Introduction to Linux

ELEC132

SECTION 1

Introduction

Embedded Systems

There is no one definition of an embedded system. Typically it is an electronic system with a micro that runs some software that is not readily modifiable. Such systems can follow one several models.

Deeply Embedded Systems

One model for an embedded system is running software written in C, targeted for a specific Microcontroller. This is the model typically used to directly interface with sensors and control in real-time.

The programmer is responsible for setting up the device, and communicating directly with the hardware via hardware ports or serial interfaces. Code is typically run with no operating system. This minimises the memory requirements and is very suited to specialist tasks, such as sensing and control.

Devices most suited to this are sometimes referred to as “deeply embedded microcontroller”. Such devices are very widely used as they offer many benefits

- They tend to have limited memory and clock speed, thus obtain very low power and cost
- Physically small
- Can interface directly to electronic hardware in real-time

- they are relatively simple.

However, there are some challenges when adopting such technology:

- You may have to write code to set up all the hardware yourself. This can be complex, error prone and is device specific (cannot be easily reused in other projects).
- Handling data from multiple devices at the same time can become complex without support from additional software
- Memory and speed are limited, making it difficult / not viable to build large complex software applications.
- Specific skills needed to write good code on such devices are not always portable to other devices.
- Writing code complex systems that embed many complex devices – such as WiFi, Ethernet, USB and screen drivers can be very challenging.
- There is less protection - accidentally writing into the wrong areas of memory is often possible, and such bugs can be hard to detect and track down.

To tackle complex tasks, **there is great advantage is being able to re-use software** and hardware. Think about how digital electronic circuits are built. They are build from **components**. Components range from the very low-level (AND/OR/NOT) through to devices such as counters, multiplexers up through memory devices, controllers, decoders, memories, converters until you reach the very complex chipsets such as bluetooth and WiFi chipsets.

The key issue here is that these tightly specified 'components' as we know them are often the product of millions of hours development and testing.

The economics of chip manufacturing tends to force manufacturers to perform extensive simulation and testing, meaning they can be re-used with confidence.

Furthermore, if you need/want to be conservative in your designs, you can often choose components that have the benefit of years of use (and refinement). For example. The 74 series logic has been around so long now that it is highly trusted – each components is thoroughly documented and understood.

So why has the same component based approach not been adopted for software? Well, in short – it has. The notion of **Software Components** has been around a long time (in computing terms). This has

been facilitated with the broad update of Object Orientated Programming (OOP). You will meet an OO language in term 2 - C++.

Arguably, in terms of reliability, documentation and testing, it has not (in general) been as broadly successful as hardware.

One view is as follows: there is an issue with human motivation and behaviour. The economic impact if making a mistake in software is not the same as hardware – software can often be changed retrospectively, so the pressure (and discipline) to strictly define and test software is less.

Hardware has 'bugs' too of course – search any manufacturer for silicon errata and you will find plenty of it – but it is published and manufacturers are rapidly notified when an issue arises. But once a components is developed, tested and has proven its worth, it becomes an entity that can be relied upon. Of course electronic devices evolve over time, but they tend to evolve and not be reinvented.

Key to reliability and rapid development is maximising reuse. If software 'components' can be written (and documented) such that they can be re-used, then with time and usage, they will become reliable and trusted entities.

Building on a foundation of reusable hardware and software components enables the engineer to work at a higher level and develop more complex and ambitious systems. Complex systems can then evolve in way that is humanly and economically viable.

Good developers can try and separate their platform specific code and their generic code so at least the generic code can be reused. Some use lots of 'pre-processor' code (#ifdef; #define etc..) to help write multi-platform code. This rapidly becomes very messy, overly complex and most of all, hard to debug.

If you study the history of computers, you will find examples of machines that resemble the modern microcontroller (maybe with most of the peripherals missing). However, even back in the 1960's, the problems described above were recognised.

Wouldn't it be great if you could write software components (in a chosen language) that is truly portable between different platforms and thus become truly reusable?

Well, to some extent you can - welcome to the world of the protected operating system.

