Terminal, environment variables, linking, Makefile, virtual environments

Академия Больших Данных

Sergey Matveev, Sergey Rykovanov

Plan for today

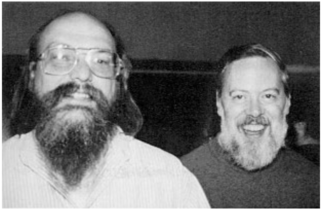
- Brief intro into brief introduction to Linux
  - Operating system and its main functions
  - Processes in OS. Types, process lifecycle, context
  - Main concepts of filesystem organization: descriptor, permissions, processing operations
- Bash- and SSH-terminal sessions:
  - ls ,cd, echo, mkdir, rm, pwd, touch, vi(m)
  - organizing for loop, head, tail, sort, and our especial friends "|, >, <, &"
  - Establish remote connection with external Linux machine and find out that you got into another linux terminal.
  - Deal with it and study where you are:
    hostname, ifconfig, df , du, exit
    Copy data to server and from it. Copy the directory zip-unzip data.
- Set of simple exercises on implementation of bash-scripts.

- Environment variables
- More useful information
  - Usergroups and permissions
  - Soft and hard links
  - wget, tar and git: gentelman's set
- Compiling
  - Compiling multiple sources with gcc, static/dynamic linking
  - Reminder on VIM
  - Makefile basics and optimization flags
- Virtual environments: concepts

Expectations during mini-practice

- You will ask questions
- You will ask right questions
- You will become able to ask Google right questions

Brief intro into brief introduction to Linux



”Revolution OS” documentary film

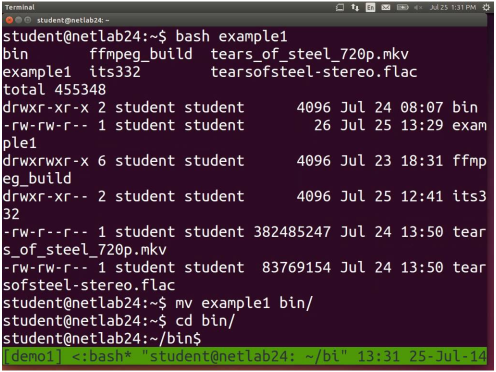
The Operating System and the User

The OS provides the User interaction with the "Programs" (processes) by means of

- Graphical Interface. Native for Windows, possible for UNIX/Linux OS: KDE, Gnome, XFce, etc.
- Command Line interface. Native for UNIX/Linux OS, Windows simulates it (power shell).
- Application Programming Interface (API) provided by OS Kernel (Interprocesses interactions, pipes, sockets), C/C++ compiler is needed. For advanced users and developers.

It is completely enough the command line interface for administration purposes.

The Command Line (shell)



```
student@netlab24:~$ bash example1
bin        ffmpeg_build  tears_of_steel_720p.mkv
example1   its332        tearsofsteel-stereo.flac
total 455348
drwxr-xr-x 2 student student      4096 Jul 24 08:07 bin
-rw-rw-r-- 1 student student        26 Jul 25 13:29 exam
ple1
drwxrwxr-x 6 student student      4096 Jul 23 18:31 ffmp
eg_build
drwxr-xr-- 2 student student      4096 Jul 25 12:41 its3
32
-rw-r--r-- 1 student student 382485247 Jul 24 13:50 tear
s_of_steel_720p.mkv
-rw-rw-r-- 1 student student  83769154 Jul 24 13:50 tear
sofsteel-stereo.flac
student@netlab24:~$ mv example1 bin/
student@netlab24:~$ cd bin/
student@netlab24:~/bin$
[demo1] <:bash* "student@netlab24: ~/bi" 13:31 25-Jul-14
```

We can easily manage and ”interact with a computer” by around two dozens of commands like **ls**, **cd**, **mkdir**, **pwd**, **cat**, **date**, **whoami**, **su**...

