Operating Systems

Susanne Cech and Michael Kropfberger

September 16, 1999

Contents

Chapter 1

Introduction to Linux

1.1 Documentation

1.1.1 The Linux Documentation Project (LDP)

- *The Linux Installation Guide* explains how to install and configure Linux.
- *The Linux Users' Guide* is a guide for first-time users.
- *The Linux System Administrators' Guide* gives information about various aspects of system administration.
- *The Linux Network Administrators' Guide* explains how to set up and use networks.
- *The Linux Programmers' Guide* is a guide to efficient C-programming.
- *The Linux Kernel Hackers' Guide* gives hints for modifying the Linux kernel.
- *HOWTOs* are documents, which describe in detail a certain aspect of configuring or using Linux.

Addititonal Information. DOS-to-Linux-HOWTO.

1.1.2 Man-Pages

The manual pages are organized in sections. They are located in `/usr/man/`. Every section listed names one subdirectory of `/usr/man/`.

`man1`	User programs
`man2`	System calls
`man3`	Library calls
`man4`	Special files
`man5`	File formats
`man6`	Games
`man7`	Miscellaneous
`man8`	System administration
`man9`	Kernel functions

The `man <command>` displays the reference pages for the command you specify. There is also a keyword function — `man -k <keyword>`. This can be very useful if you are looking for a tool but do not know the name.

Example.

```
sam-i-am:/home/suse> man -k manual
apropos (1)      - search the manual page names and descriptions
mandb (8)        - create or update the manual page index caches
manpath (1)      - determine search path for manual pages
whatis (1)       - display manual page descriptions
whereis (1)      - locate the binary, source, and manual page
                   files for a command
xman (1x)        - Manual page display program for the X Window
                   System
```

The number in brackets indicates the section of the man page.

Example. Section 5 has a manual entry on the `passwd` file. To see this rather than the manual entry for the `passwd` command, type `man 5 passwd`.

Additional Information. Try `man man`.

1.1.3 Info-Pages

`info` is the new GNU program which should replace `man`. It uses hyperlinks to navigate through the nodes.

Additional Information. Try `info info`.

1.1.4 Other Online Documentation

1.1.4.1 Magazines

`http://www.linux-magazin.de`
`http://www.linuxmama.com`

1.1.4.2 Newsgroups

`at.linux`
`de.comp.os.linux.misc`

1.1.4.3 WWW

`http://www.sunsite.com`
`http://www.linux.org`
`http://www.redhat.com`
`http://www.suse.de`
`http://atnet.Linuxberg.com`
`http://www.freshmeat.net`

1.2 File System

1.2.1 Files and Directories

Files. Linux allows filenames to be up to 256 characters long. These characters can be lower- and uppercase letters, numbers, and other characters, ususally the dash (-), the underscore (_), and the dot (.). There is no special format necessary, as in DOS (8.3).

Directories. Unlike DOS, the slash (/) is used to separate directories. Directories are named like files because they are files, too.

1.2.2 Commands

Files.

`cp <source> <dest>`	Copy files
`mv <source> <dest>`	Rename files
`rm <file>`	Remove files
`cat <file>`	Prints file content
`less <file>`	Shows file content pagewise, allows to navigate

Directories.

`ls`	List contents of directories
`cd <dirName>`	Change working directory
`mkdir <dirName>`	Make directories
`rmdir <dirName>`	Remove empty directories
`pwd`	Print name of current/working directory

1.2.3 Devices

Linux distinguishes two different kinds of devices: block devices, such as disks, and character devices, such as serial lines, of which some may be sequentially and some randomly accessible.
As Linux treats everything as a file, each supported device is represented in the file system as a device file in `/dev`.

1.2.3.1 About Hard Disks

There are two different types of hard disk controllers, these are SCSI and IDE controllers.
Linux supports up to four IDE controllers. Every IDE-controller has a primary and a secondary interface with a master and a slave device. These are known as `/dev/hda` (the primary master), `/dev/hdb` (the primary slave), `/dev/hdc` (the secondary master), and `/dev/hdd` (the secondary slave), and so on.
SCSI hard disks are known as `/dev/sda`, `/dev/sdb`, and so forth. SCSI CD-ROMs start with `/dev/scd0`.
A hard disk can be divided into several partitions. Each partition functions as if it were a separate disk.
Since there are allowed four primary partitions per hard disk, you can create an extended partition as one primary partition. This extended partition can keep several logical partitions.
Each partition and extended partition has its own device file, with the convention that 1-4 are primary partitions and partitions numbered 5 or greater are logical partitions.

Example.

```
# fdisk -l

Disk /dev/hda: 128 heads, 63 sectors, 621 cylinders
Units = cylinders of 8064 * 512 bytes
```

```
   Device Boot   Start      End   Blocks   Id  System
/dev/hda1   *        1      130   524128+   6  DOS 16-bit >=32M
/dev/hda2          131      620  1975680    5  Extended
/dev/hda5          131      260   524128+   6  DOS 16-bit >=32M
/dev/hda6          261      390   524128+   6  DOS 16-bit >=32M
/dev/hda7          391      520   524128+   6  DOS 16-bit >=32M
/dev/hda8          521      620   403168+  83  Linux native

Disk /dev/hdb: 255 heads, 63 sectors, 524 cylinders
Units = cylinders of 16065 * 512 bytes

   Device Boot   Start      End   Blocks   Id  System
/dev/hdb1            1        4    32098+  82  Linux swap
/dev/hdb2            5      524  4176900    5  Extended
/dev/hdb5            5      310  2457913+  83  Linux native
/dev/hdb6          311      417   859446   83  Linux native
/dev/hdb7          418      524   859446    6  DOS 16-bit >=32M
```

With the `fdisk` command, you can create or delete partitions of you hard disk.

1.2.4 File Permissions and Ownership

All Linux files and directories have ownership and permissions. You can change permissions, and sometimes ownership, to provide greater or lesser access to your files and directories.
When you create a file, you are the file's owner. Being the file's owner gives you the privilege of changing the file's permissions.
Files (and users) also belong to groups. Groups are a convenient way of providing access to files for more than one user but not to every user on the system.
Linux lets you specify read, write, and execute permissions for each of the following: the owner, the group, and "others" (everyone else).
The output of `ls -l` looks like:

```
-rw-r--r--   1 suse     users        4056 Apr  6 15:07 myfile
```

The `-rw-r--r--` represents the permissions for the file `myfile`. The file's ownership include `suse` as user and `users` as group.

Type	Owner			Group			Others		
-	r	w	-	r	-	-	r	-	-

The first character indicates that `myfile` is a regular file. If this were a `d`, it would be a directory. `l` would indicate a link, `c` a character device, and `b` a block device.
File permissions are often given as a three-digit number, for instance 751. The first digit codes permission for the owner, the second for the group, the last for other.
Think of `rwx` as a three-digit binary number. If permission is allowed, the corresponding digit is 1, else 0. Thus, `r-x` would be the binary number is `101`, which is 5.

Example. `rwxr-x--x` is `111`, `101`, and `001`, thus `751`.

Change Ownership. `chown <owner> <filename>`.

Change Group. `chgrp <group> <filename>`.

Change Permissions. `chmod <specification> <filename>`.

1.2.5 Filesystem Hierarchy

`/`	the root directory
`/bin`	Essential command binaries
`/boot`	Static files of the boot loader
`/dev`	Devices
`/etc`	Host-specific system configurations
`/home`	User home directories
`/lib`	Essential shared libraries and kernel modules
`/mnt`	Mount point of temporary partitions
`/opt`	Big add-on application software packages
`/proc`	Kernel and process information
`/root`	Home directory for the root user
`/sbin`	Essential system binaries
`/tmp`	Temporary files
`/usr`	Secondary hierarchy, contains shareable, read-only data
`/var`	Variable data

Additional Information. Filesystem Hierarchy Standard.

1.3 Processes

Linux supports preemptive multitasking. Every process has a unique process id (PID) and a parent process. The init process, which is the mother of all processes does not have a parent. With the command `ps a`, you get a list of all running processes.

1.3.1 Signals

Signals may be used for interprocess communication, but their normal use is error handling. Every process may receive different signals and has to react accoridingly. A process may install its own signal routines for each signal. The exception is SIGKILL (number 9), where no routine can be installed. Signals can be divided into three groups:

- system dependent signals (hardware errors, system errors ...)
- device dependent signals (SIGINTR, SIGQUIT, ...)
- user specific signals (SIGUSR1 and SIGUSR2)

You may use `kill` to send signals to a process. All available signals are shown by `kill -l`.

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR
```

So you may send signals to a process with `kill -<NUMBER> <PID>`. You can terminate a process with the command `kill <PID>`, which sends the SIGINT (number 2) to the process. If this does not work, try `kill -9 <PID>`, since this kills everything! :)

1.4 The Shell

1.4.1 What Is a Shell?

The shell is a program used to interface between the user and the Linux kernel. Every command the user types at a prompt on its screen is interpreted by the shell, then passed to the Linux kernel.
The shell is a command-language interpreter. It has its own set of built-in shell commands. The shell can also make use of all of the Linux utilities and application programs that are available on the system.
Some of the commands, such as the `pwd` command, are built into the Linux `bash` shell. Other commands, such as the `cp` command, are separate executable programs that exist in one of the directories in the file system. As the user, you do not know if the command is built into the shell or is a separate command.
To figure out what to do with user commands, the shell first checks to see if the command is one of its own built-in commands. If the command is none of these, the shell checks to see if it is an application program. The shell tries to find these application programs by looking in all of the directories that are in your search path (PATH).

1.4.2 The Most Common Shells

Several different kinds of shells are available on Linux and UNIX systems. The most common are the Bourne Shell (`sh`), the C Shell (`csh`), the Korn Shell (`ksh`), and the `tcsh`, an extension of the `csh`.
We use the Bourne Again Shell (`bash`), an extension of the Bourne Shell. The default shell for each user is specified in the system password file, called `/etc/passwd`.
The system password file contains each person's user ID, an encrypted copy of each user's password, and the name of the program to run immediately after a user logs into the system. The program specified does not have to be one of the Linux shells, but it almost always is.

```
username:password:UID:GID:comment:home dir:login command
```

Environment Variables. When you log in, Linux keeps a number of useful data items in the backgroud ready for the system to use. The actual data is held in so-called *environment variables.*
Environment variables can be *exported*, they are visible and usable for every child process of the acutal user's shell. Type `env` to get a list of all exported variables.

Example. Print out a single variable.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/openwin/bin:
/usr/lib/java/bin:/usr/games/bin:/usr/games:/opt/gnome/bin:
/opt/kde/bin:.:/usr/lib/qt/bin:/usr/lib/kaffe/bin
```

EDITOR	The default editor for every application using this environment variable.
HOME	The HOME directory of the current user.
PATH	The search path that bash uses when looking for executable files.
PS1	The first-level prompt that is displayed on the command line.
PS2	The second-level prompt that is displayed when a command is expecting more input.
USER	The current user name.
DISPLAY	The default X server, where X applications should appear.

1.4.3 The Bash

Command-Line Completion. When you enter commands into bash, the complete text is not necessary. Type in a few characters, then enter the letter <tab>, and bash will try to finish the command for you. If there are more possiblities, bash will indicate this with a beep. After pressing <tab> again, the possibilities are listed on screen, and you have to enter more characters to identify your command.

Wildcards. As known from DOS, file names can be abbreviated with the asterisk (*) or the question mark (?).

Examples.

```
$ ls
mytar.c          mytar.h          ch2.txt
mylist.c         ch1.txt          ch11.txt
$ ls my*.c
mytar.c          mylist.c
$ ls ch?.txt
ch1.txt          ch2.txt
```

Command History. bash keeps track of a certain number of previous commands that have been entered into the shell. The number of commands is given by a shell variable called HISTSIZE.

Aliases. Aliases allow a string to be substituted. The first word of each command, if unquoted, is checked to see if it has an alias.

Examples.

```
alias ll='ls -l'
alias dir='ls'
alias copy='cp'
```

Evironment Variables. Environment Variables may be set with `export MYVAR=myvalue` and used with a leading dollar sign. (`echo $PID`).

Job Control. Job control refers to the ability to control the execution behavior of a currently running process. You can suspend a running process and cause it to resume running at a later time.
A process running in the foreground blocks the parent process, your shell. To use your shell, you can stop the process by typing `CTRL-z` and continue it in the background with the command `bg`. With the command `fg`, the process runs in the foreground again.
To start a process from its beginning in the background, add an ampersand (`&`) at the end of the command.

Example. `$ updatedb &`

1.4.4 Input and Output Redirection

Following standard streams are known in UNIX and thus Linux:

File Descriptor	Name	Common Abbreviation	Default
0	Standard input	stdin	keyboard
1	Standard output	stdout	terminal
2	Standard error	stderr	terminal

Redirecting Output `">"` enables you to redirect the output from a command into a file, as opposed to having the output displayed on-screen. If you want to redirect something to a stream, use a predecessing ampersand.

Example. `$ ls /bin > commands.list`

Appending Redirected Output `">>"`

Example. `$ ls /usr/bin >> commands.list`

Redirecting Input

`"<"` enables you to redirect the input for a command, as opposed to having it from the standard input.

Example.

```
$ cat < frog
there is a frog over there.
it is really cute.
```

Redirecting Standard Output and Standard Error

Example.

```
$ make all 2>&1 > /dev/null
```

Redirects standard error to standard out, standard out to `/dev/null`. `/dev/null` swallows all input ("bit nirvana").

Pipelines

A pipeline is a sequence of one or more commands separated by the character "`|`". The format for a pipeline is:

```
command1 [ | command2 ]
```

Pipelines are a way to string together a series of commands. The standard output of `command1` is connected to the standard input of `command2`. The output of the last command is the output that you will actually see displayed on-screen.

Example.

```
$ ps | grep emacs
 269   1 S    2:01 emacs
```

Lists

A list is a sequence of one or more pipelines separated by one of the operators `;`, `&`, `&&`, or `||`, and optionally terminated by one of `;`, `&`, or `<newline>`.
Of these list operators, `&&` and `||` have equal precedence, followed by `;` and `&`, which have equal precedence, too.

;	Commands are executed sequentially. The shell waits for each command to terminate. The return status is the exit status of the last command executed.
&	The shell executes the command in the background in a sub-shell. The shell does not wait for the command to finish, and the return status is 0.
cmd1 && cmd2	cmd2 is executed if, and only if, cmd1 returns an exit status of zero.
cmd1 || cmd2	cmd2 is executed if, and only if, cmd1 returns a non-zero exit status.

1.5 Neat Programs

System Statistics

du <dirname>	Summarize disk usage of a given directory.
df	Summarize free disk space.
uptime	Tell how long the system has been running.
who	Show who is logged on and when they logged in.

Information Commands

grep	Print lines matching a pattern.
wc	Print the number of bytes, words, and lines in files.
diff	Displays the differences between two files.

Examples.

```
$ cat frog
There is a frog over there.
It is really neat.
$ cat dog
There is a dog over there.
It is really neat.
$ grep lion frog
$ grep neat frog
It is really neat.
$ wc dog
      2      10      46 dog
$ diff dog frog
1c1
< There is a dog over there.
---
```

```
> There is a frog over there.
```

1.6 The Editor Jed — an Emacs Clone

`jed <file>`	Start jed. If you start it without a file, jed will prompt to ask you for a file name.
`CTRL-a`	Go to beginning of line.
`CTRL-e`	Go to end of line.
`CTRL-d`	Delete this character.
`CTRL-k`	Delete text from here to end of line.
`CTRL-x u`	Undo.
`CTRL-x f`	Open a file.
`CTRL-x w`	Write to file.
`CTRL-x CTRL-c`	Exit.
`CTRL-x CTRL-s`	Save the buffer in its corresponding file.
`CTRL-h`	Help.

Chapter 2

Shell Programming with `bash`

2.1 Using Variables

2.1.1 Assigning a Value to a Variable

Example. `count=0; name=Gerry`
You must make sure that there are no spaces on either side of the equal sign.

2.1.2 Accessing the Value of a Variable

You have to precede the variable name with a dollar sign (`$`).

Example. `max=$count; echo $max`

2.2 Positional Parameters and Other Built-In Shell Variables

Positional parameters are used to refer to the parameters that are passed to a shell program on the command line or a shell function by the shell script that invoked the function. When you run a shell program that requires or supports a number of command-line options, each of these options is stored into a positional parameter. The first parameter is stored into a variable named `1`, the second parameter is stored into a variable named `2`, and so forth. To access the values stored in these variables, you must precede the variable name with a dollar sign (`$`) just as you do with variables you define.

`$#`	Stores the number of command-line arguments that were passed to the shell program.
`$?`	Stores the exit value of the last command that was executed.
`$0`	Stores the first word of the entered command (the name of the program).
`$*`	Stores all the arguments that were entered on the command line.
`$$`	Stores the PID of the actual process.

2.3 Commands

2.3.1 The test Command

You would use the `test` command to evaluate a condition that is used in a conditional statement or to evaluate the entrance or exit criteria for an iteration statement.

```
test expression
```

or

```
[␣expression␣]
```

The test command's integer operators

`int1 -eq int2`	True if `int1 == int2`.
`int1 -ge int2`	True if `int1 >= int2`.
`int1 -gt int2`	True if `int1 > int2`.
`int1 -le int2`	True if `int1 <= int2`.
`int1 -lt int2`	True if `int1 < int2`.
`int1 -ne int2`	True if `int1 <> int2`.

Arithmetic operations, such as `+`, `-`, `*`, `/`, need the `expr` command.

Example. `a=15; a=`expr $a + 5`; echo $a`
20
The value of `a` is substituted by the output of the command `expr $a + 5`.

The test command's string operators

`str1 = str2`	True if the strings are equal.
`str1 != str2`	True if the strings are not equal.
`str`	True if `str` is not null.
`-n str`	True if the length of `str` is non-zero.
`-z str`	True if the length of `str` is zero.

The test command's file operators

`-d file`	True if `file` exists and is a directory.
`-f file`	True if `file` exists is a regular file.
`-r file`	True if `file` exists and is readable.
`-s file`	True if `file` exists and has a size greater than zero.
`-w file`	True if `file` exists and is writable.
`-x file`	True if `file` exists and is executable.

The test command's logical operators

`! expr`	True if `expr` is not true.
`expr1 -a expr2`	True if `expr1` and `expr2` are true.
`expr1 -o expr2`	True if `expr1` or `expr2` is true.

2.3.2 Conditional Statements

2.3.2.1 The if Statement

```
if [ expression ]
then
     commands
elif [ expression2 ]
then
     commands
else
     commands
fi
```

The `elif` and `else` clauses are both optional parts of the `if` statement.

Example.

```
if [ -f .profile ]
then
     echo "There is a .profile file in the current directory."
else
     echo "Could not find the .profile file."
fi
```

2.3.2.2 The case Statement

The `case` statement enables you to compare a pattern with several other patterns and execute a block of code if a match is found.

```
case string1 in
  str1)
        commands;;
  str2)
        commands;;
  *)
        commands;;
esac
```

Example. It is often used to process the command options.

```
case $1 in
  -i|-c)
     echo "The option is -i or -c."
     ;;
  *)
     echo "$0: unrecognized option '$1'"
     ;;
esac
```

2.3.3 Iteration Statements

2.3.3.1 The shift Statement

The `shift` command moves the current values stored in the positional parameters to the left one position.

Example. If the values of the current positional parameters are

```
$1 = -r  $2 = file1  $3 = file2
```

and you execute the `shift` command, the resulting parameters would be as shown here:

```
$1 = file1  $2 = file2
```

2.3.3.2 The for Statement

```
for var1 in list
do
     commands
done
```

In this form, the `for` body executes once for each item in the list. The list can be a variable that contains several words separated by white spaces. Without the "`in list`", the shell program assumes that the `var1` variable contains all the positional parameters that were passed in to the shell program on the command line. If `list` equals `*`, it is equivalent with:

```
for var1 in `ls`
do
    commands
done
```

Example.

```
for i in *
do
     cp $i $HOME
done
```

This copies all files found in the local directory into your home directory. For unix freaks, the easier thing would be `cp * ~` :)

2.3.3.3 The while Statement

```
while expression
do
    commands
done
```

Example. The program lists all parameters that were passed to the program, along with the parameter number. `count=1`

```
while [ -n "$*" ]
do
     echo "This is parameter number $count $1"
     shift
     count=`expr $count + 1`
done
```

2.3.4 Functions

```
fname () {
  shell commands
}
```

Once you have defined your function, you can invoke it by entering the following command:

```
fname [parm1 parm2 ...]
```

Example.

```
#!/bin/bash
usage () {
  echo "Usage: $0 [options [file]]"
  echo "  -p file         prints file"
  echo "  -h, --help      display this help and exit"
  echo "  -v, --version   display version information and exit"
} #usage

#main
echo "Call usage function:"
usage
echo "Done."
```

This little script will just print out the lines:

```
Call usage function:
Usage: $0 [options [file]]
  -p file         prints file
  -h, --help      display this help and exit
  -v, --version   display version information and exit
Done.
```

2.4 First bash Script

```
 1|#!/bin/bash
 2|# Groupcopy gc
 3|#
 4|# This shell program copies all files of the current directory
 5|# to a specified directory.
 6|#
 7|# Usage: To copy all files to a specified directory, type the
 8|#        directory as parameter.
 9|#
10|#        To copy only a part of the files, thus to be prompted
```

```
11|#        for file copy, type the -i option, and then the
12|#        directory as second parameter.
13|
14|
15|# (1) Check parameters
16|
17|if [ $# -eq 0 ]
18|then
19|    echo At least one parameter must be included.
20|    exit
21|fi
22|
23|
24|# (2) Check the options
25|
26|prompt='n'
27|for i in $*
28|do
29|    case $i in
30|       -i) prompt='y'
31|           shift;;
32|       -*) echo "Unknown option."
33|           exit;;
34|    esac
35|done
36|
37|
38|# (3) Check if parameter is a directory
39|
40|if [ ! -d $1 ]
41|then
42|    echo "Parameter is not a directory."
43|    exit
44|fi
45|
46|
47|# (4) Main program
48|
49|for i in *
50|do
51|    if [ $prompt = 'y' ]                          # interactive
```

```
52|    then                              # -n doesn't add a newline
53|        echo -n "copy $i to $1? (y/n)[N] "
54|        read answer  # reads a line from STDIN, fills "answer"
55|        if [ "$answer" = 'y' ]
56|        then
57|            cp $i $1
58|            echo "$i copied to $1."
59|        else
60|            echo "$i not copied."
61|        fi
62|    else                                        # copy all files
63|        cp $i $1
64|        echo "$i copied to $1."
65|    fi
66|done
```

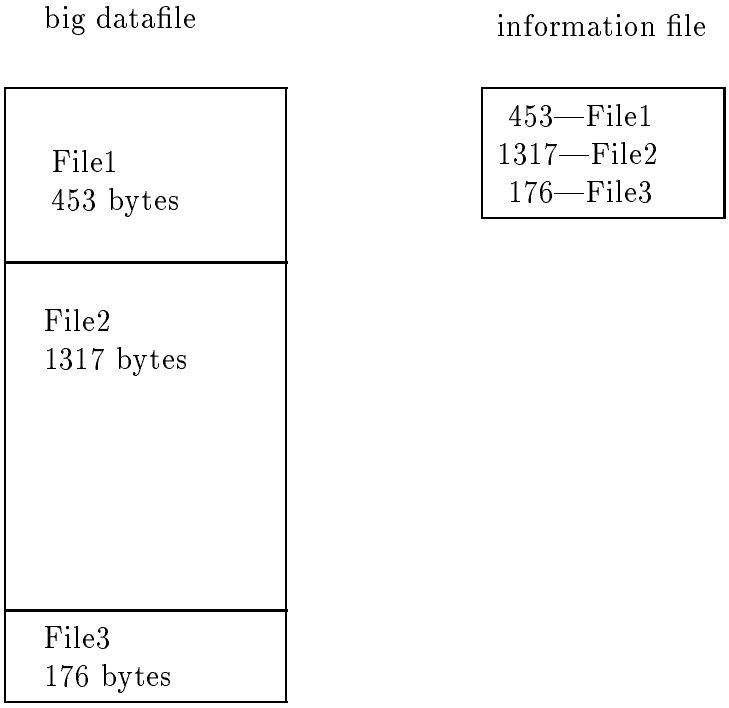
2.5 Homework: `mytar.sh`

```
USAGE: mytar.sh [--store|--restore|--help] dest src {src}
  --store:   packs all src into dest.dat and keeps information
             in dest.inf
  --restore: unpacks files stored in dest.dat into the current
             directory using the information from dest.inf
  --help:    this screen
```

We want to write a simple archiving program without compressing functionality. ASCII files are concatenated together in a so-called datafile having another file keeping track of the file names and sizes.

