# PART I Install VirtualBox and Ubuntu OS

**Step 1: Download and install VirtualBox**

*Virtual Box is a powerful virtualization product available as open source software. It runs on Windows, Linux, Macintosh and Solaris. Virtual Box supports a large number of guest operating systems such as Windows (NT 4.0, 2000, XP, Server 2003, Vista, Windows 7, Windows 8, Windows 10), DOS/Windows 3.x, Linux (2.4, 2.6, 3.x and 4.x), Solaris and OpenSolaris, OS/2, and OpenBSD.*

1. Download VM from https://www.virtualbox.org/wiki/Downloads
2. Double-click on the downloaded .exe file and follow the instructions on the screen. It is like installing any regular software on Windows.

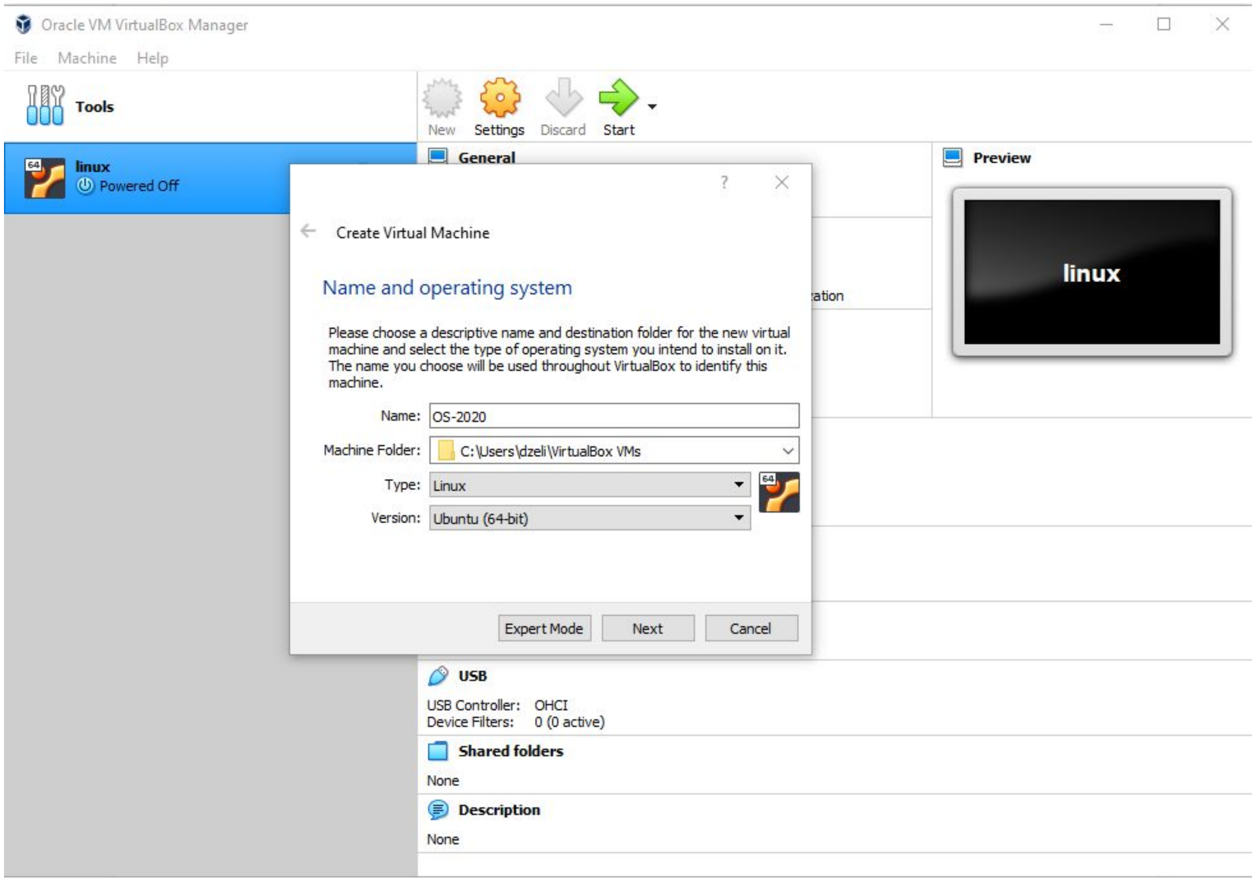
**Step 2: Download the Linux ISO**

*An ISO file (often called an ISO image), is an archive file that contains an identical copy (or image) of data found on an optical disc, like a CD or DVD. They are often used for backing up optical discs, or for distributing large file sets that are intended to burned to an optical disc.*

Next, you need to download the ISO file of the Linux distribution. You can get this image from the official website of the Linux distribution you are trying to use. We are going to use Ubuntu, and you can download ISO images for Ubuntu from https://ubuntu.com/desktop.

**Step 3: Install Linux using VirtualBox**

1. Start VirtualBox, click New and give name to your virtual OS. Select type Linux and version Ubuntu.



Name and operating system

Please choose a descriptive name and destination folder for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: OS-2020
Machine Folder: C:\Users\dzeli\VirtualBox VMs
Type: Linux
Version: Ubuntu (64-bit)

USB

USB Controller: OHCI
Device Filters: 0 (0 active)

