

Ubuntu 16.04 programming environment

Making first steps

Edited by
David Sarel
davidsa@openu.ac.il

March 2017

DISCLAIMER: UNDER NO CIRCUMSTANCES MAY OPEN UNIVERSITY AND/OR UBUNTU PROJECT AND/OR MANUAL'S WRITER BE HELD LIABLE FOR DAMAGE TO HARDWARE OR SOFTWARE, LOST DATA OR OTHER DAMAGE EITHER DIRECT OR INDIRECT CAUSED BY USING UBUNTU AND/OR THIS MANUAL. BACKUP VALUABLE DATA BEFORE LAUNCHING UBUNTU.

Table of contents

Preface	3
Part 1 – Launching virtual machine	4
Introduction	5
Hardware requirements	6
Booting Ununtu virtual machine	7
The Unity user interface	9
Configuring VMware Player	11
Where to save files	12
Shutting the system down	16
Connecting the internet	17
Part 2 – Programming environment	18
How to get some help?	19
Command line interpreters - shells	22
The structure of Linux command	23
The file system	25
Wildcards	26
Several file-related commands	27
Several directory-related commands	28
Redirecting Input/Output	28
Protecting files and directories, altering protections	29
chmod command	31
PostScript and PDF file viewer	32
Editors	33
The gcc compiler	38
Libraries	40
The make utility	43
Debugging your program	49
Part 3 – Advanced topics	59
33 useful commands	60
awk	77
Index	82

Preface

This short manual was issued to introduce students studying at the Open University of Israel to the Linux operating system.

The vast majority of our students see Linux system in their very first time. Therefore, the main goal of this introduction is to spare the venture of Linux installation and system configuration.

VMware Player hypervisor is a program that helps to achieve this target by running Linux as a guest operating system hosted by some version of Windows operating system, eventually providing that Linux runs in a window on the host operating system. Running Linux in VM Player as a guest operating system allows borrowing certain configurations and hardware recourses from the host operating system including access to CD-ROM, audio devices and internet connection adopted from the host operating system. Linux virtual machine comes equipped with 20 GB virtual hard disk that is not shared with the host operating system but several ways exist to share files between the guest and host operating systems.

This manual was initially compiled for the “Operating Systems” course at the times it was the first class introducing Linux in the Open University. Nowadays, this brochure together with a supplied Linux virtual machine is primarily used by the preliminary “System Programming Laboratory” course for the first experience of a subject while continued utilized by the OS course as repetition priming and reference.

I would like to conclude the preface with expression of especial gratitude to Ehud Lamm for thorough review, useful advises and prodding for better prose and to Dr. Jacky Hertz for reviewing the entire brochure and for tireless efforts of interweaving Linux into the CS courses. Also many thanks to the authors and people from the projects that their materials were used compiling this brochure: Karl Schock and his Knoppix tutorial that was of great help (www.elele.de/knoppix/docs/tutorial); Alex Kremer and his section about static libraries that was adopted and revised for this manual; the section from the homepage of the DDD debugger that was used here to demonstrate debugging techniques; the section "33 useful commands" that was taken from www.thegeekstuff.com/2010/11/50-linux-commands and revised; the section awk that was taken from www.thegeekstuff.com/2010/01/awk-introduction-tutorial-7-awk-print-examples and revised.

Part One

Launching Virtual Machine

Introduction

Operating systems course supplies the Linux virtual machine based on the Ubuntu operating system. Ubuntu is an open source project comprising a collection of GNU/Linux software. The system suits our main goal because of automatic hardware detection and because it comes with vast amount of preinstalled applications. Following is a partial list of what you got:

- ✓ Linux Kernel 4.4.0-21-generic
- ✓ Unity user interface
- ✓ LibreOffice (a derivative of OpenOffice.org)
- ✓ Eclipse (C, C++, JAVA)
- ✓ Emacs
- ✓ DDD
- ✓ Lots of games
- ✓ Utilities for data recovery
- ✓ Internet browsers

Ubuntu is based on Debian Linux. Source files for any of the installed packages may be downloaded from the Ubuntu Linux mirrors.¹

¹ `sudo apt-get build-dep $package`
where package=whatever package you want.

then do
`sudo apt-get source $package`
to get the source for that package.

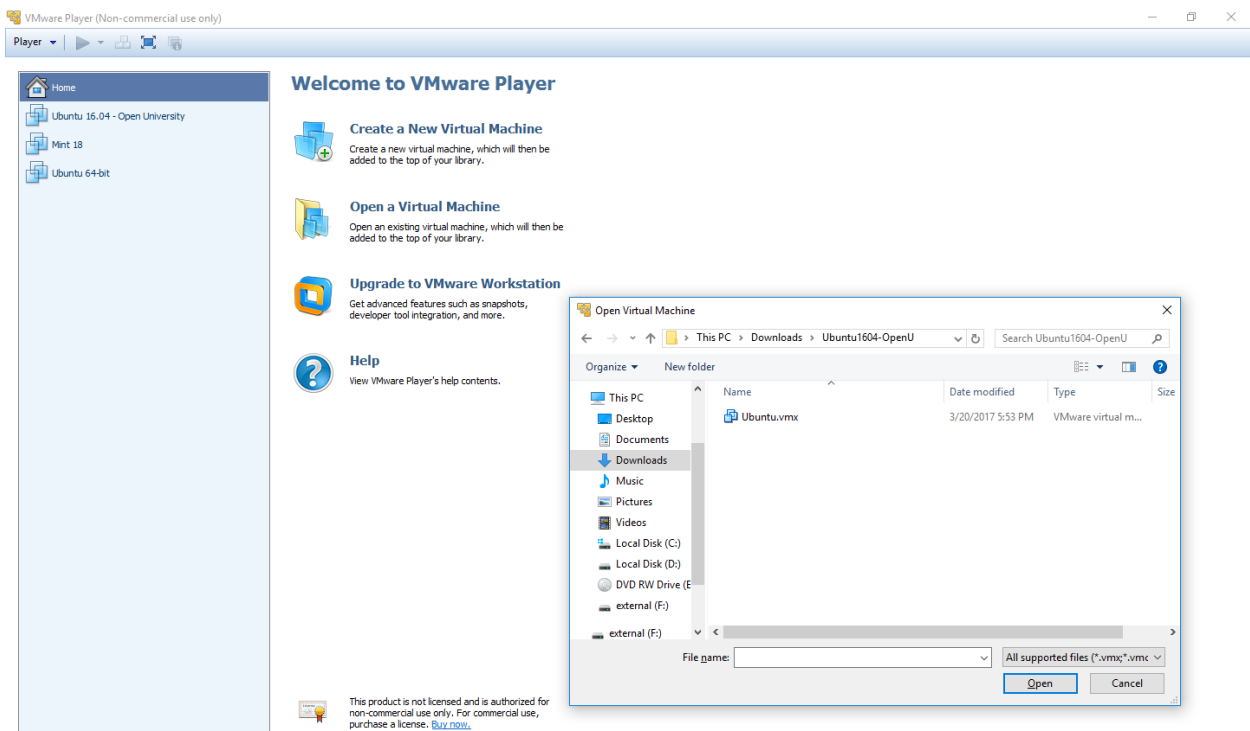
Hardware requirements

To run the Ubuntu virtual machine, you need:

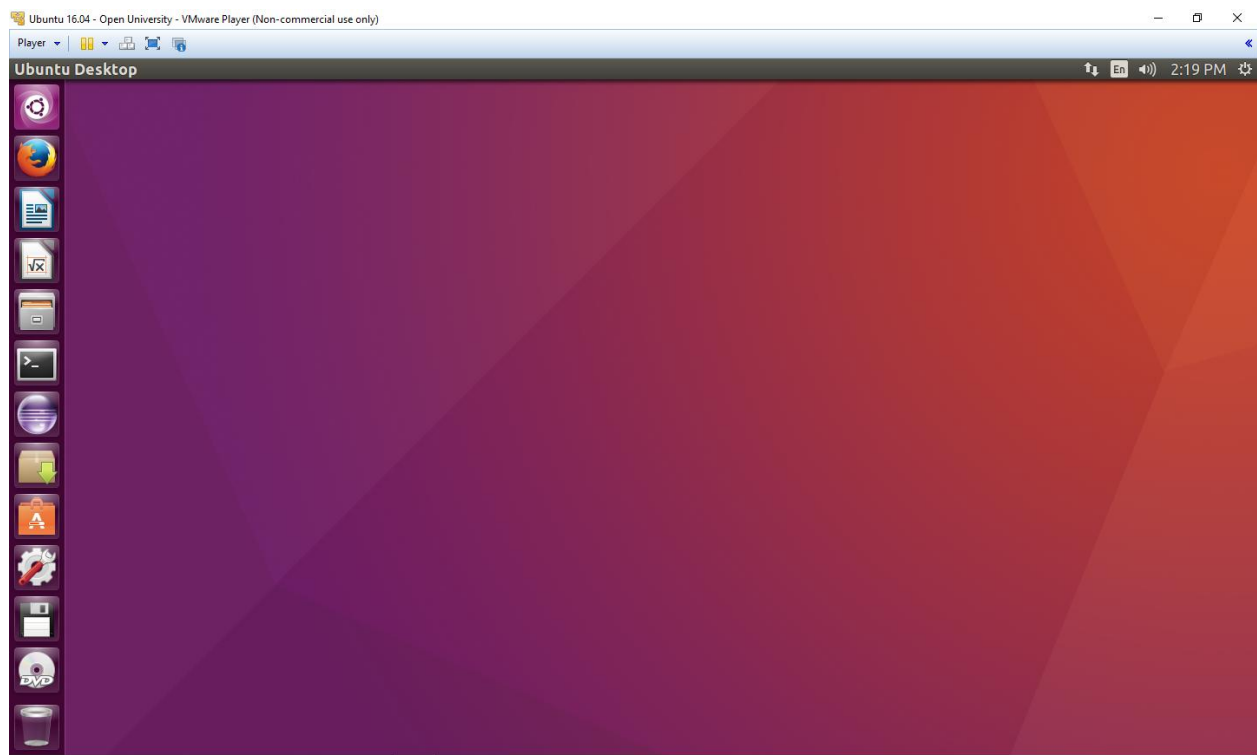
- ✓ Intel-compatible CPU
- ✓ 2 GB of RAM (4 GB is preferred)
- ✓ Standard SVGA-compatible graphic cards
- ✓ Mouse: serial or PS2 or USB
- ✓ Space on the HD (min 7 GB – max 23 Gb)

Booting Ubuntu virtual machine

In order to boot Ubuntu virtual machine it should be first saved to your hard disk. Download the virtual machine files to from the link supplied at the course site. Make sure you have at least 7 GB free space. Turn those files from being 'read only' to be 'read/write'. Download the VMware Player (either from [here](#)² or from the course site), install and run it. Then specify the location of your Ubuntu virtual machine and click "Play virtual machine":



A short while after the following Unity graphical user interface appears:



In order to continue working with Ubuntu your virtual machine needs to **get a focus**. This could be done either by:

- Maximizing the VMware Player window and running Ubuntu in the full screen mode

or by

- Clicking the mouse inside the VMplayer.

NOTE: Hitting “ALT+CTRL” combination returns back to the Windows system.

RECOMMENDATION: Full screen mode gives a feeling of working with a real machine installation.

While working in the full mode you may either pin in the VMWare pull up menu:



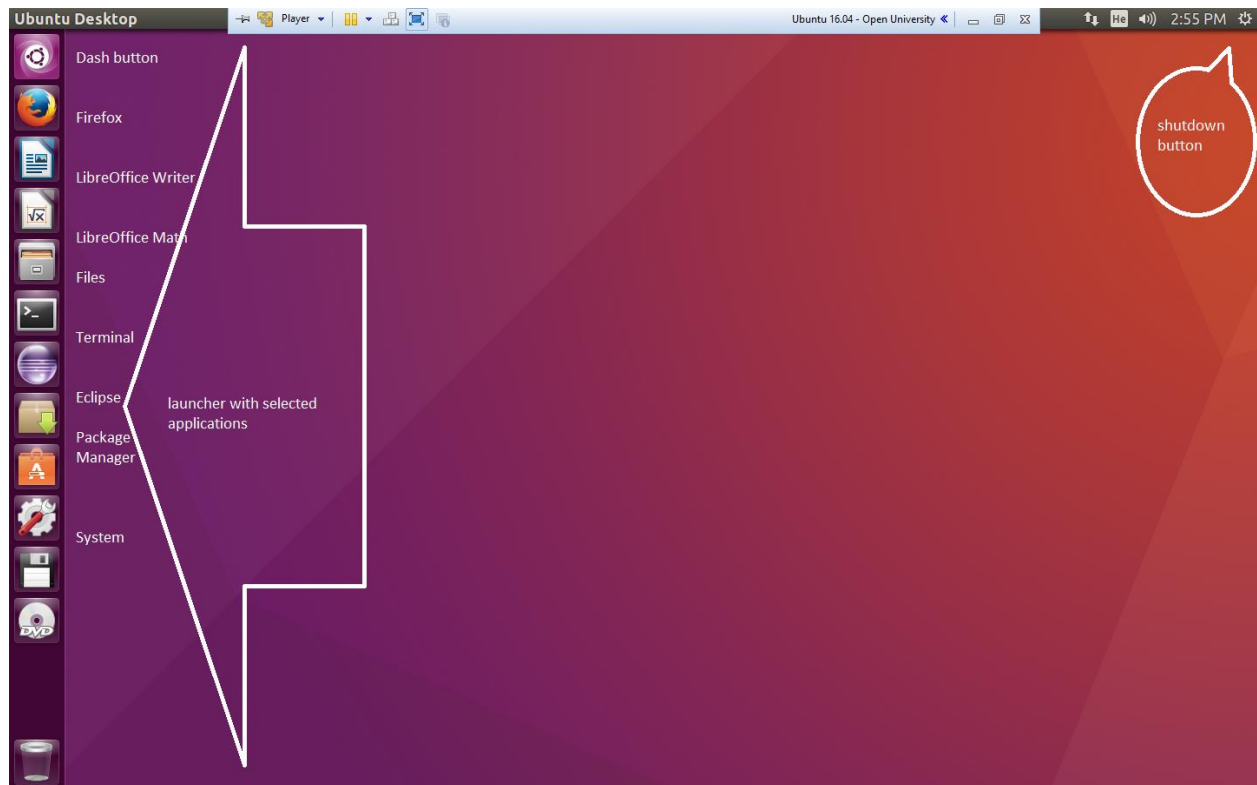
or to allow him pulling back and forth:



The Unity user interface

Unity user interface is an appealing and easy to configure graphical environment. Unity was designed to minimize distractions, give you more room to work, and help you get things done.

Among the first items you'll see are:



The Dash button -- The Dash is designed to make it easier to find, open and use applications, files, and music. For example, if you type the word "document" into the Search Bar, the Dash will show you applications that help you write and edit documents. It will also show you relevant folders and documents that you have been working on recently.

The Dash also gives you shortcuts to often-used Internet, photo, email, and music applications.

The Shutdown button -- When you've finished using your computer, you can turn it off, suspend it (to save power), or leave it powered on and log out.

The Launcher -- one of the key components of the new Unity desktop. The launcher provides you with quick access to applications, workspaces, removable devices and the trash. We have

pinned in Firefox, LibreOffice, Eclipse and Terminal to the launcher since they will be the most frequently used applications during your studies in Open University.

Firefox -- is a free and open source web browser.