Therefore, only files of the current directory need to be used for input. You do not need to worry about keeping file permissions and ownerships.

`mytar.sh` creates the destination file and appends the files to the big destination file. It keeps track of all stored filenames and -sizes in a separate file. So it can restore all files iterating through this information file, extracting the files from the big destination file.



Comments:
You can use the `cut` and `dd` commands.

- `$ echo "part1|part2|part3" | cut -f 2 -d "|"`
  `part2`

  `cut` uses the delimiter (`-d "|"`) to separate fields in order to find the second field (`-f 2`).

- `dd if=/dev/fd0 of=sector.dat bs=1 count=512 skip=2048`

`/dev/fd0`	Input file is floppy disk.
`of=sector.dat`	Output file is sector.dat.
`bs=1`	Block size is one byte.
`count=512`	Read 512 blocks (read 512 bytes).
`skip=2048`	Start reading at the 2048th byte (skip the previous 2048 bytes).

  As one sector of a floppy contains 512 bytes, the 5th sector of your floppy disk is copied into the file `sector.dat`.

Additional Information for Teachers
For storage, just cat the files and append them to the big data file. Also store the filesize with `cat $actfile | wc -c` and the filename, but – here comes the tricky part – write it with another field delimiter (eg. "—") than a known whitespace. This gives you the possibility to read the file line by line afterwards and to extract the filesize and filename with `cut -f 1 -d "|"`, where `-f 1` means the first field, in this case the filesize.
To retrieve the correct number of bytes for a specific file it's the best using
`dd if=$datafile of=$actfile bs=1 count=$filesize skip=$filepos`
where `$filepos` has to be increased by every filesize read in.

2.5.1 Solution

```
 1|#!/bin/bash
 2|
 3|#------------------------------------------------------------
 4|# PRINT USAGE
 5|function printUsage {
 6|  echo 'mytar.sh (C) 1999 Mr. X'
 7|  echo 'THIS PROGRAM COMES WITH ABSOLUTELY NO WARRANTY!'
 8|  echo
 9|  echo 'USAGE: mytar.sh [--store|--restore|--help] dest src'
10|  echo '  --store:   packs all src into dest.dat and keeps'
11|  echo '             information in dest.inf'
12|  echo '  --restore: unpacks all files stored in dest.dat into
13|  echo '             the current directory'
14|  echo '             using the information from dest.inf'
15|  echo '  --help:    this screen'
16|  echo
17|}
18|
19|#------------------------------------------------------------
20|# STORE DATA
21|function storeData {
22|  DEST=$1
23|  DATA=$DEST.dat
24|  INFO=$DEST.inf
25|  shift
26|
27|  > $DATA
28|  if [ $? -ne 0 ]; then
```

```
29|     echo -e "ERROR: can't create $DATA"
30|  fi
31|
32|  > $INFO
33|  if [ $? -ne 0 ]; then
34|     echo -e "ERROR: can't create $INFO"
35|  fi
36|
37|  FILEPOS=0
38|  for i in $*; do
39|    dd if=$i of=$DATA bs=1 seek=$FILEPOS 2> /dev/null
40|    SIZE=`cat $i | wc -c`
41|    FILEPOS=`expr $FILEPOS + $SIZE`
42|    echo "working on file position [$FILEPOS]"
43|    echo `ls -la $i`>> $INFO
44|  done
45|
46|}
47|
48|#--------------------------------------------------------------
49|# RESTORE DATA
50|function restoreData {
51|  DEST=$1
52|  DATA=$DEST.dat
53|  INFO=$DEST.inf
54|  shift
55|
56|  if [ ! -f $DATA ]; then
57|     echo -e "ERROR: can't open $DATA"
58|  fi
59|
60|  if [ ! -f $INFO ]; then
61|     echo -e "ERROR: can't open $INFO"
62|  fi
63|
64|  FILEPOS=0
65|
66|
67|}
68|
69|
```

```
70|#-------------------------------------------------------------
71|#  MAIN PROGRAM
72|if [ $# -le 2 ]; then
73|   printUsage
74|   echo -e "ERROR: Not enough arguments!\n"
75|   exit -1
76|fi
77|
78|MODE=$1
79|shift
80|if [ $MODE == "--store" ]; then
81|   storeData $*
82|   exit 0
83|fi
84|if [ $MODE == "--restore" ]; then
85|   restoreData $*
86|   exit 0
87|fi
88|
89|printUsage
```

2.6 Homework: `trashcan.sh`

The command `rm` deletes files irretrievably. We want to improve the `rm` command by adding a trashcan directory. Files are moved into this trashcan before they are definitely deleted.
`srm [options [filename]]`
Options:

—d list content of trashcan
—e delete content of trashcan after prompting the user
—f never prompt the user, ignore an eventually -i
—h display help, equvivalent to `srm` without parameters
—i interactive mode, prompt whether to remove each file
—u restore `file` and remove if from trashcan

Comments:

- When removing files, check if the trashcan directory exists. If not, create it.
- If a file is not regular or not accessable, a error message is printed on standard error.
- The number of files in the trashcan should be limited. If the trashcan is full, the program aborts with an error message.
- Don't take into account file paths. But you should prompt the user if there is already a file with the same name in the trashcan.

2.6.1 Solution

```
 1|#!/bin/bash
 2|# secure rm - srm
 3|#
 4|# srm [options [file]]
 5|
 6|
 7|TrashDir=$HOME/TrashCan
 8|MaxFiles=10                          # limited number of files
 9|
10|#_____________________________________________
11|PrintTrashCan () {                          # prints Trashcan
12|  if [ -d $TrashDir ]                   # does Trashcan exist?
13|  then
14|    ls -la $TrashDir
15|  fi
16|  exit 0
17|}
18|
19|#_____________________________________________
20|DeleteTrashcan () {                        # removes Trashcan
21|  if [ $force -eq 0 ]                 # prompt before deleting
22|  then
23|    echo "Are you sure? "
24|    read answer
25|  fi
26|  if [ $answer = 'y' -o $answer = 'Y' -o  $force = 1 ]
27|  then                                                 # force!
28|    rm -r $TrashDir
```

```
29|  else
30|    echo "Your trashcan won't be deleted"
31|  fi
32|  exit 0
33|}
34|
35|#_____________________________________________
36|SecureMove () {                      # moves files to Trashcan
37|  if [ ! -d $TrashDir ]              # trashcan does not exist
38|  then
39|    mkdir $TrashDir
40|  fi
41|
42|  for i in $files
43|  do
44|    if [ ! -f $i ]                       # not a regular file?
45|    then
46|      echo "$i is not a regular file"
47|    elif [ `ls $TrashDir | wc -w` -eq $MaxFiles ]
48|    then                                          # trashcan full
49|      echo "No remove possible! TrashCan full!!!!"
50|    else
51|      if [ $force -eq 0 ]               # interactive or force
52|      then
53|        echo "Move $i to $TrashDir?"
54|        read answer
55|      else
56|        answer=no       # pseudo -> else next if does not work
57|      fi
58|      if [ $force -eq 1 -o $answer = 'y' -o $answer = 'Y' ]
59|      then
60|        if [ -f $TrashDir/$i ]       # no doubles in Trashcan!
61|        then
62|          echo "Already $i in your TrashCan. Overwrite?"
63|          read answer
64|          case $answer in
65|             'y' | 'Y') mv $i $TrashDir;;
66|          esac
67|        else
68|          mv $i $TrashDir
69|        fi
```

```
 70|      fi
 71|    fi
 72|    shift                                          # next file
 73|  done
 74|}
 75|
 76|#_____________________________________________
 77|PrintHelp () {                            # prints help message
 78|  echo "Usage: srm [options [file]]"
 79|  echo "  -d          print the content of trashcan"
 80|  echo "  -e          prompt before removal of trashcan "
 81|  echo "  -f          never prompt before any removal"
 82|  echo "  -h          display this help and exit"
 83|  echo "  -i          prompt before any removal"
 84|  echo "  -u file     retrieves file"
 85|}
 86|
 87|#_____________________________________________
 88|Undo () {                                    # retrieves files
 89|  for i in $files
 90|  do
 91|    if [ $force -eq 0 ]                           # interactive
 92|    then
 93|      echo "Retrieving $1. Are you sure? "
 94|      read answer
 95|    else
 96|      answer=n          # pseudo -> else next if does not work
 97|    fi
 98|    if [ $force -eq 1 -o $answer = 'y' -o $answer = 'Y' ]
 99|    then
100|      echo "Enter directory outgoing from $HOME/:"
101|      read dir
102|      mv $1 $HOME/$dir
103|    else
104|      echo "Nothing will be retrieved."
105|    fi
106|    shift                                          # next file
107|  done
108|  exit
109|}
110|
```

```
111|
112|
113|#_____________________________________________
114|#---------------------------------------------   main program
115|
116|# check parameters
117|  if [ $# -eq 0 ]                               # no parameters
118|  then
119|    PrintHelp
120|    exit
121|  fi
122|
123|  force=0; elim=0; undo=0;
124|  for i in $*             # loop through positional parameters
125|  do
126|    case $i in
127|      -d)  PrintTrashCan
128|           exit;;
129|
130|      -e)  elim=1
131|           shift;;
132|
133|      -f)  force=1
134|           shift;;
135|
136|      -h)  PrintHelp
137|           exit;;
138|
139|      -i)  shift;;
140|
141|      -u)  undo=1
142|           shift;;
143|
144|      -*)  PrintHelp
145|           exit;;
146|    esac
147|  done
148|
149|  if [ $elim -eq 1 ]
150|  then
151|    DeleteTrashcan
```

```
152|  elif [ $undo -eq 1 ]
153|  then
154|    files=`ls $TrashDir/$1 2>/dev/null`       # dels errormsgs
155|    if [ $? -gt 0 ]                           # exitcode of ls
156|    then
157|      echo "Error...maybe file does not exist"
158|    else
159|      Undo $files
160|    fi
161|  else
162|    files=`ls $1`
163|    SecureMove $files
164|  fi
```

Chapter 3

Introduction to C

3.1 Basics

Naming Conventions.

`*.h`	Header files
`*.c`	Source files
`*.o`	Object files

Main Function.

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

        return 0;           /* returns successful execution */
}
```

`argc` holds the number of command line arguments. Notice, that the name of the program is also an argument!
`argv` is an array of pointers to characters, that is an array of strings. `argv[0]` holds the program name, the arguments are stored from `argv[1]` to `argv[argc-1]`, and `argv[argc]` holds a pointer to null.

Comments.
Comments are from `/*` to `*/` and may not be nested.

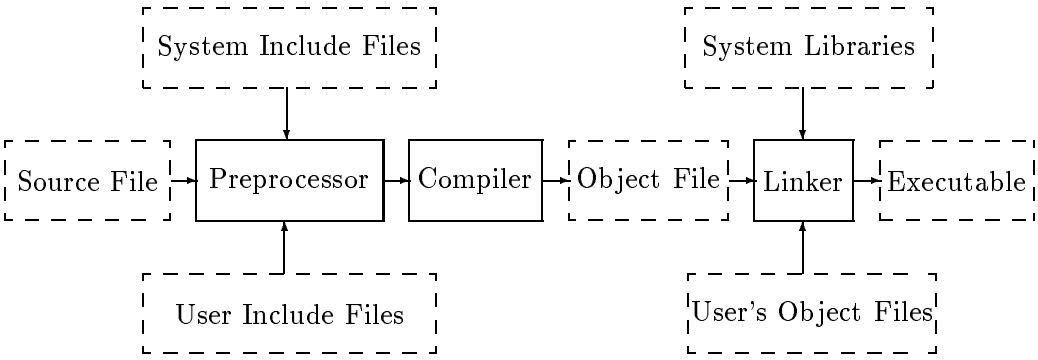
Input and Output.
Use `int printf( const char *format, arg1, arg2,..)` to print on the screen, `int scanf( const char *format, arg1, arg2,..)` to read from the keyboard.

Control Characters.

	The `int` argument is converted to a
`d` and `i`	signed decimal
`o`	unsigned octal
`u`	unsigned decimal
`x` and `X`	unsigned hexadecimal
`c`	`unsigned char`
	The `double` argument is rounded and converted
`e` and `E`	extended precision
`f`	float
`s`	string

```
char *n = "Smith";
int age = 23;
printf("Name: %s, Age: %d\n", name, age);
```

3.2 Compilation Stages



3.3 How To Compile

If you have one source file (`one_file.c`) and use the GNU-C-compiler, you can compile it with the following command. The executable file is called `one_file`.char**env

```
gcc -o one_file one_file.c
```

If you have more than one source files, each file can be compiled separately to produce an object file as follows:

```
gcc -c a_file.c
```

A file called `a_file.o` is produced. This file contains machine code, but no references to external libraries. This is done by the linker.
Suppose that two object files `file1.o` and `file2.o` have been created, the following command links the two together and produces an executable called `my_program`.

```
gcc -o my_program file1.o file2.o
```

This process is simplified using the `make` command. Create a `makefile` of the following form: separate the label from the expressions with tabs.

```
my_program:     file1.o file2.o
                gcc -o my_program file1.o file2.o

file1.o:        file1.c file1.h
                gcc -c file1.c

file2.o:        file2.c file2.h
                gcc -c file2.c
```

Each time a source or header file is edited, only the commands which depend on it are executed. Type `make` and Linux works out which commmands to execute.

3.4 The Preprocessor

The preprocessor accepts source code as input and is responsible for

- removing comments.
- interpreting special preprocessor directives denoted by `#`.

Examples.

```
#include "file.h" /* file of the current directory inserted */
#include <stdio.h>   /* file found in library path inserted */

#define DEBUG
#define MAX  10

#define MUL_1(x,y)  x*y       /* incorrect macro definition */
#define MUL_2(x,y)  ((x)*(y))   /* correct macro definition */
```

```
int main()
{
        int my_arr[MAX];
        int i;

        for (i = 0; i < MAX; i++) {
                my_arr[i] = MUL_2(a+b,b);
#ifdef DEBUG            /* compiled only if DEBUG is defined */
                printf( "debug: my_arr[%d] %d\n", i, my_arr[i]);
#endif
        }
        return 0;
}
```

3.5 Types

Built-In Types

`short, int, long`	Integral Types
`char`	Integral Type
`float, double, long double`	Floating Point Types
`void`	Void Type

There is no boolean type. `0` symbolizes *false*, and every number not equal to `0` is *true*. Integral types may be declared without a sign-bit by placing `unsigned` in front of the type.

Storage Classes

- `register`
  Often used variables can be preceeded with the keyword `register`. The compiler tries to keep the variable in the CPU registers. This flag can be ignored by the compiler!

  Example.

  ```
  register int i;

  for ( i = 0; i < 1000; i++)
          printf( "Hello &d\n", i);
  ```

- `static`
  `static` is used in two ways: The memory for `static` declared variables is never freed. Used on functions or global variables, it makes them invisible to other files.

```
#include <stdio.h>

int proc1()
{
        static int i = 0;
        i = i + 5;
        return i;
}

int main()
{
        printf("%d\n", proc1());    /* prints 5 */
        printf("%d\n", proc1());    /* prints 10 */
}
```

- `extern`
  When you want to use global variables among files, you use the keyword `extern` in a header file to export them.

3.5.1 Enumerations

Enumerated types contain a list of constants that can be addressed in integer values.

```
enum animals { dog, cat, sheep } anAnimal;
enum winter { dec = 12, jan = 1, feb, march } aMonth;

anAnimal = dog;
aMonth = feb;
printf("Animal: %d\n", anAnimal);   /* Animal: 0 */
printf("Month:  %d\n", aMonth);     /* Month:  2 */
```

3.5.2 void

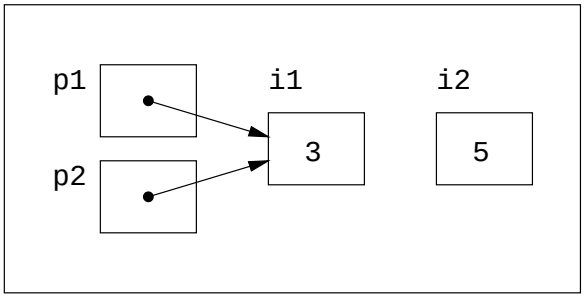
`void` is a type with no values. Thus, it is used as the return type of a function which does not return a value. `void *` is the "generic pointer" which might in fact point to any type.

3.5.3 Pointers

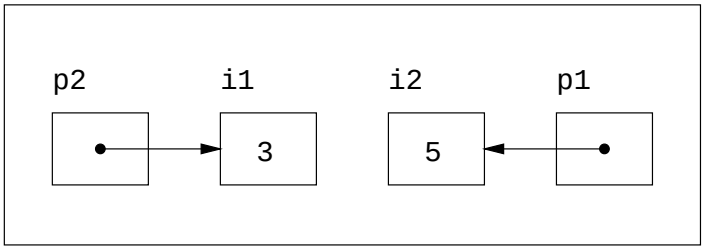
The address of a variable can be obtained by using the unary ampersand operator (&). Variables that hold such addresses are known as *pointers*.
The unary asterisk operator (*) retrieves the value of the variable stored at a particular address.

Examples.

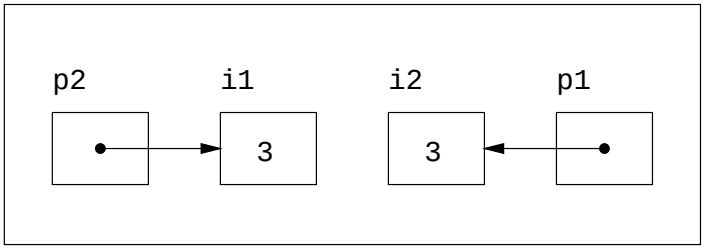
```
int i1 = 3;
int i2 = 5;
int *p1, *p2;     /* p1 and p2 point to an integer variable. */

/*   *p1 = 7;    Wrong! p1 does not point to an address yet. */
p1 = &i1;
p2 = p1;
```



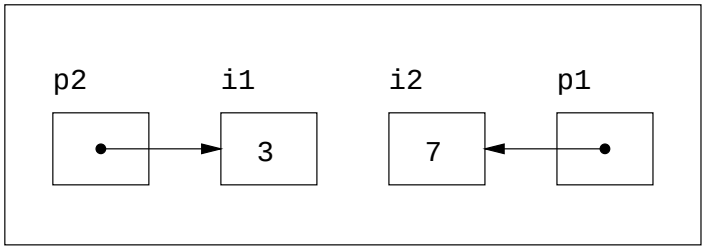

```
p1 = &i2;
```




```
i2 = *p2;
```



```
*p1 = 7;
```

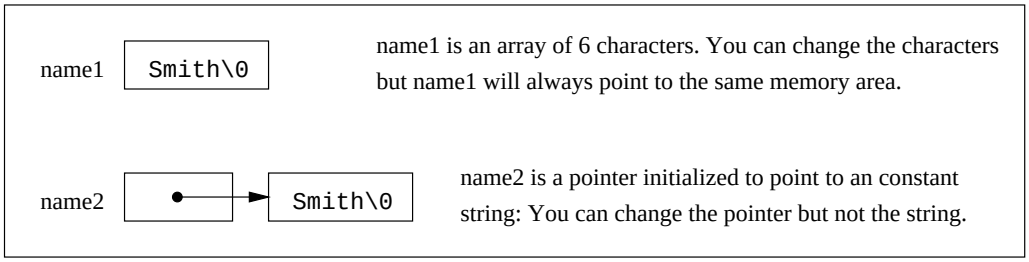


3.5.4 Arrays

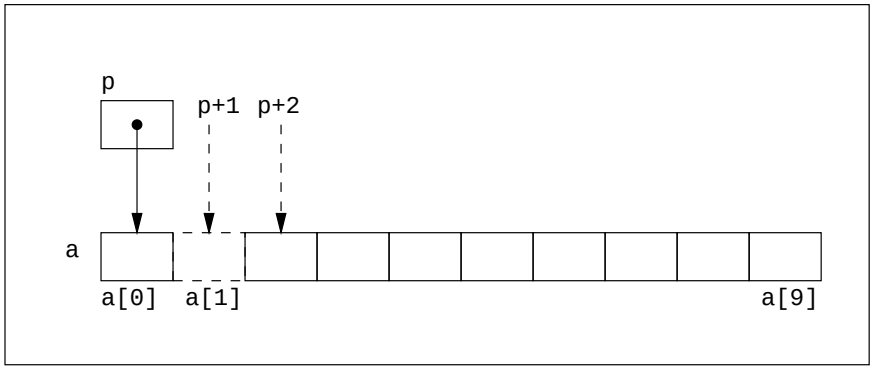
`type name [max]`: starts at index `0` and finishes at `max-1`.
`type name [max1][max2]...`: more dimensions.
A string is a null-terminated character array.

Examples.

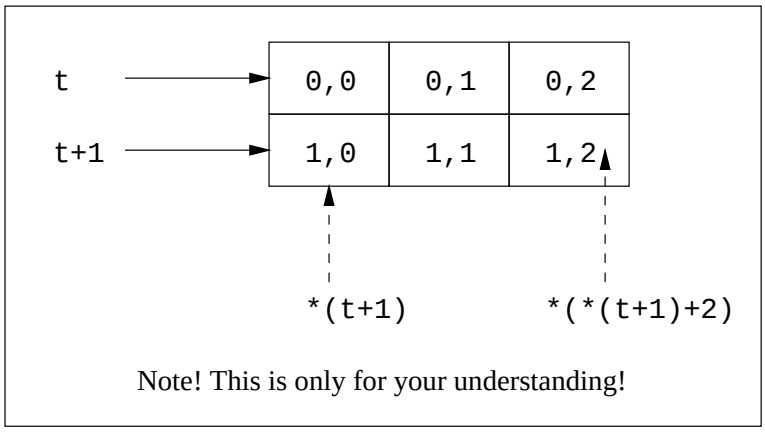
```
char name1[] = "Smith";
char name1[] = { 'S', 'm', 'i', 't', 'h', '\0' };
char *name2  = "Smith";
```




```
int a[10];
int *p,x,i;

p = &a[0];      /* p points to the address of a[0] */
p = a;          /* same as above */
x = *p;         /* <=>  x = a[0]; */
x = *(p+1);     /* <=>  x = a[1]; */
x = *(p+i);     /* <=>  x = a[i]; */
```




```
char t[2[3];

t[1][2] = 'X';  /* <=>  *(*(t+1)+2) = 'X'; */
```



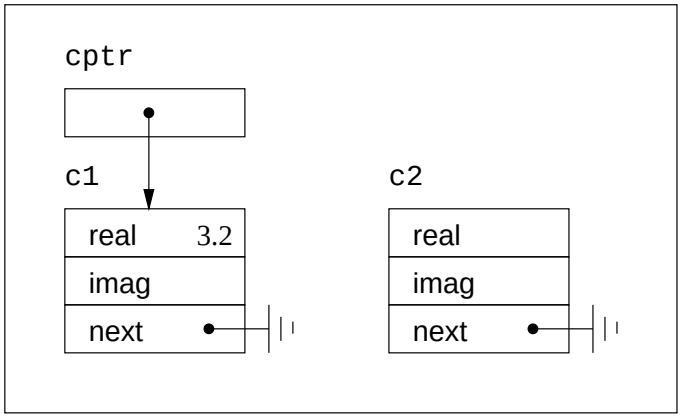
3.5.5 Structs

```
struct [structname] {declist} [namelist]
```

Example.

```
struct complex {
        float real, imag;
        struct complex *next;
} c1, *cptr;

struct complex c2;

cptr = &c1;
```

```
c1.real = 3.2;        /* these 3 statements are equivalent */
cptr->real = 3.2;
(*cptr).real = 3.2;
```



3.5.6 Unions

A *union* is a variable which may hold objects (at different times) of different sizes and types. When the C compiler is allocating memory for unions, it will always reserve enough room for the largest member.

```
typedef struct {           /* create a new type with typedef */
        int maxpassengers;
} jet;

typedef struct {
        int liftcapacity;
} helicopter;

typedef union {
        jet jetu;
        helicopter helicopteru;
} aircraft;

aircraft a;
a.helicopteru.liftcapacity = 10;
printf("h: %d\n", a.helicopteru.liftcapacity);     /* h: 10 */
a.jetu.maxpassengers = 3;
printf("j: %d\n",a.jetu.maxpassengers);            /* j: 3 */
printf("h: %d\n", a.helicopteru.liftcapacity);     /* h: 3 */
```

3.5.7 Type Conversions

Variables of the various types are automatically converted to the most general type in that expression. Explicit conversions can be forced with the cast operator `( type )`.

Examples.