SECTION 2

The Operating System

So what is an operating system? Check it out on Wikipedia if you like (it changes year to year, so I've no idea what it says right now, but at the time of writing it hosts a good article which one trusts will incrementally improve through time). Most of us are familiar with at least one operating system. On the desktop computer, the most popular today are:

- Microsoft Windows
- Apple MacOS X
- GNU / Linux or BSD

Others include FreeBSD and Solaris – there are too many to list. Also, sticking to the consumer market, in the mobile phone and small consumer device market, you are likely to have used one of these:

- Symbian
- WinCE / Windows Mobile
- iOS
- Android

Then there are operating systems which you might have used, but were unaware of. These may be running on devices such as cash machines, consumer electrical equipment, machinery and automotive products. These include (there are too many to list):

- VxWorks
- FreeRTOS
- QNX
- Embedded Linux

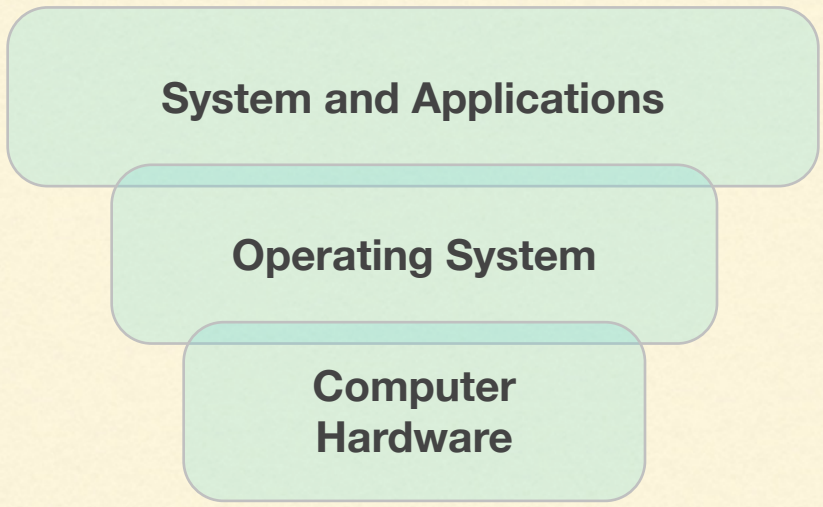
Essentially, an operating system at its core is a specific collection of software components. At the centre of all this is a '**kernel**'.

When we commonly refer to “Linux” for example, when actually we are only referring to the 'Linux Kernel'. A Linux based operating system has a lot more added to it. What is normally the case is that additional software components are added to provide additional services in a layered structure.

Reuse

Key to productivity is re-use, both in terms of other software and services an operating system might provide. You applications can reuse / leverage many facilities of the operating system. These might include:

Figure 1 Hierarchical relationship between the computer hardware, operating system and the applications.



As a developer, you can create applications built to run on a specific operating system. This way you can leverage all the facilities it provides.

Other applications or libraries

An operating system often comes with many additional applications and libraries. These cover functions from writing messages to a console to real-time video encoding.

Multi Tasking

Very often, you will want to run separate applications at the same time and/or perform many tasks simultaneously within an application. The facilities for doing this are provided by the operating systems. When this is done automatically for you, it is known as pre-emptive multi-tasking. When the developer has to do some work to relinquish control and allow another to run, then this is known as cooperative multitasking.

Filing systems

Filing systems (such as those you've already met on desktop operating systems) are a function of the operating system. They are sophisticated, often providing safeguards against file corruption. You applications can make use of the filing system just as a user would. Filing systems are protected with 'permissions'.

Networking

Networking is complex. Having to manage devices and all the protocol stacks yourself would be a complex and error prone task.

Protection, security and permissions

We live in a world where we have to protect code from unauthorized access. We also have to protect our code from our own mistakes. A protected operating system provides many

security features, including file permissions and protected memory, whereby separate applications cannot access the memory of each other.

Hardware Abstraction

When you start up an operating system (such as Linux), the hardware will already be configured for you. The applications you write cannot readily access the hardware directly as you simply don't have 'permission'. As depicted in Figure 2, you have to make 'system calls' via the kernel. In turn, the kernel will broker the request for you. There are parts of the kernel known as device drivers and modules which translate commands into hardware specific commands.

