

# OS Assignment 1 Walkthrough

---

Lead TAs: Monum, Shahpar and Mashal

# Introduction to Linux Kernel Modules

# What is a Kernel Module?

It is an object file that contains code that can extend the kernel functionality at runtime. It can be loaded and unloaded on the kernel on demand.

This means that while creating Kernel Modules in this assignment you will be interacting directly with the kernel and hence any errors could crash the system :)

**Solution???**

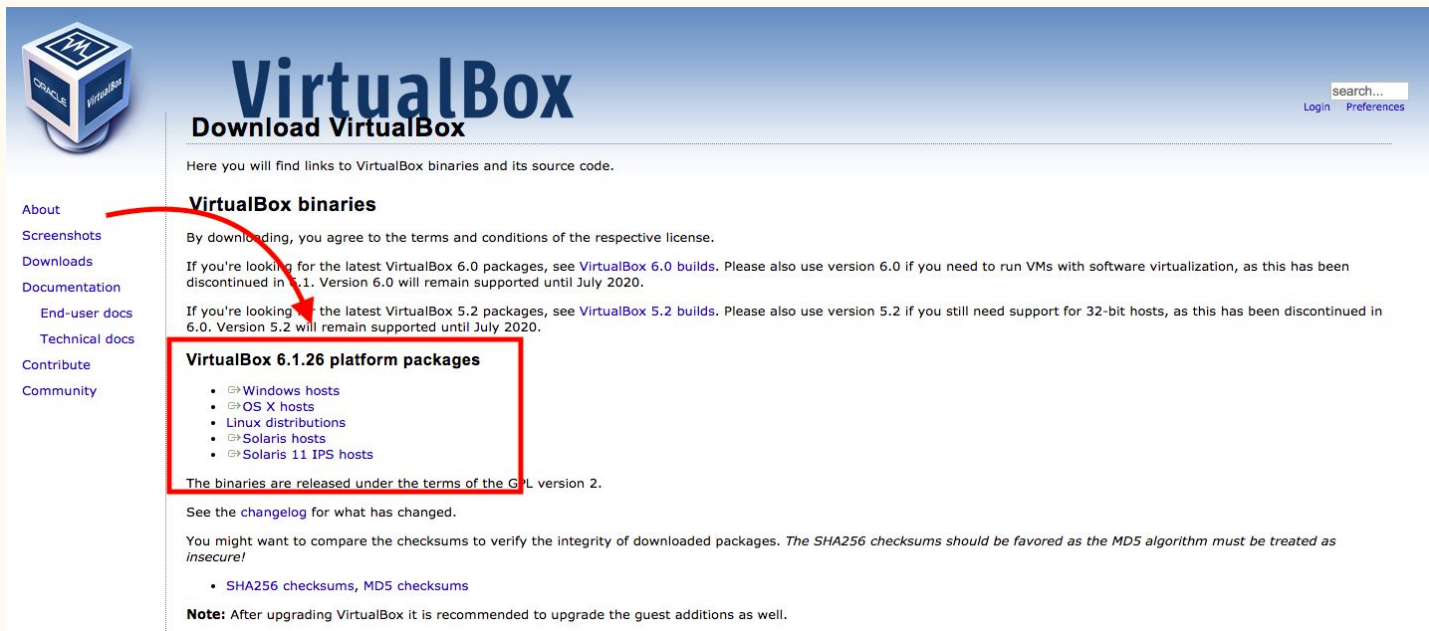
We will use Virtual Machines!!!

# VM Set-Up

—

# Downloading the Virtual Box

Link: <https://www.virtualbox.org/wiki/Downloads>



The screenshot shows the VirtualBox website's download page. On the left is a sidebar with navigation links: About, Screenshots, Downloads, Documentation, End-user docs, Technical docs, Contribute, and Community. The main content area has a header with the VirtualBox logo and the text 'Download VirtualBox'. Below this, it states 'Here you will find links to VirtualBox binaries and its source code.' A red arrow points from the 'Downloads' link in the sidebar to the 'VirtualBox binaries' section. This section contains information about the latest versions (6.0 and 5.2) and a list of platform packages for VirtualBox 6.1.26, which is highlighted with a red box. The list includes Windows hosts, OS X hosts, Linux distributions, Solaris hosts, and Solaris 11 IPS hosts. Below the list, it mentions the binaries are released under the terms of the GPL version 2. At the bottom, there is a changelog link and a note about comparing checksums (SHA256 and MD5) to verify package integrity.

