The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific and
Cultural Organization

IAEA
International Atomic Energy Agency

2273-11

Second Workshop on Open Source and the Internet for Building Global Scientific Communities with Emphasis on Environmental Monitoring and Distributed Instrumentation

28 November - 16 December, 2011

Linux Programming

R. Ijaduola
Acadiate, Toronto
Canada

GNU/Linux: Migrating to Linux

Second Workshop on Open Source and the Internet for Building Global Scientific Communities
(Trieste – Italy)

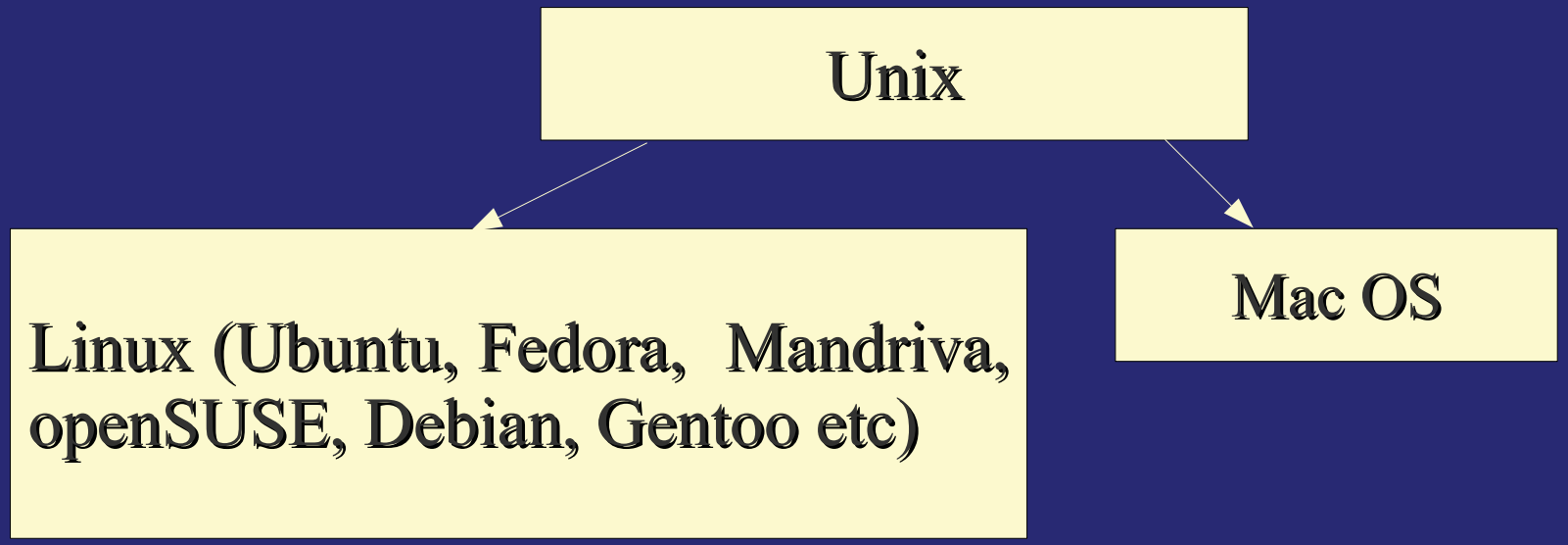
Razaq Ijaduola
28 November - 16 December 2011

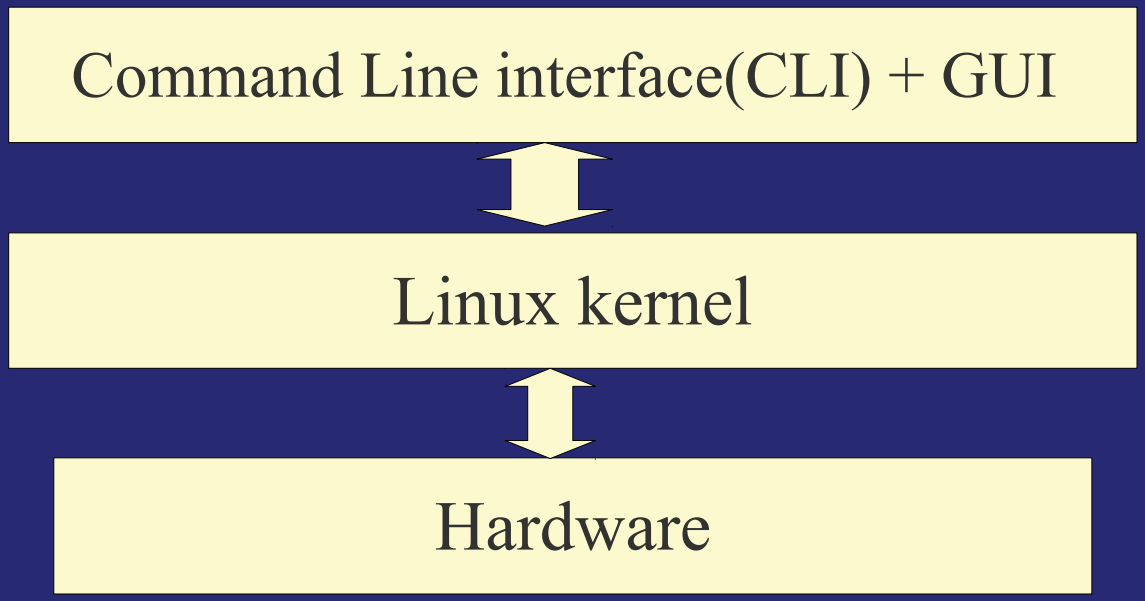
Introduction

- Linux is a 'UNIX'-like operating system
- It was launched in 1991 by Linus Torvalds, a Finnish student.
- Hundreds of people contributed to it and thousands tested and reported bugs.
- It is therefore extremely stable.
- Millions of people, institutes and firms are using it now.
- It is distributed under the 'GNU General License' and it is practically cost-free.
- You can download it at no cost at all.

At the risk of boring those who know Linux already, I will present some of its features and show you a few commands and basic graphical user interface(GUI) functions which hopefully will turn out to be useful in your work during the next weeks (and later!).

Practically all present day operating systems are sort of descendants of UNIX: even MSDOS and Windows have taken features of it. Mac OS is UNIX based.





Linux is the interface between the hardware of the machine and programs running on it.

Features of Linux

Here are some of the features of Linux:

- Files are stored in a hierarchical tree structure of directories(folders) and subdirectories (/ = top level).
  Shorthand for directories:
  - . stands for 'present directory'
  - .. stand for 'parent directory'
  - ~ stands for user's home directory.
- Preference files (hidden files) starts with “.”
  .bashrc, .mozilla/
- Files have access permissions (drwxrwxrwx).
  file type | owner | group | others (everyone else)

```
drwxr-xr-x 34 razaq razaq    4096 2011-11-07 22:31 workspace
-rw-r--r--  1 razaq razaq  110869 2011-03-30 19:44  zendwebservice.pdf
```

Features of Linux

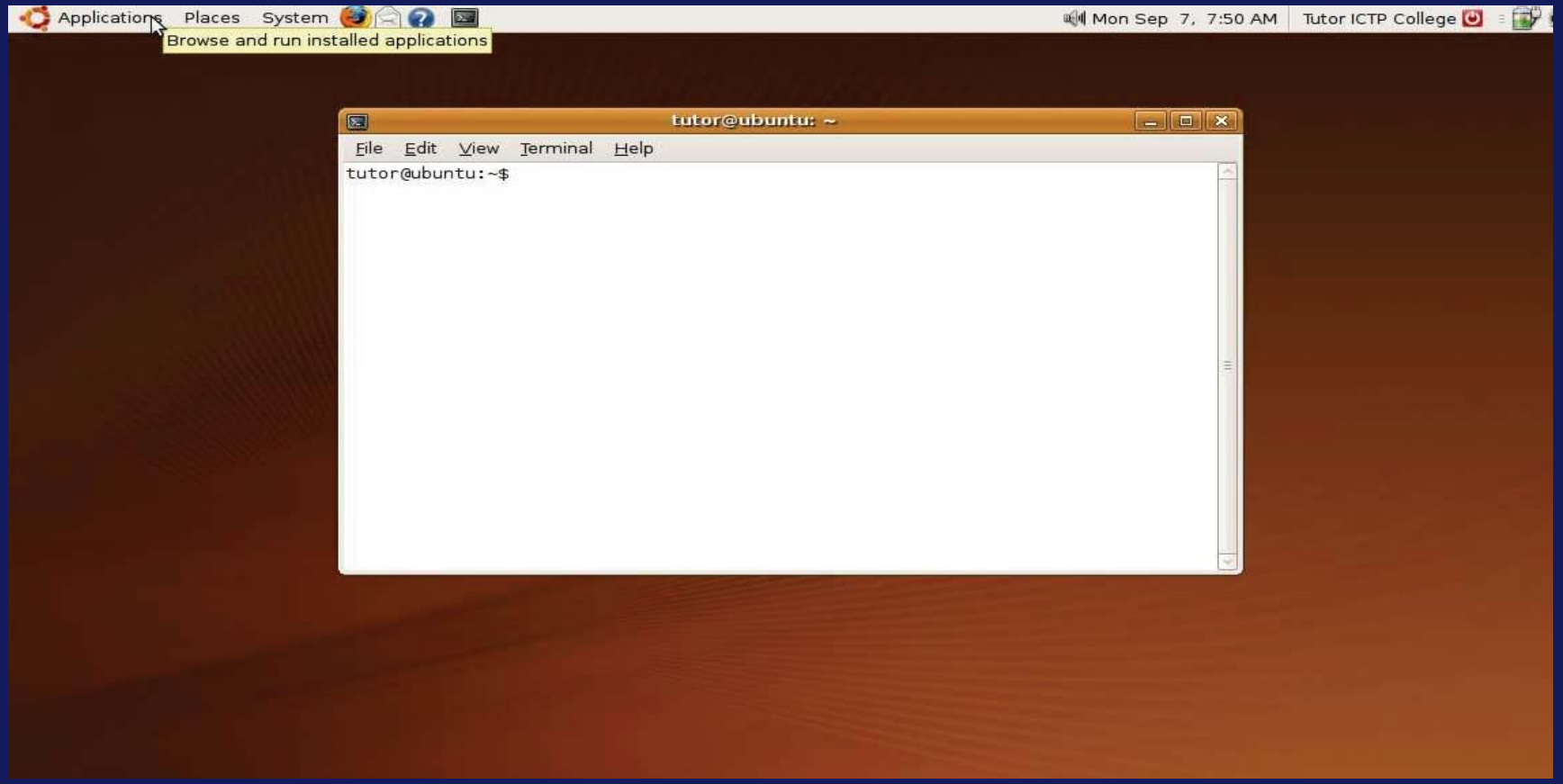
- Files are a collection of bytes. No distinction between file types.
- Filenames are not limited to eight characters, there are no suffixes.
- Linux is a multi-tasking, multi-user, time-sharing system.
- Programs of a user are protected against incursions by other users.
- Memory can be allocated dynamically.
- You must also strongly resist pushing 'reset' or hitting 'Ctrl-Alt-Delete', nowadays journaling is built into the filesystem.

