# Linux Kernel

* Core component of Linux OS
* Responsible for managing system resources, providing low-level services to other parts of OS, and controlling hardware devices

## Monolithic and Modular

**Monolithic**: All system-level services are contained within a single executable file  
*Benefits*: Improved performance and simplified system management

**Modular**: The kernel **also** includes a modular design

* Certain features can be compiled as loadable kernel modules
* Modules can be dynamically loaded or unloaded at runtime
* Provides greater flexibility and allows for customization
* Example: Adding support for hardware or file system. Dev’s can experiment without committing to main kernel source code = More flexibility, greater customization

Linux Virtual Memory

* When a Linux program allocates memory, this memory initially doesn’t exist, it’s an entry in a table of the OS.
* Only when program accesses the memory is the RAM for it found and used.
* “Memory usage”

1. How much virtual memory it uses overall

2. How much actual or ‘resident’ memory is uses, limited to systems RAM capacity + swap

## Linux Kernel Organization

* Software residing in memory that tells CPU where to look for its next task
* Acting as mediator, kernel manages the hardware, and is primary interface between hardware and any running program
* (User) processes, managed by kernel, make up user space

## Features and Services of Kernel

* **Process management**: manages all running processes on system, allocating resources and scheduling CPU time
* **Memory management:** Manages (de)allocation of system memory, as well as implementation of virtual memory
* **File system Management:** Provides File system interface for managing multiple file systems
* **Device Management:** Controls access to hardware devices, such as disk drives, network adapters, I/O devices
* **Network Management:** Provides networking stack

## User Space and User Processes

* Kernel run in **kernel mode**, user processes run in **user mode**
* User mode restricts access to a small subset of memory and safe CPU operations. Refers to parts of main memory that user processes can access. If mistakes are made, consequences are limited and can be cleaned up
* Kernel mode has unrestricted access to processor and main memory. Powerful but dangerous privilege that allows the kernel to easily corrupt and crash entire system. Memory area that only kernel can access is called kernel space

## Linux Kernel on CPU architecture

* Many modern CPU architectures include ‘ring’ protection
* Kernel space has highest authority
* User space can only access restructured resources and cannot directly access hardware devices such as memory. It must be trapped in the kernel through system calls to access these privileged resources
* Privilege rings provide way for CPU to switch between different levels of privilege during system operation
* When user process makes a system call or requests privileged operation, such as accessing a protected system resource or modifying system settings, CPU switched from ring 0 to perform requested operation. Once complete, CPU switched back to ring 3 to resume normal user-level operation

## Linux Virtual File System

* VFS provides unified view of file system to apps & kernel
* Abstracts underlying file system details for consistent interface
* File systems mounted at specific mount points in file system hierarchy
* VFS caches recently accessed files/dirs. In memory for performance improvement
* Supports various file system types: local, network, special:
* **procfs:** virtual view of running system, allows access/modification of sysinfo & config parameters
* **sysfs:** virtual view of systems hardware devices & drivers, allows view/modification of device attributes & settings
* Provides flexible & extensible framework for managing files/directories in various file system types & locations

# Kernel Process Management

* Starting , pausing, resuming, scheduling, and termination of processes
* Each process uses the CPU for a small time, then pauses, then the next process does the same, and so on.
* **Context switch:** The act of one process giving up control of the CPU to another process
* **Time slide:** Each piece of time gives a process enough time for significant computation
* Kernel also provides **System calls** for managing processes, such as creating new processes, changing process priority, and terminating processes

## Processes

* Process is an instance of a program that is currently executing
* Each process has its own memory space, program code, and execution context, including program counter, stack, and other registers
* Can communicate with each other using various interprocess communication (IPC) mechanisms, such as pipes, sockets, and shared memory segments
* Are created using the fork system call, which creates a new process by duplicating current process.
* New process is called child process, has unique PID
* Fork system call creates a copy of parent process, including its memory and state, then sets PID of child process to a new value
* Child process van then execute a new program of perform other tasks independently of parent process
* Processes can also be created using other system calls, such as exec, which replaces the current process with a new program or clone, which creates a new process with shared memory and other resources

## Types

* Init process:First process that gets started when Linux OS boots up, and it becomes parent process for all other processes
* Parent and Child Processes**:** Each user process has a parent process in the system, most commands having shell as their parent
* Orphan Process**:** When a child process is killed terminated, parent process is updated about it through SIGCHLD signal. When parent process is killed before termination of child process, child process becomes an orphan process with “init process” as its new PID
* Zombie Processes: A process which is killed but still shows its entry in the process status/table. They are dead and are not used
* Daemon Processes**:** Are system-related background processes that run with root permissions and wait for requests from other processes. Often run in background and can work with other processes.