**VirtualBox**  
Download VirtualBox

Here you will find links to VirtualBox binaries and its source code.

**VirtualBox binaries**

By downloading, you agree to the terms and conditions of the respective license.

If you're looking for the latest VirtualBox 6.0 packages, see [VirtualBox 6.0 builds](#). Please also use version 6.0 if you need to run VMs with software virtualization, as this has been discontinued in 5.1. Version 6.0 will remain supported until July 2020.

If you're looking for the latest VirtualBox 5.2 packages, see [VirtualBox 5.2 builds](#). Please also use version 5.2 if you still need support for 32-bit hosts, as this has been discontinued in 6.0. Version 5.2 will remain supported until July 2020.

**VirtualBox 6.1.26 platform packages**

- [Windows hosts](#)
- [OS X hosts](#)
- [Linux distributions](#)
- [Solaris hosts](#)
- [Solaris 11 IPS hosts](#)

The binaries are released under the terms of the [GPL version 2](#).

See the [changelog](#) for what has changed.

You might want to compare the checksums to verify the integrity of downloaded packages. *The SHA256 checksums should be favored as the MD5 algorithm must be treated as insecure!*

- [SHA256 checksums](#), [MD5 checksums](#)

**Note:** After upgrading VirtualBox it is recommended to upgrade the guest additions as well.

# Downloading the .ova file

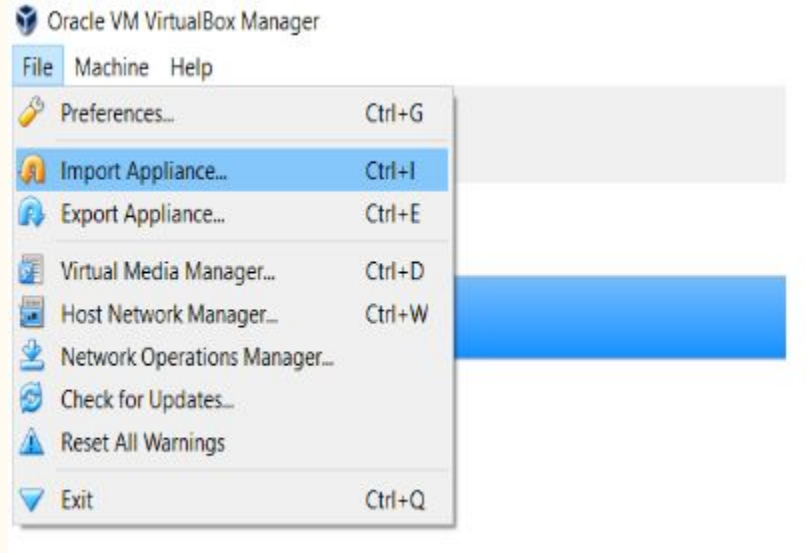
OVA file: Open Virtualisation Appliance that contains a compressed “installable” version of a Virtual Machine that can then be imported into the Virtual Box.

## **Link:**

[https://pern-my.sharepoint.com/:u:/g/personal/basit\\_lums\\_edu\\_pk/EQEeikkBZwBLmhwwysbOSHaMBn0FOei\\_UyqfLkOaI9Y4qOQ?e=Tbqy6v](https://pern-my.sharepoint.com/:u:/g/personal/basit_lums_edu_pk/EQEeikkBZwBLmhwwysbOSHaMBn0FOei_UyqfLkOaI9Y4qOQ?e=Tbqy6v)