Graphical User Interface

- Windows is familiar to everyone (MS windows, Mac, Linux users).

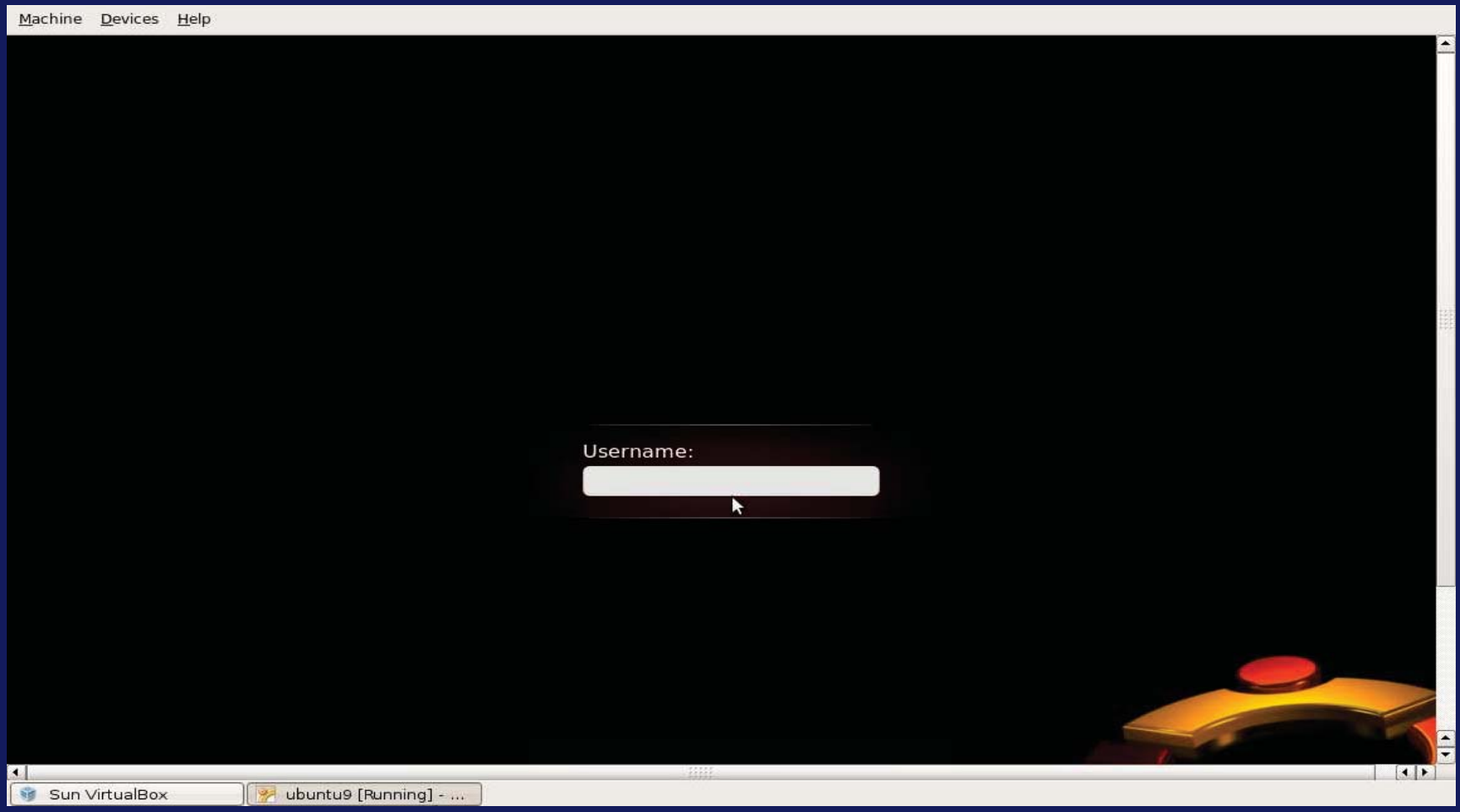
Graphical User Interface

- Command line interface(CLI) is familiar to power users.