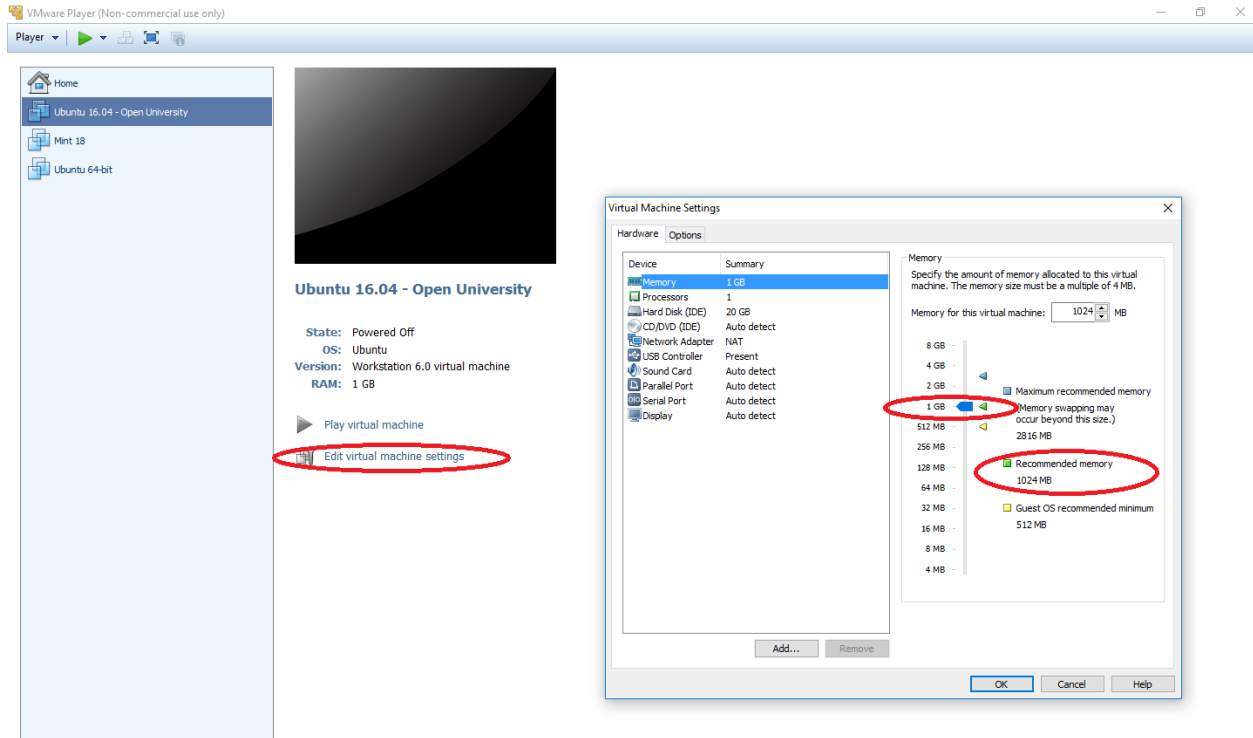
LibreOffice -- is a free and open source office suite. It is descended from OpenOffice.org. LibreOffice is designed to be compatible with other major office suites, including Microsoft Office, though some Microsoft Office layout features and formatting attributes are handled differently or unsupported.

Terminal -- Unix operating systems were originally designed as text only systems controlled by keyboard commands called **command line interpreter (CLI)**. The **X-window** system has been added since then to provide a graphical interface. However, the underlying CLI is still there and is frequently the easiest, fastest and most powerful way to perform many tasks. Terminal or Xterm (the X-Windows terminal emulation program) includes, among many other features, the ability to use multiple terminal shells in a single window, making for a less cluttered desktop.

Eclipse -- Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It can be used to develop applications in Java, C, C++ and other programming languages (by means of various plug-ins) including. Our development environments include the Eclipse Java development tools (JDT) for Java and Eclipse CDT for C/C++.

Configuring VMware Player

The VMware Player needs almost no configuration. But if you have enough RAM you can find it useful to enlarge the amount of RAM used by your virtual machine.

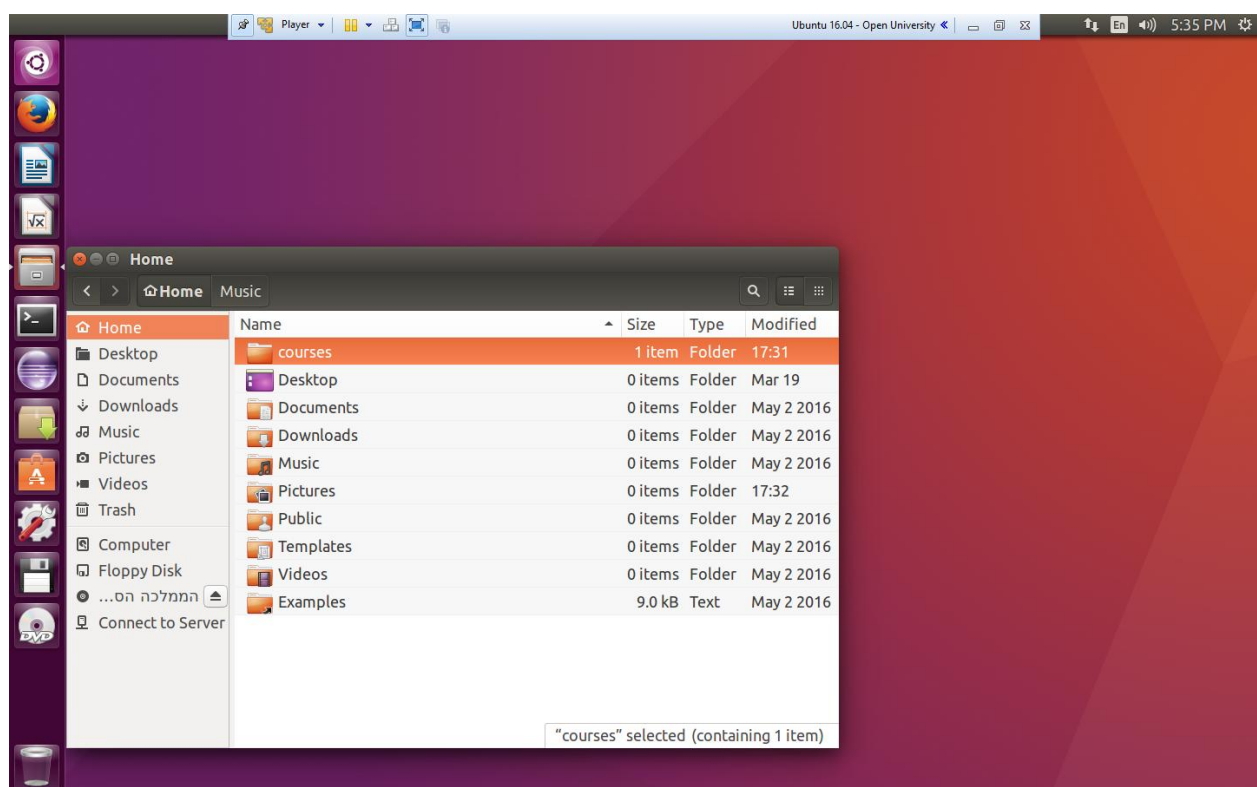


Stability of the graphical environment is tightly tied with the amount of RAM that is allocated to the Ubuntu virtual machine. The default choice should be using the RAM amount recommended by VM Player, but anyway not less than 1 GB. Pre-allocating RAM for virtual machine means that the host operating system (e.g. Windows) remains with less operational RAM by exactly the same amount one has allocated to the virtual machine. **Consult with “Hardware requirements” chapter to assure that your system has enough physical memory to run smoothly both host and guest operating systems.**

Where to save files

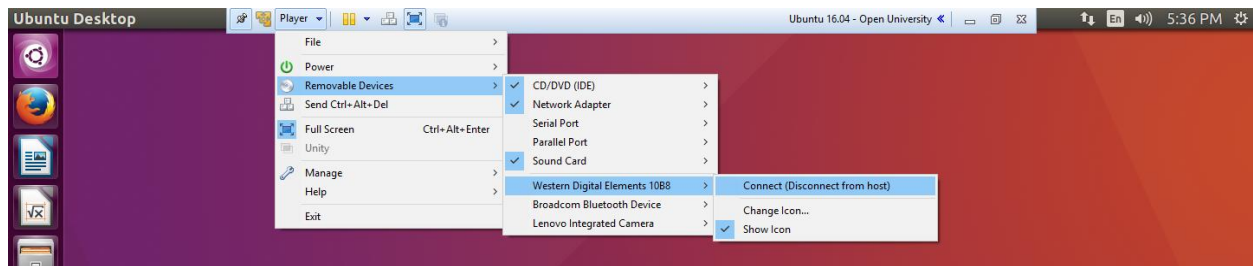
Virtual disk

The Ubuntu virtual machine supplied by the Operating systems course comes with virtual SCSI disk with maximal capacity of 20 GB. Through the Places shortcut you may open your personal folder (homedir) and save your files therein. For example, the folder courses/ already contains the folder named os/ where you can store Operating Systems course related material.



External USB devices

USB disks or DiskOnKey devices may be connected to the Ubuntu virtual machine. Ubuntu will detect them only if they were connected to the virtual machine:

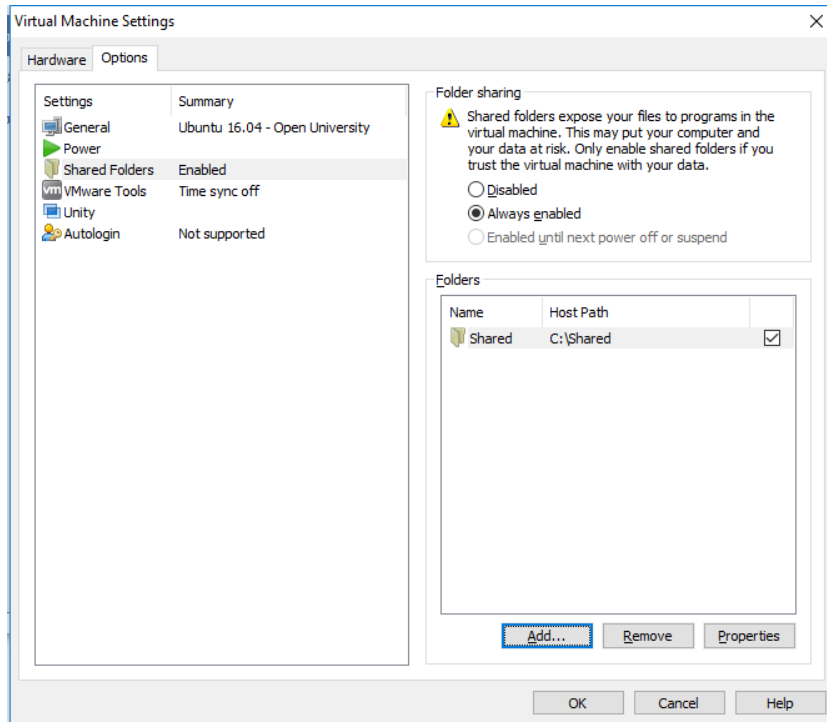


When Ubuntu detects a new USB device an appropriate icon appears on the Desktop. Since USB devices are interpreted via the SCSI interface, connected USB device will appear named `sdbX`, where X is a number ranged 1-16.

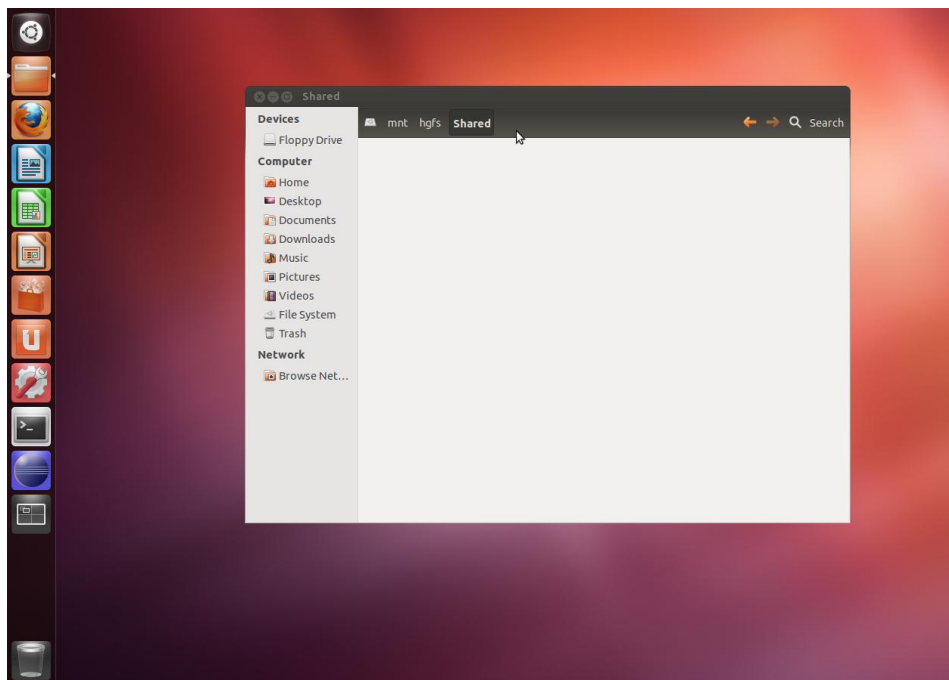
WARNING: Always perform “unmount” before you unplug the USB device (or at least run the filesystem synchronization command `sync` from the terminal that flushes filesystem buffers). Click the right mouse button on the disk icon and choose “Unmount”. Make sure to quit ALL the tasks that use the USB device to be unmounted, otherwise “unmounting” fails. To connect the USB device back to the host operating system, disconnect the device from the VMWare player (via the Devices menu).

Shared directory

You can establish sharing between the guest (Ubuntu) and host (Windows) operating systems. Create a folder that will be shared between the systems (e.g. C:\shared). Then, configure VMware Player to mount the shared folder by the next run:



Booting the virtual machine next time, you will find the shared folder available at `/mnt/hgfs/shared`



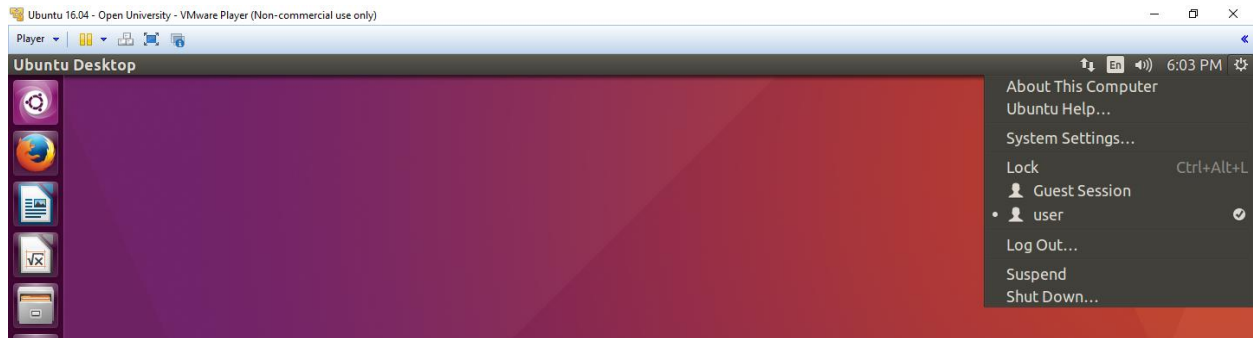
Floppy disk

If you have 3.5” floppy drive, you can save files on diskettes using `mtools`. `mtools` is a collection of DOS-like commands `mdir`, `mcopy`, `mdel` etc, designed especially to work with diskettes formatted FAT-12 from the command line. (FAT-12 is a format recognized by Windows operating systems). Following is a table of examples:

Goal	Command
mformat a:	Format the floppy that is in drive A:
mdir a:	List the files on the floppy in drive A:
mcopy file1 a:	Copy the file “file1” from the current directory to the floppy in drive A:
mcopy a:file1 .	Copy the file “file1” from the floppy in drive A: to the current working directory on the Linux system. “.” stands for the current working directory.