# Importing

1. Open the virtual box and go to “File -> Import Appliance” in the menu bar.

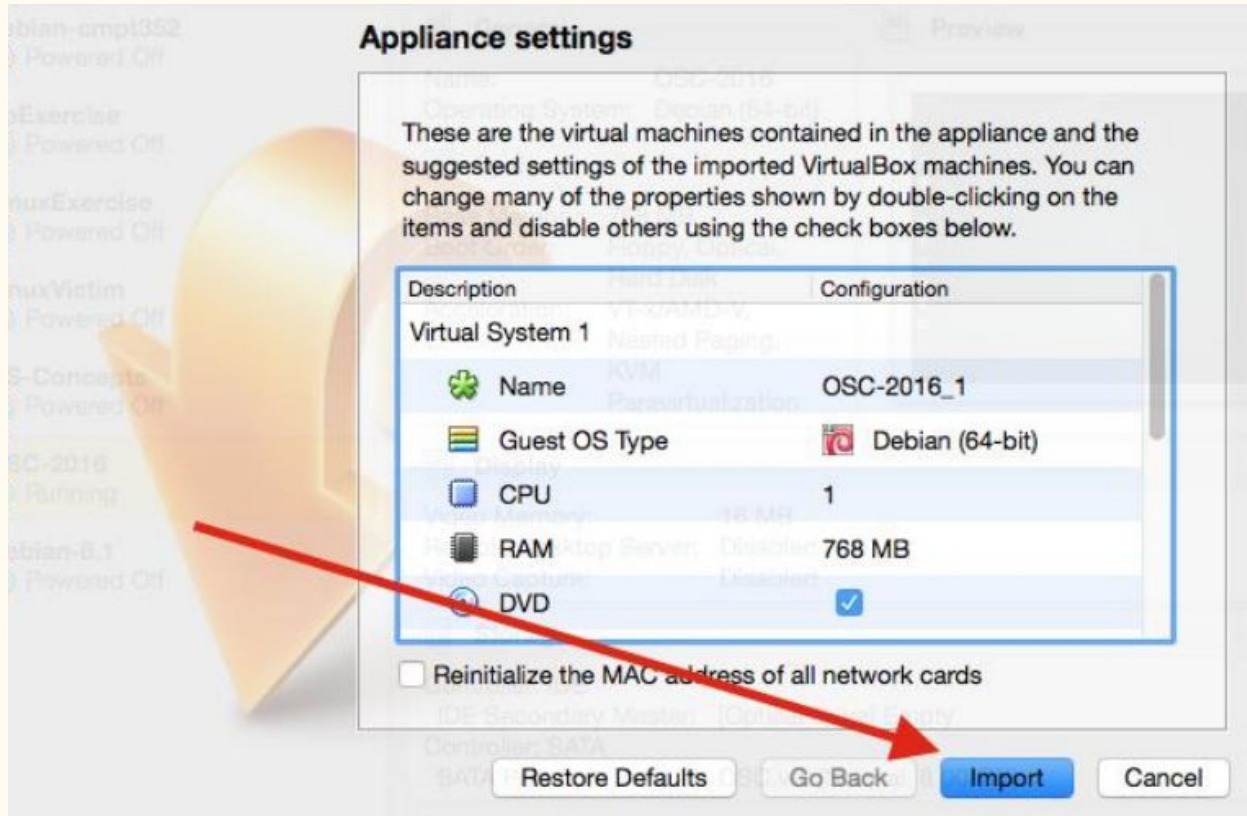


# Adding the path to the ova file

In the dialogue box, add the path to the ova file. The window will then show you the configuration of the current virtual appliance. You can scroll through the configuration list and double click on any item (or check/uncheck the box) to make changes to it. Consider increasing the dedicated RAM for a better experience. Lastly, click “Import”.