## States & Signals

* Signals are a form of IPC used to notify a process of an event or condition
* Signals can be sent to a process by another process, by the kernel, or by the process itself
* Each signal has a unique number that identifies it, such as sigterm(15) or sigkill(9)
* Signals can be sent using commands or other IPC mechanisms such as sockets or pipes
* When a process receives a signal, it can either ignore the signal, handle it, or perform default action associated with the signal
* Signal system call allows a process to specify a custom signal handler function to be executed when a particular signal is received
* Common use cases for signals include graceful termination of a process, handling user interrupts, and sending status updates between processes

## fork() and exec()

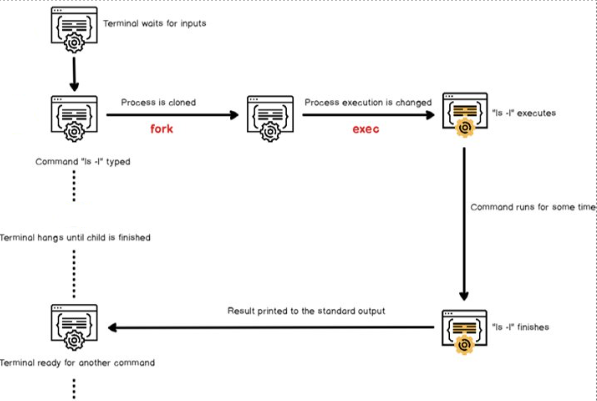
### fork()

* Is a clone operation, takes current process, also called parent process, and clones it to a new process with a new PID
* When forking, everything is copied from parent process: the stack, the heap, but also file descriptors meaning standard I/O, and standard error
* It means if parent process was writing to the current shell console, child process will also write to the shell console

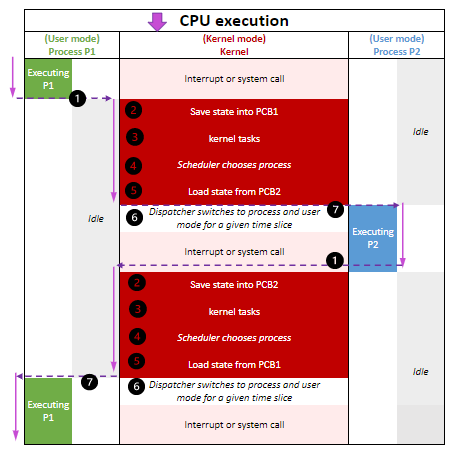
### exec()

* Execute program is used on Linux to replace current process image with image from another process

## Shell Command Execution

* Shell console is a process that waits for input from the user
* Also launches a bash interpreter when you hit Enter, and it provides an environment for your commands to run
* When you type a command and hit Enter, the **shell is forked to a child process** that will be responsible for running your command. Shell will wait until execution of child process is finished.
* On the other hand, **child process is linked to the same file descriptors** and it may share variables that were declared on a global scope
* The child process executes **exec** command in order to replace current process image (shell process image) in the process image of the command you are trying to run
* The child process will eventually finish and it will print the result to the standard output it inherited from the parent process, in this case the shell console itself.

## Kernel: Process MGMT – context switching



1. CPU (actual hardware) interrupts the current process based on internal timer, switches into kernel mode, and hands control back to kernel
2. Kernel records current state of CPU and memory, which will be essential to resuming the process that was just interrupted
3. Kernel performs any tasks that might have come up during the preceding time slice (such as collecting data from I/O or I/O operations)
4. Kernel is now ready to let another process run. Kernel analyzes list of processes ready to run and chooses one
5. Kernel prepares the memory for this new process and then prepares the CPU
6. Kernel tells the CPU how long the time slice for new process will last
7. Kernel switches the CPU into user more and hands control of the CPU to the process