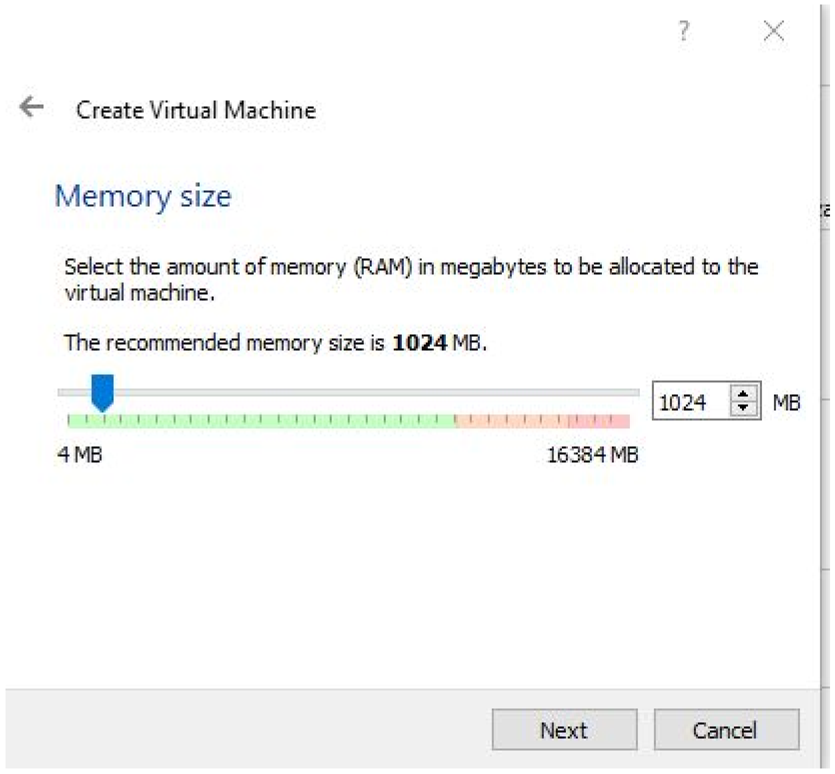
Shared folders

None

Description

None

2. Allocate RAM to the virtual OS.



Create Virtual Machine

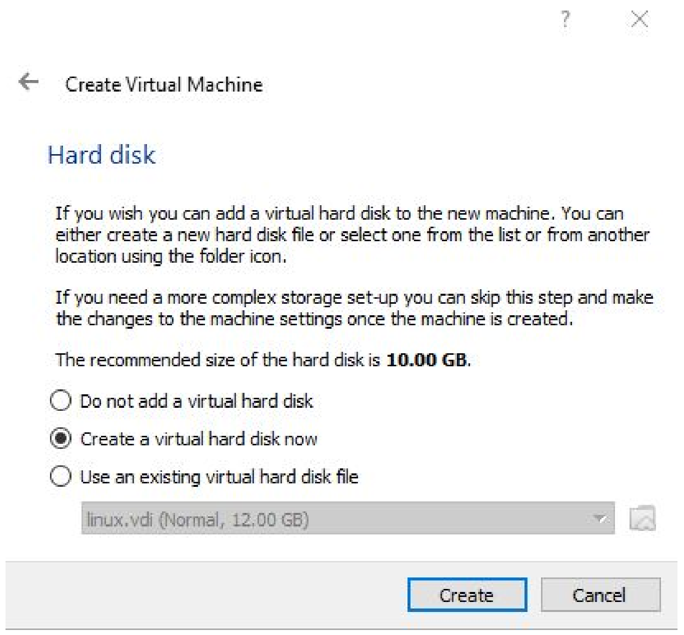
Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **1024** MB.

1024 MB

3. Create a virtual disk. This serves as the hard disk of the virtual Linux system. It is where the virtual system will store its files.



Create Virtual Machine

Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select one from the list or from another location using the folder icon.

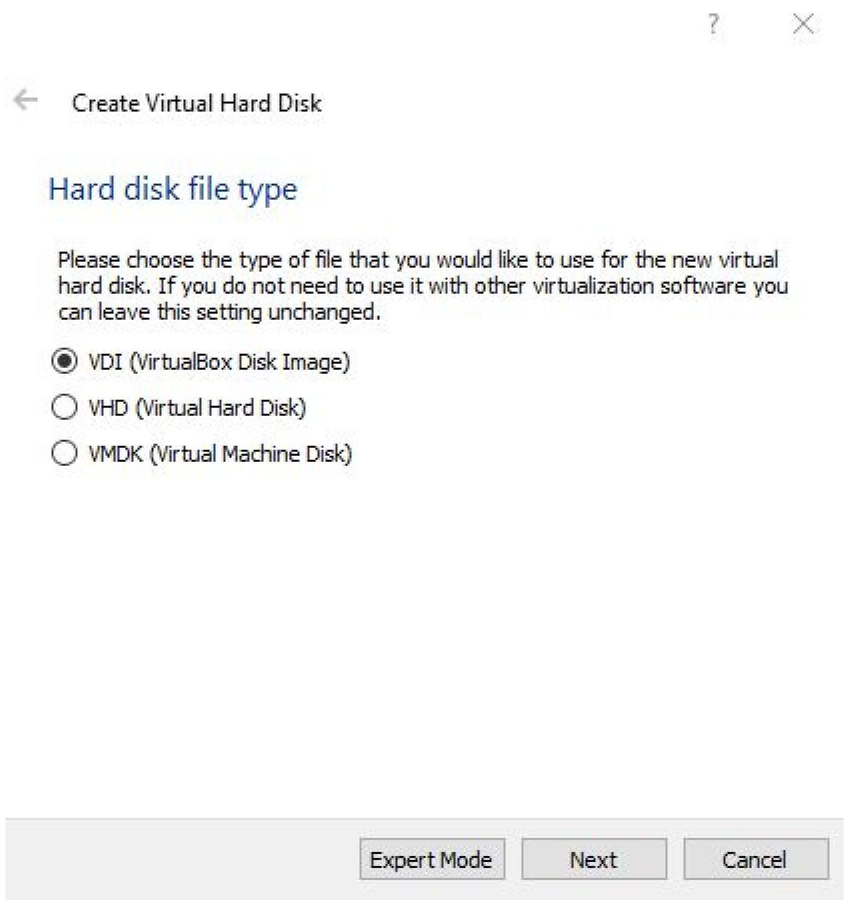
If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard disk is **10.00 GB**.

○ Do not add a virtual hard disk
● Create a virtual hard disk now
○ Use an existing virtual hard disk file

linux.vdi (Normal, 12.00 GB)

4. Use default VDI and select Next.



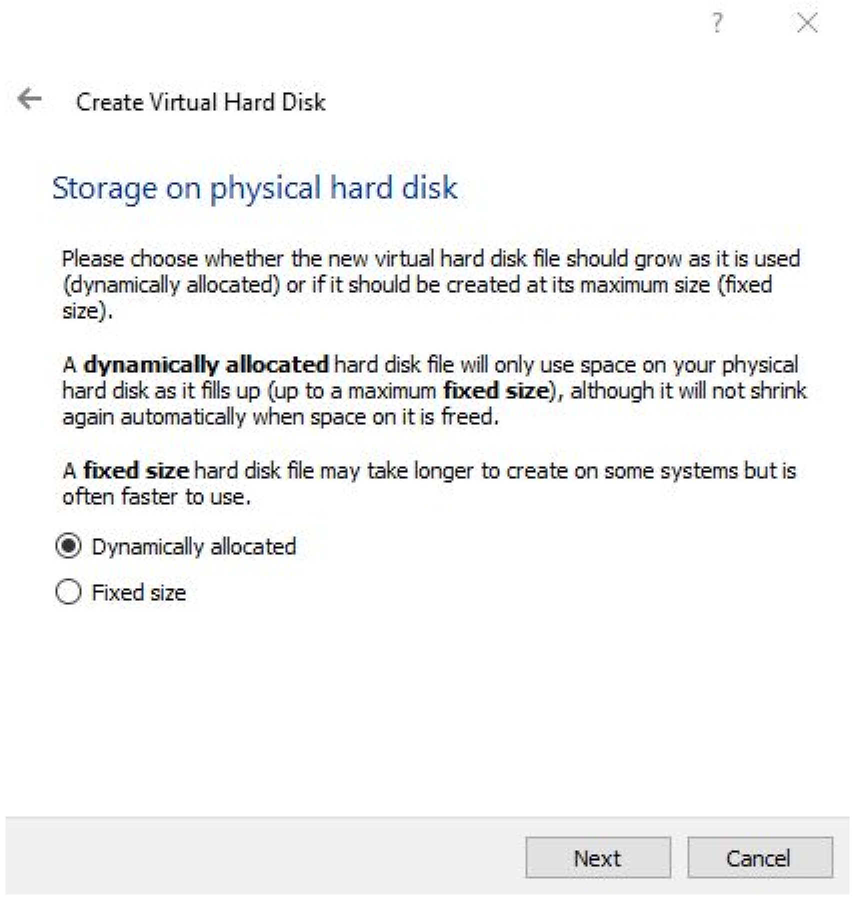
Create Virtual Hard Disk

Hard disk file type

Please choose the type of file that you would like to use for the new virtual hard disk. If you do not need to use it with other virtualization software you can leave this setting unchanged.

