https://hpcleuven.github.io/Linux-for-HPC/

Vlaanderen
is supercomputing

Linux for HPC

SCAN ME

VLAAMS SUPERCOMPUTER CENTRUM | *Innovative Computing for A Smarter Flanders*

vscentrum.be

Overview

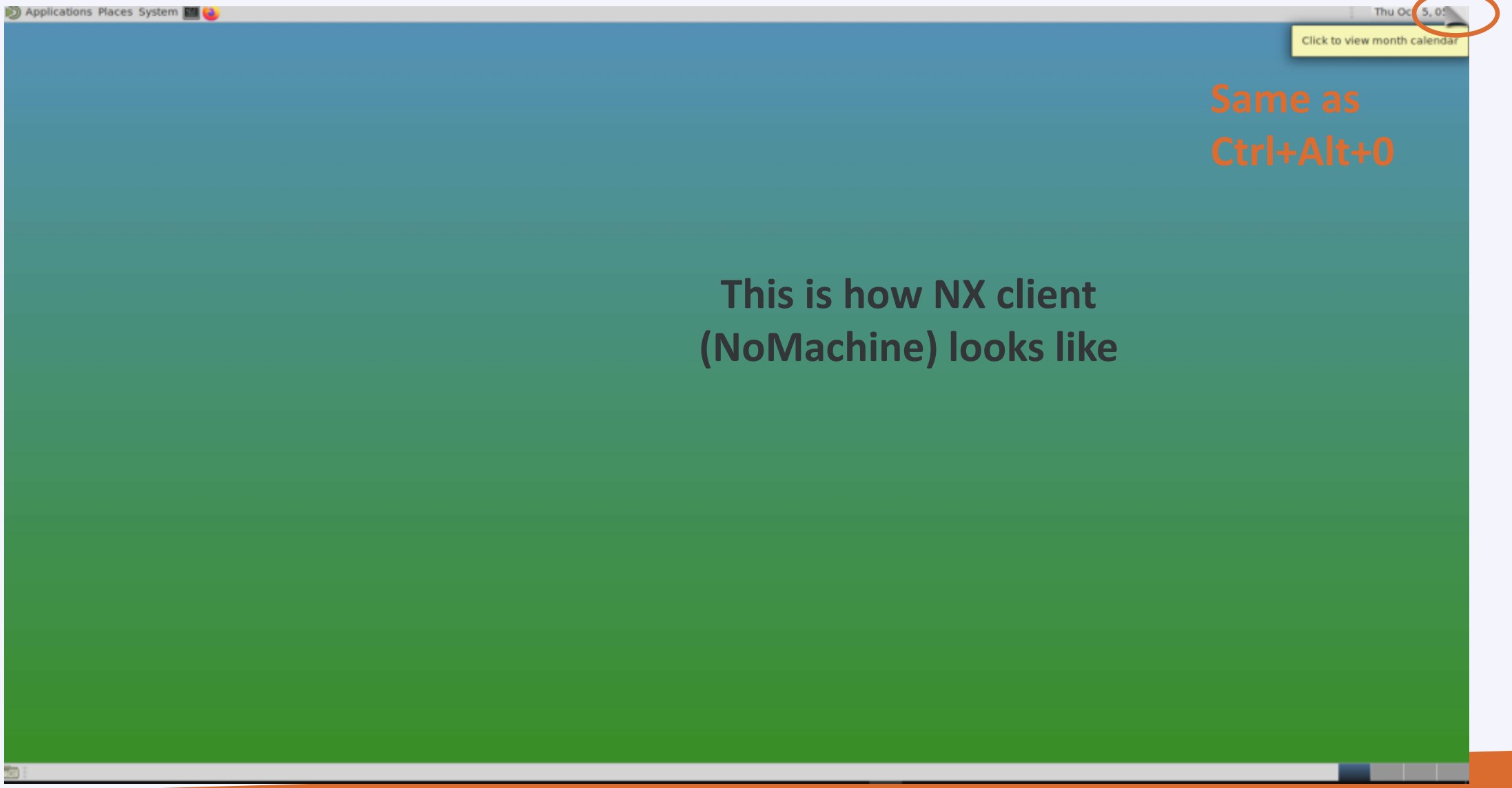
- Who is this course for?
  - You know Linux basics
  - You use HPC platforms
  - Need a generic overview of Linux tools

Outline

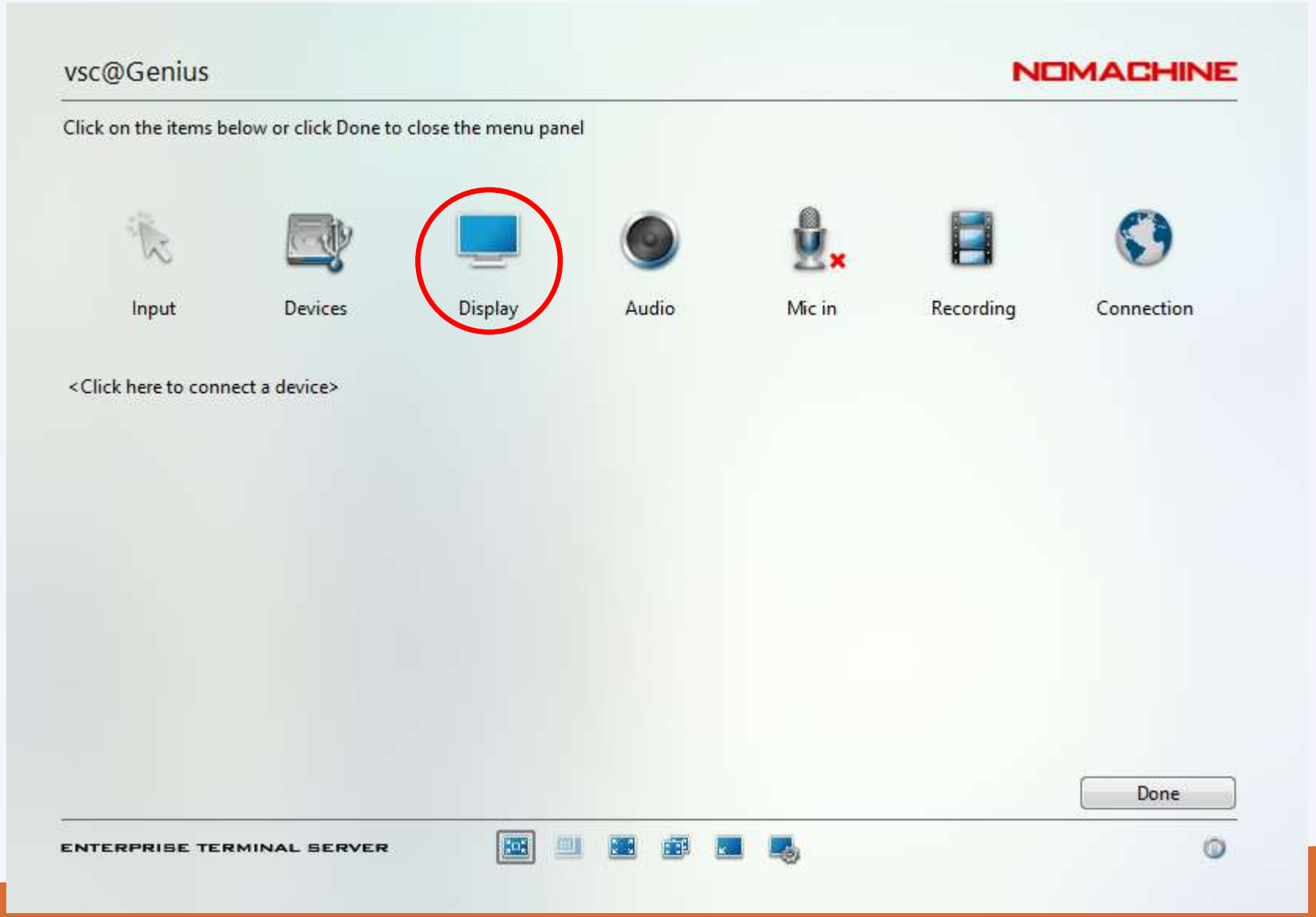
- Using NX
- More Linux commands
- Key Concepts, Redirection & Pipes
- Shell, Bash and Job Scripts
- Installing & Compiling Code

Customizing GUI (NX)