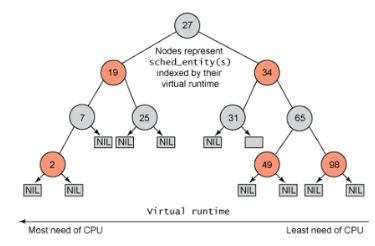
PCB: Process Control Block: The state of a process with program counter(PC), stack pointer (SP), status of opened files

**Context switch answer the important question of when the kernel runs. Answer: it runs between process time slices during a context switch.**

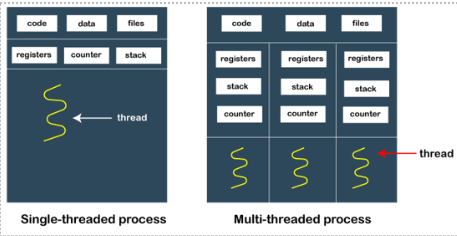
# The Linux Scheduler

* Responsible for deciding which process to run next
* Uses a variety of factors to make informed decisions, including process priorities, CPU utilization, process states, and scheduling classes
* Each process is assigned a priority value, which ranges from -20 (highest priority) to 19 (lowest priority)
* Scheduler uses priority-based scheduling algorithm, but also takes into account other factors to ensure system resources are used efficiently
* Scheduler can dynamically adjust priorities based on system load and other factors, e.g. process affinity and process real-time constraints
* Linux supports multiple scheduling classes, each with its own scheduling algorithm, to optimize for different types of workloads
* **Completely Fair Scheduler** (CFS) is the default scheduler in most Linux distributions, which uses a fairness algorithm to ensure that all processes receive an equal share of the CPU time

## CFS

* Linux scheduler manages groups of threads, multi-threaded processes, and all the processes of a given user
* Tasks are grouped and managed by scheduler as schedulable entities
* Each per-CPU run-queue of processes sorts schedulable entities structures in a time-ordered fashion into a ‘red-black’ tree.
* The leftmost node in the tree is occupied by the entity that has received the least slice of execution time, indexed by processor ‘execution time’ in nanoseconds
* When the scheduler is invoked to run a new process, it selects the leftmost node in the scheduling tree to execute
* Each process has a maximum execution time that represents the time the process would have expected to run on an ideal processor
* When a process is stopped or reaches its maximum execution time, it is reinserted into the scheduling tree based on its newly spent execution time
* The new leftmost node is selected from the tree for execution
* If a process spends a lot of its time sleeping, its spent time value is low, and it automatically gets the priority boos when it finally needs it.

## Threads

* Threads are lightweight execution contexts that share memory and code within a parent process
* For example, multiple tabs can be different threads
* Each thread has its own stack and program counter (PC), but all threads within a process share the same heap and global variables
* Threads allow a program to perform multiple tasks simultaneously
* Threads within a process can communicate with each other using shared memory or other IPC mechanisms
* Threads can synchronize their actions using mutexes, semaphores, and other synchronization primitives.

## Advantages of Threads

* **Responsiveness**: Process divided into multiple threads, if one thread completes the execution, then its output can be immediately returned
* **Faster context switch**: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU
* **Effective utilization of multiprocessor system**: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors.
* **Resource sharing**: Resources like code, data, and files can be shared amount all threads within a process.
* **Communication**: Communication between multiple threads is easier, as the threads shares common address space. Processes require IPC techniques.

## Interrupts

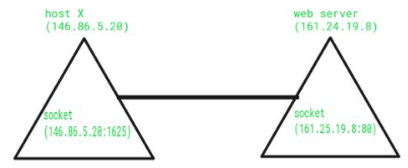
* Signal emitted by hardware or software when a process or an event needs immediate attention
* Interrupts can be generated by 3 sources: **Hardware, processor and software**
* Interrupts cause the CPU to switch to calling the interrupt handler
* **Interrupt handler** is a software routine that performs a specific task, such as reading data from a hardware device or notifying the kernel of a user input event
* Interrupts in Linux are managed by the Interrupt Service Routine (**ISR**) and the Interrupt Request (**IRQ**) subsystem
* The ISR is invoked to handle the interrupt, and the corresponding IRQ number is passed to the ISR”
* Linux kernel supports **interrupt coalescing**, combining many interrupts into one, which reduces the overall overhead of handling interrupts.