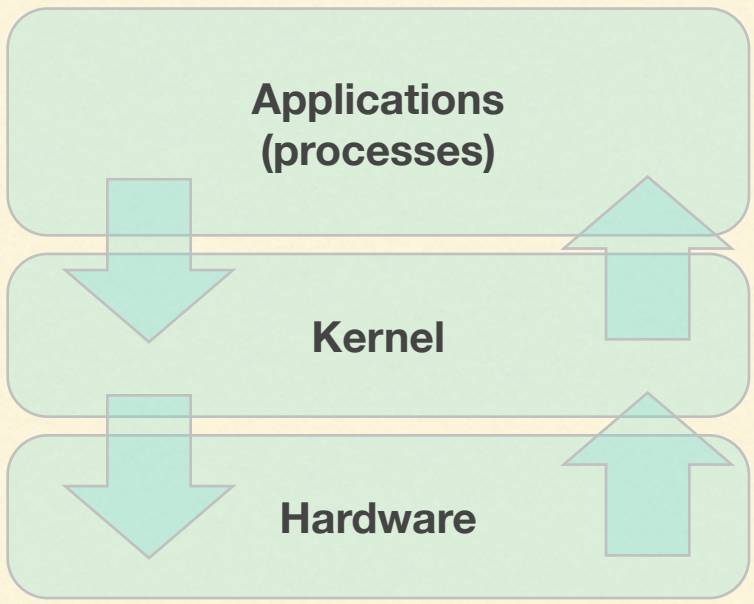


Figure 2 The three fundamental layers of an operating system.

Note that for a protected OS, only the kernel has access to the hardware. Protection is important to make a system robust

There is another subtle benefit of using this multi-tiered structure, and that is power saving. A pre-emtive kernel controls when you application can run. If you application has nothing to do, it can also be allowed to sleep, thus preserving CPU cycles and power. You application may want to wait from some data from a hardware device, such as a disk or network (this is known as blocking). If the device is busy or just slow, the operating system will allow your application to automatically idle. When the device is ready and has some data, this is passed to the kernel, which in turn wakes your application, provides the data and allows it to run one more. All of this is completely automatic to the developer.

Protection

An operating system such as Linux offers many forms of protection and much of it is supported down at hardware/CPU level and managed via the kernel.

- You can safely run multiple applications (processes) at the same time.
- Each running process is protected from all others – you cannot overwrite or easily share memory between processes. This prevents a rogue application from crashing another or stealing data.
- A running process cannot access the hardware directly – it must communicate with the hardware vi the kernel and the kernel will handle occurrences where a resource needed by more than one process. This is very different from most deeply embedded microcontrollers.
- The assembly instructions executed by a running process may only be from the set of non-privileged instructions (handled by the compiler). If an application attempts to execute a privileged instruction will raise and error and prevent the instruction / application from executing. Much of the kernel is run in 'privileged mode'.
- Errors can be caught and handled cleanly by the application (inc. divide by zero; memory errors; execution of privileged instructions).
- The operating system has full control of all running processes – it can start, stop and even kill off any process without directly impacting on any others.

Multi Tasking

You may have already experienced 'blocking hardware' (most hardware is blocking) through use of a desktop computer. Maybe it was a web page that took a long time to load, or a file that took a long time to open. Maybe you clicked a mouse button, and nothing happened immediately. These are extreme examples of blocking. Most hardware blocking is not perceivable to the user however.

Hardware devices, even fast devices such as the memory you install in your computer, is slower than the CPU. In fact the only memory in your computer that is likely to keep up with the CPU is on chip (and limited to a few kBytes at most).

If computers always waited for hardware and did nothing else, our computers would feel a lot slower. Modern micros are designed to support multi-tasking. Some are designed for real-time, others for overall performance.

By real time, we could imply many things, and this is a discussion for later.

Key objectives of multi-tasking are:

- **User Experience** - To allow smooth running of multiple applications
- **Performance** - When one task is stalled (blocking), other tasks can continue to use the CPU
- **Power Saving** - To allow the CPU to enter low-power idle states when there is no work to do

In this module, we will try and stick to the POSIX API as much as possible so that the skills you learn are as transferrable as possible.