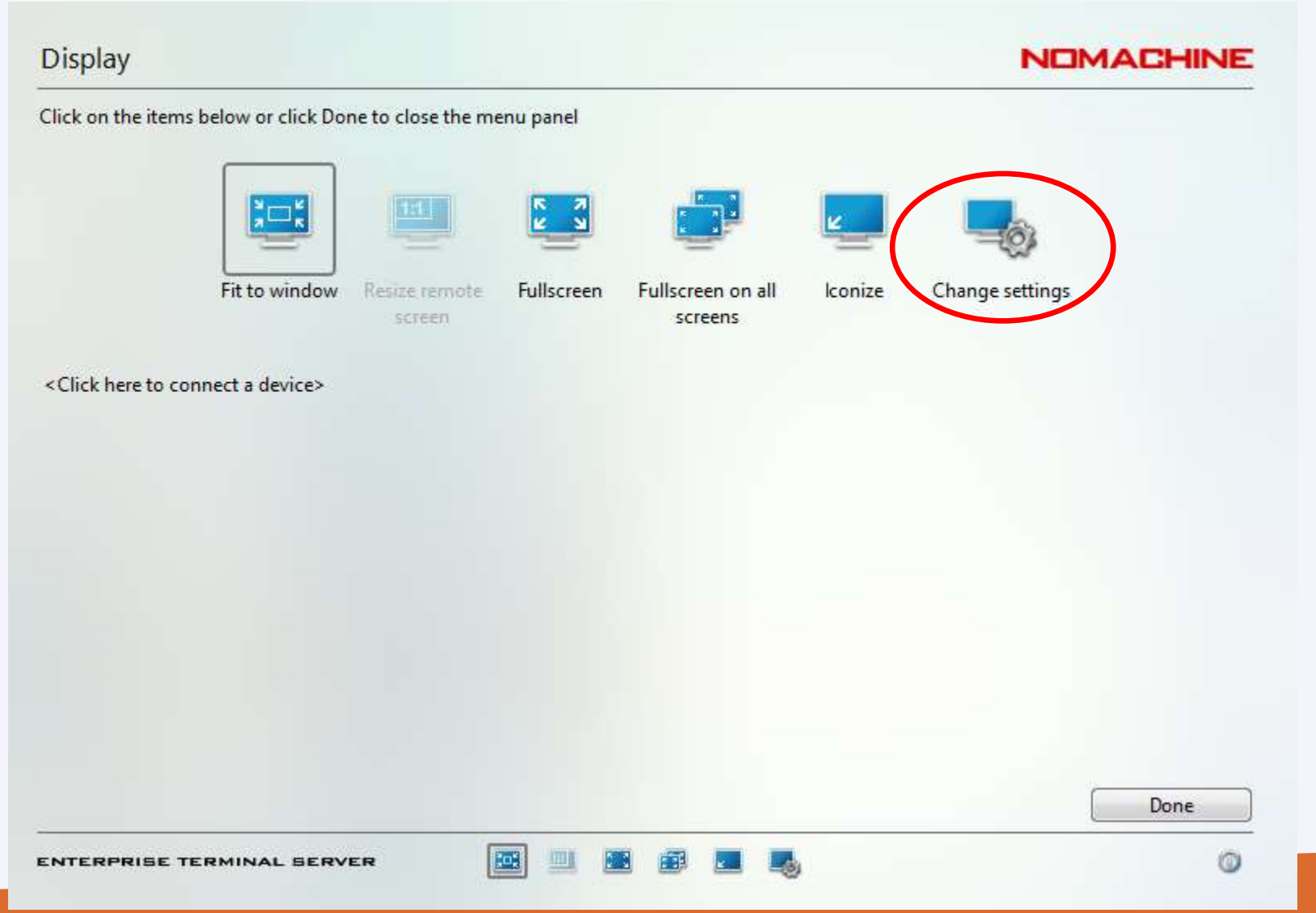
System tools



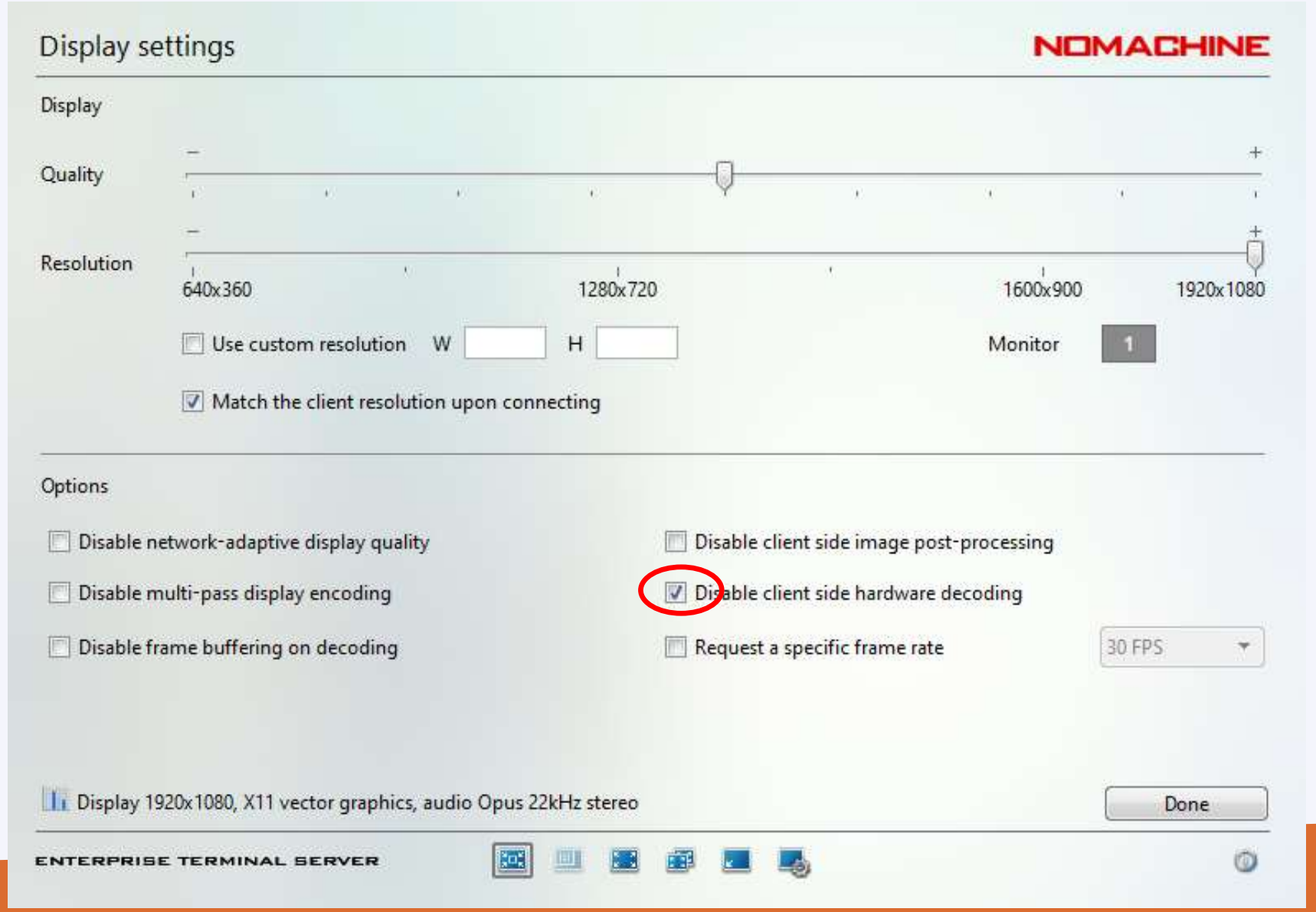
System tools – proper display



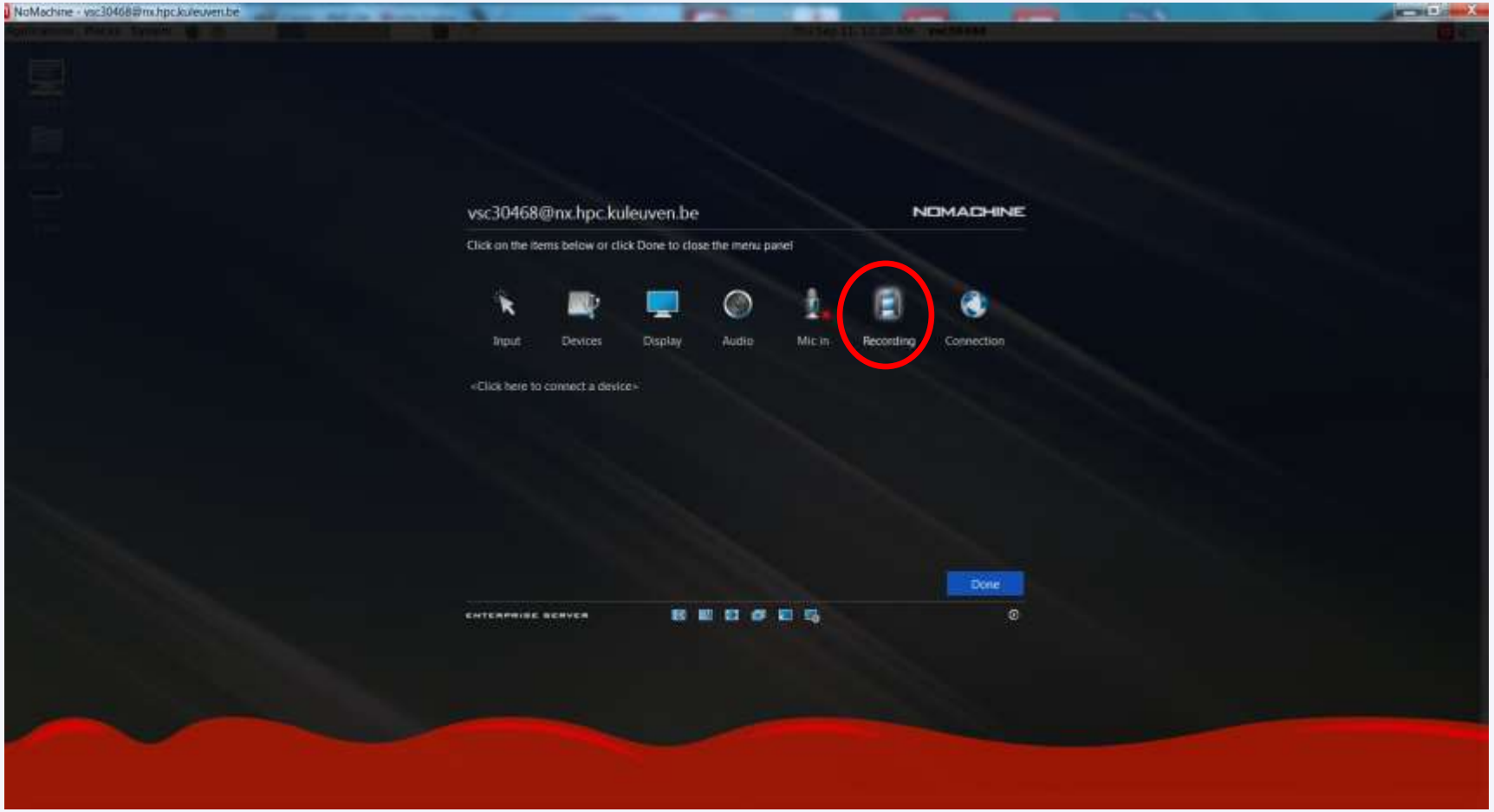
System tools – proper display



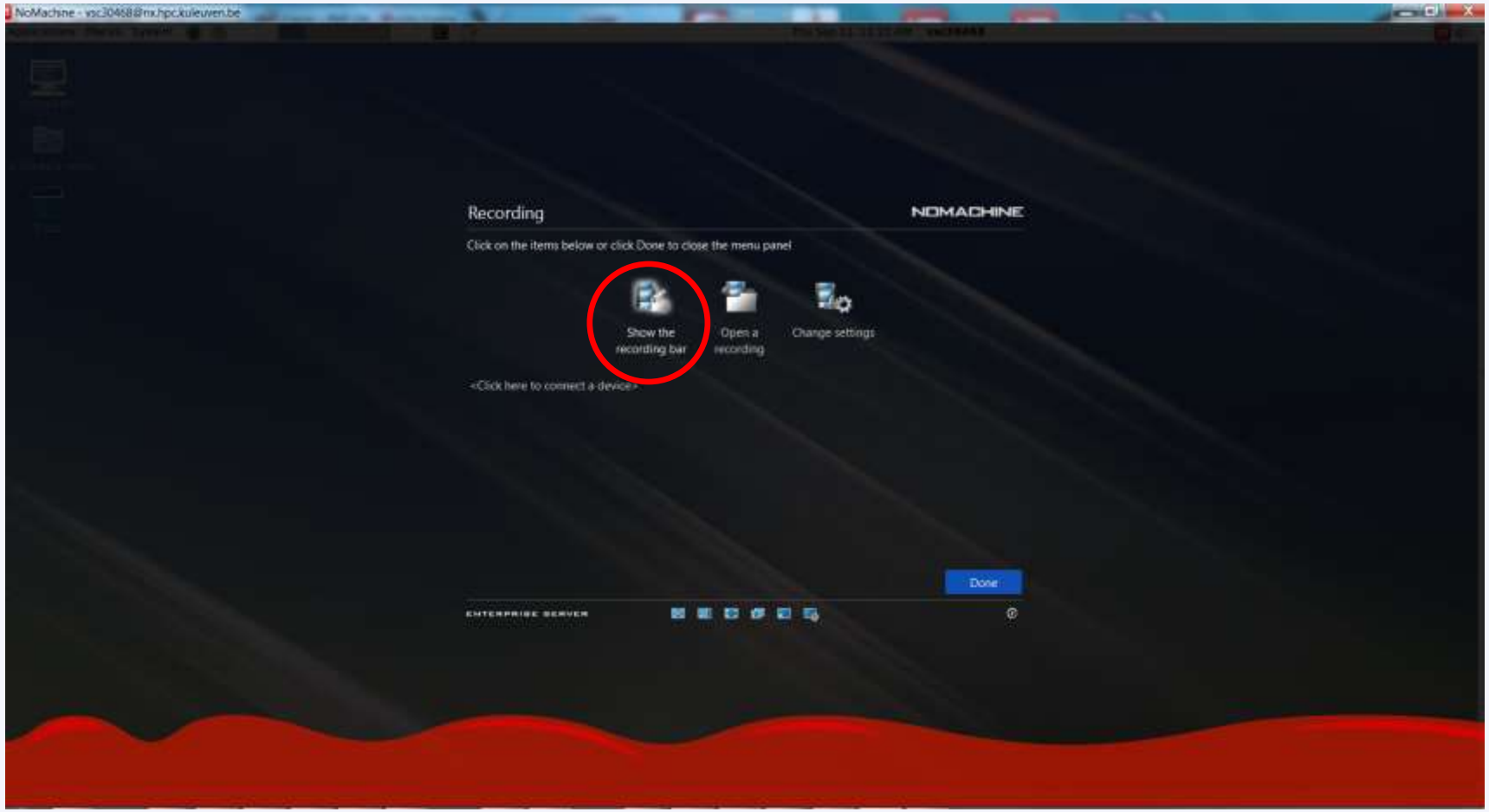
System tools – proper display



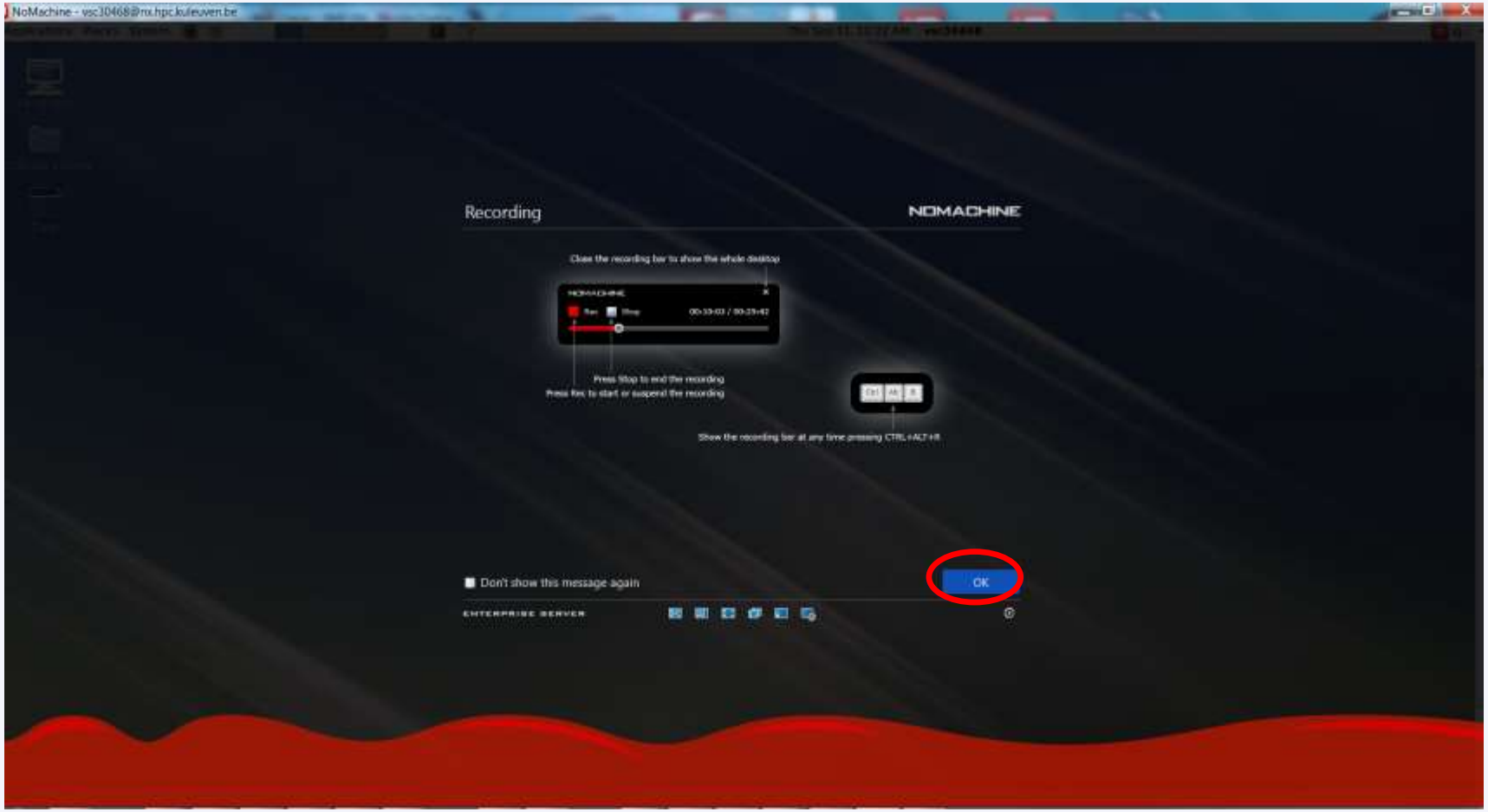
System tools - recording



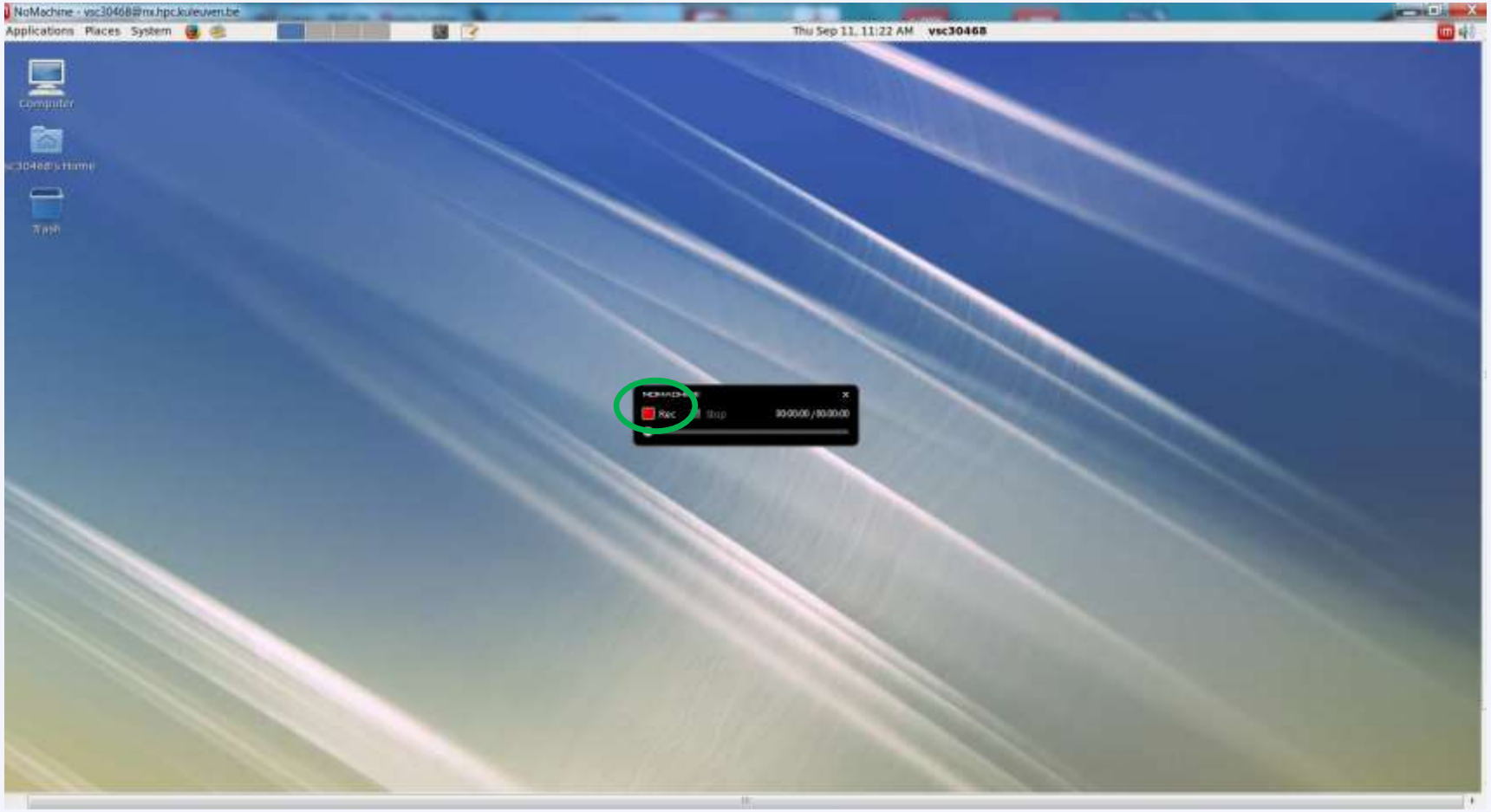
System tools - recording



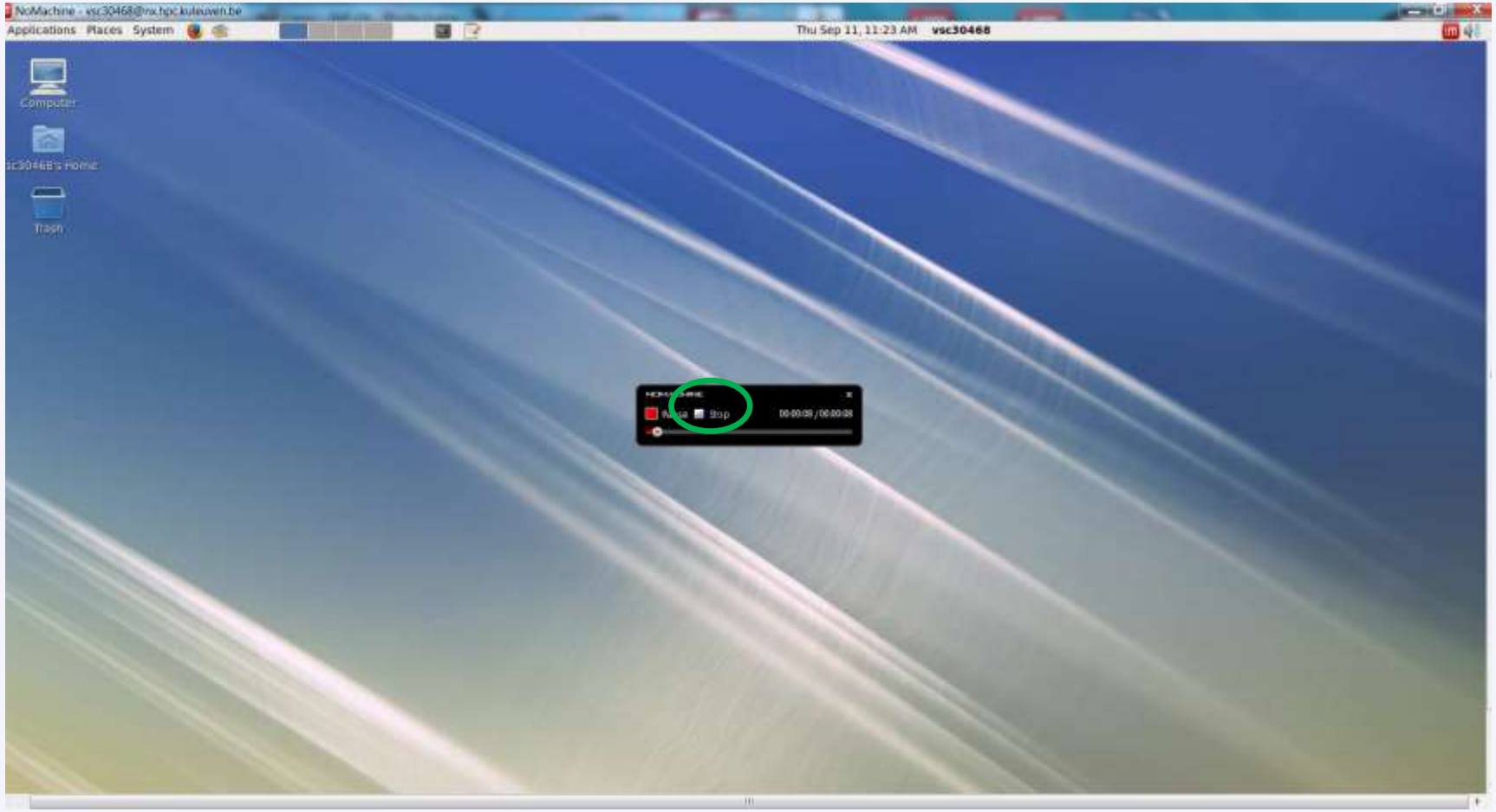
System tools - recording



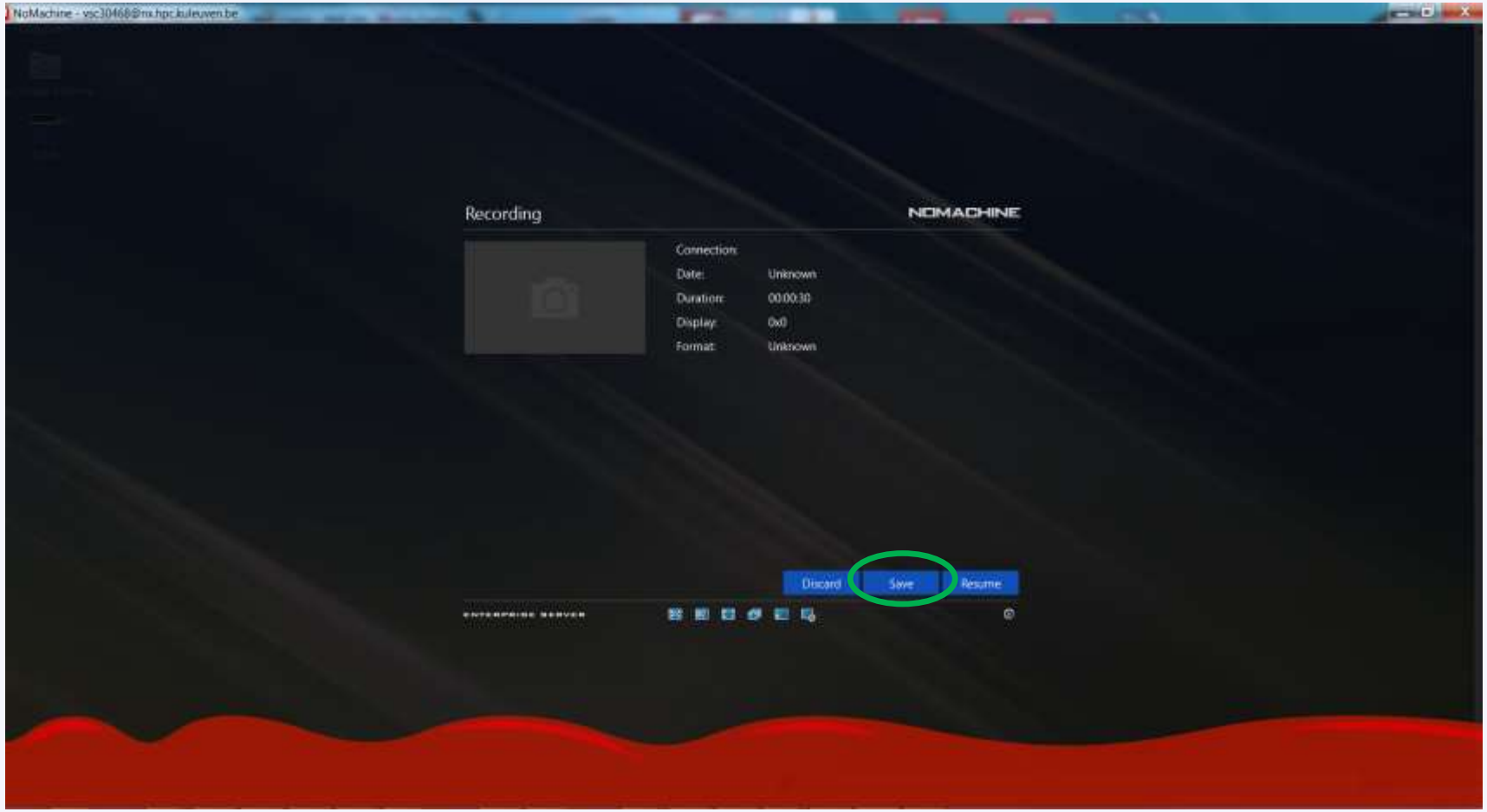
System tools - recording



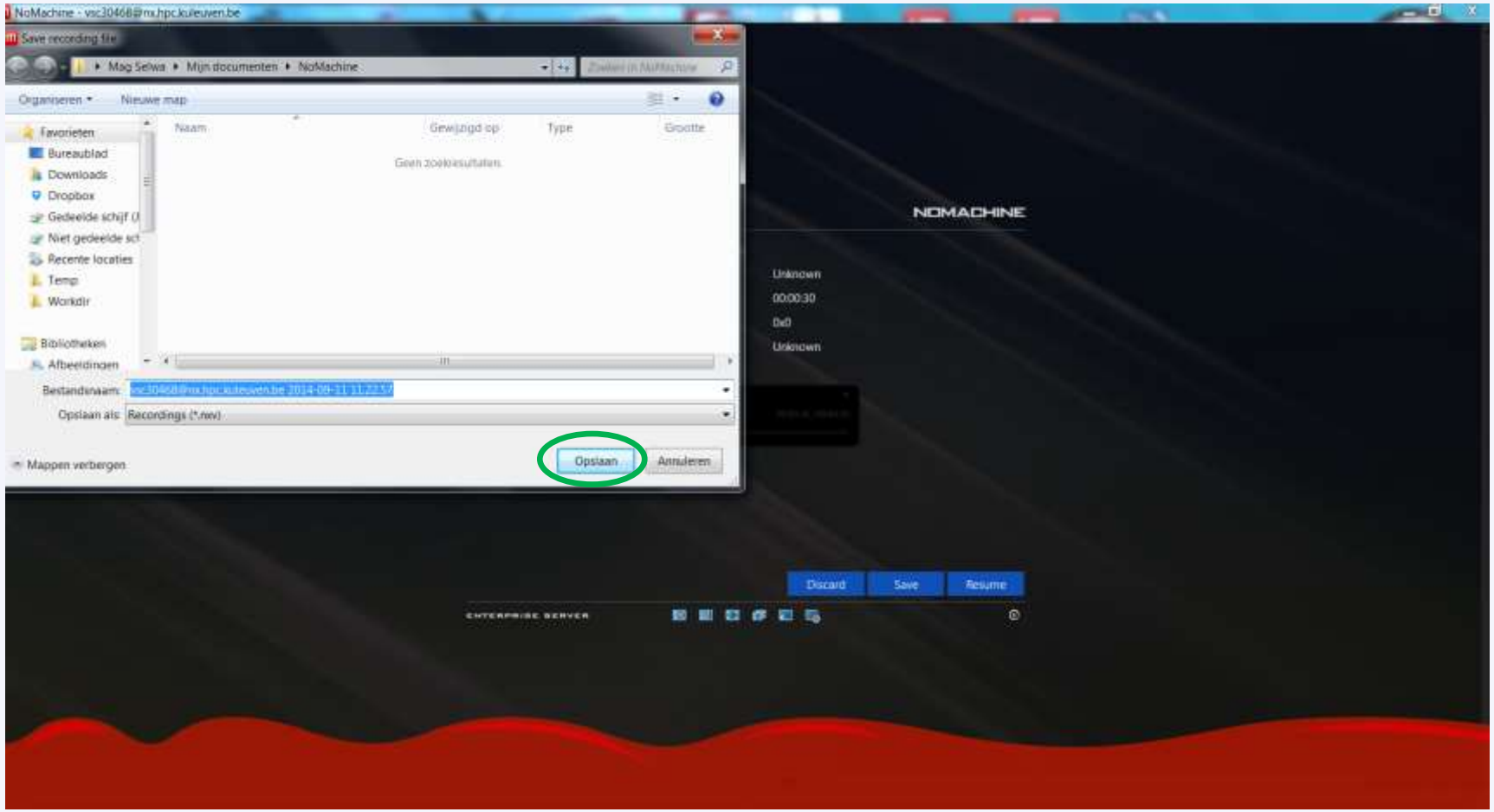
