Linux System Administration

PM-TRTNB319

Zsolt Schäffer, PTE-MIK

Legal note

These slides form an outline of the curriculum and contain multiple references to the official Red Hat training materials (codes RH124, RH134 and RH254), since students of this subject are also eligible for the named RH learning materials. Some diagrams and pictures originating from the RH courses will show up in this outline as there is an agreement in place which allows for our Faculty to teach this subject based on the Red Hat curriculum.

All other pictures and diagrams sourced from third parties are explicitly marked with a reference to the origin.

Legal note

All students of PTE-MIK who have officially taken the course „Linux System Administration" are eligible to download these slides from the Faculty's internal network, and are eligible for license keys for Red Hat System Administration online learning materials at rhlearn.gilmore.ca as part of their training program.

Do not share, redistribute, copy or otherwise offer these learning support materials, license keys or account information either for money or free of charge to anyone, as these materials contain **intellectual property.**

Unauthorized distribution is both against the law and university policy and will result in an in-campus inquiry, suing for damages and/or criminal prosecution by the University of Pécs, Red Hat Inc. and other parties.

Preamble

Lecture 1

Chapter 1

Course details

Contact information

- Code: PM-TRTNB319
- Tract: 2x90 min every week
- Location: computer lab A101
- Course leader:
  Prof. Dr. Péter Iványi
  office B140
- Lecturer:
  Zsolt Schäffer
  office B103
  schaffer.zsolt@mik.pte.hu

Objectives

- Conceive an approach of how to administer Unix-like systems
- Practical knowledge of general administration of Linux based servers (setting up networking and users; providing database, web and network services, virtualization basics, etc.)
- Pick up synthesized knowledge about troubleshooting service issues on your own
- Preparation for the optional Red Hat Certified System Administrator and Red Hat Certified Engineer (ex200, ex300) exams

Requirements to apply

- Accomplishment of course ‘Operating Systems’ (PM-TRTNB230)
  - Basic knowledge of basic shell command and concepts (e.g. cat, echo, redirects, envir. vars)
  - Knowledge of file system manipulation commands (eg. ls, cp, mv, rm, chown, chmod, etc.)
  - Being familiar with at least one console mode text editor (vi is standard)
  - Knowledge of basic Unix tools, like: more, sort, grep, mount, etc.)

Requirements to pass

- Participate on **at least 70%** of lectures and lab practices
- There will be about 5 short MCSA tests at the start of randomly chosen lectures throughout the semester. Achieve an average of **at least 65%** by the end of the course
- A practice exam will take place on the last lecture of the semester. You'll have to set-up or correct the settings of a prepared virtual PC in accordance to written instructions. Achieve **at least 50%** on this final exam
- Both types of test can be repeated separately if necessary (but only 2 times at most in a single semester). Supplement exam appointments will be provided after the study period.

What do Linux server admins do?

- Set up services provided to other nodes on the company/institutional network or to the public. (Like file storage, block storage, e-mail, database, web applications…)
  This is achieved via configuring DEAMONS
  → lectures 9-11

  This is the most 'in sight' part, the thing visible to users/customers.

  Several other skills are required towards this:

- Configure basic system settings. Set up a suiting environment for daemons. (Install packages, add users, configure network, set up hard disks and file systems)
  → lectures 3-5,7

What do Linux server admins do?

- Schedule maintenance and backup tasks
  → lecture 8
- Harden security to protect the running environment to make the provided services more robust
  → lecture 4
- Troubleshoot boot and service issues, analyze logs
  → lecture 6,8
- Set up virtualization, uniform environments, mass deployment (only partly covered in this course)
  → lecture 12