[✓ VDI (VirtualBox Disk Image)] VHD (Virtual Hard Disk) VMDK (Virtual Machine Disk)

5. Use dynamically allocated and select Next.



Storage on physical hard disk

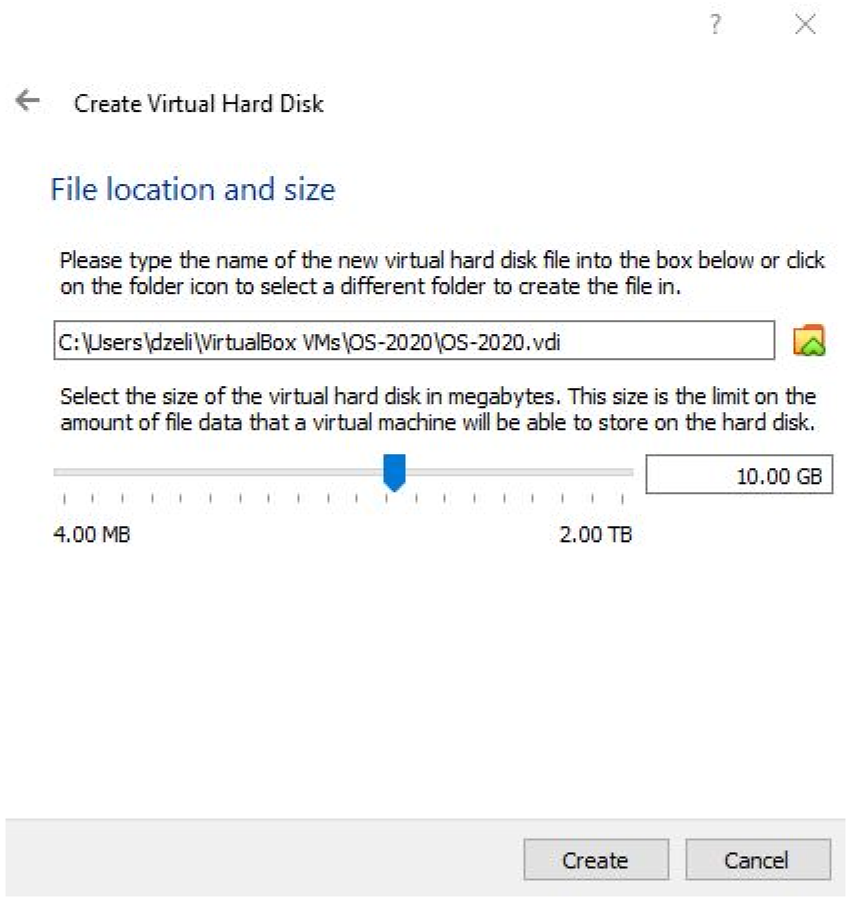
Please choose whether the new virtual hard disk file should grow as it is used (dynamically allocated) or if it should be created at its maximum size (fixed size).

A **dynamically allocated** hard disk file will only use space on your physical hard disk as it fills up (up to a maximum **fixed size**), although it will not shrink again automatically when space on it is freed.

A **fixed size** hard disk file may take longer to create on some systems but is often faster to use.

[✓ Dynamically allocated]
Fixed size

6. The recommended size of virtual hard disk is 10GB.



Create Virtual Hard Disk

File location and size

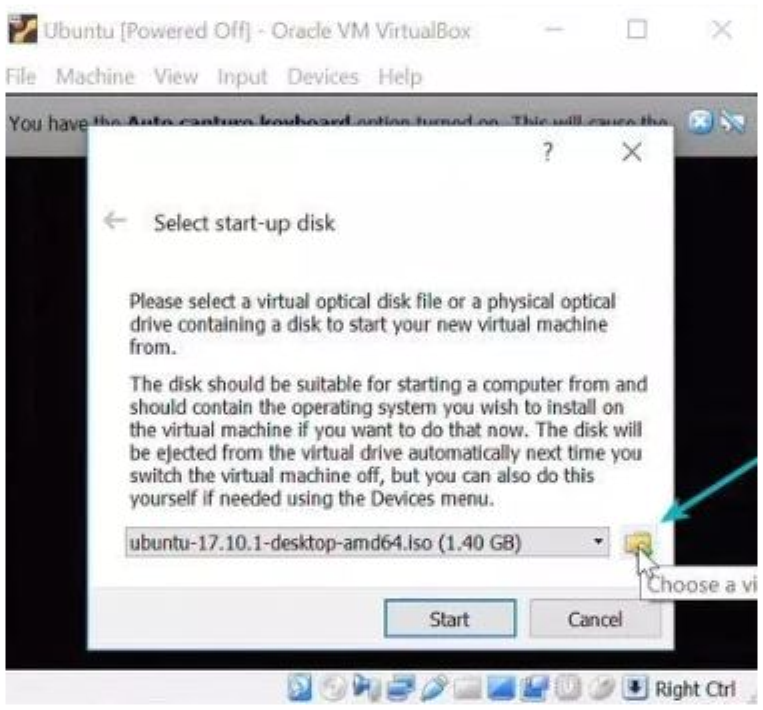
Please type the name of the new virtual hard disk file into the box below or click on the folder icon to select a different folder to create the file in.

C:\Users\dzeli\VirtualBox VMs\OS-2020\OS-2020.vdi

Select the size of the virtual hard disk in megabytes. This size is the limit on the amount of file data that a virtual machine will be able to store on the hard disk.

10.00 GB

7. Select your virtual OS and Start. If VirtualBox does not detect the Linux ISO, browser to its location by clicking the folder icon.


Ubuntu [Powered Off] - Oracle VM VirtualBox
File Machine View Input Devices Help
?
Select start-up disk
Start
Cancel
Choose a vi
Right Ctrl