GUI - login

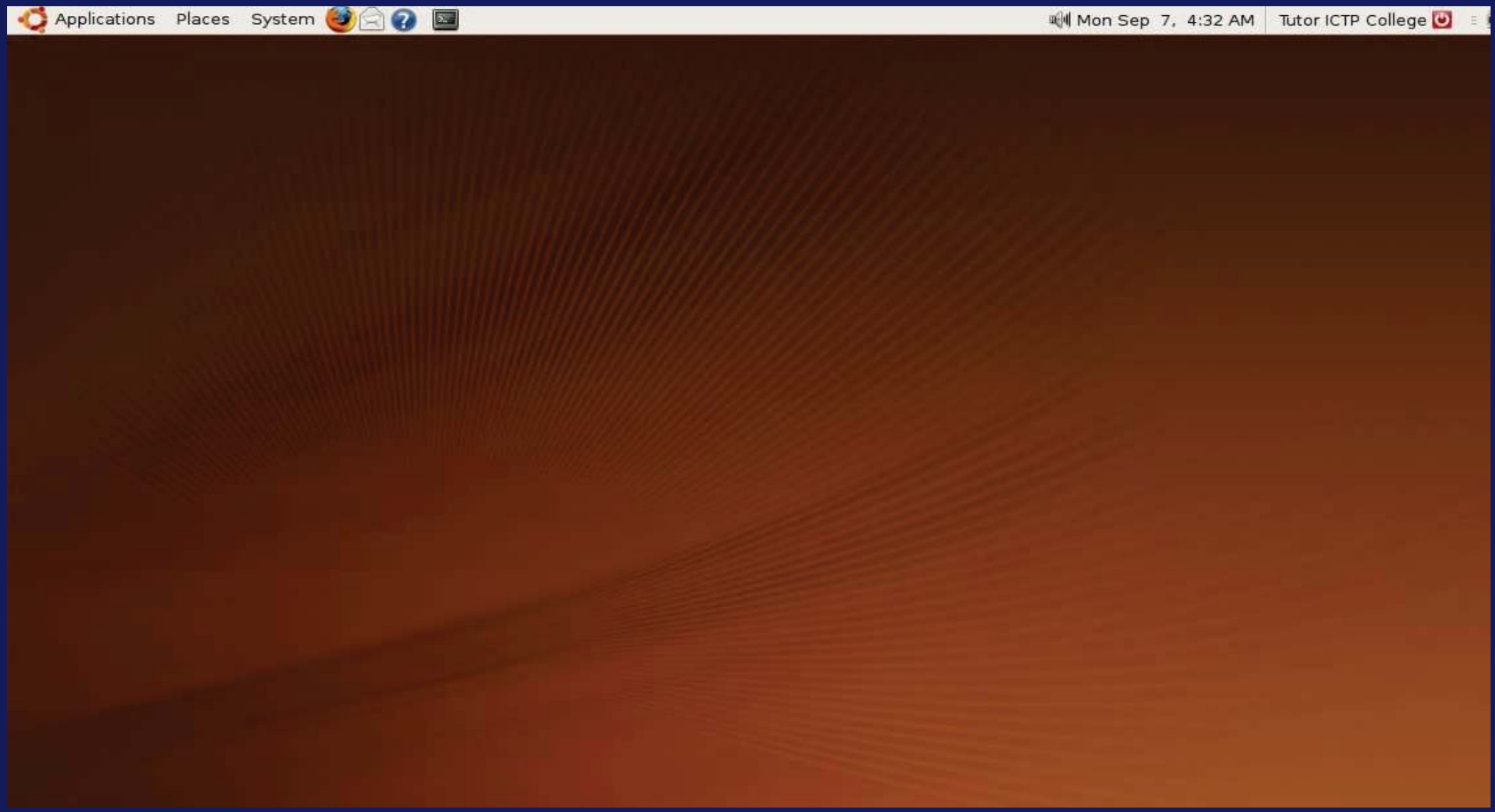
- username



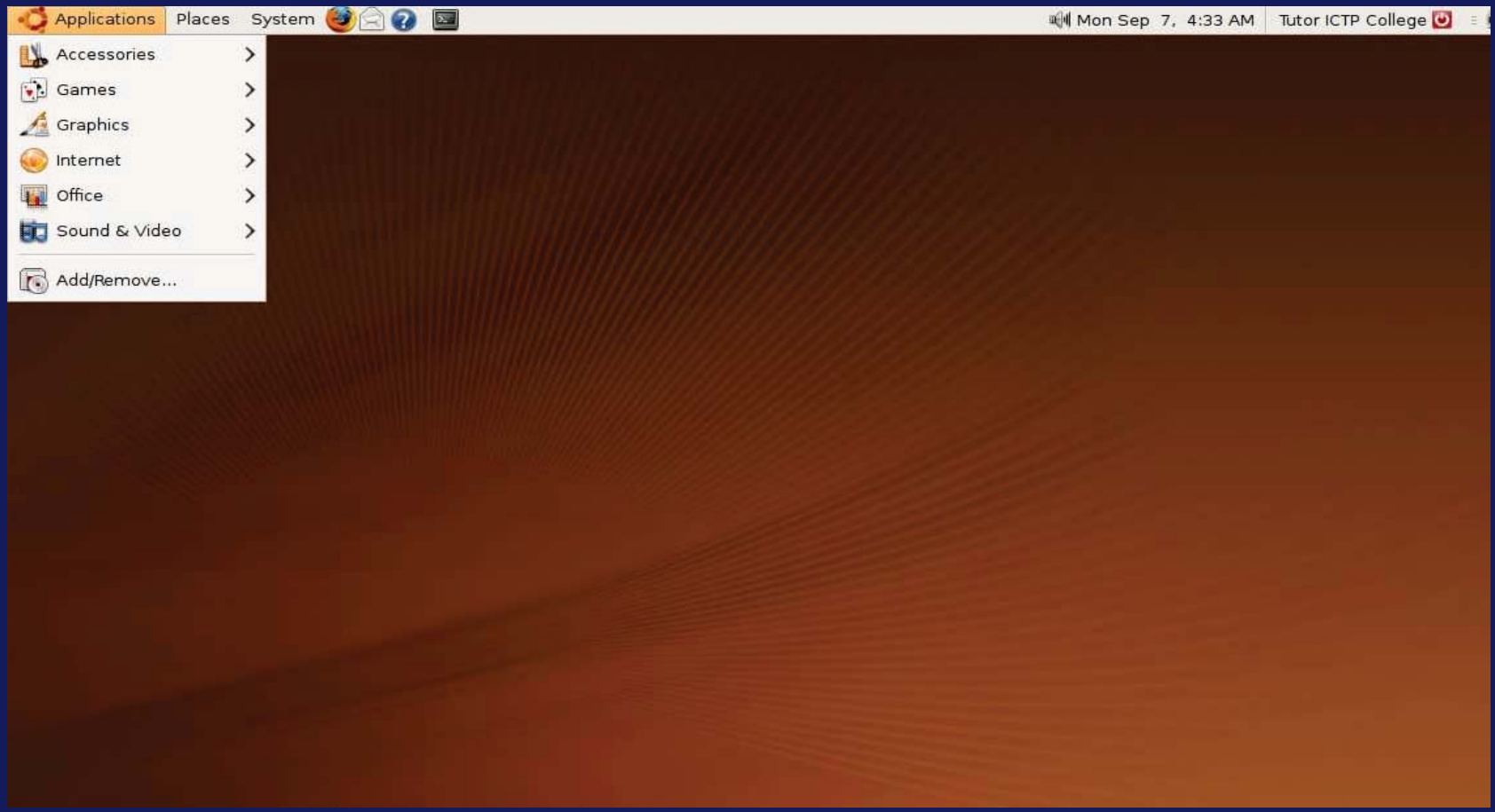
GUI - login

- password

GUI - Desktop

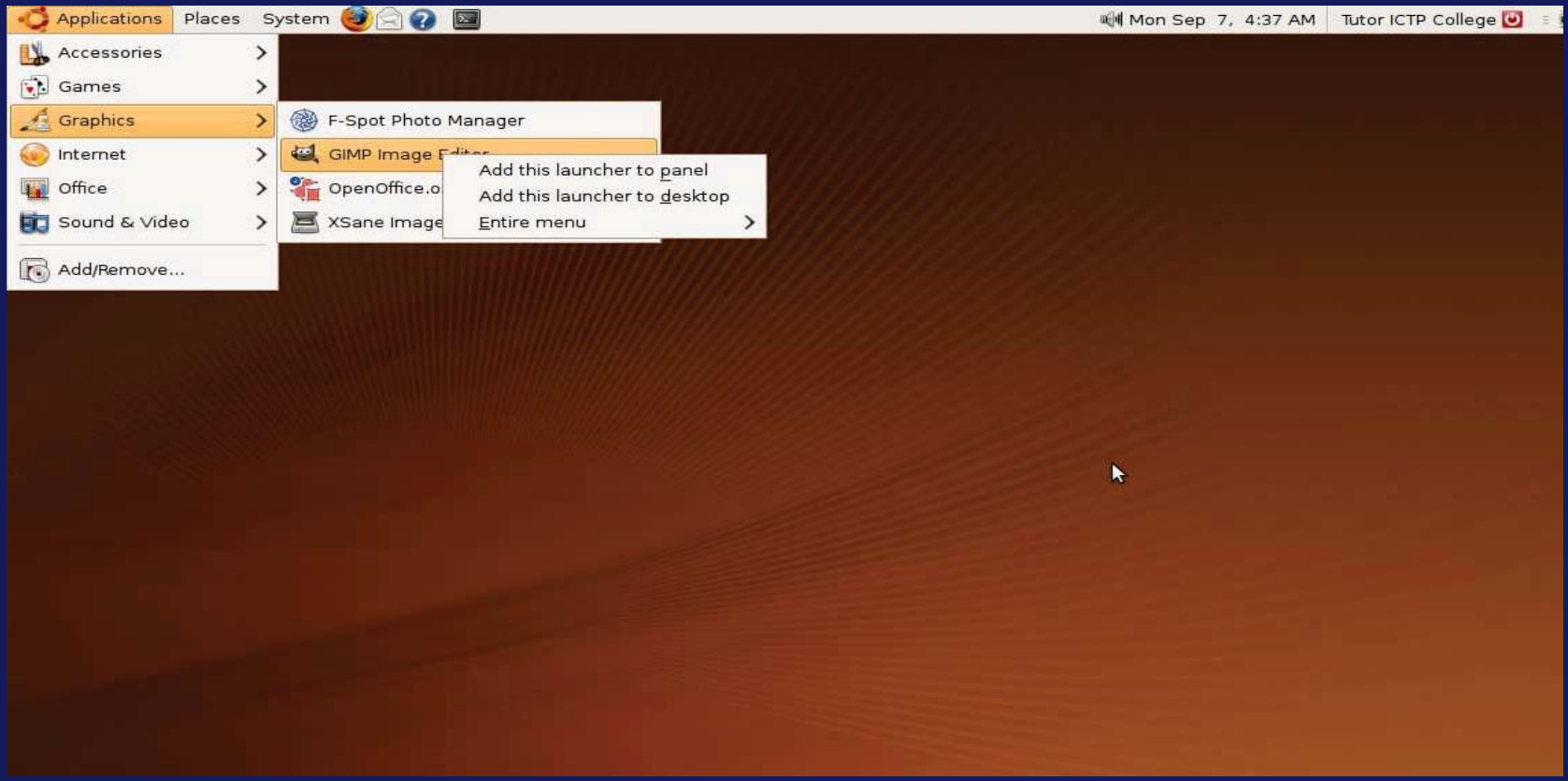

Applications
Places
System
Mon Sep 7, 4:32 AM
Tutor ICTP College

GUI - Applications



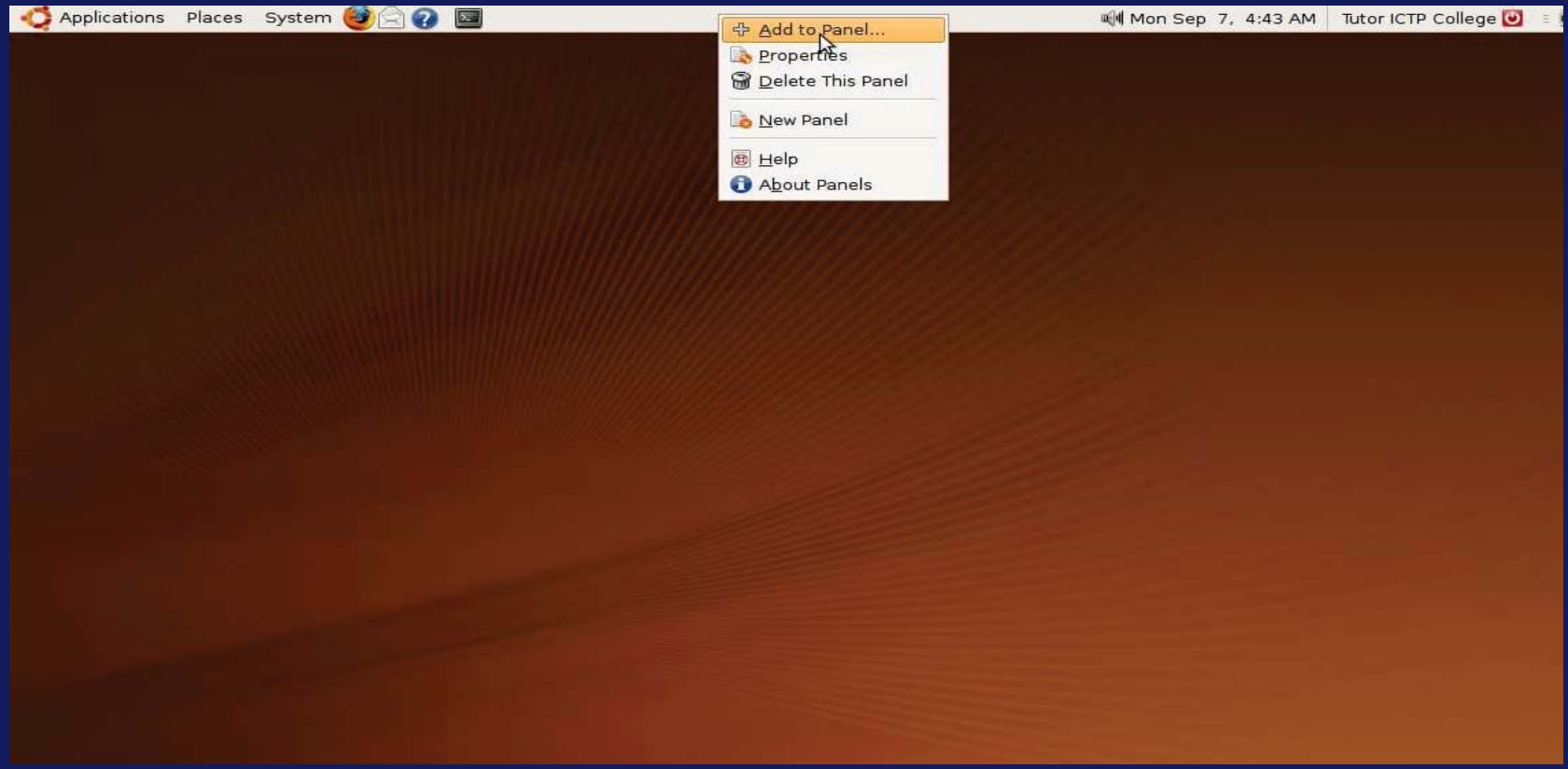
GUI - Applications

- Add application to panel



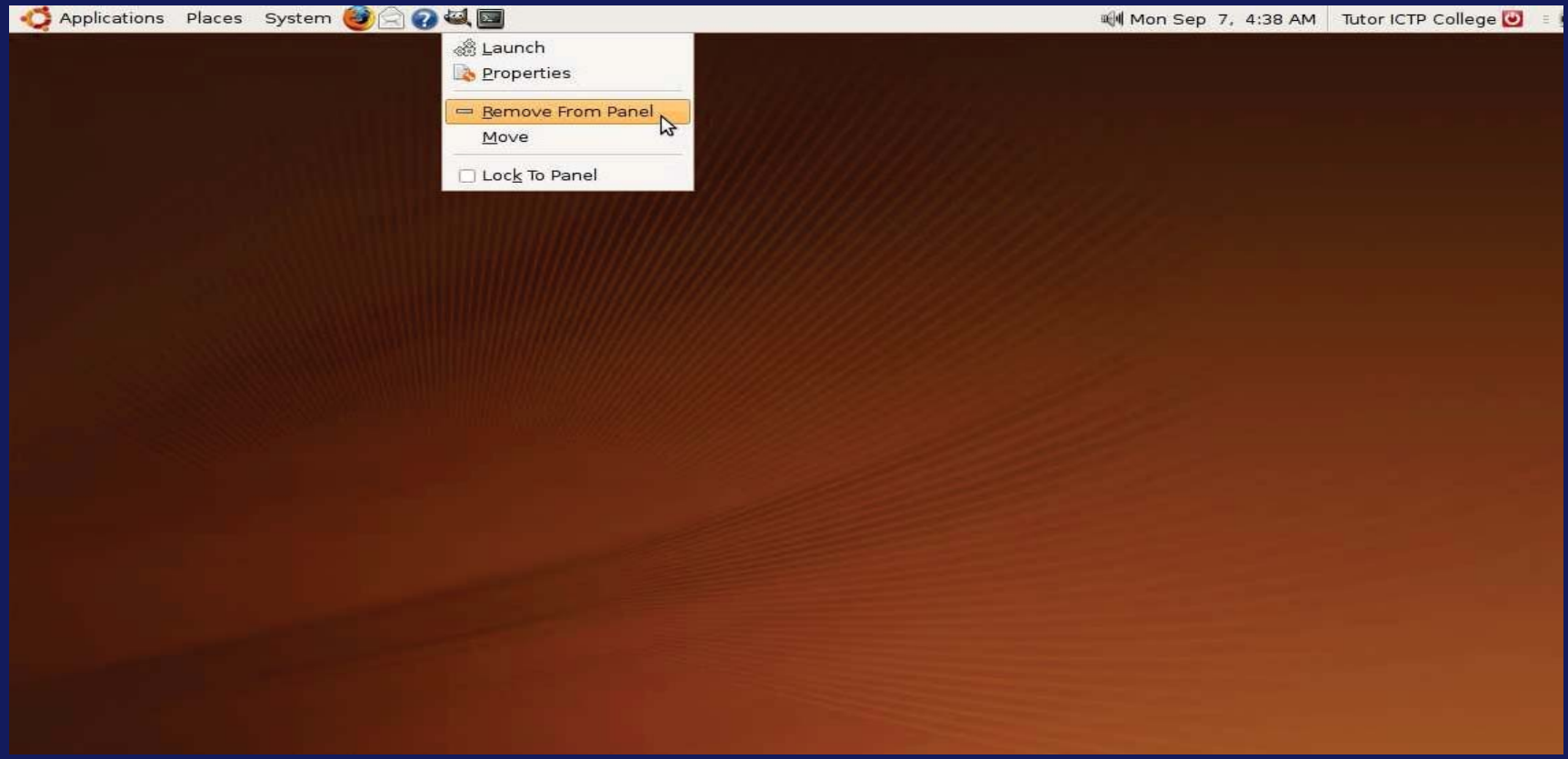