System tools - recording



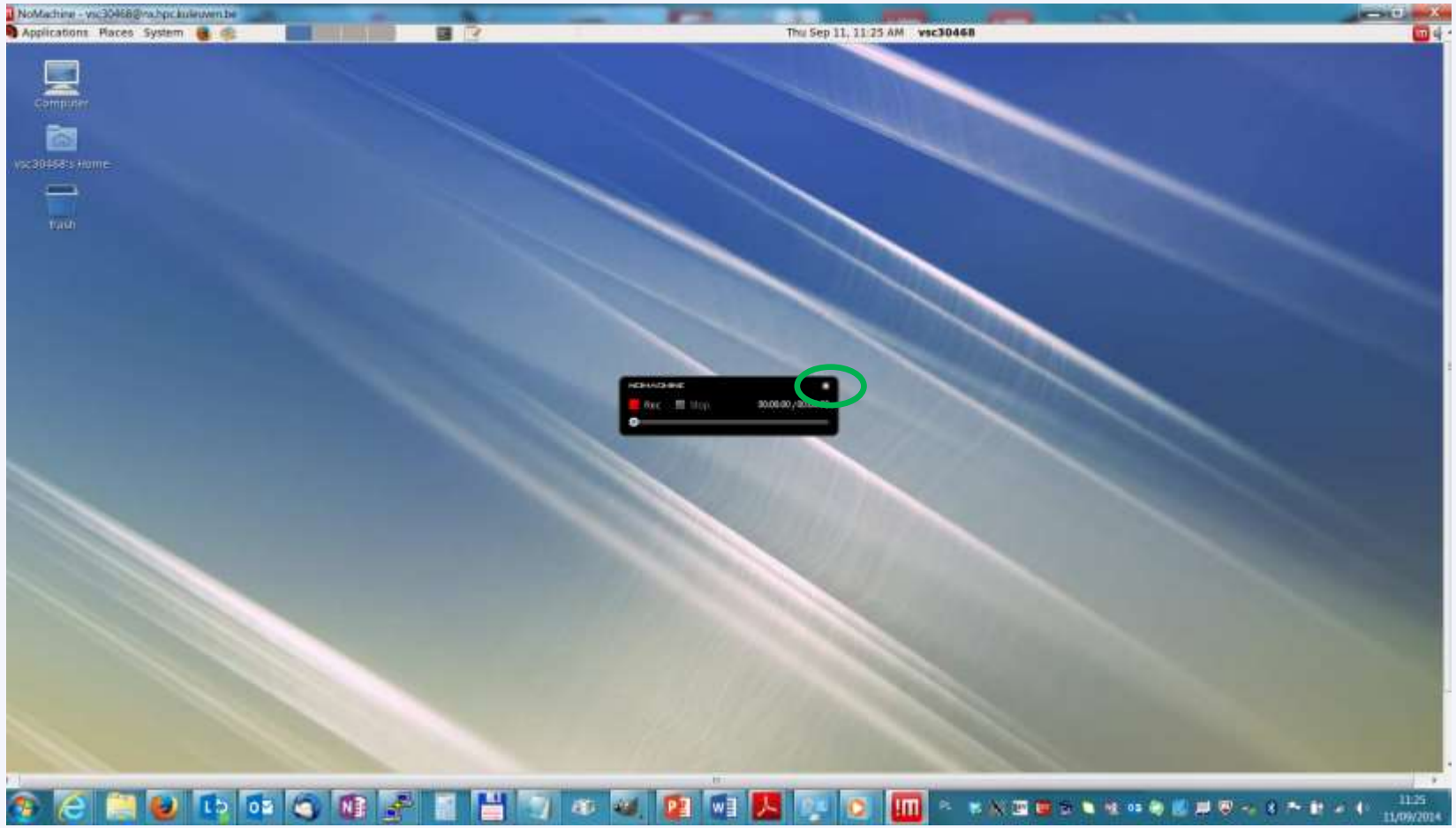
System tools - recording



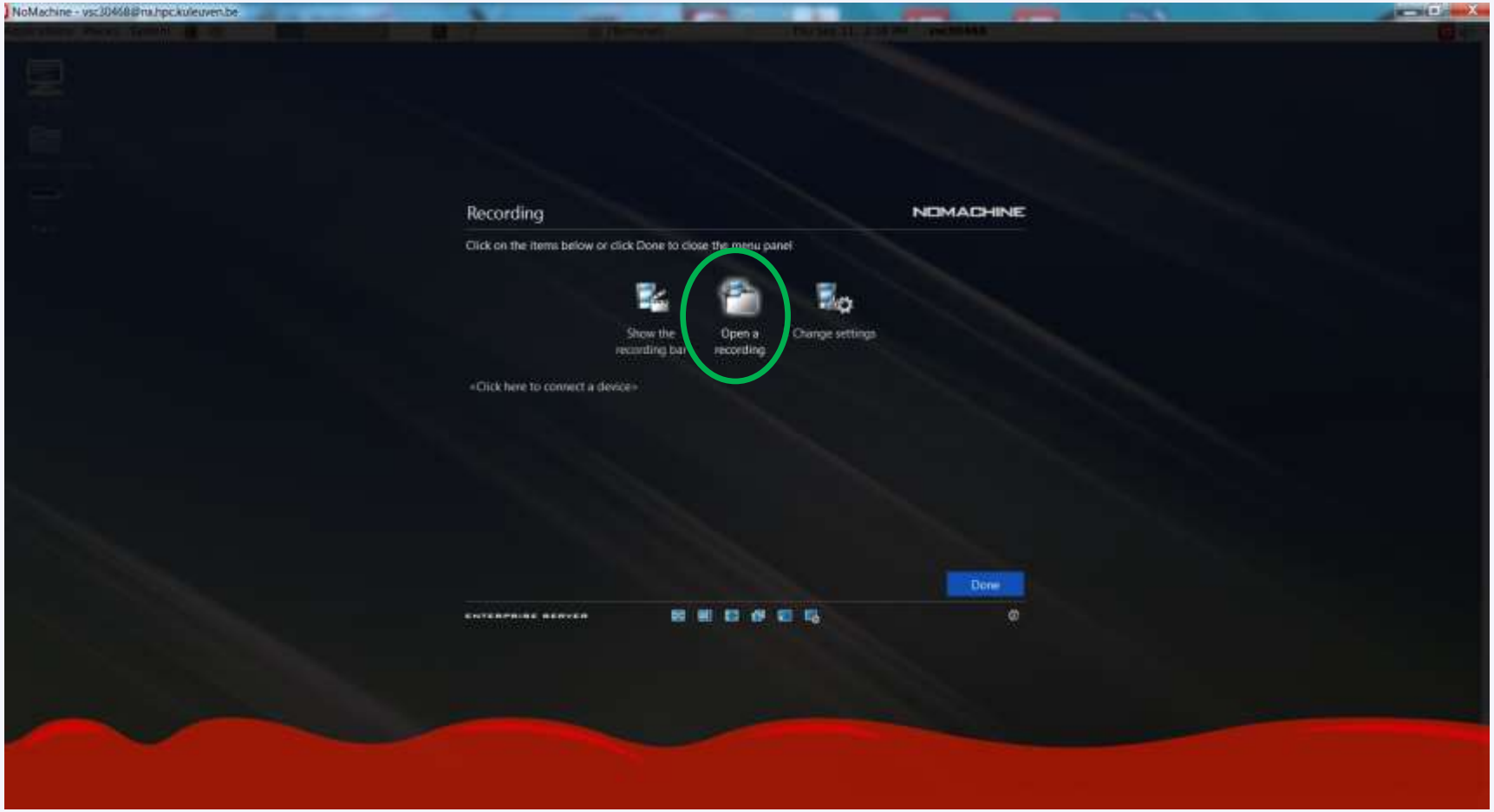
System tools - recording



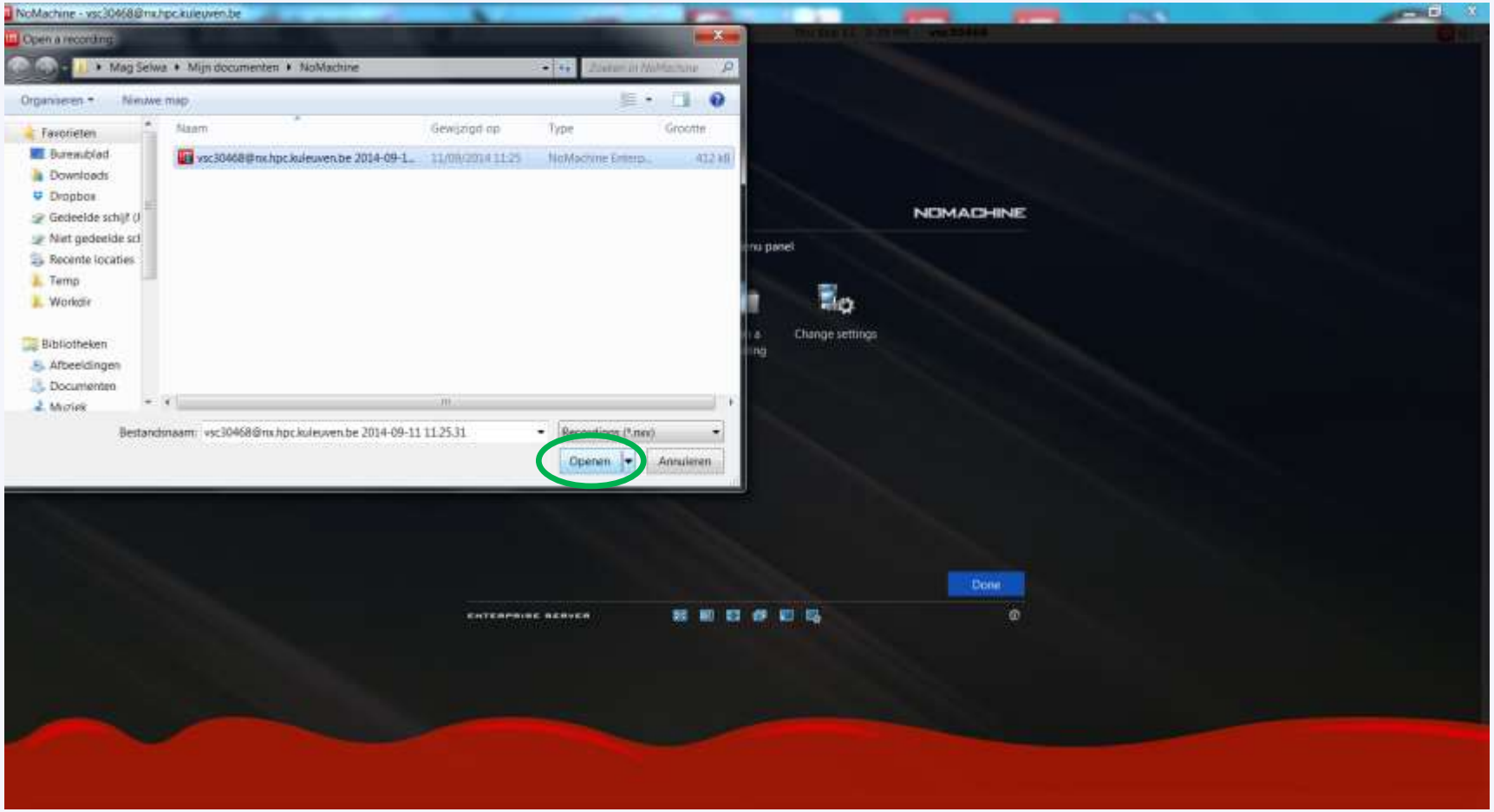
System tools - recording



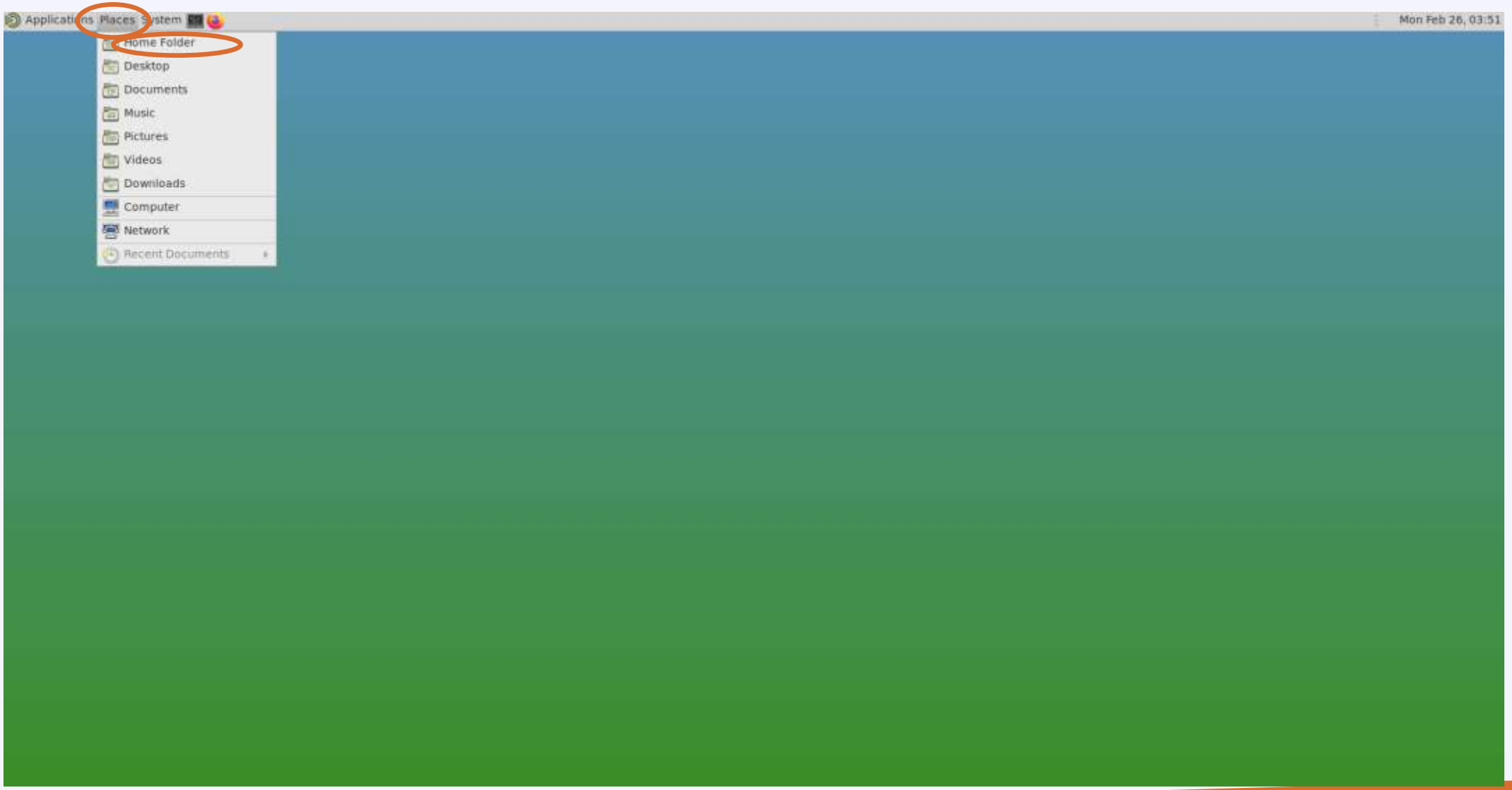
System tools



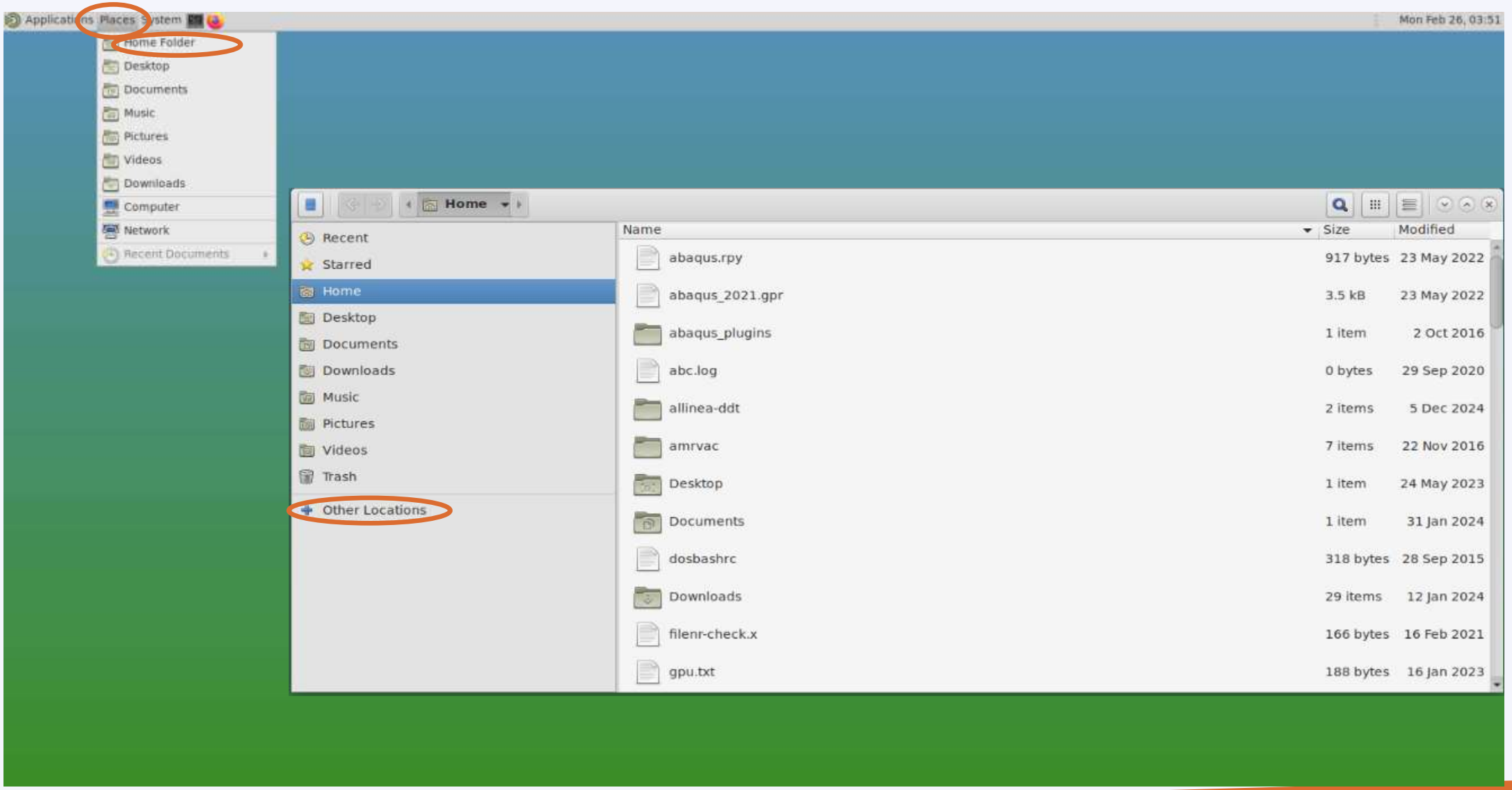
System tools - recording



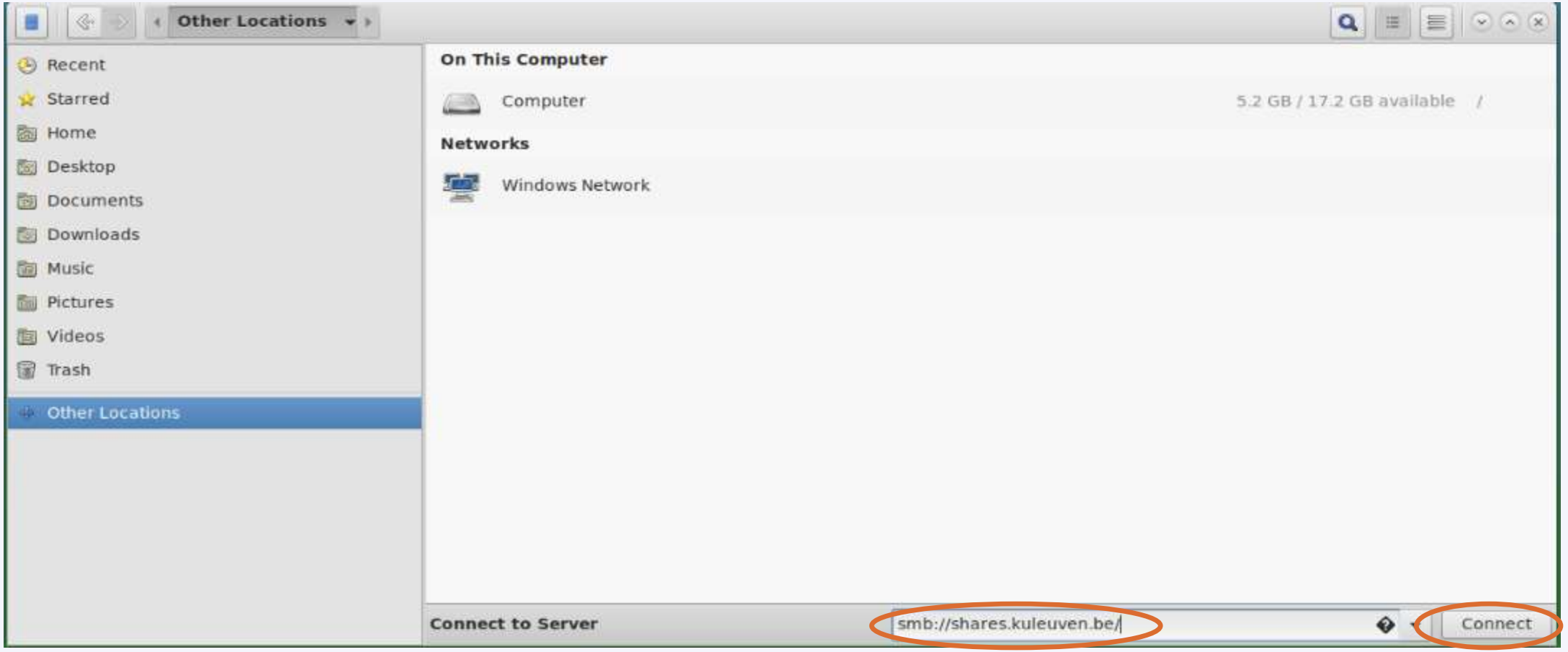
Connecting shared drives (Windows)



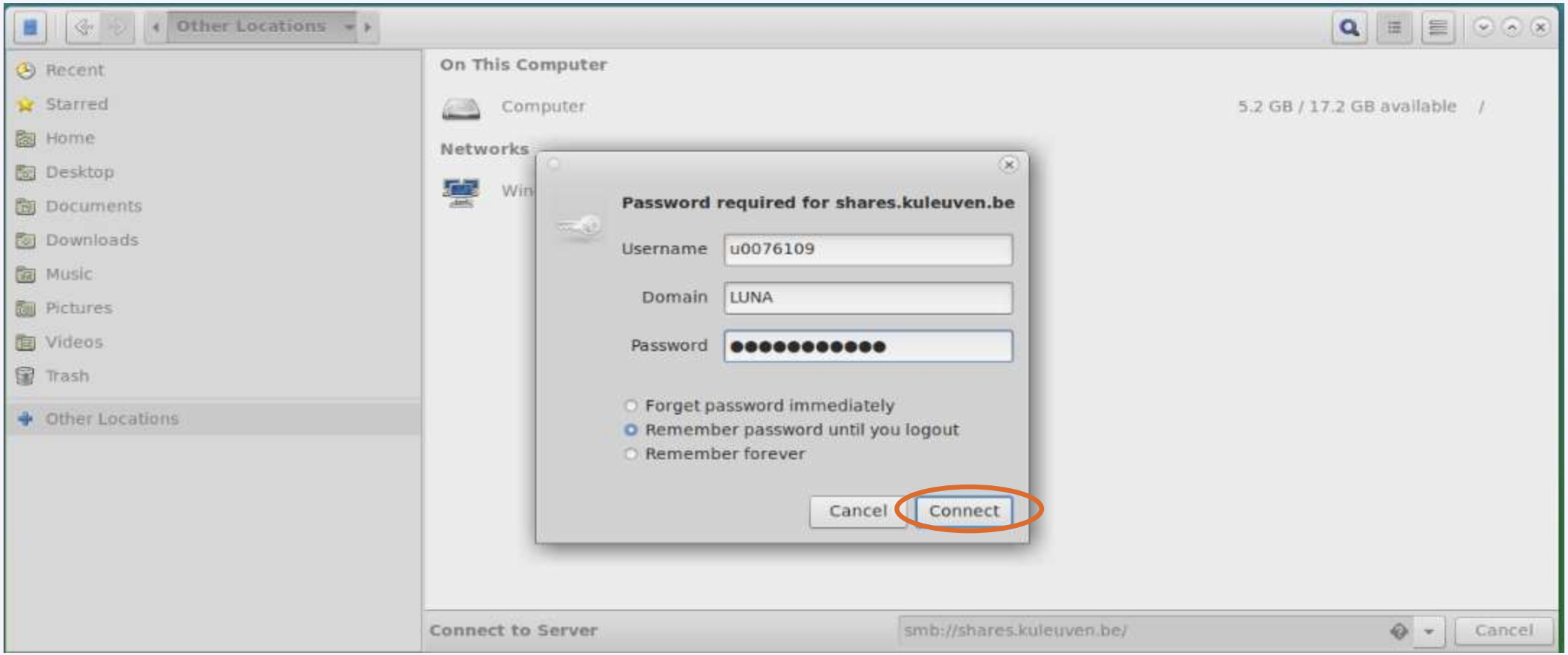
Connecting shared drives (Windows)



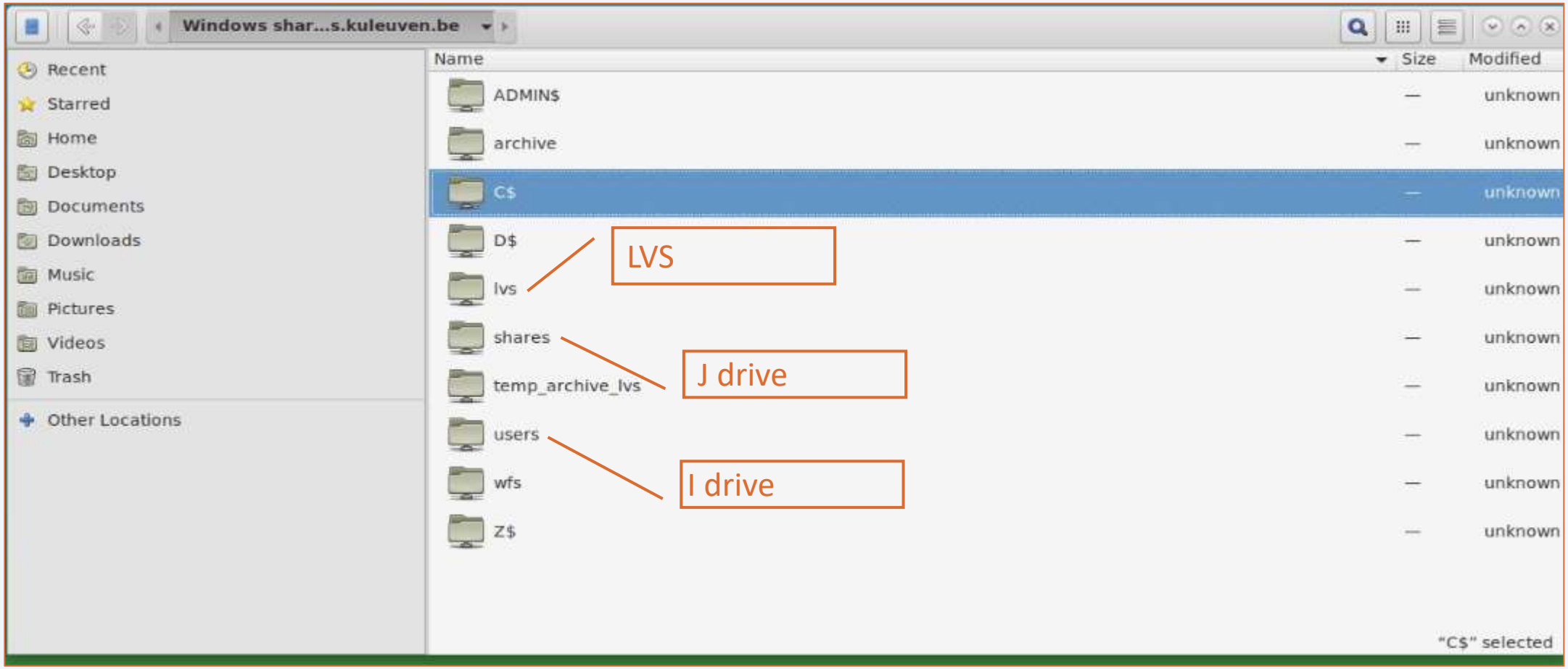
Connecting shared drives (Windows)



