hardware/Software

The device is powered by an Intel StrongArm (SA-1110) CPU. The flash memory size is either 32Mb or 16Mb and RAM is 64Mb or 32Mb. The peripheral features include:

- USB master as well as slave ports.
- Standard serial port
- Infra Red communication port
- Smart card reader

Some of these features are enabled by using a 'docking cradle' provided with the base unit. Power can be provided either by rechargeable batteries or external AC mains.

Simputer is powered by GNU/Linux - kernel version 2.4.18 (with a few patches) works fine. The unit comes bundled with binaries for the X-Window system and a few simple utility programs. More details can be obtained from the project home page at *http://www.simputer.org*.

17.3. Powering up

There is nothing much to it, other than pressing the 'power button'. You will see a small tux picture coming up and within a few seconds, you will have X up and running . The LCD screen is touch sensitive and you can use a small 'stylus' (geeks use finger nails!) to select applications and move through the graphical interface. If you want to have keyboard input, be prepared for some agonizing manipulations using the stylus and a 'soft keyboard' which is nothing but a GUI program from which you can select single alphabets and other symbols.

17.4. Waiting for bash

GUI's are for kids. You are not satisfied till you see the trusted old bash prompt. Well, you don't have to try a lot. The Simputer has a serial port - attach the provided serial cable to it - the other end goes to a free port on your host Linux PC (in my case, /dev/ttyS1). Now fire up a communication program (I use 'minicom') - you have to first configure the program so that it uses /dev/ttyS1 with communication speed set to 115200 (that's what the Simputer manual says - if you are using a similar handheld, this need not be the same) and 8N1 format, hardware and software flow controls disabled. Doing this with minicom is very simple - invoke it as:

```
minicom -m -s
```

Once configuration is over - just type:

```
minicom -m
```

and be ready for the surprise. You will immediately see a login prompt. You should be able to type in a user name/password and log on. You should be able to run simple commands like 'ls', 'ps' etc - you may even be able to use 'vi' .

If you are not familiar with running communication programs on Linux, you may be wondering what really happened. Nothing much - it's standard Unix magic. A program sits on the Simputer watching the serial port (the Simputer serial port, called ttySA0) - when you run minicom on the Linux PC, you establish a connection with that program, which sends you a login prompt over the line, reads in your response, authenticates you and spawns a shell with which you can interact over the line.

Once minicom initializes the serial port on the PC end, you can 'script' your interactions with the Simputer. You are exploiting the idea that the program running on the Simputer is watching for data over the serial line - the program does not care whether the data comes from minicom itself or a script. You can try out the following experiment:

- Open two consoles (on the Linux PC)
- Run minicom on one console, log on to the simputer
- On the other console, type 'echo ls > /dev/ttyS1'
- Come to the first console - you will see that the command 'ls' has executed on the Simputer.

17.5. Setting up USB Networking

The Simputer comes with a USB slave port. You can establish a TCP/IP link between your Linux PC and the Simputer via this USB interface. Here are the steps you should take:

- Make sure you have a recent Linux distribution - Red Hat 7.3 is good enough.
- Plug one end of the USB cable onto the USB slave slot in the Simputer, then boot the Simputer.

- Boot your Linux PC. DO NOT connect the other end of the USB cable to your PC now. Log in as root on the PC.
- Run the command 'insmod usbnet' to load a kernel module which enables USB networking on the Linux PC. Verify that the module has been loaded by running 'lsmod'.
- Now plug the other end of the USB cable onto a free USB slot of the Linux PC. The USB subsystem in the Linux kernel should be able to register a device attach. On my Linux PC, immediately after plugging in the USB cable, I get the following kernel messages (which can be seen by running the command 'dmesg'):

  ```
  usb.c: registered new driver usbnet
  hub.c: USB new device connect on bus1/1, assigned device
  number 3
  usb.c: ignoring set_interface for dev 3, iface 0, alt 0
  usb0: register usbnet 001/003, Linux Device
  ```

After you have reached this far, you have to run a few more commands:

- Run 'ifconfig usb0 192.9.200.1' - this will assign an IP address to the USB interface on the Linux PC.
- Using 'minicom' and the supplied serial cable, log on to the Simputer as root. Then run the command 'ifconfig usbf 192.9.200.2' on the Simputer.
- Try 'ping 192.9.200.2' on the Linux PC. If you see ping packets running to and fro, congrats. You have successfully set up a TCP/IP link!