# Adding the path to the ova file

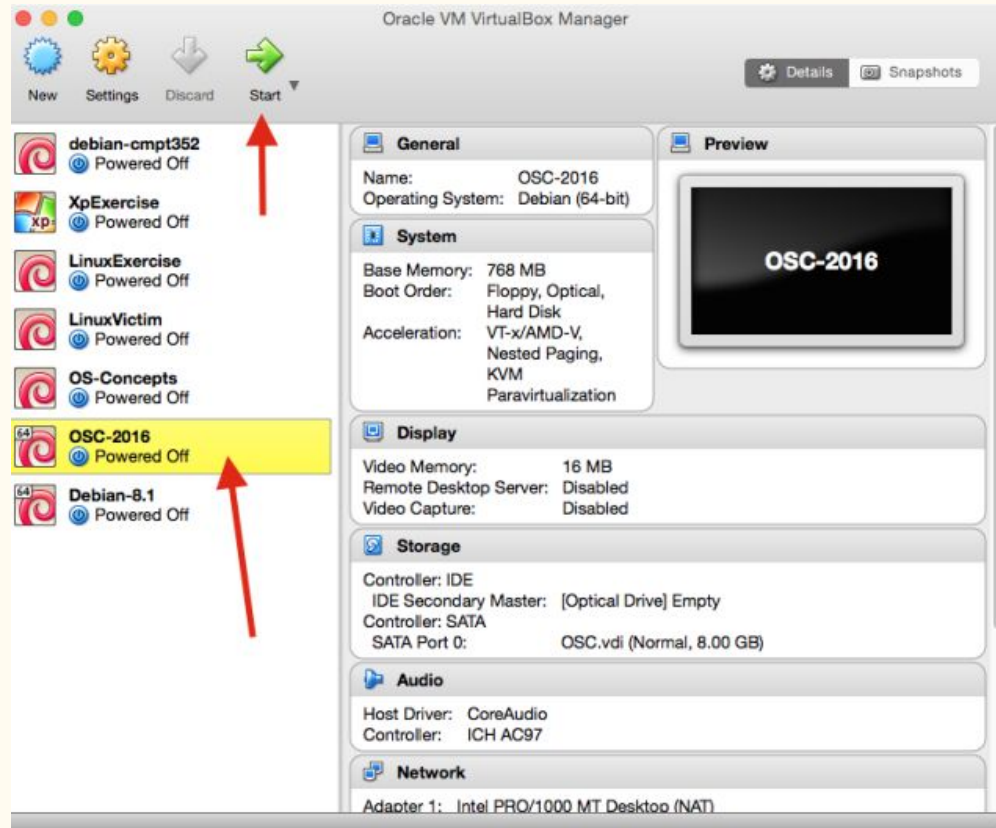


# And done!

VirtualBox will proceed to import the virtual machine into your library. Depending on the size of your OVA file, the import process could take quite a while. Once the process is completed, you should be able to see the new VM in your list

# Running the VM

Now you can run an instance of the given OS in the virtual box by clicking on start. The password for the machine is **osc**.



# Downloading Assignment Code

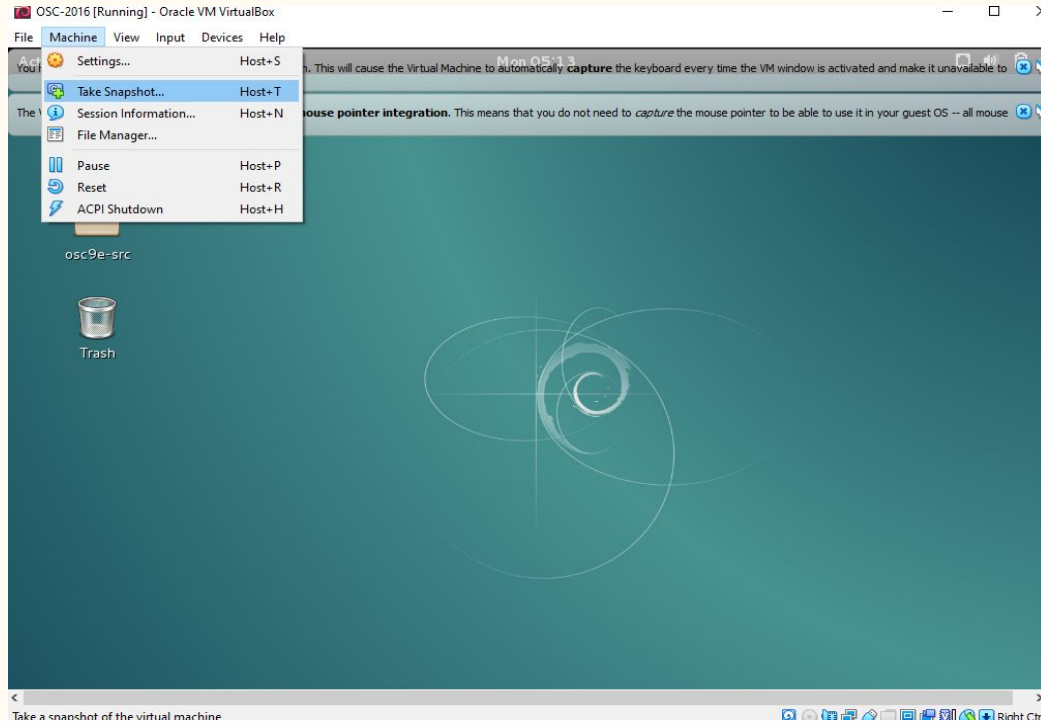
- Once the OS starts, open a terminal (Ctrl+Alt+T or Right click -> Open terminal).
- On the terminal type:  
wget <http://cs.westminstercollege.edu/~greg/osc10e/final-srcosc10e.zip>
- This will download a zip file called “final-src-osc10e.zip”.
- Extract this file (right click->extract here). You will see a folder named “final-srcosc10e”. All the code you need to edit for the first assignment will be in this folder.