Shutting the system down

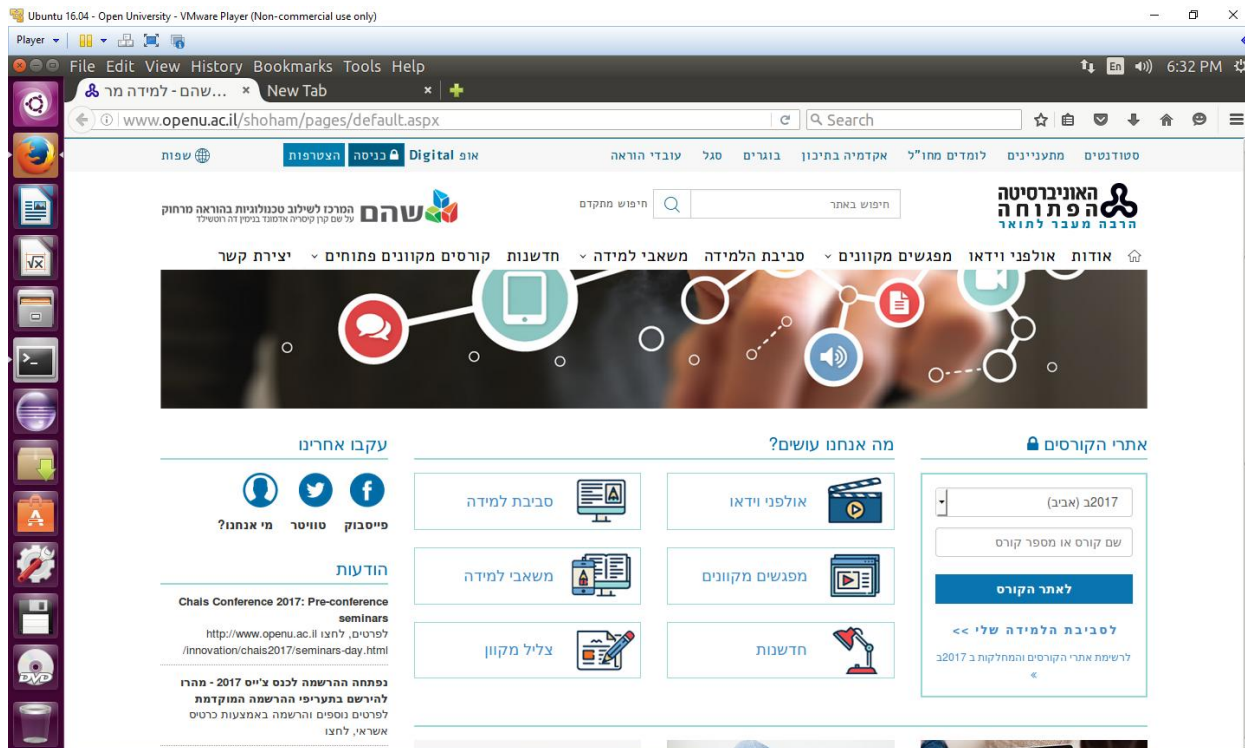
To terminate the session and shut down the virtual machine, just click on the shutdown button.



WARNING: Linux systems are different from Windows systems in the way they operate their hard disks. Saving file in Windows system causes it to be immediately written to the hard disk. Linux systems often defer physical writing in order to optimize disk arm motions. Therefore, in order to avoid losing data don't power off VMware Player from Windows.

Connecting the Internet

The virtual machine was configured to use the Internet connection of the host operating system. Once internet connection is established for Windows, switch to the virtual machine; launch Firefox Internet browser and start browsing the Internet:



Part 2

Programming environment

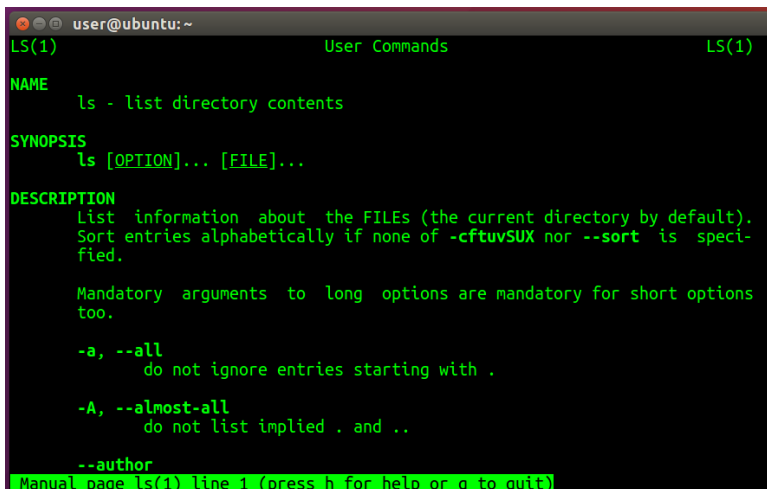
How to get some help?

Linux has a very extensive online documentation system documenting every available command and many library routines. To get manual page run from the terminal “man” command followed by the subject of your interest.

For example, launch a terminal and type “man ls” in the command line.

A terminal window with a dark background. The prompt is 'user@ubuntu: ~'. The command 'man ls' has been entered, and a green cursor is at the end of the line.

Comprehensive help on the “ls” command (which is similar to the “dir” command in MSDOS) will be displayed.

A terminal window showing the output of the 'man ls' command. The output is formatted with green text on a dark background. It includes sections for NAME, SYNOPSIS, and DESCRIPTION. The DESCRIPTION section explains that 'ls' lists information about files and directories, and lists various options like -a, -A, and --author. At the bottom, it says 'Manual page ls(1) line 1 (press h for help or q to quit)'.

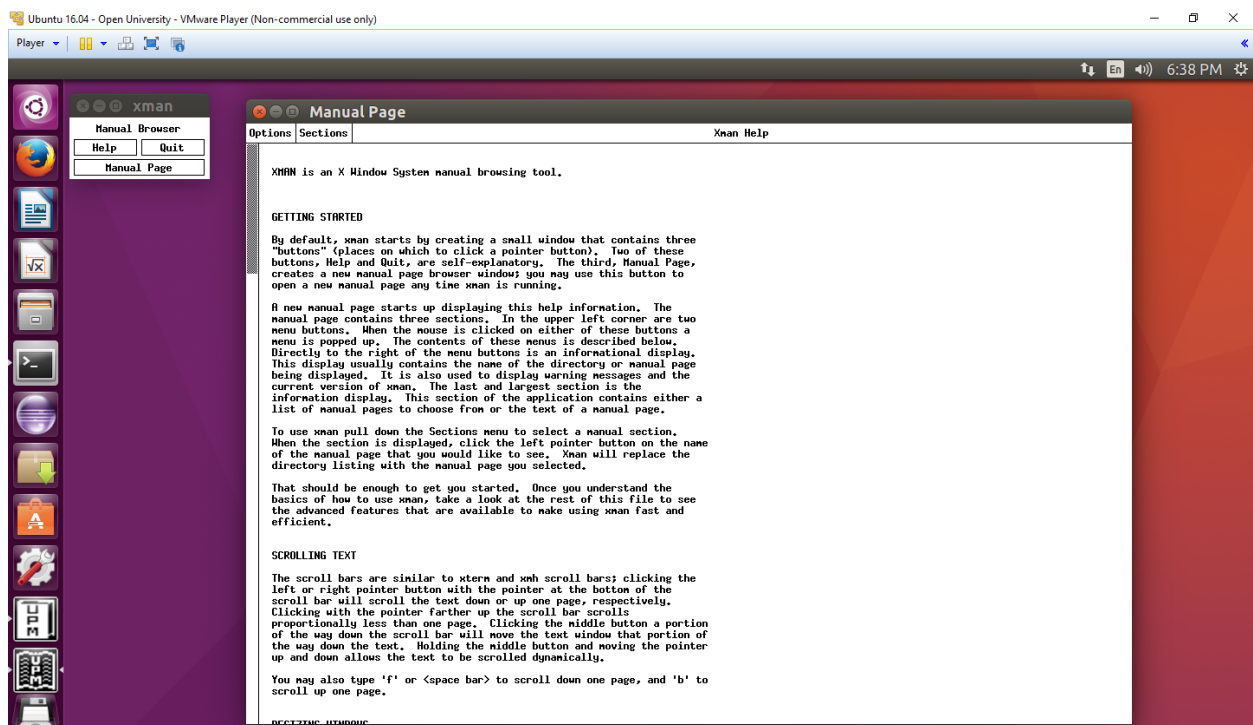
The manual (man pages) is organized in sections. Section 1 describes general user commands and sections 2 and 3 are a Linux programmer manual. The first line in the output of the “man” command displays the section number (see the top left corner of the screen). Sections 2 and 3 will be the most widely used preparing your programming assignments.

Sometimes, a subject of your interest may have similar syntax with other command, function etc. For example, typing “man read” displays the information related to the Tcl built-in command in the “Tcl built-in commands” section, while typing “man 2 read” supplies the library function description from the Linux programmer’s manual (man-pages, section 2).

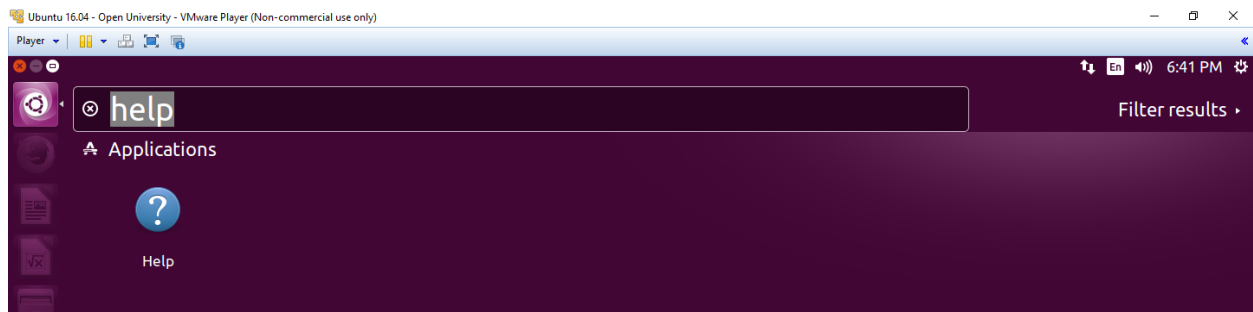
If the exact syntax is unknown, use the flag `-k`. For example, “man `-k` read” displays a list of subjects that have a string "read" in their short description of the manual pages database. Then one can choose which commands to read about.

You can alternatively use a graphical version of “man” invoked by the “xman” command. Click on the button named “Manual pages” and then select “Section” from the menu.

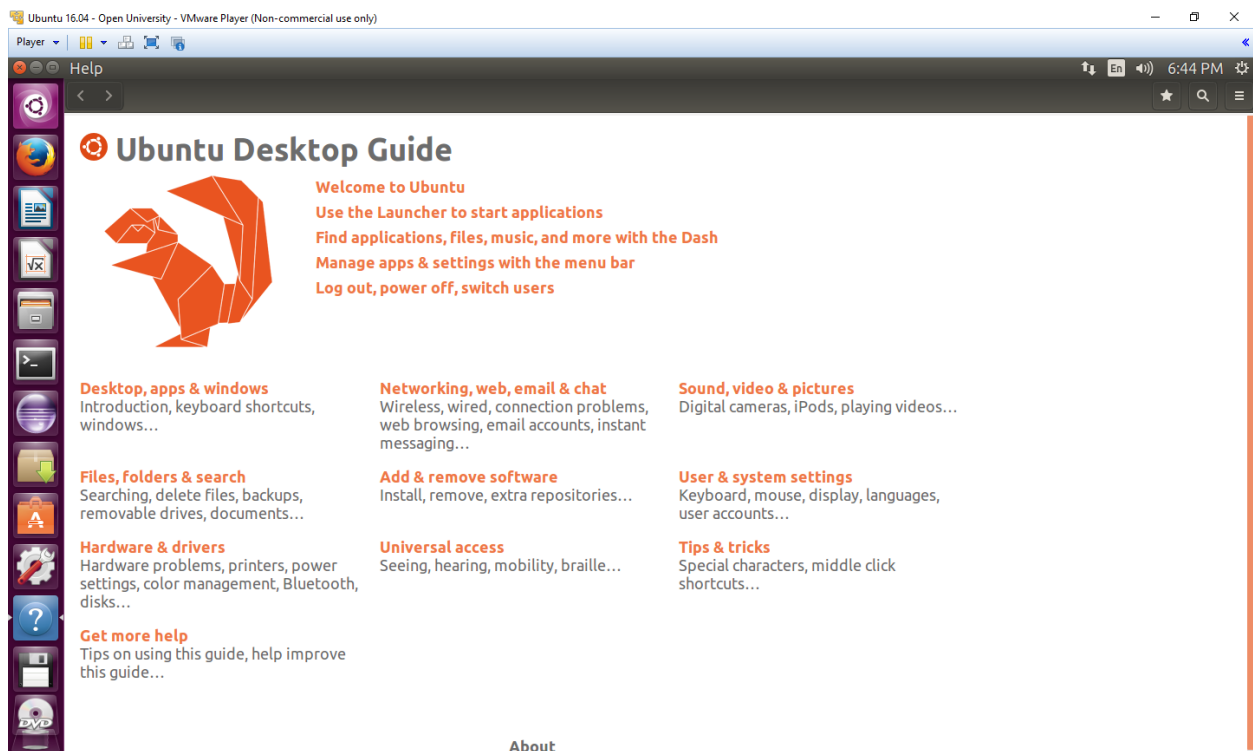
NOTE: To scroll use the middle mouse button.



Ubuntu related help topics are organized by the Ubuntu documentation project. Click on Dash button and type 'help'.



Double click on the “Help” application will launch the “Ubuntu desktop guide”:



Command line interpreters - shells

Shells are special programs which act as command-line interpreters. Many shells are available on Linux. The default shell launched with the Ubuntu terminal window is ``bash``. BASH stands for “Bourne-Again Shell” where the Bourne Shell (sh) was the first UNIX shell and today is the standard UNIX shell available on every UNIX/Linux workstation. With that, it is not widely used nowadays as there are newer and more advanced shells.

Bash is a sh-compatible command language interpreter that executes commands read from the standard input or from a file. Bash incorporated a bunch of useful features from the Korn and C shells (ksh and csh), which makes ``bash`` to interpret a syntax reminiscent of the programming language C.

The following table presents what can you do before hitting ``Enter`` button when you type a command:

Goal	Action
To move in the command line	Use the left-arrow and right-arrow keys
To insert new characters to the left of the cursor	Type new character
To delete the character to the left of the cursor	Use the Backspace or Delete key
To erase the character at the cursor	Hit Ctrl-D
To delete the characters from the cursor position to the end of the word	Hit ESC-D
To move the cursor to the beginning of the command line	Hit Ctrl-A
To move the cursor to the end of the command line	Hit Ctrl-E
To kill the command line from the cursor to the end of the line.	Hit Ctrl-K
To kill the command line to the beginning of the command line	Hit Ctrl-U

To clear the screen, puts the current command line at the top of the screen	Hit Ctrl-L
To transpose the two characters to the right and left of the cursor	Hit Ctrl-T

The structure of Linux commands

As mentioned earlier, in the paragraph “How to get some help?”, section 1 of the man-pages is the section describing general user commands.

Commands are made up from a verb - the command itself, which is single word, usually consisting of lowercase letters. The verb may be followed by switches(flags) and/or arguments. Spaces separate the different components from each other.