GUI - Applications

- Add application by right clicking the panel

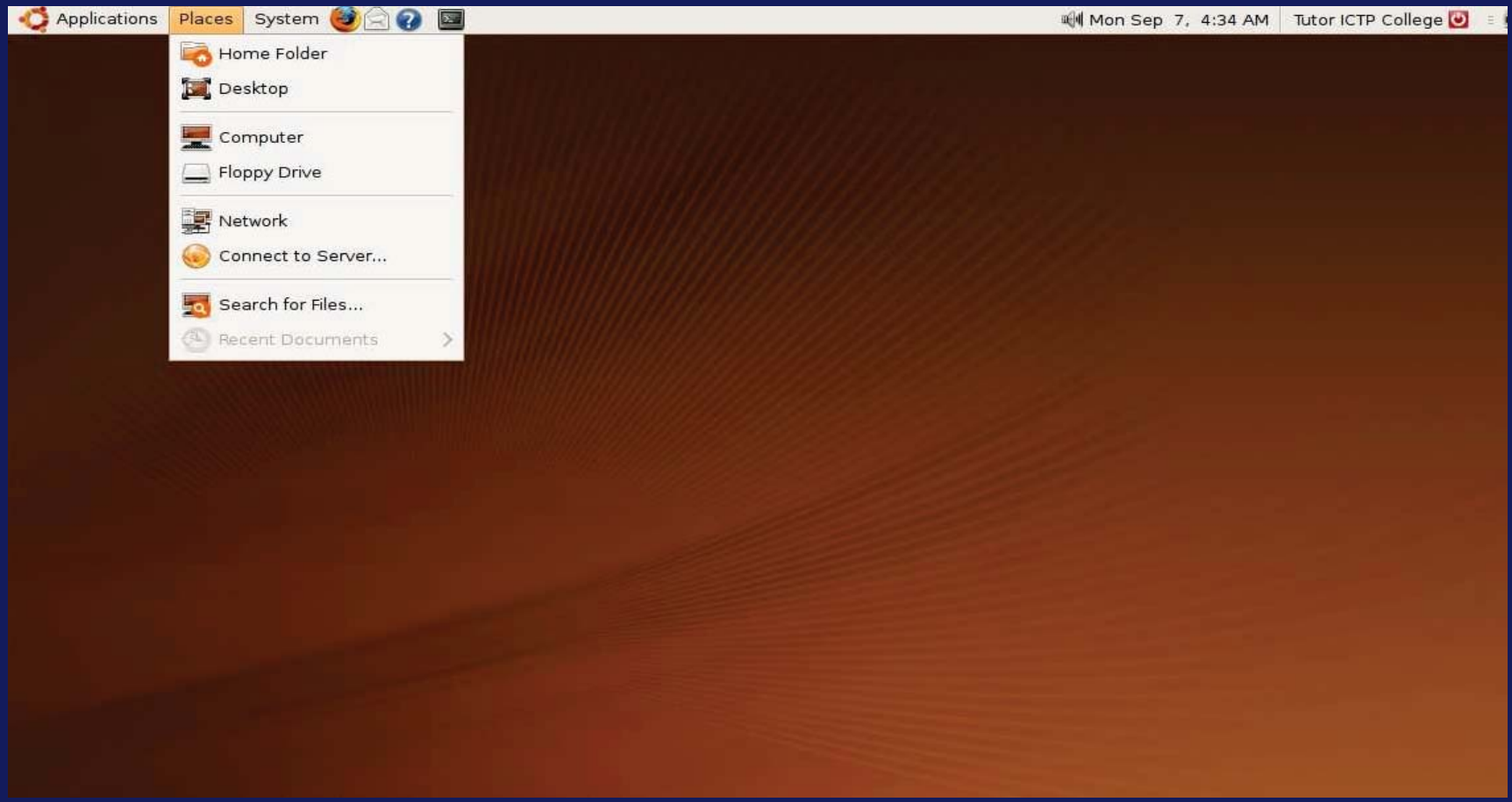


GUI - Applications

- Remove from panel

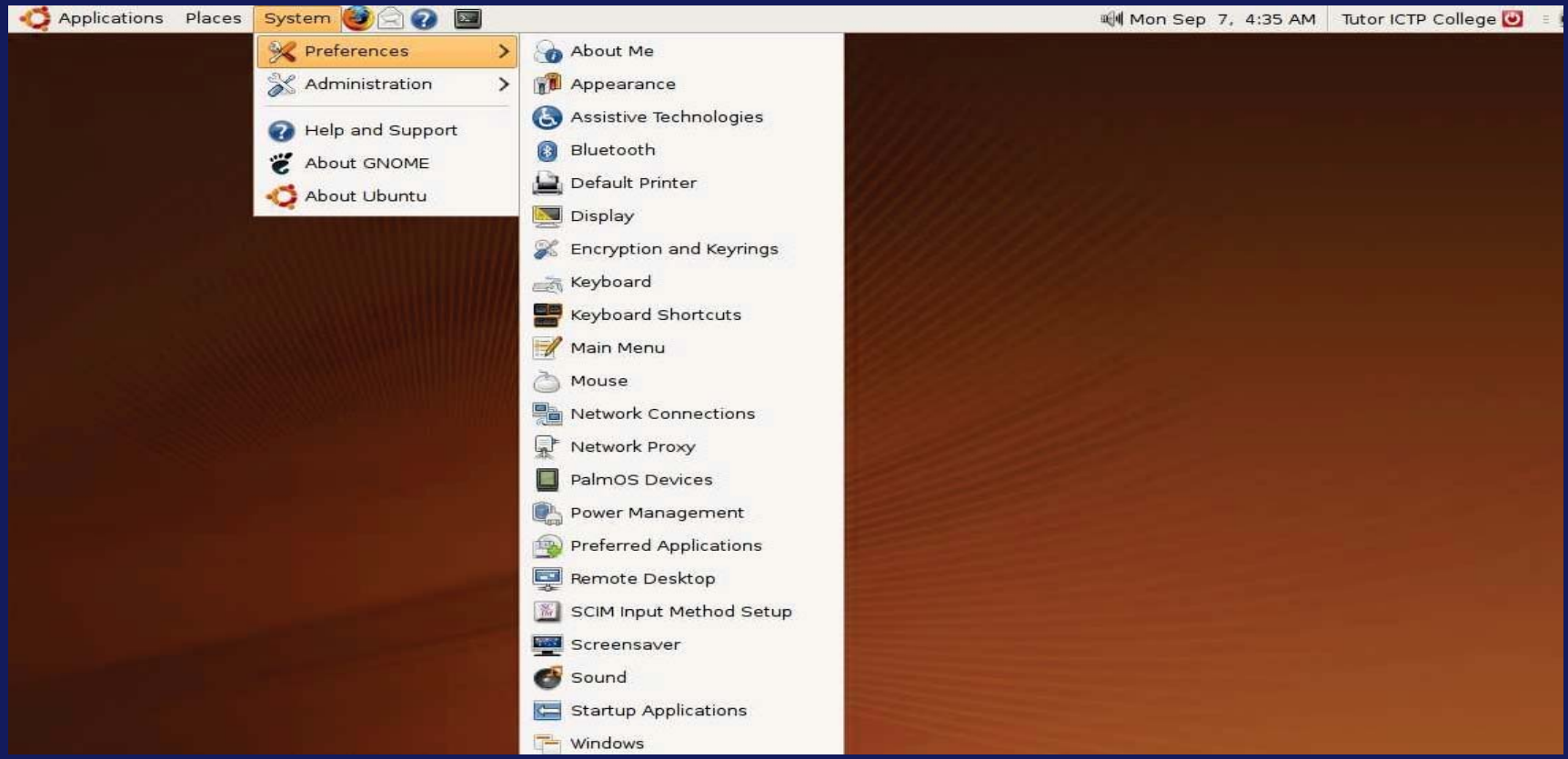


GUI - Places


Applications
Places
System
Mon Sep 7, 4:34 AM
Tutor ICTP College
Home Folder
Desktop
Computer
Floppy Drive
Network
Connect to Server...
Search for Files...
Recent Documents