You can now telnet/ftp to the Simputer through this TCP/IP link.

17.6. Hello, Simputer

It's now time to start real work. Your C compiler (gcc) normally generates 'native' code, ie, code which runs on the microprocessor on which gcc itself runs - most often, an Intel (or clone) CPU. If you wish your program to run on the Simputer (which is based on the StrongArm microprocessor), the machine code generated by gcc should be understandable to the StrongArm CPU - your 'gcc' should be a cross compiler. If you download the gcc source code (preferably 2.95.2) together with 'binutils', you should be able to configure and compile it in such a way that you get a cross compiler (which could be invoked like, say, arm-linux-gcc). This might be a bit tricky if you are doing it for the first time - your handheld vendor should supply you with a CD which contains the required tools in a precompiled form - it is recommended that you use it (but if you are seriously into embedded development, you should try downloading the tools and building them yourselves).

Assuming that you have arm-linux-gcc up and running, you can write a simple 'Hello, Simputer' program, compile it into an 'a.out', ftp it onto the Simputer and execute it (it would be good to have one console on your Linux PC running ftp and another one running telnet - as soon as you compile the code, you can upload it and run it from the telnet console - note that you may have to give execute permission to the ftp'd code by doing 'chmod u+x a.out' on the Simputer).

17.6.1. A note on the Arm Linux kernel

The Linux kernel is highly portable - all machine dependencies are isolated in directories under the 'arch' subdirectory (which is directly under the root of the kernel source tree, say, /usr/src/linux). You will find a directory called 'arm' under 'arch'. It is this directory which contains ARM CPU specific code for the Linux kernel.

The Linux ARM port was initiated by Russell King. The ARM architecture is very popular in the embedded world and there are a LOT of different machines with fantastic names like Itsy, Assabet, Lart, Shannon etc all of which use the StrongArm CPU (there also seem to be other kinds of ARM CPU's - now that makes up a really heady mix). There are minor differences in the architecture of these machines which makes it necessary to perform 'machine specific tweaks' to get the kernel working on each one of them. The tweaks for most machines are available in the standard kernel itself, and you only have to choose the actual machine type during the kernel configuration phase to get everything in order. But to make things a bit confusing with the Simputer, it seems that the tweaks for the initial Simputer specification have got into the ARM kernel code - but the vendors who are actually manufacturing and marketing the device seem to be building according to a modified specification - and the patches required for making the ARM kernel run on these modified configurations is not yet integrated into the main kernel tree. But that is not really a problem, because your vendor will supply you with the patches - and they might soon get into the official kernel.

17.6.2. Getting and building the kernel source

You can download the 2.4.18 kernel source from the nearest Linux kernel ftp mirror. You will need the file 'patch-2.4.18-rmk4' (which can be obtained from the ARM Linux FTP site ftp.arm.linux.org.uk). You might also need a vendor supplied patch, say, 'patch-2.4.18-rmk4-vendorstring'. Assume that all these files are copied to the /usr/local/src directory.

- First, untar the main kernel distribution by running 'tar xvfz kernel-2.4.18.tar.gz'
- You will get a directory called 'linux'. Change over to that directory and run 'patch -p1 < ../patch-2.4.18-rmk4'.
- Now apply the vendor supplied patch. Run 'patch -p1 < ../patch-2.4.18-rmk4-vendorstring'.