UNIX, LINUX and the POSIX standards

One of the earliest operating systems was UNIX developed by AT&T in 1969 (the transistor was becoming popular about then!). It is still used today.

Linux stands for "Linux Is Not UniX" - yet oddly enough, it resembles it is many ways.

True UNIX operating systems are said to be fully **POSIX** compliant (e.g. Mac OS X). Many embedded operating systems are UNIX variants and are either partially or fully POSIX compliant. Linux is partially POSIX compliant. Part of the POSIX standard is a set of programming functions, or "Application Programming Interface" (**API**).

SECTION 3

Linux Practical Getting acquainted with Linux

Intended Learning Outcomes

By the end of this session, you should be able to:

- Open a terminal shell
- Use shell commands and GUI environment to modify, list and traverse the file hierarchy.
- View, create and edit text files with character based applications.
- Create and unpack portable archives of subfolders.
- Create, edit and run scripts to automate repetitive tasks.

For practicals, you will be using the Linux Operating System. There are many Linux distributions, offering different combinations of install applications, kernel etc. Some are tailored for specific use. Ubuntu is a popular general purpose distribution for general desktop computing. Others such as Ångström Linux are smaller distribution tailored for embedded devices. For the desktop, we have chosen the Ubuntu LTS distribution for a number of reasons: One reason is that it is so popular, it is easy to get help on the internet. It is also officially supported by popular **virtualisation** tools. The main reason however is that it offers a Long Term Support (LTS) version which is supported for 5 years.

Managing files and directories

Linux has a directory structure that is conceptually 'similar' to Windows or OS X. There is a root folder '/', within there are a number of subdirectories. You should not be working with the root folder very often. More often, you work within your home folder.

Creating and traversing directories

Open a command shell and type

```
pwd
```

`pwd` stands for **print working directory**. It echoes back your current directory. You should be in your home folder (`/home/student`).

Note that Unix and Linux use the forward slash / as opposed to MS Windows which uses the \ back-slash.

You can list the contents of you home directory using the `ls` command.

Create a new directory **myfiles** using the `mkdir` command

(type `man mkdir` to find out how to do this)

Type `ls` to verify this has been done.

Navigate into your new folder by **c**hanging **d**irectory using the `cd` command


```
cd myfiles
```

Verify you are in the correct directory using the `pwd` command.

Now create and edit a new text file called text1 by typing the following:

```
gedit text1
```

Type the following into the file and save:

> *hello, this is a text file created with gedit:*
> *my name is <enter your name here>*

Note that `gedit` DOES require a GUI.

Close gedit and inspect the contents of the file as follows:

```
cat text1
```

Now create another text file, only this time we are going to use the (in)famous and ever present `vi` editor

Type the following:

```
vi text2
```

This opens the vi editor in **command mode**. To insert some text at the cursor press lowercase `i`

Type the following:

```
This is another text file
My name is now changed to jim
```

To return to the command mode, press the **escape key**. Some useful commands are given below.

Save your file and quit.

Note that the `vi` editor, although hard to use at first, does NOT require a GUI. Also, `vi` is (almost) always present on any UNIX or Linux system with very few exceptions.

ASSESSED TASK 1 - folders and files

Using only the command line, do the following: (keep a note of everything you type)

1. In your home folder, create a subfolder called `mydetails`

2. Inside that folder, use the vi editor to create a file called `name`

3. Type your name into this file, save and quit

4. Echo the contents of this file to the terminal to verify it has worked

For all the above, write down everything you typed and show the tutor.

You must also show the tutor your university ID

Common commands used with the vi editor. Press esc to enter command mode.

x	delete a character	b	jump to beginning of a word / previous word
d	delete a line	e	jump to the end of a word / next word
a	append the text (and enter insertion mode)	/	/<expression> to find expression / alone will find next
i	enter insertion mode	:w	write to file
v	yank (copy line)	:q	quit
	mark text (for copy), then press x for to cut or y to copy	:wq	write to file and quit
p	put (paste).	:q!	quit (without save)

Concatenating files

Now you are going to create a new file, constructed from the two you have already created.