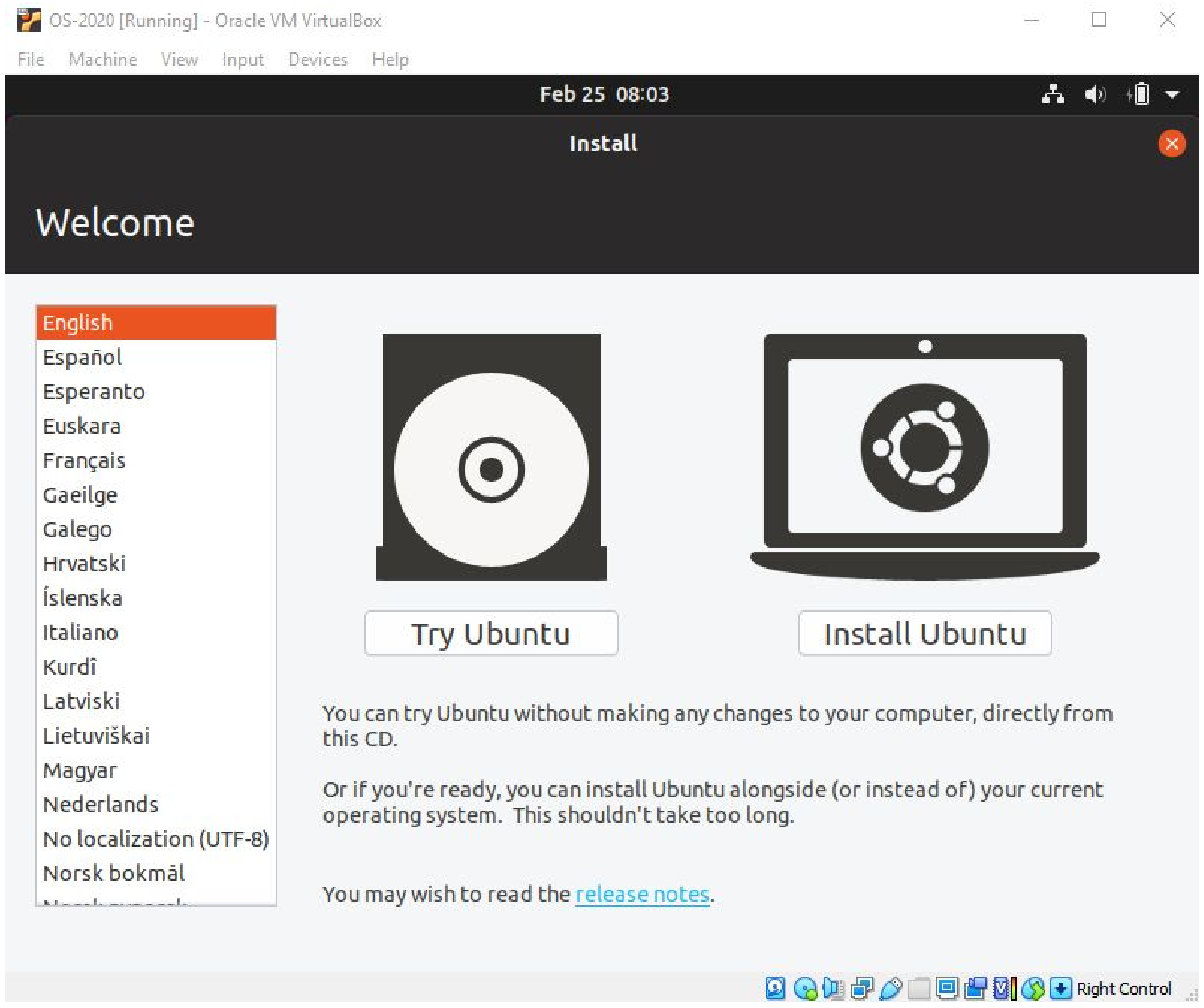
You have the Auto capture keyboard option turned on. This will cause the

Please select a virtual optical disk file or a physical optical drive containing a disk to start your new virtual machine from.

The disk should be suitable for starting a computer from and should contain the operating system you wish to install on the virtual machine if you want to do that now. The disk will be ejected from the virtual drive automatically next time you switch the virtual machine off, but you can also do this yourself if needed using the Devices menu.

ubuntu-17.10.1-desktop-amd64.iso (1.40 GB)

8. Install Ubuntu



Welcome

English
Español
Esperanto
Euskara
Français
Gaeilge
Galego
Hrvatski
Íslenska
Italiano
Kurdî
Latviski
Lietuviškai
Magyar
Nederlands
No localization (UTF-8)
Norsk bokmål

Try Ubuntu

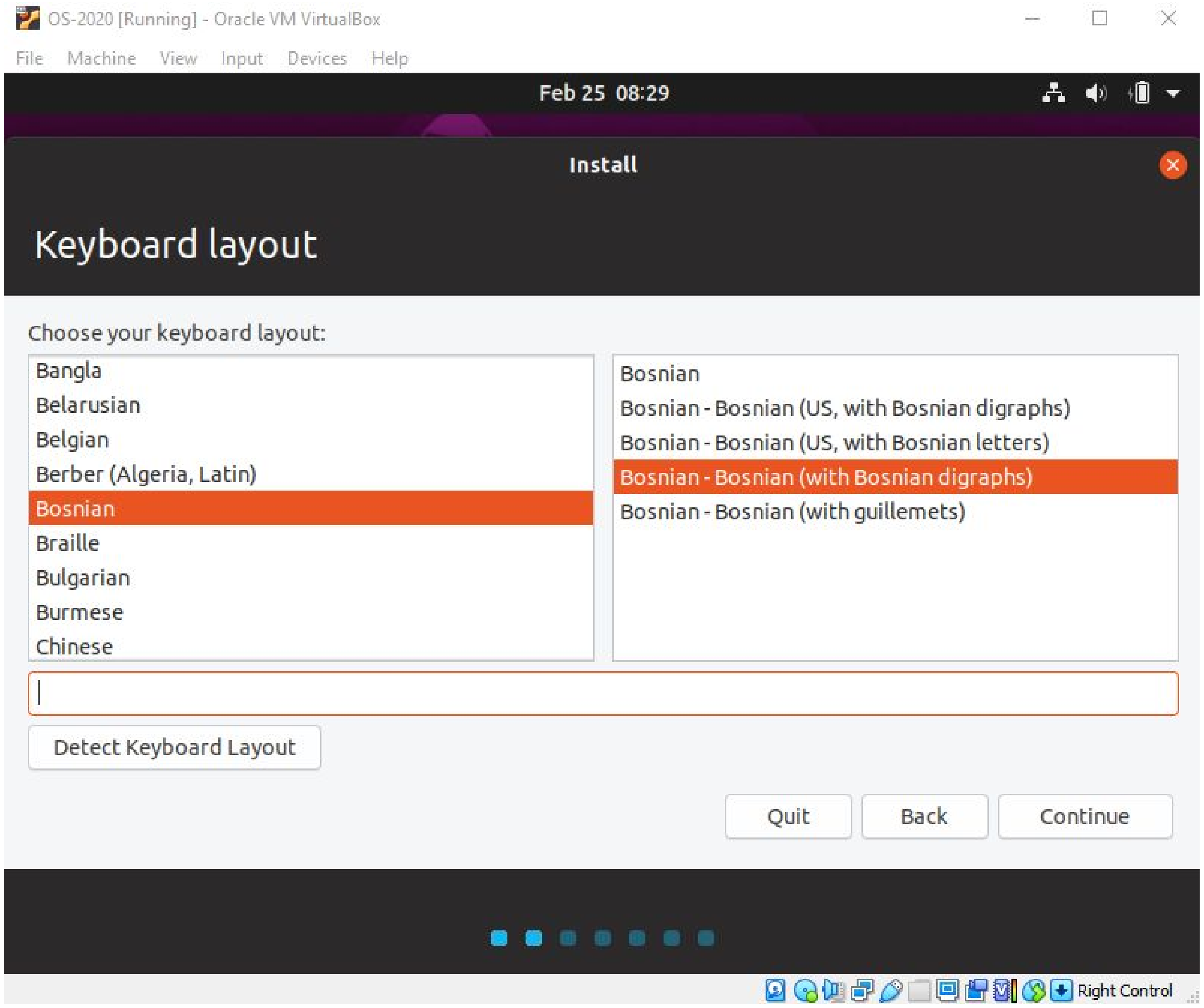
Install Ubuntu

You can try Ubuntu without making any changes to your computer, directly from this CD.

Or if you're ready, you can install Ubuntu alongside (or instead of) your current operating system. This shouldn't take too long.

You may wish to read the release notes.