But what is beyond the ”user-friendly commands”?

– The main functions of the Operating system –

Operating System (OS) and its main functions

Operating system is a set of programs controlling

- Existence
- Usage
- Distribution

of the computing system resources.

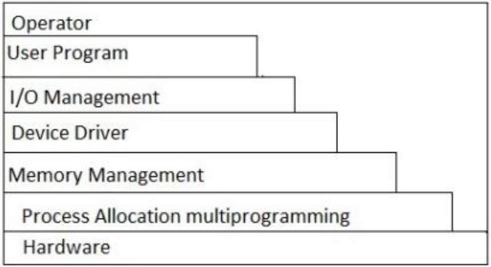


fig:- layered Architecture

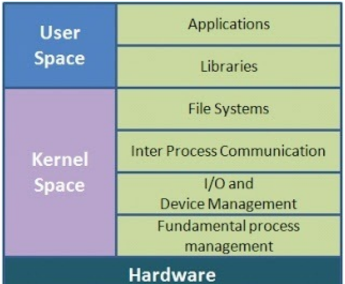
Each operating system works with basic its units: processes

OS kernel

Kernel is a part of OS working **permanently** in RAM, in **privileged regime** processing control of

- Process activities scheduling
- Interaction of processes
- Memory distribution
- Hardware and software drivers
- I/O and filesystem management

All of this functions are provided by **one** kernel program in case of **monolithic** kernel – Unix/Linux are organized exactly in this way.


User Space
Applications
Libraries
Kernel Space
File Systems
Inter Process Communication
I/O and Device Management
Fundamental process management
Hardware

Process

Process is a set of commands and data processes by computer having permissions to use computing resources. OS manages implementation of scheduling and interaction between many processes inside single computer. Each process goes though several stages during its **lifecycle**:

1. Generation of process
2. Execution at CPU
3. Pending for allocation of resources and interations with other processes
4. Completion of process and release of allocated resources

Process context

Context of process is data characterizing its current state. Consists from

- User context (commands, code segment and data)
- System context (id, permissions, opened files and i/o sstreams etc)
- Hardware context (e.g. registers and setup at stopping point)

At any one time during execution a **process runs in the context of itself** and the **kernel runs in the context of the currently** running process.

Process and Thread are different.

Filesystem basic concepts

Filesystem is a part of OS consisting from

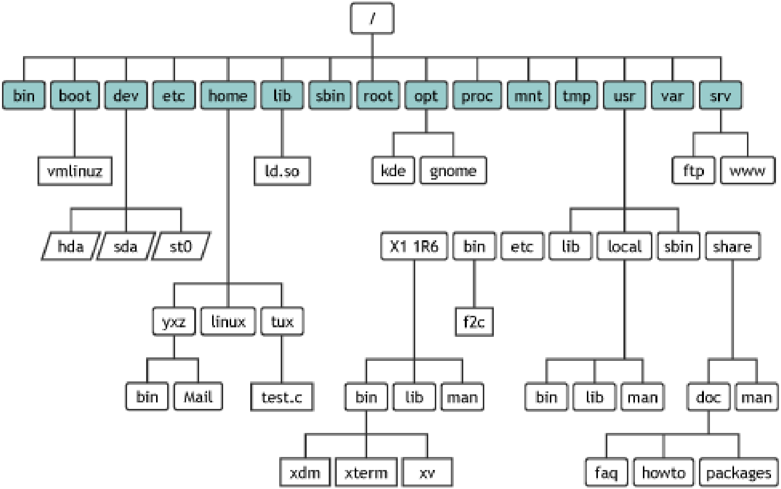
- Organized datasets
- Stored at external devices
- and **software** tools guaranteeing name-addressed access to these datasets and their protection

Thus, **file** is just series of bytes associated with **attributes**

File attributes

- Name – just set of symbols
- Acess perimssions
- Personification (stores info about file's "owner" and "usergroup")
- Type of file (there are files for devices and regular files; files for execution and data)
- Block size
- Size of file
- Read/Write pointer
- Etc (e.g. last modification time)