Command switches (or flags) modify the command's action. Switches are usually preceded by a ``-`` sign, although there are exceptions (exceptions are out of our concern currently). Commands can have an arbitrary number of switches, and you may specify as many of them as you need (of course, you shouldn't specify contradictory ones). For many commands, you may specify all the switches in sequence, with one ``-`` preceding them, and no spaces (e.g. `-xyz`). You may also specify each switch separately, with the ``-`` preceding each (e.g. `-x -y -z`). **Refer to the appropriate man page to find the command's synopsis!**

Command arguments specify the entities to which the command is applied. The number of arguments a command may take can be arbitrary or fixed, and spaces separate the different arguments.

To summarize, let us look at the command ``ls``. Used by itself, with no switches and arguments, it produces a list of files and folders in the current directory.

The following figure presents this list in the user's homedir (home directory):

```
user@ubuntu:~$ ls
courses  Documents  examples.desktop  Pictures  Templates  vmware-tools-patches
Desktop  Downloads  Music             Public    Videos
```

One of 'ls' switches is a '-a' switch, which forces printing of hidden files and hidden directories which otherwise is suppressed (hidden directory or file starts with '.').

Following figure presents the list of all files and folders in user's homedir:

```
user@ubuntu:~$ ls -a
.      Desktop      .ICEauthority  .sudo_as_admin_successful
..     Documents     .local         Templates
.bash_history Downloads      .mozilla       .thunderbird
.bash_logout examples.desktop .mozilla-sav   Videos
.bashrc   .gconf        Music          vmware-tools-patches
.cache    .gnome2       Pictures       .Xauthority
.config   .gnome2_private .profile       .xsession-errors
courses  .gnupg        Public         .xsession-errors.old
```

The following figure shows an example of "ls" command using arguments and flag concatenation. "file*" is a 'wildcard' argument. To learn more please refer to the paragraph "Wildcards" and "-al" is equal to "-a -l". Files "file1", ..., "file3" were created by command "touch <fileN>"):

```
user@ubuntu:~$ touch file1
user@ubuntu:~$ touch file2
user@ubuntu:~$ touch file3
user@ubuntu:~$ ls -al file*
-rw-rw-r-- 1 user user 0 Mar 20 18:50 file1
-rw-rw-r-- 1 user user 0 Mar 20 18:50 file2
-rw-rw-r-- 1 user user 0 Mar 20 18:50 file3
user@ubuntu:~$
```


The file system

Under Linux, all files are organized in one file system in a tree-like structure. Files can be grouped together in directories (or folders), which themselves are grouped together in other directories, And finally everything is grouped together under the file system root, which is denoted by the symbol /.

The directories ``bin'`, `etc'`, `tmp'`, `var'`, `dev'`, `home` and `usr` are standard Linux directories containing system files, user accounts etc. The location of user accounts, and therefore user files and directories is chosen by the system administrator. On the Ubuntu user accounts are located under the directory `/home/' which has subdirectories named by the users' names. So, user named `user` has a homedir named “user” and its full path is /home/user.`

Following is a table of commands related to file system traversal:

Goal	Command	Example
To move from directory to directory (“..” stands for the parent directory)	cd	cd ..
To print the full name of working directory	pwd	pwd
To list the contents of working directory	ls	ls ~user

Note: ``~<username>`` is shorthand of ``/home/<username>`` (e.g. ``~user`` = ``/home/user``).

Wildcards

Linux shells let you use wildcards as shortcuts in specifying file names. Wildcards are special characters which are used to group many files together:

- The `*` matches zero or more characters. For instance, `ls *` will display all the files in the current directory, and `ls a*b` will display all the files whose names start with an `a`, and end with a `b` (including the file `ab`, if it exists).
- The `?` matches a single character. For instance, `ls a?b` will display all the files whose names are 3-characters long, starting with an `a` and ending with a `b`, and `ls ?????` will display all files whose names are exactly 5-characters long.
- The `[]` match any single character from the set specified inside the brackets. Thus, `ls [abc]d[abc]` will display all files whose names are exactly 3 characters long, have an `a`, `b` or `c` as the first and last character in their name, and a `d` as the middle character. If you want to match a consecutive range of letters or digits, you may use a `-` to specify the range. For example, `ls [a-z]*[0-9]` will display all files whose names are at least 2-characters long, which start with a lowercase letter, and end with a digit.

Note, that on the Linux systems the period (`.`) in the middle of the file (directory) name has no special significance (to stress this fact, note that even executable files on Linux/UNIX systems have no `.exe` extension). Therefore, wildcards will also match the period.

Several file-related commands

The reader is invited to look for more information in the man pages:

Goal	Command
To print the contents of the file onto standard output	cat
To print the contents of the file to standard output allowing only forward movement (by hitting the `Enter` or `space`)	more
To print the contents of the file onto standard output allowing both forward movement (`Enter` or `space`) and backward movement (Ctrl-`up arrow`)	less
To copy files	cp
To move files	mv
To remove all files whose names were specified as arguments	rm
To create a new file	touch
To change the modification date of files	touch
To compare between two files	diff
To find the location of a file in the file system	find
To search for a pattern in a file	grep
To display the first few lines of the specified file	head
To display the last few lines of the specified file	tail

Caution: Be careful when deleting files. There is no way to restore them afterwards.

Several directory-related commands

The reader is invited to look for more information in the man pages:

Goal	Command
To copy one directory onto another	cp
To move one directory onto another	mv
To print the contents of the directory onto standard output	ls
To remove empty directory	rmdir
To remove non-empty directory	rm -r
To create a new directory	mkdir
To find directory in the file system	find

Caution: Be careful when deleting directories. There is no way to restore them afterwards.

Redirecting Input/Output

By default, shell commands take their input from the standard input (which usually is the keyboard), and direct their output to the standard output (usually the terminal's screen). You may modify this behavior by using the `<`, `>` and `>>` symbols.

The `<` defines a file to be used instead of the standard input. The `>` defines a file to be used instead of the standard output. Every time you use the `>` symbol, the file is recreated, i.e. its previous contents is lost. If you want to append new data to an existing file, use `>>` instead.

For example, typing ``ls -al > file_list`` causes to file `file_list` to be created/recreated and the output of ``ls -al`` to be positioned therein.

Bash provides one more way of I/O redirection called pipes. Pipes are used to concatenate two or more commands, so that the output of one command will be the input of the next command. Pipe-line symbol ``|`` is used to indicate shell that pipes are used.

For example, ``ls | sort`` will sort the output of `ls` command alphabetically, saving keystrokes and eliminating the necessity of creating intermediate files when no pipes were used.

Another example demonstrates the way to count the number of words in some given file:

``cat some_file | wc -w``.

Protecting files and directories, altering protections

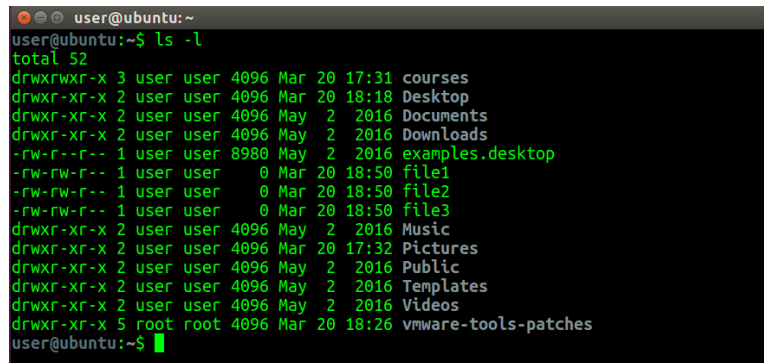
Protection scheme in native Linux file system applies to three categories of populations:

Notation of population	Description of population
u	the user's (or owner's) protections – reflects the state of protection ‘owner vs. everybody else’.
g	the owner's group protections – reflects the state of protection ‘owner vs. users that belong to the groups that the owner belongs to also’. Group of users may be a group of all 3rd-year students, defined by system administrator.
o	all other users protections – reflects the state of protection ‘owner vs. everybody else’

There are three basic categories of protections:

Notation of category	Description of category
R	for read protection. This protection determines whether a user may read a file or directory, and get a list of the files in a directory.
W	for write protection. This protection determines whether a user may write to a file , and create or delete files in a directory.
X	for execute protection. This protection determines whether a user may execute (run) a file (assuming it an binary executable or script file), and whether a user may perform <code>`cd`</code> command to a directory.

For example, see the following figure listing the contents of folder ~user :



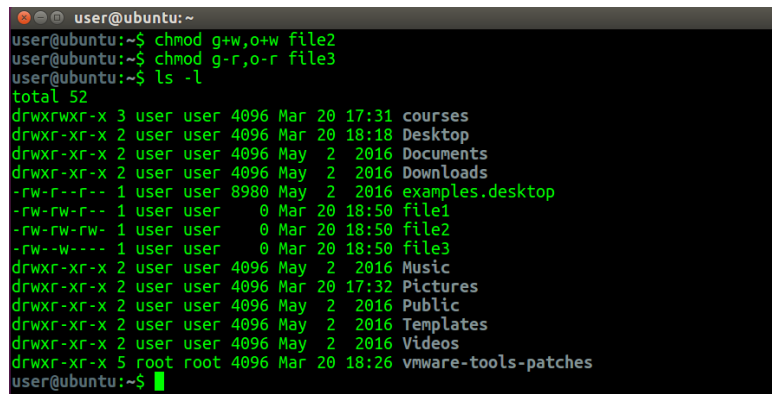
```
user@ubuntu:~  
user@ubuntu:~$ ls -l  
total 52  
drwxrwxr-x 3 user user 4096 Mar 20 17:31 courses  
drwxr-xr-x 2 user user 4096 Mar 20 18:18 Desktop  
drwxr-xr-x 2 user user 4096 May 2 2016 Documents  
drwxr-xr-x 2 user user 4096 May 2 2016 Downloads  
-rw-r--r-- 1 user user 8980 May 2 2016 examples.desktop  
-rw-rw-r-- 1 user user 0 Mar 20 18:50 file1  
-rw-rw-r-- 1 user user 0 Mar 20 18:50 file2  
-rw-rw-r-- 1 user user 0 Mar 20 18:50 file3  
drwxr-xr-x 2 user user 4096 May 2 2016 Music  
drwxr-xr-x 2 user user 4096 Mar 20 17:32 Pictures  
drwxr-xr-x 2 user user 4096 May 2 2016 Public  
drwxr-xr-x 2 user user 4096 May 2 2016 Templates  
drwxr-xr-x 2 user user 4096 May 2 2016 Videos  
drwxr-xr-x 5 root root 4096 Mar 20 18:26 vmware-tools-patches  
user@ubuntu:~$
```

The first character on each line is usually the `-` for a file and a `d` for a directory. The next 9 characters specify the protections, and are grouped in 3 groups of 3 characters each, for the user (=owner), group and other protections. Thus in the example above, `file1` is a file, whose owner (`user`) has read and write privileges on it, and the group and others have only read privileges on it. That means that the user `user` may read the file and modify it, and everybody else may only read it. The directory `tmp` has the `rw` bits set for the owner. Therefore, `user` may see a list of files in `tmp` (because of `r`), may create and delete files there (because of `w`) and may `cd` to `tmp` (because of `x`). All other users (including the group) may see a list of files, may make a `cd` to the `tmp` directory, but may not modify it.

NOTE: The protection mechanism has more than 3 basic schemes. You are invited to refer to the `chmod`, `chgrp`, `chusr` commands' man-pages.

chmod command

To change a file or directory protections one can use the command `chmod`. For example, to give a write permission to everybody for the file2 and to revoke the read permission for everybody except the owner of file3, enter the command `chmod g+w,o+w file2` and immediately afterwards the enter the command `chmod g-r,o-r file3`. The result is displayed bellow:

A terminal window screenshot showing the execution of chmod commands and the resulting file permissions. The terminal output is as follows:

```
user@ubuntu:~  
user@ubuntu:~$ chmod g+w,o+w file2  
user@ubuntu:~$ chmod g-r,o-r file3  
user@ubuntu:~$ ls -l  
total 52  
drwxrwxr-x 3 user user 4096 Mar 20 17:31 courses  
drwxr-xr-x 2 user user 4096 Mar 20 18:18 Desktop  
drwxr-xr-x 2 user user 4096 May 2 2016 Documents  
drwxr-xr-x 2 user user 4096 May 2 2016 Downloads  
-rw-r--r-- 1 user user 8980 May 2 2016 examples.desktop  
-rw-rw-r-- 1 user user 0 Mar 20 18:50 file1  
-rw-rw-rw- 1 user user 0 Mar 20 18:50 file2  
-rw--w---- 1 user user 0 Mar 20 18:50 file3  
drwxr-xr-x 2 user user 4096 May 2 2016 Music  
drwxr-xr-x 2 user user 4096 Mar 20 17:32 Pictures  
drwxr-xr-x 2 user user 4096 May 2 2016 Public  
drwxr-xr-x 2 user user 4096 May 2 2016 Templates  
drwxr-xr-x 2 user user 4096 May 2 2016 Videos  
drwxr-xr-x 5 root root 4096 Mar 20 18:26 vmware-tools-patches  
user@ubuntu:~$
```

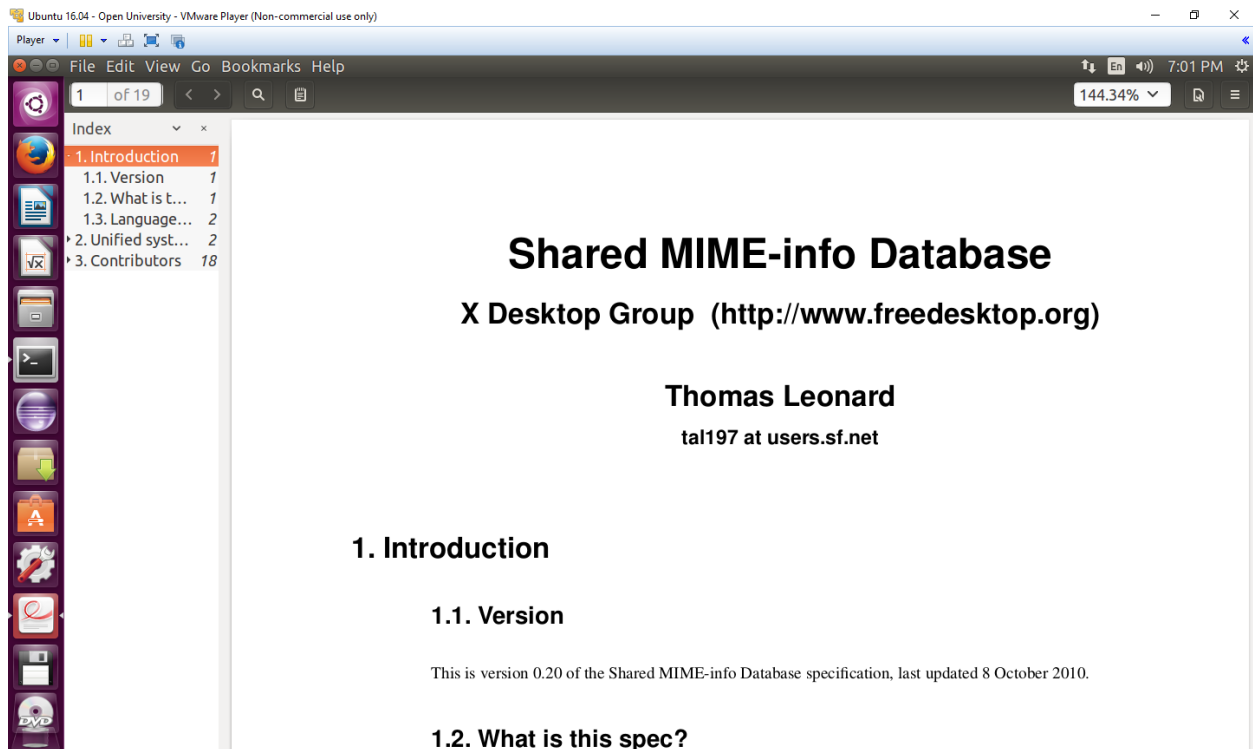
For more information about the chmod command the reader is invited to look in the man pages.