9. Select language you want.



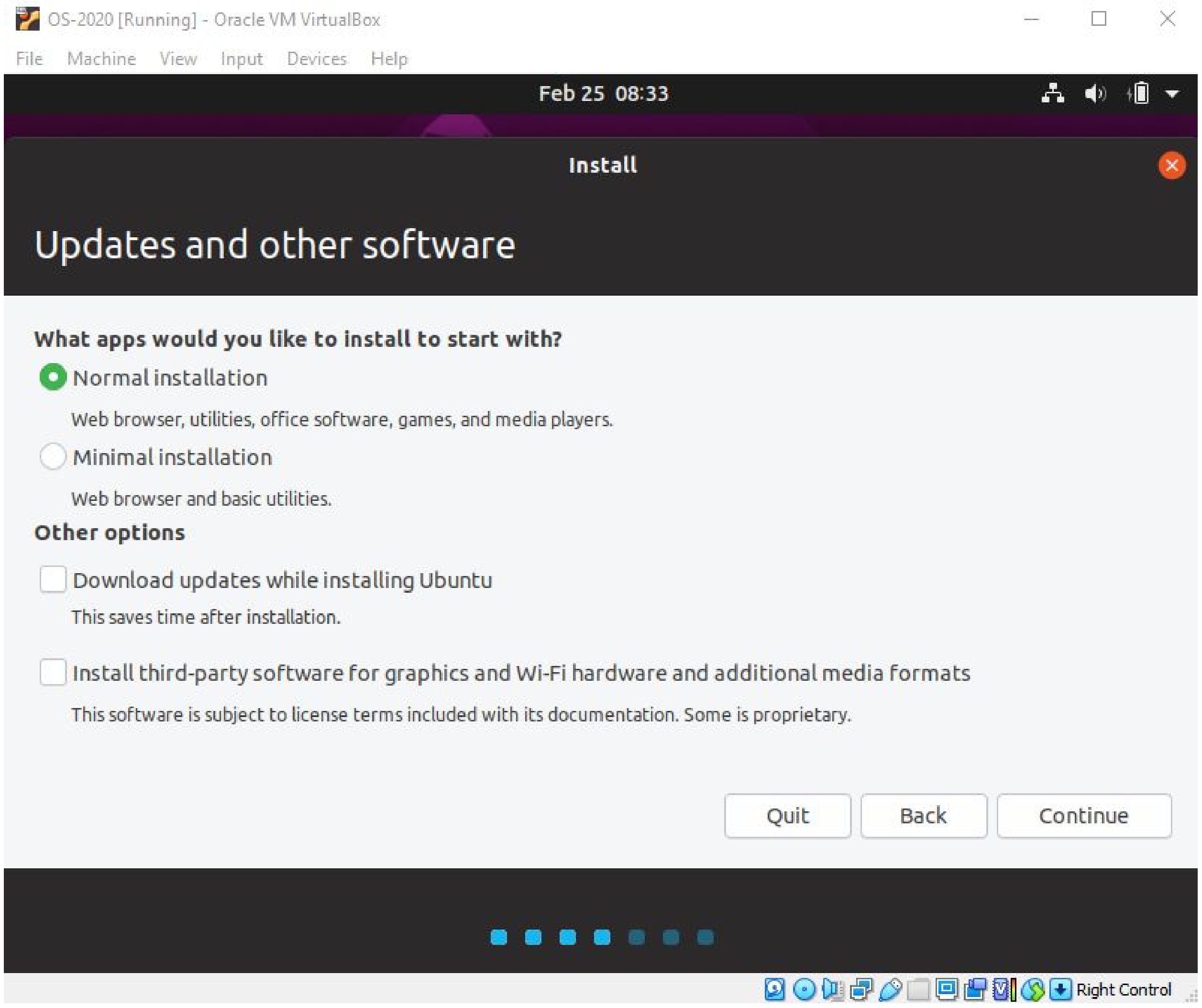
OS-2020 [Running] - Oracle VM VirtualBox

## Keyboard layout

Choose your keyboard layout:

| | |
|---|---|
| Bangla | Bosnian |
| Belarusian | Bosnian - Bosnian (US, with Bosnian digraphs) |
| Belgian | Bosnian - Bosnian (US, with Bosnian letters) |
| Berber (Algeria, Latin) | [✓ Bosnian - Bosnian (with Bosnian digraphs)] |
| [✓ Bosnian] | Bosnian - Bosnian (with guillemets) |
| Braille | |
| Bulgarian | |
| Burmese | |
| Chinese | |

10. Uncheck ‘Download updates while installing Ubuntu’ and continue.


OS-2020 [Running] - Oracle VM VirtualBox

Updates and other software

What apps would you like to install to start with?

(●) Normal installation

Web browser, utilities, office software, games, and media players.

( ) Minimal installation

Web browser and basic utilities.

Other options

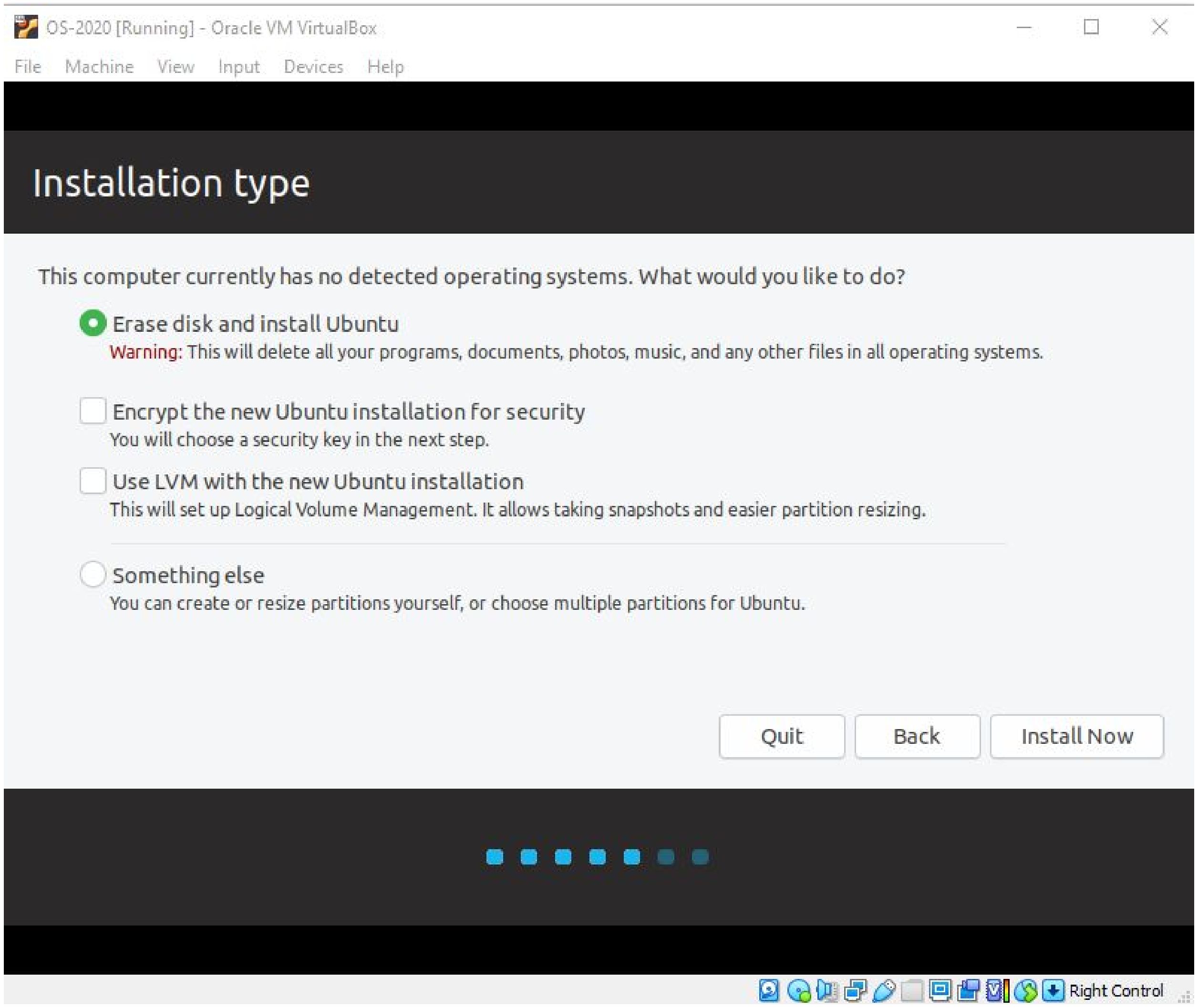
[ ] Download updates while installing Ubuntu