Linux filesystem



Plan for today

1. Brief intro into brief introduction to Linux
   - Operating system and its main functions
   - Processes in OS. Types, process lifecycle, context
   - Main concepts of filesystem organization: descriptor, permissions, processing operations
2. **Bash- and SSH-terminal sessions:**
   - ls ,cd, echo, mkdir, rm, pwd, touch, vi(m)
   - organizing for loop, head, tail, sort, and our especial friends "$|, >, <, \&$"
   - Establish remote connection with external Linux machine and find out that you got into another linux terminal.
   - Deal with it and study where you are:
     hostname, ifconfig, df , du, exit
     Copy data to server and from it. Copy the directory zip-unzip data.
3. Set of simple exercises on implementation of bash-scripts.

Bash

Bourne again shell is a Unix shell written by Brian Fox for the GNU Project as a free software. It has been distributed widely as the default login shell for most Linux distributions.

In fact, it is a programming language with interpreter supporting work either in interative or scripting mode.

Let's start with basic commands in interactive mode

command	result
ls	**l**i**s**t of files in the current directory
cd	**c**hange **d**irectory
pwd	**p**rint **w**orking **d**irectory
echo	print the parsed string
mkdir	**m**a**k**e (create) **dir**ecrory
rm	**re**m**o**ve the file
head	print first lines from the file
tail	print last lines from the file
touch	create empty file
sort	prints the lines of its input in sorted order
chmod	**ch**ange **mod**e

...and our especial friends

- Redirect standard output stream:

  ```
  ls > list_of_directory.log
  ```

  Symbol “>” redirects output into the file “list_of_directory.log”. By the way: “>>” redirects output into the **end** of file “list_of_directory.log”.

- Organize a pipe:

  ```
  ls | head -n 3
  ```

  Symbol “|” will **redirect standard output** stream of the first command and the second command will use it as **standard input** stream.

- Leave process at the background:

  ```
  ls &
  ```

  Symbol “&” will leave execution of command at the background. If the session of its **parental terminal stops** during execution then **kernel process becomes its parental** and by default sends **SIGTERM** to it.

Organizing first script.

So let's prepare for creating our first script in terminal... Call **vim**.

Time to give comments on use of VIM

Let’s do one more script

Task to prepare script using loop-ssh-scp

```
for i in 1 2 3 4 5 6
 do
    echo $i
done

for i in ‘seq 1 6‘
 do
    echo $i
done
```

HW: organize FOR loop printing the even numbers only from 100 to 1000

Arrays in bash

```
#!/bin/bash
my_array=(apple banana "Fruit␣Basket" orange)
new_array[2]=apricot

my_array=(apple banana "Fruit␣Basket" orange)
echo  ${#my_array[@]}                   # 4
```

Arrays in bash

```
my_array=(apple banana "Fruit␣Basket" orange)
echo ${my_array[3]}
# orange - note that curly brackets are needed
# adding another array element
my_array[4]="carrot"
# value assignment without a $ and curly brackets
echo ${#my_array[@]}
# 5
echo ${my_array[${#my_array[@]}-1]}
# carrot
my_array[0]="bad␣apple"
echo {$my_array[0]}
```

HW: organize FOR loop printing the elements of array

Arithmetics and pipes

```
A=3
B=$((100 * $A + 5)) # 305
cat somefile.txt | sort
```

HW: compute 100 + 0.5 in bash

Tip: float operations need pipe and *bc*

If fi

```
NAME="John"
if [ "$NAME" = "John" ]; then
  echo "True␣-␣my␣name␣is␣indeed␣John"
fi

NAME="George"
if [ "$NAME" = "John" ]; then
  echo "John␣Lennon"
elif [ "$NAME" = "George" ]; then
  echo "George␣Harrison"
else
  echo "This␣leaves␣us␣with␣Paul␣and␣Ringo"
fi
```

HW: check if subdirectory "Linux" exists in present directory. If yes, print message "course". If no, print message "very easy" and create the "Linux" directory.