PostScript and PDF file viewer

Ubuntu comes with several viewers for PostScript (*.ps) and PDF (*.pdf) files. But, probably, the most friendly one is a *evince* previewer. Launch:

```
“evince evince /usr/share/doc/shared-mime-info/shared-mime-info-spec.pdf &”
```

from the terminal.



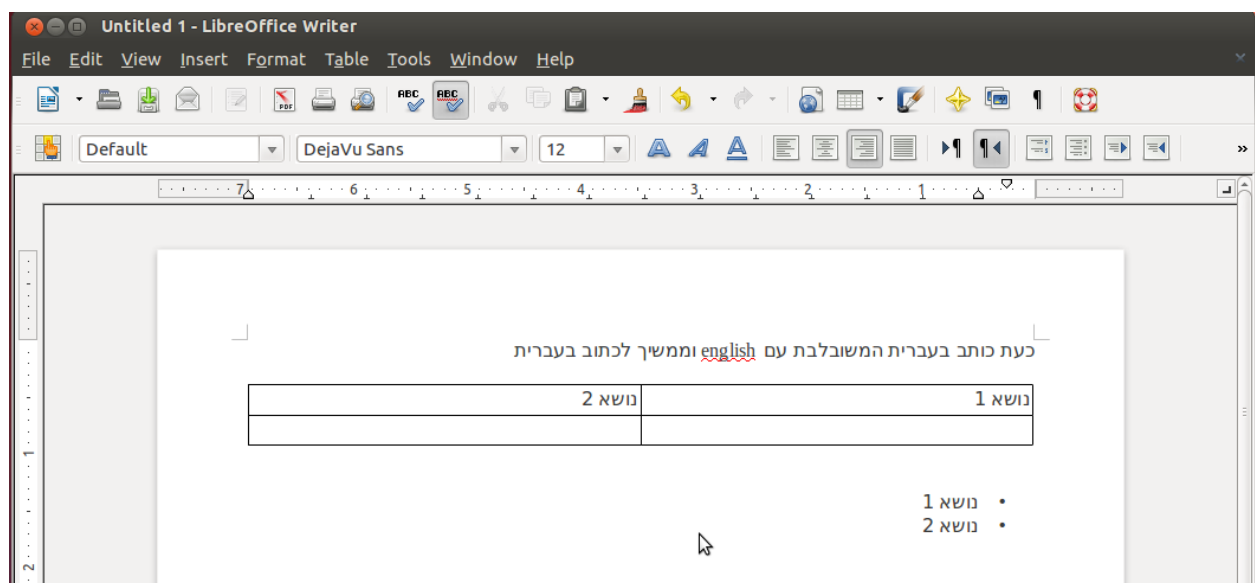
Editors

Find experimentally the editor that is most suitable for you (from the Applications menu). This manual briefly relates some of them.

LibreOffice writer

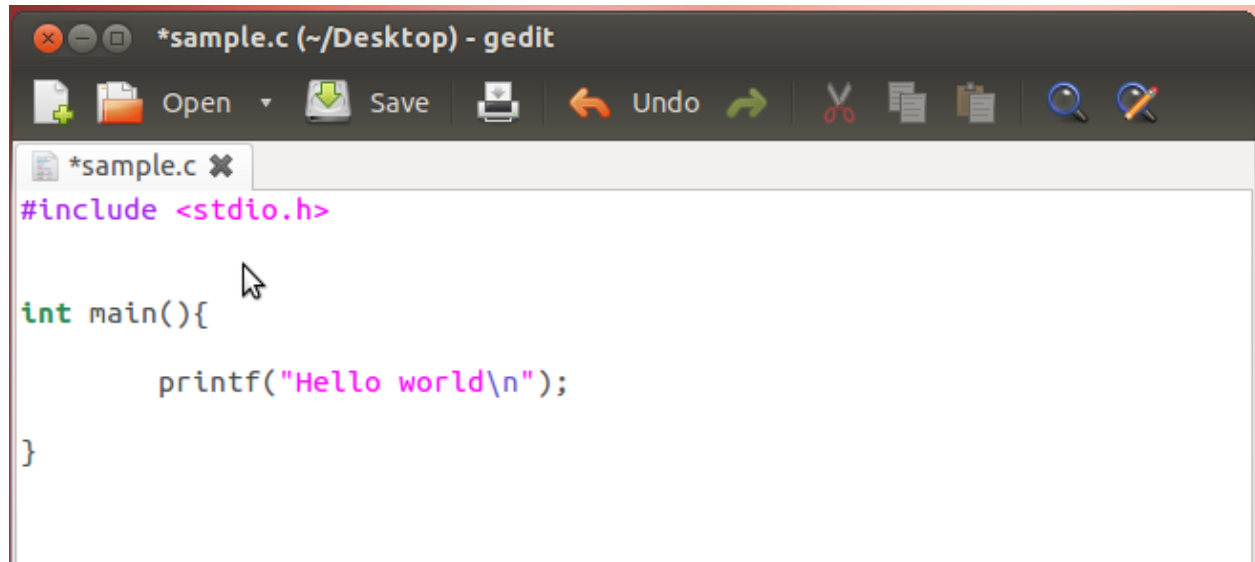
LibreOffice is a free and open source office suite. It is descended from OpenOffice.org. LibreOffice is designed to be compatible with other major office suites, including Microsoft Office, though some Microsoft Office layout features and formatting attributes are handled differently or unsupported.

Note: To change keyboard map from English to Hebrew use familiar ALT-Shift combination



gedit editor

gedit editor is a simple and self explanatory text editor that supports automatic highlighting any *.c or *.h files. *gedit* is launched by typing “*gedit &*” from the terminal:



Emacs editor

Emacs is a complete working environment, rather than just a text editor. Among many other features Emacs has programming modes that support an:

- automatic color-syntax highlighting for programming languages
- automatic indentation and code formatting
- interfacing with one's favorite debugger
- addition of language-specific menus to the Emacs menu bar

Properly configured, Emacs is quite like a commercially marketed Integrated Development Environment for programming languages like C++ and Java. But configuring Emacs is a subject to be covered in a different manual. **Beginners are better to use Emacs as a code editor with automatic support of programming languages highlighting, indentation and formatting.**

Emacs is invoked by the command emacs (type “emacs &” from the terminal). Graphical user interface provides elaborate menu. But beginners may find emacs to be a little bit nonhabitual in several ways. So, reading “Emacs Tutorial” that is available from the program’s Help menu might be useful.

Vi editor

Vi is a very powerful editor found on every Unix/Linux machine. Ubuntu comes with enhanced version that is backward compatible with vi and called vim (vi improved). But beginners may discover it somehow tedious to use non-graphical text processor, however powerful is it. Vi supplies features that no other editor familiar so far can supply. Bellow is a summary intended to provide only a basic outline (everything is referred to VISUAL mode, unless a preceding colon ':' is specified):

Invocation	
vi file	Edit at first line of 'file'
Exiting	
:wq	Write buffer, then quit.
:w	Write buffer to original file
:w file	Write buffer to 'file'
:w>> file	Append to 'file'
:q	Quit out of editor
:q!	Quit discarding changes
Text input (ESCape key terminates Input mode)	
i	Insert before cursor
Insert Mode Commands	
Control-@	Repeat Last Insertion
Control-W	Erase last word entered
DElete or Backspace	Same as Control-H
Control-U	Erase entire line
Cursor Motion	
down-arrow or j	Move cursor down one line
up-arrow or k	... up one line, staying in same column
left-arrow or h	... to the left
right-arrow or l	... to the right
Substitution	

:s/X/Y/opt	Substitute 'Y' for first occurrence of 'X' Options: g - Change every occurrence in line c - Confirm each change p - Print each change (Command mode only)
:g/X/s//Y/opt	Globally find the first 'X' on each line and substitute 'Y' for the string 'X'
Deleting Text	
x	Delete character under cursor
nx	... 'n' characters
dd	... entire line
D	... to end of line
Undo	
u	Undo last change
U	Restore current line

The gcc compiler

gcc is a program used to compile C programs. Process of compilation consists of 3-steps: preprocessing, compilation and linkage. In the first stage, the user's program is preprocessed. During that stage all preprocessor directives are fulfilled and preprocessed code files are prepared for the second stage – a compilation. Compilation is a translation from the programming language to the machine language. During this stage an intermediate files, called an object files, are created. The last stage is a linkage, standing for the linkage of created object files in an executable program.

In many cases this 3-step process is transparent for the user. For example a command ``gcc sample.c`` creates executable named `a.out` and combines all above steps. But while writing big projects using many files, we can find it useful to split the steps of compilation. For more details see the paragraph “Make utility”.

Following table lists several gcc–related switches. For detailed information refer to man-pages:

Switch	Example	Description
-c	<code>gcc -c sample.c</code>	Creates object file <code>sample.o</code> (proceeding the preprocessing and compilation steps)
-o	<code>gcc sample.c -o sample</code>	Creates executable file with the name <code>sample</code> , instead of the default <code>`a.out`</code>
-Wall	<code>gcc -Wall sample.c</code>	Displays compilation warnings that were suppressed otherwise
-g	<code>gcc -g sample.c</code>	Links debugging libraries into the executable which is necessary for further debugging.

Combining all above results in ``gcc -Wall -g sample.c -o sample`` directing the ‘gcc’ to create executable file named ‘sample’ that incorporates debugging libraries while compiler doesn’t suppress compilation warnings.

There are lots of options for ‘gcc’, which are all in the man page. Here are a few of the advanced ones, with examples of how to use them:

Switch	Example	Description
-ansi	gcc -ansi sample.c	<p>Check that your code complies to the relevant international standard, often referred to as the ANSI standard, though strictly speaking it is an ISO standard. One of the main aims of the standard is to allow people to write code that will work with any compiler on any system. This is known as portable code.</p> <p>‘-ansi’ turn off most, but not all, of the non-ANSI C features provided by ‘gcc’. Despite the name, it does not guarantee strictly that your code will comply to the standard</p>
-pedantic	gcc -Wall -ansi -pedantic sample.c -o sample	<p>‘-pedantic’ turns off all ‘gcc’ 's non-ANSI C features.</p> <p>This example produces an executable ‘sample’ after checking ‘sample.c’ for standard compliance. Generally, you should try to make your code as portable as possible, as otherwise you may have to completely re-write the program later to get it to work somewhere else - and who knows what you may be using in a few years time?</p>
-O	gcc -O sample.c -o sample	<p>‘-O’ creates an optimized version of the executable. The compiler performs various clever tricks to try and produce an executable that runs faster than normal. You can add a number after the ‘-O’ to specify a higher level of optimization. Optimization is usually only turned on when compiling a release version.</p>

Libraries

Libraries provide a way of organizing modules into larger entities. We often need a way to make a collection of modules (e.g. object files), that provide common functionality, things that many programs will want to do – working with files, for example. The pieces that do these things may be provided for you in the form of ‘static’ or ‘dynamic’ libraries. Library can be thought as a super-object file, providing collection of capabilities with certain functionality (functions, classes, etc.). Libraries are usually created from a set of objects. Library interface (C/C++ header files, Java API docs, ...) is often provided to the developers together with the binary library.

The difference between static and dynamic libraries is in the type of linking. Static libraries are physically copied into the executable file, while dynamic libraries are loaded on demand. Dynamic libraries are often called ‘shared libraries’ because library files are usually shared by different programs thus saving disk space by reducing the size of executable files.

Static libraries have names like `libname.a`. They are created running from the command line:

```
ar rcu libname.a object1.o object2.o ...
ranlib libname.a
```

'ar' utility simply 'archives' the .o files into the repository named 'libname.a'. The suffix .a comes from 'archive'. 'ranlib' utility goes over the archive and extracts the symbols tables from individual .o files writhing the archive and builds a large symbol table for all the library. Symbol table contains names (function names, global variables' names, ...) and the position of the machine code implementing them. Linker uses this information when producing executables to find the code of each function and to check whether the global variables are defined.

Suppose, files 'my_read.c', 'my_write.c' contain library functions for I/O. Following lines create static library for I/O:

```
gcc -Wall -c my_read.c
gcc -Wall -c my_write.c
ar rcu libmy_io.a my_read.o my_write.o
ranlib libmy_io.a
```


And finally, one should link all object files and libraries altogether. Again, suppose that 'module1.c',..., 'moduleN.c' are compiled and some of them are using I/O library functions. Then the following line compiles 'driver.c' that contains 'main()' and links program modules and libmy_io.a into the executable named 'driver':

```
gcc -Wall -L. driver.c module.o module2.o ... moduleN.o -lmy_io -o driver
```

'-lmy_io' stands for the library 'libmy_io.a' and the search for the file 'libmy_io.a' is made in the current directory which is indicated by the '-L.' flag ('.' stands for the current directory).

Shared libraries have names like lib**name**.so.x.y.z where x.y.z is a form of version number. They are created running from the command line:

```
gcc -fPIC -c file1.c
...
gcc -fPIC -c fileN.c
gcc -shared libname.so file1.o file2.o ... fileN.o
```

'gcc -fPic' or '-fpic' directs to compile for "Position Independent Code". When object files are generated, we have no idea where in memory they will be inserted in a program that will use them. Many different programs may use the same library, and each load it into a different memory in address. Thus, we need that all jump calls ("goto", in assembly speak) and subroutine calls will use relative addresses, and not absolute addresses. Thus, we need to use a compiler flag that will cause this type of code to be generated. The first three commands compile the source files with the 'fPIC' option, so they will be suitable for use in a shared library (they may still be used in a program directly, even though they were compiled with 'fPIC').

'gcc -shared' used to generate shared library, and directs compiler (or linker) that it should relate a shared library, not a final program file.

The compilation part is easy. It is done almost the same as when linking with static libraries:

```
gcc -Wall driver.c -L. -lname -o driver
```

The linker will look for the file 'libutil.so' (-lutil) in the current directory (-L.), and link it to the program, but will not place its object files inside the resulting executable file 'driver'.

If both a static (lib**name**.a) and dynamic (lib**name**.so) libraries are present in the current directory, the default behaviour is to link the shared library). To force the linker to link statically you need to specify the '-static' option.

The make utility

As mentioned in previous paragraph, sometimes it is useful to deliberately split the compilation process into separate stages. For example, suppose we wrote a text editor of ten thousand lines of code that spread over the twenty source code files. It worth only to compile files first to produce an object files and then, as a final stage, to link them to produce an executable file. Doing it this way we can significantly reduce the time of consequent compilations when some of a source files will be changed, avoiding a compilation of an unchanged source files.

In many cases, especially when working with big projects, another dimension is added to the compilation process: we need to produce a number of targets (for example a number of executables or a shared libraries) and it seems to be very tedious to check every source file whether it was changed or whether not just to save a time of recompilation. To optimize this process utility called a make utility was created. Make utility automatically determines modified source files concluding that corresponding object files (libraries, executables) need to be updated and issues commands to update them.

To use the ‘make’ utility one have to write a structured file (often called a makefile) that describes a relationships among your source files in each of your target programs and states commands generating those targets.

Once a suitable makefile exists, each time you change some source files, running ‘make’ command from the command line suffices to perform all necessary recompilations. The make utility uses the makefile data and files’ last-modification times to decide which of the files need to be updated. For each of those files, ‘make’ utility issues commands stated in the makefile.