This saves time after installation.

[ ] Install third-party software for graphics and Wi-Fi hardware and additional media formats

This software is subject to license terms included with its documentation. Some is proprietary.

11. Use default option and install.


OS-2020 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Installation type

This computer currently has no detected operating systems. What would you like to do?

[✓ Erase disk and install Ubuntu]
Warning: This will delete all your programs, documents, photos, music, and any other files in all operating systems.

☐ Encrypt the new Ubuntu installation for security
You will choose a security key in the next step.

☐ Use LVM with the new Ubuntu installation
This will set up Logical Volume Management. It allows taking snapshots and easier partition resizing.

○ Something else
You can create or resize partitions yourself, or choose multiple partitions for Ubuntu.

**PART II Basic Ubuntu Commands**

We use commands to do most of our tasks on Ubuntu and other Linux distributions.

1. sudo (SuperUser DO) Linux command allows you to run programs or other commands with administrative privileges, just like "Run as administrator" in Windows.
2. apt-get is used to install, update, upgrade and remove package
3. ls - (list) command lists all files and folders in your current working directory. You can also specify paths to other directories if you want to view their contents.
4. **cd** (change director") Linux command also known as chdir used to change the current working directory. You can use full paths to folders or simply the name of a folder within the directory you are currently working. Some common uses are:
   - cd / - takes you to the root directory.
   - cd .. - takes you up one directory level.
   - cd – - takes you to the previous directory.
5. **pwd** (print working directory) Ubuntu command displays the full pathname of the current working directory
6. **cp** (copy) Linux command allows you to copy a file. You should specify both the file you want to be copied and the location you want it copied to – f*or example, cp* xyz */home/*myfiles *would copy the file "*xyz*" to the directory "/home/*myfiles*"*.
7. **mv** (move) command allows you to move files. You can also rename files by moving them to the directory they are currently in, but under a new name. The usage is the same as cp – f*or example mv* xyz */home/*myfiles *would move the file "*xyz*" to the directory "/home/*myfiles*"*.
8. **rm** (remove) command removes the specified file.
   - rmdir ("remove directory") – Removes an empty directory.
   - Rm -r ("remove recursively") – Removes a directory along with its content.
9. **mkdir** (make directory) command allows you to create a new directory. You can specify where you want the directory created – if you do not do so, it will be created in your current working directory.
10. **history** command displays all of your previous commands up to the history limit.
11. **df** (display filesystem) command displays information about the disk space usage of all mounted filesystems.
12. **du** (directory usage) command displays the size of a directory and all of its subdirectories.
13. **free** – Displays the amount of free space available on the system.
14. **uname -a** – Provides a wide range of basic information about the system.
15. **top** – Displays the processes using the most system resources at any given time. "q" can be used to exit.

16. **man** command displays a “manual page”. Manual pages are usually very detailed, and it’s recommended that you read the man pages for any command you are unfamiliar with. Some uses are :
    - man man – Provides information about the manual itself.
    - man intro – Displays a brief introduction to Linux commands.
17. **passwd** Ubuntu basic command is used to change user password using Terminal. What you have to do is run the command *passwd user,* where user is the username whose password has to change
18. **whatis** command shows a brief description of what is the functionality of specific built-in Linux command: *whatis cd*

# UNIX Crash Course

[Created by Sean Barker, available at: https://www.bowdoin.edu/~sbarker/unix/]

## Terminal Basics

Access the terminal with the terminal shortcut key (Ctrl+Alt+T Ubuntu).
You are greeted with the terminal window, which looks something like this:

```
elma@elma-Inspiron-7537:~$
```

The text showing in your terminal window is called the **command prompt** (usually ending in a % or $ depending on your system). The command prompt waits for you to enter a Unix command, which is then executed by pressing *enter*.
Any output that the command produces will be shown below the command prompt. After the command has been executed, another command prompt is displayed so that you can enter another command, and so forth. If you are familiar with the Python shell, this is the same idea.

### Escaping the Command Prompt

You won't need this right now, but almost certainly will at some point: if at any point you are in a terminal window and don't have a command prompt, but want or need one, type *Control-C* to interrupt the currently executing command and give you a fresh command prompt. An example where you might want this is if you executed a program that contains an infinite loop - since you normally don't get a new terminal prompt until the previous command has finished executing, you can use *Control-C* to terminate whatever is running and get back to the shell.

# Navigating the File System

When using the terminal, at any given point your terminal session "exists" within a specified directory. This is called your **working directory**. The output of many commands depends on your working directory. Initially, your working directory is your **home directory**, which is typically where all of your files are stored. Each user on a machine has a home directory, which is normally accessible only by that user and no others.

To see your current working directory, execute the *pwd* command (print working directory):

```
elma@elma-Inspiron-7537:~$ pwd
/home/elma
```

This says that my current (home directory) is the directory called 'sbarker', located in the directory called 'home'.

To see the files that are in your current directory, use the *ls* command (list files):

```
elma@elma-Inspiron-7537:~$ ls
```

You can also list the files in a directory *other* than your current directory by passing an argument to *ls*.

```
elma@elma-Inspiron-7537:~$ ls /home
elma
```

Note that while you can see all of the other home directories, you can't actually look inside any but your own. To see this, try to list the files inside my home directory (/home/elma).
Then look inside one of the directories in the same way.

```
elma@elma-Inspiron-7537:~$ ls /home/elma/Docear
projects
```

The terminal, when used appropriately, can actually save you time and be much more efficient than using a GUI interface to navigate the filesystem. However, if used improperly, you can waste a lot of time typing long commands over and over again. Here are some simple time-saving features that you should immediately get in the habit of using.

**Command History:** The first is accessing your command history using the arrow keys. Pressing the up arrow will autofill your terminal prompt with the last command you entered. Pressing up again will move back in time again to the 2nd most recent command you entered (and so forth). Similarly, pressing down moves forward in time in your command history. A very common scenario is either rerunning a command that you recently ran, or correcting a command that you ran but mistyped (and thus did not run correctly the first time). Don't waste your time retyping commands! Instead, use the arrow keys to access your terminal history.