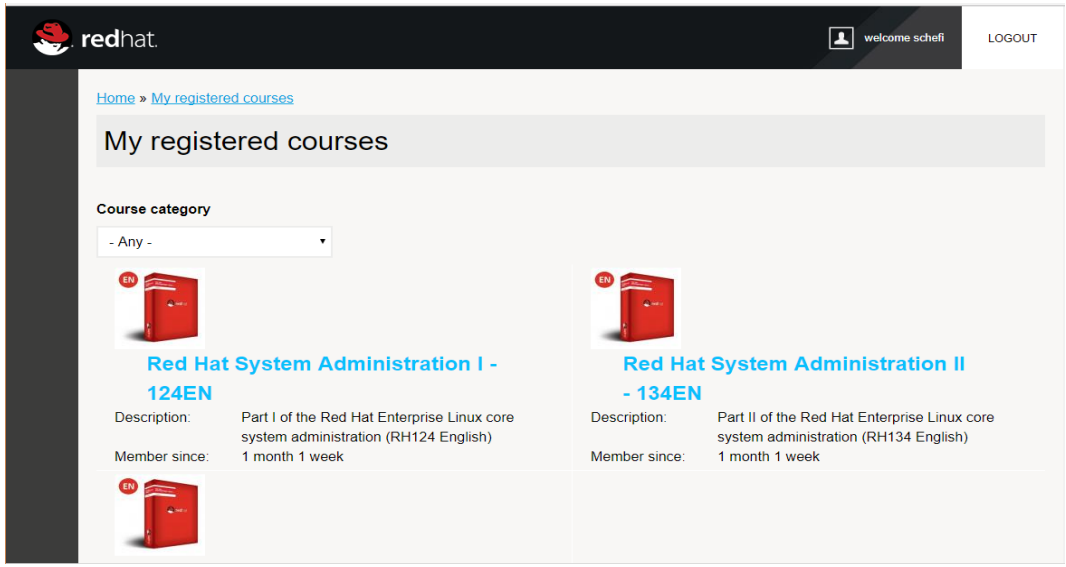
Intro to the Red Hat Lab environment

Lecture 1

Chapter 2

Sign-up to an RHLearn account

- Open https://rhlearn.gilmore.ca/ in your browser
- Create a new user account. (License key will be provided to your e-mail)
- Once logged in, select My Courses on the Dashboard
- All learning material will be available to you for 6 months after redeeming the key



Classroom Lab environment

- The lab consist of Classroom server and several student computers.
- Each student computers is a host for two virtual machines. Hands on practice experience throughout the course happens by manipulating these virtual computers. You'll have root access to them.
- The virtual machines are called serverX and desktopX, where X is a number denoting the seat you are in.
- The virtualization host can only be accessed as a restricted user, but you can still use the web browser, take notes, etc…
- The virtual machines are snapshoted, they can be reset to a ready made starting condition if something would go really wrong.

Logical layout

- Each student is provided with both a virtual server and client computer in order to be able to set up and test network services throughout the course.
- Some tasks only require one the virtual computers, some need both.
- The IP address of your personal virtual **server** is 172.25.X.11 where X denotes your seat number. Hostname resolving to this IP is serverX.example.com
- The IP address of your personal virtual **desktop** is 172.25.X.10 where X denotes your seat number. Hostname desktopX.example.com resolves to this IP address.

Controlling the virtual machines

- You can access the console of the virtual computers with clicking on the icon on your hosts dekstop.
  This opens up a VNC like connection (spice protocol) to the screen of the VM.
- Clicking the icon starts the VM if it isn't already running.
- You can control the VM through its graphical console (spice window) or using the rht-vmctl command on the host computer.
- You can restore the VMs to the starting state using the command line.

Controlling the virtual machines

- Also consider the following rht-vmctl commands:
  - start, stop
  - reset, save
  - fullreset
  - view

Host shell command line

```
# rht-vmctl start server

# rht-vmctl start desktop

# rht-vmctl start all

# rht-vmctl reset server
```

Reminder of some general OS and basic Linux concepts

Lecture 1

Chapter 3

In this chapter

- Kernel space, user space, system call, driver, daemon
- Scheduler, task, process, thread, interrupt, signal
- Standard streams, file descriptor, redirects
- “Everything is a file file” - file-centric concept of *nix
- Terminal, console, teletype, pty, command line, shell
- How to get help?

Kernel space

- Kernel space and user space separation is a way of sandboxing applications, in order to prevent them from doing globally harmful things like crashing the machine.
- CPUs usually have hard-wired features to support privilege distinction. Intel x86 CPUs have 4 hardware privilege rings. Linux kernel runs in ring 0. User applications run in ring 3. 1 and 2 is not used in Linux.
- Code running in kernel space has access to everything, including the whole range of physical memory and all bus I/O addresses, meaning it can send and receive data directly to any hardware.
- Kernel code consist of the kernel binary itself and several loadable extensions, called kernel modules.