# VM snapshots

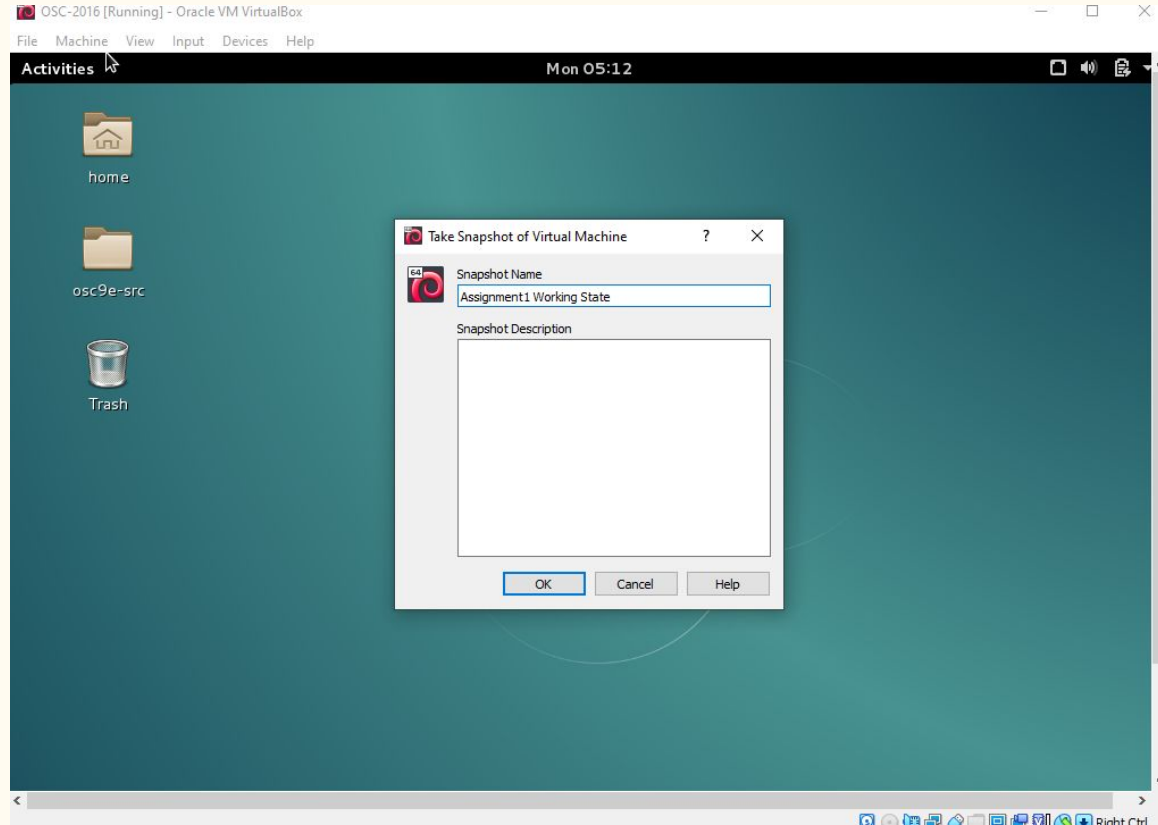
- The basic idea of a snapshot is that you setup your virtual machine exactly how you want it, take a snapshot, and then you can make any changes you want. You could even install something awful, because it doesn't matter – all you have to do is roll back the snapshot, and your virtual machine will be exactly how it was before.
- This is particularly helpful if your kernel code led to hanging or corrupting the VM.

# Taking snapshot

- Click the Machine menu item, then select Take Snapshot...