```
float d = 3.3;
int   x = 2;

d += x;        /* x is converted to a float. => d is 5.3 */
x += (int)d;   /* Casting, d is truncated.   => x is 7   */
```

3.6 Operators

3.6.1 Arithmetic Operators

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus (remainder)

3.6.2 Assignment Operators

Assignment is an operator that has lower precedence than the arithmetic operators. Thus, the value of the right hand side is calculated and then assigned to the variable of the left.

`=`	Simple assignment
`++,--`	Auto increment and decrement
op `=`	Compound assignment where *op* is any arithmetic or bitwise operator

The unary operators `++` and `--` can be used for pre or post increment (decrement). It always increments (decrements) its operand by one, the difference is *when* the increment (decrement) takes place.

Examples.

```
int x = 4;
int y = 0;
```

```
y = ++x;    /*  x is incremented before its value
             *  is assigned to y. => x == 5, y == 5
             */
y = x++;    /*  x is incremented after its value
             *  is assigned to y. => x == 6, y == 5
             */
y *= x;     /*  shorthand for y = y * x; => y == 30
             */
```

3.6.3 Relational And Logical Operators

Relational Operators

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical Operators

&& (binary)	And
|| (binary)	Or
! (unary)	Negation

3.6.4 Bitwise Operators

&	binary	And
|	binary	Or
^	binary	Exclusive or
>>	binary	Shift right
<<	binary	Shift left
~	unary	Complement

The bitwise operators operate on integers (`char`, `short`, `int`, and `long`) as if they were sequences of binary bits (which, of course, internally to the computer they are).

The << operator shifts its first operand left by a number of bits given by its second operand, filling in new 0 bits at the right. Similarly, the >> operator shifts its first operand to the right. For both of the shift operators, bits that scroll "off the end" are discarded.

Using a signed integer value is dangerous because some systems shift regardless of the sign (*logical shift*) whereas other systems take care of the sign (*arithmetic shift*).

Examples.

```
unsigned int x;

x |= BIT_MASK;       /* set BIT_MASK bits */
x &= ~BIT_MASK;      /* clear BIT_MASK bits */
x ^= BIT_MASK;       /* toggle BIT_MASK bits */
```

3.7 Precedences

The precedences of expressions are usually evaluated from left to right. Unary Operators are evaluated from right to left.

Selections	`( )`
	`[]`
	`->`
	`.`

Unary Operators	`!`
	`~`
	`++,--`
	`+,-`
	`*`
	`&`
	`(type)`
	`sizeof`

Multiplicative Operators	`*,/,%`
Additive Operators	`+,-`
Shift Operators	`<<,>>`
Relational Operators	`<=,<,>,>=` `==,!=`

Bitwise Operators	`&`
	`^`
	`|`

Logical Operators	`&&` `||`
Conditional Operator	`? :`
Assignment Operators	`=` *op* `=`

3.8 Statements

Selection Statements

```
if ( expression ) statement
if ( expression ) statement else statement
expression1 ? expression2 : expression3
switch ( expression ) {
        case const-expr: statements
        ...
        default: statements
}
```

Iteration Statements

```
while ( expression ) statement
do statement while ( expression ) ;
for ( [init] ; [testing] ; [incrementing] ) statement
```

If there is more than one statement, you must put them between curly brackets ({ }).

Jump Statements

`goto` identifier ;	Jumps to label.
`continue;`	Skip 1 iteration of loop.
`break;`	Exit from loop or switch.
`return` [expression] ;	

Example.

```
if ( a > b )                /* <=>  z = (a>b) ? a : b; */
        z = a;
else
        a = b;
```

Example.

```
switch (letter) {
    case 'A':
    case 'E':
    case 'I':
    case 'O':
    case 'U':                 /* count number of vowels */
        numberofvowels++;     /* ('A','E','I','O','U') */
```

```
        break;
    case ' ':
        numberofspaces++;   /* count number of white spaces */
        break;
    default:                /* count number of other letters */
        numberofothers++;
    break;
}
```

3.9 Functions

Funtions use *call by value* in argument passing. Thus, the values of the arguments are copied, and so the values in the calling function cannot be modified.
To effect *call by reference*:

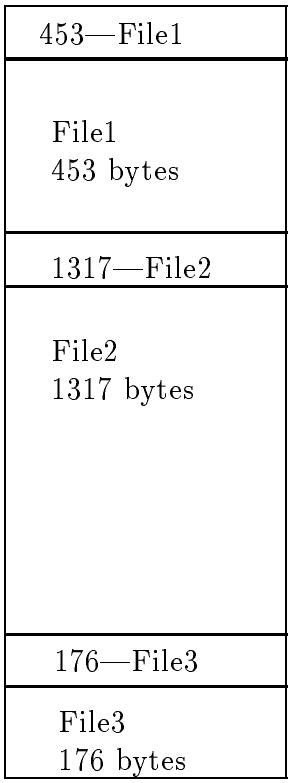
- Declare the argument as a pointer.
- Pass across the address of a variable.

3.10 Homework: `mytar.c`

```
USAGE: mytar [--store|--restore|--help] dest src {src}
  --store:   packs all src into [dest]
  --restore: unpacks all files stored in [dest] into the
             current directory
  --help:    this screen
```

As in the previous class, we want to write a simple archiving program. `mytar.c` takes files from the command line and stores them into one big datafile. Also, the files stored in this archive may be unpacked into the current directory.
Only files of the current directory need to be used for input. You do not worry about keeping file permissions and ownerships.
`mytar.c` creates the destination file and appends every file to store to the big destination file. It keeps track of all stored filenames and -sizes and stores this information as a header before every file data, so it can restore all files by sequentially parsing this big file.
It first reads the actual header, extracts the filename and size, retrieves the following `size` bytes and stores it into `filename`

big datafile



Comments:
`fgetc` and `fputc` reads and writes single bytes to a `FILE*`. `fseek` allows us to position the filepointer in a file, so we could use this to find out the file size, too. Just position the filepointer at `SEEK_END` and read the position with `ftell`.

Additional Information for Teachers
When parsing the arguments, create a list of filestructs, through which you can iterate for the storing process.
Store the file header (filename and -size) as strings, so this information can be easily extracted by using `fscanf` and `fgets`.

3.10.1 Solution

`Extract of mytar.c:`

⋮

```
 1|
 2|#include <stdio.h>
 3|#include <strings.h>
 4|#include <stdlib.h>
 5|
 6|#define STORE_MODE   1
 7|#define RESTORE_MODE 2
 8|
 9|typedef struct File {
10|   char*       name;
11|   long int    size;
12|   struct File *next;
13|} File, *FilePtr;
14|
15|
16|/*****************************************************************************/
17|void printUsage() {
18|  printf("USAGE: mytar [--store|--restore|--help] dest src {src}\n");
19|  printf("  --store:   packs all src into [dest]\n");
20|  printf("  --restore: unpacks all files stored in [dest] into the current
directory \n");
21|  printf("  --help:    this screen\n\n");
22|}//printUsage
23|
24|
25|/*****************************************************************************/
26|/** parseArgs
27| *  checks the arguments and stores the destfile and a list of all
28| *  files to store
29| *  if the action parameter is not "-store" files will stay untouched
30| */
31|int parseArgs(int argc, char *argv[], char ** destfile, FilePtr *files) {
32| int     act;
33| FilePtr actFile;
34| FILE    *f;
35|
36| //extract the big datafile from the arguments
37| *destfile=(char*)malloc(strlen(argv[2])+1);
38| strcpy(*destfile,argv[2]);
39|
40| if ( strcmp(argv[1],"--store") == 0 ) {
41|     act=3;
42|     *files=NULL; //make a safe null head
43|     while ( act < argc ) { //for every additional file argument
44|         actFile=(FilePtr)malloc(sizeof(File));
45|         if ( !actFile ) {
46|             printf("parseArgs: not enough memory!\n");
47|             return -1;
48|         }
```

```
49|          //fill up actFile with real file data
50|          actFile->name=(char*)malloc(strlen(argv[act])+1);
51|          strcpy(actFile->name,argv[act]);
52|          if ( (f=fopen(actFile->name,"r")) == NULL ) {
53|              printf("couldn't open %s\n",actFile->name);
54|              return -1;
55|          }
56|          fseek(f,0,SEEK_END); //go to very last byte
57|          actFile->size=ftell(f); //get filesize
58|          fclose(f);
59|          actFile->next=*files; //append last
60|          *files=actFile; //new head
61|          printf("extracting [%s] (%ld  bytes)\n",actFile->name,actFile->size);
62|          act++;
63|      }
64|      return STORE_MODE;
65|  }
66|  else if ( strcmp(argv[1],"--restore") == 0 )
67|      return RESTORE_MODE; //all additional parameters are ignored
68|
69|  return -1; //else
70|}//parseArgs
71|
72|/*****************************************************************************/
73|/** storeData
74|  *  stores files from the files list into the datafile
75|  *  returns -1 if an error occurs, 0 else
76|  */
77|int storeData(const char* datafile, FilePtr files) {
78|   FILE    *data;
79|   FILE    *src;
80|   FilePtr actFile;
81|   int     i;
82|
83|   printf("storeData to [%s]\n",datafile);
84|   if ( (data=fopen(datafile,"w")) == NULL ) {
85|       printf("storeData: couldn't create %s\n",datafile);
86|       return -1;
87|   }
88|   //for every file in the list
89|   for (actFile=files;actFile!=0;actFile=actFile->next) {
90|       if ((src=fopen(actFile->name,"r")) == NULL) {
91|           printf("storeData: couldn't open %s\n",actFile->name);
92|           return -1;
93|       }
94|       printf("working on [%s] with %ld bytes\n",actFile->name,actFile->size);
95|       //writing header
96|       fprintf(data,"\n%s\n",actFile->name);
97|       fprintf(data,"%ld\n",actFile->size);
```

```
 98|       //adding file contents
 99|       while ( (i=fgetc(src))!=EOF )
100|           fputc(i,data);
101|       fclose(src);
102|   }
103|   fclose(data);
104|   return 0;
105|}//storeData
106|
107|
108|/*****************************************************************************/
109|/** restoreData
110| * restores files from datafile and fills up the files list
111| * returns -1 if an error occurs, 0 else
112| */
113|int restoreData(const char* datafile, FilePtr *files) {
114|   FILE    *data;
115|   FILE    *dest;
116|   FilePtr actFile;
117|   int     pos;
118|   char    str[255];
119|
120|   printf("restoreData from [%s]\n",datafile);
121|   if ( (data=fopen(datafile,"r")) == NULL ) return -1;
122|   *files=NULL; //make a save null head
123|   while ( fgets(str,sizeof(str),data) != NULL ) { // new file found
124|       fgets(str,sizeof(str),data); //filename
125|       actFile=(FilePtr)malloc(sizeof(File));
126|       if ( !actFile ) {
127|          printf("restoreData: not enough memory!\n");
128|          return -1;
129|       }
130|       //fill up actFile structure
131|       actFile->name=(char*)malloc(strlen(str)+1);
132|       strcpy(actFile->name,str);
133|       actFile->name[strlen(actFile->name)-1]='\0'; //get rid of trailing \n
134|       fscanf(data,"%ld\n",&actFile->size); //also scans the \n now!!!
135|       //creates file
136|       if ( (dest=fopen(actFile->name,"w")) == NULL ) {
137|           printf("restoreData: couldn't create %s\n",actFile->name);
138|           return -1;
139|       }
140|       //write the file contents
141|       pos=actFile->size;
142|       while ( pos != 0 ) {
143|           fputc(fgetc(data),dest);
144|           pos--;
145|       }
146|       fclose(dest);
```

```
147|      printf("working on [%s] with %ld bytes\n",actFile->name,actFile->size);
148|      actFile->next=*files; // add tail
149|      *files=actFile; //new head
150|  }//while new file found
151|  fclose(data);
152|  return 0;
153|}//restoreData
154|
155|
156|/*****************************************************************************/
157|int main(int argc, char *argv[]) {
158|
159| char    *datafile;
160| FilePtr files;
161|
162| if ( argc<3 ) {
163|     printUsage();
164|     return -1;
165| };//if
166|
167| switch (parseArgs(argc,argv,&datafile,&files)) { //what action to take?
168|   case STORE_MODE:
169|       return storeData(datafile,files);
170|   case RESTORE_MODE:
171|       return restoreData(datafile,&files);
172|   default:
173|        printUsage();
174|        return -1;
175| }//switch modes
176|
177| return -1; //not reachable
178|}//main
```

⋮

Chapter 4

Sophisticated C-Programming

4.1 Pointers and Memory Allocation

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

`calloc()` allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero.
`malloc()` allocates `size` bytes and returns a pointer to the allocated memory. The memory is not cleared.
`free()` frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()`, `calloc()` or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behaviour occurs. If `ptr` is NULL, no operation is performed.
`realloc()` changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If `ptr` is NULL, the call is equivalent to `malloc(size)`; if `size` is equal to zero, the call is equivalent to `free(ptr)`. Unless `ptr` is NULL, it must have been returned by an earlier call to `malloc()`, `calloc()` or `realloc()`.

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

The `memset()` function fills the first `n` bytes of the memory area pointed to by `s` with the constant byte `c`.

```
#include <string.h>

char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

The `strcpy()` function copies the string pointed to be src (including the terminating \0 character) to the array pointed to by `dest`. The strings may not overlap, and the destination string dest must be large enough to receive the copy.
The strncpy() function is similar, except that not more than `n` bytes of `src` are copied. Thus, if there is no null byte among the first `n` bytes of `src`, the result wil not be null-terminated.
In the case where the length of `src` is less than that of `n`, the remainder of dest will be padded with nulls.

My String Length
Because we are eager beavers, here is our own `strlen()` function.

```
int mystrlen(char *s) {
    char *ps = s;
    while(*(ps++))       /* exits when *ps == '\0'   */
       ;
    return ps - s - 1;   /* without the '\0'   */
}
```

4.2 POSIX System Calls

Every program needs a way to communicate with the kernel. The kernel provides a set of system calls to access files, communicate with processes and so on.
POSIX is a Unix-wide standard set of system calls and their signatures. It defines the most important features an operating system has to offer.
Linux has 190 system calls, so this a big superset of the POSIX standard. You may find the list in `include/asm-i386/unistd.h`.
We will discuss the most important system calls listed here by six groups.

Process Management	`pid = fork()`	Create a child process identical to the parent (cloning)
	`pid = waitpid(pid, &statloc, opts)`	Wait for a child to terminate
	`s = execve(name,argv,envp)`	Replace a process core image
	`exit(status)`	Terminate process execution and return status
	`pid = getpid()`	Retrun the caller's process id
Signals	`s = sigaction(sig, &act, &oldact)`	Define action to take on signals
	`s = sigreturn(&context)`	Return from a signal
	`s = kill(pid, sig)`	Send a signal to a process
	`s = pause()`	Suspend the caller until the next signal
File Management	`fd = creat(name, mode)`	Obsolete way to create a new file
	`fd = mknod(name, mode, addr)`	Create a regular, special, or directory i-node
	`fd = open(file, how, ...)`	Open a file for reading, writing or both
	`s = close(fd)`	Close an open file
	`n = read(fd, buffer, nbytes)`	Read data from a file into a buffer
	`n = write(fd, buffer, nbytes)`	Write data from a buffer into a file
	`pos = lseek(fd, offset, whence)`	Move the file pointer
	`s = stat(name, &buf)`	Get a file's status information
	`s = fstat(fd, &buf)`	Get a file's status information
	`fd = dup(fd)`	Allocate a new file descriptor for an open file
	`s = pipe(&fd[0])`	Create a pipe
	`s = ioctl(fd, request, argp)`	Perform special operations on a file
	`s = access(name, amode)`	Check a file's accessibility
	`s = rename(old, new)`	Give a file a new name
	`s = fcntl(fd, cmd, ...)`	File locking and other operations
Directory & File System Management	`s = mkdir(name, mode)`	Create a new directory
	`s = rmdir(name)`	Remove an empty directory
	`s = link(name1, name2)`	Create a new entry, name2, pointing to name1 (hard link)
	`s = symlink(name1, name2)`	Create a symbolic link name2 pointing to name1
	`s = unlink(name)`	Remove a directory entry
	`s = mount(special, name, flag)`	Mount a file system
	`s = umount(special)`	Unmount a file system
	`s = sync()`	Flush all cached blocks to the disk(s)
	`s = chdir(dirname)`	Change the working directory
	`s = chroot(dirname)`	Change the root directory
Protection	`s = chmod(name, mode)`	Change a file's protection bits
	`uid = getuid()`	Get the caller's user-id
	`gid = getgid()`	Get the caller's primary group-id
	`s = setuid(uid)`	Set the caller's uid
	`s = setgid(gid)`	Set the caller's gid
	`s = chown(name, owner, group)`	Change a file's owner and group
	`oldmask = umask(complmode)`	Change the mode mask
Time Management	`seconds = time(&seconds)`	Get the elapsed time since Jan. 1, 1970
	`s = stime(&tp)`	Set the elapsed time since Jan. 1, 1970
	`s = utime(file, timep)`	Set a file's "last access" time
	`s = times(buffer)`	Get the user and system times used so far

Every system call returns -1 if an error occurs, and sets `errno` appropriately. For a short introduction to every above mentioned system call, look at Tanenbaum's Operating Systems - Design and Implementation, ISBN 0-13-638677-6, 1997 Prentice-Hall at 1.4 System Calls, pages 21-36. You can also find a description of the syntax and usage of these system calls in the man-pages, section 2 (e.g `man 2 mkdir`).

4.3 Homework

4.3.1 Pointers and Memory Allocation

Access the following variables without the `[..]` operator.

```
int iv[3];
int im[4][5];

iv[i] = 99;
```

```
im[i][j] = 18;
```

Write the following statements without increment and decrement operators and describe their functionality.

```
char *p1, *p2;

*++p1 = *++p2;
*p1-- = *p2--;
*p1-- = (*p2)--;
```

Write the typedef-declaration of a "function returning a pointer to an integer" "pointer to a function returning an integer" and describe the difference.

4.3.2 System Calls: `pipe` and `fork`

This time we will use the Posix system calls supported by Linux. Write a program, which creates a pipe and then forks itself. This enables the parent to communicate with the forked child process via the pipe.
`pipe(int)` takes an integer vector and fills it with two file descriptors, one for reading and one for writing.
`fork()` creates an exact copy of the actual process, including all file descriptors and variable contents.

Additional Information for Teachers
It may confuse the students that the child's ouput (`printf`) cannot be seen on the terminal. Only the parent has control on the actual terminal.

4.3.3 System Calls: `stat`

Write a simple program, which takes a file via command line. Then it should print the most important file attributes like file size, inode number etc.
`int stat(const char *file_name, struct stat *buf)` fills `buf` with the attributes of the file `file_name`.
Allocate the memory needed for `struct stat *buf` with `malloc`!
You should already know how to parse the command line from former examples.

Comments:
`struct stat` is defined in `/usr/src/linux/include/asm-i386/stat.h`

4.3.4 Sort and Search

For this program you should use the `qsort()` and `bsearch()` functions.

```
void qsort(void *base, size_t nmemb, size_t size,
       int (*compar)(const void *, const void *))


void *bsearch(const void *key, const void *base, size_t nmemb,
      size_t size, int (*compar)(const void *, const void *));
```

The `qsort()` function sorts an array with `nmemb` elements of size `size` (quick sort). The `base` arguments points to the start of the array.
The `bsearch()` function searches an array of `nmemb` objects, the initial member of which is pointed to by `base`, for a member that matches the object pointed to by `key`. The size of each member of the array is specified by `size` (binary search).
The last argument is a pointer to a comparison function. The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.
Download the binary file "`data.bin`". It contains records of the form

```
#define MAX_NAME  50

typedef struct {
        int  nr;
        char name[MAX_NAME];
} Student;
```

Read the file contents into an array. Write two comparison functions for comparing the `nr` field and the `name` field. Use these functions to sort and search in the array.

4.3.5 Solution

```
*(iv+3) = 99;
*(im+(i*4)+j) = 18;
p1 += 1; p2 += 1; *p1 = *p2;
*p1 = *p2; p1 -= 1; p2 -= 1;
*p1 = *p2; p1 -= 1; *p2 -= 1;
typedef int* (f1)();
typedef int (*f2)();
```

4.3.6 Solution

Extract of fork+pipe.c:

```
  1|#include <unistd.h>
  2|
  3|int main() {
  4|  int   filed[2]; //pipe handles
  5|  char  buffer[1024];
  6|  int   pid;
  7|
  8|
  9|  if (pipe(filed)<0) {
 10|    printf("Error creating pipe!\n");   exit(-1);
 11|  }
 12|
 13|  printf("Until here we are still one!\n");
 14|
 15|  if ((pid=fork())<0) {
 16|    printf("Error forking!\n");   exit(-1);
 17|  }
 18|
 19|  if (pid>0) { //parent gets childpid
 20|    printf("Hi! I'm the parent!\n");
 21|    close(filed[0]);  //parent doesn't read
 22|    strncpy(buffer,"I'm your parent, son!",sizeof(buffer));
 23|    write(filed[1],buffer,sizeof(buffer)); //writes into the pipe
 24|    close(filed[1]); //not needed anymore
 25|
 26|  }
 27|  if (pid==0) { //child could find out own pid with getpid()
 28|    printf("Hi! I'm the newborn child!\n");
 29|    close(filed[1]);  //child doesn't write
 30|    read(filed[0],buffer,sizeof(buffer)); //reads from the pipe
 31|    system("echo $$ > CHILD.pid"); //stores pid into file CHILD.pid
 32|    printf("My dad said: %s\n",buffer);
 33|    close(filed[0]); //not needed anymore
 34|  }
 35|
 36|}; //main
```

4.3.7 Solution

Extract of stat.c:

```
 1|#include <sys/stat.h>
 2|#include <unistd.h>
 3|#include <errno.h>
 4|
 5|int main(int argc, char * argv[]) {
 6|  struct stat *buf;
 7|  char *filename;
 8|
 9|  if (argc<2) { //filename is missing
10|    printf("Usage: %s filename\n",argv[0]);
11|    exit(-1);
12|  }
13|  filename=(char *)malloc(strlen(argv[1])+1); //+1 for zero-byte
14|  strncpy(filename,argv[1],strlen(argv[1]));
15|
16|  buf=(struct stat *)malloc(sizeof(struct stat));
17|  if (buf==NULL) {
18|    printf("Not enough memory for struct stat!\n");
19|    exit(-1);
20|  }
21|
22|  printf("Stats for %s\n", filename);
23|  if (stat(argv[1],buf)!=0) {
24|    printf("Error during reading stats of %s\n",filename);
25|    if (errno==EACCES) printf("Permission to %s denied!\n",filename);
26|    exit(-1);
27|  }
28|
29|  printf("dev:    %i\n",buf->st_dev); //same as (*buf).st_dev
30|  printf("inode:  %i\n",(*buf).st_ino);
31|  printf("rights: %o (oct)\n",buf->st_mode);
32|  printf("owner:  %i\n",buf->st_uid);
33|  printf("group:  %i\n",buf->st_gid);
34|  printf("size:   %li bytes\n",buf->st_size);
35|
36|}//main
```

⋮

4.3.8 Solution

```
1|#include <stdio.h>
2|#include <stdlib.h>
3|
4|#define MY_FILE "data.bin"
5|
6|#define GREATER  1
```

```
 7|#define EQUAL    0
 8|#define LESS    -1
 9|
10|#define MAX_NAME 50
11|
12|typedef struct {
13|        int  nr;
14|        char name[MAX_NAME];
15|} Student;
16|
17|void printArr( Student *a, int size)
18|{
19|        int i;
20|
21|        for ( i=0; i<size; i++ )
22|                printf("%d %s\n", a[i].nr, a[i].name);
23|        printf("----------------------------------------\n");
24|}
25|
26|int readOneStudent( Student *s)
27|{
28|        int  i = 0;
29|
30|        printf( "Nr:   ");
31|        if ( scanf( "%d", &(s->nr)) ) {
32|                printf( "Name: ");
33|                if ( scanf( "%s", &(s->name)) )
34|                        i = 1;
35|        }
36|        return i;
37|}
38|
39|int createFile( char *fname)
40|{
41|        FILE    *fp;
42|        Student s, *sp;
43|
44|        sp = malloc( sizeof( Student));
45|        if ( (fp = fopen( fname, "w")) != NULL ) {
46|                while ( readOneStudent(sp) )
47|                        fwrite( sp, sizeof( Student), 1, fp);
```

```
 48|                fclose( fp);
 49|                return 1;
 50|        }
 51|        return 0;
 52|}
 53|
 54|Student * readFile( char *fname, int *size)
 55|{
 56|        FILE    *fp;
 57|        Student s, *arr = NULL;
 58|        int     i = 0;
 59|
 60|        if ( (fp = fopen( fname, "r")) != NULL ) {
 61|                *size = 0;
 62|                while ( fread( &s, sizeof(s), 1, fp) > 0 )
 63|                        (*size)++;
 64|                if ( fseek( fp, 0L, SEEK_SET) == 0 ) {
 65|                        arr = (Student *)calloc( *size,
sizeof( Student));
 66|                        if ( arr!= NULL )
 67|                             while ( fread(&arr[i++],sizeof(Student),1,fp)
)
 68|                                        ;
 69|                }
 70|                fclose( fp);
 71|        }
 72|        return arr;
 73|}
 74|
 75|int nrCmp( const void *v1, const void *v2)
 76|{
 77|        int i1 = ((Student *)v1)->nr;
 78|        int i2 = ((Student *)v2)->nr;
 79|
 80|        if ( i1 > i2 )
 81|                return GREATER;
 82|        else if ( i1 < i2 )
 83|                return LESS;
 84|        else
 85|                return EQUAL;
 86|}
```

```
 87|
 88|int nameCmp( const void *v1, const void *v2)
 89|{
 90|        return strncmp( ((Student *)v1)->name,
 91|                        ((Student *)v2)->name, MAX_NAME);
 92|}
 93|
 94|int main( int argc, char *argv[])
 95|{
 96|        Student *a, *res;
 97|        Student s;
 98|        int     i;
 99|
100|        createFile( MY_FILE);
101|        a = readFile( MY_FILE, &i);
102|        printArr( a, i);
103|
104|        s.nr = 1;
105|        s.name[0] = 's';
106|        s.name[1] = 'u';
107|        s.name[2] = 's';
108|        s.name[3] = 'i';
109|        s.name[4] = '\0';
110|
111|        printf( "Sort numbers\n");
112|        qsort( a, i, sizeof( Student), nrCmp);
113|        printArr( a, i);
114|        printf( "Search for %d\n", s.nr);
115|        if ( res = (Student *)bsearch( &s, a, i, sizeof(
Student), nrCmp) )
116|                printf( "Found student with number %d\n",
res->nr);
117|        else
118|                printf( "Student not found\n");
119|
120|        printf( "Sort names\n");
121|        qsort( a, i, sizeof( Student), nameCmp);
122|        printArr( a, i);
123|        printf( "Search for %s\n", s.name);
124|        if ( res = (Student *)bsearch( &s, a, i, sizeof(
Student), nameCmp) )
```

```
125|                printf( "Found student with name %s\n",
res->name);
126|        else
127|                printf( "Student not found\n");
128|}
129|
130|
131|
```

Chapter 5

Follow a POSIX System Call through the Kernel

5.1 POSIX System Call: `stat()`

Try to understand the internal tasks of the stat() system call.