## Sockets and Pipes

**Sockets**

* Sockets are a type of file descriptor that allow processes to communicate with each other over a network or between processes on the same system
* The socket mechanism provides a means of inter-process communication IPC, by establishing named contact points between which the communication take place and provide a standardized way to send and receive data between processes or between a process and a network service
* Sockets can be created and managed using various system calls and APIs, such as socket(), bind(), listen(), accept(), connect(), send(), and recv().
* The Socket API is implemented in the Linux kernel and is used by many network services and applications
* Sockets are widely used for various network protocols, such as TCP/IP, UDP, and Unix domain sockets.

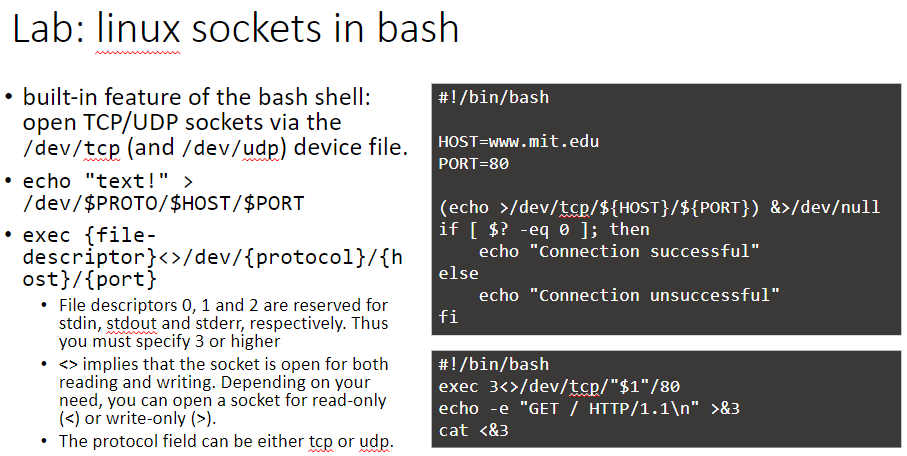
## What are they used for?

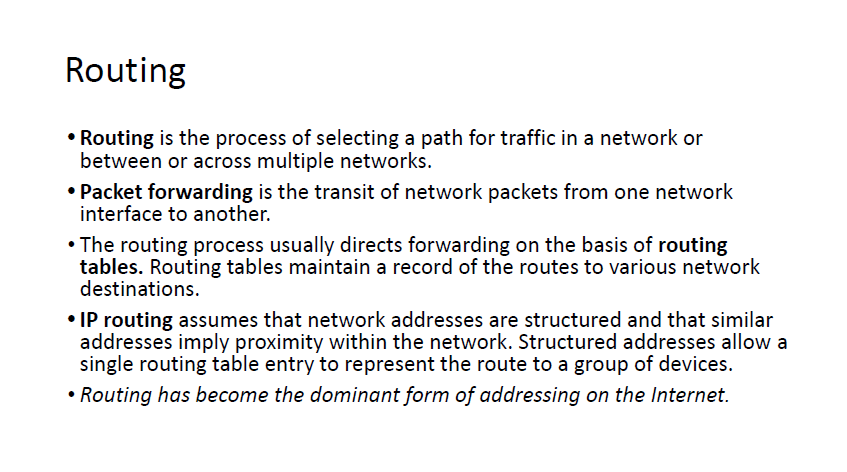
* Sockets are generally employed in client-server applications
* Server creates a socket, attaches it to a network port address and then waits for the client to contact it
* Client creates a socket and then attempts to connect to the server socket
* When the connection is established, transfer of data takes place