# Name your snapshot



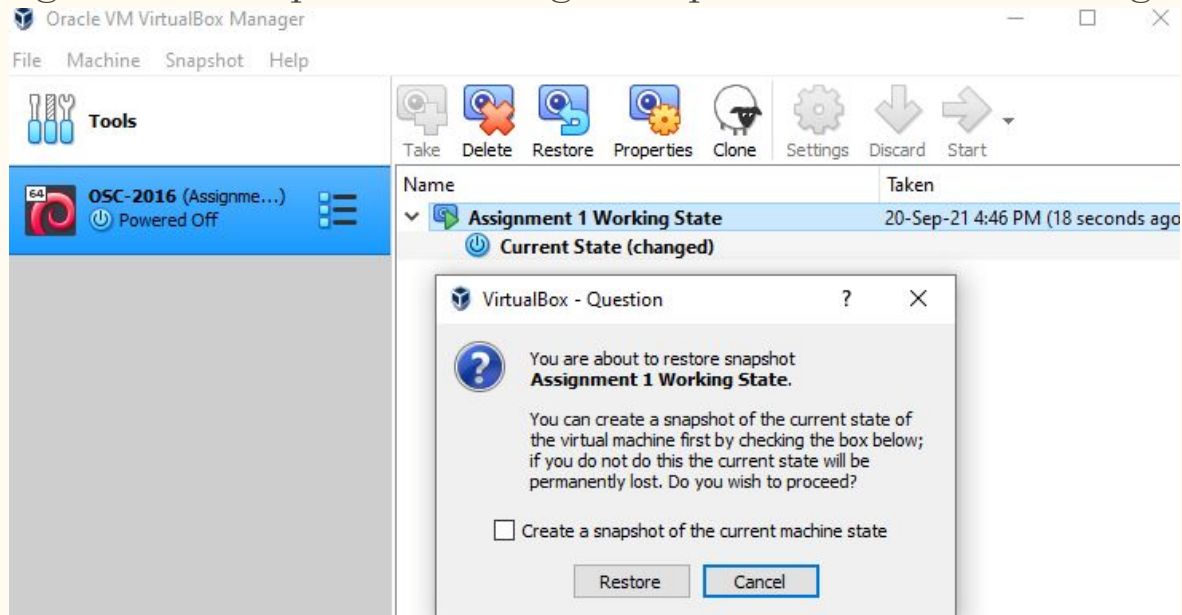
# Reverting to your snapshot

- The purpose of reverting to a snapshot is so that you can go back in time to a particular state, in our case a clean state just after we installed the OS.
- When your VM is powered off, select your machine, right click on the Snapshot Name, and select **Restore Snapshot**.