Connecting shared drives (Windows)



Connecting shared drives (Windows)



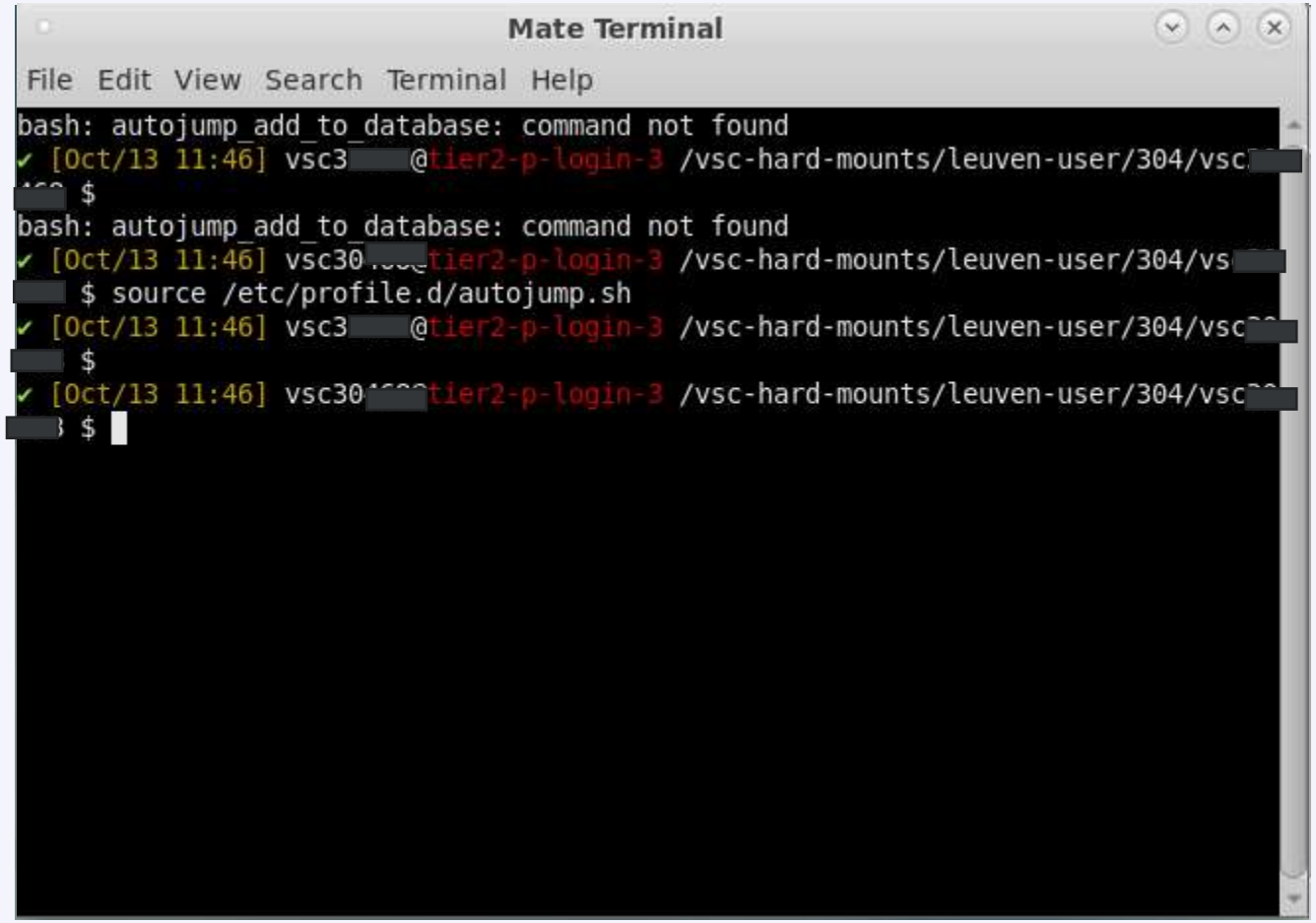
Connecting shared drives (Windows)

Server:
- smb://shares.kuleuven.be

Username: uXXXXX
Domain Name: LUNA
Password: password for u-number

Directories:
- lvs for LVS
- shares for J-drive
- users for I-drive

NX – adjusting terminal



To remove the autojump line
type in the terminal

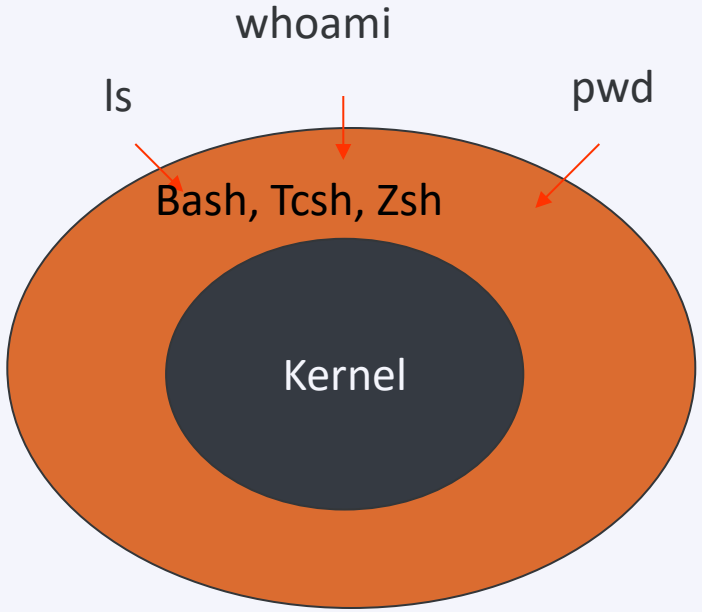
```
$ source
/etc/profile.d/autojump.sh
```

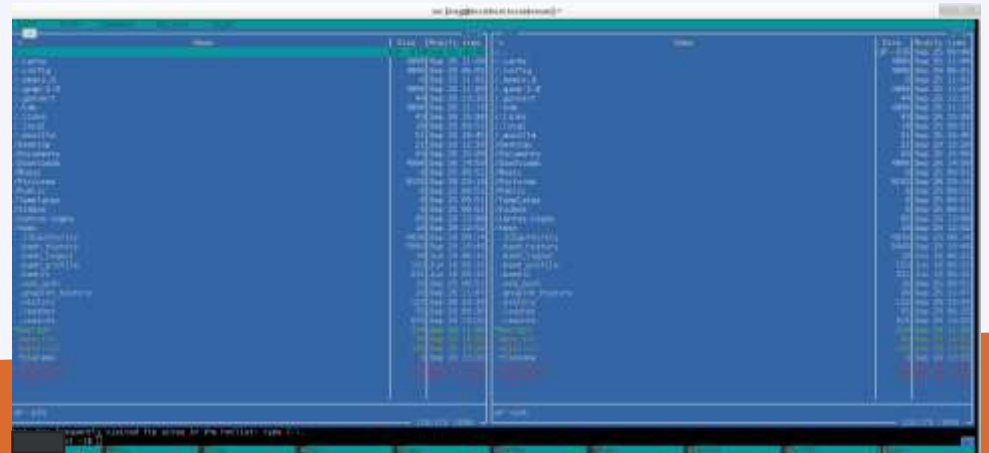
Or add into .bashrc file in your home directory

```
source /etc/profile.d/autojump.sh
```

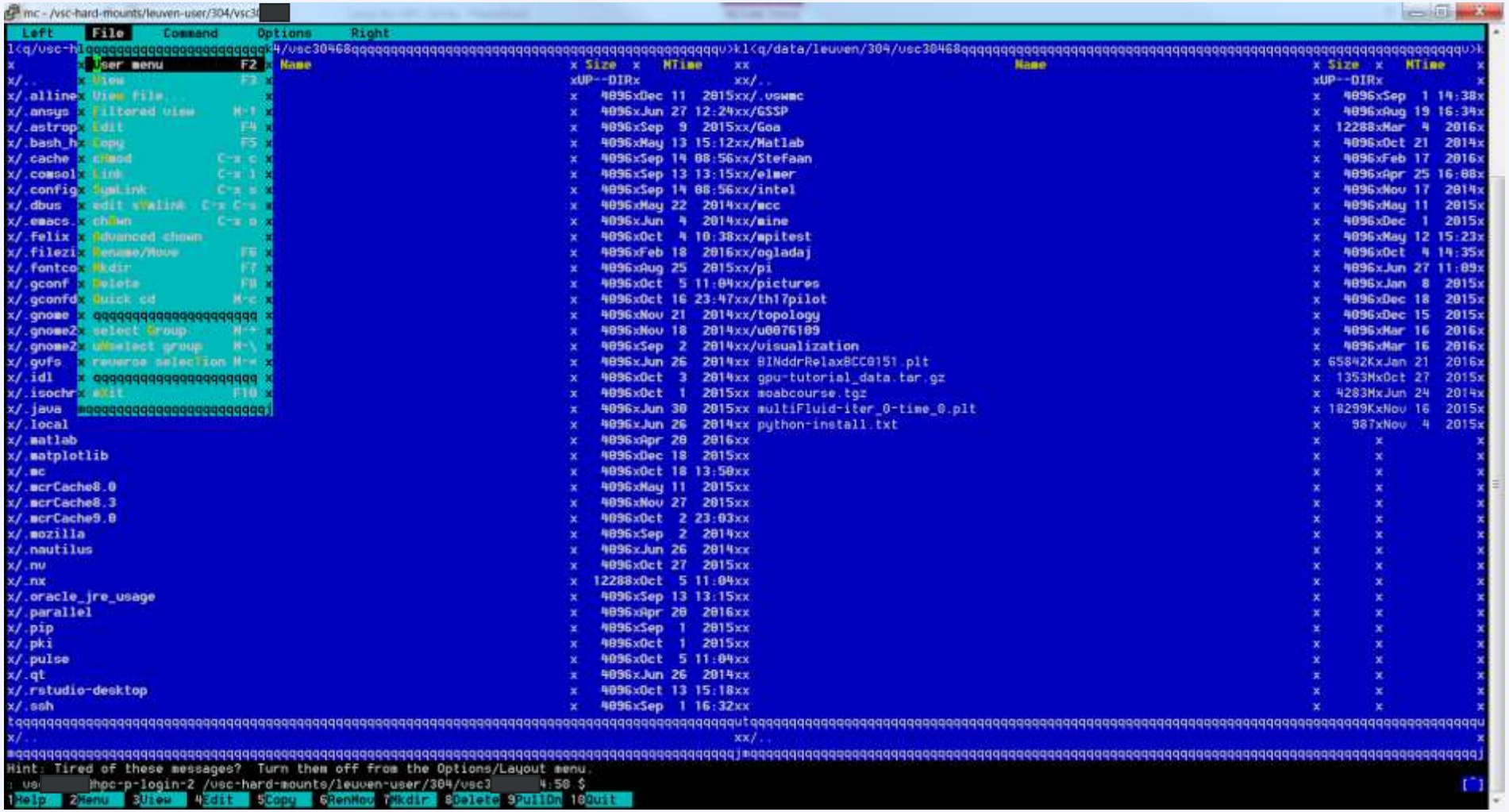
Linux shell

- Shell interprets the command and request service from kernel
- Similar to DOS but DOS has only one set of interface while Linux can select different shell
- Bourne Again shell (Bash), TC shell (Tcsh), Z shell (Zsh)
- Different shell has similar but different functionality
- Bash is the default for Linux
- Graphical user interface of Linux is in fact an application program work on the shell
- Special (visual) type of shell is mc (midnight commander) – easy to process files





Midnight commander



Midnight commander

- **Keyboard Shortcuts**

- In normal browsing mode:
  - F1 - help. More readable than the 2000-line man page, although difficult to browse.
  - F2 - user menu (offers option to gzip files, etc.)
  - F3 - view (handy to check the contents of an rpm or tgz file, or read contents of files)
  - F4 - edit with internal editor, mcedit
  - F5 - copy
  - F6 - rename or move
  - F7 - create a directory
  - F8 - delete
  - F9 - pull-down - accesses the menu bar at the top.
  - F10 - quit. Closes mc, as well as mcedit and any unwanted open menu.

Midnight commander

Selecting files

- Insert (Ctrl + t alternatively) - select files (for example, for copying, moving or deleting).
- + - select files based on a pattern.
- \ -unselect files based on a pattern.
- * - reverse selection. If nothing was selected, all files will get selected.

Accessing the shell

- There's a shell awaiting your command at the bottom of the screen - just start typing (when no other command dialog is open, of course).
- Since Tab is bound to switching panels (or moving the focus in dialogs), you have to use Esc Tab to use autocompletion. Hit it twice to get all the possible completions (just like in a shell). This works in dialogs too.
- If you want inspect the output of the command, do some input or just prefer a bigger console, no need to quit mc. Just hit **Ctrl + o** - the effect will be similar to putting mc in the background. Your current working directory from mc will be passed on to the shell… and vice versa! Hit Ctrl + o again to return to mc.

Midnight commander

Panels

- Alt + , - switch mc's layout from left-right to top-bottom. Useful for operating on files with long names.
- Alt + t - switch the panel's listing mode in a loop: default, brief, long, user-defined. "long" is especially useful, because it maximizes one panel so that it takes full width of the window and longer filenames fit on screen.
- Alt + i - synchronize the active panel with the other panel. That is, show the current directory in the other panel.
- Ctrl + u - swap panels.
- Alt + o - if the currently selected file is a directory, load that directory on the other panel and move the selection to the next file. If the currently selected file is not a directory, load the parent directory on the other panel and moves the selection to the next file. Ctrl + PgUp (or just left arrow, if you've enabled Lynx-like motion, see later) - move to the parent directory.
- Alt + Shift + h - show the directory history. Might be easier to navigate than going back one entry at a time.
- Alt + y - move to the previous directory in history.
- Alt + u - move to the next directory in history.

Midnight commander

Searching files

- Alt + ? - shows the full Find dialog.
- Alt + s or Ctrl + s - quick search mode. Start typing and the selection will move to the first matching file. Press the shortcut again to jump to another match. Use wildcards (*, ?) for easier matching.

Common actions

- Ctrl + Space - calculate the size of the selected directories. Press this shortcut when the selection is on .. to calculate the size of all the directories in the current directory.
- Ctrl + x s (that is press Ctrl + x, let it go and then press s) - create a symbolic link (change s to l for a hardlink). I find it very useful and intuitive - the link will, of course, be created in the other panel. You can change it’s destination and name, like with any other file operation.
- Ctrl + x c - open the chmod dialog.
- Ctrl + x o - open the chown dialog.

Hands-on 1

VSC-related commands

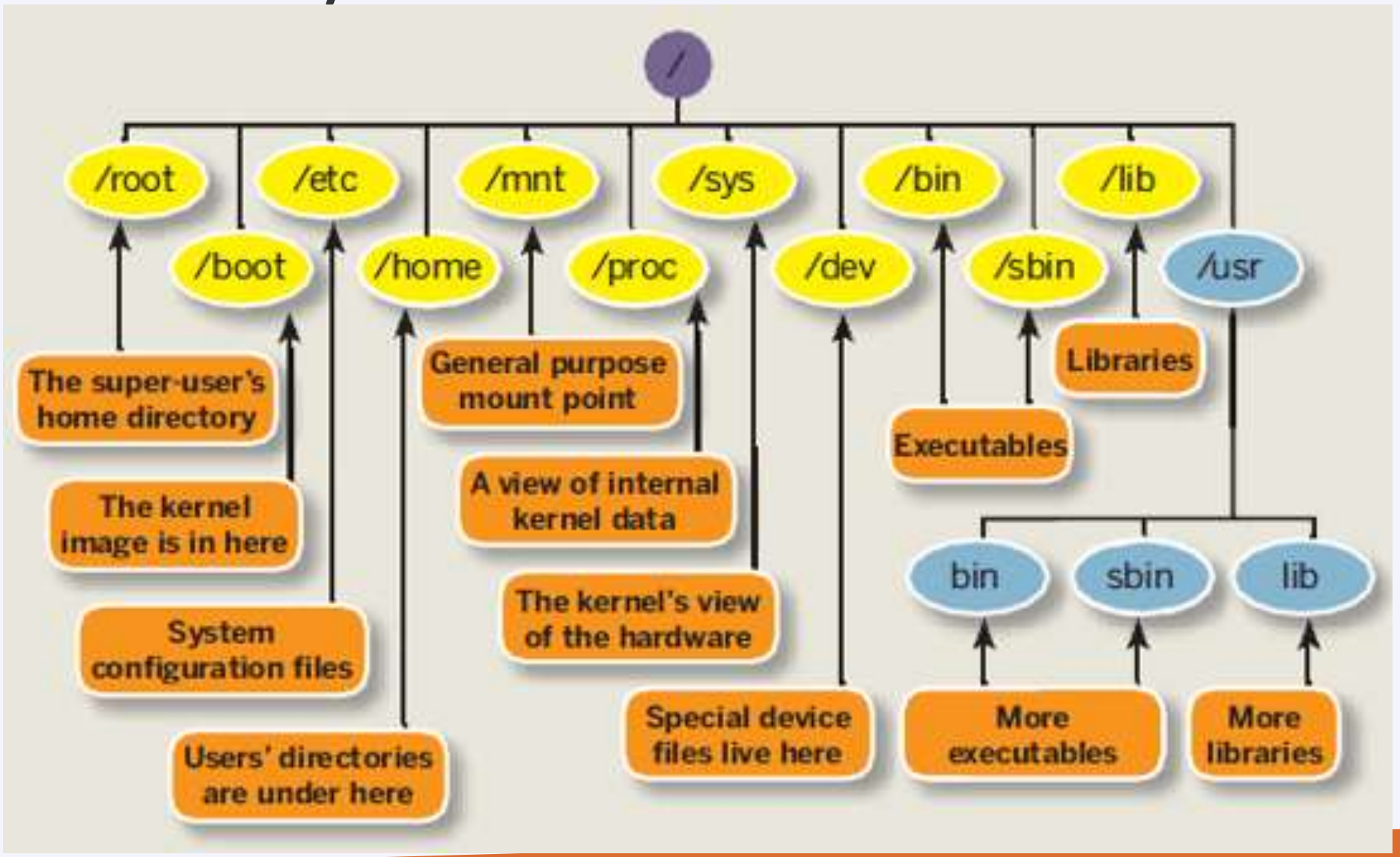
Information about users and processes

- `who`
  Lists all the users logged on the system.
- `groups`
  Tells which groups I belong to – important to check if already assigned to credits
- `top`
  Displays all processes (change with < or > for different parameters).
- `pstree`
  Displays process tree (`pstree -u $USER`).
- `tree`
  Lists contents of directories in a tree-like format (`tree $VSC_HOME`).