5.1.1 Solution

Extract of `linux/fs/open.c`:

```
⋮
 15|asmlinkage int sys_statfs(const char * path, struct statfs * buf)
 16|{
 17|        struct dentry * dentry;
 18|        int error;
 19|
 20|        lock_kernel();
 21|        dentry = namei(path);
 22|        error = PTR_ERR(dentry);
 23|        if (!IS_ERR(dentry)) {
 24|                struct inode * inode = dentry->d_inode;
 25|                struct super_block * sb = inode->i_sb;
 26|
 27|                error = -ENODEV;
 28|                if (sb && sb->s_op && sb->s_op->statfs)
 29|                        error = sb->s_op->statfs(sb, buf, sizeof(struct
statfs));
 30|
 31|                dput(dentry);
 32|        }
 33|        unlock_kernel();
 34|        return error;
```

```
 35|}
⋮
```

Extract of linux/include/linux/dcache.h:

```
⋮
 56|#define DNAME_INLINE_LEN 16
 57|
 58|struct dentry {
 59|        int d_count;
 60|        unsigned int d_flags;
 61|        struct inode  * d_inode;         /* Where the name belongs to - NULL is
negative */
 62|        struct dentry * d_parent;        /* parent directory */
 63|        struct dentry * d_mounts;        /* mount information */
 64|        struct dentry * d_covers;
 65|        struct list_head d_hash;         /* lookup hash list */
 66|        struct list_head d_lru;          /* d_count = 0 LRU list */
 67|        struct list_head d_child;        /* child of parent list */
 68|        struct list_head d_subdirs;      /* our children */
 69|        struct list_head d_alias;        /* inode alias list */
 70|        struct qstr d_name;
 71|        unsigned long d_time;            /* used by d_revalidate */
 72|        struct dentry_operations  *d_op;
 73|        struct super_block * d_sb;       /* The root of the dentry tree */
 74|        unsigned long d_reftime;         /* last time referenced */
 75|        void * d_fsdata;                 /* fs-specific data */
 76|        unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
 77|};
⋮
```

Extract of linux/include/linux/fs.h:

```
⋮
518|struct super_block {
519|        struct list_head        s_list;          /* Keep this first */
520|        kdev_t                  s_dev;
521|        unsigned long           s_blocksize;
522|        unsigned char           s_blocksize_bits;
523|        unsigned char           s_lock;
524|        unsigned char           s_rd_only;
525|        unsigned char           s_dirt;
526|        struct file_system_type *s_type;
527|        struct super_operations *s_op;
528|        struct dquot_operations *dq_op;
529|        unsigned long           s_flags;
530|        unsigned long           s_magic;
531|        unsigned long           s_time;
532|        struct dentry           *s_root;
533|        struct wait_queue       *s_wait;
534|
```

```
535|        struct inode            *s_ibasket;
536|        short int               s_ibasket_count;
537|        short int               s_ibasket_max;
538|        struct list_head        s_dirty;        /* dirty inodes */
539|
540|        union {
541|                struct minix_sb_info    minix_sb;
542|                struct ext2_sb_info     ext2_sb;
543|                struct hpfs_sb_info     hpfs_sb;
544|                struct ntfs_sb_info     ntfs_sb;
545|                struct msdos_sb_info    msdos_sb;
546|                struct isofs_sb_info    isofs_sb;
547|                struct nfs_sb_info      nfs_sb;
548|                struct sysv_sb_info     sysv_sb;
549|                struct affs_sb_info     affs_sb;
550|                struct ufs_sb_info      ufs_sb;
551|                struct romfs_sb_info    romfs_sb;
552|                struct smb_sb_info      smbfs_sb;
553|                struct hfs_sb_info      hfs_sb;
554|                struct adfs_sb_info     adfs_sb;
555|                struct qnx4_sb_info     qnx4_sb;
556|                void                    *generic_sbp;
557|        } u;
558|     /*
559|      * The next field is for VFS *only*. No filesystems have any business
560|      * even looking at it. You had been warned.
561|      */
562|        struct semaphore s_vfs_rename_sem;      /* Kludge */
563|};
⋮
⋮
622|struct super_operations {
623|        void (*read_inode) (struct inode *);
624|        void (*write_inode) (struct inode *);
625|        void (*put_inode) (struct inode *);
626|        void (*delete_inode) (struct inode *);
627|        int (*notify_change) (struct dentry *, struct iattr *);
628|        void (*put_super) (struct super_block *);
629|        void (*write_super) (struct super_block *);
630|        int (*statfs) (struct super_block *, struct statfs *, int);
631|        int (*remount_fs) (struct super_block *, int *, char *);
632|        void (*clear_inode) (struct inode *);
633|        void (*umount_begin) (struct super_block *);
634|};
⋮
```

After the kernel is locked, the path's directory entry is looked up with `namei(path)`. The according inode to the returned `dentry` helps to find the

inode's superblock. The device (filesystem) of this superblock has to support the `statfs()` operation. This fills up the buffer `buf`.

Extract of linux/include/linux/fs.h:

```
⋮
702|extern char * getname(const char * filename);
703|#define __getname()     ((char *) __get_free_page(GFP_KERNEL))
704|#define putname(name)   free_page((unsigned long)(name))
⋮
⋮
802|/*
803| * Kernel pointers have redundant information, so we can use a
804| * scheme where we can return either an error code or a dentry
805| * pointer with the same return value.
806| *
807| * This should be a per-architecture thing, to allow different
808| * error and pointer decisions.
809| */
810|#define ERR_PTR(err)    ((void *)((long)(err)))
811|#define PTR_ERR(ptr)    ((long)(ptr))
812|#define IS_ERR(ptr)     ((unsigned long)(ptr) > (unsigned long)(-1000))
⋮
⋮
826|extern struct dentry * lookup_dentry(const char *, struct dentry *,
unsigned int);
827|extern struct dentry * __namei(const char *, unsigned int);
828|
829|#define namei(pathname)         __namei(pathname, 1)
830|#define lnamei(pathname)        __namei(pathname, 0)
⋮
```

Extract of linux/fs/namei.c:

```
⋮
 92|static inline int do_getname(const char *filename, char *page)
 93|{
 94|        int retval;
 95|        unsigned long len = PAGE_SIZE;
 96|
 97|        if ((unsigned long) filename >= TASK_SIZE) {
 98|                if (!segment_eq(get_fs(), KERNEL_DS))
 99|                        return -EFAULT;
100|        } else if (TASK_SIZE - (unsigned long) filename < PAGE_SIZE)
101|                len = TASK_SIZE - (unsigned long) filename;
102|
103|        retval = strncpy_from_user((char *)page, filename, len);
104|        if (retval > 0) {
```

```
105|                if (retval < len)
106|                        return 0;
107|                return -ENAMETOOLONG;
108|        } else if (!retval)
109|                retval = -ENOENT;
110|        return retval;
111|}
112|
113|char * getname(const char * filename)
114|{
115|        char *tmp, *result;
116|
117|        result = ERR_PTR(-ENOMEM);
118|        tmp = __getname();
119|        if (tmp)  {
120|                int retval = do_getname(filename, tmp);
121|
122|                result = tmp;
123|                if (retval < 0) {
124|                        putname(tmp);
125|                        result = ERR_PTR(retval);
126|                }
127|        }
128|        return result;
129|}
⋮
⋮
436|/*
437| *     namei()
438| *
439| * is used by most simple commands to get the inode of a specified name.
440| * Open, link etc use their own routines, but this is enough for things
441| * like 'chmod' etc.
442| *
443| * namei exists in two versions: namei/lnamei. The only difference is
444| * that namei follows links, while lnamei does not.
445| */
446|struct dentry * __namei(const char *pathname, unsigned int lookup_flags)
447|{
448|        char *name;
449|        struct dentry *dentry;
450|
451|        name = getname(pathname);
452|        dentry = (struct dentry *) name;
453|        if (!IS_ERR(name)) {
454|                dentry = lookup_dentry(name, NULL, lookup_flags);
455|                putname(name);
456|                if (!IS_ERR(dentry)) {
```

```
457|                        if (!dentry->d_inode) {
458|                                dput(dentry);
459|                                dentry = ERR_PTR(-ENOENT);
460|                        }
461|                }
462|        }
463|        return dentry;
464|}
```

⋮

`getname()` uses `__getname()` to get a free page in kernel space. `do_getname()` makes a few checks if the filename's string fits into the memory page, then copies the filename from user space into the kernel space's page.
So `getname()` returns a kernel page with the filename in the first few bytes. This page is then also used for `dentry` via overlaying and casting the same page. This doesn't imply any changes to the bytes in the page! Only `lookup_dentry()` fills it with the real dentry data. The `lookup_flags` are either `1` if symlinks should be followed, `0` else.

`Extract of linux/include/linux/dcache.h:`

⋮

```
181|/* Allocation counts.. */
182|static __inline__ struct dentry * dget(struct dentry *dentry)
183|{
184|        if (dentry)
185|                dentry->d_count++;
186|        return dentry;
187|}
188|
189|extern void dput(struct dentry *);
```

⋮

`Extract of linux/fs/dcache.c:`

⋮

```
 86|/*
 87| * dput()
 88| *
 89| * This is complicated by the fact that we do not want to put
 90| * dentries that are no longer on any hash chain on the unused
 91| * list: we'd much rather just get rid of them immediately.
 92| *
 93| * However, that implies that we have to traverse the dentry
 94| * tree upwards to the parents which might _also_ now be
 95| * scheduled for deletion (it may have been only waiting for
 96| * its last child to go away).
 97| *
 98| * This tail recursion is done by hand as we don't want to depend
```

```
 99| * on the compiler to always get this right (gcc generally doesn't).
100| * Real recursion would eat up our stack space.
101| */
102|void dput(struct dentry *dentry)
103|{
104|        int count;
105|
106|        if (!dentry)
107|                return;
108|
109|repeat:
110|        count = dentry->d_count - 1;
111|        if (count != 0)
112|                goto out;
113|
114|      /*
115|       * Note that if d_op->d_delete blocks,
116|       * the dentry could go back in use.
117|       * Each fs will have to watch for this.
118|       */
119|        if (dentry->d_op && dentry->d_op->d_delete) {
120|                dentry->d_op->d_delete(dentry);
121|
122|                count = dentry->d_count - 1;
123|                if (count != 0)
124|                        goto out;
125|        }
126|
127|        if (!list_empty(&dentry->d_lru)) {
128|                dentry_stat.nr_unused--;
129|                list_del(&dentry->d_lru);
130|        }
131|        if (list_empty(&dentry->d_hash)) {
132|                struct dentry * parent;
133|
134|                list_del(&dentry->d_child);
135|                dentry_iput(dentry);
136|                parent = dentry->d_parent;
137|                d_free(dentry);
138|                if (dentry == parent)
139|                        return;
140|                dentry = parent;
141|                goto repeat;
142|        }
143|        list_add(&dentry->d_lru, &dentry_unused);
144|        dentry_stat.nr_unused++;
145|      /*
146|       * Update the timestamp
147|       */
```

```
148|        dentry->d_reftime = jiffies;
149|
150|out:
151|        if (count >= 0) {
152|                dentry->d_count = count;
153|                return;
154|        }
155|
156|        printk(KERN_CRIT "Negative d_count (%d) for %s/%s\n",
157|                count,
158|                dentry->d_parent->d_name.name,
159|                dentry->d_name.name);
160|        *(int *)0 = 0;
161|}
```

⋮

Extract of linux/fs/namei.c:

⋮

```
188|/*
189| * "." and ".." are special - ".." especially so because it has to be able
190| * to know about the current root directory and parent relationships
191| */
192|static struct dentry * reserved_lookup(struct dentry * parent, struct qstr *
name)
193|{
194|        struct dentry *result = NULL;
195|        if (name->name[0] == '.') {
196|                switch (name->len) {
197|                default:
198|                        break;
199|                case 2:
200|                        if (name->name[1] != '.')
201|                                break;
202|
203|                        if (parent != current->fs->root)
204|                                parent = parent->d_covers->d_parent;
205|                        /* fallthrough */
206|                case 1:
207|                        result = parent;
208|                }
209|        }
210|        return dget(result);
211|}
212|
213|/*
214| * Internal lookup() using the new generic dcache.
215| */
216|static struct dentry * cached_lookup(struct dentry * parent, struct qstr *
name, int flags)
```

```
217|{
218|        struct dentry * dentry = d_lookup(parent, name);
219|
220|        if (dentry && dentry->d_op && dentry->d_op->d_revalidate) {
221|                if (!dentry->d_op->d_revalidate(dentry, flags) &&
!d_invalidate(dentry)) {
222|                        dput(dentry);
223|                        dentry = NULL;
224|                }
225|        }
226|        return dentry;
227|}
228|
229|/*
230| * This is called when everything else fails, and we actually have
231| * to go to the low-level filesystem to find out what we should do..
232| *
233| * We get the directory semaphore, and after getting that we also
234| * make sure that nobody added the entry to the dcache in the meantime..
235| */
236|static struct dentry * real_lookup(struct dentry * parent, struct qstr *
name, int flags)
237|{
238|        struct dentry * result;
239|        struct inode *dir = parent->d_inode;
240|
241|        down(&dir->i_sem);
242|      /*
243|       * First re-do the cached lookup just in case it was created
244|       * while we waited for the directory semaphore..
245|       *
246|       * FIXME! This could use version numbering or similar to
247|       * avoid unnecessary cache lookups.
248|       */
249|        result = cached_lookup(parent, name, flags);
250|        if (!result) {
251|                struct dentry * dentry = d_alloc(parent, name);
252|                result = ERR_PTR(-ENOMEM);
253|                if (dentry) {
254|                        result = dir->i_op->lookup(dir, dentry);
255|                        if (result)
256|                                dput(dentry);
257|                        else
258|                                result = dentry;
259|                }
260|        }
261|        up(&dir->i_sem);
262|        return result;
263|}
```

```
264|
265|static struct dentry * do_follow_link(struct dentry *base, struct dentry
*dentry, unsigned int follow)
266|{
267|        struct inode * inode = dentry->d_inode;
268|
269|        if ((follow & LOOKUP_FOLLOW)
270|            && inode && inode->i_op && inode->i_op->follow_link) {
271|                if (current->link_count < 5) {
272|                        struct dentry * result;
273|
274|                        current->link_count++;
275|                        /* This eats the base */
276|                        result = inode->i_op->follow_link(dentry, base,
follow);
277|                        current->link_count--;
278|                        dput(dentry);
279|                        return result;
280|                }
281|                dput(dentry);
282|                dentry = ERR_PTR(-ELOOP);
283|        }
284|        dput(base);
285|        return dentry;
286|}
287|
288|static inline struct dentry * follow_mount(struct dentry * dentry)
289|{
290|        struct dentry * mnt = dentry->d_mounts;
291|
292|        if (mnt != dentry) {
293|                dget(mnt);
294|                dput(dentry);
295|                dentry = mnt;
296|        }
297|        return dentry;
298|}
299|
300|/*
301| * Name resolution.
302| *
303| * This is the basic name resolution function, turning a pathname
304| * into the final dentry.
305| */
306|struct dentry * lookup_dentry(const char * name, struct dentry * base,
unsigned int lookup_flags)
307|{
308|        struct dentry * dentry;
309|        struct inode *inode;
```

```
310|
311|        if (*name == '/') {
312|                if (base)
313|                        dput(base);
314|                do {
315|                        name++;
316|                } while (*name == '/');
317|                __prefix_lookup_dentry(name, lookup_flags);
318|                base = dget(current->fs->root);
319|        } else if (!base) {
320|                base = dget(current->fs->pwd);
321|        }
322|
323|        if (!*name)
324|                goto return_base;
325|
326|        inode = base->d_inode;
327|        lookup_flags &= LOOKUP_FOLLOW | LOOKUP_DIRECTORY | LOOKUP_SLASHOK;
328|
329|        /* At this point we know we have a real path component. */
330|        for(;;) {
331|                int err;
332|                unsigned long hash;
333|                struct qstr this;
334|                unsigned int flags;
335|                unsigned int c;
336|
337|                err = permission(inode, MAY_EXEC);
338|                dentry = ERR_PTR(err);
339|                if (err)
340|                        break;
341|
342|                this.name = name;
343|                c = *(const unsigned char *)name;
344|
345|                hash = init_name_hash();
346|                do {
347|                        name++;
348|                        hash = partial_name_hash(c, hash);
349|                        c = *(const unsigned char *)name;
350|                } while (c && (c != '/'));
351|                this.len = name - (const char *) this.name;
352|                this.hash = end_name_hash(hash);
353|
354|                /* remove trailing slashes? */
355|                flags = lookup_flags;
356|                if (c) {
357|                        char tmp;
358|
```

```
359|                         flags |= LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
360|                         do {
361|                                 tmp = *++name;
362|                         } while (tmp == '/');
363|                         if (tmp)
364|                                 flags |= LOOKUP_CONTINUE;
365|                 }
366|
367|                 /*
368|                 * See if the low-level filesystem might want
369|                 * to use its own hash..
370|                 */
371|                 if (base->d_op && base->d_op->d_hash) {
372|                         int error;
373|                         error = base->d_op->d_hash(base, &this);
374|                         if (error < 0) {
375|                                 dentry = ERR_PTR(error);
376|                                 break;
377|                         }
378|                 }
379|
380|                 /* This does the actual lookups.. */
381|                 dentry = reserved_lookup(base, &this);
382|                 if (!dentry) {
383|                         dentry = cached_lookup(base, &this, flags);
384|                         if (!dentry) {
385|                                 dentry = real_lookup(base, &this, flags);
386|                                 if (IS_ERR(dentry))
387|                                         break;
388|                         }
389|                 }
390|
391|                 /* Check mountpoints.. */
392|                 dentry = follow_mount(dentry);
393|
394|                 base = do_follow_link(base, dentry, flags);
395|                 if (IS_ERR(base))
396|                         goto return_base;
397|
398|                 inode = base->d_inode;
399|                 if (flags & LOOKUP_DIRECTORY) {
400|                         if (!inode)
401|                                 goto no_inode;
402|                         dentry = ERR_PTR(-ENOTDIR);
403|                         if (!inode->i_op || !inode->i_op->lookup)
404|                                 break;
405|                         if (flags & LOOKUP_CONTINUE)
406|                                 continue;
407|                 }
```

```
408|return_base:
409|                return base;
410|/*
411| * The case of a nonexisting file is special.
412| *
413| * In the middle of a pathname lookup (ie when
414| * LOOKUP_CONTINUE is set), it's an obvious
415| * error and returns ENOENT.
416| *
417| * At the end of a pathname lookup it's legal,
418| * and we return a negative dentry. However, we
419| * get here only if there were trailing slashes,
420| * which is legal only if we know it's supposed
421| * to be a directory (ie "mkdir"). Thus the
422| * LOOKUP_SLASHOK flag.
423| */
424|no_inode:
425|                dentry = ERR_PTR(-ENOENT);
426|                if (flags & LOOKUP_CONTINUE)
427|                        break;
428|                if (flags & LOOKUP_SLASHOK)
429|                        goto return_base;
430|                break;
431|        }
432|        dput(base);
433|        return dentry;
434|}
```

⋮

`lookup_dentry()` first sets the `base` to the root of the actual working directories filesystem and increases the allocation counter with `dget()`.

In the endless for-loop we run through every subdirectory, check the permissions and follow over mounted filesystems and symlinks.

The actual (pseudo-recursive) lookups are done by first checking, if the name is a special name like . or .. , if not, we check the cache. As a last resort we go to disk.

We break exit the loop when we finally find the right file.

The above shown functions are not complete, so if you need more detailed information, take a look at the real kernel sources.

5.2 Homework

5.2.1 POSIX System Call: `pipe`

Try to understand the internal tasks of the pipe system call and answer the following questions:

- `get_empty_filp` reserves a number of filps especially for root. How many and why?
- How big is the FIFO-buffer of a pipe and what defines its size?
- In which memory space is the buffer located?
- Where are the typical ring-buffer variables (buffer base, start, len) stored?

5.2.2 Solution

- 10 file pointers are reserved for root to allow a safe shutdown/kill/rm even if all other file pointers are used up by other user processes.
- A pipe's FIFO buffer is exactly one page, defined as `PAGE_SIZE=1UL << PAGE_SHIFT` with a defined `PAGE_SHIFT=12`, so 4 KB. These definitions are found in the file `linux/include/asm-i386/page.h`. The buffer's memory is a free page out of the user space. This all happens in `get_free_inode`.
- The typical ring-buffer variables `start` and `base` are found in the `struct pipe_inode_info`, defining the filesystem-specific data in the `struct inode`'s union `u`. The length is stored in `struct inode`'s `i_size`.

Extract of `linux/arch/i386/kernel/sys_i386.c`:

```
 27|/*
 28| * sys_pipe() is the normal C calling standard for creating
 29| * a pipe. It's not the way Unix traditionally does this, though.
 30| */
 31|asmlinkage int sys_pipe(unsigned long * fildes)
 32|{
 33|        int fd[2];
 34|        int error;
 35|
 36|        lock_kernel();
 37|        error = do_pipe(fd);
 38|        unlock_kernel();
```

```
 39|        if (!error) {
 40|                if (copy_to_user(fildes, fd, 2*sizeof(int)))
 41|                        error = -EFAULT;
 42|        }
 43|        return error;
 44|}
```

⋮

After creating a local pair of file descriptors, the kernel is locked.
`int do_pipe(int *fd)` is found in `fs/pipe.c` but we will trace it later.
Then the kernel is unlocked. If no error occured, the local vector of file descriptors pointing to memory in the kernel-space is copied to `fildes`, which is pointing to memory in user-space.