Kernel space

- A specific functionality of the kernel can usually be set to 3 different states at compile time: disabled, enabled, or module. Not everything is selectable as a module.
- Modules can be loaded or unloaded at runtime from binary files called kernel object (*.ko). They are stored under /lib/modules/linux-[kernel version] directory.
- You can compile your own kernel module against the source code of the very specific version of the desired kernel. Kernel object files are usually not compatible across different kernel versions.
- Kernel objects can implement device drivers, filesystem drivers, cryptography functions, and a lot more, like intermediary layers between other kernel modules. There is a dependency tree involved.

User space

- User space applications are all the other software code that does not belong to the kernel. This isn't restricted to just applications with a user interface (like a spreadsheet editor), but a lot more.
- An executing instance of a program is called a process. Each user space process has it's own virtual address space, separated from all other processes.
- All processes have a unique numerical identifier (PID).
- All processes have a user ID (who own the process) and also an SELinux context. (More on this in lecture 3 and 4)
- The virtual address space of each process is mapped to physical RAM ranges (usually non-contiguous) and/or swap space. This is also done with the help of hard-wired CPU features (Memory Management Unit). Only kernel can manipulate the MMU mappings.

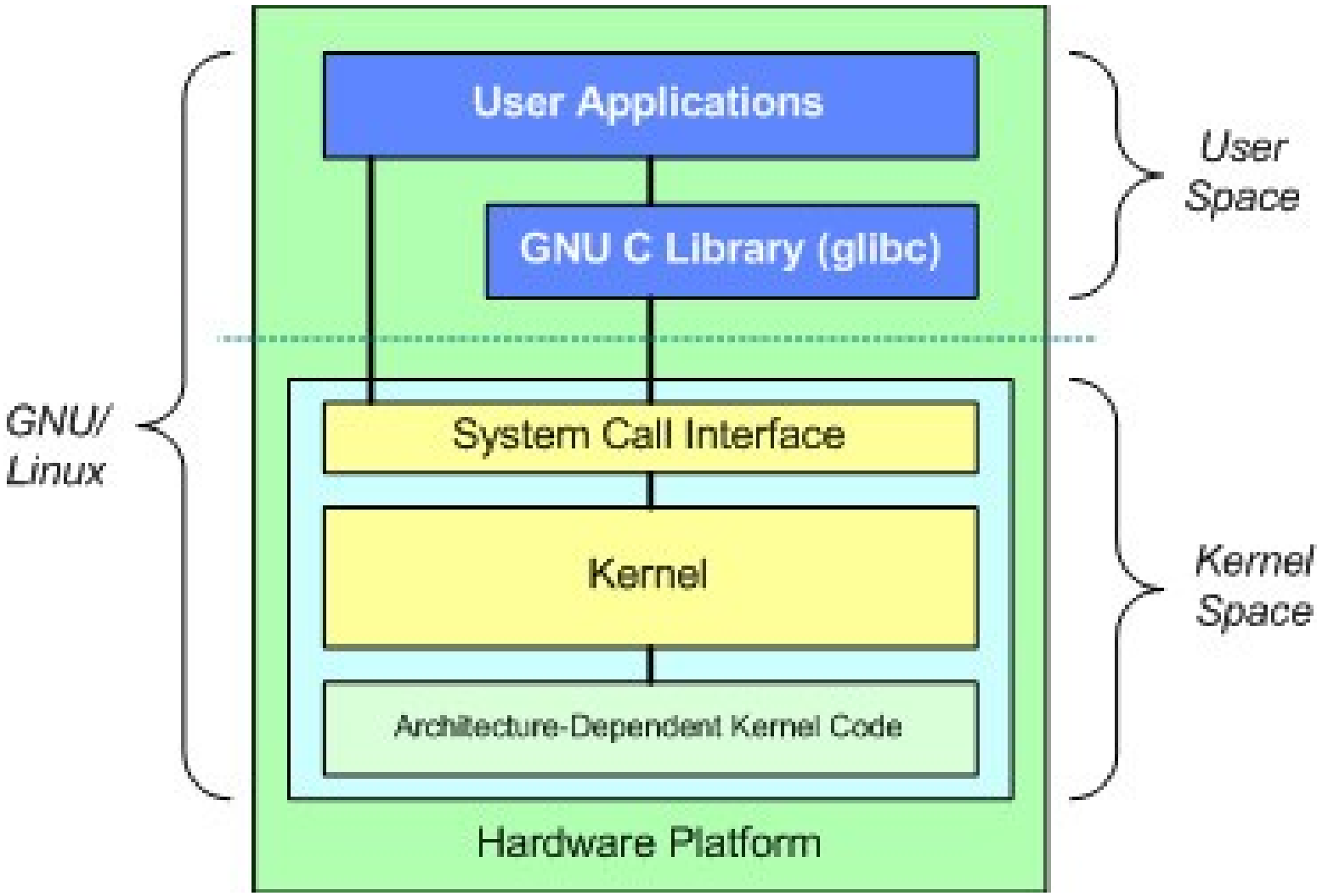
Scheduler, processes

- Userland processes don't have access to I/O buses either. Hardware is access through device files and system calls.
- Processes don't get unrestricted access to CPU resources either. Time slices are given to them by the kernel's scheduler. It decides which process to give the next time slice to, based on a configurable algorithm.
- Switching between processes involves saving and loading the execution state of the CPU several times a second (context switching). This is done by the kernel.
- Kernel itself doesn't behave like a process, it doesn't get scheduled. After init is complete it either just services interrupts or runs system calls upon userspace requests.
- More on schedulers, tasks, processes, threads, interrupts, signals will be discussed in lecture 4.

System calls

- System calls are requests in a Unix-like operating system by an active process for a service performed by the kernel, such as input/output (I/O) or process creation, opening of a file, memory allocation,etc.)
- Kernel version can change frequently in a system. The stable interface of requesting kernel functions to be performed is the standard C library (libc).
- Libc versions are widely compatible (within major versions), even vendor inter-operability is possible (glibc ↔ eglibc are ABI compatible, uclibc ↔ glibc are only API compatible)
- E.g.: calling the printf() function from a program works on a wide range of systems without recompiling. Several kernel layers are involved, we get expected results independent of actual output device. (serial port, VGA console, SSH session or X11 terminal window)