Temporary storage

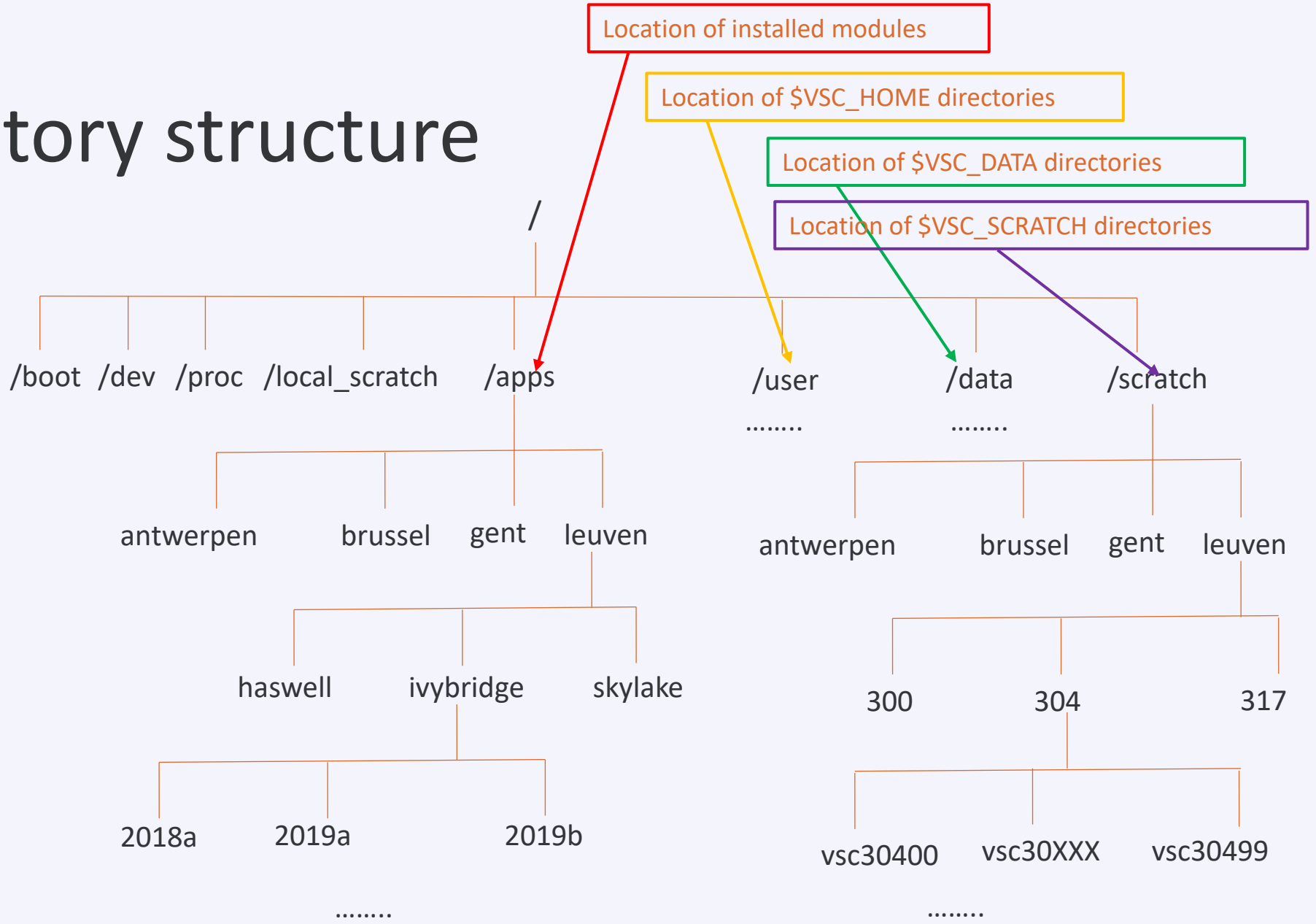
- Temporary storage on the cluster nodes has a different location compared to regular Linux system.
- `$VSC_SCRATCH_NODE` defines space that is available per node (to be used only during the job execution, need to copy the data as everything will be erased after the job ends).
- Do not use `/tmp` directory on compute node (very limited space ~10GB, once exceeded the system and your job will crash).
- Use `$VSC_SCRATCH_NODE` (`/local_scratch`) instead (~200GB)

Linux File System

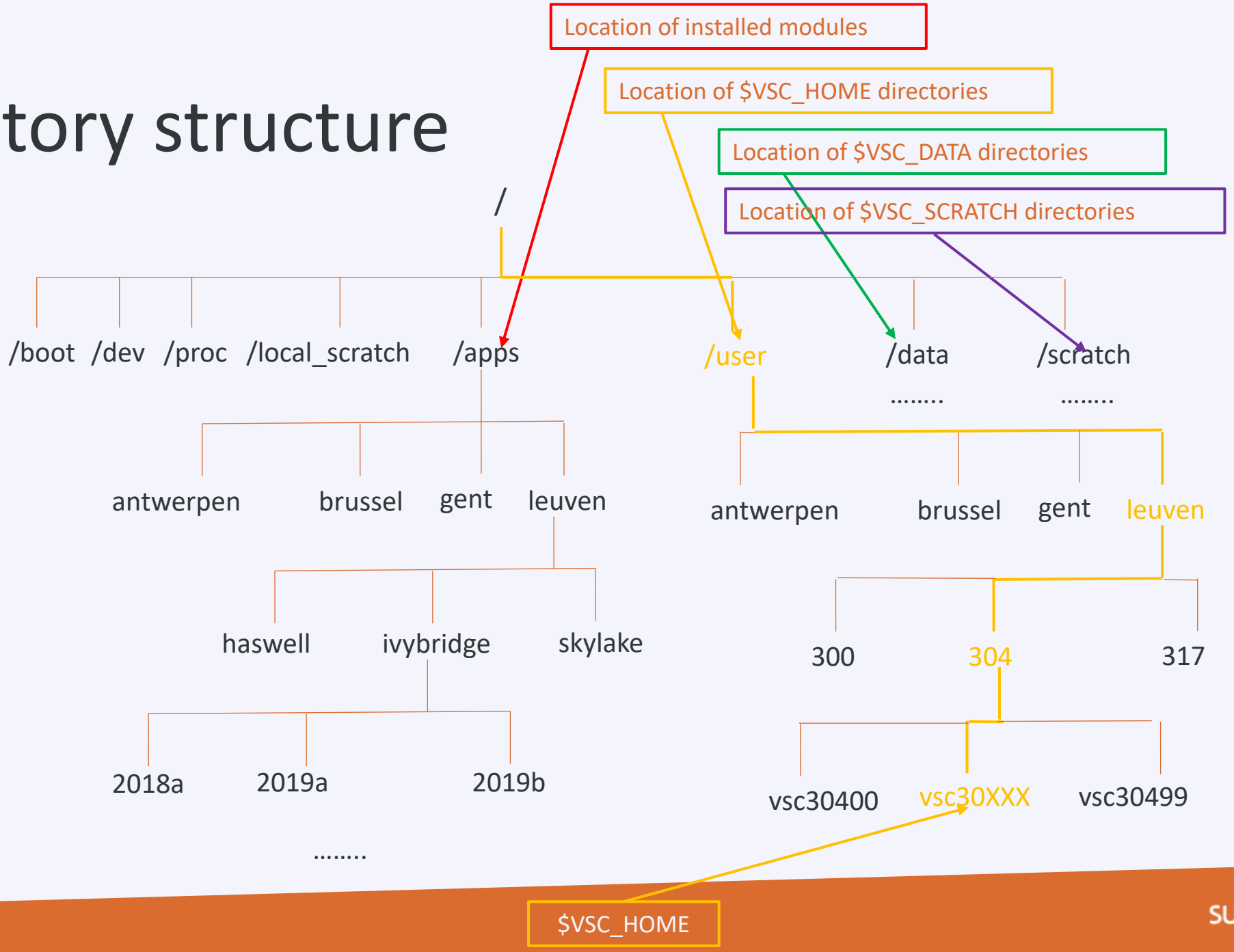


Source: http://linuxsuperuser07.blogspot.be/2011/09/rhel-6-file-system.html

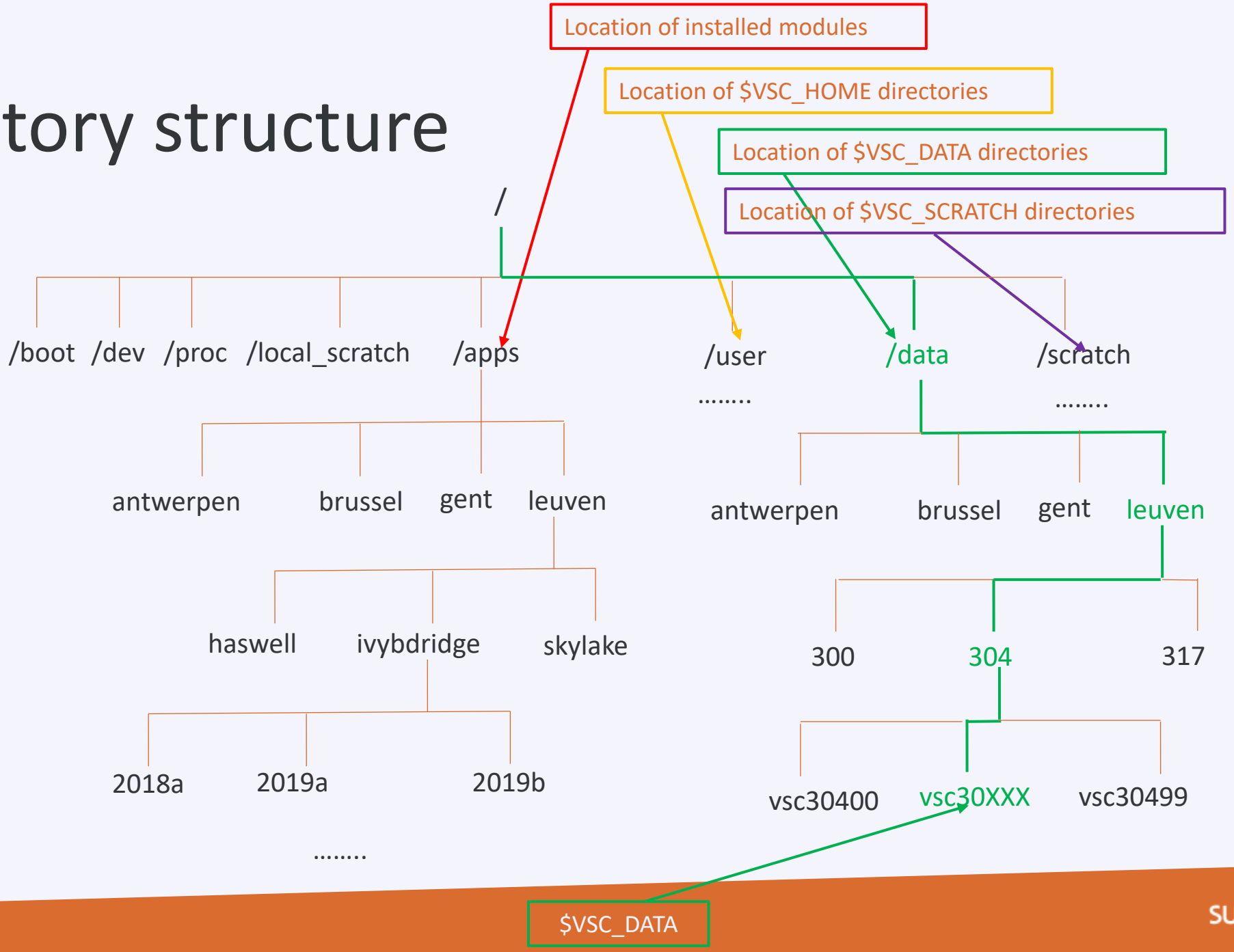
Directory structure



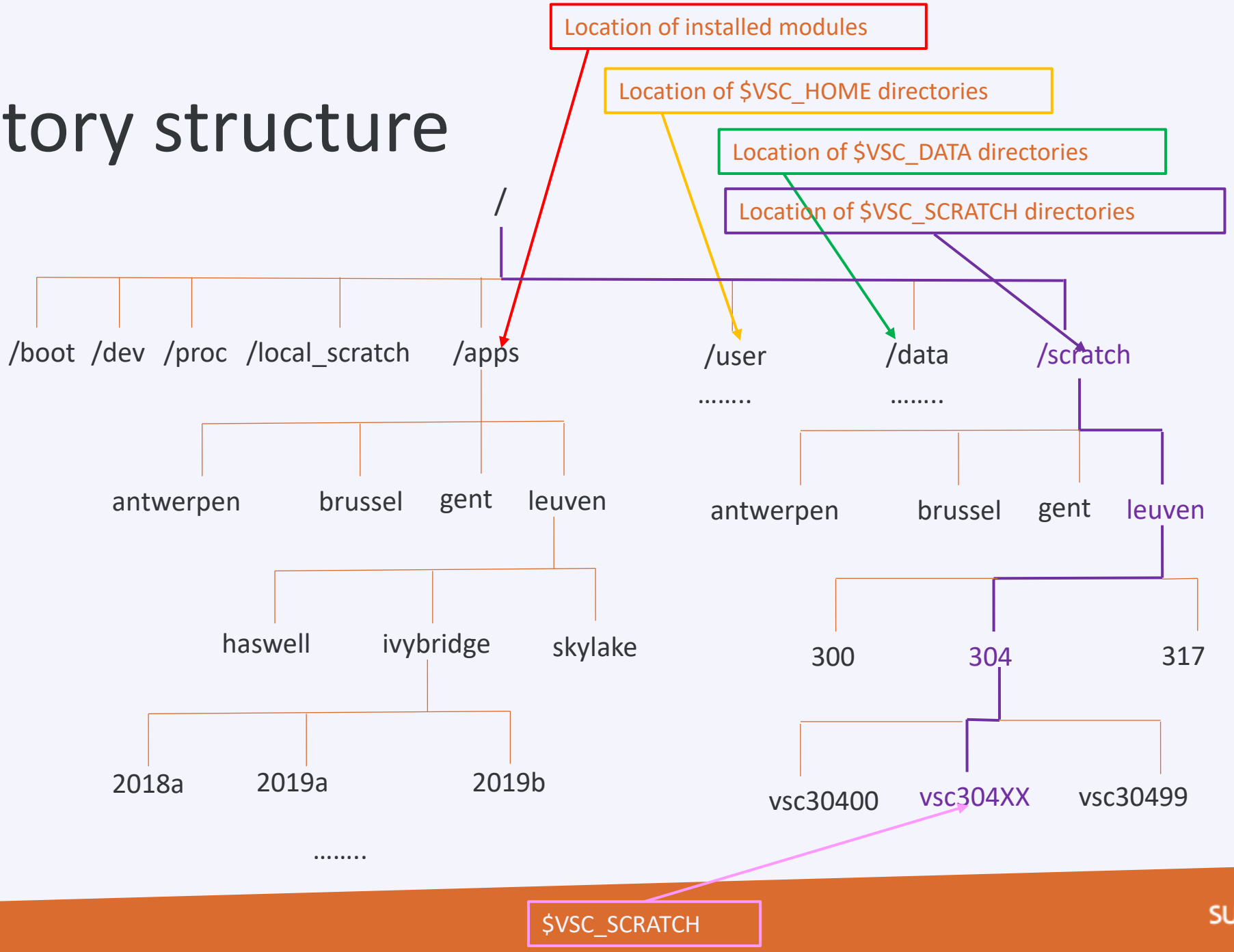
Directory structure



Directory structure



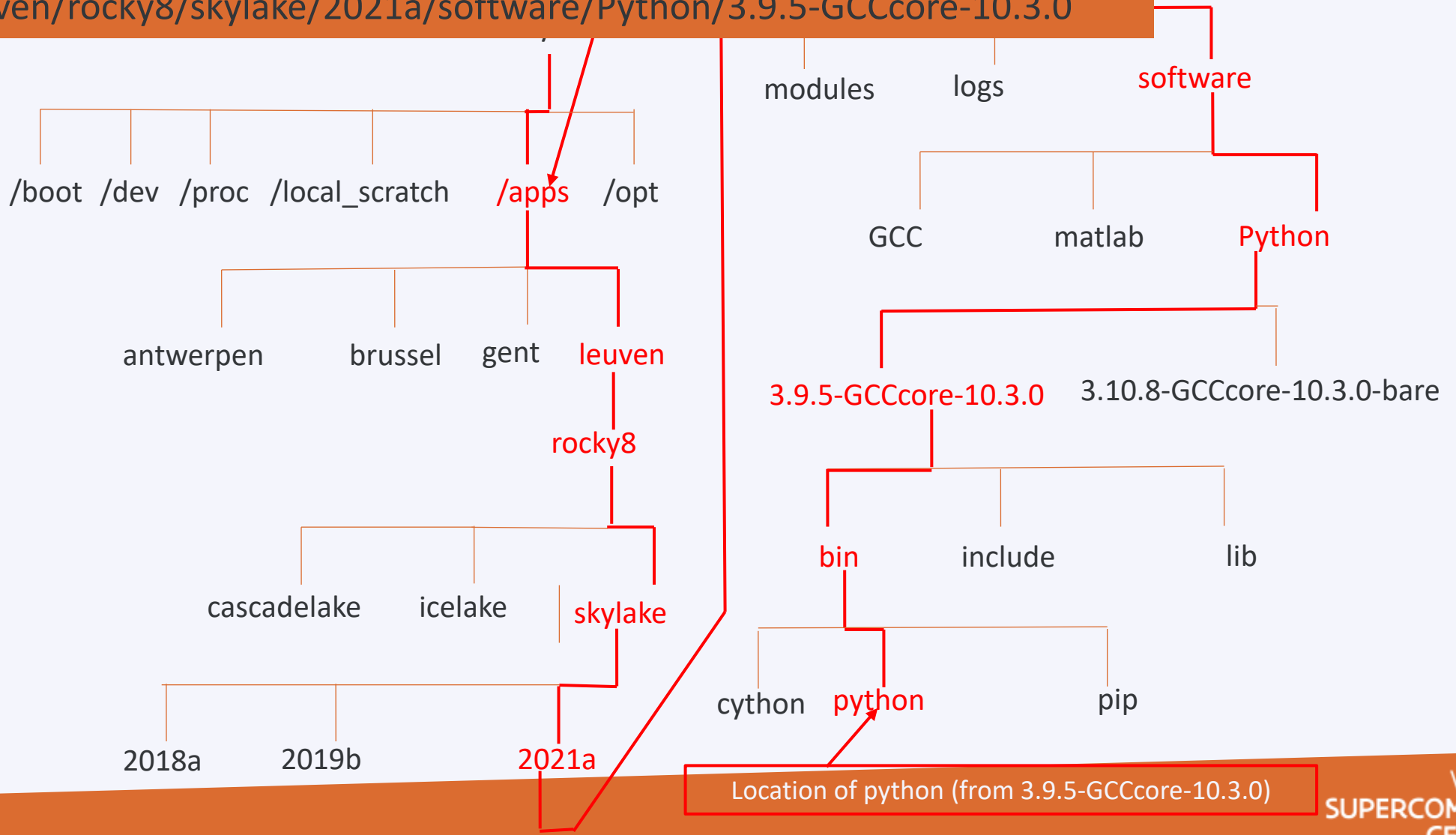
Directory structure



Location of installed modules Genius

Path on Genius:

/apps/leuven/rocky8/skylake/2021a/software/Python/3.9.5-GCCcore-10.3.0

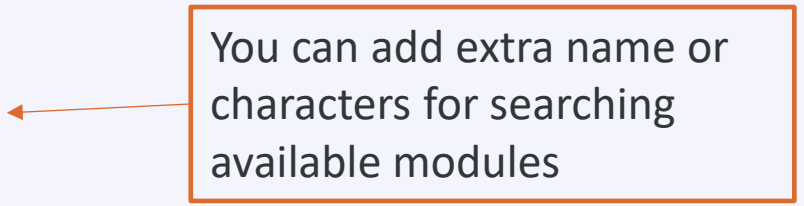


Location of python (from 3.9.5-GCCcore-10.3.0)

Modules

Set the environment to use software packages:

- `$ module available` or `module av R`
  - Lists all installed software packages
- `$ module av |& grep -i python`
  - To show only the modules that have the string 'python' in their name, regardless of the case
- `$ module list`
  - Lists all ‘loaded’ modules in current session
- `$ module load matlab/R2014a`
  - Adds the ‘matlab’ command in your PATH
- `$ module load GCC`
  - ‘Load’ the (default) GCC version – not recommended (cannot be reproduced)
- `$ module unload R/3.1.2-foss-2014a-x11`
  - Removes all only the selected module, other loaded modules – dependencies are still loaded
- `$ module purge`
  - Removes all loaded modules from your environment



Modules

- A few small tips for HPC intros regarding modules:
  - `module -t av python | grep -i -e '^python'`
  - `module -r av '^Python'`
  - `module -r av '^Python.+foss.+'`

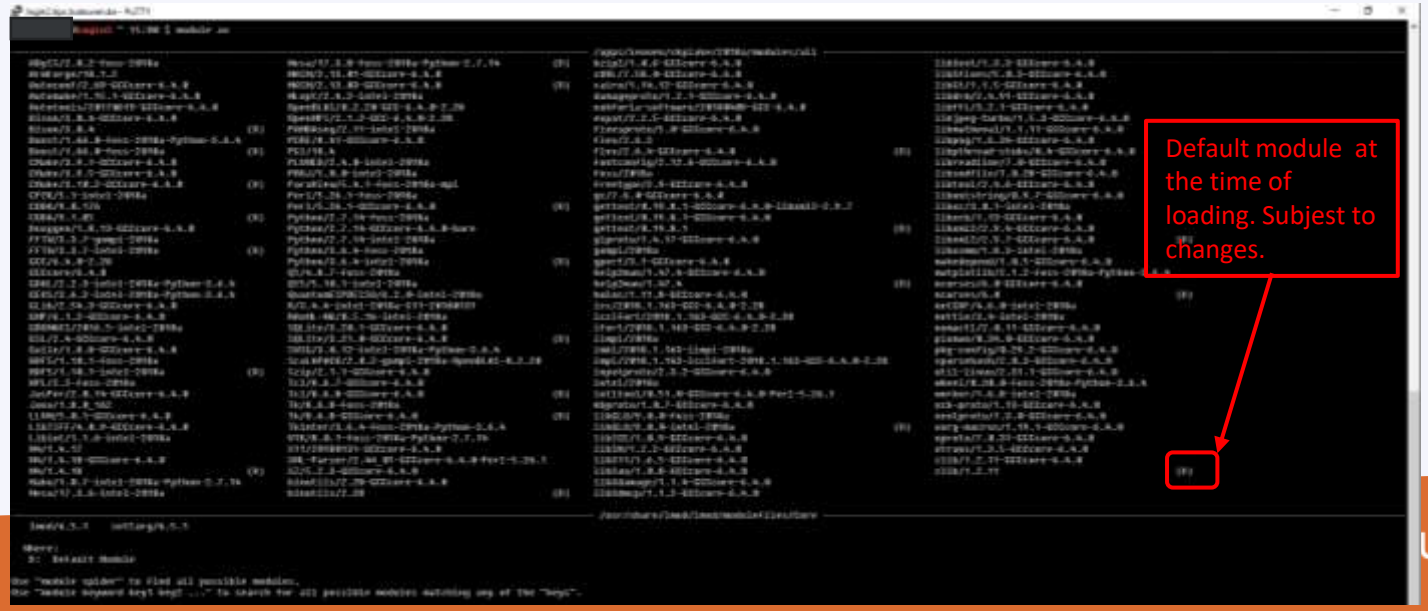
-i -> case insensitive
Works only for availibility, module load is always case sensitive

- The `-t` option forces terce output, i.e., something that is machine readable, and hence better to pipe through grep.

- The `-r` option allows you to use regular expressions, e.g., ^Python means that the module name should start with Python, while ^Python.+foss means that the module name should start with Python and contain Foss as a substring.