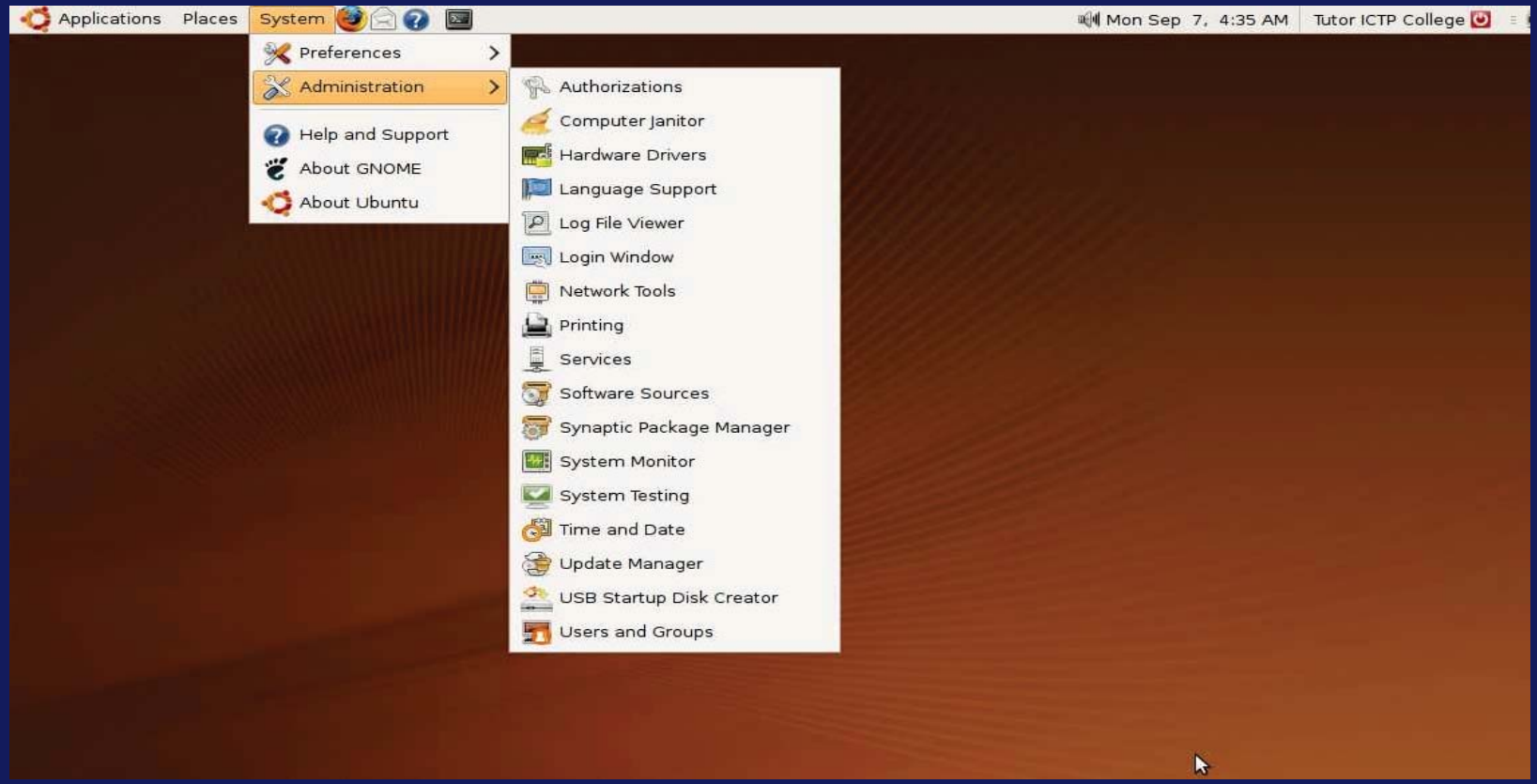
GUI - System

- Preferences



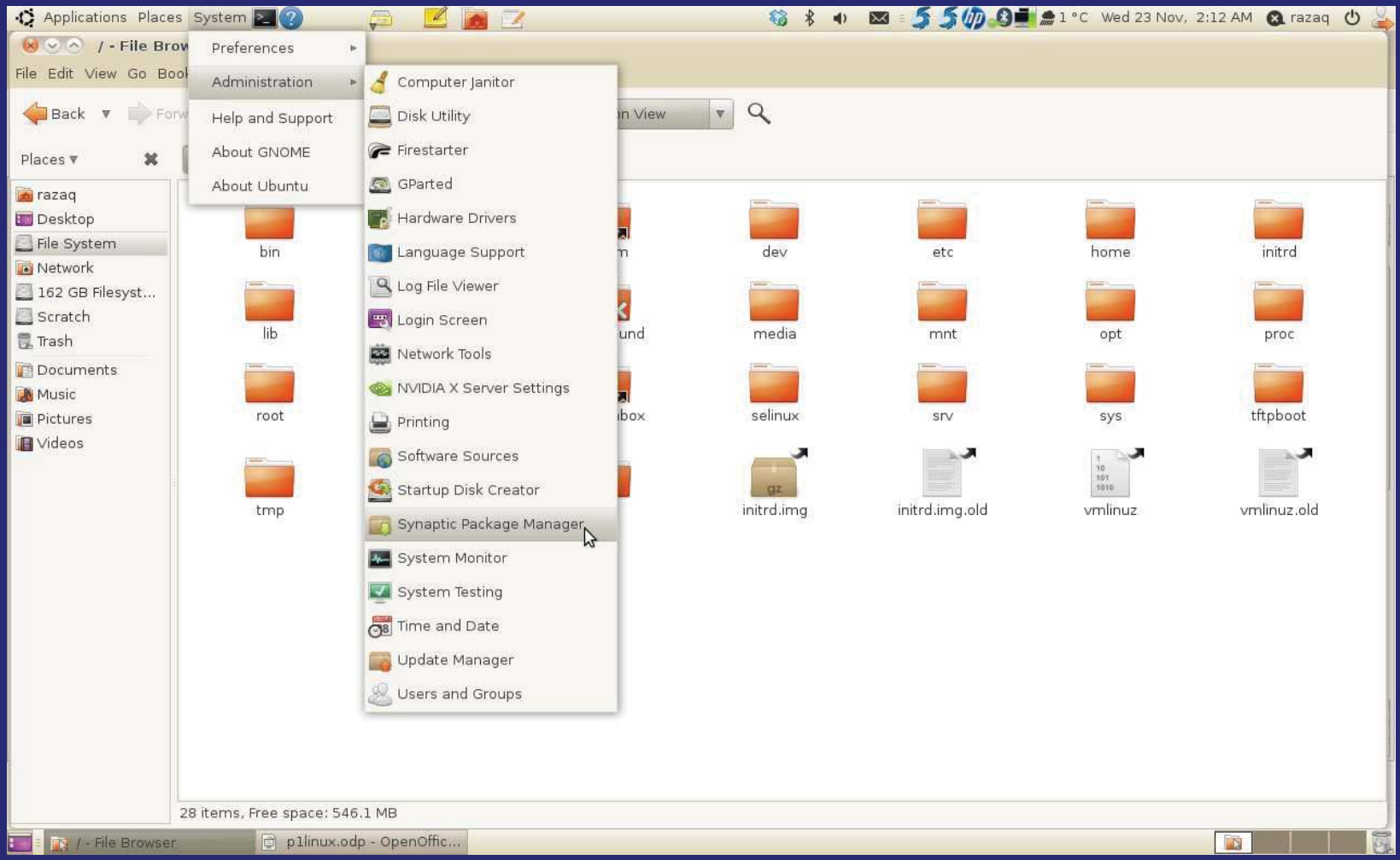
GUI - System

- Administration



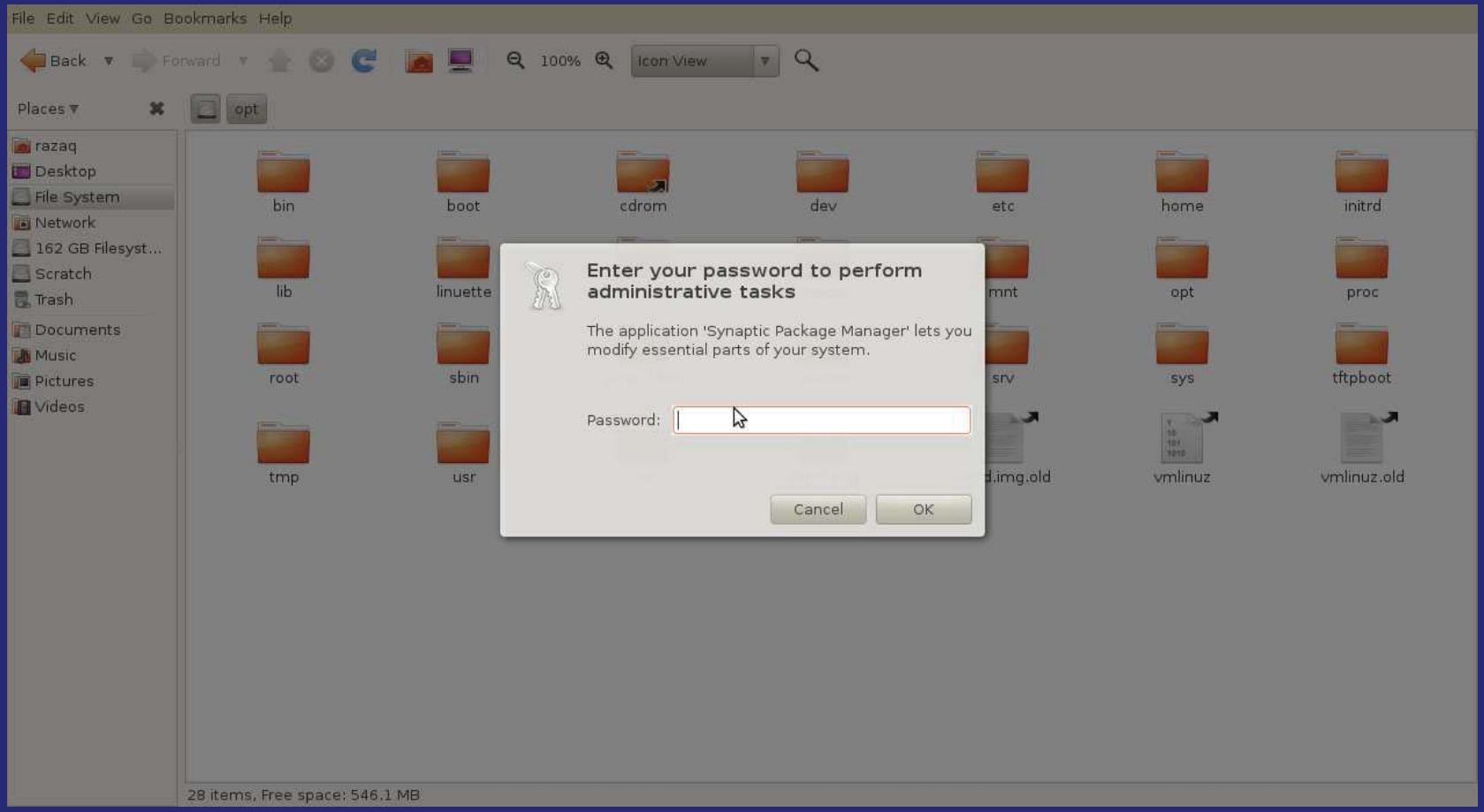
GUI – System Administration

- Synaptic Package Manager



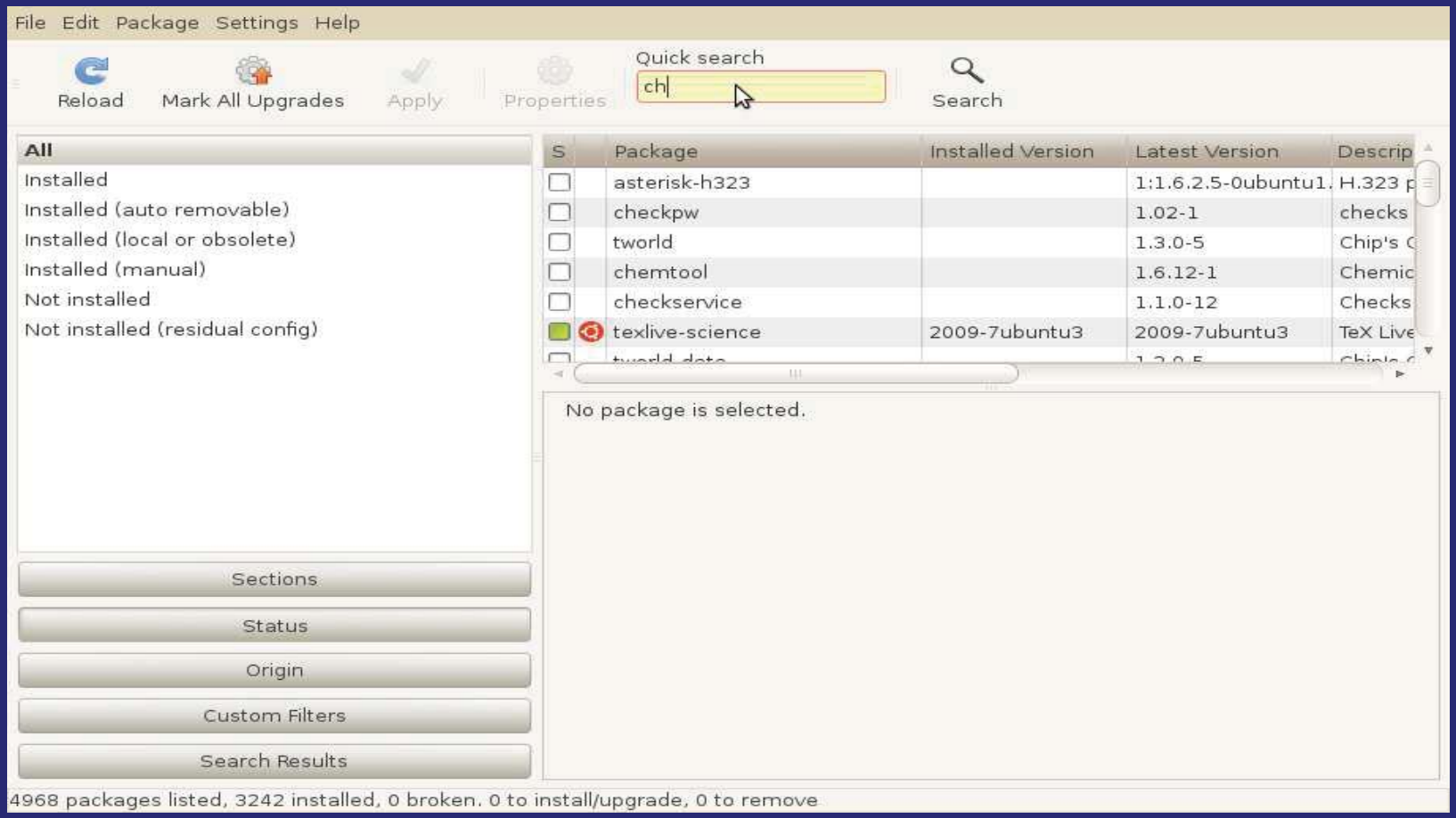
GUI – System Administration

- Synaptic Package Manager



GUI – System Administration

- Synaptic Package Manager

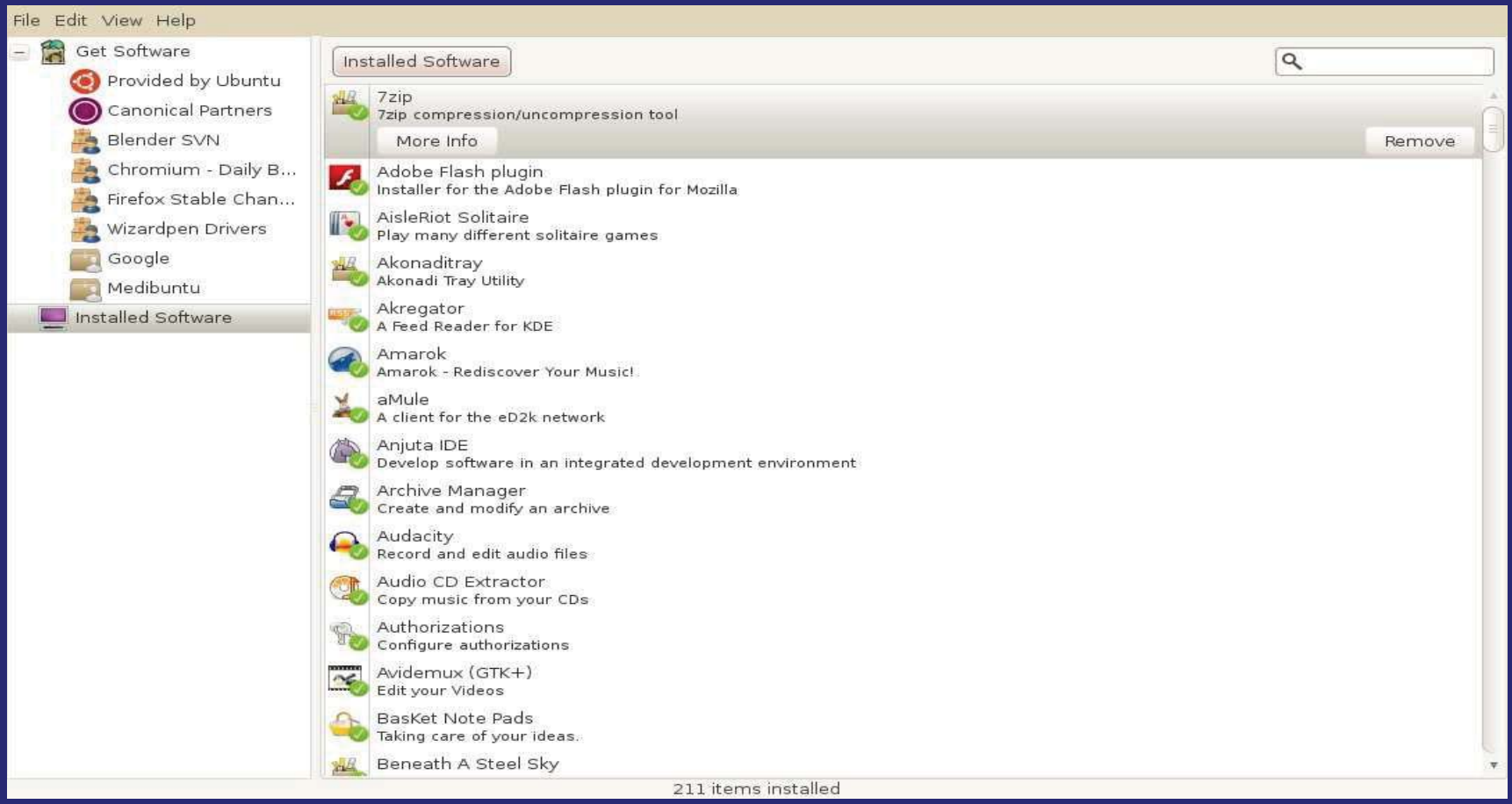


GUI – Applications

- Add/remove OR Ubuntu Software centre

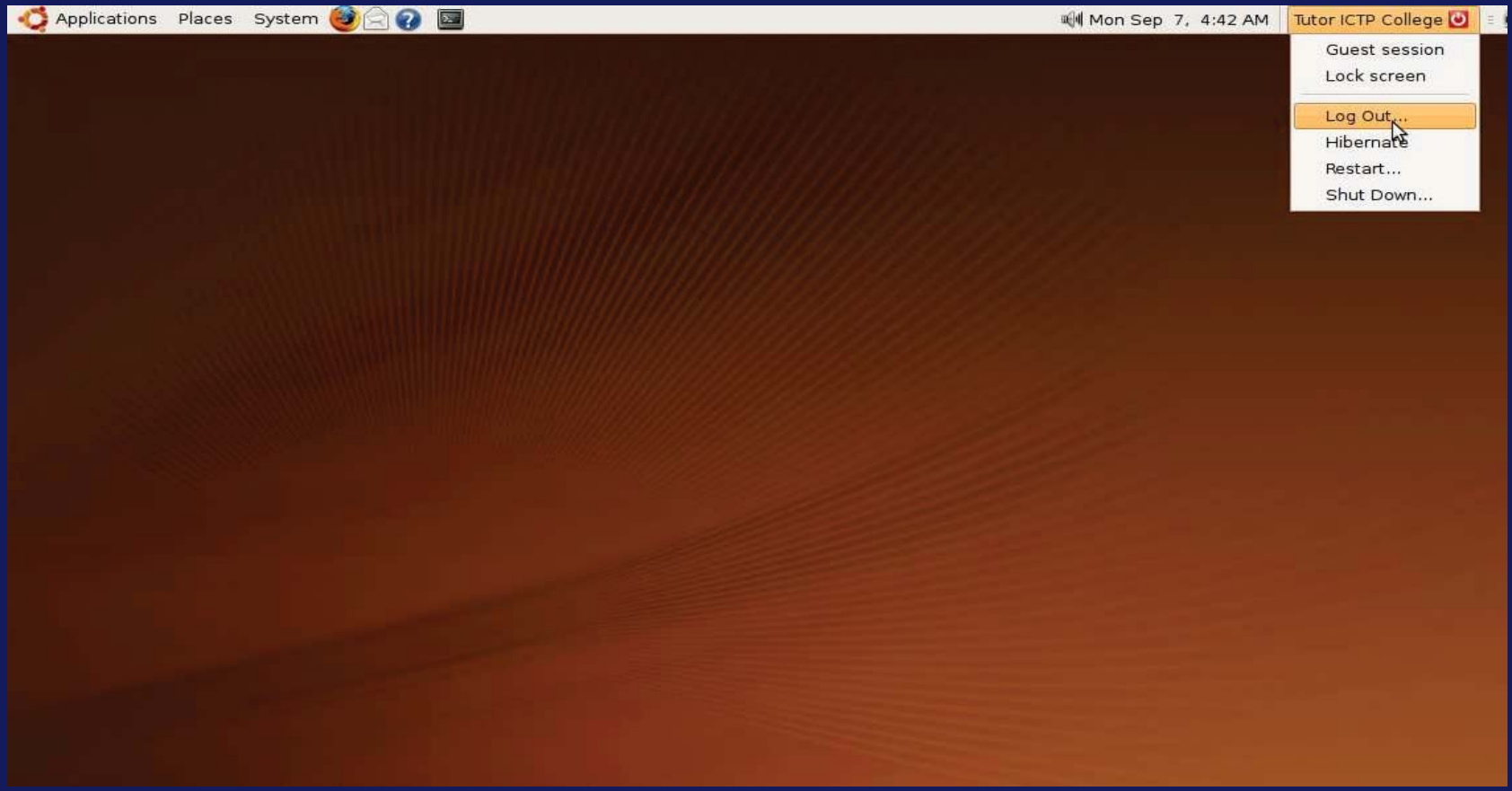
GUI – Applications

- Add/remove OR Ubuntu Software centre



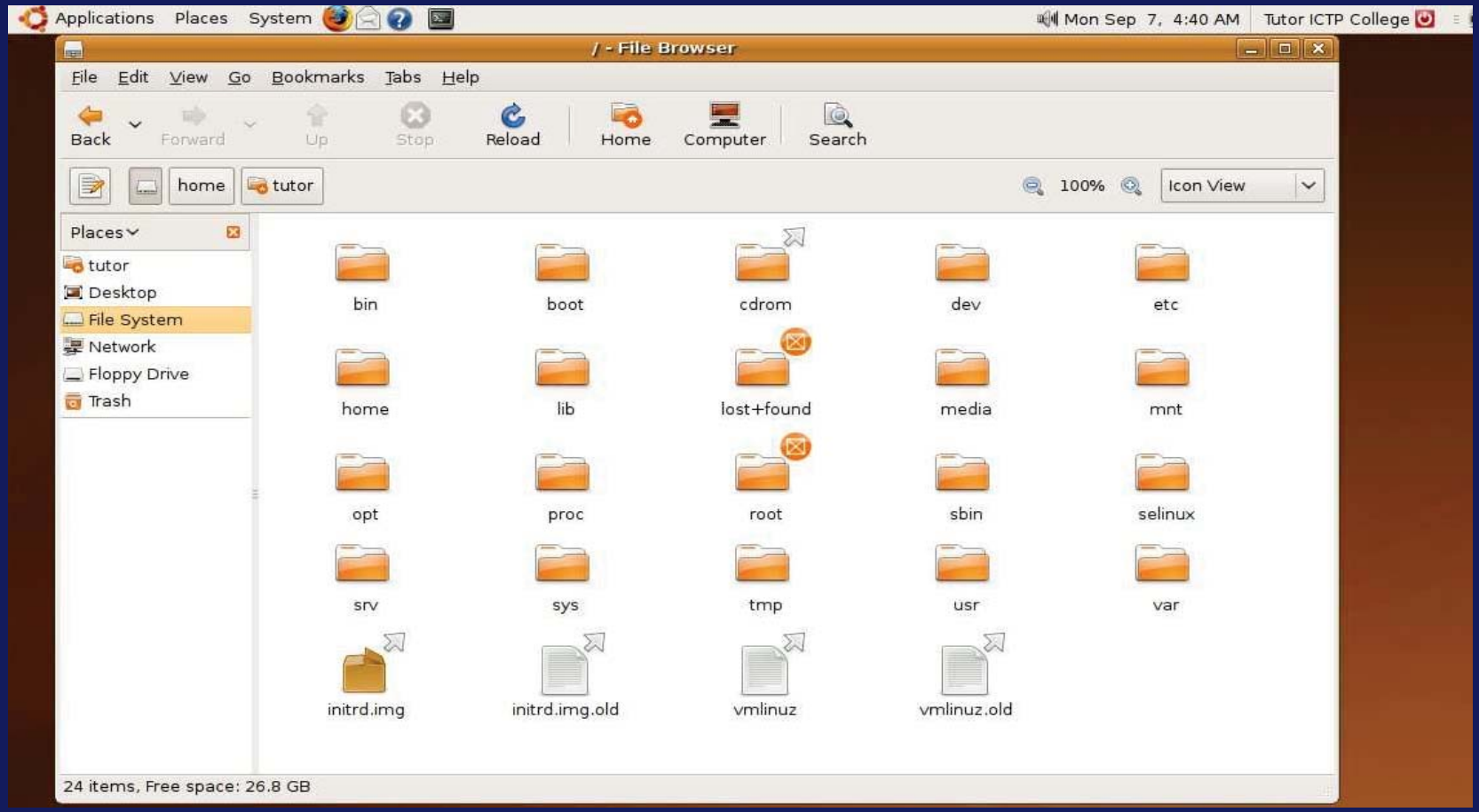
GUI – System tray

- Log out etc



GUI - filesystem

- /



Command Line Interface (CLI)

A program requests services from the system via a system call. Such a system call could request to read a given sector from disk, or to allocate a block of memory, or to send out a character on the serial port, etc.
A user, sitting in front of his terminal, interacts with the system via a command interpreter or shell.

The shell translates the command into sequences of system calls.
There are a number of shells in circulation ('sh', 'ash', 'ksh', 'csh', 'tcsh', 'dash' etc.).