Now, your kernel is ready to be configured and built. Before that, you have to examine the top level Makefile (under /usr/local/src/linux) and make two changes - there will be a line of the form *ARCH := <lots-of-stuff>* near the top. Change it to *ARCH := arm*

You need to make one more change. You observe that the Makefile defines:

```
AS = ($CROSS_COMPILE)as
LD = ($CROSS_COMPILE)ld
CC = ($CROSS_COMPILE)gcc
```

You note that the symbol CROSS_COMPILE is equated with the empty string. During normal compilation, this will result in AS getting defined to 'as', CC getting defined to 'gcc' and so on which is what we want. But when we are cross compiling, we use arm-linux-gcc, arm-linux-ld, arm-linux-as etc. So you have to equate CROSS_COMPILE with the string arm-linux-, ie, in the Makefile, you have to enter *CROSS_COMPILE = arm-linux-*

Once these changes are incorporated into the Makefile, you can start configuring the kernel by running 'make menuconfig' (note that it is possible to do without modifying the Makefile. You run 'make menuconfig ARCH=arm'). It may take a bit of tweaking here and there before you can actually build the kernel without error. You will not need to modify most things - the defaults should be acceptable.

- You have to set the system type to SA1100 based ARM system and then choose the SA11x0 implementation to be 'Simputer(Clr)' (or something else, depending on your machine). I had also enabled SA1100 USB function support, SA11x0 USB net link support and SA11x0 USB char device emulation.
- Under Character devices->Serial drivers, I enabled SA1100 serial port support, console on serial port support and set the default baud rate to 115200 (you may need to set differently for your machine).
- Under Character devices, SA1100 real time clock and Simputer real time clock are enabled.
- Under Console drivers, VGA Text console is disabled
- Under General Setup, the default kernel command string is set to 'root=/dev/mtdblock2 quite'. This may be different for your machine.

Once the configuration process is over, you can run *make zImage* and in a few minutes, you should get a file called 'zImage' under arch/arm/boot. This is your new kernel.

17.6.3. Running the new kernel

I describe the easiest way to get the new kernel up and running.

Just like you have LILO or Grub acting as the boot loader for your Linux PC, the handheld too will be having a bootloader stored in its non volatile memory. In the case of the Simputer, this bootloader is called 'blob' (which I assume is the boot loader developed for the Linux Advanced Radio Terminal Project, 'Lart'). As soon as you power on the machine, the boot loader starts running - If you start minicom on your Linux PC, keep the 'enter' key pressed and then power on the device, the bootloader, instead of continuing with booting the kernel stored in the device's flash memory, will start interacting with you through a prompt which looks like this:

```
blob>
```

At the bootloader prompt, you can type:

```
blob> download kernel
```

which results in blob waiting for you to send a uuencoded kernel image through the serial port. Now, on the Linux PC, you should run the command:

```
uuencode zImage /dev/stdout > /dev/ttyS1
```

This will send out a uuencoded kernel image through the COM port - which will be read and stored by the bootloader in the device's RAM. Once this process is over, you get back the boot loader prompt. You just have to type:

```
blob> boot
```

and the boot loader will run the kernel which you have right now compiled and downloaded.

17.7. A bit of kernel hacking

What good is a cool new device if you can't do a bit of kernel hacking? My next step after compiling and running a new kernel was to check out how to compile and run kernel modules. Here is a simple program called 'a.c':

```

#include <linux/module.h>
#include <linux/init.h>

/* Just a simple module */

int
init_module(void)
{
    printk("loading module...\n");
    return 0;
}

void
cleanup_module(void)
{
    printk("cleaning up ...\n");
}
```

You have to compile it using the command line:

```
arm-linux-gcc -c -O -DMODULE -D__KERNEL__ a.c  -I/usr/local/src/linux-
2.4.18/include
```

You can ftp the resulting 'a.o' onto the Simputer and load it into the kernel by running *insmod ./a.o* You can remove the module by running *rmmod a*

17.7.1. Handling Interrupts

After running the above program, I started scanning the kernel source to identify the simplest code segment which would demonstrate some kind of physical hardware access - and I found it in the hard key driver. The Simputer has small buttons which when pressed act as the arrow keys - these buttons seem to be wired onto the general purpose I/O pins of the ARM CPU (which can also be configured to act as interrupt sources - if my memory of reading the StrongArm manual is correct). Writing a kernel module which responds when these keys are pressed is a very simple thing - here is a small program which is just a modified and trimmed down version of the hardkey driver - you press the button corresponding to the right arrow key - an interrupt gets generated which results in the handler getting executed. Our handler simply prints a message and does nothing else. Before inserting the module, we must make sure that the kernel running on the device does not incorporate the default button driver code - checking /proc/interrupts would be sufficient.