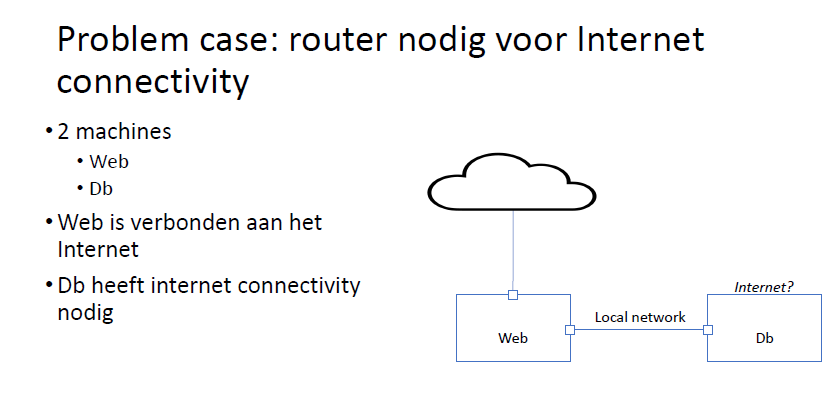
## Sockets – practically

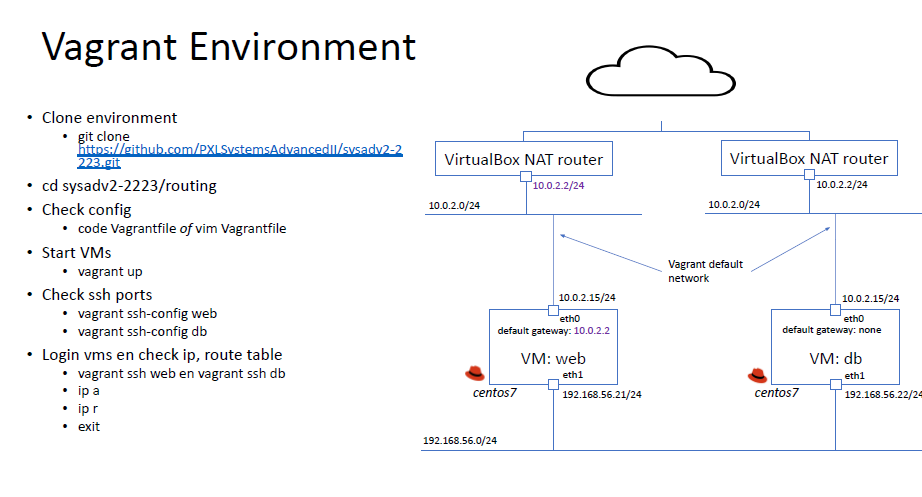
* A socket connecting to the network is created at each end of the communication
* Each socket has a specific address, composed of an IP address and a port number