Modules

- By default all the software is listed (`$ module available`)
- The *modules* software manager is now **Lmod**. Lmod is a Lua based module system, but it is fully compatible with the TCL modulefiles we've used in the past. All the module commands that you are used to will work. But Lmod is somewhat faster and adds a few additional features on top of the old implementation.
- To (re)compile – ask for interactive job



Modules

- `$ module swap foss intel`
  - = module unload foss; module load intel
- `$ module try-load packageXYZ`
  - try to load a module with no error message if it does not exist
- `$ module keyword word1 word2 ...`
  - Keyword searching tool, searches any help message or whatis description for the word(s) given on the command line
- `$ module help foss`
  - Prints help message from modulefile
- `$ module spider foss`
  - Describes the module

Modules

- **ml** – convenient tool
- `$ ml`
  - = module list
- `$ ml foss`
  - =module load foss
- `$ ml -foss`
  - =module unload foss (not purge!)
- `$ ml show foss`
  - Info about the module

- Possible to create user collections:
  - module save <collection-name>
  - module restore <collection-name>
  - module describe <collection-name>
  - module savelist
  - module disable <collection-name>

More info: http://lmod.readthedocs.io/en/latest/010_user.html

Find the right command

- Executing the right command can be vital for your system. However in Linux there are so many different command lines that they are often hard to remember. So how do you search for the right command you need? The answer is **apropos**:
- ```
  $ apropos ownership
    chgrp (1)              - change group ownership
    chgrp (1p)             - change the file group ownership
    chown (1p)             - change the file ownership
    chown (2)              - change ownership of a file
    chown32 (2)            - change ownership of a file
    fchown (2)             - change ownership of a file
    fchown32 (2)           - change ownership of a file
    ……
  ```
  ```

Monitoring the system:

- `free`
  - Displays the status of RAM and VRAM
  - Mem: refers to RAM
  - Swap: refers to virtual RAM (the swap file)
  - Too little RAM will cause ‘thrashing’ (constantly moving information from RAM to VRAM)
- `top`
  - Displayes all the tasks, but also available CPU and memory in the top bar
- File `/proc/cpuinfo (/proc/meminfo)`
  - contains info about procesor/memory, no CPU usage
- GPUs: `nvidia-smi (watch “nvidia-smi”)`

scp

- The scp command allows you to copy files over ssh connections.
- scp examplefile

```
touch test.txt
scp $HOME/test.txt vsc30XXX@login.hpc.kuleuven.be:$VSC_DATA/
```

Reconginzed only on the cluster, not on the remote host

Syntax similar to cp:

scp location-of-the-file-on-one-computer location-of-the-file-on-second-computer

scp

Own PC

VSC cluster

File exists: /home/mag/test.txt

want to copy to the cluster into

$VSC_DATA

```
scp /home/mag/test.txt
vsc30XXX@login.hpc.kuleuven.be:$VSC_DATA/
```

Reconginzed only on the cluster, not on the remote host

```
scp /home/mag/test.txt
vsc30XXX@login.hpc.kuleuven.be:/data/leuven/304/vsc30XXX
```

Path to the file can be checked with e.g. `echo $VSC_DATA` on the cluster

rsync

- `rsync`, which stands for remote sync, is a remote and local file synchronization tool. It uses an algorithm to minimize the amount of data copied by only moving the portions of files that have changed.
- `rsync options source destination`
- -v : verbose
- -r : copies data recursively (but don’t preserve timestamps and permission while transferring data.
- -a : archive mode, which allows copying files recursively and it also preserves symbolic links, file permissions, user & group ownerships, and timestamps.
- -z : compress file data.
- -h : human-readable, output numbers in a human-readable format.

rsync

- Let us prepare some files first

```
[Dec/02 20:49] vsc3____@tier2-p-login-3 ~/course/Linux-HPC $ mkdir dir1
[Dec/02 20:49] vsc3____@tier2-p-login-3 ~/course/Linux-HPC $ mkdir dir2
[Dec/02 20:49] vsc3____@tier2-p-login-3 ~/course/Linux-HPC $ touch dir1/file{1..100}
[Dec/02 20:50] vsc3____@tier2-p-login-3 ~/course/Linux-HPC $ ls dir1
file1    file16  file23  file30  file38  file45  file52  file6   file67  file74  file81  file89  file96
file10   file17  file24  file31  file39  file46  file53  file60  file68  file75  file82  file9   file97
file100  file18  file25  file32  file4   file47  file54  file61  file69  file76  file83  file90  file98
file11   file19  file26  file33  file40  file48  file55  file62  file7   file77  file84  file91  file99
file12   file2   file27  file34  file41  file49  file56  file63  file70  file78  file85  file92
file13   file20  file28  file35  file42  file5   file57  file64  file71  file79  file86  file93
file14   file21  file29  file36  file43  file50  file58  file65  file72  file8   file87  file94
file15   file22  file3   file37  file44  file51  file59  file66  file73  file80  file88  file95
```

- To sync the contents of dir1 to dir2 on the same system, you will run rsync and use the -r flag, which stands for “recursive” and is necessary for directory syncing:
- `$ rsync -r dir1/ dir2`

rsync

- Another option is to use the `-a` flag, which is a combination flag and stands for “archive”. This flag syncs recursively and preserves symbolic links, special and device files, modification times, groups, owners, and permissions. It’s more commonly used than `-r` and is the recommended flag to use.
- Another tip is to double-check your arguments before executing an rsync command. Rsync provides a method for doing this by passing the `-n` or `--dry-run` options. The `-v` flag, which means “verbose”, is also necessary to get the appropriate output. We can combine the a, n, and v together.
- `$ rsync -anv dir1/ dir2`

vs

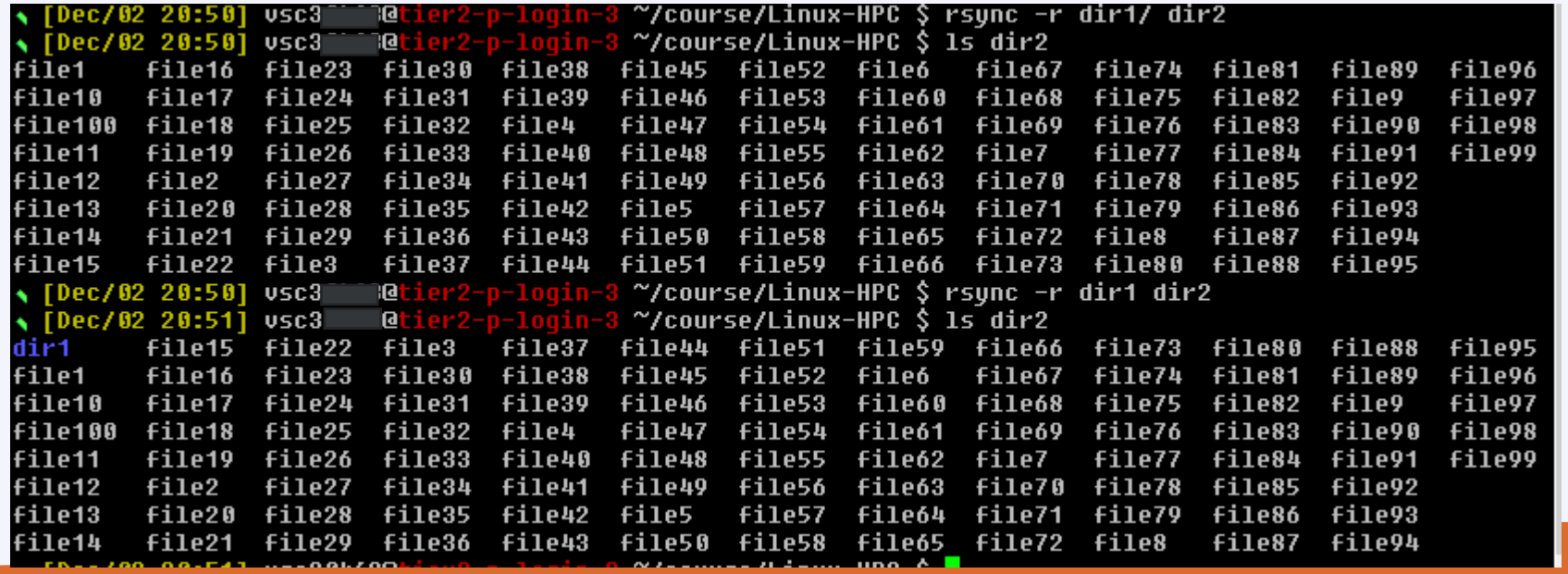
- `$ rsync -anv dir1 dir2`

rsync

- there is a trailing slash (/) at the end of the first argument in the

```
$ rsync -a dir1/ dir2
```

- This trailing slash signifies the contents of dir1. Without the trailing slash, dir1, including the directory, would be placed within dir2. The outcome would create a hierarchy like the following: `~/dir2/dir1/[files]`



rsync with remote system

- To use rsync to sync with a remote system, we only need SSH access configured between your local and remote machines, as well as `rsync` installed on **both** systems. Having SSH access verified between the two machines, we can sync the dir1 folder from the previous section to a remote machine. In this case, that you want to transfer the actual directory, so we will omit the trailing slash:

```
$ rsync -a ~/dir1
vsc3XXXX@login.hpc.kuleuven.be:destination_directory
```

rsync with remote system

- This process is called a push operation because it “pushes” a directory from the local system to a remote system. The opposite operation is pull, and is used to sync a remote directory to the local system. If the dir1 directory were on the remote system instead of your local system, the syntax would be the following:

```
$ rsync -a
vsc3XXXX@login.hpc.kuleuven.be:/data/leuven/3XX/vsc3XXX
X/dir1 place_to_sync_on_local_machine
```

- Like cp and similar tools, the source is always the first argument, and the destination is always the second.

rsync extra flags

- If you're transferring files that have not already been compressed, like text files, you can reduce the network transfer by adding compression with the -z option:

```
$ rsync -az source destination
```

- The -P flag is also helpful. It combines the flags --progress and --partial. This first flag provides a progress bar for the transfers, and the second flag allows to resume interrupted transfers:

```
$ rsync -azP source destination
```

- If you run the command again, you'll receive a shortened output since no changes have been made. This illustrates rsync's ability to use modification times to determine if changes have been made:

rsync extra flags

- In order to keep two directories truly in sync, it's necessary to **delete** files from the destination directory if they are removed from the source. By default, `rsync` does not delete anything from the destination directory.
- You can change this behavior with the `--delete` option. Before using this option, you can use -n, the --dry-run option, to perform a test to prevent unwanted data loss:

```
$ rsync -an --delete source destination
```

rsync extra flags

- If you prefer to exclude certain files or directories located inside a directory you are syncing, you can do so by specifying them in a comma-separated list following the `--exclude=` option:

```
$ rsync -a --exclude=pattern_to_exclude source
destination
```

- If you have a specified pattern to exclude, you can override that exclusion for files that match a different pattern by using the --include= option:

```
$ rsync -a --exclude=pattern_to_exclude --
include=pattern_to_include source destination
```

rsync extra flags

- Finally, rsync's `--backup` option can be used to store backups of important files. It's used in conjunction with the `--backup-dir` option, which specifies the directory where the backup files should be stored:

```
$ rsync -a --delete --backup --backup-
dir=/path/to/backups /path/to/source destination
```

Searching

- A large majority of activity on UNIX systems involve searching for files and information.
  - **find** – utility to find files

    ```
    find $VSC_HOME -name "test9*"
    ```

    searches for files with name starting with test9 in $VSC_HOME directory
  - **grep** – great utility, searches for patterns inside files and will return the line, if found

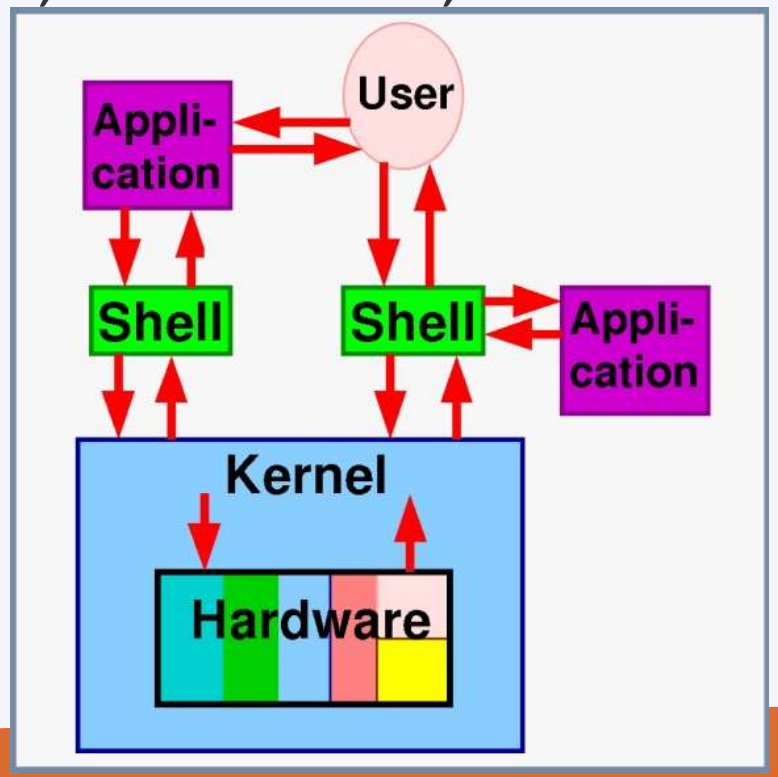
    ```
    grep -H -R  test9 $VSC_HOME
    ```

    searches for files with test9 in their text in $VSC_HOME directory

The Shell revisited: features

The shell

- Not just an interface to the computer, also a scripting language – allows automation of tasks
- Shells can be scripted: provide all the resources to write complex programs (variables, conditionals, iterations...)



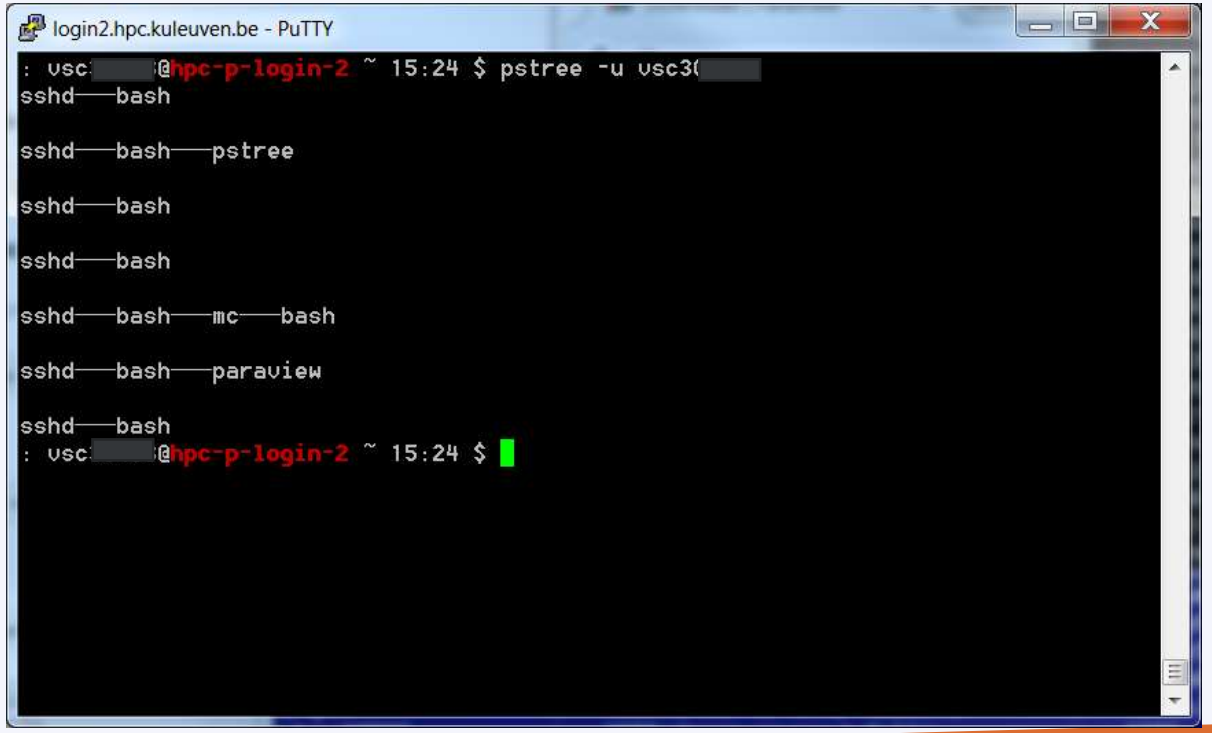
- Upon startup, shell executes commands found in the ~/.*rc file, allowing users to customize their shell.

Source: opensuse.org

The shell

- After login to the cluster or starting a job: a new shell is opened
- Processes originating from each shell can be checked with `pstree`

  - `pstree` shows only processes on the current node (login1, or login2 or NX or compute node)



Most popular shells

- There are several types of shells for Linux.
- Check it with

```
$ echo $SHELL
```

Shell	Prompt	Name	Note
sh	$	Bourne Shell	Default on some Unix systems
bash	$	Bourne Again Shell	Enhanced replacement for the Bourne shell Default on most Linux and Mac OS X systems
csh	%	C Shell	Default on many BSD systems
tcsh	>	TC Shell	Enhanced replacement for the C shell
ksh	$	Korn Shell	Default on AIX systems

Starting shells

- In practice, users seldom need to start a shell manually. Whenever someone logs in, or opens a terminal, a shell is started automatically.
- You can start a different shell, or another instance of the same shell. Because the shell is "just another" program, new shells can be launched from an existing shell.
- The new shell is referred to as a **subshell** of the original shell. When the subshell is exited, control is returned to the original shell.
- Subshell inherits all variables/context from the parent shell.
- If you quit the subshell, its context will purge.
- The apparent differences between the subshell and the parent shell are minimal, and care must be taken to keep track of which shell you are in.

Return/Exit Code

- Shell commands are **CASE SENSITIVE!**
- Upon exiting, every command returns an integer to its parent called a return value.
- The shell variable $? expands to the return value of previously executed command (e.g. 0 when success).

  compare:

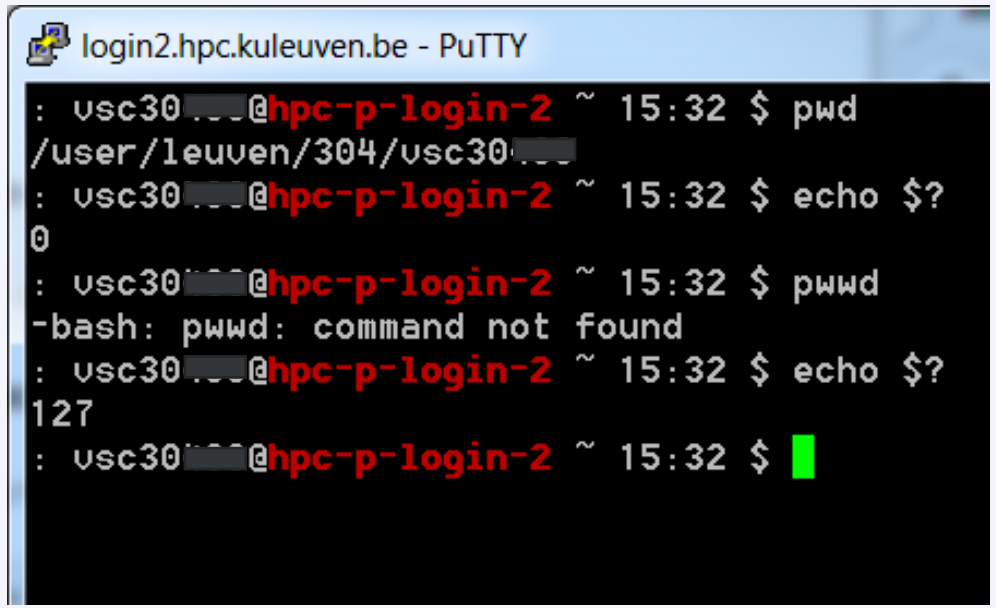
  `pwd`

  `echo $?`

  with

  `pwwd` (does not exist, mistyped)

  `echo $?`



AND &&

- Multiple commands can be separated with a **;**

  e.g. `cd $VSC_HOME;pwd`

- **&&** and **||** conditionally separate multiple commands.
- When commands are conditionally joined, the first will always execute. The second command may execute or not, depending on the return value of the first command.
- For example, a user may want to create a directory, and then move a new file into that directory. If the creation of the directory fails, then there is no reason to move the file. The two commands can be coupled as follows:
  - `echo "one two three four five" > numbers.txt;`
  - `mkdir $VSC_DATA/my-dir && mv numbers.txt $VSC_DATA/my-dir`

OR ||

- Similarly, multiple commands can be combined with **||**.
- In this case, **bash** will execute the second command only if the first command "fails" (has a non zero return value). This is similar to the "or" operator found in programming languages.
- E.g., we attempt to change the permissions on a file. If the command fails, a message to that effect is echoed to the screen.
  - `chmod 600 $VSC_DATA/my-dir/numbers.txt || echo "chmod failed"`
- In the first case, the **chmod** command succeeded, and no message was echoed. In the second case, the **chmod** command failed (because the file didn't exist), and the "chmod failed" message was echoed (in addition to **chmod**'s standard error message).

Escape character

- ", $, `, and \ are still interpreted by the shell, even when they're in double quotes.
- The backslash (\) character is used to mark these special characters so that they are not interpreted by the shell, but passed on to the command being run (for example, echo)
- E.g. to output the string: (Assuming that the value of $X is 5):

```
A quote is ", backslash is \, backtick is `.
A few spaces are   and dollar is $. $X is 5.
```

we would have to write:

```
$ echo "A quote is \", backslash is \\, backtick is \`."
A quote is ", backslash is \, backtick is `.

$ echo "A few spaces are   ; dollar is \$. \$X is ${X}."
A few spaces are   ; dollar is $. $X is 5.
```

Escape character

- $ is used for interpreting variable which has some value assigned

- You can define a variable locally in the shell, e.g.
  `now=`date``
  and use it later on in your shell/scripts:
  `echo "Now, the date/hour is: ${now}"`
- You can define a variable, and let subshell inherit its value

```
export now=`date`     # in parent shell
bash                  # start the child/sub shell
echo "Subshell: time in parent shell was: ${now}"
```

- The `export` command defines a variable in the parent and child shells

Escape character

- When you create a file that contains space in it, e.g. `touch "my file"` it is difficult to use it later
  - How to copy the file (cp source destination)
  - -> use escape character so that space is understood as a part of the file and not as a separator in comand syntax
  - `cp my\ file myfile`

- Better avoid using "special" characters (", $, `, \, …) in your filenames!

Auto-Completion

- Have the shell automatically complete commands or file paths.
- Activated using the **<TAB>** key on most systems
- examples
  - **$ whe<TAB>**
  - **$ whereis**
  - **$ ls -l /etc/en<TAB>**
  - **$ ls -l /etc/environment**
- When more than one match is found, the shell will display all matching results (use **<TAB>** twice)
  - **$ ls -l /etc/host<TAB>**

Globbing: use wildcard

Wildcard	Function
*	Matches 0 or more characters
?	Matches 1 character
[abc]	Matches one of the characters listed
[a-c]	Matches one character in the range
[!abc]	Matches any character not listed
[!a-c]	Matches any character not listed in the range
{tacos,nachos}	Matches one word in the list

```
$ ls -l /etc/host*
$ ls -l /etc/hosts.{allow,deny}
$ ls -l /etc/hosts.[!a]*
$ ls -l /etc/host?
```

quoting

- Double (") quotes can be used:
  - to prevent the shell from interpreting spaces as argument separators,
  - to prevent file name pattern expansion.