Compile the program shown below into an object file (just as we did in the previous program), load it using 'insmod', check /proc/interrupts to verify that the interrupt line has been acquired. Pressing the button should result in the handler getting called - the interrupt count displayed in /proc/interrupts should also change.

```
#include <linux/module.h>
#include <linux/ioport.h>
#include <linux/sched.h>
#include <asm-arm/irq.h>
#include <asm/io.h>

static void
key_handler(int irq, void *dev_id, struct pt_regs *regs)
{
  printk("IRQ %d called\n", irq);
}

static int
init_module(void)
{
  unsigned int res = 0;
  printk("Hai, Key getting ready\n");
  set_GPIO_IRQ_edge(GPIO_GPIO12, GPIO_FALLING_EDGE);
  res = request_irq(IRQ_GPIO12, key_handler, SA_INTERRUPT,
  "Right Arrow Key", NULL);
  if(res) {
     printk("Could Not Register irq %d\n", IRQ_GPIO12);
     return res;
   }
  return res ;
}

static void
cleanup_module(void)
{
  printk("cleanup called\n");
  free_irq(IRQ_GPIO12, NULL);
}
```

Chapter 18. Programming the SA1110 Watchdog timer on the Simputer

18.1. The Watchdog timer

Due to obscure bugs, your computer system is going to lock up once in a while - the only way out would be to reset the unit. But what if you are not there to press the switch? You need to have some form of 'automatic reset'. The watchdog timer presents such a solution.

Imagine that your microprocessor contains two registers - one which gets incremented every time there is a low to high (or high to low) transition of a clock signal (generated internal to the microprocessor or coming from some external source) and another one which simply stores a number. Let's assume that the first register starts out at zero and is incremented at a rate of 4,000,000 per second. Lets assume that the second register contains the number 4,000,000,0. The microprocessor hardware compares these two registers every time the first register is incremented and issues a reset signal (which has the result of rebooting the system) when the value of these registers match. Now, if we do not modify the value in the second register, our system is sure to reboot in 10 seconds - the time required for the values in both registers to become equal.

The trick is this - we do not allow the values in these registers to become equal. We run a program (either as part of the OS kernel or in user space) which keeps on moving the value in the second register forward before the values of both become equal. If this program does not execute (because of a system freeze), then the unit would be automatically rebooted the moment the value of the two registers match. Hopefully, the system will start functioning normally after the reboot.

Note: Pramode had published this as an article in a recent issue of Linux Gazette.

18.1.1. Resetting the SA1110

The Intel StrongArm manual specifies that a software reset is invoked when the Software Reset (SWR) bit of a register called RSRR (Reset Controller Software Register) is set. The SWR bit is bit D0 of this 32 bit register. My first experiment was to try resetting the Simputer by setting this bit. I was able to do so by compiling a simple module whose 'init_module' contained only one line:

```
RSRR = RSRR | 0x1
```

18.1.2. The Operating System Timer

The StrongArm CPU contains a 32 bit timer that is clocked by a 3.6864MHz oscillator. The timer contains an OSCR (operating system count register) which is an up counter and four 32 bit match registers (OSMR0 to OSMR3). Of special interest to us is the OSMR3.

If bit D0 of the OS Timer Watchdog Match Enable Register (OWER) is set, a reset is issued by the hardware when the value in OSMR3 becomes equal to the value in OSCR. It seems that bit D3 of the OS Timer Interrupt Enable Register (OIER) should also be set for the reset to occur.