**Tab-completion:** Another very useful feature that will help you type less is tab completion. Basically, whenever you're typing a filename or directory name (like /home/elma), pressing tab in the middle of entry will automatically complete as much of the name as possible with the name of a file or directory that already exists. For instance, if you type only /home/el (as an argument to a command like ls) and then hit tab, the path will autocomplete to /home/elma so long as no other directories exist in /home that start with the letters "el". Try this out. Pressing tab multiple times will display all files that match what you've currently entered, while leaving your existing command fragment intact. Get in the habit of pressing tab when entering file and directory names!

---

To move around in the filesystem (that is, to change your current working directory), use the *cd* command (change directory):

```
elma@elma-Inspiron-7537:~$ cd /home
elma@elma-Inspiron-7537:/home$
```

Now your current working directory is /home. Try running ls again (without any arguments). Now change back to your home directory.

## Relative vs Absolute Paths

Whenever you specify a filename or a pathname in a Unix command, you do so either using a **relative path** or an **absolute path**. An absolute path starts with a forward slash (/) and specifies all parent directories of the path in question, e.g., /home/elma is an absolute path.

A relative path, on the other hand, does not start with a forward slash, and refers to a pathname **relative to the current working directory**. So, for example, if I am currently located in /home and I try to cd to the directory named elma (NOT /home/elma), then I am saying I want to go to the directory named elma, *located within the current working directory*. Obviously, the file indicated by a relative path depends on the current working directory. In contrast, an absolute path does not depend on the current directory. If you cd to an absolute pathname, it will work the same way regardless of where you currently are in the filesystem.

In most cases, relative pathnames are used, as they usually involve less typing.

One special and important case of relative pathnames are the special directories 'dot' (.) and 'dot dot' (..). The . directory refers to the current working directory, while the .. directory refers to the *parent* directory of the current working directory. So, for example, if you are located in your home folder, running the following:

```
elma@elma-Inspiron-7537:/home$ cd ..
```

Would move you to the /home directory.

Now let's try making our own directories. First, *cd* back to your home directory if you aren't already there (tip: you can run the *cd* command without any arguments to go back to your home directory, regardless of where you currently are).

Now, let's make a new directory called unixworkshop using the *mkdir* command (make directory):

```
elma@elma-Inspiron-7537:~$ mkdir unixworkshop
```

Note again that here we're using a relative path -- we're saying to create a directory using the relative pathname *unixworkshop*, which creates a directory in our home directory since that's where we currently are. We could've specified the same thing by writing an absolute path:

```
elma@elma-Inspiron-7537:~$ mkdir home/elma/unixworkshop
```

but this would've been much more typing than necessary.

Now that we have a new directory, *cd* into it (remember to use tab completion!) and then run *pwd* to verify that you're in the right place.

## Working with Files

Let's start by copying an existing file into your new directory. You can copy files using the *cp* command, which has the basic form:

```
cp [sourcefile] [destfile]
```

This copies the first named file to the second second named file. If the second argument is an existing file, that file is overwritten. If the second argument is not an existing file, that file is created.

Get inside any directory where some files exist. Try copying an existing file to a new one (further referring to this file as file-you-copied)

Now that you've made a copy of the file, let's view the contents of your new copy. A convenient utility to read a file is called *less*, which takes a filename as an argument and opens the file for viewing in the terminal window.

```
less [file-you-copied]
```

Try opening the new copy using *less* to see what's in the file.

Inside the less interface, use j/k to scroll down/up (or you can use arrow down / arrow up, but getting used to j/k will save time). To quit and return to your command prompt, type *q*.

## Asking for Help!

Unix has lots of commands, and many of these commands also have lots of different command line flags and options. For example, try running ls with the command line flag -l (long listing), as follows:

```
ls -l
```

This will show the files in the current directory in an extended format that shows a lot more information, such as who owns the files, who has permissions to read the files, the file sizes, and when the files were last modified.

Due to the number of different commands and command-line options for many commands, it can be hard to keep track of them all! Luckily, there is a handy built-in manual that contains detailed information on how to use every individual command.

To view the the manual on a particular command (which most notably, describes all the command line flags that a command accepts), use the *man* command, e.g.:

```
man ls
```

The manual interface behaves just like the interface for less. If you are ever in doubt as to how to use a command, consult man!

Try this as an exercise: create another directory named dir1, copy [file-you-copied] into it, then copy the entire dir1 directory to dir2 (i.e., make an entire copy of the directory, including the enclosed file) using cp. You'll need to consult the manual for cp to determine how to copy a directory using cp (hint: copying a directory and everything in it is a recursive copy).

A command that's very similar to cp is the *mv* (move) command, which is called in the same way as cp but simply moves a file to the desired location (as opposed to making a copy there). This is also how you can rename a file - by simply calling *mv* with a different destination filename than the source filename.

```
mv [sourcefile] [destfile]
```

Try renaming your [file-you-copied] file to something.txt using *mv*. Check using the *ls* command that you successfully renamed the file.

Finally, to delete a file, us the *rm* (remove) command:

```
rm [file-to-delete]
```

**Be very careful with the rm command! We don't want you to lose important data!**

Try deleting dir1/file-you-copied (which you should've created earlier) using *rm*. Now try deleting the entire dir2 directory (which includes the other copy of the file). You might need to consult man again.

## Using a Command-Line Editor

So far we've just manipulated files that someone else created (or that we copied). Now let's try creating our own files! To create (or edit) files via the command line, we use a **command-line text editor**. This is an editor that has its entire interface within a Terminal window. There are many command-line text editors that have various pros and cons. The two most well-known command-line editors are Vim and Emacs, but we will use Nano. To start editing a new file using Nano, simply call the *nano* command with the desired filename of the new file to create.

```
nano myfile.txt
```

You are now in the nano interface, and can type just as you would in any other editor. Some commands are shown along the bottom of the window -- note that the caret symbol (^) refers to the control key. The most important commands in nano are ^O (control-O) to save the file and ^X (control-X) to quit nano. Nano includes many other commands for things like cutting/pasting text and searching through the file. A cheatsheet of Nano commands is here.

New users of command-line editors (including simpler editors like nano) are often frustrated at the lack of mouse control and familiar shortcuts found in most graphical applications. Channel this frustration into learning editor commands! These editors have commands to do nearly anything you might want to do, and usually faster than using a mouse in a conventional word processor. As a concrete example, novices often just scroll through an entire line from the end using the arrow keys when they want to go to the start of the line. This is slow, frustrating, and a waste of time! In Vim, this can be done with a single keystroke (and similarly efficient commands exist for other editors). Bottom line - don't settle for doing things slowly and awkwardly. If you find yourself repeatedly losing time on editor tasks, consult Google (or ask someone) how to do it faster!

Type some text in your new document, then save and quit the editor.

Verify the contents of your new file with the *cat* command (man cat!).

To open the now existing file and change it, simply run *nano myfile.txt* again, which will open up the existing file in the Nano window (hopefully you didn't type myfile.txt again and either used your terminal history or tab-completion on the filename!).

We've now covered the basics of navigating in a Unix command line environment. While there are many more commands than those described above, these should be enough to get you started working with the command line.

Before starting the next part, cd back to your home directory. You can also clear the contents of the terminal window using the *clear* command.