`Extract of linux/include/linux/fs.h:`

⋮

```
 50|#define NR_FILE  4096    /* this can well be larger on a larger system */
 51|#define NR_RESERVED_FILES 10 /* reserved for root */
```

⋮

⋮

```
331|struct inode {
332|        struct list_head        i_hash;
333|        struct list_head        i_list;
334|        struct list_head        i_dentry;
335|
336|        unsigned long           i_ino;
337|        unsigned int            i_count;
338|        kdev_t                  i_dev;
339|        umode_t                 i_mode;
340|        nlink_t                 i_nlink;
341|        uid_t                   i_uid;
342|        gid_t                   i_gid;
343|        kdev_t                  i_rdev;
344|        off_t                   i_size;
345|        time_t                  i_atime;
346|        time_t                  i_mtime;
347|        time_t                  i_ctime;
348|        unsigned long           i_blksize;
349|        unsigned long           i_blocks;
350|        unsigned long           i_version;
351|        unsigned long           i_nrpages;
352|        struct semaphore        i_sem;
353|        struct semaphore        i_atomic_write;
354|        struct inode_operations *i_op;
355|        struct super_block      *i_sb;
356|        struct wait_queue       *i_wait;
357|        struct file_lock        *i_flock;
```

```
358|        struct vm_area_struct   *i_mmap;
359|        struct page             *i_pages;
360|        struct dquot            *i_dquot[MAXQUOTAS];
361|
362|        unsigned long           i_state;
363|
364|        unsigned int            i_flags;
365|        unsigned char           i_pipe;
366|        unsigned char           i_sock;
367|
368|        int                     i_writecount;
369|        unsigned int            i_attr_flags;
370|        __u32                   i_generation;
371|        union {
372|                struct pipe_inode_info          pipe_i;
373|                struct minix_inode_info         minix_i;
374|                struct ext2_inode_info          ext2_i;
375|                struct hpfs_inode_info          hpfs_i;
376|                struct ntfs_inode_info          ntfs_i;
377|                struct msdos_inode_info         msdos_i;
378|                struct umsdos_inode_info        umsdos_i;
379|                struct iso_inode_info           isofs_i;
380|                struct nfs_inode_info           nfs_i;
381|                struct sysv_inode_info          sysv_i;
382|                struct affs_inode_info          affs_i;
383|                struct ufs_inode_info           ufs_i;
384|                struct romfs_inode_info         romfs_i;
385|                struct coda_inode_info          coda_i;
386|                struct smb_inode_info           smbfs_i;
387|                struct hfs_inode_info           hfs_i;
388|                struct adfs_inode_info          adfs_i;
389|                struct qnx4_inode_info          qnx4_i;
390|                struct socket                   socket_i;
391|                void                            *generic_ip;
392|        } u;
393|};
394|
395|/* Inode state bits.. */
396|#define I_DIRTY         1
397|#define I_LOCK          2
398|#define I_FREEING       4
399|
400|extern void __mark_inode_dirty(struct inode *);
401|static inline void mark_inode_dirty(struct inode *inode)
402|{
403|        if (!(inode->i_state & I_DIRTY))
404|                __mark_inode_dirty(inode);
405|}
```

```
⋮
⋮
413|struct file {
414|        struct file             *f_next, **f_pprev;
415|        struct dentry           *f_dentry;
416|        struct file_operations  *f_op;
417|        mode_t                  f_mode;
418|        loff_t                  f_pos;
419|        unsigned int            f_count, f_flags;
420|        unsigned long           f_reada, f_ramax, f_raend, f_ralen,
f_rawin;
421|        struct fown_struct      f_owner;
422|        unsigned int            f_uid, f_gid;
423|        int                     f_error;
424|
425|        unsigned long           f_version;
426|
427|        /* needed for tty driver, and maybe others */
428|        void                    *private_data;
429|};
⋮
```

Extract of linux/include/linux/pipe_fs_i.h:

```
⋮
  3|struct pipe_inode_info {
  4|        struct wait_queue * wait;
  5|        char * base;
  6|        unsigned int start;
  7|        unsigned int lock;
  8|        unsigned int rd_openers;
  9|        unsigned int wr_openers;
 10|        unsigned int readers;
 11|        unsigned int writers;
 12|};
⋮
```

Extract of linux/fs/pipe.c:

```
⋮
361|struct file_operations read_pipe_fops = {
362|        pipe_lseek,
363|        pipe_read,
364|        bad_pipe_w,
365|        NULL,            /* no readdir */
366|        pipe_poll,
367|        pipe_ioctl,
368|        NULL,            /* no mmap on pipes.. surprise */
369|        pipe_read_open,
370|        NULL,            /* flush */
```

```
371|        pipe_read_release,
372|        NULL
373|};
374|
375|struct file_operations write_pipe_fops = {
376|        pipe_lseek,
377|        bad_pipe_r,
378|        pipe_write,
379|        NULL,           /* no readdir */
380|        pipe_poll,
381|        pipe_ioctl,
382|        NULL,           /* mmap */
383|        pipe_write_open,
384|        NULL,           /* flush */
385|        pipe_write_release,
386|        NULL
387|};
388|
389|struct file_operations rdwr_pipe_fops = {
390|        pipe_lseek,
391|        pipe_read,
392|        pipe_write,
393|        NULL,           /* no readdir */
394|        pipe_poll,
395|        pipe_ioctl,
396|        NULL,           /* mmap */
397|        pipe_rdwr_open,
398|        NULL,           /* flush */
399|        pipe_rdwr_release,
400|        NULL
401|};
402|
403|static struct inode * get_pipe_inode(void)
404|{
405|        extern struct inode_operations pipe_inode_operations;
406|        struct inode *inode = get_empty_inode();
407|
408|        if (inode) {
409|                unsigned long page = __get_free_page(GFP_USER);
410|
411|                if (!page) {
412|                        iput(inode);
413|                        inode = NULL;
414|                } else {
415|                        PIPE_BASE(*inode) = (char *) page;
416|                        inode->i_op = &pipe_inode_operations;
417|                        PIPE_WAIT(*inode) = NULL;
418|                        PIPE_START(*inode) = PIPE_LEN(*inode) = 0;
419|                        PIPE_RD_OPENERS(*inode) = PIPE_WR_OPENERS(*inode) =
```

```
0;
420|                       PIPE_READERS(*inode) = PIPE_WRITERS(*inode) = 1;
421|                       PIPE_LOCK(*inode) = 0;
422|                     /*
423|                * Mark the inode dirty from the very beginning,
424|                * that way it will never be moved to the dirty
425|                * list because "mark_inode_dirty()" will think
426|                * that it already _is_ on the dirty list.
427|                */
428|                       inode->i_state = I_DIRTY;
429|                       inode->i_mode = S_IFIFO | S_IRUSR | S_IWUSR;
430|                       inode->i_uid = current->fsuid;
431|                       inode->i_gid = current->fsgid;
432|                       inode->i_atime = inode->i_mtime = inode->i_ctime
= CURRENT_TIME;
433|                       inode->i_blksize = PAGE_SIZE;
434|                }
435|        }
436|        return inode;
437|}
438|
439|struct inode_operations pipe_inode_operations = {
440|        &rdwr_pipe_fops,
441|        NULL,                   /* create */
442|        NULL,                   /* lookup */
443|        NULL,                   /* link */
444|        NULL,                   /* unlink */
445|        NULL,                   /* symlink */
446|        NULL,                   /* mkdir */
447|        NULL,                   /* rmdir */
448|        NULL,                   /* mknod */
449|        NULL,                   /* rename */
450|        NULL,                   /* readlink */
451|        NULL,                   /* readpage */
452|        NULL,                   /* writepage */
453|        NULL,                   /* bmap */
454|        NULL,                   /* truncate */
455|        NULL                    /* permission */
456|};
457|
458|int do_pipe(int *fd)
459|{
460|        struct inode * inode;
461|        struct file *f1, *f2;
462|        int error;
463|        int i,j;
464|
465|        error = -ENFILE;
466|        f1 = get_empty_filp();
```

```
467|        if (!f1)
468|                goto no_files;
469|
470|        f2 = get_empty_filp();
471|        if (!f2)
472|                goto close_f1;
473|
474|        inode = get_pipe_inode();
475|        if (!inode)
476|                goto close_f12;
477|
478|        error = get_unused_fd();
479|        if (error < 0)
480|                goto close_f12_inode;
481|        i = error;
482|
483|        error = get_unused_fd();
484|        if (error < 0)
485|                goto close_f12_inode_i;
486|        j = error;
487|
488|        error = -ENOMEM;
489|        f1->f_dentry = f2->f_dentry = dget(d_alloc_root(inode, NULL));
490|        if (!f1->f_dentry)
491|                goto close_f12_inode_i_j;
492|
493|        f1->f_pos = f2->f_pos = 0;
494|
495|        /* read file */
496|        f1->f_flags = O_RDONLY;
497|        f1->f_op = &read_pipe_fops;
498|        f1->f_mode = 1;
499|
500|        /* write file */
501|        f2->f_flags = O_WRONLY;
502|        f2->f_op = &write_pipe_fops;
503|        f2->f_mode = 2;
504|
505|        fd_install(i, f1);
506|        fd_install(j, f2);
507|        fd[0] = i;
508|        fd[1] = j;
509|        return 0;
510|
511|close_f12_inode_i_j:
512|        put_unused_fd(j);
513|close_f12_inode_i:
514|        put_unused_fd(i);
515|close_f12_inode:
```

```
516|        free_page((unsigned long) PIPE_BASE(*inode));
517|        iput(inode);
518|close_f12:
519|        put_filp(f2);
520|close_f1:
521|        put_filp(f1);
522|no_files:
523|        return error;
524|}
```

⋮

Roughly spoken `int do_pipe(int *fd)` just initializes two new file pointers and one inode. Then it gets two unused file descriptors and lets the two file pointers reference the same inode.
One of the file pointers is set to read-only, the other write-only with the according `struct` for pipe-file operations.
The final step is to join the file descriptors with their corresponding file pointers and to return the filled `fd[]`.

`Extract of linux/fs/file_table.c:`

⋮

```
 16|/* sysctl tunables... */
 17|int nr_files = 0;        /* read only */
 18|int nr_free_files = 0;   /* read only */
 19|int max_files = NR_FILE;/* tunable */
 20|
 21|/* Free list management, if you are here you must have f_count == 0 */
 22|static struct file * free_filps = NULL;
```

⋮

⋮

```
 33|/* The list of in-use filp's must be exported (ugh...) */
 34|struct file *inuse_filps = NULL;
 35|
 36|static inline void put_inuse(struct file *file)
 37|{
 38|        if((file->f_next = inuse_filps) != NULL)
 39|                inuse_filps->f_pprev = &file->f_next;
 40|        inuse_filps = file;
 41|        file->f_pprev = &inuse_filps;
 42|}
 43|
 44|/* It does not matter which list it is on. */
 45|static inline void remove_filp(struct file *file)
 46|{
 47|        if(file->f_next)
 48|                file->f_next->f_pprev = file->f_pprev;
```

```
 49|        *file->f_pprev = file->f_next;
 50|}
⋮
⋮
 67|/* Find an unused file structure and return a pointer to it.
 68| * Returns NULL, if there are no more free file structures or
 69| * we run out of memory.
 70| */
 71|struct file * get_empty_filp(void)
 72|{
 73|        static int old_max = 0;
 74|        struct file * f;
 75|
 76|        if (nr_free_files > NR_RESERVED_FILES) {
 77|        used_one:
 78|                f = free_filps; //list of empty file pointers
 79|                remove_filp(f); //use the first now
 80|                nr_free_files--;
 81|        new_one:
 82|                memset(f, 0, sizeof(*f));
 83|                f->f_count = 1;
 84|                f->f_version = ++event;
 85|                f->f_uid = current->fsuid;
 86|                f->f_gid = current->fsgid;
 87|                put_inuse(f); //add to list of used filp's
 88|                return f;
 89|        }
 90|      /*
 91|       * Use a reserved one if we're the superuser
 92|       */
 93|        if (nr_free_files && !current->euid)
 94|                goto used_one;
 95|      /*
 96|       * Allocate a new one if we're below the limit.
 97|       */
 98|        if (nr_files < max_files) {
 99|                f = kmem_cache_alloc(filp_cache, SLAB_KERNEL);
100|                if (f) {
101|                        nr_files++;
102|                        goto new_one;
103|                }
104|                /* Big problems... */
105|                printk("VFS: filp allocation failed\n");
106|
107|        } else if (max_files > old_max) {
108|                printk("VFS: file-max limit %d reached\n", max_files);
109|                old_max = max_files;
110|        }
```

```
111|        return NULL;
112|}
⋮
:
```

Extract of linux/fs/open.c:

```
⋮
688|/*
689| * Find an empty file descriptor entry, and mark it busy.
690| */
691|int get_unused_fd(void)
692|{
693|        struct files_struct * files = current->files;
694|        int fd, error;
695|
696|        error = -EMFILE;
697|        fd = find_first_zero_bit(&files->open_fds, NR_OPEN);
698|        // find_first_zero_bit is in linux/include/asm-i386/bitops.h
699|      /*
700|       * N.B. For clone tasks sharing a files structure, this test
701|       * will limit the total number of files that can be opened.
702|       */
703|        if (fd >= current->rlim[RLIMIT_NOFILE].rlim_cur)
704|                goto out;
705|
706|        /* Check here for fd > files->max_fds to do dynamic expansion */
707|
708|        FD_SET(fd, &files->open_fds);
709|        FD_CLR(fd, &files->close_on_exec);
710|#if 1
711|        /* Sanity check */
712|        if (files->fd[fd] != NULL) {
713|                printk("get_unused_fd: slot %d not NULL!\n", fd);
714|                files->fd[fd] = NULL;
715|        }
716|#endif
717|        error = fd;
718|
719|out:
720|        return error;
721|}
⋮
:
```

The above shown functions are not complete, so if you need more detailed information, take a look at the real kernel sources.

Chapter 6

Scheduling in Linux

6.1 The Linux Scheduler

The Linux scheduler is splitted in two levels: One level is responsible for user process scheduling, the other one for kernel handlers. These are functions in the kernel which handle work for user processes.

6.1.1 User Process Scheduling

A few facts first:

- Linux uses preemtive scheduling with a timeslice of $\sim$ 200 ms per process.
- Linux implements process aging (`update_process_times()`, *linux/kernel/sched.c, line 1461*).
- The scheduler combines priority scheduling with Round Robin or First In First Out. Thus, changing the priority of a process simply changes its timeslice.
- The scheduler has SMP-capabilities. This means that different processes may run on different CPUs. To prevent the scheduler from being entered of different CPUs at the same time, it has to be locked globally.
- Interrupts can occur during scheduling processes and add new runnable processes to the run queue.
- The scheduler supports "realtime processes". Realtime processes are scheduled using Round Robin or First In First Out strategy.

The data structure which is used for scheduling user processes is a double-linked ring which is called "run queue" (the dummy element is the init-task). This structure is called `task_struct` and declared in *linux/include/linux/sched.h*. The process table for all processes in the system is called `*task[NR_TASKS]` and can be found there, too. The runnable processes are connected with the attributes `next_run` and `prev_run`.
The scheduler itself is placed in the file *linux/kernel/sched.c* and is implemented in the function `schedule()`.

Now let's take a look at the scheduling mechanism in the function `schedule()`: First, the scheduler runs its own task queue (`run_task_queue(&tq_scheduler);`). Second, the scheduler looks if any kernel handlers wish to run (`if( bh_mask & bh_active)`). What is done here will be discussed in section 6.1.2. After that, the scheduler is ready to work on the user processes. This is done in the following code extract:

```
while (p != &init_task) {
  if (can_schedule(p)) {
    int weight = goodness(p, prev, this_cpu);
    if (weight > c)
      c = weight, next = p;
  }
  p = p->next_run;
}
```

The "weight" of a process is calculated in the function `goodness()`, which generally will return the attribute `counter` of the process. `counter` describes how many milliseconds a process has left to run. `goodness()` returns zero if the process "yields" the scheduler (gives up the CPU on its own). After running through all runnable processes, the scheduler hopefully has found the "heaviest" process and can switch to it. If no process is ready to run, the init-task [1] is selected.

6.1.2 Kernel Handler Scheduling

It is implemented in the file *linux/kernel/softirq.c*. Linux uses two variables, `bh_active` and `bh_mask`, which are checked at different places in the kernel (e.g. in the scheduler or when returning from a system call). Every bit in these variables represent a different bottom half handler. In combination with the array `void (*bh_base[32])(void)`, which allocates 32 pointers to functions in the kernel, the

[1] The init-task implements the functionality that is needed to initialize the kernel. It is only needed during boot time.

bottom-half scheduler knows which function it has to call if one bit in `bh_active` differs from the counterpart in `bh_mask`. `bh_mask` has its bits set according to which handlers are installed.

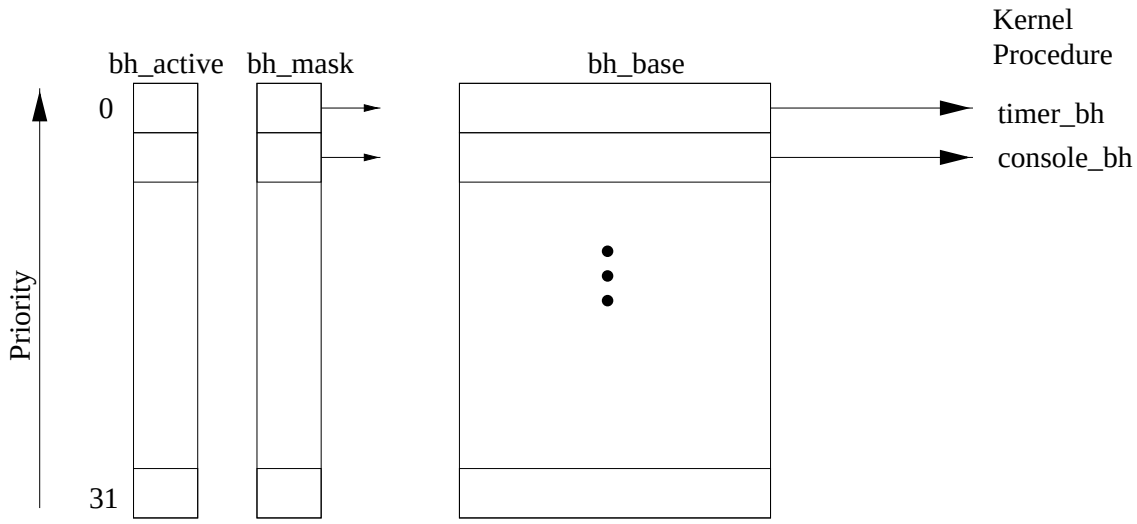


Figure 6.1: The data structure used in the bottom half scheduler

The bottom half handlers, which exist in the kernel are defined in the file *linux/include/linux/interrupt.h*. The handler for the timer has the highest priority.
The bottom half scheduler is placed in the following function:

```
 1: static inline void run_bottom_halves(void)
 2: {
 3:      unsigned long active;
 4:      void (**bh)(void);
 5:
 6:      active = get_active_bhs();
 7:      clear_active_bhs(active);
 8:      bh = bh_base;
 9:      do {
10:           if (active & 1)
11:                (*bh)();
12:           bh++;
13:           active >>= 1;
14:      } while (active);
15: }
```

It is quite simple and works as follows.
First, the scheduler looks which handlers want to run (line 6). In lines 9 to 14 the scheduler calls (line 11) all handlers which have work to do.

These handlers can implement their own scheduling strategy now.
Extract of sched.c:

```
 53|/* for linear pseudo random number generator */
 54|#define A      5
 55|#define C      1
 56|#define X0     1
 57|
 58|int z = X0;
⋮
⋮
704|static inline int suRANDOM( int m)
705|{
706|        z = ( z * A + C ) % m;
707|        return z;
708|
709|}
⋮
⋮
726|asmlinkage void schedule(void)
727|{
728|        struct schedule_data * sched_data;
729|        struct task_struct *prev, *next, *p;
730|        int this_cpu;
731|
732|        if (tq_scheduler)
733|                goto handle_tq_scheduler;
734|tq_scheduler_back:
735|
736|        prev = current;
737|        this_cpu = prev->processor;
738|
739|        if (in_interrupt())
740|                goto scheduling_in_interrupt;
741|
742|        release_kernel_lock(prev, this_cpu);
743|
744|        /* Do "administrative" work here while we don't hold any locks */
745|        if (bh_mask & bh_active)
746|                goto handle_bh;
747|handle_bh_back:
748|
```

```
749|      /*
750|       * 'sched_data' is protected by the fact that we can run
751|       * only one process per CPU.
752|       */
753|          sched_data = & aligned_data[this_cpu].schedule_data;
754|
755|          spin_lock_irq(&runqueue_lock);
756|
757|          /* move an exhausted RR process to be last.. */
758|          if (prev->policy == SCHED_RR)
759|                  goto move_rr_last;
760|move_rr_back:
761|
762|          switch (prev->state) {
763|                  case TASK_INTERRUPTIBLE:
764|                          if (signal_pending(prev)) {
765|                                  prev->state = TASK_RUNNING;
766|                                  break;
767|                          }
768|                  default:
769|                          del_from_runqueue(prev);
770|                  case TASK_RUNNING:
771|          }
772|          prev->need_resched = 0;
773|
774|/* not needed for lottery scheduling
775| * repeat_schedule:
776| */
777|
778|      /*
779|       * this is the scheduler proper:
780|       */
781|
782|
783|      /*
784|       * This is subtle.
785|       * Note how we can enable interrupts here, even
786|       * though interrupts can add processes to the run-
787|       * queue. This is because any new processes will
788|       * be added to the front of the queue, so "p" above
789|       * is a safe starting point.
790|       * run-queue deletion and re-ordering is protected by
791|       * the scheduler lock
792|       */
793|/*
794| * Note! there may appear new tasks on the run-queue during this, as
795| * interrupts are enabled. However, they will be put on front of the
796| * list, so our list starting at "p" is essentially fixed.
797| */
```

```
798|
799|/* ------------------------------------------
800| *  BEGIN: lottery scheduling changes
801| */
802|        next = NULL;
803|        {
804|          int tickets = 0;
805|          int winner, x;
806|
807|          /* get the number of tickets */
808|          for (p = init_task.next_run; p != &init_task; p = p->next_run)
809|                  if ( can_schedule(p) )
810|                          tickets += p->counter;
811|
812|          if ( tickets <= 0 ) {     /* rescheduling idle-task */
813|                  next = init_task.next_run;
814|                  goto done;
815|          }
816|
817|          /* get the ticket */
818|          winner = suRANDOM( tickets);
819|        /* printk("#Pr [%2d] tickets[%3d] won[%3d]\n",
820|        *       nr_running,tickets,winner);
821|        */
822|
823|          /* search for winner task */
824|          x = 0;
825|          for (p = init_task.next_run; p != &init_task; p = p->next_run) {
826|                  if ( can_schedule(p) ) {
827|                          x += p->counter;
828|                          if ( winner <= x ) {
829|                                  next = p;
830|                                  break;
831|                          }
832|                  }
833|          }
834|
835|
836|        }
837|
838|      /*
839|       * from this point on nothing can prevent us from
840|       * switching to the next task, save this fact in
841|       * sched_data.
842|       */
843|
844|done:
845|         sched_data->curr = next;
846|#ifdef __SMP__
```

```
847|        next->has_cpu = 1;
848|        next->processor = this_cpu;
849|#endif
850|        spin_unlock_irq(&runqueue_lock);
851|
852|        /* decrease the number of tickets of the previous process */
853|        if ( prev->counter > 30 )
854|                prev->counter -= 10;
855|
856|/*  END: lottery scheduling changes
857| *  ----------------------------------------
858| */
859|
860|        if ( prev == next )
861|                goto same_process;
862|#ifdef __SMP__
863|     /*
864|      * maintain the per-process 'average timeslice' value.
865|      * (this has to be recalculated even if we reschedule to
866|      * the same process) Currently this is only used on SMP,
867|      * and it's approximate, so we do not have to maintain
868|      * it while holding the runqueue spinlock.
869|      */
870|        {
871|                cycles_t t, this_slice;
872|
873|                t = get_cycles();
874|                this_slice = t - sched_data->last_schedule;
875|                sched_data->last_schedule = t;
876|
877|             /*
878|           * Exponentially fading average calculation, with
879|           * some weight so it doesnt get fooled easily by
880|           * smaller irregularities.
881|           */
882|                prev->avg_slice = (this_slice*1 + prev->avg_slice*1)/2;
883|        }
884|
885|     /*
886|      * We drop the scheduler lock early (it's a global spinlock),
887|      * thus we have to lock the previous process from getting
888|      * rescheduled during switch_to().
889|      */
890|
891|#endif /* __SMP__ */
892|
893|
894|        kstat.context_swtch++;
895|        get_mmu_context(next);
```

```
896|         switch_to(prev, next, prev);
897|         __schedule_tail(prev);
898|
899|same_process:
900|         reacquire_kernel_lock(current);
901|         return;
902|       /* not needed for lottery scheduling
903|        * recalculate:
904|        *     {
905|        *             struct task_struct *p;
906|        *             spin_unlock_irq(&runqueue_lock);
907|        *             read_lock(&tasklist_lock);
908|        *             for_each_task(p)
909|        *                     p->counter = (p->counter >> 1) + p->priority;
910|        *             read_unlock(&tasklist_lock);
911|        *             spin_lock_irq(&runqueue_lock);
912|        *             goto repeat_schedule;
913|        *     }
914|        */
915|
916|       /* not needed for lottery scheduling
917|still_running:
918|       c = prev_goodness(prev, prev, this_cpu);
919|       next = prev;
920|       goto still_running_back;
921|       */
922|handle_bh:
923|         do_bottom_half();
924|         goto handle_bh_back;
925|
926|handle_tq_scheduler:
927|         run_task_queue(&tq_scheduler);
928|         goto tq_scheduler_back;
929|
930|move_rr_last:
931|         if (!prev->counter) {
932|                 prev->counter = prev->priority;
933|                 move_last_runqueue(prev);
934|         }
935|         goto move_rr_back;
936|
937|scheduling_in_interrupt:
938|         printk("Scheduling in interrupt\n");
939|         *(int *)0 = 0;
940|         return;
941|}
⋮
```

Chapter 7

Memory Management in Linux

7.1 The Intel Pentium Architecture

The memory management facilities of the Intel Architecture are divided into two parts; segmentation and paging. *Segmentation* provides a mechanism of isolating individual code, data, and stack modules so that multiple tasks can run without interfering another. *Paging* provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed.