You can use the `cat` command to echo back the contents of a text file. Formally, the cat command "echos the contents of a file to **standard out** (stdout)". Standard out is known as a **stream**. Unless redirected, stdout for a given application is the terminal shell in which it was executed. It can be redirected, and this is what we are going to do here:

Type the following:

```
cat text1 > newtext
```

Inspect the contexts of newtext to verify it is simply a copy.

The '>' operator redirects standard out (on the left) away from the running terminal to the device/file/application specified (on the right). This implies the use of another 'stream' **standard in** (stdin) which we will discuss later.

Type the following:

```
cat text1 >> newtext
```

Now inspect `newtext`. What has this done?

So far you have created a folder, used two different text editors to create and edit text files, and concatenated them together. Now you are going to delete some files and archive the folder.

Creating archives

You may be familiar with creating zip archives, probable by using a GUI. You can also do this from the terminal.

Traverse back up to your home director

```
cd
```

Confirm you can see your folder `myfiles` with the `ls` command

Archive your files in a zip file as follows:

```
zip -r myarchive myfiles
```

Type `man zip` to find out the meaning of the `-r` option. Use the `ls` command to verify you have created the zip archive correctly.

Delete the original directory `myfiles` and everything inside it.

```
rm -r myfiles
```

Use the man pages to find out what `rm` does.

Recreate the folder by unzipping

```
unzip myarchive.zip
```

Delete the zip archive myarchive.zip

```
rm myarchive.zip
```

Now you need to try and work out something for yourself. If you get stuck, ask the tutor.

Using the `mv` command, and with the help of the man pages, rename the directory `myfiles` to `mytextfiles`.

It is important to begin to learn to use the manpages as your first and preferred source of help.

Automation with bash scripts.

The shell you are using is known as 'bash' – the 'Bourne Again Shell' – inspired by the original UNIX 'Bourne Shell'. One of the most important features of this is the ability to write and execute scripts that perform multiple actions.

We will illustrate this by automating some of what we did in the earlier exercise. In this task we will write a script to archive a folder and delete the original folder. For now we will hard code the folder name.

You should have a folder called `mytextfiles`.

Confirm this using the `ls` command

You should be in your home folder.

Confirm this using the `pwd` command

Now create a text file `zipmywork` in your home directory and write the following inside it:

```
zip -r myarchive mytextfiles

rm -r mytextfiles
```

Having done this, save this file. Confirm there is a file zipmywork in your home directory. Check the contents using the cat command.

Method 1 – run your script by executing a new bash shell, and passing the script name as an argument:

```
/bin/bash ./zipmywork
```

Recreate you folder by unzipping the archive and deleting the archive (to get back to where you were).

Method 2 – make your script executable, you need to add permissions to execute the file (the current user student). To do this, type the following:

```
chmod +x ./zipmywork
```

Type `ls`, you will see your script is listed in a new colour (to indicate it is now executable).

Run your script by simply typing the following:

```
./zipmywork
```

Now write a script to do the following:

• unzip your files

• delete the archive

Using command line arguments

Now let's make this script more useful so that we are not restricted to using the folder name mytextfiles every time.

You should still have a folder called `mytextfiles`. Confirm this using the `ls` command

---

You should be in your home folder. Confirm this using the `pwd` command

---

Edit `zipmywork` in your home directory and change to content to read as follows:

```
zip -r $1.zip $1

rm -r $1
```

Save this and execute it as follows:

```
./zipmyfiles mytextfiles
```

Note there are no file extensions.

In the script, `$1` is substituted for the first **argument** passed to the script '`zipmyfiles`'.

In effect, it has run the following:

```
zip -r mytextfiles.zip mytextfiles

rm -r mytextfiles
```

There is a lot more you can do with bash scripting. The script you have written is functional, but what would happen if the directory did not exist, or the argument was left off, or two arguments were added? Here is a script that improves matters somewhat.

ASSESSED TASK 2 - script

Now create a new script called `restoremyfiles`

This should the reverse the actions of the script zipmyfiles. The script should...

- take a single argument (a zip archive).
- unzip the archive and restore the folder and its contents
- delete the zip archive

Test it then show it to the tutor:

You must also show the tutor your university ID

Substitution in strings

Note that the string such as `"Archiving folder $1"` actually substitute the argument $1 into the string. This is the defined behavior when using double quotes.