Most likely, you will use bash, the 'Bourne Again Shell'.

CLI - shell

- Linux, with the help of the 'shell' is very flexible:
  - Most commands have (lots of) options.
  - You may change the name of the commands, or define (a set of) standard options for each of them: aliases.
    Examples:
    alias zf='/home/razaq/testarea/ZendFramework/bin/zf.sh'
    alias cp "cp -p"
  - You can also access files with a sort of 'pseudonym', using 'symbolic links'.
    ln -s targetname linkname
  - Command 'completion':
    You type:
    less longf'tab' (longf, followed by a 'tab')
    and the shell translates into:
    less longfilename.longpostfix
    if that is an unambiguous, existing filename.

CLI - shell

- User environment – a collection of specially named variables that have specific values.
  export LOG=/opt/android/platform-tools
- They are called environment variables:
  to view the environment use:
- env or printenv (depending on the type of shell)
  LOG=/opt/android/platform-tools
- If you now type: $LOG/adb logcat
  the shell will launch execution of:
  /opt/android/platform-tools/adb logcat
- Use the echo command to list specific environment variable:
  echo $HOME or echo $LOG

CLI - shell

- An important environment variable is PATH.

PATH contains the list of directories where shell will look for the executable specified (by you) on the command line. You can set the PATH:

PATH=$PATH:/home/razaq/workarea

- Getting help
  - You can get help on practically all commands from the man pages (manual pages).
  - $ man command
  - It is wise to invoke xman & after logging in.
  - Other way to find help is to use -h for the command option.

CLI - Using the commands

- You must get accustomed to command names:
- 'passwd' for changing the password
- 'cp' for copy,
- 'rm' for del,
- 'ls' for directory list.
- 'mv' for moving or renaming a file
- 'mkdir' for creating a directory
- 'chmod' for changing the permission on a file or dir.
- 'less' or 'more' or 'cat' for listing the contents of a file
- 'ps' for reporting a snapshot of the current processes
- 'pwd' for showing the current location or current working directory
- 'chown' for changing the ownership of a file or dir.

CLI - Using the commands

- When you have logged in, you will find yourself working in your home directory.
- When login for the first time, you may want to change your password using passwd. You will be asked for your current password and a new one.