The Intel Architecture provides a *physical address space* of 4 GB ($= 2^{32}$). This is the address space that the processor can address on its address bus.

A *logical address* consists of a 16 bit segment selector and a 32 bit offset. The processor translates every logical address into a *linear address* in the processor's linear address space ($= 2^{32}$).

When using the i386's paging mechanism, the linear address space is divided into fixed-size pages (4 KB in length). The operating system maintains a two-level page directory and a set of page tables to keep track of the pages.

When a task attemps to access an address in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address.

If the page containig the linear address is not currently in physical memory, the processor generates a page-fault exception.

The processor stores the most recently used page-directories and page-table entries in on-chip caches called translation lookaside buffers (TLB).

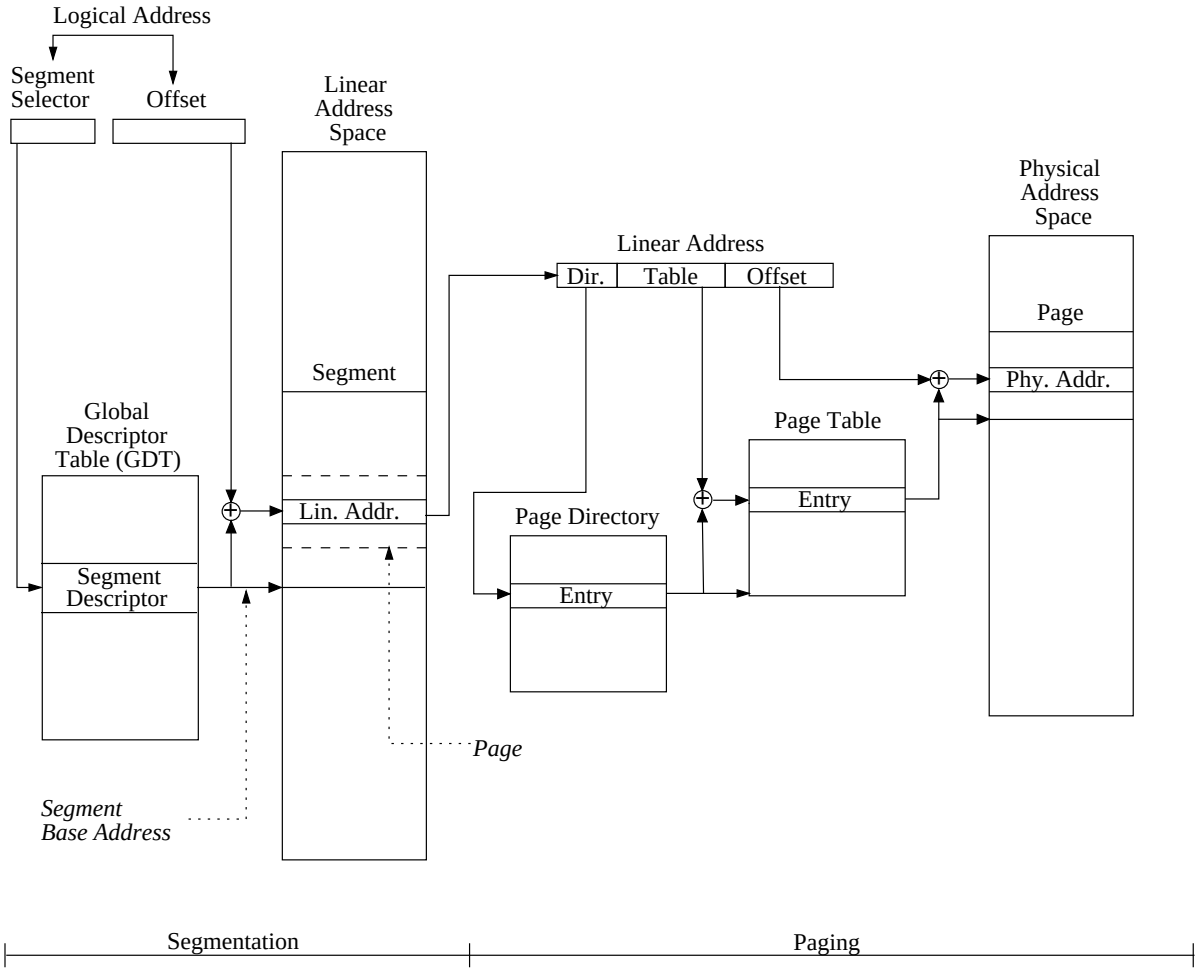


Figure 7.1: Segmentation and Paging

7.2 The Linux Memory Management

7.2.1 The Page Table Structure of Linux

The Linux memory management assumes a three-level page table setup shown in figure 7.2. As the i386 handles two levels, the middle level is "fold" into the top level. The three levels are the page directory level (PGD), the 2^{nd} page directory level (PMD), and the page table entries (PTE).

The functions and defines for modifying page tables can be found in `include/asm-i386/pgtable.h`.

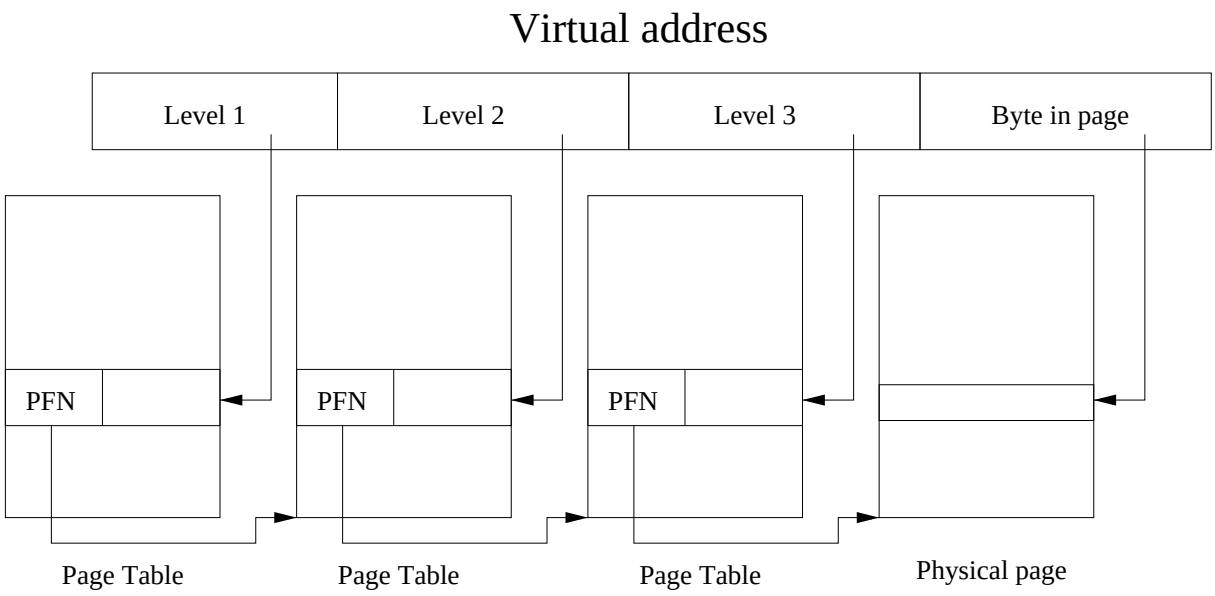


Figure 7.2: Three level page tables

7.2.2 Virtual Memory

Each process in the system has its own virtual address space. These virtual address spaces are completely separated from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by evil applications.
Memory mapping is used to map image and data files into a processes address space. The contents of a file are linked directly into the virtual address space of a process.
Every process has a pointer to one `mm_struct`, *linux/include/sched.h, line 163.* This struct contains a pointer to the head of `vm_area_structs`, *linux/include/linux/mm.h, line 34.* Every process keeps its own virtual memory area list, sorted by address to speed up the search if a page fault occurs. When there are more than 32 vmas, they are arranged in an AVL-tree. Every vma also includes a set of operations like open, close, nopage, swapin and so on. This way, Linux can use the same structure for executables, shared memory, data etc.

7.2.3 Page Allocation and Deallocation

Linux uses the Buddy algorithm to allocate and deallocate pages.

```
#define NR_MEM_LISTS 10
```

```
struct free_area_struct {
         struct page *next;
         struct page *prev;
         unsigned int * map;
};

static struct free_area_struct free_area[NR_MEM_LISTS];
```

Pages are allocated in blocks, which are powers of 2 in size. Thus, each element of `free_area`, *linux/mm/page_alloc.c, line 48* contains information about blocks of pages. The first element in the array describes single pages available in the system, the second element holds blocks of 2 pages, the third element blocks of 4 pages and so on.
Therefore, the pages are linked using the data structure `free_area_struct`, *linux/mm/page_alloc.c, line 40*. The `map`-element of this structure contains information of the blocks in one element. This is needed to merge two small page blocks into a bigger one.

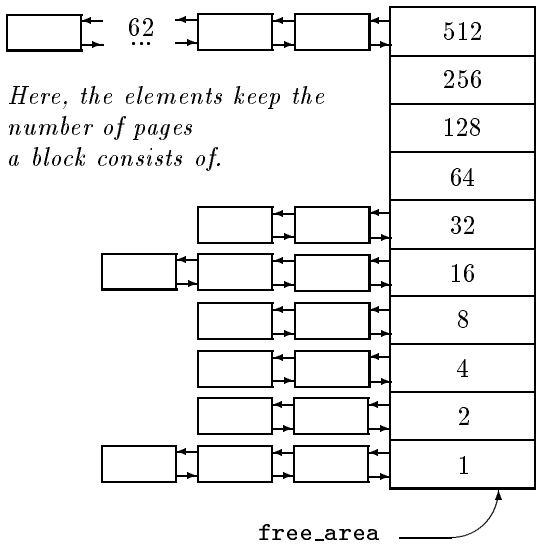


Figure 7.3: Free area array with free blocks.

Every page request ends up in the function `__get_free_pages()`, *linux/mm/page_alloc.c, line 194*. This function uses the macro `RMQUEUE`, *linux/mm/page_alloc.c, line 158* (= *Remove from Memory Queue*) to allocate a block of pages of the requested size.

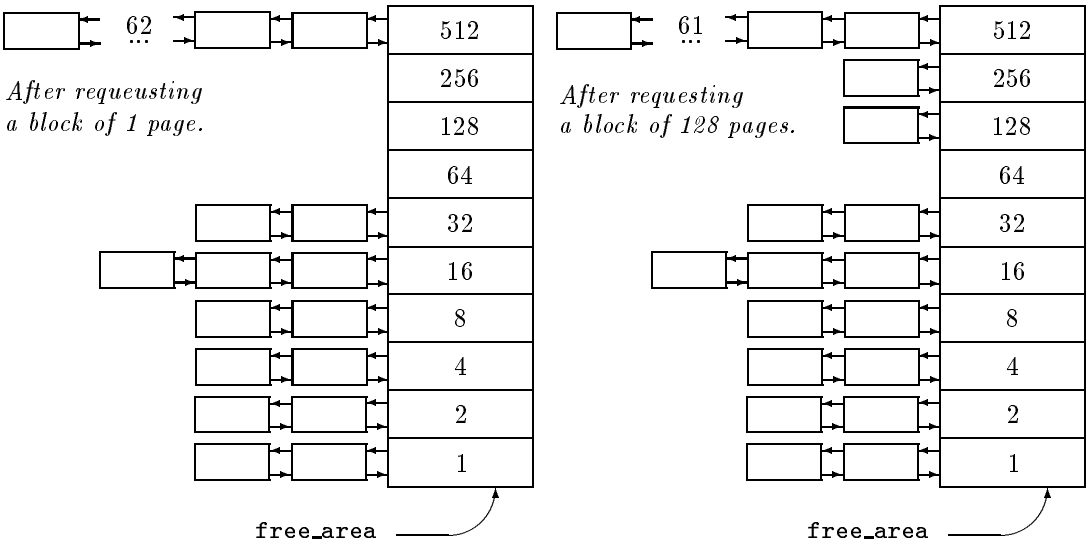


Figure 7.4: Two requests for blocks of 1 and 128 pages, respectively.

When freeing pages, `free_pages()`, *linux/mm/page_alloc.c, line 133* is called. If only one page is freed, the function `__free_page()`, *linux/mm/page_alloc.c, line 122* is called. Both functions call `free_pages_ok()`, *linux/mm/page_alloc.c, line 93*.
This is the counterpart of the macro `RMQUEUE`. When a block of pages is freed, the buddy block is checked to see if it is free. In this case, both blocks are merged. As it is twice as big, it is inserted into its appropriate queue. Note that this processing is recursive.

7.2.4 Page Sharing

Page sharing is possible in three cases:

- The fork system call creates a copy of a process.
- A shared library is accessed by more than one processes.
- We have a System V shared memory area.

The fork system call is placed in the file *linux/kernel/fork.c, line 534* in a function called `do_fork()`. To do memory-related work, `copy_mm()`, *linux/kernel/fork.c, line 358* is called. When calling `dup_mmap()`, the kernel wants to copy the memory mapping of the parent to the child. The function `copy_page_range()`, *linux/mm/memory.c, line 189* does not copy physical pages, it copies the PTEs

of the parent process to the child and marks all pages for both processes write-protected. If one wants to write such a page, it is copied (Copy-On-Write).
To load a shared library, the function `do_load_elf_library()`, *linux/fs/binfmt_elf.c line 901* is called. This function calls `do_mmap()`, *linux/mm/mmap.c, line 172* to map the library file to the memory. To load the pages into memory, the function `make_pages_present()`, *linux/mm/memory.c, line 955* is called. This function loads the whole file into memory, so there is no demand loading for shared libraries.
Pages for System V shared memory areas are created using page faults. The code for creating a shared memory page is in the function `shm_nopage()`, *linux/ipc/shm.c, line 607*. This means there is a sort of "demand loading" implemented for shared memory, because a shared memory page is only created, if at least one process is accessing it.

7.2.5 How Linux handles a Page Fault

You should read this section and the code of the kernel at the same time to understand what's going on.
The ball starts rolling in the function `do_page_fault()`, *linux/arch/i386/mm/fault.c*. The function is set as page fault handler in the file *linux/arch/i386/kernel/entry.S*, which sets all other architecture specific fault handlers, too.
If a valid virtual memory area (`find_vma()`, *linux/mm/mmap.c, line 381*) exists for that page fault and the page is owned by the current process, the function `handle_mm_fault()` is called, if not the kernel signals a SIGSEGV to the current process. Kernel page faults are managed especially.
`handle_mm_fault()`, *linux/mm/memory.c, line 932* extracts the PTE offset of the page fault address and creates a new PTE if needed or extracts the PTE from the fault address (`pte_alloc()`, *linux/include/asm-i386/pgtable.h, line 507*). Now we know the page table entry for the faulting address and pass over to `handle_pte_fault()`.
`handle_pte_fault()`, *linux/mm/memory.c, line 899* has to handle 3 possibilities: The PTE is new, the page is swapped out, or it is a copy-on-write (COW) page table entry.
`do_no_page()`, *linux/mm/memory.c, line 843* manages the case of a really new page. Most time we will have a `vma->vm_ops->nopage`-function, so the real work is done there. This function depends on the underlying file system and is for example defined in *linux/mm/filemap.c, line 930* in the function `filemap_nopage()`. This function tries to find the page in the page cache. If it is not present there, it reads the entire cluster of the executable from the disk. This function implements demand loading, this means if you want to run a executable file from

your disk (`do_execve()`, *linux/fs/exec.c, line 804*), the kernel does not load the whole executable into the memory, only the file header. Page faults do the actual loading.
`do_swap_page()`, *linux/mm/memory.c, line 786* handles the case of a swapped-out data page. As `do_no_page()`, it checks for an `vma->vm_ops->swapin()` function first. If there is none, it takes the generic one, `swapin()` which can be found in *linux/mm/page_alloc.c, line 390*.
`do_wp_page()`, *linux/mm/memory.c, line 616* handles the case of a COW page. Such a PTE is created during the fork system call. All data PTEs from the child process are marked as copy-on-write and are physically copied when the child wants to write on that page.

7.2.6 The Kernel Swap Daemon

Pages are swapped out by the kernel swap daemon (`kswapd()`, *linux/mm/vmscan.c, line 454*). The kernel swap daemon is a kernel thread. It has an entry in the process table, but it runs in kernel mode. Type *ps aux* on a console and you will find its pid.
The `ksapwd` is woken once a second. It is woken earlier if the system is low on memory. In `__get_free_pages()`, *linux/mm/page_alloc.c, line 206*, the function `try_to_free_pages()`, *linux/mm/vmscan.c, line 515* is called, which wakes the `kswapd`.
The `kswapd` checks if the number of free pages is greater or equal than `freepages.high`. If that is the case, it sleeps until it is woken again.
Otherwise it calls `do_try_free_pages()`, *linux/mm/vmscan.c, line 379*, which first of all trims the SLAB caches.
In the follwoing, it tries to free a maximum of `SWAP_CLUSTER_MAX` (= 32) pages by

- Reducing the size of the buffer and page caches,
- Swapping out System V shared memory pages,
- Swapping out and discarding pages.

`shrink_mmap`. *linux/mm/filemap.c, line 136* uses the clock algorithm to free pages. Pages from the swap cache, buffer cache, and page cache are freed.
When swapping out ordinary pages, `swap_out()`, *linux/mm/vmscan.c, line 308* is called. This function selects the process with the most resident pages and calls `swap_out_process()`, *linux/mm/vmscan.c, line 268*, which basically tries to swap out one page of the selected process.
If one page successfully was freed, it returns. Otherwise `swap_out()` uses a given `priority` to determine how many processes it should try.

7.2.7 Caches

Well, Linux has many of them. We will describe in this section only a few of those which affect memory management.

- *Page Cache:*
  Whenever a page is read from a memory mapped file, the page is read through the page cache. The page cache consists of the `page_hash_table`, which is a vector of pointers to `mem_map_t` data structures (*linux/mm/pagemap.c*).

  As each file in Linux is identified by a VFS inode data structure and each VFS inode is unique and fully describes one and only one file, the index into the page table is derived from the file's VFS inode and the offset into the file.

  Read the section about Page Fault Handling to understand how it works.

- *Buffer cache:*
  All block devices are viewed as linear collections of blocks of the same size. To speed up access to the physical block devices, Linux maintains a cache of block buffers.

  If valid data is available from the buffer cache, an access to a physical device is saved. Any block buffer that has been used to read data from a block device or to write data to it goes into the buffer cache.

  All block data read and write requests are given to the device drivers in the form of `buffer_head` data structures via standard kernel routine calls. These give all of the information that the block device drivers need; the device identifier uniquely identifies the device and the block number tells the driver which block to read.

  Buffers are created via `create_buffers()`, *linux/fs/buffer.c, line 1029* and may be synchronous or asynchronous buffers.

  The most important function here is `brw_page()`, *fs/buffer.c, line 1212.* This function will create a buffer for asynchronous IO. The function `ll_rw_block()`, *drivers/block/ll_rw_block.c, line 569* will fill this buffer. When IO has completed, `mark_buffer_uptodate()`, *fs/buffer.c* will mark the buffer as up-to-date. Now a user process may access the requested data using the buffer. Freeing buffers is done from a kernel thread called `bdflush()`, *fs/buffer.c, line 1373.*

7.3 Homework

Change the Linux swapping algorithm. Choose the process that has used least CPU time relative to its start time.

7.4 Solution

First, you have to extend the `mm_struct` in `linux/include/linux/sched.h` to store its relative start time.
`Extract of sched.h:`

```
...
163|struct mm_struct {
164|        struct vm_area_struct *mmap;             /* list of VMAs */
165|        struct vm_area_struct *mmap_avl;         /* tree of VMAs */
166|        struct vm_area_struct *mmap_cache;       /* last find_vma result */
167|        pgd_t * pgd;
168|        atomic_t count;
169|        int map_count;                            /* number of VMAs */
170|        struct semaphore mmap_sem;
171|        unsigned long context;
172|        unsigned long start_code, end_code, start_data, end_data;
173|        unsigned long start_brk, brk, start_stack;
174|        unsigned long arg_start, arg_end, env_start, env_end;
175|        unsigned long rss, total_vm, locked_vm;
176|        unsigned long def_flags;
177|        unsigned long cpu_vm_mask;
178|        unsigned long swap_cnt; /* number of pages to swap on next pass */
179|        unsigned long swap_address;
180|      /*
181|       * This is an architecture-specific pointer: the portable
182|       * part of Linux does not know about any segments.
183|       */
184|          void * segments;
185|   /*unsigned long calc_time;*/
186|        unsigned long suTIME;
187|};
...
```

To initialize this value, a little change in `linux/kernel/fork.c` is necessary.
The main work is done in `linux/mm/vmscan.c` in the function `swap_out()` where – depending on the priority – from a number of processes one page is tried to be swapped out.
For every existing process in the system, the `cpu_share` is calculated. The `cpu_share` is the relation of the CPU time to the the start time of a process.
When a process is selected, a page of this process is tried to be swapped out. If

this is not successful, the `suTIME` is set to `jiffies` to guarantee that all other processes are more likely to be selected.

`Extract of suVMSCAN.c:`

```
303|
304|static int swap_out(unsigned int priority, int gfp_mask)
305|{
306|        struct task_struct * p, * pbest;
307|        int counter, assign, max_cnt;
308|        long cpu_max, cpu_share;
309|        long time;
310|        int i;
311|
312|
313|        counter = nr_tasks / (priority+1);
314|        if (counter < 1)
315|                counter = 1;
316|        if (counter > nr_tasks)
317|                counter = nr_tasks;
318|
319|        for (; counter >= 0; counter--) {
320|                assign = 0;
321|                max_cnt = 0; cpu_max = 0; cpu_share = 0;
322|                pbest = NULL;
323|        select:
324|                read_lock(&tasklist_lock);
325|                p = init_task.next_task; cpu_max = 0;
326|                for (; p != &init_task; p = p->next_task) {
327|                        if (!p->swappable)
328|                          continue;
329|#ifdef __SMP__
330|                        time = 0;
331|                        for (i=0; i<NR_CPUS; i++)
332|                          time += p->per_cpu_utime[i] +
p->per_cpu_stime[i];
333|#else
334|                        time = p->per_cpu_utime[0] + p->per_cpu_stime[0];
335|#endif /* __SMP__ */
336|                        cpu_share = (jiffies - p->mm->suTIME) / (time+1);
337|                        if ( cpu_share > cpu_max ) {
338|                          cpu_max = cpu_share;
339|                          pbest = p;
340|                        }
341|                }
342|                read_unlock(&tasklist_lock);
343|                if (!pbest) {
344|                    printk("Swap: no process found!\n");
345|                    goto out;
```

```
346|                }
347|
348|                printk("Swap: process (%d) found. share: (%d)!\n",
349|                       pbest->pid,cpu_max);
350|                if (swap_out_process(pbest, gfp_mask)) {
351|                    printk("Swap successful.");
352|                    return 1;
353|                }
354|                pbest->mm->suTIME = jiffies;
355|                printk("Swap not successful.\n");
356|        }
357|out:
358|        return 0;
359|}
```

⋮

Chapter 8

Linux File Systems

8.1 The Virtual File System

Linux uses a three-tier layered architecture to make different types of file systems accessible.
The *kernel* communicates with every *file system* via the *Virtual File System* (= VFS). The VFS offers well defined structures for super-blocks, inodes, and directory entries, which every file system has to implement.