Fundamental architecture of Linux



https://www.ibm.com/developerworks/library/l-linux-kernel/figure2.jpg

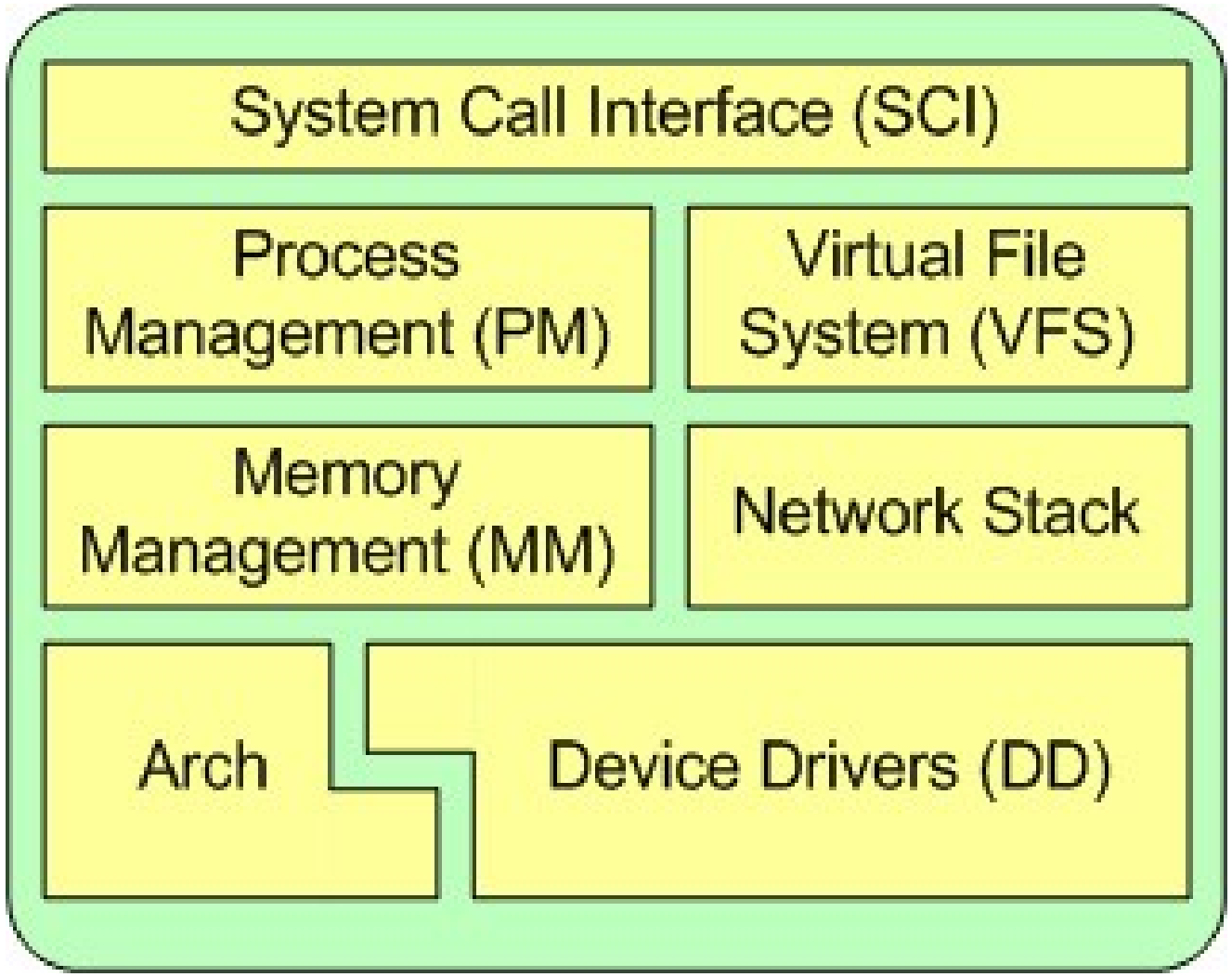
Hardware devices

- Access to hardware for userland processes is provided via device files, usually located in the /dev directory or using the sysfs filesystem. (More on this in lecture 5)
- For e.g. reading input from a serial port is performed by opening the /dev/ttyS1 device, then performing an tcsetattr() call to set baud rate and other serial preferences. After that the port can be written or read just like a text file. E.g. read(12,buffer*,10); This line of C code reads 10 bytes from file ID 12 (the serial device in this example) to an array pointed by buffer* pointer.
- You can get snap images from a webcam device or read the orientation sensor of your handheld device in a similar way.
- E.g. turning an LED on or off in a Raspberry Pi is done by writing a 0 or 1 to a file like /sys/class/gpio/gpio17/value

Drivers

- Simply put drivers are kernel objects that configure, initialize and perform raw I/O to hardware devices and represent the device as a file to userspace programs.
- A driver is programmed by implementing interrupt handlers and file operation functions in C code. (like open, seek ,read, ioctl, etc.) These function are compiled against a specific kernel source code into a ko file.
- The hooks of your implemented functions are imported by the kernel and executed appropriately every time a userspace process wants to access the device.
- The user process only performs file operations, subject to standard filesystem permissions. Actual bus I/O is performed in kernel space by functions compiled into the kernel object. The application makes the request by asking the kernel to call the read() or other function implemented in the given device driver.

Major subsystems of the Linux kernel



https://www.ibm.com/developerworks/library/l-linux-kernel/figure3.jpg

Daemons

- Daemons are userspace processes. Just like the ones discussed earlier.
- The special thing is that they don't interact with a user directly. Instead their input (rules of how they should operate) are read from a config file and their output is written to a log. Other than that, daemons only communicate to other processes on the same host or over the network.
- Therefore they are often referenced as background processes. This is not quite accurate, since any other process can be put in the background too.
- Usually both the config and log contains only human readable text. (This is strong Unix concept)
- The names originates in ancient Greek, it means "personal protecting spirit", like a guardian angel.