Tip: Google for corresponding flag

Secure Shell

SSH provides a secure channel over an unsecured network in a client-server architecture, connecting an SSH client application with an SSH server. Type:

```
ssh studentname@SOMEHOSTIP
```

and after password you will obtain remote connection via bash terminal with server hosted having ip-address SOMEHOSTIP

- **hostname**
- **ifconfig**
- **df**
- **du**
- **exit**

Secure Copy

- Copy file to host:
  scp LocalFile user@remotehost:directory/RemoteFile
- Copy file/folder from host:
  scp -r user@remotehost:directory/Folder LocalFolder

Commands Fest

- **man bash**
- **date**, **df**, **man date**, **cal**
- **pwd**, **cd**, **cd** $\sim$, **cd -**, **cd /usr/bin**, **ls**
- **file filename**, **less filename**, **ls** $\sim$ **/usr -rt**
- **cp**, **mv**, **mkdir**, **rm**, **ln**
- **type**, **which**, **man**, **apropos**, **info**, **whatis**, **alias**, **help**

Sum up of HW

Scripts should be submitted through MADE system within single zip archive or as link to github-rep devoted to course:

1. Organize FOR loop printing the even numbers only from 100 to 1000
2. Initialize the array of 10-20 elements and organize FOR loop printing the elements of array
3. Compute 100 + 0.5 in bash
4. Check if file "Linux" exists in present directory. If yes, print message "course". If no, print message "very easy" and create the "Linux" file with text "course is easy".

Reminder

- We can login to the server (ssh -X user@ipaddress, scp)
- We can surf through the file system (cd, cp, ls, mv, rm, mkdir)
- We can get help/info (type, which, man, apropos, info, whatis, alias, help)
- In fact **cd**, **cp**, **ls**, **mv**, **rm**... are programs (executables). We can run them from anywhere. Is this right for all programs?
- *Example with "hello world!"* – No!

The PATH environment variable.

Type a command

```
>echo $PATH
```

The PATH *environment variable* defines additional paths where the *shell* should search executables.

The first is found, the first is executed.

- *Example with "hello world 1 and hello world 2"*

Example with "hello world 1 and hello world 2"

```
cd .
cd ..
cd ~
cd
cd -

cd ./dirhw1
g++ hworld_01.cpp -o hw_0x
cd ..  ; cd ./dirhw2
cd ..  && cd ./dirhw2
g++ hworld_02.cpp -o hw_0x
pwd
export PATH=$(HOME)path_to_current_location:$PATH
echo $PATH
```

Environment Variables

`env` – set environment and execute command, or print environment

HOME

PATH

OMP_NUM_THREADS

PWD

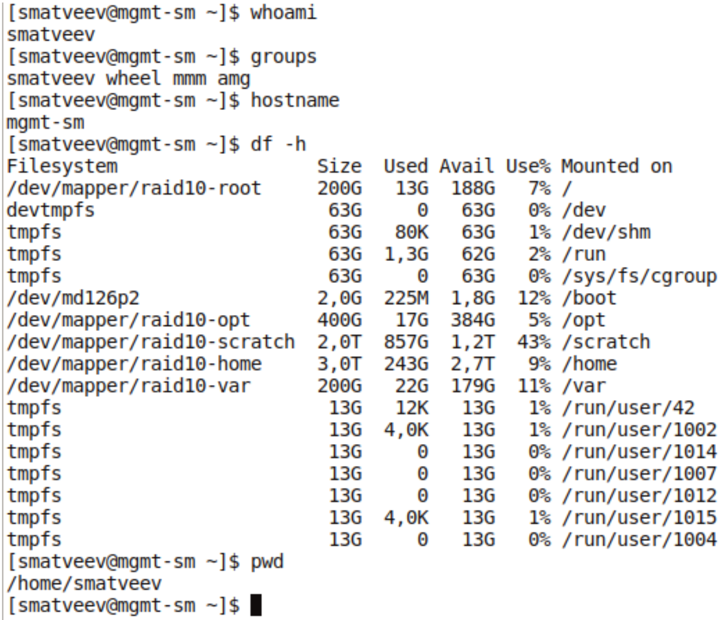
I_MPI_ROOT=/opt/intel/compilers_and_libraries_2017.0.098/linux/mpi

```
env | grep OMP
```