8.1.1 VFS Inodes in `struct inode`

Amongst other information, VFS inodes contain the following fields:

- The *inode number* `i_ino` is unique within this file system.
- A *device identifier* `i_dev` of the device holding this VFS inode.
- A *reference counter* `i_count`.
- The *mode* with rwx-rights and the VFS inode's type (directory, regular file, special file (character, block, pipe device)).
- The owner's user and group id `i_uid, i_gid`.
- Access times `i_atime, i_mtime, i_ctime`.
- The size `i_size`.
- A rounded number of blocks `i_blocks`.
- A pointer ot its operations `*i_op`.

- A pointer to its super-block `*i_sb`.
- The `union u` stores specific inode information for every file system.

8.1.2 VFS Super-Blocks in `struct super_block`

The `struct super_block` stores the block device that this file system is contained in, the block size in bytes, a mountstate flag (rw/ro) and a dirty flag. Every file system has a different magic number, so you cannot make the mistake to mount and treat an ext2-fs as an minix-fs. A super-block knows about the `struct file_system_type *s_type`, quota-specific operations and the `struct super_operations *s_op`. The root directory is in `struct dentry *s_root`, all dirty inodes go to the `s_dirty` list. The `union u` stores specific super-block information for every file system.

8.1.3 `struct inode_operations`

As we all know, every inode is linked into a directory structure, stores some information about rights etc. and – this is the most important thing – points to the blocks on disk where the data is stored.
To manipulate the data itself, we need the `struct file_operations *default_file_ops`. Possible inode manipulations are create, link, unlink, rename, and – if it is a directory inode – mkdir, rmdir, and some more.

8.1.4 `struct file_operations`

A file may be opened, released, read or written at a `lseek`ed position, or `sync`ed to get all buffers out to disk immediately.

8.1.5 `struct super_operations`

These function pointers are used to read, write, or delete `struct inode`s, but also to update the super-block on disk, get information about free blocks eg. on the partition via `statfs`.

8.1.6 `struct dquot_operations`

When a file quota for a user is set and the fs supports quota, these function calls are used to do the accounting of block and inode usage.

8.1.7 `struct file_system_type`

This struct is used by the VFS to get a linked list of all registered file systems. Before a file system can be mounted, the corresponding type has to be registered. This may happen at boot time, or as a loadable module, which may be unregistered (unloaded) when not needed. Every file system has a name like "minix", "ext2", "msdos" etc. They are mounted by calling a fs-specific `read_super`, which fills a given `struct super_block` with the file system specific information.

8.1.8 VFS struct definitions

`Extract of linux/include/linux/fs.h:`

```
⋮
332|struct inode {
333|        struct list_head        i_hash;
334|        struct list_head        i_list;
335|        struct list_head        i_dentry;
336|
337|        unsigned long           i_ino;
338|        unsigned int            i_count;
339|        kdev_t                  i_dev;
340|        umode_t                 i_mode;
341|        nlink_t                 i_nlink;
342|        uid_t                   i_uid;
343|        gid_t                   i_gid;
344|        kdev_t                  i_rdev;
345|        off_t                   i_size;
346|        time_t                  i_atime;
347|        time_t                  i_mtime;
348|        time_t                  i_ctime;
349|        unsigned long           i_blksize;
350|        unsigned long           i_blocks;
351|        unsigned long           i_version;
352|        unsigned long           i_nrpages;
353|        struct semaphore        i_sem;
354|        struct semaphore        i_atomic_write;
355|        struct inode_operations *i_op;
356|        struct super_block      *i_sb;
357|        struct wait_queue       *i_wait;
358|        struct file_lock        *i_flock;
359|        struct vm_area_struct   *i_mmap;
360|        struct page             *i_pages;
361|        struct dquot            *i_dquot[MAXQUOTAS];
362|
363|        unsigned long           i_state;
364|
365|        unsigned int            i_flags;
```

```
366|        unsigned char           i_pipe;
367|        unsigned char           i_sock;
368|
369|        int                     i_writecount;
370|        unsigned int            i_attr_flags;
371|        __u32                   i_generation;
372|        union {
373|                struct pipe_inode_info          pipe_i;
374|                struct minix_inode_info         minix_i;
375|                struct ext2_inode_info          ext2_i;
376|                struct hpfs_inode_info          hpfs_i;
377|                struct ntfs_inode_info          ntfs_i;
378|                struct msdos_inode_info         msdos_i;
379|                struct umsdos_inode_info        umsdos_i;
380|                struct iso_inode_info           isofs_i;
381|                struct nfs_inode_info           nfs_i;
382|                struct sysv_inode_info          sysv_i;
383|                struct affs_inode_info          affs_i;
384|                struct ufs_inode_info           ufs_i;
385|                struct romfs_inode_info         romfs_i;
386|                struct coda_inode_info          coda_i;
387|                struct smb_inode_info           smbfs_i;
388|                struct hfs_inode_info           hfs_i;
389|                struct adfs_inode_info          adfs_i;
390|                struct qnx4_inode_info          qnx4_i;
391|                struct socket                   socket_i;
392|                void                            *generic_ip;
393|        } u;
394|};
⋮
⋮
414|struct file {
415|        struct file             *f_next, **f_pprev;
416|        struct dentry           *f_dentry;
417|        struct file_operations  *f_op;
418|        mode_t                  f_mode;
419|        loff_t                  f_pos;
420|        unsigned int            f_count, f_flags;
421|        unsigned long           f_reada, f_ramax, f_raend, f_ralen,
f_rawin;
422|        struct fown_struct      f_owner;
423|        unsigned int            f_uid, f_gid;
424|        int                     f_error;
425|
426|        unsigned long           f_version;
427|
428|        /* needed for tty driver, and maybe others */
429|        void                    *private_data;
```

```
430|};
 ⋮
 ⋮
518|struct super_block {
519|        struct list_head        s_list;         /* Keep this first */
520|        kdev_t                  s_dev;
521|        unsigned long           s_blocksize;
522|        unsigned char           s_blocksize_bits;
523|        unsigned char           s_lock;
524|        unsigned char           s_rd_only;
525|        unsigned char           s_dirt;
526|        struct file_system_type *s_type;
527|        struct super_operations *s_op;
528|        struct dquot_operations *dq_op;
529|        unsigned long           s_flags;
530|        unsigned long           s_magic;
531|        unsigned long           s_time;
532|        struct dentry            *s_root;
533|        struct wait_queue       *s_wait;
534|
535|        struct inode            *s_ibasket;
536|        short int               s_ibasket_count;
537|        short int               s_ibasket_max;
538|        struct list_head        s_dirty;        /* dirty inodes */
539|
540|        union {
541|                struct minix_sb_info    minix_sb;
542|                struct ext2_sb_info     ext2_sb;
543|                struct hpfs_sb_info     hpfs_sb;
544|                struct ntfs_sb_info     ntfs_sb;
545|                struct msdos_sb_info    msdos_sb;
546|                struct isofs_sb_info    isofs_sb;
547|                struct nfs_sb_info      nfs_sb;
548|                struct sysv_sb_info     sysv_sb;
549|                struct affs_sb_info     affs_sb;
550|                struct ufs_sb_info      ufs_sb;
551|                struct romfs_sb_info    romfs_sb;
552|                struct smb_sb_info      smbfs_sb;
553|                struct hfs_sb_info      hfs_sb;
554|                struct adfs_sb_info     adfs_sb;
555|                struct qnx4_sb_info     qnx4_sb;
556|                void                    *generic_sbp;
557|        } u;
558|      /*
559|       * The next field is for VFS *only*. No filesystems have any business
560|       * even looking at it. You had been warned.
561|       */
562|        struct semaphore s_vfs_rename_sem;      /* Kludge */
```

```
563|};
⋮
⋮
580|struct file_operations {
581|        loff_t (*llseek) (struct file *, loff_t, int);
582|        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
583|        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
584|        int (*readdir) (struct file *, void *, filldir_t);
585|        unsigned int (*poll) (struct file *, struct poll_table_struct *);
586|        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
587|        int (*mmap) (struct file *, struct vm_area_struct *);
588|        int (*open) (struct inode *, struct file *);
589|        int (*flush) (struct file *);
590|        int (*release) (struct inode *, struct file *);
591|        int (*fsync) (struct file *, struct dentry *);
592|        int (*fasync) (int, struct file *, int);
593|        int (*check_media_change) (kdev_t dev);
594|        int (*revalidate) (kdev_t dev);
595|        int (*lock) (struct file *, int, struct file_lock *);
596|};
597|
598|struct inode_operations {
599|        struct file_operations * default_file_ops;
600|        int (*create) (struct inode *,struct dentry *,int);
601|        struct dentry * (*lookup) (struct inode *,struct dentry *);
602|        int (*link) (struct dentry *,struct inode *,struct dentry *);
603|        int (*unlink) (struct inode *,struct dentry *);
604|        int (*symlink) (struct inode *,struct dentry *,const char *);
605|        int (*mkdir) (struct inode *,struct dentry *,int);
606|        int (*rmdir) (struct inode *,struct dentry *);
607|        int (*mknod) (struct inode *,struct dentry *,int,int);
608|        int (*rename) (struct inode *, struct dentry *,
609|                        struct inode *, struct dentry *);
610|        int (*readlink) (struct dentry *, char *,int);
611|        struct dentry * (*follow_link) (struct dentry *, struct dentry *,
unsigned int);
612|        int (*readpage) (struct file *, struct page *);
613|        int (*writepage) (struct file *, struct page *);
614|        int (*bmap) (struct inode *,int);
615|        void (*truncate) (struct inode *);
616|        int (*permission) (struct inode *, int);
617|        int (*smap) (struct inode *,int);
618|        int (*updatepage) (struct file *, struct page *, unsigned long,
unsigned int, int);
619|        int (*revalidate) (struct dentry *);
620|};
621|
```

```
622|struct super_operations {
623|        void (*read_inode) (struct inode *);
624|        void (*write_inode) (struct inode *);
625|        void (*put_inode) (struct inode *);
626|        void (*delete_inode) (struct inode *);
627|        int (*notify_change) (struct dentry *, struct iattr *);
628|        void (*put_super) (struct super_block *);
629|        void (*write_super) (struct super_block *);
630|        int (*statfs) (struct super_block *, struct statfs *, int);
631|        int (*remount_fs) (struct super_block *, int *, char *);
632|        void (*clear_inode) (struct inode *);
633|        void (*umount_begin) (struct super_block *);
634|};
635|
636|struct dquot_operations {
637|        void (*initialize) (struct inode *, short);
638|        void (*drop) (struct inode *);
639|        int (*alloc_block) (const struct inode *, unsigned long, uid_t,
char);
640|        int (*alloc_inode) (const struct inode *, unsigned long, uid_t);
641|        void (*free_block) (const struct inode *, unsigned long);
642|        void (*free_inode) (const struct inode *, unsigned long);
643|        int (*transfer) (struct inode *, struct iattr *, char, uid_t);
644|};
645|
646|struct file_system_type {
647|        const char *name;
648|        int fs_flags;
649|        struct super_block *(*read_super) (struct super_block *, void *,
int);
650|        struct file_system_type * next;
651|};
652|
653|extern int register_filesystem(struct file_system_type *);
654|extern int unregister_filesystem(struct file_system_type *);
```

⋮

Extract of linux/include/linux/dcache.h:

⋮

```
 56|#define DNAME_INLINE_LEN 16
 57|
 58|struct dentry {
 59|        int d_count;
 60|        unsigned int d_flags;
 61|        struct inode  * d_inode;         /* Where the name belongs to - NULL is
negative */
 62|        struct dentry * d_parent;        /* parent directory */
 63|        struct dentry * d_mounts;        /* mount information */
 64|        struct dentry * d_covers;
```

```
65|        struct list_head d_hash;          /* lookup hash list */
66|        struct list_head d_lru;           /* d_count = 0 LRU list */
67|        struct list_head d_child;         /* child of parent list */
68|        struct list_head d_subdirs;       /* our children */
69|        struct list_head d_alias;         /* inode alias list */
70|        struct qstr d_name;
71|        unsigned long d_time;             /* used by d_revalidate */
72|        struct dentry_operations  *d_op;
73|        struct super_block * d_sb;        /* The root of the dentry tree */
74|        unsigned long d_reftime;          /* last time referenced */
75|        void * d_fsdata;                  /* fs-specific data */
76|        unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
77|};
78|
79|struct dentry_operations {
80|        int (*d_revalidate)(struct dentry *, int);
81|        int (*d_hash) (struct dentry *, struct qstr *);
82|        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
83|        void (*d_delete)(struct dentry *);
84|        void (*d_release)(struct dentry *);
85|        void (*d_iput)(struct dentry *, struct inode *);
86|};
87|
88|/* the dentry parameter passed to d_hash and d_compare is the parent
89| * directory of the entries to be compared. It is used in case these
90| * functions need any directory specific information for determining
91| * equivalency classes. Using the dentry itself might not work, as it
92| * might be a negative dentry which has no information associated with
93| * it */
```

⋮

8.2 A VFS Implementation: minix-fs

I try to show the general thoughts of a VFS implementation, but examples are described by showing some details of the minix-fs. The Minix code is maybe not the simplest one because of the two different minix versions existing out there, but the implementation of an inode-based unix-fs is straight forward.

8.2.1 Basic Ideas about Partitions and Inode-Based Unix-fs

Every partition on disk (or every loopback-mounted file) has a boot sector of 512 bytes.

8.2.1.1 Super-Block

The Linux kernel has a predefined `BLOCK_SIZE` of 1K. As the first 512 bytes are used by the boot block – it needs one block, thus 512 bytes are wasted – so the super-block starts at 1024 bytes. The super-block stores all important information about the whole partition like the number of available blocks, the number of inodes, if the device was cleanly unmounted, the address of the start of data blocks, a magic number to identify the file system-type, and the location of the root-inode.

8.2.1.2 Inodes and Directories and Files

On the other hand, – as users see it – every file system consists of directories with files in it. Directories and files are called directory entries (`dentry`). These dentries point to inodes. So if the file system allows hard links, multiple dentries may point to the same inodes and – by that – to the same data.
Symbolic links are just inodes pointing to other inodes with the data, so it is an indirect addressing.
Inodes refer to data blocks on disk, have access times, access rights and special modes. You may query the mode using the following defines found in `linux/include/linux/stat.h`:

```
#define S_ISLNK(m)      (((m) & S_IFMT) == S_IFLNK)  /* link */
#define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)  /* regular */
#define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)  /* directory */
#define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)  /* character device */
#define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)  /* block device */
#define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)  /* fifo (=named pipe) */
#define S_ISSOCK(m)     (((m) & S_IFMT) == S_IFSOCK) /* socket */
```

The root inode is set to `S_IFDIR` and is the one, which is visible when the file system is mounted on the old mount-point directory. Inodes are numbered starting at 1 for the root inode. Due to the Linux kernel implementation, 0 is not a valid inode number.
So every file system has to store a list of all inodes on disk, but where are the directories with their files in it? They are stored into `S_IFDIR` inodes data blocks. A list of `struct minix_dir_entry` is stored into its data blocks, including the "." and ".." entries. The "." and ".." entries of the root-inode have to point back to the root-inode number again.
To speed up file operations, the file system should additionally keep track of all free&used data blocks and of all free&used inodes. (eg. a bitfield)

A file system does not need to use data blocks of `BLOCK_SIZE` size, but a multiple of it, because of data locality of sequential read&write. Minix calls this a zone, Micro$ofts MS-DOS uses the same principle and calls it a cluster.
The Minix file system always uses a zone size of 1KB, accidentially equal to the kernel's `BLOCK_SIZE`. Ext2-fs lets you choose between 1,2,4,8 K, MS-DOS decides at formatting-time about the right size to squeeze a block-bit field in the super-block with a maximum of 2^{16} bits. This is needed to access the bit field with a 16-bit short integer.
To make our following examples more interesting, we will use a zone (= cluster) size of 4K, even when we talk about Minix.

8.2.1.3 Data Block Access

Every inode has to know where to find its data blocks on disk. Since every data block can be referenced by a unique number, we only need an array of integers. (ATTENTION: This little decision already restricts the maximum size of a partition to 2^{32} blocks. Given a block size of 4 K, this means only 16 Gigabyte!)
To speed up inode access, we need to have all inodes to be the same size on disk. So let's say, we store an array with 2^{14} integers for a maximum file size of 64 M. 64 M are not much but we already have to add an array of $16384 * 4 = 64K$ to each and every inode!
This is why Minix uses indirect access with an array of only 10 zone pointers. 0-7 are for accessing real data blocks directly, number 8 points to a data block where a list of real data blocks is stored (indirect access), number 9 points to a data block with a list of data blocks, which then point to the real data blocks (double indirect access), number 10 is triple indirect access.
Number 8-10 are only used when needed!
Given a block size of 4 K, we may access $8 * 4 = 32K$ directly. If we use a block as a list of integer pointers, we can store 1024 pointers. So with single indirect access we reach $1024 * 4 = 4M$. Double indirect access is for $1024 * 1024 * 4 = 4G$ and with triple indirect access we go to $1024 * 1024 * 1024 * 4 = 4T$.

8.2.2 Including a File System to the Kernel

8.2.2.1 Let the Kernel Know...

In `linux/fs/Config.in` all filesystems are listed for the menu of the `make xconfig` command. Here you may enter your own one with a tri-state (yes—module—off) button and a variable name like `CONFIG_MINIX_FS`.

In the `linux/fs/Makefile` you need to have your file system directory added to `ALL_SUB_DIRS`. Your file system has to be added to the sub directories of either the `kernel` or the `modules` during compile time.

```
ifeq ($(CONFIG_MINIX_FS),y)
SUB_DIRS += minix
else
  ifeq ($(CONFIG_MINIX_FS),m)
  MOD_SUB_DIRS += minix
  endif
endif
```

Every file system has its own structs of how the data is stored on disk. So you find a `struct minix2_inode`, a `minix_super_block` and a `minix_dir_entry` stored in `linux/include/linux/minix_fs.h`.
In `linux/include/linux/minix_fs_i.h` and `linux/include/linux/minix_fs_sb.h` you find the definitions of the inode and super-block specific structs `struct minix_inode_info` and `struct minix_sb_info` .
There are some changes to `linux/include/linux/fs.h`:

- The special file system dependent `struct minix_inode_info` has to be added to the struct inodes union `u`.
- The special file system dependent `struct minix_sb_info` has to be added to the `struct super_block`s union `u`.
- Do not forget to `#include` the `.h`-files!

8.2.2.2 Initialization and Mounting

The kernel has a list of all installed file systems, so we have to fill the `struct file_system_type` with the name of our file system and the main function to read (mount) the superblock from disk.

```
static struct super_block *myfs_read_super(struct super_block *s,
                                           void *data, int silent);

static struct file_system_type myfs_type = {
        "myfs",
        FS_REQUIRES_DEV,
        myfs_read_super,
        NULL
};
```

Our function `myfs_read_super` has to read the super-block from disk and to fill the given pointer variable `struct super_block *s`. Use `bread` to directly read a block from a device, but do not forget to release it with `brelse`, if there were any problems!
When you fill the super-block, you also have to set the file system specific `struct super_operations`. Then we need to call `myfs_read_inode` to get our root inode for the super-block in memory.

8.2.2.3 Mounting The Root Inode

When the `myfs_read_super` was successful, the kernel uses the root inode (which is a directory inode) to read the root directory. It calls `myfs_readdir` from the `struct inode_operations i_op`, which has to read the data blocks from this inode, where the directory entries are stored. `myfs_read_dir` calls `myfs_bread` to get the blocks from disk.

8.2.3 File System Specific Operations

8.2.3.1 myfs_read_inode

signature: `static void myfs_read_inode(struct inode inode)`

`myfs_read_inode` gets an empty `struct inode` with only `inode->i_ino` set. This is enough to find it on disk and fill the rest of the struct.
Depending on the inodes mode, set the inode operations to file-ops or dir-ops.

8.2.3.2 myfs_bread and myfs_bmap

signature: `struct buffer_head myfs_bread(struct inode inode, int block, int create)`
signature: `static int myfs_bmap(struct inode inode, int block)`

Since an inodes reference to a data block does not reflect the correct position on disk nor the same block size, there has to be a mapping.
The kernel's `bread` only reads `BLOCK_SIZE` sized blocks from a given device, starting at block 0.
For example, Minix has a super-block, inode-tables, zone-tables, and inodes stored in the first `N` blocks of the device, so the real data blocks start way behind the "real" block 0.
Minix-inodes reference their data blocks starting with 1 and use a different block size. We want to retrieve the zone (cluster, data block, whatever) number 3, each block is 4 K, and the super-block etc. needs 64 K. Linux uses a `BLOCK_SIZE` of

1 K, so the data blocks start at disk block 64+1 (the bootsector+waste). Every zone consists of 4 disk blocks, so our 3rd zone starts at $65 + 3 * 4 = 77$.

8.2.3.3 myfs_readdir

signature: `static int myfs_readdir(struct file filp, void dirent, filldir_t filldir)`
Directory inodes use the data blocks they are referring to as space for a list of directory entries.
Every directory entry consists of the filename and an inode-pointer. There is at least the "." and ".." entry pointing to itself and the upper directory inode. It is easy to implement hard links just by having different filenames pointing to the same inode numbers.

8.2.3.3.1 Hard vs. Soft Links Hard links are only allowed to regular file-inodes. This is very clear because if you would point to an directory inode, the ".." entry has to point back to your actual inode. If you hardlink to the same directory inode from another directory, the ".." entry would have to dynamically change.
Soft links are just inodes set to the `S_IFLNK` mode, which have stored a full path name in their first data block. This enables jumping to a directory from different directory entries, too.

8.2.3.4 myfs_lookup

signature: `struct dentry myfs_lookup(struct inode  dir, struct dentry dentry)`
When a directory entry (= file name) is used, the whole directory is searched for this file name.
If it is found, a `struct inode*` is filled and put into the kernels hashtables with `d_add(dentry,inode)`. If not, `struct inode*` remains NULL and `d_add` is called with a NULL value.

8.2.3.5 myfs_create

signature: `int myfs_create(struct inode  dir, struct dentry dentry, int mode)`
So if the above `myfs_lookup` failed and we used something like `touch,cp,cat`, the file has to be created. For this we have to find an unused inode, set all data (uid, gid, access times, super block...) and mark the inode dirty to force an update on disk. Then we have to add the new directory entry with its referring inode to the given directory inode's list of directory entries.

8.2.3.6 myfs_truncate

signature: `void myfs_truncate(struct inode  inode)`
If `myfs_lookup` didn't fail but we want to overwrite a file with eg. `cp,cat,echo`, the file has to be truncated first.
For this we have to run through all used blocks by this file to check, if they are in the kernels internal buffers.
We do this by using `get_hash_table(inode->i_dev,myfs_bmap(inode,i),BLOCK_SIZE)` where `i` is a counter variable for every block. This returns a `struct buffer_head*` if it is in the buffer, or NULL else. For every buffer not used by somebody else is still using (`bh->b_count > 1`) we have to loop until we are the only one using it. Then we can mark this buffer clean and release it. After all buffers are released, we set the inode size to zero and mark the inode dirty.

8.2.3.7 myfs_file_write

signature: `static ssize_t myfs_file_write(struct file filp, const char buf, size_t count, loff_t ppos)`

The `struct file *filp` contains a `struct dentry *f_dentry`, which contains the `struct inode d_inode`. This is our inode in our file system, where we want to write the buffer `buf` with `count` size at position `ppos`.
If `filp->f_flags & O_APPEND` , the buffer is appended. Now we have to get the first block into memory, write the first part of the data and so forth.
If we write over the old file size, we have to find unused blocks and use them for this inode. If we don't have any left, we return with `-ENOSPC`.
Since the buffer may start at a position not equal a block offset equal zero, and may end in the middle of a block, special care has to be taken in the algorithm.