* https://i0.wp.com/blog.chudinov.net/wp-content/uploads/2014/12/tuxdemon.png?w=250

Daemons

- Daemons are strictly focused on doing one very specific (sometimes even simplistic) task, but with the goal of doing it most proficiently. (This is a strong Unix concept.)
- Daemons are the most prevalent way of providing automated services. (For e.g. serving .jpg and .html files from a web server or providing network addresses in reply to DHCP requests.)
- Hence daemons are a very important part of this subject. There is a daemon for every task somewhere.
- It is impossible to cover them all. What will be covered (through several examples) is the concept itself to configure services through text files and troubleshoot through logs. (More in lecture 9 and 10)

Standard streams

- Remember the example of the printf() function. The application calling this function is completely unaware of the output devices.For e.g. it can be a graphic terminal on HDMI screen or a serial port on the same computer like a Raspberry Pi or a remote terminal through a network session. It still works in all cases without recompiling or modifying the a program.
- The kernel contains the required layers for a printf() call to become actual output.
- But how is this interfaced to the application? How is it standardized?
- As discussed earlier, applications do only file operations instead of directly manipulating the hardware.

File descriptors

- When a process executes the open() syscall to open a file, the function (on success) returns a positive integer, this will be a file descriptor.
- This number uniquely identifies the file within the specific process which opened it. File descriptors are not globally unique, but process are sandboxed anyway. This info is public, it is exposed through /proc/[PID]/fds dirs.
- Every process gets three file descriptors opened upon their creation. No manual opening is necessary. This is handled by standard libc during application initialization.
- These file descriptors are the following:
  - 0 for standard input (stdin)
  - 1 for standard output (stdout)
  - 2 for standard error output (stderr)

Standard streams

- So the printf() function actually only writes characters to a “text file”, specifically file nr. 1, which is standard output for all Unix processes. The kernel handles all the rest.
- The terminal handles line editing and can pass each “sentence” (finished with the Enter key) to the process. The process just basically request to read a line from the file called stdin.
- Streams can be redirected to/from disk files or other processes (even to other computers - with SSH). More on this in lecture 2.
- In correlation to the known Unix concept of an application doing only one thing but doing it efficiently, redirection makes it possible to solve complex problems with channeling data between simple building blocks on-the-fly.

Everything is a file

- So far several perspectives have been shown, how the unifying concept of filesystem objects can work in Unix.
- The benefit is that a set of utilitys, APIs can be used on a wide range of resources. Documents, modems, keyboards, inter-process communication, even internal kernel variables and structures, etc… are represented as files.
- The following list of file types are known in Linux, these are all file system objects (FSO): regular files, directories (they are files too), UNIX domain sockets, named pipes, device node files, symlinks. Also have to mention hard links and virtual files.
- More on named pipes in lecture 2, directories, links and virtual files in lecture 5.

Everything is a file

- A full list of file types with notion of first output field of the ls command, and the related command to create the object.
  - `-` regular file — touch
  - `d` directory — mkdir
  - `c` character device node — mknod
  - `b` block device node — mknod
  - `p` named pipe — mkfifo
  - `s` socket — socket() syscall
  - `l` symbolic link — ln -s
  - `-` hard link — ln
  - `-` virtual files (/proc, /sys) — -

Terminal, tty

- Terminal from an electronics point of view is an end point, where an external device is connected to the service/network.
- In the early days of Unix mostly mainframe computers where current. Multiple users connected to a single MF using dumb devices (consisting only of a keyboard and a screen or printer and a means of transcieving signals to/from the mainframe). They where called terminals.
- Terminal is synonymous with tty. It's the abreviation of TeleType, which is a sort of electromechanical typewriter with a long "wire", capable of printing characters remotely corresponding to locally pressed keys. Original device is from 19th century.
- In Linux any character device can be a tty that can perform some ioctl commands and receive/transmit series of bytes, like an RS-232 serial port.

http://www.harriscomm.com/media/catalog/product/cache/1/image/9df78eab33525d08d6e5fb8d27136e95/u/t/uti-sp4425_1.jpg

https://c1.staticflickr.com/3/2553/3916969051_ecc076867e_b.jpg

https://library.ryerson.ca/asc/files/2013/01/img026.jpg

https://upload.wikimedia.org/wikipedia/commons/thumb/7/71/IBM_3277_Display.jpg/1024px-IBM_3277_Display.jpg

Terminal

- The terminal is where the user logs in, presses keys, and reads answers. Terminal handles printable I/O and control characters (like the arrow keys) one by one. No interpretation of commands is done by the terminal.
- A terminal usually has a buffer, contents can be scrolled back. (Shift + PgUp, Shift + PgDn on Linux)
- A terminal can have line editing features but not necessarily.
- Terminals are provided by the cooperation of several kernel layers, some of them closely bound to the hardware itself.
- A terminal can only support characters and key presses, not graphics.