## Lab example

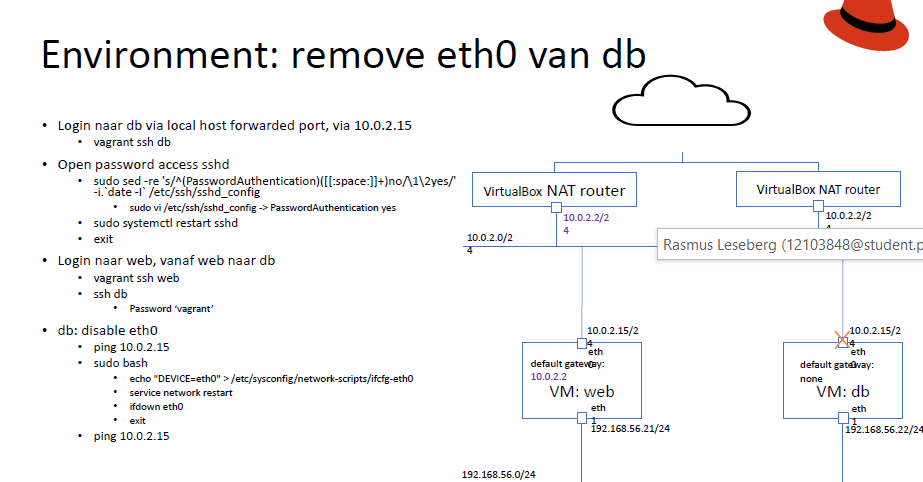
****

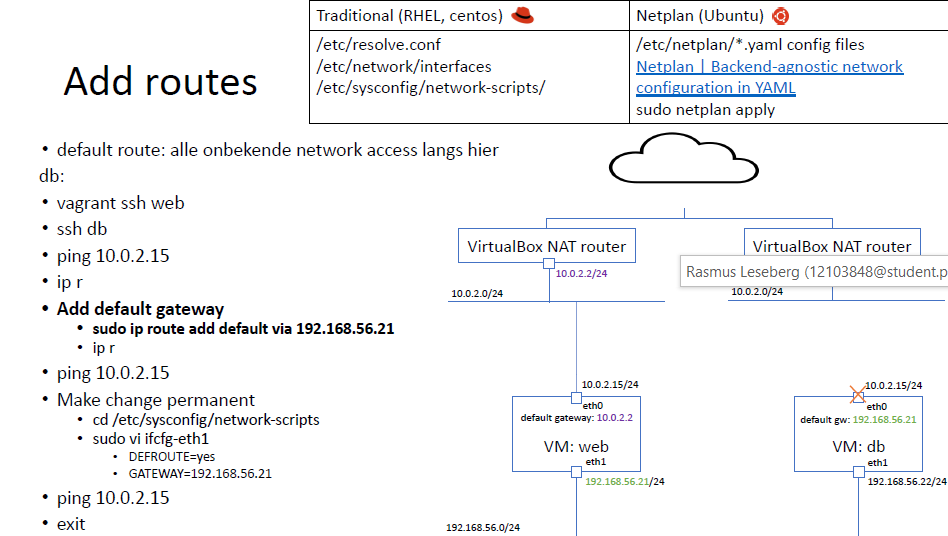
****

****

****

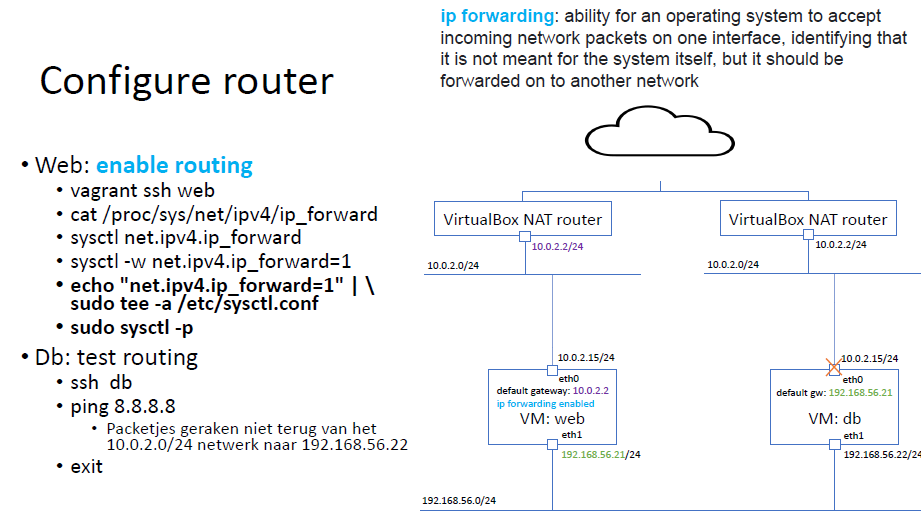
****

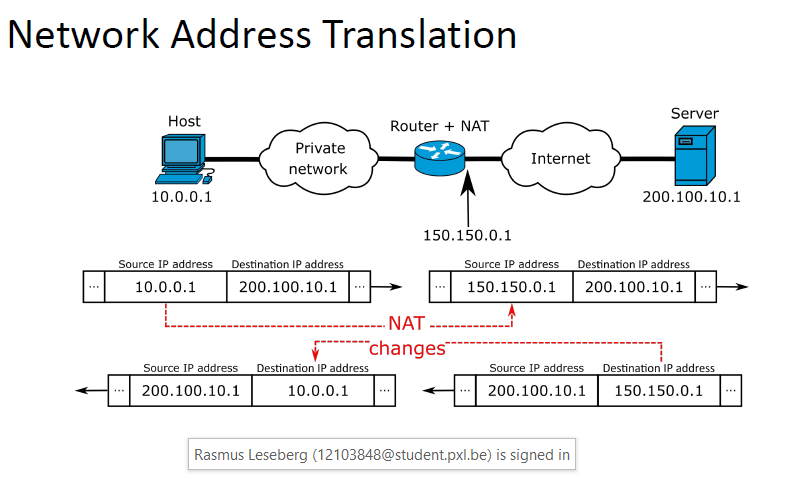