.bashrc

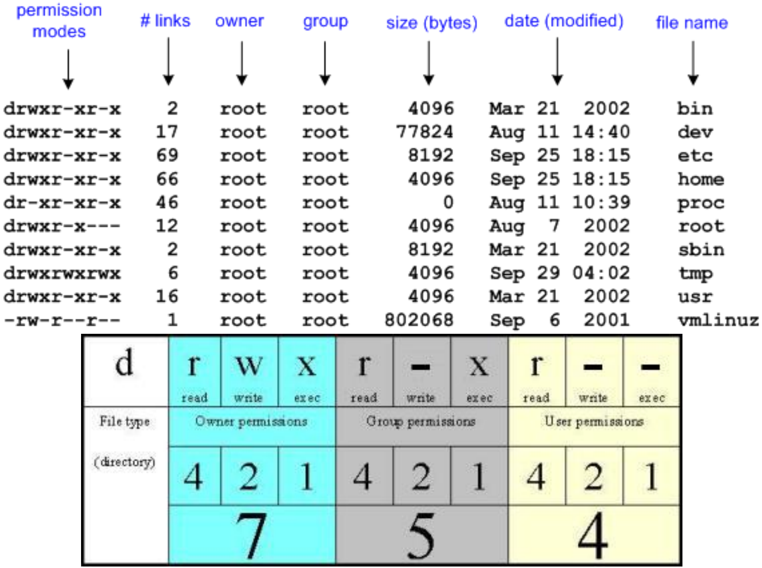
```
export OMP_NUM_THREADS=36
ulimit -u 32000
ulimit -n 10000
export PATH="/usr/local/make-4.2/bin:$PATH"
export MAKE=make-4.2
export LD_LIBRARY_PATH="/usr/lib/gcc/4.4.7:$LD_LIBRARY_PATH"
alias cdlec="cd /Users/data/projects/ISP_Linux/02_Lecture"
```

Learn your usergroup



```
[smatveev@mgmt-sm ~]$ whoami
smatveev
[smatveev@mgmt-sm ~]$ groups
smatveev wheel mmm amg
[smatveev@mgmt-sm ~]$ hostname
mgmt-sm
[smatveev@mgmt-sm ~]$ df -h
Filesystem                   Size  Used Avail Use% Mounted on
/dev/mapper/raid10-root      200G   13G  188G   7% /
devtmpfs                      63G     0   63G   0% /dev
tmpfs                         63G   80K   63G   1% /dev/shm
tmpfs                         63G  1,3G   62G   2% /run
tmpfs                         63G     0   63G   0% /sys/fs/cgroup
/dev/md126p2                 2,0G  225M  1,8G  12% /boot
/dev/mapper/raid10-opt       400G   17G  384G   5% /opt
/dev/mapper/raid10-scratch  2,0T  857G  1,2T  43% /scratch
/dev/mapper/raid10-home      3,0T  243G  2,7T   9% /home
/dev/mapper/raid10-var       200G   22G  179G  11% /var
tmpfs                         13G   12K   13G   1% /run/user/42
tmpfs                         13G  4,0K   13G   1% /run/user/1002
tmpfs                         13G     0   13G   0% /run/user/1014
tmpfs                         13G     0   13G   0% /run/user/1007
tmpfs                         13G     0   13G   0% /run/user/1012
tmpfs                         13G  4,0K   13G   1% /run/user/1015
tmpfs                         13G     0   13G   0% /run/user/1004
[smatveev@mgmt-sm ~]$ pwd
/home/smatveev
[smatveev@mgmt-sm ~]$ █
```