```
$echo "Hello World"
Hello World
$echo "You are logged as $USER"
You are logged as vsc30XXX
$echo *.log
$echo "*.log"
*.log
```

quoting

- Single quotes bring a similar functionality, but what is between quotes is never substituted

```
$echo 'You are logged as $USER'
You are logged as $USER
```

- Back quotes (`) can be used to call a command within another

```
$cd /lib/modules/`uname -r`; pwd
/lib/modules/2.6.9-1.6_FC2
```

Back quotes can be used within double quotes

```
$echo "You are using Linux `uname -r`"
You are using Linux 2.6.9-1.6_FC2
```

What was I doing???

- Not to loose your job after closing your laptop:
  - Use NX GUI connection
  - Use command line + tmux
    - Start session: `$ tmux new -s test`
    - Detach session: `Ctrl+b+d` (safe to go)
    - List session: `$ tmux ls`
    - Reattach session: `$ tmux a -t test`
    - Exit screen session (within tmux): `$ exit`

What was I doing???

Manage Windows and Session Tabs

Ctrl+B C	Create window
Ctrl+B W	List Windows
Ctrl+B N	Next window
Ctrl+B P	Previous window
Ctrl+B F	Find window
Ctrl+B ,	Name window
Ctrl+B &	Kill window

What was I doing???

Panes/splits

Ctrl+B %	Vertical split
Ctrl+B "	Horizontal split
Ctrl+B O	Swap panes
Ctrl+B Q	Show pane numbers
Ctrl+B X	Kill pane
Ctrl+B Arrow Keys	Move to pane

What was I doing???

- Not to loose your job after closing your laptop:
  - Use NX GUI connection
  - Use command line + screen
    - Start session: `$ screen -S test`
    - Detach session: `Ctrl+a+d` (safe to go)
    - List session: `$ screen -ls`
    - Reattach session: `$ screen -r test`
    - Exit screen session (within screen): `$ exit`

Screen

- Create new window: ctrl-a c
- Go to previous/next window: ctrl-a p/n
- Go to window by number: ctrl-a <window-nr>
- Show current windows, move: ctrl-a ", <window-nr>
- Close window: ctrl-a K
- Detach screen: ctrl-a d
- List current screen sessions: $ screen -ls
- Re-attach to session: $ screen -r <session-id>
- Kill dead session: $ screen -wipe
- Get help: ctrl-a ?
- Monitor for activity: ctrl-a M (same to stop monitoring)
- Monitor for inactivity: ctrl-a _ (same to stop monitoring)

Screen

- Split screen horizontally: ctrl-a S
- Split screen vertically: ctrl-a |
- Go to next screen region: ctrl-a <tab>
- Remove current region: ctrl-a X
- Remove all but current region: ctrl-a Q
- Enter copy mode: ctrl-a [
- Paste: ctrl-a]
- Dump window contents to file: ctrl-a h
- Enable logging: ctrl-a H
- Useful .screenrc file that eliminates some of screen's nuisances:
  - # Turn off that annoying start up message
  - startup_message off
  - # Increase scroll back buffer to a more useful number of lines
  - defscrollback 10000

Screen - settings

- In your **`.bashrc`** file

```
case ${TERM} in
   xterm)
echo "Hello terminal!!!"
;;
   screen)
echo "Hello screen!!!"
;;
esac
```

&

- **`&`** is a command line operator that instructs the shell to start the specified program **in the background**.
- This allows you to have more than one program running at the same time without having to start multiple terminal sessions.
- Starting a process in background: add & at the end of your line:
  `gedit &`
  check with **`ps`**

Command history: Arrow Up

- Previously executed commands can be recalled by using the **Up Arrow** key on the keyboard.
- Most Linux distributions remember the last 500 commands by default.
- Display commands that have recently been executed
  - The `history` command displays a user's command line history.
  - You can execute a previous command using `![NUM]` where NUM is the line number in history you want to recall.
  - The **history** command itself comes at the end of the list. From the command line, the **UP** and **DOWN** arrow keys will quickly traverse this list up and down, while the **LEFT** and **RIGHT** arrow keys will move the cursor to allow the user to edit a given command.

Command history and sessions

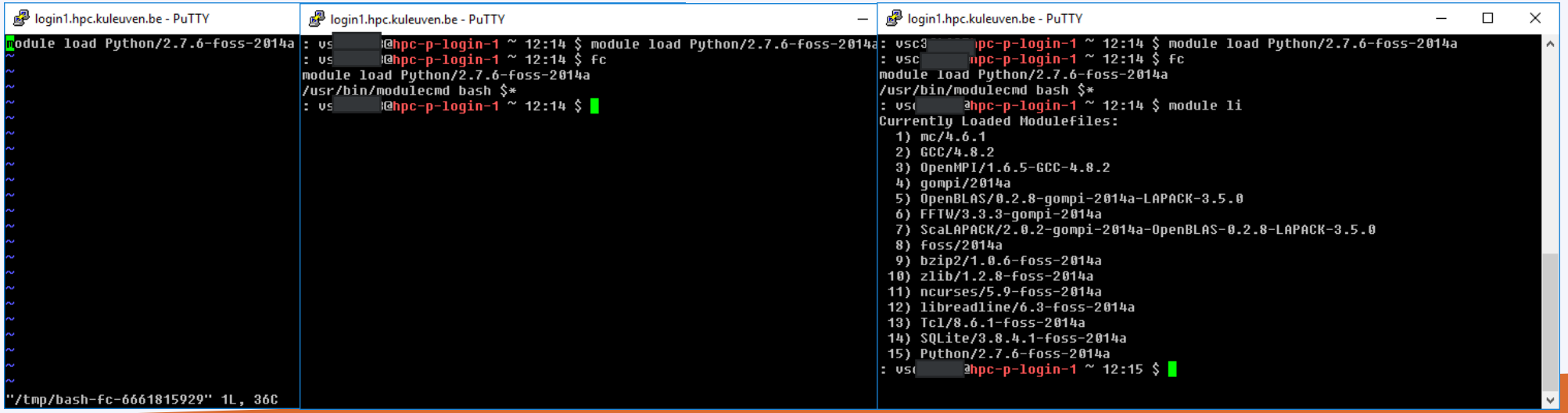
- Not only does the bash shell maintain a command history within a session, but the shell also preserves command histories between sessions. When the bash shell exits, it dumps the current command history into a file called **`.bash_history`** in a user's home directory. Upon startup, the shell initializes the command history from the contents of this file.
- What repercussions does this have for multiple interactive shells (owned by the same user) running at the same time? Because the history is only saved to disk as the shell exits, commands executed in one bash process are not available in the command history of a simultaneously running bash process. Also, the last shell to exit will overwrite the histories of any shells that exited previously.

Command history: Ctrl +R

- This key sequence mimics !cmd in spirit. Text typed after the CTRL+R key sequence is matched against previously typed commands, with the added advantage that matching command lines are viewed immediately as the text is typed.
- You also have the opportunity to edit the recalled line (using the LEFT and RIGHT arrow keys, or other command line editing keystrokes) before executing the command.

Command history: fc

- The fc command allows users to "fix" the previously entered command, by opening up the user's default editor (vi by default) with the previously entered command as text. Upon exiting the editor (presumably after somehow editing the command), the new text will be immediately executed. For those proficient in quickly exiting an editor, the command comes in handy.



Command history

`!$`

Repeats the last argument of the last command.

`:h`

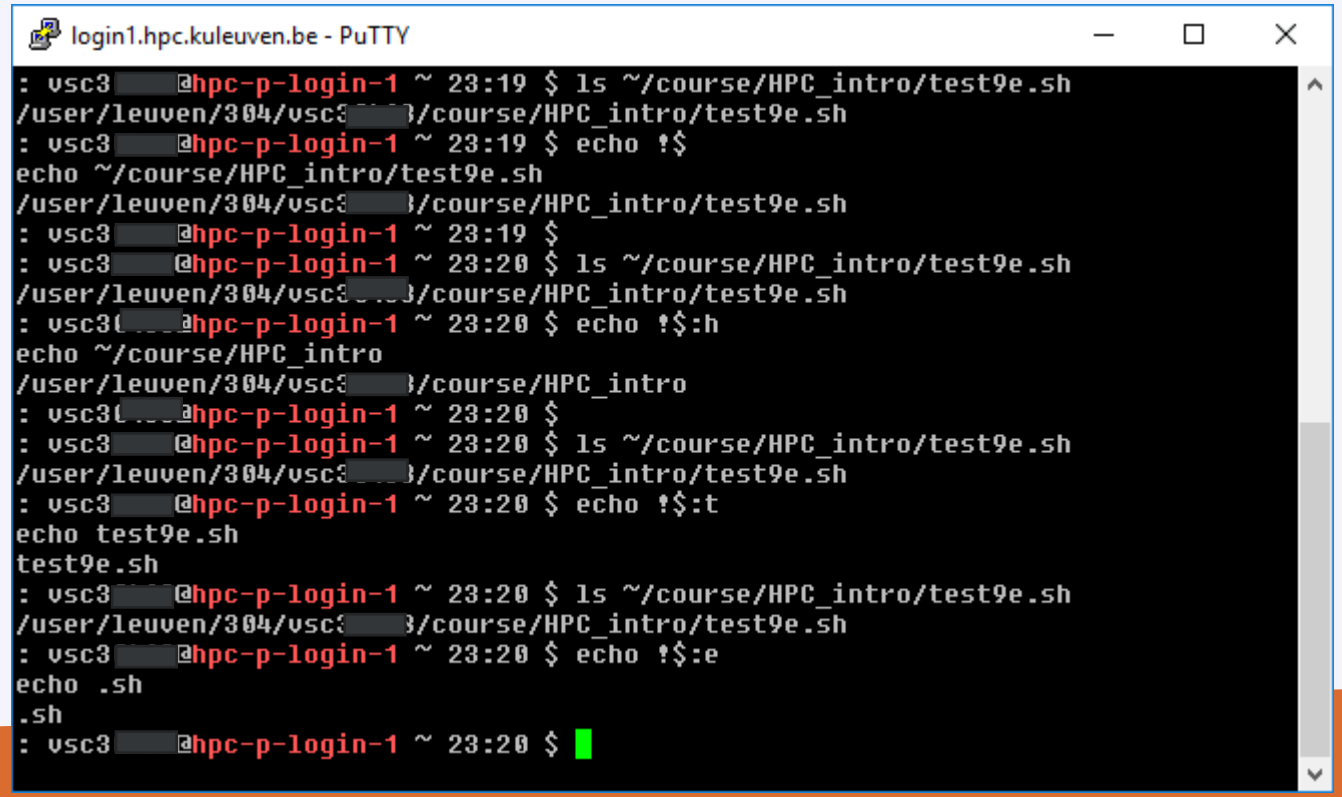
If you put it after a filename, it will change that filename to remove everything up to the folder.

`:t`

Leaves only filename

`:e`

Leaves only the extension



List Of Useful Bash Keyboard Shortcuts

- **ALT+B** – Move backward.
- **ALT+F** – Move forward.
- **ALT+T** – Swaps the last two words.
- **ALT+U** – Capitalize all characters in a word after the cursor.
- **ALT+L** – Uncaptalize all characters in a word after the cursor.
- **ALT+.** – Use the last word of the previous command.
- **!!** – Repeats the last command.
- **ESC+t** – Swaps the last two words.
- **CTRL+A** – Quickly move to the beginning of line.
- **CTRL+B** – To move backward one character.
- **CTRL+C** – Stop the currently running command. (Putty and MFA firewall!)

List Of Useful Bash Keyboard Shortcuts

- **CTRL+D** – Delete one character (backward).
- **CTRL+E** – Move to the end of line.
- **CTRL+F** – Move forward one character.
- **CTRL+H** – Delete the characters before the cursor, same as BASKSPACE.
- **CTRL+J** or **CTRL+M** – Same as ENTER/RETURN key.
- **CTRL+K** – Delete all characters after the cursor.
- **CTRL+L** – Clears the screen and redisplay the line.
- **CTRL+T** – Swaps the last two characters.
- **CTRL+U** – Delete all characters before the cursor (Kills backward from point to the beginning of line).
- **CTRL+W** – Delete the words before the cursor.
- **CTRL+Y** – Retrieves last item that you deleted or cut.
- **CTRL+Z** – Stops the current command.

Input and Output

- Programs and commands can contain an input and output. These are called ‘streams’. UNIX programming is oftentimes stream based.

- STDIN – ‘standard input,’ or input from the keyboard
- SDTOUT – ‘standard output,’ or output to the screen
- STDERR – ‘standard error,’ error output which is sent to the screen.

File Redirection

- Often we want to save output (stdout) from a program to a file. This can be done with the ‘redirection’ operator.
  - `myprogram > myfile`

- Similarly, we can **append** the output to a file instead of rewriting it with a double ‘>>’
  - `myprogram >> myfile`

Redirecting stderr

- Performing a normal redirection will not redirect sdterr. In Bash, this can be accomplished with ‘2>’
  - `command 2> file1`

- Or, one can merge stderr to stdout (most popular) with ‘2>&1’
  - `command > file 2>&1`

Input Redirection

- Input can also be given to a command from a file instead of typing it to the screen, using “<” operator
  - `mycommand < programinput`
  - Not all commands read standard input (ls, date, who, pwd, cd, ps, …)

Pipes

- Using a pipe operator ‘|’ commands can be linked together. The pipe will link the standard output from one command to the standard input of another.

- Very helpful for searching files
- e.g. when we want to list the files, but only the ones that contain test in their name:

```
ls -la | grep test
```

Hands-on 2

Shell

Environment variables

- Shells let the user define *variables*.
  They can be reused in shell commands. By convention, shell variables have lower case names.
- You can also define *environment variables*: variables that are also visible within scripts or executables called from the shell. By convention, environment variables have UPPER CASE names.
- `env`
  Lists all exported environment variables and their value.

Environment variables

- We can view the environment variables through `set` or `env` commands
- The `set` command will display all the global functions written by the user
- The `env` command displays only the variables and not the functions
- We can reassign values for the variables either temporarily or permanently
  - Temporary
    - Type `export varname=value` at the command prompt
  - Permanent
    - Type `export varname=value` in `.bashrc` in your `$VSC_HOME` directory

Environment variables

- Control the characteristics of the shell
  - View them with [set]`env`, or `$VARIABLE`
  - Set them with export
- Change up your prompt! `export PS1="myNEWprompt: "`
- Modify PATH:
  `export PATH=${PATH}:/home/student/program`
- But these have to be declared every time you use your shell.
- Solution: save them inside `$VSC_HOME/.bashrc`

Shell variables examples

Shell variables (bash)

```
projdir=$VSC_HOME/Downloads
ls -la $projdir; cd $projdir
```

Environment variables (bash)

```
cd $HOME

export DOC=$HOME/Documents
echo $DOC

/user/leuven/30X/vsc30XXX/Documents
```

(displays the information if parameter is set)

Standard environment variables

Used by lots of applications!

LD_LIBRARY_PATH
Shared library search path

DISPLAY
Screen id to display X (graphical) applications on.

EDITOR
Default editor (vi, emacs...)

HOME
Current user home directory

HOSTNAME
Name of the local machine

MANPATH
Manual page search path

PATH
Command search path

PRINTER
Default printer name

SHELL
Current shell name

TERM
Current terminal type

USER
Current user name

PATH environment variables

e.g. `which` or `whereis` searches in that location

PATH

Specifies the shell search order for commands

```
/home/acox/bin:/usr/local/bin:/usr/kerberos/bin:/usr/bin:/bin
:/usr/X11R6/bin:/bin:/usr/bin
```

e.g. `whereis` searches in that location

LD_LIBRARY_PATH

Specifies the shared library (binary code libraries shared by applications, like the C library) search order for ld

```
/usr/local/lib:/usr/lib:/lib:/usr/X11R6/lib
```

MANPATH

Specifies the search order for manual pages

e.g. `whereis` searches in that location

```
/usr/local/man:/usr/share/man
```

Environment variables

- Paths (`$PATH, $LD_LIBRARY_PATH, $MAN_PATH, $CPATH, …`) are modified when modules are loaded
- After "`module load`" env will display a new value:

Example

```
$> echo $PATH
/user/leuven/30X/vsc30XXX/.tmux/bin:/apps/leuven/bin:/usr/local/bin:/usr/lpp/mmfs/bin:.:/usr/bin:/usr/sbin:/opt/moab/bin:/opt/mam/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/opt/ibutils/bin
$> module load matlab/R2019a
$> echo $PATH
/apps/leuven/skylake/2018a/software/MATLAB/R2019/bin:/user/leuven/30X/vsc30XXX/.tmux/bin:/apps/leuven/bin:/usr/local/bin:/usr/lpp/mmfs/bin:.:/usr/bin:/usr/sbin:/opt/moab/bin:/opt/mam/bin:/usr/local/sbin
```

Do-It-Yourself

Example

```
$> module purge          # start from clean slate
$> env | grep PATH       # default PATH(s)
$> which python
$> module load Python/3.6.4-intel-2018a
$> env | grep PATH       # PATH(s) are modified
$> which python
```

Questions

1. Which new directories are now added to `$PATH`?
2. Which python was originally used (after login)?
3. Which python is used after `module load`?

Aliasing

- Alias – Alternate name for a command(s)
- You can be inventive with it, but be careful
- If you log out from bash, your aliases will be purged!
- To define an alias permanently, put it in your `.bashrc`
- E.g.
  Image, your jobs always store the results in your scratch folder.
  So, you want an easy way to copy them to your data folder:

```
alias backup="cp -r $VSC_SCRATCH/results $VSC_DATA/backup"
```

- To disable an alias, do:

```
unalias backup
```

Alias and Unalias

- alias newname=oldname
  - eg. *alias copy=cp*
- Then we can use copy in the same way we use cp command
  - eg. *copy file1 file2* //copies content of file1 to file2
- To remove alias use unalias command
  - *unalias copy*
- After this we cannot use copy to perform copying function

Alias

Shells let you define command *aliases*: shortcuts for commands you use very frequently.

Examples

```
alias la='ls -la'
```
Useful to always run commands with default arguments.

```
alias rmi='rm -i'
```
Useful to make rm always ask for confirmation.

```
alias data='cd /data/leuven/304/vsc30XXX'
```
Useful to replace very long and frequent commands.

```
alias schck='. /home/mag/env/chck.sh'
```
Useful to set an environment in a quick way
(. is a shell command to execute the content of a shell script).

which command

Before you run a command, `which` tells you where it is found

```
which ls
alias ls='ls --color=auto'
        /usr/bin/ls

which alias
/usr/bin/alias

which help
/usr/bin/which: no help in
(/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin
:/sbin:/home/mag/.local/bin:/home/mag/bin)
```

~/.bashrc file

`~/.bashrc`
Shell script read each time a bash shell is started (login, or when the job starts on a compute node)

You can use this file to define

- Your default environment variables (PATH, EDITOR...).
- Your aliases.
- Your prompt (see the bash manual for details).
- A greeting message.

Do NOT put "`module load`" in your .bashrc. It creates conflicts

bash configuration Files

- bash has two different login files.
  - **.bashrc** gets read when you open a local shell on a machine
  - **.bash_profile** only gets read if and only if you login from a remote machine. Note that **.bash_profile** itself reads in your **.bashrc** file as well.

- If you want aliases to be executed regardless, then you should put them in the **.bashrc** file.

- On the cluster please edit only .bashrc file – in case of problem we can always allow you access thanks to correct .bash_profile

~/.bash_profile

- This is how your ~/.bash_profile looks like
- Tip: never touch it

~/.bash_profile

```
# File:  .bash_profile
# Get the aliases and functions
# Get whatever is in your
# .bashrc config file
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

```
# File: .bashrc #
# Description: A default .bashrc
###Source global defs ###
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

###set the prompt ###
# uncomment out only one
# this is hostname and time
PS1="\h-(\@): "
# this is hostname and
# history number
#PS1="\h-(\!)# "
# this is hostname and
# working directory
#PS1="\h-(\w)# "
# this is hostname and
# shortened working
# directory
#PS1="\h-(\W)# "

### path manipulation ###
# add ~/bin to the path,
# cwd as well
PATH="$PATH:$HOME/bin:./“

### env variables ###
# make sure that you
# change this to your
# username
MAIL="/afs/umbc.edu/users/u/s/username/M
    ail/inbox“
export PATH
unset USERNAME

### User-specific aliases
### and functions ###
alias rm="rm -i"
```

Flavours of Unix Shells

- Two main flavours of Unix Shells
  - Bourne (or Standard Shell): `sh, ksh, bash, zsh`
    - Fast
    - $ for command prompt
  - C shell : `csh, tcsh`
    - easier for scripting
    - %, > for command prompt
- To check shell:
  - `% echo $SHELL` (shell is a pre-defined variable -default)
  - `% echo $shell` (shell that is running)
- To switch shell:
  - `% exec <shellname>  #e.g., % exec bash`
  - `$ shellname`

What shell am I running?

- Use the `echo` command
  E.g., to check your `SHELL` environment variable:

```
mag@localhost ~]$ tcsh
[mag@localhost ~]$ echo $SHELL
/bin/bash
[mag@localhost ~]$ echo $shell
/bin/tcsh
[mag@localhost ~]$
```

- Issue a `ps` command to see all the processes in your current login session

Customization of a Session

- Each shell supports some customization.
  - User prompt
  - Where to find mail
  - Shortcuts (alias)
- The customization takes place in *startup* files
  - Startup files are read by the shell when it starts up
  - The Startup files can differ for different shell

Startup files

- **sh,ksh:**

  /etc/profile (out-of-the-box login shell settings)
  /etc/profile.local (addtnl. local system settings)
  ~/.profile (addtnl. user customized settings)
  ~/.kcshrc (non-login shell user customization)

- **bash:**
  /etc/profile (out-of-the-box login shell settings)
  /etc/bash.bashrc (out-of-box non-login settings)
  /etc/bash.bashrc.local (global non-login settings)

  ~/.bash_profile (login shell user customization)

  ~/.bashrc (non-login shell user customization)

  ~/.bash_logout (user exits from interactive login shell)

- **csh/tcsh:**
  /etc/login (out-of-the-box login shell settings)
  /etc/csh.login (non-login shell customizations)
  /etc/csh.login.local (global non-login settings)

  ~/.login: (login shell user customizations)

  ~/.cshrc: (non-login shell user customizations)

  ~/.cshrc.logout: (non-login shells at logout)

  ~/.logout: (read by login shells at logout)

Customization of a Session - prompt

- LOGNAME: contains the user name
- HOSTNAME: contains the computer name.
- RANDOM: random number generator
- SECONDS: seconds from the beginning of the execution

- PS1: sequence of characters shown before the prompt

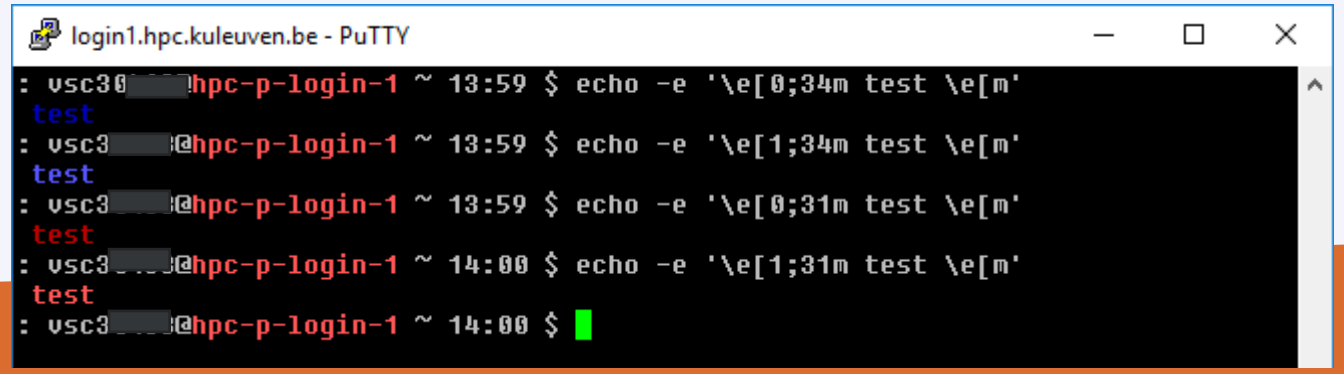
\t	hour
\d	date
\w	current directory
\W	last part of the current directory
\u	user name
\$	prompt character

Customization of a Session

- To add colors to the shell prompt check the following command syntax:
  $ `echo '\e[x;ym test \e[m'`
  Where,
  - \e[: Start color scheme.
  - x;y : Color pair to use (x;y)
  - test: to be printed
  - \e[m : Stop color scheme.
- More info about colors e.g. at http://misc.flogisoft.com/bash/tip_colors_and_formatting

Color	Code
Black	0;30
Blue	0;34
Green	0;32
Cyan	0;36
Red	0;31
Purple	0;35
Brown	0;33

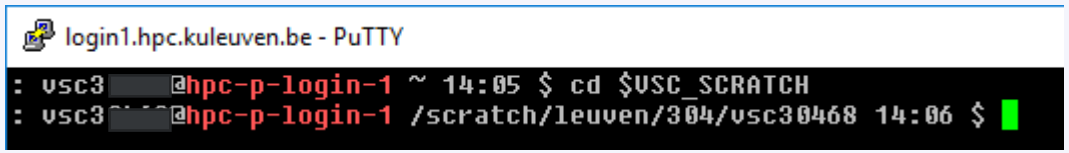
Note: You need to replace digit 0 with 1 to get light color version.