****

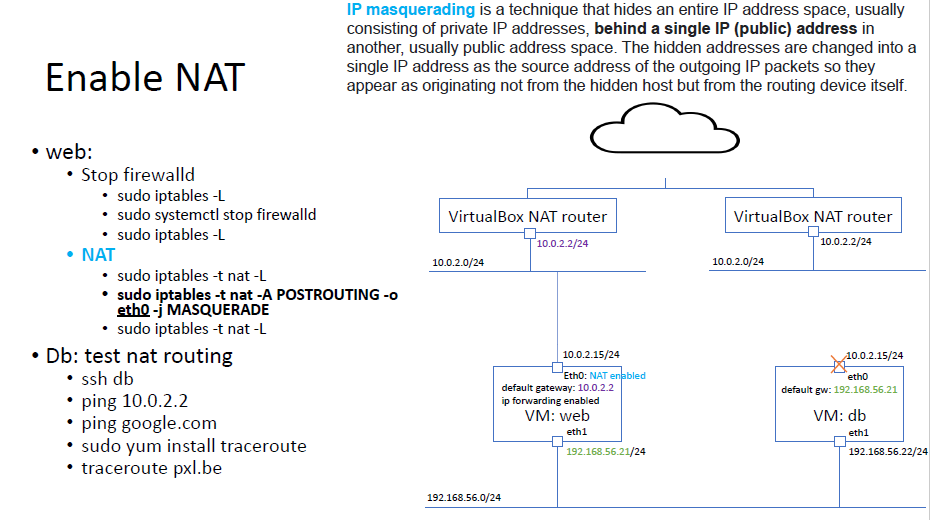
****

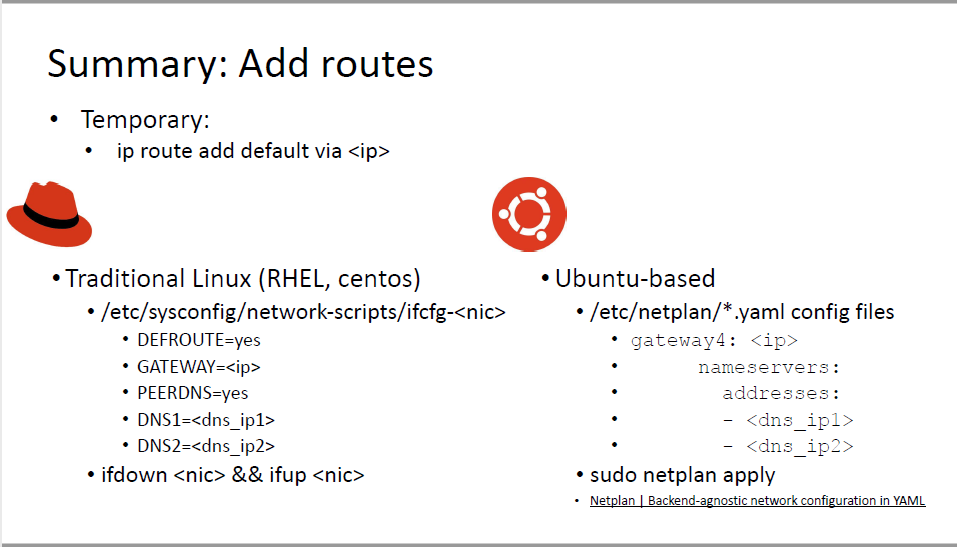
**Timeline

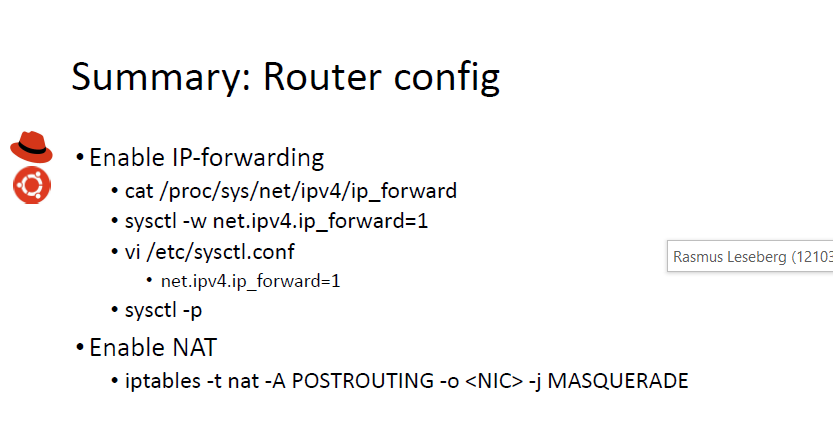
Description automatically generated**

****

****

****

****

****

**Firewall**