As was mentioned, makefiles are structured files. As a matter of fact, elements of that structure form a versatile scripting language, but we’ll see only some of those elements:

Scripting language element	Description
<pre>file.o: file.c utils.c file.h gcc -c -g file.c utils.c</pre>	<p>This is an example of explicit rule which states that the target file.o depends on dependencies file.c, utils.c and file.h and the command to produce the target is <code>`gcc -c -g file.c utils.c`</code>.</p> <p>NOTE, that the command should start with TAB in the beginning of the command line. If TAB is missing or space keys are stroked instead of TAB, ‘make’ utility stops with error message:</p> <p>“Makefile:line#**Missing separator. Stop”</p>
<pre>libutils.a: utils1.o utils2.o ar rcu libutils.a utils1.o utils2.o ranlib libutils.a</pre>	<p>This is an example of explicit rule with target that needs two commands to be issued. In general, there may be more than one command line.</p>
<pre>SRC_FILES = file1.c file2.c file3.c ex1: \$(SRC_FILES) gcc -o ex1 \$(SRC_FILES)</pre>	<p>This is an example of variable definition and its usage. We defined a SRC_FILES variable and the consequent use of that variable saves us several keystrokes and is seems to be also less error-prone.</p> <p>\$(SRC_FILES) is a directive to use the value of variable SRC_FILES.</p>
<pre># ***** # this makefile is last modified at 2/3/90 #*****</pre>	<p><code>`#`</code> starts a comment which spans until end of the line</p>
<pre>ex1: file1.c file2.c file3.c gcc -Wall -o ex1 file1.c \ file2.c file3.c</pre>	<p><code>`\`</code> indicates that the next line is a continuation of the current command</p>
<pre>all: my_server my_client my_server: server.c gcc server.c -o my_server my_client: client.c</pre>	<p>This is an example of phoney node usage. Phoney nodes are special targets that have no commands followed by.</p> <p>The only target that make utility eventually</p>

<pre>gcc client.c -o my_client</pre>	<p>produces is the first target that is bumped into when parsing the makefile. While executing, a make utility builds a tree of target dependences according to supplied makefile with the first target located in the root of a dependency tree. Those targets that were not included in the dependency tree are silently disregarded.</p> <p>The `all` phoney node prompts the make utility to generate several dependency trees. In our example, one for the target my_server and another one for the target my_client.</p>
<pre>clean:</pre> <pre>rm *.o ex1 *~</pre>	<p>Running ‘make clean’ from the command line invokes ‘rm -f *.o ex1 *~’ forcing (‘-f’ flag) all files with suffix ‘.o’, the file named executable ‘ex1’ and all files with suffix ‘~’ to be deleted. Files with suffix ‘~’ are old copies saved by the Emacs editor.</p>

Let’s see now a demonstration of to how put all those elements together. Suppose, files celsius.h, celsius.c, kelvin.h, kelvin.c, fahrenheit.h, fahrenheit.c provide functions to convert to Celsius, Kelvin and Fahrenheit representations accordingly from the remaining two. Files convert_celsius.c, convert_kelvin.c and convert_fahrenheit.c contain each a ‘main()’ function. An argument that main() receives is a temperature in Celsius, Kelvin and Fahrenheit representation accordingly. The output is a temperature converted to the other two representations. You can see the source code in “Makefile Examples” directory (the directory was archived with this manual). Let’s see three different ways to write a makefile generating executables ‘convcelsius’, ‘convkelvin’ and ‘convfahrenheit’.

Example 1:

#simple Makefile with all the dependencies explicitly listed

all: convkelvin convfahrenheit convcelsius

convfahrenheit: celsius.o kelvin.o convert_fahrenheit.o

gcc -o convfahrenheit celsius.o kelvin.o \
convert_fahrenheit.o

convcelsius: kelvin.o fahrenheit.o convert_celsius.o

gcc -o convcelsius kelvin.o fahrenheit.o \
convert_celsius.o

convkelvin: celsius.o fahrenheit.o convert_kelvin.o

gcc -o convkelvin celsius.o fahrenheit.o \
convert_kelvin.o

celsius.o: celsius.c celsius.h

gcc -c celsius.c

kelvin.o: kelvin.c kelvin.h

gcc -c kelvin.c

fahrenheit.o: fahrenheit.c fahrenheit.h

gcc -c fahrenheit.c

convert_kelvin.o: convert_kelvin.c fahrenheit.h celsius.h

gcc -c convert_kelvin.c

convert_fahrenheit.o: convert_fahrenheit.c kelvin.h celsius.h

gcc -c convert_fahrenheit.c

convert_celsius.o: convert_celsius.c fahrenheit.h kelvin.h

gcc -c convert_celsius.c

cleanup

clean:

rm *.o *~

Example 2

all: convkelvin convfahrenheit convcelsius

```
# $@ and $^ are 'automatic variables'. They are updated before
# the command is issued. The value of $@ is set to the target
# and the value of $^ is set to the list of all the dependencies
# of the rule
```

```
convfahrenheit: celsius.o kelvin.o convert_fahrenheit.o
    gcc -o $@ $^
```

```
convcelsius: kelvin.o fahrenheit.o convert_celsius.o
    gcc -o $@ $^
```

```
convkelvin: celsius.o fahrenheit.o convert_kelvin.o
    gcc -o $@ $^
```

```
# '%.o: %.c' is a 'pattern rule' telling the 'make' utility to
# search for files with suffix '.c' and create targets
# with the same name replacing '.c' by '.o'.
# $< is an automatic variable that holds the name of the current
# dependency.
```

```
%.o: %.c
    gcc -c $<
```

```
# cleanup
```

```
clean:
    rm *.o *~
```

Example 3

```
# 'make' utility allows variable definitions. For example,
# the value of the variable CC is set to the name of compiler
# and the value of HDRS is set to the list of header files
```

```
CC = gcc
```

```
CFLAGS = -Wall
```

```
LFLAGS = -L.
```

```
LIBS = -lm
```

```
HDRS = celsius.h fahrenheit.h kelvin.h
```

```
OBJS_convfahrenheit = celsius.o kelvin.o convert_fahrenheit.o
```

```
OBJS_convcelsius = kelvin.o fahrenheit.o convert_celsius.o
```

```
OBJS_convkelvin = celsius.o fahrenheit.o convert_kelvin.o
```

```
# $(VAR) is a directive to use the value of variable VAR defined before
```

```
OBJS = $(OBJS_convfahrenheit) $(OBJS_convcelsius) $(OBJS_convkelvin)
```

```
# 'OBJS:.o=.c' is a construction that directs "take the members of the list
# OBJS and create a new list substituting .o by .c"
SRCS = $(OBJS:.o=.c)
```

```
all: convkelvin convfahrenheit convcelsius
```

```
convfahrenheit: $(OBJS_convfahrenheit)
$(CC) -o $@ $^ $(LFLAGS) $(LIBS)
```

```
convcelsius: $(OBJS_convcelsius)
$(CC) -o $@ $^ $(LFLAGS) $(LIBS)
```

```
convkelvin: $(OBJS_convkelvin)
$(CC) -o $@ $^ $(LFLAGS) $(LIBS)
```

```
%.o: %.c
$(CC) $(CFLAGS) -c $<
```

```
# phony nodes
```

```
clean:
rm *.o *~ *.bak
```


Debugging your program

Ubuntu comes with a standard GNU gdb debugger. Please refer to info pages to learn about the gdb debugger as a standalone program or to get acquainted of how to run the gdb from emacs. To run the gdb debugger type 'gdb' from the command line. It worth also to mention, that we supplied Ubuntu with Eclipse Integrated Development Environment which is parallel to the Microsoft Visual Studio. We will not cover Eclipse in this brochure.

This chapter will teach you an additional way: invoking gdb using the graphical front-end named DDD.

Guidance will be based on the sample program 'sample.c'. Create in your homedir (~user) a folder named 'sample'. Copy to that folder files 'Makefile' and 'sample.c' (files are located in the '~user/courses/os' directory).

Here's the source sample.c of the sample program.

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

static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

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

```

{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}

```

The sample program ‘sample.c’ exhibits the following bug. Normally, sample should sort and print its arguments numerically, as in the following example:

```

$ ./sample 8 7 5 4 1 3
1 3 4 5 7 8
$ _

```

However, with certain arguments, this goes wrong:

```

$ ./sample 8000 7000 5000 1000 4000
1000 1913 4000 5000 7000
$ _

```

Although the output is sorted and contains the right number of arguments, some arguments are missing and replaced by bogus numbers; here, 8000 is missing and replaced by 1913.

Let us use DDD to see what is going on. First, you must compile sample.c for debugging. Run the make utility from the command line:

```

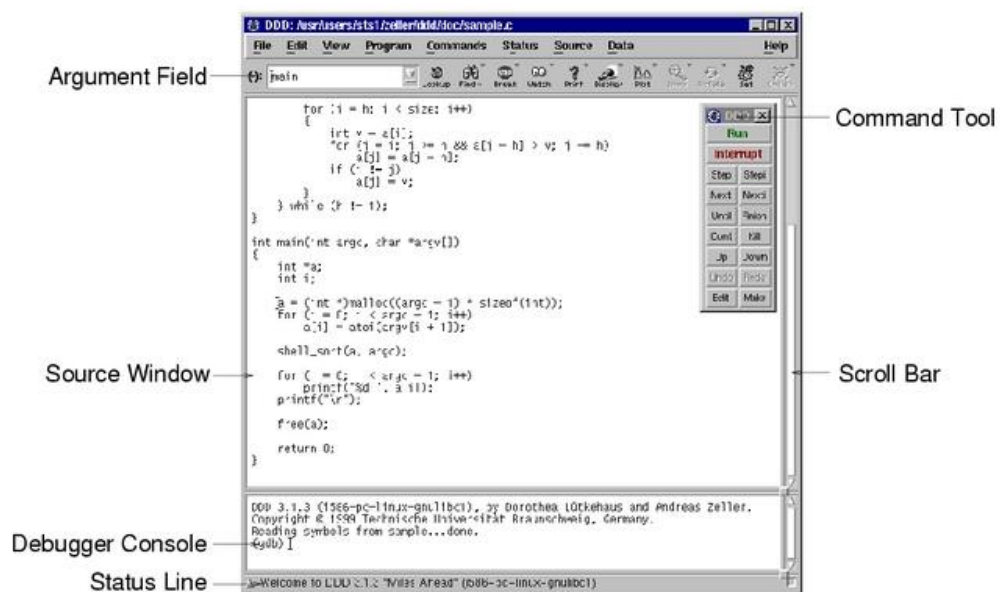
$ make

```

Now, you can invoke DDD on the sample executable:

```
$ ddd sample
```

After a few seconds, DDD comes up. The *Source Window* contains the source of your debugged program; use the *Scroll Bar* to scroll through the file.



Initial DDD Window

The *Debugger Console* (at the bottom) contains DDD version information as well as a GDB prompt.

GNU DDD Version 3.3.9, by Dorothea L tkehaus and Andreas Zeller.

Copyright   1995-1999 Technische Universit t Braunschweig, Germany.

Copyright   1999-2001 Universit t Passau, Germany.

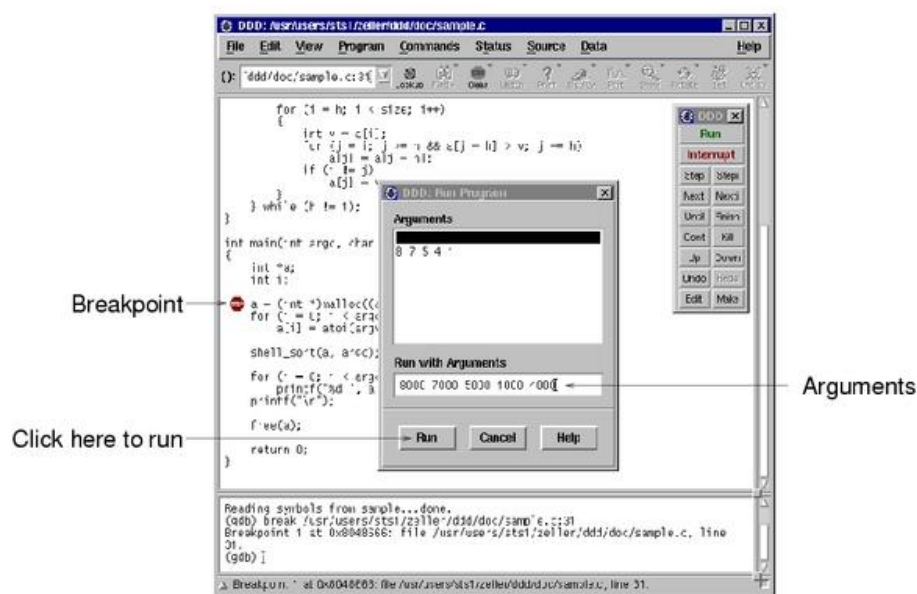
Copyright   2001-2004 Universit t des Saarlandes, Germany.

Reading symbols from sample...done.

(gdb) _

The first thing to do now is to place a *Breakpoint* , making sample stop at a location you are interested in. Click on the blank space left to the initialization of a. The *Argument field* (): now contains the location (sample.c:31). Now, click on Break to create a breakpoint at the location in (). You see a little red stop sign appear in line 31.

The next thing to do is to actually *execute* the program, such that you can examine its behavior. Select Program => Run to execute the program; the Run Program dialog appears.



Running the Program

In Run with Arguments, you can now enter arguments for the sample program. Enter the arguments resulting in erroneous behavior here--that is, 8000 7000 5000 1000 4000. Click on Run to start execution with the arguments you just entered.

GDB now starts sample. Execution stops after a few moments as the breakpoint is reached. This is reported in the debugger console.

```
(gdb) break sample.c:31
```

```
Breakpoint 1 at 0x8048666: file sample.c, line 31.
```

```
(gdb) run 8000 7000 5000 1000 4000
```

```
Starting program: sample 8000 7000 5000 1000 4000
```

Breakpoint 1, main (argc=6, argv=0xbffff918) at sample.c:31

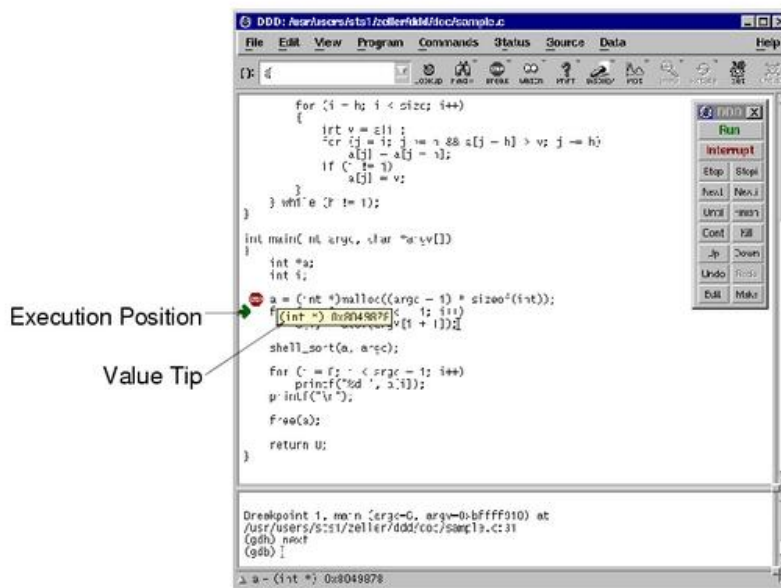
(gdb) _

The current execution line is indicated by a green arrow.