Customization of a Session

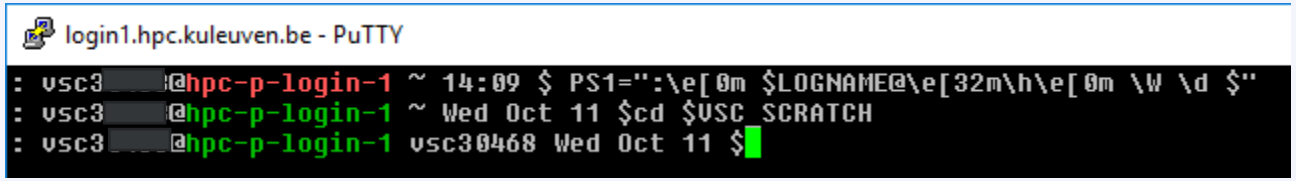
- Standard PS1:

  \u@\[\e[1;31m\]\h\[\e[0m\] \w `date +%H:%M`



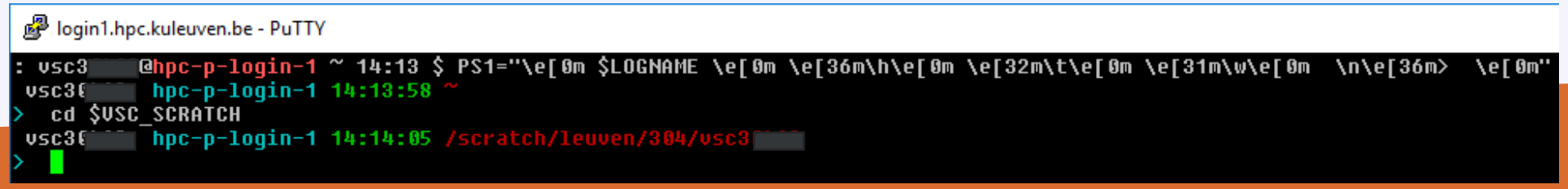
- Change PS1, e.g.

  PS1=":\e[0m $LOGNAME@\e[32m\h\e[0m \W \d $“



- Another example:

  PS1="\e[0m $LOGNAME \e[0m \e[36m\h\e[0m \e[32m\t\e[0m \e[31m\w\e[0m \n\e[36m> \e[0m"



VLAAMS SUPERCOMPUTER CENTRUM

Environment Variables

- Use the env command to see all environment variables
- set/export to see all shell variables
- Set or change environment variables from the command-line:
  new values last only for current login session.

sh/bash/ksh
```
set: NEW_VARIABLE=newvalue
append: OLD_VARIABLE=$OLD_VARIABLEnewvalue
prepend: OLD_VARIABLE=newvalue$OLD_VARIABLE
add: OLD_VARIABLE=${OLD_VARIABLE}:newvalue
export OLD_VARIABLE
```

csh/tcsh
```
set:  set NEW_VAR=newvalue
append:  set OLD_VAR=($OLD_VAR newvalue)
prepend:  set OLD_VAR=(newvalue $OLD_VAR)

set: setenv OLD_VAR newvalue
append: setenv OLD_VAR ${OLD_VAR}newvalue
prepend: setenv OLD_VAR newvalue${OLD_VAR}
```

The order decides where system checks for command first (important if you have your own version and there is another vesion on the cluster)

Introduction to bash

- The bash shell is one of the many shells that are available to you on the VSC nodes.
- Almost any installation of Linux defaults to the bash shell.
- bash is one of the many GNU.org (http://www.gnu.org) projects.
- bash manuals:
  - A comprehensive online manual is provided at http://www.gnu.org/software/bash/manual/bashref.html
  - Aliases - http://www.gnu.org/software/bash/manual/bashref.html#Aliases
  - Controlling the Prompt - http://www.gnu.org/software/bash/manual/bashref.html#Controlling-the-Prompt

Universal customization

- Universal .bashrc - written to run on all (relevant) clusters:

```
case ${VSC_INSTITUTE_CLUSTER} in
    wice)
        ulimit -c 500000
        export LD_LIBRARY_PATH="${HOME}/lib:${LD_LIBRARY_PATH}"
        export PATH="${HOME}/bin:${HOME}/sbin:${PATH}"
        export EDITOR="/usr/bin/vim"
        export PS1=': \u@\[\e[1;31m\]\h\[\e[0m\] \w `date +%H:%M` $'
        source ${HOME}/.autoenv/activate.sh
        ;;
    genius)
        export EDITOR="/usr/bin/vim"
        alias vim="vim -u .vimrc-simple“
        export PS1=': \u@\[\e[1;34m\]\h\[\e[0m\] \w `date +%H:%M` $ '
        ;;
esac
```

Hands-on 3

Shell scripts

Using the Shell

Command Line Interpreter or CLI

To understand scripts let's practice a bit with the CLI.

At the shell prompt try:

```
# cd; echo “Hello, World” > test.txt;
  cp test.txt test.txt.bak; vi test.txt
```

The above is all on one line.

What happened?

Using the Shell

In a file

Create a new file and place the some of commands in it:

```
$ cd $VSC_DATA
$ vim newscript.sh
   echo “Hello world” > hello.txt
   cp hello.txt hello.txt.bak
   cat hello.txt hello.txt.bak > new.txt
   cat new.txt
```

Using the Shell

In a file

Now we can execute those commands in the order in which they appear in the file by doing this.
There are 3 ways to execute a script:

```
$> bash newscript
$> sh newscript
$> . newscript
```

In a sub-shell

In the current shell

Using the Shell

As a shell script

Now we can take the last step and start to create self-contained scripts that run on their own.

We'll need to do two things:

1. Specify the CLI to use, and
2. Make the script executable

```
chmod +x <script_name>
```

The “Shebang”

To specify that a file is to become a shell script you specify the interpreter like this at the very start of the file:

```
#!/bin/bash
```

- Shebang is a comment line, so practically not executed
- Shebang starts with #!
- Then, shebang gives the path to an executable that will interpret the script
- E.g. other typical shebangs could be:
  - `#!/bin/python`
  - `#!/bin/perl`
  - `#!/bin/csh`

When Not to Use Scripts?

- Resource-intensive tasks, especially where speed is a factor (sorting, hashing, etc.)
- Procedures involving heavy-duty math operations, especially floating point arithmetic arbitrary precision calculations, or complex numbers
- Cross-platform portability required
- Complex applications, where structured programming is a necessity (need type-checking of variables, function prototypes, etc.)
- Project consists of subcomponents with interlocking dependencies
- Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion)
- Need native support for multi-dimensional arrays or data structures, such as linked lists or trees
- Need to generate or manipulate graphics or GUIs
- Need direct access to system hardware or port or socket I/O

What's Next?

Now let's create a very simple shell script. This will simply echo back what you enter on the command line:

```
#!/bin/bash
echo $1
```

Enter this in a file new.sh, then do:

```
# chmod 755 new.sh
```

To run the script do:

```
# ./new.sh text
```

Shell scripts

- Shell scripts are “programs” that are completely uncompiled, but read and executed by the shell line by line.
- Typically end in .sh
- Must be executable with `chmod`
- Start with a “shebang” – tells the shell what to use to interpret it. e.g.,
  - #! /bin/bash for a bash script.

Bash vs. C

Bash

```
#! /bin/bash


number=3
name="bob"
echo "$name is your chosen name, $number your chosen number."

let inc=number+1
if [ "$inc" == "4" ]
    then echo "Addition works like a charm."
fi
```

C

```
#include <stdio.h>
#include <cs50.h>

int number = 3;
string name = "bob";
printf("%s is your chosen name, %d your chosen number.\n", number, name);

int inc = number++;
if ( inc == 4 ) {
        printf("Addition works like a charm.\n");
}
```

Bash vs. C

	Bash	C
Language	interpreted	compiled
Variable types	Everything is string	Multiple types; declaration needed
Variable access	Via $	By name
At runtime	Uses other Linux programs to work	Uses subroutines & functions from libraries to work
Spacing	Matters a lot	Matters much less
Line endings	None	;

Resources

The on-line *Advanced Bash Scripting Guide*, available at:

http://www.tldp.org/LDP/abs/html/

Languages

Interpreted

Bash, Perl, Python

Compiled

C, C++, Fortran

Tools

- sed: Stream EDitor
- awk: Pattern scanning & processing
- bc: Arbitrary precision calculator
- tr: Translate or delete characters
- grep: Print lines matching a pattern

Example: Slurm Job Script

Slurm scripts do not need to be executable – they are submitted to the queue and executed some other way.

```
#!/bin/bash -l
#SBATCH --cluster=wice
#SBATCH --job-name="hello_world"
#SBATCH -N 1
#SBATCH --ntasks-per-node=72
#SBATCH -t 1:00:00
#SBATCH --mem-per-cpu=3400M
#SBATCH -A lp_hpcinfo

module load intel/2021a
which icc

cd $SLURM_SUBMIT_DIR
cp /apps/leuven/training/HPC-intro/cpujob.pbs $VSC_SCRATCH
touch output.log
echo I am done
```

Shebang

Resource List

Module load(s)

Move data

Execure commands

Hands-on 4

Application development

Installing applications – home linux

- Automatic way: **`yum`**
- Yum is an interactive, rpm based, package manager.
- To check the package: `yum search phrase`
- To install `yum install package-name` (root only!)
- To check installed packages: `yum list installed`

- Possibility to install downloaded rpm package: **`rpm -i package`**

Manual installations

- Some applications provide binary files – ready to use after unpacking.
- Check the system – different flavours and architectures!

Compatibility Statement

- **Definition of the OS version**
  - Kernel version (i.e. kernel 4.18.0-477.10.1.el8_8.x86_64)
  - glibc version (i.e. glibc 2.28)
- **Determining the OS Version**
  - Kernel version (`$ uname -r ->` 4.18.0-477.10.1.el8_8.x86_64)
  - System distribution (`$ uname -a`)
  - glibc version (`$ ldd --version -> 2.28`)
  - `$ rpm -qa | grep libc` -> glibc-2.28)
- **Determining the OS Flavour/release**
  - `$ cat /etc/*-release`

Manual installations

- Some simple programs will only require compiling (hello world example from HPC intro:

```
cp -r /apps/leuven/training/HPC_intro $VSC_HOME; cd HPC_intro
module load foss/2018a; mpicc helloworldmpi.c -o hello.exe
mpirun ./hello.exe
```

- Lots of specific (less frequently used) applications will require the whole configuration (configure-make process) and installation process. Usually manual is provided with instructions (README.txt)

Compiling simple applications

- The compiler used for all Linux systems is GCC
  http://gcc.gnu.org
- To compile a single-file application, developed in C :
  `gcc -o test test.c`
  Will generate a `test` binary, from the `test.c` source file
- For C++ :
  `g++ -o test test.cc`
- The `-Wall` option enables more warnings
  To compile sources files to object files and link the application :

```
gcc -c test1.c
gcc -c test2.c
gcc -o test test1.o test2.o
```

gcc automatically calls the linker ld

make

- The compilation process can be automated using GNU `make` tool: http://www.gnu.org/software/make/manual/
- make reads a file called `Makefile` from the current directory, and executes the rules described in this file
- Every rule has a target name, a colon, and a list of dependencies, and the list of commands to generate the target from the dependencies
- When simply running make, the default target that is generated is “all”. A target is only re-generated if dependencies have changed
- Every year, we offer make training (half a day).
  Check: https://www.vscentrum.be/training

make

- Makefiles are nice, but they don't easily allow easy adaptation to the different build environment and different build options
- More elaborated build systems have been developed
  Autotools (automake, autoconf), based on Makefiles and shell scripts. Even though they are old and a little bit difficult to understand, they are the most popular build system for free software packages.
  CMake, a newer, cleaner build system
  Sconcs and Waf, other build systems based on Python

- The typical steps to compile a autotools based package are:

```
./configure (--prefix=/location-of-the-installation)
make
sudo make install (when admin rights necessary)
 or
 make install
```

Python and R packages installation

Installing your own packages using conda

Installing Miniconda

- Download the Bash script that will install it from conda.io using, e.g., wget:
  ```
  $ wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
  ```

- Once downloaded, run the installation script:
  ```
  $ bash Miniconda3-latest-Linux-x86_64.sh -b -p $VSC_DATA/miniconda3
  ```

- Optionally, you can add the path to the Miniconda installation to the PATH environment variable in your .bashrc file. This is convenient, but may lead to conflicts when working with the module system or OS, so make sure that you know what you are doing in either case.
  The line to add to your .bashrc file would be:
  ```
  export PATH="${VSC_DATA}/miniconda3/bin:${PATH}
  ```
  May create conflicts for NX login!

Installing Python packages using conda

Creating an environment

- First, ensure that the Miniconda installation is in your PATH environment variable. The following command should return the full path to the conda command:

```
$ which conda
```

- If the result is blank, or reports that conda can not be found, modify the `PATH` environment variable appropriately by adding miniconda's bin directory to PATH.
- Creating a new conda environment is straightforward:

```
$ conda create -n science numpy scipy matplotlib
```

- This command creates a new conda environment called science, and installs a number of Python packages.
- This will default to the latest Python 3 version, if you need a specific version, e.g., Python 2.7.x, this can be specified as follows:

```
$ conda create -n science python=2.7 numpy scipy matplotlib
```

Installing Python packages using conda

Working with the environment

- To work with an environment, you have to activate it. This is done with, e.g.,
  ```
  $ source activate science
  ```
  Here, science is the name of the environment you want to work in.

Install an additional package

- To install an additional package, e.g., `pandas`, first ensure that the environment you want to work in is activated.
  ```
  $ source activate science
  ```

- Next, install the package:
  ```
  $ conda install pandas
  ```
  Note that conda will take care of all dependencies, including non-Python libraries. This ensures that you work in a consistent environment.

Installing Python packages - alternatives

Checking for installed packages

- Pip utility will list all packages that are installed for the Python distribution you are using, including those installed by you, i.e., those in your `PYTHONPATH` environment variable.

- Load the module for the Python version you wish to use, e.g.,:
  ```
  $ module load Python/2.7.14-foss-2018a
  ```

- Run pip:
  ```
  $ pip freeze
  ```

- Note that some packages, e.g., mpi4py, h5py, pytables, ..., are available through the module system, and have to be loaded separately. These packages will not be listed by pip unless you loaded the corresponding module.

- If you have any packages installed in **`.local`** directory, it will always take priority on whatever the Python version used (conda, module, system). That can lead to strange problems, so please avoid using that location.

Installing Python packages - pip

1. Load the appropriate Python module, i.e., the one you want the python package to be available for:
   ```
   $ module load Python/3.9.5-GCCcore-10.3.0
   ```
2. Create a directory to hold the packages you install, the last three directory names are mandatory:
   ```
   $ mkdir -p "${VSC_DATA}/python_lib/lib/python3.9/site-packages/"
   ```
3. Add that directory to the PYTHONPATH environment variable for the current shell to do the installation:
   ```
   $ export PYTHONPATH="${VSC_DATA}/python_lib/lib/python3.9/site-packages/:${PYTHONPATH}"
   ```
4. Add the following to your .bashrc so that Python knows where to look next time you use it:
   ```
   export PYTHONPATH="${VSC_DATA}/python_lib/lib/python3.9/site-packages/:${PYTHONPATH}"
   ```
5. Install the package, using the prefix install option to specify the install path (this would install the sphinx package):
   ```
   $ pip install --user sphinx
   ```
   or
   ```
   $ pip install --install-option="--prefix=${VSC_DATA}/python_lib" sphinx
   ```

Installing Python packages – easy_install

1. Load the appropriate Python module, i.e., the one you want the python package to be available for:
   ```
   $ module load Python/3.9.5-GCCcore-10.3.0
   ```
2. Create a directory to hold the packages you install, the last three directory names are mandatory:
   ```
   $ mkdir -p "${VSC_DATA}/python_lib/lib/python3.9/site-packages/"
   ```
3. Add that directory to the PYTHONPATH environment variable for the current shell to do the installation:
   ```
   $ export PYTHONPATH="${VSC_DATA}/python_lib/lib/python3.9/site-packages/:${PYTHONPATH}"
   ```
4. Add the following to your .bashrc so that Python knows where to look next time you use it:
   ```
   export PYTHONPATH="${VSC_DATA}/python_lib/lib/python3.9/site-packages/:${PYTHONPATH}"
   ```
5. Install the package, using the prefix option to specify the install path (this would install the sphinx package):
   ```
   $ easy_install --prefix="${VSC_DATA}/python_lib" sphinx
   ```

Installing R packages using conda

Creating an environment

- First, ensure that the Miniconda installation is in your PATH environment variable. The following command should return the full path to the conda command:
  ```
  $ which conda
  ```

- If the result is blank, or reports that conda can not be found, modify the `PATH` environment variable appropriately by adding miniconda's bin directory to PATH.

- Creating a new conda environment is straightforward:
  ```
  $ conda create -n science -c r r-essentials r-rodbc
  ```
  This command creates a new conda environment called science, and installs essentials and required packages.

Each package separately in a new env – developpers do not care about versions!

Installing R packages using conda

Working with the environment

- To work with an environment, you have to activate it. This is done with, e.g.,
  ```
  $ source activate science
  ```
  Here, science is the name of the environment you want to work in.

Install an additional package

- To install an additional package, e.g., `r-ggplot2`, first ensure that the environment you want to work in is activated.
  ```
  $ source activate science
  ```

- Next, install the package:
  ```
  $ conda install -c r r-ggplot2
  ```
  Note that conda will take care of all independencies. This ensures that you work in a consistent environment.

Installing R packages using conda

Updating/removing

- Using conda, it is easy to keep your packages up-to-date. Updating a single package (and its dependencies) can be done using:
  ```
  $ conda update r-rodbc
  ```
- Updating all packages in the environement is trivial:
  ```
  $ conda update --all
  ```
- Removing an installed package:
  ```
  $ conda remove r-mass
  ```

Deactivating an environment

- To deactivate a conda environment, i.e., return the shell to its original state, use the following command
  ```
  $ source deactivate
  ```

Installing other R packages using conda

- Installing CRAN package:
```
> install.packages('readr', repos='http://cran.us.r-project.org')
```

- Alternative approach:
```
$ conda skeleton cran readr
$ conda build r-readr
$ conda install --use-local r-readr
```
Doing that for the first time you need to install conda-build before:
```
$ conda install conda-build
```

- Some packages not available in r-essentials are still available on conda channels, in that case, it's simple:
```
$ conda config --add channels r; conda install r-readxl
```

Installing R packages – alternatives

1. Load the appropriate R module, i.e., the one you want the package to be available for:
   ```
   $ module load R/3.6.0-foss-2018a-bare
   ```

2. start R and install the package from there:
   ```
   > install.packages("DEoptim")
   ```

3. Alternatively you can download the desired package:
   ```
   $ wget cran.r-project.org/src/contrib/Archive/DEoptim/DEoptim_2.0-0.tar.gz
   ```

   And Install the package from the command line:
   ```
   $ R CMD INSTALL DEoptim_2.2-3.tar.gz  -l /$VSC_DATA/R/
   ```

4. These packages might depend on the specific R version, so you may need to reinstall them for the other version.

Hands-on 5

Usefull link(s) – tips and tricks:

- http://gjbex.github.io/training-material/LinuxTools/
  - bash
  - cdargs
  - grep
  - network
  - screen
  - ssh
  - tmux
  - top
  - vim

Questions

Helpdesk:
hpcinfo@kuleuven.be or https://admin.kuleuven.be/icts/HPCinfo_form/HPC-info-formulier

VSC web site:
http://www.vscentrum.be/

VSC documentation: https://docs.vscentrum.be

VSC agenda: training sessions, events

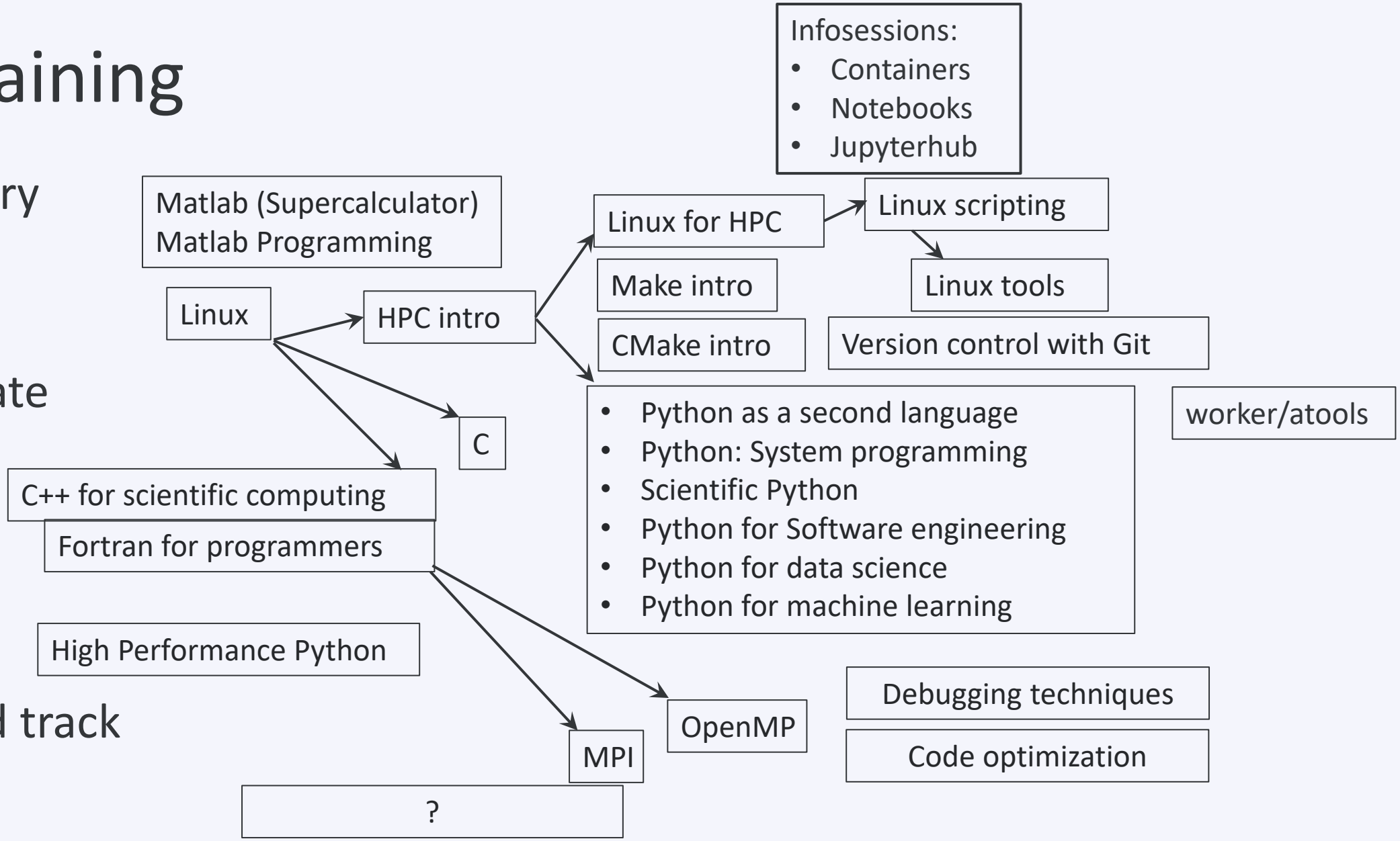
Systems status page:
http://status.vscentrum.be

Stay Connected to VSC

LinkedIn®

VSC training

- Introductory
- Intermediate
- Advanced
- Specialized track



PRACE MOOC Defensive programming and debugging and Fortran for Programmers (announced by e-mail)