Using these ideas, it is easy to write a simple character driver with only one method - ‘write’. A write will delay the reset by a period defined by the constant ‘TIMEOUT’.

```

/*
 * A watchdog timer.
 */

#include <linux/module.h>
#include <linux/ioport.h>
#include <linux/sched.h>
#include <asm-arm/irq.h>
#include <asm/io.h>

#define WME 1
#define OSCLK 3686400 /* The OS counter gets incremented
                       * at this rate
                       * every second
                       */

#define TIMEOUT 20 /*  20 seconds timeout */

static int major;
static char *name = "watchdog";

void
enable_watchdog(void)
{
    OWER = OWER | WME;
}

void
enable_interrupt(void)
{
    OIER = OIER | 0x8;
}

ssize_t
watchdog_write(struct file *filp, const char *buf,
 size_t count, loff_t *offp)
{
    OSMR3 = OSCR + TIMEOUT*OSCLK;
    printk("OSMR3 updated...\n");
    return count;
}

static struct file_operations
 fops = {write:watchdog_write};

int
init_module(void)
{
    major = register_chrdev(0, name, &fops);
    if(major < 0) {
```

```
        printk("error in init_module...\n");
        return major;
    }
    printk("Major = %d\n", major);
    OSMR3 = OSCR + TIMEOUT*OSCLK;
    enable_watchdog();
    enable_interrupt();
    return 0;
}


void
cleanup_module()
{
    unregister_chrdev(major, name);
}

```

It would be nice to add an ‘ioctl’ method which can be used at least for getting and setting the timeout period.

Once the module is loaded, we can think of running the following program in the background (of course, we have to first create a device file called ‘watchdog’ with the major number which ‘init_module’ had printed). As long as this program keeps running, the system will not reboot.

```

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

#define TIMEOUT 20

main()
{
        int fd, buf;
        fd = open("watchdog", O_WRONLY);
        if(fd < 0) {
            perror("Error in open");
            exit(1);
        }
        while(1) {
            if(write(fd, &buf, sizeof(buf)) < 0) {
                  perror("Error in write, System may
                  reboot any moment...\n");
                  exit(1);
            }
            sleep(TIMEOUT/2);
        }
}

```

Appendix A. List manipulation routines

A.1. Doubly linked lists

The header file `include/linux/list.h` presents some nifty macros and inline functions to manipulate doubly linked lists. You might have to stare hard at them for 10 minutes before you understand how they work.

A.1.1. Type magic

What does the following program do?

Example A-1. Interesting type arithmetic

```
struct baz {
 int i, j;
};

struct foo{
 int a, b;
 struct baz m;
};

main()
{
 struct foo f;
 struct baz *p = &f.m;
 struct foo *q;

  printf("p = %x\n", p);

  printf("offset of baz in  foo = %x\n",&(((struct foo*)0)->m));
  q = (struct foo *)((char*)p -
    (unsigned long)&(((struct foo*)0)->m));
  printf("computed address of struct foo f = %x,", q);
  printf("which should be equal to %x\n",&f);
}
```

Our objective is to extract the address of the structure which encapsulates the field "m" given just a pointer to this field. Had there been an object of type *struct foo* at memory location 0, the address of its field "m" will give us the offset of "m" from the start of an object of type struct foo placed anywhere in memory. Subtracting this offset from the address of the field "m" will give us the address of the structure which encapsulates "m".

> **Note:** The expression &(((struct foo*)0)->m) does not generate a segfault because the compiler does not generate code to access anything from location zero - it is simply computing the address of the field "m", assuming the structure base address to be zero.