=> a = (int *)malloc((argc - 1) * sizeof(int));

You can now examine the variable values. To examine a simple variable, you can simply move the mouse pointer on its name and leave it there. After a second, a small window with the variable value pops up. Try this with argc to see its value (6). The local variable a is not yet initialized; you'll probably see 0x0 or some other invalid pointer value.

To execute the current line, click on the Next button on the command tool. The arrow advances to the following line. Now, point again on a to see that the value has changed and that a has actually been initialized.



Viewing Values in DDD

To examine the individual values of the a array, enter `a[0]` in the argument field (you can clear it beforehand by clicking on `()`) and then click on the Print button. This prints the current value of `()` in the debugger console. In our case, you'll get

(gdb) print a[0]

\$1 = 0

(gdb) _

or some other value (note that `a` has only been allocated, but the contents have not yet been initialized).

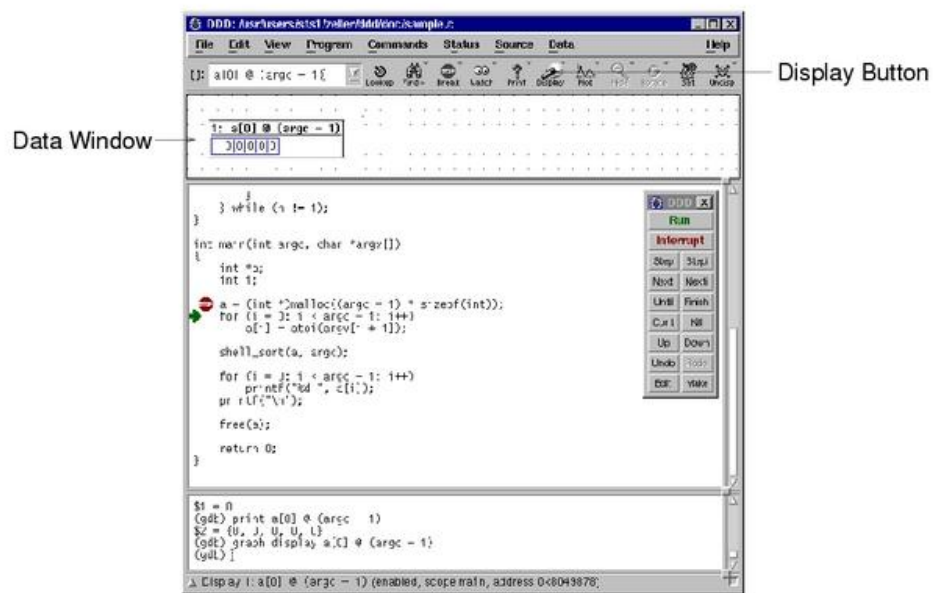
To see all members of `a` at once, you must use a special GDB operator. Since `a` has been allocated dynamically, GDB does not know its size; you must specify it explicitly using the `@` operator. Enter `a[0]@(argc - 1)` in the argument field and click on the Print button. You get the first `argc - 1` elements of `a`, or

```
(gdb) print a[0]@(argc - 1)
```

```
$2 = {0, 0, 0, 0, 0}
```

```
(gdb) _
```

Rather than using Print at each stop to see the current value of `a`, you can also *display* `a`, such that its is automatically displayed. With `a[0]@(argc - 1)` still being shown in the argument field, click on Display. The contents of `a` are now shown in a new window, the *Data Window*. Click on Rotate to rotate the array horizontally.



Data Window

Now comes the assignment of `a`'s members:

```
=> for (i = 0; i < argc - 1; i++)
```

```
a[i] = atoi(argv[i + 1]);
```

You can now click on Next and Next again to see how the individual members of a are being assigned. Changed members are highlighted.

To resume execution of the loop, use the Until button. This makes GDB execute the program until a line greater than the current is reached. Click on Until until you end at the call of shell_sort in

```
=> shell_sort(a, argc);
```

At this point, a's contents should be 8000 7000 5000 1000 4000. Click again on Next to step over the call to shell_sort. DDD ends in

```
=> for (i = 0; i < argc - 1; i++)  
    printf("%d ", a[i]);
```

and you see that after shell_sort has finished, the contents of a are 1000, 1913, 4000, 5000, 7000-
-that is, shell_sort has somehow garbled the contents of a.

To find out what has happened, execute the program once again. This time, you do not skip through the initialization, but jump directly into the shell_sort call. Delete the old breakpoint by selecting it and clicking on Clear. Then, create a new breakpoint in line 35 before the call to shell_sort. To execute the program once again, select Program => Run Again.

Once more, DDD ends up before the call to shell_sort:

```
=> shell_sort(a, argc);
```

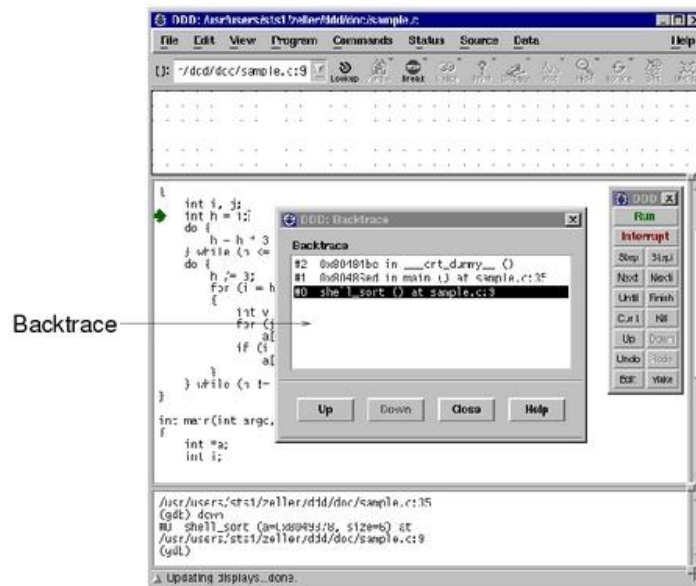
This time, you want to examine closer what shell_sort is doing. Click on Step to step into the call to shell_sort. This leaves your program in the first executable line, or

```
=> int h = 1;
```

while the debugger console tells us the function just entered:

```
(gdb) step  
shell_sort (a=0x8049878, size=6) at sample.c:9  
(gdb) _
```

This output that shows the function where sample is now suspended (and its arguments) is called a *stack frame display*. It shows a summary of the stack. You can use Status => Backtrace to see where you are in the stack as a whole; selecting a line (or clicking on Up and Down) will let you move through the stack. Note how the a display disappears when its frame is left.



The DDD Backtrace

Let us now check whether `shell_sort`'s arguments are correct. After returning to the lowest frame, enter `a[0]@size` in the argument field and click on Print:

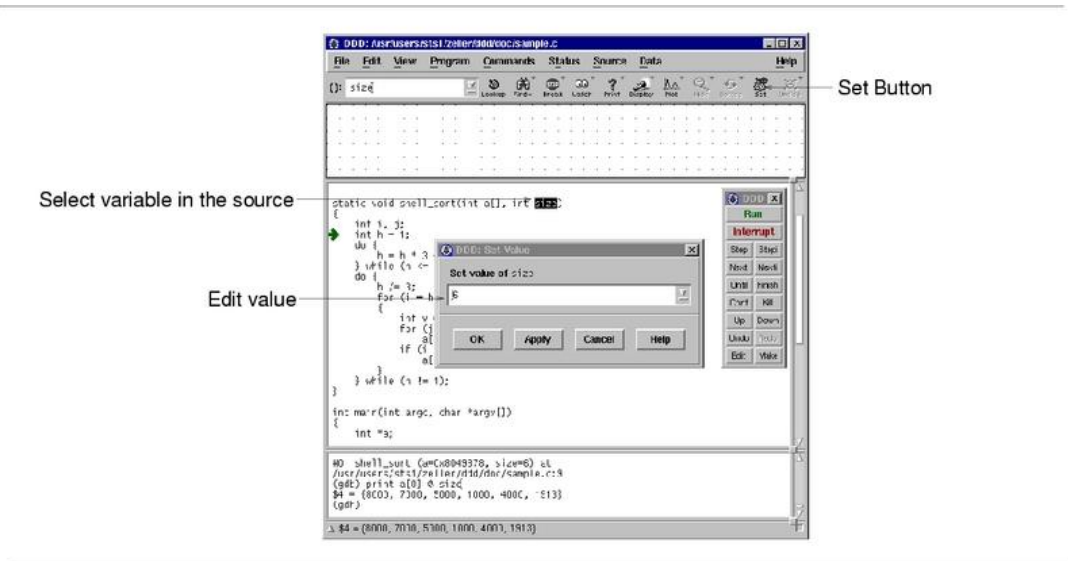
```
(gdb) print a[0] @ size
```

```
$4 = {8000, 7000, 5000, 1000, 4000, 1913}
```

```
(gdb) _
```

Surprise! Where does this additional value 1913 come from? The answer is simple: The array size as passed in `size` to `shell_sort` is *too large by one*--1913 is a bogus value which happens to reside in memory after `a`. And this last value is being sorted in as well.

To see whether this is actually the problem cause, you can now assign the correct value to size. Select size in the source code and click on Set. A dialog pops up where you can edit the variable value.



Setting a Value

Change the value of size to 5 and click on OK. Then, click on Finish to resume execution of the `shell_sort` function:

(gdb) set variable size = 5

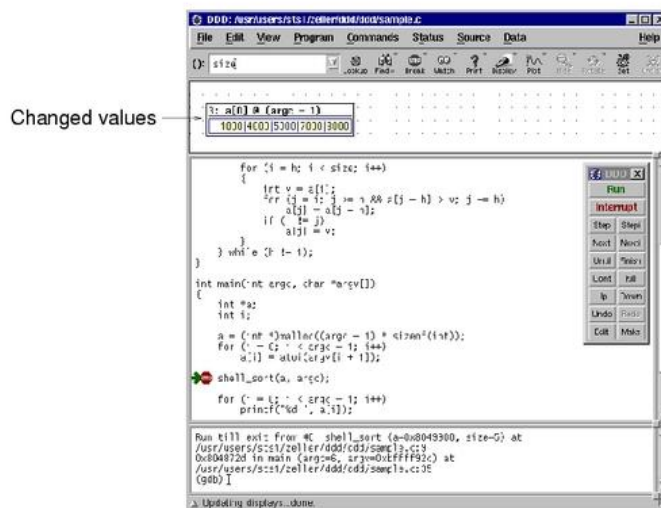
(gdb) finish

Run till exit from #0 shell_sort (a=0x8049878, size=5) at sample.c:9

0x80486ed in main (argc=6, argv=0xbffff918) at sample.c:35

(gdb) _

Success! The a display now contains the correct values 1000, 4000, 5000, 7000, 8000.



Changed Values after Setting

You can verify that these values are actually printed to standard output by further executing the program. Click on Cont to continue execution.

(gdb) cont

1000 4000 5000 7000 8000

Program exited normally.

(gdb) _

The message Program exited normally. is from GDB; it indicates that the sample program has finished executing.

Having found the problem cause, you can now fix the source code. Click on Edit to edit sample.c, and change the line

shell_sort(a, argc);

to the correct invocation

shell_sort(a, argc - 1);

You can now recompile sample

\$ gcc -g -o sample sample.c

\$ _

and verify (via Program => Run Again) that sample works fine now.

(gdb) run

`sample' has changed; re-reading symbols.

Reading in symbols...done.

Starting program: sample 8000 7000 5000 1000 4000

1000 4000 5000 7000 8000

Program exited normally.

(gdb) _

All is done; the program works fine now. You can end this DDD session with Program => Exit or Ctrl+Q.

Part 3

Advanced topics

Useful utilities

1. apt-get command examples

To install any program or package using apt-get (use TAB to auto-complete the name of the program/package):

```
# sudo apt-get install program_or_package_name  
3-22.0.1.el5.i386.rpm
```

To uninstall/remove (use TAB to auto-complete the name of the program/package):

```
# sudo apt-get install program_or_package_name
```

2. tar command examples

Create a new tar archive.

```
$ tar cvf archive_name.tar dirname/
```

Extract from an existing tar archive.

```
$ tar xvf archive_name.tar
```

View an existing tar archive.

```
$ tar tvf archive_name.tar
```

3. gzip command examples

To create a *.gz compressed file:

```
$ gzip test.txt
```

To uncompress a *.gz file:

```
$ gunzip test.txt.gz
```

Display compression ratio of the compressed file using gzip -l

```
$ gzip -l *.gz
```

compressed	uncompressed	ratio	uncompressed_name
23709	97975	75.8%	asp-patch-rpms.txt

4. bzip2 command examples

To create a *.bz2 compressed file:

```
$ bzip2 test.txt
```

To uncompress a *.bz2 file:

```
bzip2 -d test.txt.bz2
```

5. unzip command examples

To extract a *.zip compressed file:

```
$ unzip test.zip
```

View the contents of *.zip file (Without unzipping it):

```
$ unzip -l jasper.zip
Archive: jasper.zip
 Length  Date   Time    Name
-----  -
 40995  11-30-98 23:50  META-INF/MANIFEST.MF
 32169  08-25-98 21:07  classes_
 15964  08-25-98 21:07  classes_names
 10542  08-25-98 21:07  classes_ncomp
```

6. grep command examples

Search for a given string in a file (case in-sensitive search).

```
$ grep -i "some_string" demo_file
```

Print the matched line, along with the 3 lines after it.

```
$ grep -A 3 -i " some_string " demo_file
```

Search for a given string in all files recursively

```
$ grep -r " some_string " *
```

7. find command examples

Find files using file-name (case in-sensitive find)

```
# find -iname "MyCProgram.c"
```

Execute commands on files found by the find command

```
$ find -iname "*.c" -exec ls -al {} \;
```

Find all empty files in home directory

```
# find ~ -empty
```

8. sort command examples

Sort a file in ascending order

```
$ sort names.txt
```

Sort a file in descending order

```
$ sort -r names.txt
```

Sort passwd file by 3rd field.

```
$ sort -t: -k 3n /etc/passwd | more
```


9. ls command examples

Display filesize in human readable format (e.g. KB, MB etc.,)

```
$ ls -lh
-rw-r----- 1 user user 8.9M Jun 12 15:27 arch-linux.txt.gz
```

Order Files Based on Last Modified Time (In Reverse Order) Using ls -ltr

```
$ ls -ltr
```

Visual Classification of Files With Special Characters Using ls -F

```
$ ls -F
```

10. pwd command

pwd stands for Print working Directory. What else can be said about the good old pwd who has been printing the current directory name for ages.

11. cd command examples

Use “cd -” to toggle between the last two directories

Use “shopt -s cdspell” to automatically correct mistyped directory names on cd

12. mkdir command examples

Following example creates a directory called temp under your home directory.

```
$ mkdir ~/temp
```

Create nested directories using one mkdir command. If any of these directories exist already, it will not display any error. If any of these directories doesn't exist, it will create them.

```
$ mkdir -p dir1/dir2/dir3/dir4/
```

13. rm command examples

Get confirmation before removing the file.

```
$ rm -i filename.txt
```

It is very useful while giving shell metacharacters in the file name argument. Print the filename and get confirmation before removing the file.

```
$ rm -i file*
```

Following example recursively removes all files and directories under the example directory. This also removes the example directory itself.

```
$ rm -r example
```

14. cp command examples

Copy file1 to file2 preserving the mode, ownership and timestamp.

```
$ cp -p file1 file2
```

Copy file1 to file2. if file2 exists prompt for confirmation before overwriting it.

```
$ cp -i file1 file2
```

15. mv command examples

Rename file1 to file2. if file2 exists prompt for confirmation before overwriting it.

```
$ mv -i file1 file2
```

Note: mv -f is just the opposite, which will overwrite file2 without prompting.
mv -v will print what is happening during file rename, which is useful while specifying shell metacharacters in the file name argument.

```
$ mv -v file1 file2
```

16. tail command examples

Print the last 10 lines of a file by default.

```
$ tail filename.txt
```

Print N number of lines from the file named filename.txt

```
$ tail -n N filename.txt
```

View the content of the file in real time using tail -f. This is useful to view the log files, that keeps growing. The command can be terminated using CTRL-C.

```
$ tail -f log-file
```

17. less command examples

less is very efficient while viewing huge log files, as it doesn't need to load the full file while opening.

```
$ less huge-log-file.log
```

Once you open a file using less command, following two keys are very helpful.