- The contents of the any directory can be listed:

  ls [options]

- You move to another directory with:

  cd dir-path, where 'dir-path' is either a full or a relative pathname.

CLI - Using the commands

- If I am in /home/razaq and I want to go to /home/jim/ I do:
  cd /home/jim/ or
  cd ../jim/
  cd $LOG takes me straight to:
  /opt/android/platform-tools
  To go to Documents in home directory:
  cd Documents
- If you are lost and don't know where you are, using pwd will show your current location.
- creating a new directory:
- mkdir newdirname

CLI - Using the commands

- moving a file from one directory to another:

  mv source existing_dir.
- can also use mv to rename a file or directory

  mv source newname
- To delete a directory:

  rmdir dir_to_delete
- To delete a file(s) or directory:

  rm filename

  rm -rf directory_to_delete
- THINK TWICE BEFORE USING rm and rmdir !!!!
- You can change the permission mode for file and dir.

  chmod [ugoa] [+-=] [rwx]

CLI - Using the commands

- To display information about the active processes one can use ps:
  ps [options]
- Pipes:

command1 | command2 means:
Output from 'command1' is fed into 'command2' as its input. 'command1' and 'command2' are so-called Filters.

  ls | wc

- Normally when shell launches a command, it waits for its completion. Appending an '&' to a command changes this behaviour:
  the shell takes back control immediately after having launched the command.

CLI - Using the commands

- Consecutive commands:
  using ; to separate two commands: they will be executed one after the other.

Redirection:

- > file --Output destined for the screen

(stdout) is written to 'file'. The file is overwritten.

```
ls -al > text.txt
```

- >> file --As above, but output is appended to existing file.

```
cat text1.txt >> text.txt
```

- 2> file and 2>> file : same as above, but now concerning error output (stderr).
- < file --Input is taken from 'file', instead of keyboard (stdin).

```
grep text < text.txt
```

CLI - Using the commands

- Superuser (root, administrator, supervisor)
  su [options] [username] - change user ID or become superuser
  su
  su username
  su - username
- NEVER use root as normal user account
- NEVER login as root, use normal user account to login and then change to root when necessary.
- Best practice use sudo.

CLI - Using the commands

- Shell scripts:

You may collect a series of shell commands into a file.

After having set the execute permission for this file, you can execute its contents with a single command:

the name of the shell script.

The first executable line in the script must be:

#!/bin/sh or #!/bin/bash or similar.

CLI - Utility Programs

- Utility Programs

There are a number of utility programs you should be aware of. They can be very useful and you may find yourself using them repeatedly when you are developing software.

- The programs we want to talk about very briefly are:

sudo, cat, more, less, whereis, find, grep, tar, ssh and scp.

- We will also dwell on:

make, gcc, gdb , to conclude with the development cycle:

--> edit --> compile --> debug -->

| |

<-- <-- <-- <--

cat, more or less

- cat filename lists a text file to your screen, you can read the last few lines.

- more filename does the same, but shows one screenful at a time.

(SPC) scrolls one entire page, (RET) one line.

- less filename does it better: you can go back also, one screenful at a time, typing a 'b'.

- cat file1 file2 ..... filen > newfile

concatenates the files and writes result to newfile.

- You may also try head and tail.

whereis, find

- whereis name-of-executable will show where the executable file is stored.

- find is of much more substance. It will find files of a given name, date, or length or older than date, longer than length, etc.

What is more, once a satisfactory file is found, you can make 'find' to do something with it.

whereis, find cont...

- One can explain 'find' for hours. For now, four "simple" examples:

```
find /usr/bin -name unzip

find . -cmin -60

find .. -size +110k

find .. -size +110k -printf “\t %s \t %p \n”

find .. -size +110k -exec tar zcvf {};
```

whereis, find cont...

Reading the 10 or so man pages is a MUST , if you wish to make reasonable use of 'find'.

tar

'tar' stands for 'tape archiving' which was the original aim of this useful program. It is now often used to make a nice compressed package of a (large) number of files:

tar zcvf parcel.tar.gz file1 file2 .... filen or
tar zcvf parcel2.tgz ./*

The latter will create (option 'c') verbosely ('v') a compressed ('z') tar' file ('f') parcel2.tgz from all files in the current directory.

Once created, you can inspect the contents:

tar ztvf parcel2.tgz | less

tar continue...

You can unpack your 'parcel2.tgz' with:

tar zxvf parcel2.tgz

grep

'grep' is a most useful program:

grep -n string *.c *.h
will print for you all lines in all '.c' and
'.h' files in the current directory where the
string has been found, preceded by the
filename and the linenumber.
ls -l ~ | grep \.ps
will show all your '.ps' files and only those.

grep continue...

Similarly for creation dates:
ls -l ~ | grep May or even:
ls -l ~ | grep \.ps | grep -v May

The '-v' option will show all lines that do NOT contain the string.

ssh, scp

- Remote login ssh – OpenSSH client

ssh [username@]hostname
ssh orange@abdc.com
ssh abdc.com

- Remote file copy scp – secure copy

Copying file to a remote host:
scp source-file user@host:directory/target-file
scp myfile.pdf orange@abdc.com:
Copying file from a remote host:
scp user@host:directory/source-file target-file
scp orange@abdc.com:Pictures/mypic.jpg Pictures
scp -r user@host:directory/source-folder target-folder

make

- 'make' is a command generator.
- From a description file and general templates it creates commands for the shell.
- 'make' sorts out dependencies among files.

make *continue..*

A program often consists of several source files, header files, libraries.

When one of these is modified, you must rebuild the program, by re-compiling some of the files, but not necessarily all, and then re-linking the object files.

'make' decides what must be done, basing itself on the dependencies and the last modification date/time of each file.

If file A depends on file B and B was modified after A had been built, then A must be re-built:
compiled, linked, edited, substituted in a library or what have you.

make *continue..*

Generally you invoke 'make' by typing:

make myprogram

'myprogram' is the target, it is built from one or more files, the 'prerequisites' or 'dependencies'.

The dependencies are specified in a description file (Makefile or makefile), together with the commands to be executed to build the target.

make continue..

Example of a Makefile:

```
program : main.o iodat.o dorun.o lo.o /usr/lib/crtn.a
        cc -o program main.o iodat.o dorun.o lo.o /usr/lib/crtn.a

main.o : main.c
        cc -c main.c

iodat.o : iodat.c
        cc -c iodat.c

dorun.o : dorun.c
        cc -c dorun.c

lo.o : lo.s
         as -o lo.o lo.s
```

Note that there are dependency lines or rule lines containing a ':' and command lines, starting with a 'tab' character.

make *continue..*

The example is rather clumsy, repetitive. Things become simpler by using macros and by exploiting the suffix rules.

A macro is defined as:

name = "a text string" ('quoted' if necessary).

You refer to a macro with:

$(name) or ${name}

make *continue..*

Example:

```
LIB = -lX11
objs = drawable.o plot_points.o root_data.o
CC = /usr/bin/gcc
23 = "This is the 23rd run"
OPT = # empty now, use later
DB = -g
BINDIR = /usr/local/bin
plot : ${objs}
        ${CC} -o plot ${DB} ${OPT} ${objs} ${LIB}
        mv plot ${BINDIR}
```

make continue..

When you now type: make plot the shell will receive the following commands:

/usr/bin/gcc -o plot -g drawable.o plot_points.o root_data.o -lX11

mv plot /usr/local/bin

make *continue..*

Macros can be nested. The order of definition is immaterial. 'make' has also macros defined internally.

Shell environment variables can be used as macros in a Makefile. So, if you have a shell environment variable:

DIR = /usr/proj
and you have 'exported' it:
export DIR (for bash) or:
setenv DIR /usr/proj (for tcsh)

make continue..

then you may use ${DIR} in your Makefile:

```
SRC = ${DIR}/src

myprog : ....
        cd ${SRC}
```

make *continue..*

These are the very essentials only of 'make'. For the details (many!), see the man pages or the book by A.Oram and S.Talbott, Managing projects with make. The examples were taken from this book.

'make' is absolutely essential for developing software of some complexity, especially if done by a team of programmers.

gcc

gcc stands for "GNU Compiler Collection."
It contains compilers for C, C++, Objective C, Fortran, Java and a few more.

gcc continue..

Usually the compiler chain is invoked with:

gcc [-options] -o Prog file1.c file2.c...filek.s

(or something similar), where Prog is the name of the executable file that is to be produced as the end product.

gcc continue..

C compiler consists of a chain of programs:

- preprocessor (cpp) transforms the '.c' program into a '.i' file, using the '.h' files.
- compiler (cc1) does the real job:

translates C code into assembly code ('.s' file)

- assembler (as) translates the '.s' file into object code ('.o' file)
- linker (ld) 'glues' all '.o' files together

and includes functions from libraries.

A cross-compiler has an overall wrapper program, usually called xgcc.

gcc continue..

The most important options are:

- -g produce information for the debugger.
- -I incdir include '.h' files from 'incdir'.
- -L libdir search 'libdir' for libraries.
- -l libname search the library 'libname' for functions to 'link' in.
- -o prog name the executable: 'prog'.
- -S produce only '.s' file(s). Don't assemble.
- -c produce object ('.o') files. Don't call the linker to produce an executable.
- -Wall give all possible warnings.
- -O2 optimize the code produced.
- -v verbose mode (I find it useful).

gcc continue..

Normally, you will wisely specify the options to use in your Makefile.

gdb

gdb is a symbolic debugger, which means that you can use line numbers and names of variables and functions as defined in the C source code.

The executable to be debugged must have been compiled using the '-g' option of gcc.

What Next?

Finally, after this workshop those who are new to linux and windows users can continue to work in a linux environment such as:

- cygwin can be use to run unix commands and applications in windows. Can also be use to connect to linux remotely. http://www.cygwin.com

- There are projects like Ubuntu, knoppix etc for those who are new to linux.They are easy to install and work with.

http://www.ubuntulinux.org/
http://www.knoppix.org/

What Next? continue...

- Small linux - Damn Small Linux(DSL) and Puppy Linux. These are fully functional linux that can fit on a small CD or jumpdrive between 50-70Meg.

http://www.damnsmalllinux.org/
http://www.puppyos.com/

For the various distributions of linux, try:
http://distrowatch.com

More information about linux from:
http://www.linux.org
http://www.gnu.org
http://en.wikipedia.org/wiki/Unix_shell