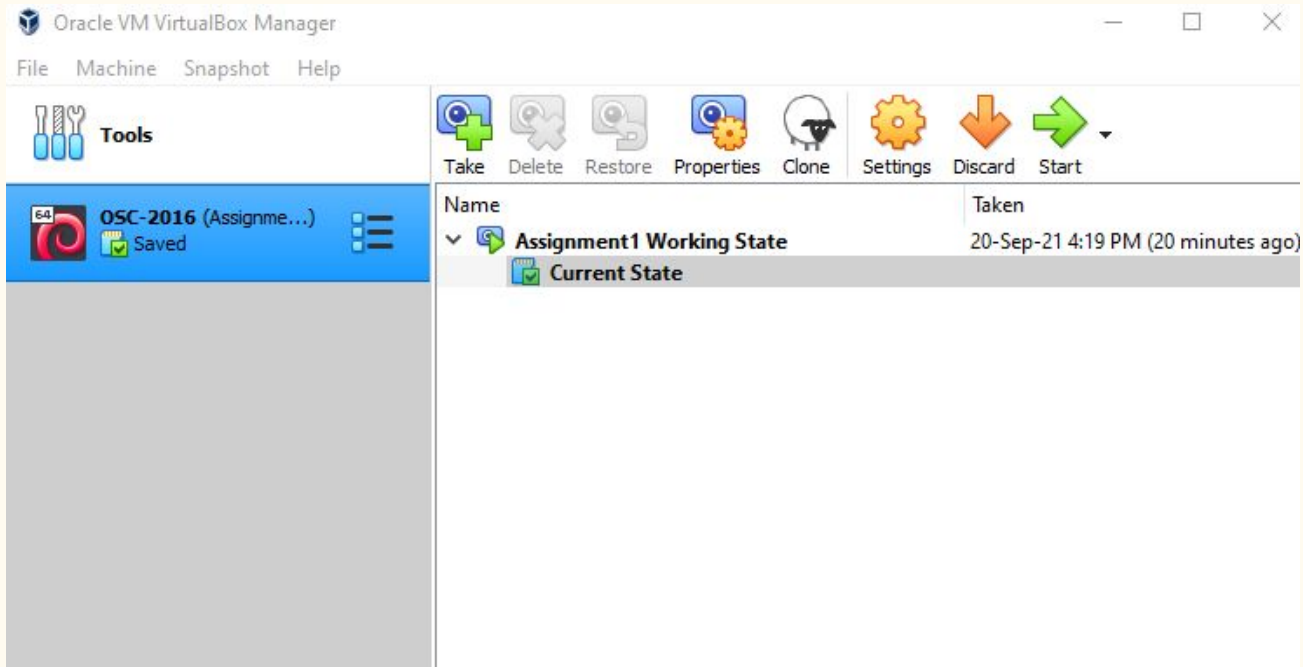
# Reverting to your snapshot

- Uncheck the option to create a snapshot of the virtual machine current state. The reason being is that you will normally want to restore when you have broken something, there's no point in taking a snapshot of a broken configuration.



# Reverting to your snapshot

You will then see that the “Current State” will become the same as the snapshot you selected to restore to.



# Hardware Constraints

- Some laptops may not be able to handle running a VM due to limited RAM or may face lags
- SSE labs equipped with Virtual Box and you can import the snapshots of your VM (from a USB) and work from there
- Remote access can also be granted
- Email IST with details of the course whether you would like remote access or lab access
- Act Fast!

# Linux Commands and Kernel Modules

—

# Handy Commands

**lsmod:** Lists down loaded kernel modules

**make:** Invokes the execution of the makefile

**dmesg:** To check the messages in the log buffer

**sudo dmesg -c:** To clear the kernel buffer (it fills up very quickly)

**sudo:** “Super user do” It allows you to run the program with the privileges of the superuser

**sudo insmod simple.ko:** Loads the kernel module simple.ko

**sudo rmmod simple.ko:** Removes the kernel module simple.ko

**cat:** concatenate, usually used to concat the contents of the file to the terminal (view file content)

**nano:** command-line text editor.

# Example Kernel Module

- `simple_init()` is the module entry point, this function is invoked when the module is loaded into the kernel
- `simple_exit()` is the module exit point, it is invoked when the module is removed from the kernel
- `printk()` is the kernel equivalent of `printf()`, `printk()` allows for priority flags such as `KERN_INFO`

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
    printk(KERN_INFO "Loading Kernel Module\n");

    return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
    printk(KERN_INFO "Removing Kernel Module\n");
}

/* Macros for registering module entry and exit points. */
module_init(simple_init);
module_exit(simple_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

# Commands to load Module

```
make
```

```
sudo insmod simple.ko
```

```
dmesg
```

```
sudo rmmod simple
```

# Assignment Tasks

—



# Task 1

- Print out the `GOLDEN_PRIME_RATIO` in the `simple_init()` function
  - The `GOLDEN_PRIME_RATIO` is defined in `linux/hash.h`
- Print out greatest common divisor of 3300 and 24 in the `simple_exit()` function
  - `linux/gcd.h` contains the function:
    - `unsigned long gcd(unsigned long a, unsigned long b)`
- You will take the screenshot of the `dmesg` after inserting and removing the above kernel module

# Task 2

- Print out the values of jiffies and HZ in the simple\_init() function
- Print out the values of jiffies in the simple\_exit() function
- You will take the screenshot of the dmesg after inserting and removing the above kernel module

## What are HZ and jiffies?

- HZ is defined in asm/param.h, determines the frequency of the timer interrupt
- Jiffies is defined in linux/jiffies.h and maintains the number of timer interrupts since the system was booted

How can these be used together? (Food for thought)

# Task 3

## /proc File System

- Virtual file system created at the time of system boot and dissolved on shut down
- Contains important information/stats of the processes running

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "hello"

ssize_t proc_read(struct file *file, char __user *usr_buf,
                  size_t count, loff_t *pos);

static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
};

/* This function is called when the module is loaded. */
int proc_init(void)
{
    /* creates the /proc/hello entry */
    proc_create(PROC_NAME, 0666, NULL, &proc_ops);

    return 0;
}

/* This function is called when the module is removed. */
void proc_exit(void)
{
    /* removes the /proc/hello entry */
    remove_proc_entry(PROC_NAME, NULL);
}
```

```

#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "hello"

ssize_t proc_read(struct file *file, char __user *usr_buf,
    size_t count, loff_t *pos);

static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
};