CTRL+F – forward one window

CTRL+B – backward one window

Forward/backward search:

/string_to_search ENTER

/ENTER – repeats the search again

?string_to_search ENTER

?ENTER – repeats the search again

18. cat command examples

You can view multiple files at the same time. Following example prints the content of file1 followed by file2 to stdout.

```
$ cat file1 file2
```

While displaying the file, following cat -n command will prepend the line number to each line of the output.

```
$ cat -n /etc/logrotate.conf
```

```
1      /var/log/btmp {
2      missingok
3      }
```

19. chmod command examples

chmod command is used to change the permissions for a file or directory.

Give full access to user and group (i.e read, write and execute) on a specific file.

```
$ chmod ug+rx file.txt
```

Revoke all access for the group (i.e read, write and execute) on a specific file.

```
$ chmod g-rwx file.txt
```

Apply the file permissions recursively to all the files in the sub-directories.

```
$ chmod -R ug+rx file.txt
```

20. chown command examples

chown command is used to change the owner and group of a file. \

To change owner to user_name and group to group_name on a file. i.e Change both owner and group at the same time.

```
$ chown user_name: group_name file.txt
```

Use -R to change the ownership recursively.

```
$ chown -R user_name: group_name /home/oracle
```

21. ps command examples

ps command is used to display information about the processes that are running in the system.

While there are lot of arguments that could be passed to a ps command, following are some of the common ones.

To view current running processes.

```
$ ps -ef | more
```

To view current running processes in a tree structure. H option stands for process hierarchy.

```
$ ps -efH | more
```

22. top command examples

top command displays the top processes in the system (by default sorted by cpu usage). To sort top output by any column, Press O (upper-case O) , which will display all the possible columns that you can sort by as shown below.

Current Sort Field: P for window 1:Def

Select sort field via field letter, type any other key to return

a: PID	= Process Id	v: nDRT	= Dirty Pages count
d: UID	= User Id	y: WCHAN	= Sleeping in Function
e: USER	= User Name	z: Flags	= Task Flags
.....			

To displays only the processes that belong to a particular user use -u option. The following will show only the top processes that belongs to user named user_name:

```
$ top -u user_name
```

23. jobs command example

Use jobs command to see the list of a processes running from the current terminal.

```
$ jobs
[1]+  Running          eclipse &
[2]-  Running          emacs &
[3]+  Running          ddd &
```

24. fg/bg command examples

Use fg command to move the eclipse process to the foreground.

```
$ fg %1
eclipse
```

Note that the shell will not display the shell prompt. To return the eclipse to the background use CTRL-Z to stop the execution of the process in the foreground. The shell prompt will return and the the indication that eclipse was stopped will be displayed.

```
^Z
[1]+ Stopped   eclipse
```

Then move the eclipse to the background:

```
$ bg %1
[1]+ eclipse &
$
```

Note that the shell prompt will return when eclipse continues in the background.

23. kill command examples

Use kill command to terminate a process. First get the process id using ps -ef command, then use kill -9 to kill the running Linux process as shown below. You can also use killall, pkill, xkill to terminate a unix process.

```
$ ps -ef | grep vim
ramesh  7243  7222  9 22:43 pts/2    00:00:00 vim

$ kill -9 7243
```

Another example backs to the fg/bg command examples. One can ungracefully kill eclipse by

```
$ kill %1
```


25. free command examples

This command is used to display the free, used, swap memory available in the system. Typical free command output. The output is displayed in bytes.

```
$ free
      total        used        free      shared    buffers     cached
Mem:   3566408    1580220    1986188         0     203988     902960
-/+ buffers/cache:    473272    3093136
Swap:   4000176         0     4000176
```

If you want to quickly check how many GB of RAM your system has use the -g option. -b option displays in bytes, -k in kilo bytes, -m in mega bytes.

```
$ free -g
      total        used        free      shared    buffers     cached
Mem:         3          1          1         0         0         0
-/+ buffers/cache:         0          2
Swap:         3          0          3
```

If you want to see a total memory (including the swap), use the -t switch, which will display a total line as shown below.

```
$ free -t
      total        used        free      shared    buffers     cached
Mem:   3566408    1592148    1974260         0     204260     912556
-/+ buffers/cache:    475332    3091076
Swap:   4000176         0     4000176
Total:  7566584    1592148
```

26. df command examples

Displays the file system disk space usage. By default df -k displays output in bytes.

```
$ df -k
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/sda1        29530400  3233104 24797232 12% /
/dev/sda2       120367992 50171596 64082060 44% /home
```

df -h displays output in human readable form. i.e size will be displayed in GB's.

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        29G  3.1G   24G  12% /
/dev/sda2       115G   48G   62G  44% /home
```

Use -T option to display what type of file system.

```
$ df -T
Filesystem Type 1K-blocks    Used Available Use% Mounted on
/dev/sda1 ext4 29530400 3233120 24797216 12% /
/dev/sda2 ext4 120367992 50171596 64082060 44% /home
```

27. uname command examples

Uname command displays important information about the system such as — Kernel name, Host name, Kernel release number, Processor type, etc.,

Sample uname output from a Ubuntu laptop is shown below.

```
$ uname -a
Linux ubuntu 3.2.0-23-generic-pae #36-Ubuntu SMP Tue Apr 10 22:19:09 UTC 2012 i686 i686
i386 GNU/Linux
```

28. whereis command examples

When you want to find out where a specific Unix command exists (for example, where does ls command exists?), you can execute the following command.

```
$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz /usr/share/man/man1p/ls.1p.gz
```

When you want to search an executable from a path other than the whereis default path, you can use -B option and give path as argument to it. This searches for the executable lsmk in the /tmp directory, and displays it, if it is available.

```
$ whereis -u -B /tmp -f lsmk
lsmk: /tmp/lsmk
```

29. whatis command examples

Whatis command displays a single line description about a command.

```
$ whatis ls
ls                (1) - list directory contents

$ whatis ifconfig
ifconfig (8)      - configure a network interface
```

30. locate command examples

Using locate command you can quickly search for the location of a specific file (or group of files). Locate command uses the database created by updatedb.

The example below shows all files in the system that contains the word crontab in it.

```
$ locate crontab
/etc/anacrontab
/etc/crontab
/usr/bin/crontab
/usr/share/doc/cron/examples/crontab2english.pl.gz
/usr/share/man/man1/crontab.1.gz
/usr/share/man/man5/anacrontab.5.gz
/usr/share/man/man5/crontab.5.gz
/usr/share/vim/vim72/syntax/crontab.vim
```

31. su command examples

Switch to a different user account using su command. Super user can switch to any other user without entering their password.

```
$ su - USERNAME
```

Execute a single command from a different account name. In the following example, john can execute the ls command as raj username. Once the command is executed, it will come back to john's account.

```
$ su - raj -c 'ls'

$
```

Login to a specified user account, and execute the specified shell instead of the default shell.

```
$ su -s 'SHELLNAME' USERNAME
```

32. sed command examples

When you copy a DOS file to Unix, you could find `\r\n` in the end of each line. This example converts the DOS file format to Unix file format using sed command.

```
$sed 's/$//' filename
```

Print file content in reverse order

```
$ sed -n '1!G;h;$p' thegeekstuff.txt
```

Add line number for all non-empty-lines in a file

```
$ sed '/./=' thegeekstuff.txt | sed 'N; s/\n/ /'
```

33. awk command examples

Remove duplicate lines using awk

```
$ awk '!($0 in array) { array[$0]; print }' temp
```

Print all lines from `/etc/passwd` that has the same uid and gid

```
$awk -F ':' '$3==$4' passwd.txt
```

Print only specific field from a file.

```
$ awk '{print $2,$5;}' employee.txt
```

AWK

Awk is a programming language which allows easy manipulation of structured data and the generation of formatted reports. Awk stands for the names of its authors “Aho, Weinberger, and Kernighan”

The Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then perform associated actions.

Some of the key features of Awk are:

- Awk views a text file as records and fields.
- Like common programming language, Awk has variables, conditionals and loops
- Awk has arithmetic and string operators.
- Awk can generate formatted reports
- Awk reads from a file or from its standard input, and outputs to its standard output. Awk does not get along with non-text files.

Syntax:

```
awk '/search pattern1/ {Actions}  
    /search pattern2/ {Actions}' file
```

In the above awk syntax:

- search pattern is a regular expression.
- Actions – statement(s) to be performed.
- several patterns and actions are possible in Awk.
- file – Input file.
- Single quotes around program is to avoid shell not to interpret any of its special characters.

Awk Working Methodology

1. Awk reads the input files one line at a time.
2. For each line, it matches with given pattern in the given order, if matches performs the corresponding action.
3. If no pattern matches, no action will be performed.
4. In the above syntax, either search pattern or action are optional, But not both.
5. If the search pattern is not given, then Awk performs the given actions for each line of the input.
6. If the action is not given, print all that lines that matches with the given patterns which is the default action.
7. Empty braces with out any action does nothing. It wont perform default printing operation.
8. Each statement in Actions should be delimited by semicolon.

Let us demonstrate the awk on employee.txt file which has the following content, which will be used in the examples mentioned below.

```
$cat employee.txt
100 Thomas Manager Sales $5,000
200 Jason Developer Technology $5,500
300 Sanjay Sysadmin Technology $7,000
400 Nisha Manager Marketing $9,500
500 Randy DBA Technology $6,000
```

Awk Example 1. Default behavior of Awk

By default Awk prints every line from the file.

```
$ awk '{print;}' employee.txt
100 Thomas Manager Sales $5,000
200 Jason Developer Technology $5,500
300 Sanjay Sysadmin Technology $7,000
400 Nisha Manager Marketing $9,500
500 Randy DBA Technology $6,000
```

In the above example pattern is not given. So the actions are applicable to all the lines. Action print without any argument prints the whole line by default. So it prints all the lines of the file without fail. Actions has to be enclosed within the braces.

Awk Example 2. Print the lines which matches with the pattern.

```
$ awk '/Thomas/  
> /Nisha/' employee.txt  
100 Thomas Manager Sales $5,000  
400 Nisha Manager Marketing $9,500
```

In the above example it prints all the line which matches with the ‘Thomas’ or ‘Nisha’. It has two patterns. Awk accepts any number of patterns, but each set (patterns and its corresponding actions) has to be separated by newline.

Awk Example 3. Print only specific field.

Awk has number of built in variables. For each record i.e line, it splits the record delimited by whitespace character by default and stores it in the \$n variables. If the line has 4 words, it will be stored in \$1, \$2, \$3 and \$4. \$0 represents whole line. NF is a built in variable which represents total number of fields in a record.

```
$ awk '{print $2,$5;}' employee.txt  
Thomas $5,000  
Jason $5,500  
Sanjay $7,000  
Nisha $9,500  
Randy $6,000  
  
$ awk '{print $2,$NF;}' employee.txt  
Thomas $5,000  
Jason $5,500  
Sanjay $7,000  
  
Nisha $9,500  
  
Randy $6,000
```

In the above example \$2 and \$5 represents Name and Salary respectively. We can get the Salary using \$NF also, where \$NF represents last field. In the print statement ‘,’ is a concatenator.

Awk Example 4. Initialization and Final Action

Awk has two important patterns which are specified by the keyword called BEGIN and END.

Syntax:

```
BEGIN { Actions }
{ACTION} # Action for everyline in a file
END { Actions }
```

is for comments in Awk

Actions specified in the BEGIN section will be executed before starts reading the lines from the input. END actions will be performed after completing the reading and processing the lines from the input.

```
$ awk 'BEGIN {print "Name\tDesignation\tDepartment\tSalary";}
> {print $2,"\t",$3,"\t",$4,"\t",$NF;}
> END{print "Report Generated\n-----";
> }' employee.txt
Name      Designation    Department      Salary
Thomas    Manager        Sales           $5,000
Jason     Developer      Technology      $5,500
Sanjay     Sysadmin       Technology      $7,000
Nisha     Manager        Marketing       $9,500
Randy     DBA            Technology      $6,000
Report Generated
-----
```

In the above example, it prints headline and last file for the reports.

Awk Example 5. Find the employees who has employee id greater than 200

```
$ awk '$1 >200' employee.txt
300 Sanjay Sysadmin Technology $7,000
400 Nisha Manager Marketing $9,500
500 Randy DBA Technology $6,000
```

In the above example, first field (\$1) is employee id. So if \$1 is greater than 200, then just do the default print action to print the whole line.

Awk Example 6. Print the list of employees in Technology department

Now department name is available as a fourth field, so need to check if \$4 matches with the string “Technology”, if yes print the line.

```
$ awk '$4 ~/Technology/' employee.txt
200 Jason Developer Technology $5,500
300 Sanjay Sysadmin Technology $7,000
500 Randy DBA Technology $6,000
```

Operator ~ is for comparing with the regular expressions. If it matches the default action i.e print whole line will be performed.

Awk Example 7. Print number of employees in Technology department

The below example, checks if the department is Technology, if it is yes, in the Action, just increment the count variable, which was initialized with zero in the BEGIN section.

```
$ awk 'BEGIN { count=0;}
$4 ~ /Technology/ { count++; }
END { print "Number of employees in Technology Dept =",count;}' employee.txt
Number of employees in Tehcnology Dept = 3
```

Then at the end of the process, just print the value of count which gives you the number of employees in Technology department.

Index

A

altering protections	29
ar	40
arguments	23
automatic variables	47

B

BASH.....	22
-----------	----

C

cat	27
cd	25
chgrp	30
chmod	30
chusr.....	30
command line interpreter	10
compilation	38
cp	27, 28
csh.....	22
Ctrl-A	22
Ctrl-D.....	22
Ctrl-E.....	22
Ctrl-K	22
Ctrl-L.....	23
Ctrl-T	23
Ctrl-U.....	22

D

Debugging	49
dependencies	44
diff.....	27

E

Editors	33
Emacs editor	34, 35
ESC-D.....	22

F

FAT-12	15
file system	25
find.....	27
flags.....	23

G

gcc	38
gcc compiler.....	38

gdb	49
GNU gdb debugger.....	49
grep	27

H

head	27
hidden directories.....	24
hidden files	24
home	25
home directory	23
homedir	23

K

KDE Help Center	21
ksh.....	22

L

less	27
Libraries.....	40
linkage.....	38
Linux programmer's manual	20
ls	25, 28

M

make	43
make utility	43
Make utility	38
makefile.....	43
man	19
mcopv.....	15
mdel	15
mdir.....	15
mformat	15
middle mouse button	20
mkdir	28
more.....	27
mttools	15
mv	27, 28

O

Optimization.....	39
-------------------	----

P

Panel	9
patern rule.....	47
phoney node.....	44
pipes.....	28
preprocessing	38

Protecting files and directories	29
pwd	25

R

ranlib	40
Redirecting Input/Output	28
rm	27
rmdir	28

S

sh	22
shells	22
sort	29
Static libraries	40
switches	23

T

tail	27
target	44
touch	27

U

USB flash memory	13
------------------------	----

V

verb	23
vi	36
Vi editor	36
vim	36

W

wc	29
Where to save files	12
Wildcards	26

X

xman	20
Xterm	10
X-window	10