Shell

- A process usually communicates its standard in- and output through a terminal.
- The shell is a userspace process that also runs in a terminal. The shell is responsible for interpreting commands, providing an environment for execution of other processes and much more. Command history, TAB completion and such features are provided by the shell, not the terminal.
- The shell that executes text applications relates to the terminal like a window manger that executes graphical applications relates to the graphics canvas provided by the video card driver.
- The most common shell used today is bash. More on this in lecture 2.

Console, vty

- A console is like a primary terminal connected directly to the system. For e.g. A BIOS based computer's console consist of the standard 101-key keyboard and the standard 80x25 character VGA text output screen. (This is just a text buffer, automatically rendered to characters on the screen by the VGA card itself.)
- /dev/console is a link to one of the terminal devices.
- A computer equipped with UEFI firmware has a standard graphical surface as output, called the uefifb. BIOS boxes can still use vesafb with fbcon (vtcon). This is a graphical surface but still a terminal, not a desktop.
- A Linux PC installs with 6 terminals by default. These are called virtual terminals and can be cycled with Ctrl+Alt+ F1 to F6. Only one of them can display on the physical console at once (Be it vesa or text don't matter). A userspace process called getty is involved in cycling.

Terminals

- List current user session with the **who** or **w** command.
- Also consider the **who -a** switch.

Shell command line

```
# w
 10:22:47 up 2 days, 22:56,  2 users,  load average:
0,04, 0,06, 0,05
USER     TTY        LOGIN@   IDLE   JCPU   PCPU WHAT
root     pts/0     10:21    0.00s  0.29s  0.02s w
root     pts/1     10:21    1:08   0.25s  0.25s -bash
```

- The file concept also applies to terminals. You can write to other users terminals, just like writing to text files.

Shell command line

```
# echo -e 'Hello\n' > /dev/pts/1
```

- This will produce a line of text on pts/1 terminal.

Terminals

- To enable a mouse cursor in text console, type the following:

Shell command line

```
# systemctl start gpm
```

- To select text press the left mouse button and drag the mouse.
- To paste text in the same or another console, press the middle button.
- The right button is used to extend the selection.
- You can also use ctrl + ins and shift + ins to use the clipboard (copy/paste).

Pty

- A pty (pseudo terminal) acts just like a normal terminal to the process that runs inside it, but it is not connected to a piece of hardware. Rather it is connected to a software layer provided by another process, like **screen**.
- One example would be an X11 terminal emulator window or a Gnome Terminal, which is a user space application providing a graphical window with a scrollbar, selectable font and clipboard functions. If a shell is started in this pseudo terminal, it will not be relevant to it, since a pty acts just like a tty.
- An other example would be an ssh session, when managing a remote server through a shell. When logging in to a remote SSH server, it would create a pseudo terminal for a new shell process. Then all terminal I/O will be transferred to/from the ssh daemon through a network tunnel from/to the local ssh process.

How to get help?

- Try executing one of these commands
  - man
  - info
  - help
- Search in man pages for specific keywords

Shell command line

```
# man -k keyword
```

- Man pages are also available on Google. Try: “man ls”
- Search for exact error message on Google. Try something like this: “PTY allocation request failed on channel 0”

How to get help?

- Read text files in /usr/share/doc directory
- For tutorials look at http://www.howtoforge.com
- Search the RedHat documentation at https://access.redhat.com/documentation/en/red-hat-enterprise-linux/
- Use the redhat-support-tool command

Recommended reading

Lecture 1

Chapter 4

Relevant chapters in RH:

- RH124 Chapter 3 (getting help)
- RH124 Chapter 1 (command line)

Read by next lecture:

- RH124 Chapter 2 (filesystem layout, file manipulation)
- RH124 Chapter 4 (redirect, cat, vim)
- RH134 Chapter 2 (regexp, grep)
- RH134 Chapter 3 (vim)

Use your rhlearn.gilmore.ca login to access the learning materials.

Thank you for your attention!

Lecture 1, PM-TRTNB319, Linux System Administration

Zsolt Schäffer, PTE-MIK