# Compiling and Running a Program

1. **Install GCC on Ubuntu**
   The default Ubuntu repositories contain a meta-package named build-essential that contains the GCC compiler and a lot of libraries and other utilities required for compiling software. Perform the steps below to install the GCC Compiler Ubuntu:
   a) Start by updating the packages list:
   ```
   sudo apt update
   ```
   b) Install the build-essential package by typing:
   ```
   sudo apt install build-essential
   ```
   The command installs a bunch of new packages including gcc, g++ and make.
   c) You may also want to install the manual pages about using GNU/Linux for development:
   ```
   sudo apt-get install manpages-dev
   ```
   d)To validate that the GCC compiler is successfully installed, use the gcc --version command which prints the GCC version:
   ```
   gcc --version
   ```

Now let's try compiling and running a C program using the command line. First, in your home directory, create a new directory called *myprogram.*

Cd into that directory and create a new file called **hello.c**. Type the following Hello World program into your new file:

```
#include <stdio.h>

int main() {

  printf("Hello World!\n");

  return 0;

}
```

Now let's compile the program by calling the standard C compiler program, *gcc*:

```
gcc -Wall -o hello hello.c
```

The *gcc* command takes a list of source files to compile (in this case, just hello.c) and outputs the compiled executable (or a list of errors if the program does not compile). In the above command, we are passing two flags in addition to the filename: -Wall (Warnings: all) says to turn on all compiler warnings, and -o hello says we want the output executable file to be named hello. If we omit the -o hello option, then gcc defaults to producing an executable named a.out (not very informative).

Note that if we are writing C++ code instead of plain C code, everything is exactly the same except that we use the g++ command instead of gcc.

To run the compiled executable, we run the executable name as a command:

```
./hello
```

Note that the ./ indicates that we're running a program located in the current directory.

## General GCC Tips

**Always compile using the -Wall flag**. This flag essentially instructs the compiler to give you as much programming feedback as possible, and will often output warnings about things that aren't strictly wrong, but are still symptoms of bugs in your code. Let the compiler help you as much as it can! If you use a Makefile (see below), then you can easily automate including this flag so you don't have to worry about forgetting it.

Also, get in the habit of **treating all warnings like errors**, even if the warnings only appear when using the -Wall flag and not without. Warnings are very often symptoms of bugs, and even if they aren't, they're usually an indication of bad programming style or gaps in understanding. Never be satisfied by a program that produces warnings, even if it compiles in spite of them! Fix all warnings and only then try running the program.

# Using a Makefile

Typing compilation commands over and over each time you change your program wastes a lot of time (and encourages you to leave off things like *-Wa*ll, which you should never do). As our programs get more complicated and involve multiple source files, the compilation command gets longer and this problem gets worse. Since our objective in using the terminal is to be ~~lazy~~ efficient, we can automate the build process by using a special file called a *Makefile*. This is a file used by the program *make* that says how to compile your program.

Create a file called *Makefile* alongside your source file and input the example contents below.

```
CC = gcc

CFLAGS = -Wall

hello: hello.c

    $(CC) $(CFLAGS) -o $@ hello.c

clean:

    rm -f hello
```

**Note that the two indented lines must be indented with actual tabs (not spaces).** If you copy and pasted from above, you'll likely need to fix this.

A *Makefile* consists of a set of **targets**, each of which tells *make* how to do something. The example above contains two targets: *hello*, which says how to compile the *hello* executable, and *clean*, which says how to clean up after the compiler by deleting generated files (in this case, just the *hello* executable). Each target has a set of commands that are executed when make is run like so:

```
make [target]
```

For example, running *make clean* would execute the *clean* target, running the *rm* command to delete the compiled executable. If you run *make* by itself (i.e., without any argument), it defaults to building the first target (which is hello in this case).

Run *make clean* to delete the existing executable (if it exists), then run *make* to build the *hello* executable.

One of the reasons *make* is useful is that it only compiles files when it actually needs to -- i.e., only if the corresponding source files have actually changed. This is the significance of the *hello.c* located on the same line as the *hello* target. This is called a **dependency** and tells *make* to recompile if and only if the file *hello.c* has changed since the last compilation.

Try running *make* again. What happens? Now edit your source code (say by changing the message that prints) and run *make* again.

This is the basic idea behind Makefiles -- rather than having to repeatedly type long compilation commands, we just type *make* to compile the program. For large projects with many files and dependencies, Makefiles can get very complicated and scary looking. Focus on just producing simple

Makefiles that do what you need and try not to get bogged down in advanced features of make that you don't need! For the masochists among us, here is the make online documentation.

---

## Using Multiple Terminal Windows

A very common scenario in programming is that you are using a command-line editor to edit your source code, and periodically recompiling by running make and then executing the program to test. If you are doing this all in one terminal window, this is very time consuming, since you are constantly having to quit your editor to rebuild and run, then going back into the editor to make more changes.

Instead, you will have a much easier time if you open up a second terminal window. Whenever you open up a new Terminal window, you will be located in the home folder of the local machine. You can then use one terminal window to keep your source code open in the editor, and the other terminal window to compile and run your code. Note that each terminal window is independent and has its own working directory, so remember you'll need to *cd* to the appropriate location when you open the second window.

Multiple terminal windows in general let you avoid switching back and forth between servers, working directories, etc as much, so be liberal in opening new windows.

---

# Command-Line Arguments

The standard way that information is passed to programs executed on the command line is via *command-line arguments*. For example, consider running a command to copy the file foo.txt to the file foo2.txt:

```
cp foo.txt foo2.txt
```

Here, the name of the program is cp, while foo.txt and foo2.txt are command-line arguments. The cp program is given these arguments and acts accordingly. Most programs expect command-line arguments to run, and may behave differently depending on how many arguments are given. For example, the normal use of the cd command is with one argument specifying the directory to change to, but you can also call cd without any arguments, which will change to your home directory regardless of your current working directory.

Many of the command-line programs that you write will also use command-line arguments. A program can read the arguments that it's passed using the *argc* and *argv* parameters to the main function that you may recognize. Here is an example of a C program that prints out the number of arguments that it is given and then prints out the first such argument (if it exists):

```
#include <stdio.h>


// a test of command-line arguments

int main(int argc, char** argv) {

  printf("program was called with %d arguments\n", argc - 1);

  if (argc > 1) {

    printf("the first argument is %s\n", argv[1]);

  }

  return 0;

}
```

Save this program as *argtest.c* and compile it with *gcc* (remember to use -Wall and -o to specify the name of the compiled executable). In the following example, I'll assume the compiled program is named argtest:

```
sbarker@dover$ ./argtest bowdoin computer science

program was called with 3 arguments

the first argument is bowdoin
```

This program demonstrates the behavior of the main function parameters - argc is the number of command line arguments passed to the program, and argv is the arguments themselves (represented as an array of char* objects, i.e., essentially a bunch of strings).

You hopefully noticed something odd in the above program - namely, that we printed argc - 1 and argv[1] instead of argc and argv[0], respectively. The reason for this is that the way command-line arguments are defined, the first argument is *the name of the program itself* - i.e., argtest in the above example. In other words, a program will **never** have an argc value of zero, since the name of the executing program will always be available as argv[0]. Since we don't really think of the name of the program itself as a true command-line argument, however, we are excluding it from consideration. In the above example, the actual value of argc is 4 (not 3).

While the argc and argv parameters of the main function are optional (note that we did not include them in the earlier Hello World example), you must include them if your program needs to make use of command-line arguments.