/* This function is called when the module is loaded. */
int proc_init(void)
{
    /* creates the /proc/hello entry */
    proc_create(PROC_NAME, 0666, NULL, &proc_ops);

    return 0;
}

/* This function is called when the module is removed. */
void proc_exit(void)
{
    /* removes the /proc/hello entry */
    remove_proc_entry(PROC_NAME, NULL);
}

```

```

/* This function is called each time /proc/hello is read */
ssize_t proc_read(struct file *file, char __user *usr_buf,
    size_t count, loff_t *pos)
{
    int rv = 0;
    char buffer[BUFFER_SIZE];
    static int completed = 0;

    if (completed) {
        completed = 0;
        return 0;
    }

    completed = 1;

    rv = sprintf(buffer, "Hello World\n");

    /* copies kernel space buffer to user space usr_buf */
    copy_to_user(usr_buf, buffer, rv);

    return rv;
}

module_init(proc_init);
module_exit(proc_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");

```

# Task 3

- Design a Kernel Module that creates a /proc file names proc/jiffies that reports the current number of jiffies when the file is read with the command:
  - `cat /proc/jiffies`
- Design a Kernel Module that creates a /proc file names proc/seconds that reports the number of seconds since the file was loaded till it was read with the command: (You will be using jiffies and HZ in this)
  - `cat /proc/seconds`
- You will take the screenshot of the dmesg after inserting and removing the above kernel module

# Makefile

- Is a special file containing shell commands that is named makefile/Makefile (depends on system)
- Defines a set of tasks to be executed
- Running the command “make” will execute the makefile (ensure that you are in the same directory as the makefile)
- [Useful link](#)

```
obj-m += simple.o
```

```
all:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

```
clean:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```



# Intro to git



# What is git?

- It is a version control system that lets you track and manage the changes in your source code

## Why use git?

- Traceability
- Efficiency
- Merging and Branching
- History





# Git Intro

- Github provides an internet hosting platform for development and version control using git
- Install git and create a github account

# Create a local git repo

- A Repository/repo contains all your project's folders and files
- In your project directory you will type the following command on the terminal:

```
git init
```

- To know if git recognizes the files in your directory you can type the following command:

```
git status
```

# Initializing GitHub repo

- LogIn to GitHub and create a new repository
- Now you need to connect your local repo to your GitHub repo
- You will do the following command in your directory:

```
git remote add origin https://github.com/moshma/mynewrepository.git
```

```
git branch -M main
```

# Committing your code

- Whenever a new file is added to the repo it will be untracked by git, you will need to add it by using the following command (the “.” refers to all the files in the directory)

```
git add .
```

- Now we can commit our code meaning

```
git commit -m “Task1 done”
```

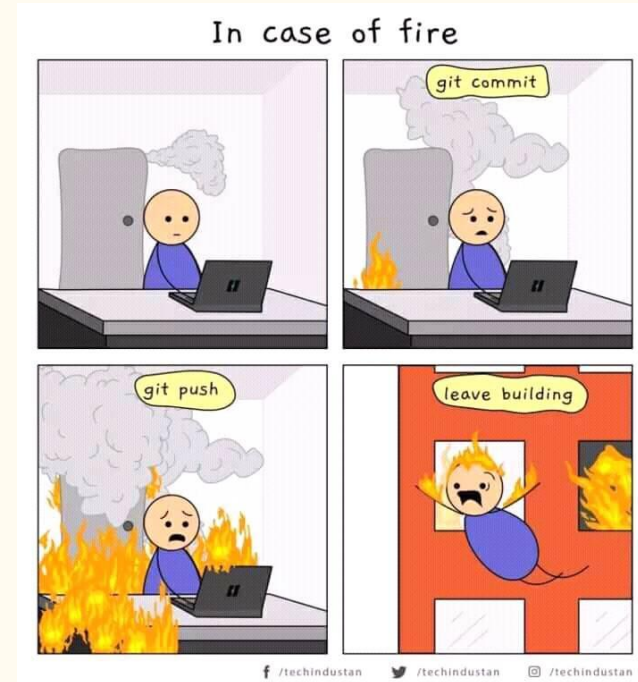
Always remember to add meaningful messages to your commit

# Pushing your code on GitHub

- The last step is to push your code onto your remote repo

```
git push -u origin main
```

Fun helpful resource: <https://dangitgit.com/>



Easy assignment  
Start Early :)