If 'single quotes' were used, this would not happen and you would get a literal string instead.

The following are very different:

```
echo "Hello $1"

echo 'Hello $1'
```

The first would substitute the first argument for the $1 placeholder

The second would echo the literal string `Hello $1` to the terminal.

Conditional statements in bash

Bash scripts are somewhat unconventional (you may have noticed ;o), especially conditional statements.

The `if` statement is structured as follows:

```
if [ conditional test ]
then
      <commands>
elif [ alternative conditional test ]
  <commands>
else
  <commands>
fi
```

Notes for existing programmers

- The spaces in the `[ square braces ]` are needed.
- The NOT operator is !
- The 'then' statement must be on a line of its own.
- Instead of `else if` (as found in some other languages), in bash you use the statement `elif`.
- There are no curly braces like C. The end of an if block is marked with `fi`.

We can only assume the creators of bash thought this amusing.

Don't Panic

This is not a module about bash scripting.

You won't be expected to write many bash scripts. You just need some basic skills.

You won't be expected to write any complex bash scripts

Common Utilities

There are thousands of different possible utilities for UNIX and Linux, nearly all of which are freely available and opensource (the source code is freely available).

You will be introduced to these on a need-to-know basis. However, there are some that are so widely used, that you should learn about them now.

The `less` command

`Less` is a command to inspecting the contents of a text file without leaving up your terminal session filled with unwanted text. It is useful when you want to take a quick look at a long text file without interrupting leaving a load of unwanted text in your terminal session (the man function uses less).

It is best described by example:

Configuration files for Linux are stored in the `/etc` folder. One such file is `/etc/apt/sources.list` which is quite long (it contains the address of all the Ubuntu software repositories)

Imagine you are in the middle of some complex work, but you want to quickly look into the `sources.list` file.

A. Type the following:

```
cat /etc/apt/sources.list
```

A problem here is that the file is very long and does not fit on one page.

B. Now type

```
less /etc/apt/sources.list
```

- Press space to scroll down a page;
- press u to scroll up a page;
- press the up and down arrows to scroll a line;
- press / to search (just like vi and man);
- press q to quit

The `less` command allows you to browse and even search through a long file. The exact same features are also used in the manpages.

The `grep` command

"Global/regular expression/print", or grep for short, is a highly useful (and complex) utility for finding content in files (or streams of characters). Again, it is best described by example:

The basic syntax (according to its man page) is

```
grep [options] pattern [file]
```

where `[square braces]` indicate an optional field.

Again, it is best explained by example:

Imagine we want to find out if we have enabled the 'multiverse' repositories in Ubuntu (these contain additional and often useful software).

There will be references to the term multiverse inside the file `/etc/apt/sources.list`

A. To search for the pattern "multiverse" in `/etc/apt/sources.list`, type the following:

```
grep multiverse
/etc/apt/sources.list
```

This should echo each line where the pattern is matched.

Of course, there are many options and ways to specify patterns you can use to make your search infinitesimally more complex. A common one is -i for a case insensitive search.

Compare the output from the following two lines:

```
 grep major
/etc/apt/sources.list

 grep -i Major
/etc/apt/sources.list
```

ASSESSED TASK 3 - Grep

Using the man pages, add an option to also print out the line number where the pattern is matched.

hint: type `man grep` and use the search to find something about line numbers.

Show the tutor both the mage page entry and the result of the search

You must also show the tutor your university ID

The `find` command

Find does, as you might assume, it enables you to find files/directories in the filing system. It is a very useful command that is worth remembering.

```
find [options] path expression
```

The path is the root path for the find utility to begin.

The expression is the pattern you are searching for. This allows for complex (and non-intuitive) searches.

Finding files

Most often, you will want to simply find a file with a particular name.

The expression syntax you use is

```
find -name <filename>
```

Again, an example will probably best illustrate:

Configuration files for Linux are stored in the /etc folder, but it is easy to forget which subfolder it is in.

To find `sources.list`, type the following:

```
find /etc -name sources.list
```

This will echo back all the explicit paths to files with this name. It might complain about permission denied as the find utility tries to enter directories that you don't have access to.

(ADVANCED) -If you want to suppress error messages.