A.1.2. Implementation

The kernel doubly linked list routines contain very little code which needs to be executed in kernel mode - so we can simply copy the file, take off a few things and happily write user space code. Here is our slightly modified `list.h`:

Example A-2. The list.h header file

```
#ifndef _LINUX_LIST_H
#define _LINUX_LIST_H

/*
 * Simple doubly linked list implementation.
 *
 * Some of the internal functions ("__xxx") are useful when
 * manipulating whole lists rather than single entries, as
 * sometimes we already know the next/prev entries and we can
 * generate better code by using them directly rather than
 * using the generic single-entry routines.
 */

struct list_head {
 struct list_head *next, *prev;
};

typedef struct list_head list_t;

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
 struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
 (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)

/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static __inline__ void
__list_add(struct list_head * new,
 struct list_head * prev,
 struct list_head * next)
{
 next->prev = new;
 new->next = next;
 new->prev = prev;
 prev->next = new;
}

/**
 * list_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
```

```
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static __inline__ void
list_add(struct list_head *new, struct list_head *head)
{
 __list_add(new, head, head->next);
}

/**
 * list_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
static __inline__ void
list_add_tail(struct list_head *new, struct list_head *head)
{
 __list_add(new, head->prev, head);
}

/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static __inline__ void __list_del(struct list_head * prev,
      struct list_head * next)
{
 next->prev = prev;
 prev->next = next;
}

/**
 * list_del - deletes entry from list.
 * @entry: the element to delete from the list.
 * Note: list_empty on entry does not return true after
 * this, the entry is in an undefined state.
 */
static __inline__ void list_del(struct list_head *entry)
{
 __list_del(entry->prev, entry->next);
}

/**
 * list_del_init - deletes entry from list and reinitialize it.
 * @entry: the element to delete from the list.
 */
static __inline__ void list_del_init(struct list_head *entry)
{
 __list_del(entry->prev, entry->next);
 INIT_LIST_HEAD(entry);
```

```
}

/**
 * list_empty - tests whether a list is empty
 * @head: the list to test.
 */
static __inline__ int list_empty(struct list_head *head)
{
 return head->next == head;
}


/**
 * list_entry - get the struct for this entry
 * @ptr: the &struct list_head pointer.
 * @type: the type of the struct this is embedded in.
 * @member: the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
 ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))


#endif
```

The routines are basically for chaining together objects of type *struct list_head*. Then how is it that they can be used to create lists of arbitrary objects? Suppose you wish to link together two objects of type say *struct foo*. What you can do is maintain a field of type *struct list_head* within struct foo. Now you can chain the two objects of type struct foo by simply chaining together the two fields of type list_head found in both objects.

Traversing the list is easy. Once we get the address of the *struct list_head* field of any object of type *struct foo*, getting the address of the *struct foo* object which encapsulates it is easy - just use the macro *list_entry* which perform the same type magic which we had seen eariler.

A.1.3. Example code

Example A-3. A doubly linked list of complex numbers

```

#include <stdlib.h>
#include <assert.h>
#include "list.h"

struct complex{
 int re, im;
 list_t p;
};

LIST_HEAD(complex_list);

struct complex *new(int re, int im)
{
```

```
 struct complex *t;
 t = malloc(sizeof(struct complex));
 assert(t != 0);
 t->re = re, t->im = im;
 return t;
}

void make_list(int n)
{
 int i, re, im;
 for(i = 0; i < n; i++) {
  scanf("%d%d", &re, &im);
  list_add_tail(&(new(re,im)->p), &complex_list);
 }
}

void print_list()
{
 list_t *q = &complex_list;
 struct complex *m;

 while(q->next != &complex_list) {
  m = list_entry(q->next, struct complex, p);
  printf("re=%d, im=%d\n", m->re, m->im);
  q = q->next;
 }
}

void delete()
{
 list_t *q;
 struct complex *m;

 /* Try deleting an element */
 /* We do not deallocate memory here */
 for(q=&complex_list; q->next != &complex_list; q = q->next) {
  m = list_entry(q->next, struct complex, p);
  if((m->re == 3)&&(m->im == 4)) list_del(&m->p);
 }
}

main()
{
 int n;
 scanf("%d", &n);
 make_list(n);
 print_list();
 delete();
 printf("----------------------\n");
 print_list();
}

```