8.2.3.8 myfs_file_read

signature: `static ssize_t myfs_file_read(struct file  filp, char buf, size_t count, loff_t ppos)`

Works quite the same way as `myfs_file_write`, but the only change to the file is the inode's access time.
The Linux kernel offers may generic functions like `generic_file_read` which uses `myfs_bread, myfs_bmap` etc. , but this may not be sufficient for special file systems like encrypted ones.

8.3 Caesar File System

We will implement our own encrypted file system based on Caesars well-known algorithm.
The passphrase will be given as a mount parameter. Only caveat is, that everybody will see it when he is looking at all mounted file systems with `mount` :(
But it's just for teaching purposes, so improvements are welcome!
To make testing easier I recommend using a loopback file and not a partition for writing the filesystem on it. If possible, use VMware © to save precious time for rebooting...
The loopback device has to be compiled into the kernel. This only works on local files! You can't use NFS-mounted partitions!

```
dd if=/dev/zero of=/tmp/mycaesarfs.part count=10240
mkfs.caesar mycaesarfs.part
modprobe -a caesar
mount -tcaesar /tmp/mycaesarfs.part /mnt -o loop,mypass
```

8.3.0.9 Encrypting and Decrypting

Good old Caesar used a letter of the alphabet (one out of 26) to shift every letter of a secret message by this one. Since we have 8-bit characters containing not only letters and digits, we can choose from 256 possibilities. To make the whole thing more interesting, we use a longer passphrase (length `N`, where a letter of the secret message at position `n` is cyphered with the passphrase letter `n mod N`. Passphrase `"ABCDE"` (65-66-67-68-69) used on `"Secret"` (83-101-99-114-101-116) goes to (148-167-166-182-170-181).
If the passphrase letter + secret letter extends 0xFF, the byte is wrapped around and starts with 0 again, so 161 + 110 is 15.
To revert to the original message, just substract the passphrase.

8.3.0.10 Simplified File System Implementation

To keep everything easy, we just store the `CAESAR_MAX_INODES` inodes in the superblock. To keep in line with the multiple naming conventions of data blocks, clusters and zones we will call ours `chunk`s.
We use a fixed number of chunks and divide the rest of the partition size by `CAESAR_MAX_CHUNKS` to get the maximum (and minimum) file size for each and every file.
So a regular inode points to a chunk with an exact size of (partition size / CAESAR_MAX_CHUNKS).

Of course this is a complete waste of space, since the average file size will never be exactly this calculated value, but this makes a lot of things easier to handle!

To keep track of unused files, we need a bitfield in the super-block. Unused inodes are recognized by a `i_nlink` equal to zero.

There is only one directory, the root directory with all entries in it. Directory entries may have a maximum name length of `CAESAR_FILENAME_LEN-1`.

⋮

```
|#define CAESAR_SUPER_MAGIC      0xEFFA              /* magic header */
|#define CAESAR_VALID_FS         0x0001             /* Clean fs. */
|#define CAESAR_ERROR_FS         0x0002             /* fs has errors. */
|
|#define CAESAR_ROOT_INO 1
|#define CAESAR_MAX_CHUNKS 10
|#define CAESAR_MAX_INODES 35          /* keep small! must fit into superblock */
|#define CAESAR_FILENAME_LEN 18        /* don't forget the null-byte */
|
|/*
| * ATTENTION: every structure is word aligned!!!
| */
|
|struct caesar_disk_inode {   /* 28 bytes */
|         __u16 i_mode;
|         __u16 i_nlink;
|         __u16 i_uid;
|         __u16 i_gid;
|         __u32 i_chunk;
|         __u32 i_size;
|         __u32 i_atime;
|         __u32 i_mtime;
|         __u32 i_ctime;
|};
|
|struct caesar_disk_dir_entry {
|         __u16 inode;                    /* points into the inode array */
|         char name[CAESAR_FILENAME_LEN];
|};
|
|/*
| * superblock also contains inode list
| */
|struct caesar_disk_super_block {   /* 16 + 35*28 = 996 bytes */
|         __u16 s_magic;
|         __u16 s_state;
|         __u32 s_partition_size;  /* how big is the caesar-fs partition */
|         __u32 s_max_chunk_size;   /* s_partition_size DIV MAX_CHUNKS  */
|         __u32 s_fmap;             /* bitmap for used files */
|      /*
```

```
|       * Linux doesn't allow an inodeNr 0
|       * so we have to decrease by one whenever we access s_inodes[]
|       */
|         struct caesar_disk_inode s_inodes[CAESAR_MAX_INODES];
|};
⋮
```

8.4 Exercise I

Get in touch with the file system sources and try to understand the basics.
Just copy the patch file into the root directory of your kernel sources and type
`gunzip caesar-modbasic-2.2.9.patch.gz`
`patch -p1 < caesar-modbasic-2.2.9.patch.gz`
Then configure and make the kernel. Add the caesar-fs and minix-fs as modules, also don't forget the loopback device! Now you are able to use our provided modules `caesar-log.o` and `minix-log.o` instead of the compiled ones, which are only stubs. Just type `insmod caesar-log.o` and `insmod minix-log.o` to get them loaded.
Those modules will help you to understand the calling hierarchy of the different functions, because every call is logged to `/var/log/messages` and is also visible on your console with `STRG-ALT-F10`.
Follow the minix source code and logs and create a tree of all called functions for the following actions:

- `mount` a minix-fs file/disk on `/mnt`
- `ls /mnt`
- `echo "hello" > /mnt/text`
- `cat /mnt/text`
- `echo " world" >> /mnt/text`
- `echo "bye" > /mnt/text`
- `umount /mnt`

You should be able to understand the reason for the diffent functions to be called! Now try to mount the supported caesar-file (*loopfile.cfs*) and copy the files stored in it into your working directory. The loopfile's secret passphrase is *"ossecret"* . Here you will find the sources of the caesar-fs in a patch file again. Don't forget to read the README file!

Since you will need the sources for the next exercise, there are some functions missing. So compiling this will not result in the same `caesar-log.o`, but with less functionality.
Do the same trace and hierarchy tree with the caesar-fs based on the module `caesar-log.o` and find the analog parts. For here, it may be helpful to snoop into the next exercise's source.

8.4.1 Solution

This shows the log entries of the caesar file system for the different actions.

- `mount` a minix-fs file/disk on `/mnt`

```
kernel: caesar_read_super: opts [mypass]
kernel: caesar_read_inode 1
kernel: CAESAR-fs: mounting unchecked file system
kernel: caesar_write_super
```

- `ls /mnt`

```
kernel: caesar_readdir ino 1 size 40
kernel: caesar_bread dev 07:00 ino 1 with block 0
kernel: caesar_bmap for ino 1 with block 0
kernel: caesar_readdir ino 1 size 40
kernel: caesar_bread dev 07:00 ino 1 with block 0
kernel: caesar_bmap for ino 1 with block 0
kernel: caesar_write_inode 1
```

- `echo "hello" > /mnt/text`

```
kernel: caesar_lookup ino 1
kernel: caesar_bread dev 07:00 ino 1 with block 0
kernel: caesar_bmap for ino 1 with block 0
kernel: caesar_lookup: dentry [text] not found!
kernel: caesar_create [text] in ino 1
kernel:    found free ino 2 + chunk 1
kernel: caesar_read_inode 2
kernel: caesar_read_inode: reading unused inode 2
kernel: caesar_add_entry [2|text] to ino 1
kernel: caesar_bread dev 07:00 ino 1 with block 0
kernel: caesar_bmap for ino 1 with block 0
```

```
kernel: caesar_file_write ino 2 (6 bytes)
kernel: caesar_bread dev 07:00 ino 2 with block 0
kernel: caesar_bmap for ino 2 with block 0
kernel: caesar_write_inode 1
kernel: caesar_write_inode 2
```

- `cat /mnt/text`

```
kernel: caesar_readdir ino 1 size 60
kernel: caesar_bread dev 07:00 ino 1 with block 0
kernel: caesar_bmap for ino 1 with block 0
kernel: caesar_readdir ino 1 size 60
kernel: caesar_file_read ino 2 at pos 0
kernel: caesar_bread dev 07:00 ino 2 with block 0
kernel: caesar_bmap for ino 2 with block 0
kernel: caesar_file_read ino 2 at pos 6
kernel: caesar_write_inode 1
kernel: caesar_write_inode 2
```

- `echo " world" >> /mnt/text`

```
kernel: caesar_file_write ino 2 (7 bytes)
kernel: caesar_bread dev 07:00 ino 2 with block 0
kernel: caesar_bmap for ino 2 with block 0
kernel: caesar_write_inode 2
```

- `echo "bye" > /mnt/text`

```
kernel: caesar_truncate ino 2 with 1 blocks
kernel: caesar_bmap for ino 2 with block 0
kernel: caesar_file_write ino 2 (4 bytes)
kernel: caesar_bread dev 07:00 ino 2 with block 0
kernel: caesar_bmap for ino 2 with block 0
kernel: caesar_write_inode 2
```

- `umount /mnt`

```
kernel: caesar_put_super
```

8.5 Exercise II

Here you have to implement the empty stubs of the functions

- caesar_bmap
- caesar_bread
- caesar_file_read
- caesar_readdir

To do this correctly, use your knowledge gained on the minix file system and the existing source code of the caesar-fs.

8.5.1 Solution

The full caesar file system is found in the patch file `caesar-full-2.2.9.patch.gz`

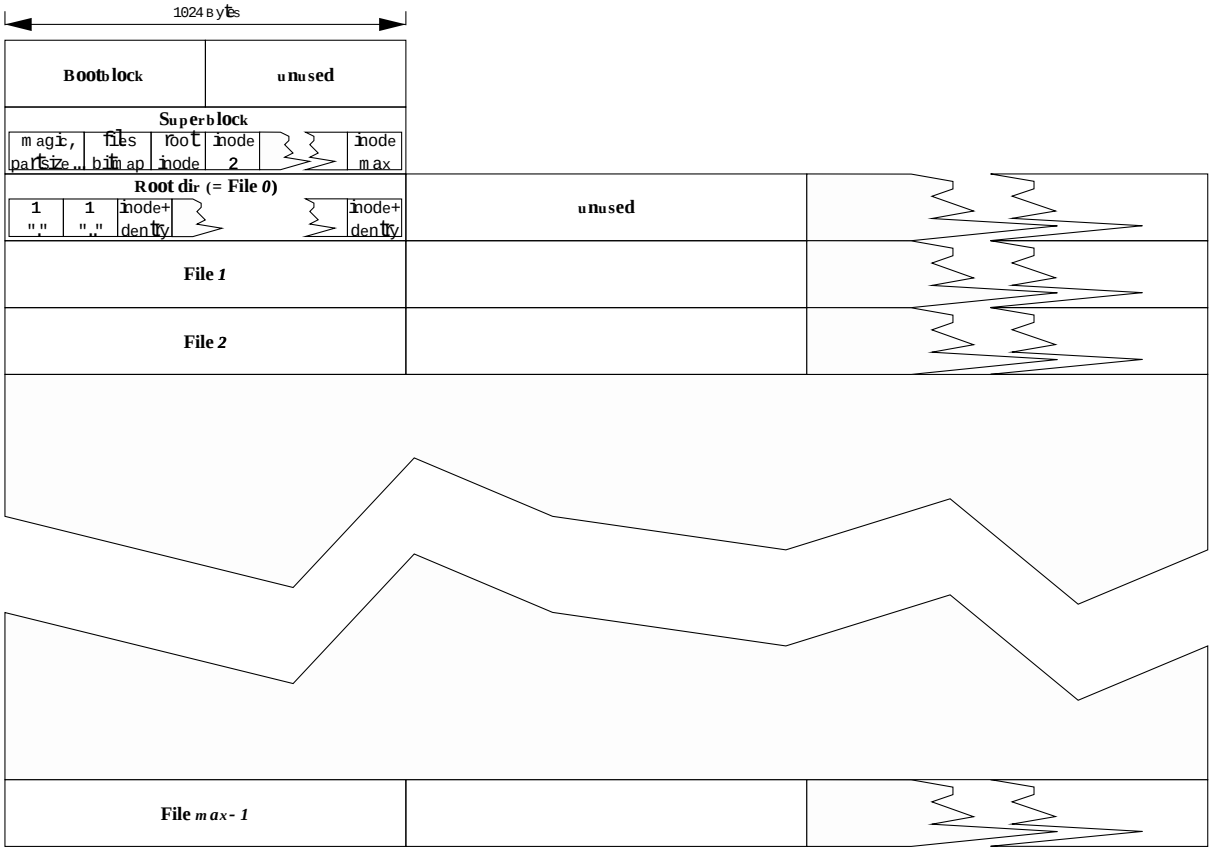


Figure 8.1: Caesar Partiton Layout

Chapter 9

The Linux /proc File System

9.1 General

The proc file system is a purely virtual file system mounted in `/proc` with lots of information about the running system.
For example, you find the entry `/proc/interrupts`, which contains a list of all used interrupts, or `/proc/meminfo` with statistics about used and free memory. All those entries are updated whenever the virtual inode is read. Although it is not a "real" file system, it completely complies with the virtual file system's standards. For example, it uses the magic file system identifier `#define PROC_SUPER_MAGIC 0x9fa0`.
The directory hierarchy is defined in `linux/include/linux/proc_fs.h` using `enum`s.
There are all inodes listed, starting at `enum root_directory_inos`.
`Extract of linux/include/linux/proc_fs.h`:

⋮

```
15|enum root_directory_inos {
16|        PROC_ROOT_INO = 1,
17|        PROC_LOADAVG,
18|        PROC_UPTIME,
19|        PROC_MEMINFO,
20|        PROC_KMSG,
21|        PROC_VERSION,
22|        PROC_CPUINFO,
23|        PROC_PCI,
24|        PROC_MCA,
25|        PROC_NUBUS,
26|        PROC_SELF,       /* will change inode # */
27|        PROC_NET,
28|        PROC_SCSI,
29|        PROC_MALLOC,
```

```
30|        PROC_KCORE,
31|        PROC_MODULES,
32|        PROC_STAT,
33|        PROC_DEVICES,
34|        PROC_PARTITIONS,
35|        PROC_INTERRUPTS,
36|        PROC_FILESYSTEMS,
37|        PROC_KSYMS,
38|        PROC_DMA,
39|        PROC_IOPORTS,
40|        PROC_PROFILE, /* whether enabled or not */
41|        PROC_CMDLINE,
42|        PROC_SYS,
43|        PROC_MTAB,
44|        PROC_SWAP,
45|        PROC_MD,
46|        PROC_RTC,
47|        PROC_LOCKS,
48|        PROC_HARDWARE,
49|        PROC_SLABINFO,
50|        PROC_PARPORT,
51|        PROC_PPC_HTAB,
52|        PROC_STRAM,
53|        PROC_SOUND,
54|        PROC_MTRR, /* whether enabled or not */
55|        PROC_FS
56|};
⋮
```

9.2 Example: /proc/meminfo

Let's have a deeper look at `/proc/meminfo`, which is in the root directory of the proc file system. It works very similar to `/proc/cpuinfo` or `/proc/uptime` and many others, so it is just one out of many. :)
Every entry has to fill a `struct proc_dir_entry` with its specific data like the file name and the used inode operations.
`Extract of linux/include/linux/proc_fs.h:`

```
⋮
257|/*
258| * This is not completely implemented yet. The idea is to
259| * create an in-memory tree (like the actual /proc filesystem
260| * tree) of these proc_dir_entries, so that we can dynamically
261| * add new files to /proc.
262| *
263| * The "next" pointer creates a linked list of one /proc directory,
```

```
264| * while parent/subdir create the directory structure (every
265| * /proc file has a parent, but "subdir" is NULL for all
266| * non-directory entries).
267| *
268| * "get_info" is called at "read", while "fill_inode" is used to
269| * fill in file type/protection/owner information specific to the
270| * particular /proc file.
271| */
272|struct proc_dir_entry {
273|        unsigned short low_ino;
274|        unsigned short namelen;
275|        const char *name;
276|        mode_t mode;
277|        nlink_t nlink;
278|        uid_t uid;
279|        gid_t gid;
280|        unsigned long size;
281|        struct inode_operations * ops;
282|        int (*get_info)(char *, char **, off_t, int, int);
283|        void (*fill_inode)(struct inode *, int);
284|        struct proc_dir_entry *next, *parent, *subdir;
285|        void *data;
286|        int (*read_proc)(char *page, char **start, off_t off,
287|                          int count, int *eof, void *data);
288|        int (*write_proc)(struct file *file, const char *buffer,
289|                           unsigned long count, void *data);
290|        int (*readlink_proc)(struct proc_dir_entry *de, char *page);
291|        unsigned int count;      /* use count */
292|        int deleted;             /* delete flag */
293|};
```

⋮

Extract of `linux/fs/proc/root.c`:

⋮

```
518|static struct proc_dir_entry proc_root_meminfo = {
519|        PROC_MEMINFO, 7, "meminfo",
520|        S_IFREG | S_IRUGO, 1, 0, 0,
521|        0, &proc_array_inode_operations
522|};
```

⋮

⋮

```
677|        proc_register(&proc_root, &proc_root_meminfo);
```

⋮

`PROC_MEMINFO` is the inode number from the `enum root_directory_inos`, with the file name `meminfo` with a length of 7.

`S_IRUGO` is a #define for access rights of 0444, so Read only for User, Group and

Others.
Finally, this entry has to be registered into the root directory tree.
Extract of linux/fs/proc/array.c:

```
⋮
1491|static struct file_operations proc_array_operations = {
1492|        NULL,            /* array_lseek */
1493|        array_read,
1494|        NULL,            /* array_write */
1495|        NULL,            /* array_readdir */
1496|        NULL,            /* array_poll */
1497|        NULL,            /* array_ioctl */
1498|        NULL,            /* mmap */
1499|        NULL,            /* no special open code */
1500|        NULL,            /* flush */
1501|        NULL,            /* no special release code */
1502|        NULL             /* can't fsync */
1503|};
1504|
1505|struct inode_operations proc_array_inode_operations = {
1506|        &proc_array_operations, /* default base directory file-ops */
1507|        NULL,                   /* create */
1508|        NULL,                   /* lookup */
1509|        NULL,                   /* link */
1510|        NULL,                   /* unlink */
1511|        NULL,                   /* symlink */
1512|        NULL,                   /* mkdir */
1513|        NULL,                   /* rmdir */
1514|        NULL,                   /* mknod */
1515|        NULL,                   /* rename */
1516|        NULL,                   /* readlink */
1517|        NULL,                   /* follow_link */
1518|        NULL,                   /* readpage */
1519|        NULL,                   /* writepage */
1520|        NULL,                   /* bmap */
1521|        NULL,                   /* truncate */
1522|        NULL                   /* permission */
1523|};
⋮
```

`array_read` just mallocs a page of free memory and then calls `fill_array`, which calls `get_root_array` to call the specific function of eg. `meminfo` via a big case block over all possible inodes (eg. `PROC_MEMINFO`).
`get_meminfo` is straight forward and fills the page with the actual data.
Back to `array_read`, the page is copied to user space and these are the contents of this "file" then.
Extract of linux/fs/proc/array.c:

```
⋮
```

```
340|static int get_meminfo(char * buffer)
341|{
342|        struct sysinfo i;
343|        int len;
344|
345|        si_meminfo(&i);
346|        si_swapinfo(&i);
347|        len = sprintf(buffer, "        total:    used:    free:  shared:
buffers:  cached:\n"
348|                "Mem:  %8lu %8lu %8lu %8lu %8lu %8lu\n"
349|                "Swap: %8lu %8lu %8lu\n",
350|                i.totalram, i.totalram-i.freeram, i.freeram, i.sharedram,
i.bufferram, page_cache_size*PAGE_SIZE,
351|                i.totalswap, i.totalswap-i.freeswap, i.freeswap);
352|     /*
353|      * Tagged format, for easy grepping and expansion. The above will go away
354|      * eventually, once the tools have been updated.
355|      */
356|        return len + sprintf(buffer+len,
357|                "MemTotal:  %8lu kB\n"
358|                "MemFree:   %8lu kB\n"
359|                "MemShared: %8lu kB\n"
360|                "Buffers:   %8lu kB\n"
361|                "Cached:    %8lu kB\n"
362|                "SwapTotal: %8lu kB\n"
363|                "SwapFree:  %8lu kB\n",
364|                i.totalram >> 10,
365|                i.freeram >> 10,
366|                i.sharedram >> 10,
367|                i.bufferram >> 10,
368|                page_cache_size << (PAGE_SHIFT - 10),
369|                i.totalswap >> 10,
370|                i.freeswap >> 10);
371|}
```

9.3 Homework: Usage Counter of File System Types

When you look at `/proc/filesystems` you see all the registered filesystems. Additionally to that, we want to know how often each file system type was used and how many devices are currently mounted using a file system type.
For example, if we mount and umount a disk with a minix-fs for two times and mount if for a third time, we should see a line like:

```
minix    1   3
```

9.3.1 Hints

Follow the way through the /proc file system sources until you find the function, which generates the list of all registered file systems. Do not forget to add the increasing/decreasing of the counters for actual and overall mount times as late as possible to make sure that you count only really mounted ones.

9.3.2 Solution

After adding the counter variables to the `struct file_system_type`, we use every `struct super_block`'s entry `struct file_system_type *s_type` to access them.

`Extract of linux/include/linux/fs.h:`

```
⋮
644|struct file_system_type {
645|        const char *name;
646|        int fs_flags;
647|        struct super_block *(*read_super) (struct super_block *, void *,
int);
648|        long all_nr_mounts;
649|        long act_nr_mounts;
650|        struct file_system_type * next;
651|};
⋮
```

Just increase both counters in `linux/fs/super.c` at `add_vfsmnt()`.
We have to decrease the `act_nr_mounts` in `do_umount()` because the super-block is freed there.
The generated list with our new counters is created in `get_filesystem_list()`.

`Extract of linux/fs/super.c:`

```
⋮
 91|static struct vfsmount *add_vfsmnt(struct super_block *sb,
 92|                        const char *dev_name, const char *dir_name)
 93|{
⋮
⋮
134|        if (!sb->s_type) /* first mount of root_fs is w/o s_type! */
135|                printk("add_vfsmnt: fs_type of %s is NULL!\n",dev_name);
136|        else {
137|          sb->s_type->all_nr_mounts++;
138|          sb->s_type->act_nr_mounts++;
139|        }
140|out:
```

```
141|        return lptr;
142|}
⋮
⋮
176|int register_filesystem(struct file_system_type * fs)
177|{
⋮
⋮
190|        fs->all_nr_mounts=0;
191|        fs->act_nr_mounts=0;
192|        *tmp = fs;
193|        return 0;
194|}
⋮
⋮
389|int get_filesystem_list(char * buf)
390|{
391|        int len = 0;
392|        struct file_system_type * tmp;
393|
394|        tmp = file_systems;
395|        while (tmp && len < PAGE_SIZE - 80) {
396|                len += sprintf(buf+len, "%s\t%s\t%ld\t%ld\n",
397|                        (tmp->fs_flags & FS_REQUIRES_DEV) ? "" : "nodev",
398|                        tmp->name, tmp->act_nr_mounts,
tmp->all_nr_mounts);
399|                tmp = tmp->next;
400|        }
401|        return len;
402|}
⋮
⋮
648|static int do_umount(kdev_t dev, int unmount_root, int flags)
649|{
650|        struct super_block * sb;
651|        int retval;
652|
653|        retval = -ENOENT;
654|        sb = get_super(dev);
655|        if (!sb || !sb->s_root)
656|                goto out;
⋮
⋮
721|        if (!sb->s_type)  /* first mount of root_fs is w/o s_type! */
```

```
722|          printk("remove_vfsmnt: fs_type of %s is NULL!\n",kdevname(dev));
723|        else {
724|          sb->s_type->act_nr_mounts--;
725|        }
726|
727|        sb->s_dev = 0;           /* Free the superblock */
728|        unlock_super(sb);
729|
730|        remove_vfsmnt(dev);
731|out:
732|        return retval;
733|}
```

References

- The Linux Users' Guide
- The Linux System Administrators' Guide
- Filesystem Hierarchy Standard
- Slackware Linux Unleashed, ©1997 by Sams Publishing
- Diverse man-pages