Output is written to ‘stdout’, errors are written to ‘stdin’. We will learn more about these later in the term. For now, we want to disregard stderr. We do this by ‘redirecting the stderr stream to a ‘null device’.


```
find /etc -name sources.list 2> /dev/null
```

Wildcards - To find a file with a pattern, such as `*.list`, type the following:

```
find /etc -name *.list
```

You can search for text within these files using grep

```
grep -i multiverse $(find /etc -name *.list 2> /dev/null)
```

In this compound statement,

```
$(find /etc -name *.list 2> /dev/null)
```

is evaluated first and passed into grep as a list of arguments.

Permissions

UNIX and Linux based systems support multiple users.

> Linux is a multi-user, multi-tasking operating system.

A computer running Linux can host multiple users at the same time, and for each user, run multiple applications.

Each user has a user id, and belong to any number of groups. Both users and groups has certain permissions.

Normal users are mostly limited to modifying files in their home folder. They have some read access to system folders.

This has a number of advantages:

- A user cannot accidentally delete or modify critical system files, rending the system unstable
- An application run by a user adopts that users permissions - any erroneous or malicious software is theoretically limited to damaging files in the users own home folder.
- Different users can be given different roles.

As a developer, we sometimes need higher level permissions (we will cover more on permissions later in the course).

There is one special user that can do almost anything - that is the **root** user. You should never login as the root user with a very good reason!

However, if you want to install or update software, you will need 'super user' permissions.

Try typing the following commands:

```
apt-get update
```

This command is used to update the database of software hosted by Ubuntu. It fails as you don't have permission. However, your student account has been added to the list of "sudo'ers"

Type the following:

```
sudo apt-get update
```

Type in your password.

Now it runs, but you are still logged in as the user 'student'.

You can use sudo to damage the Linux installation, so use it with care and sparingly.

OPTIONAL ADVANCED TASK 2 (not assessed)

Adapt the script in Listing 1 to write an improved **restore** script.

- It should check the zip archive exists
- If should check if the directory already exists, and delete if if it does
- It should echo back progress

Show the tutor when you are done.

ONLY ATTEMPT THIS TASK IF YOU HAVE EXTRA TIME

API

An application programming interface (API) specifies is typically a lib
accompanying specification. For C Programming, this is typically a se
tions the developer can call. For C++, this would (typically) also includ
tion of objects and methods.

Related Glossary Terms

Drag related terms here

Index Find Term

Section 2 - The Operating System

Hardware Abstraction

A critical feature of most operating systems and micro kernels, a layer
ware that hides the hardware specifics from the developer. For examp
(and related) operating systems, most hardware devices are represente
When you open, read, write and close these files, you are in fact intera
a piece of hardware. Software device drivers will translate these transa
hardware specific instructions. This has the great advantage of hiding
specifics from the developer. This is not only good for productivity an
use, but is also considered safer. Assuming the device driver is correct
it also means the developer cannot crash the computer through incorr
ware configuration or instructions. In an protected operating system,
impossible for an application (userland) developer to access the hardw
rectly. For further information, a good summary can be found on Wik
http://en.wikipedia.org/wiki/Hardware_abstraction

Related Glossary Terms

Drag related terms here

Multi Tasking

The ability to run more than one process concurrently. Typically drive
hardware timer, this is achieved by allowing each process or thread a s
slice to execute before a context switch is made and another process /
allowed to execute. Multicore CPU's can perform full multitasking wh
esses / threads can truly run concurrently without interruption.

Related Glossary Terms

Drag related terms here

POSIX

The "Portable Operating System Interface" (POSIX) is a set of IEEE
to help maintain compatibility between operating systems.

“POSIX defines the application programming interface (API), along v
mand line shells and utility interfaces, for software compatibility with
Unix and other operating systems.” (Wikipedia - accessed 17/09/2013)

Related Glossary Terms

Drag related terms here

Virtualisation

Creation of a virtual environment that resembles the hardware of a re
puter. This allows the running of an unmodified operating system as t
another, or the running of several operating systems simultaneously o
cal computer.

In our learning environment, you can liken this to running an operati
as a regular Windows application.

Related Glossary Terms

Drag related terms here