Reminder about attributes and permissions

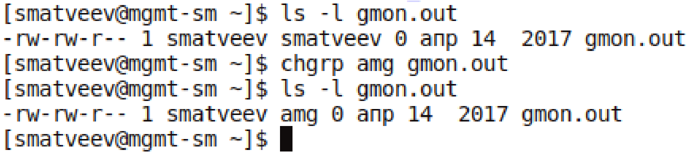


```
drwxr-xr-x    2   root   root     4096   Mar 21  2002    bin
drwxr-xr-x   17   root   root    77824   Aug 11 14:40    dev
drwxr-xr-x   69   root   root     8192   Sep 25 18:15    etc
drwxr-xr-x   66   root   root     4096   Sep 25 18:15    home
dr-xr-xr-x   46   root   root        0   Aug 11 10:39    proc
drwxr-x---   12   root   root     4096   Aug  7  2002    root
drwxr-xr-x    2   root   root     8192   Mar 21  2002    sbin
drwxrwxrwx    6   root   root     4096   Sep 29 04:02    tmp
drwxr-xr-x   16   root   root     4096   Mar 21  2002    usr
-rw-r--r--    1   root   root   802068   Sep  6  2001    vmlinuz
```

d	r	w	x	r	-	x	r	-	-
	read	write	exec	read	write	exec	read	write	exec
File type (directory)	Owner permissions			Group permissions			User permissions		
	4	2	1	4	2	1	4	2	1
	7			5			4		

chown and **chgrp** Changing owner and usergroup

- chgrp – change file usergroup
  Syntax: **chgrp usergroup filename**[1]



```
[smatveev@mgmt-sm ~]$ ls -l gmon.out
-rw-rw-r-- 1 smatveev smatveev 0 апр 14  2017 gmon.out
[smatveev@mgmt-sm ~]$ chgrp amg gmon.out
[smatveev@mgmt-sm ~]$ ls -l gmon.out
-rw-rw-r-- 1 smatveev amg 0 апр 14  2017 gmon.out
[smatveev@mgmt-sm ~]$ 
```

- chown – change file ownership (may require priviledged permissions)
  Syntax: **chown owner filename**

[1] use -**R** key for directories

Managing permissions

chmod

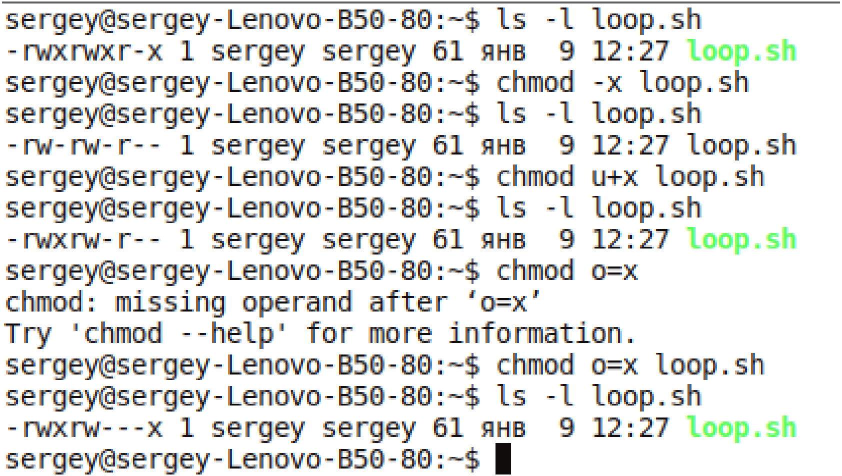
Flags for different roles:

- **u** is for user
- **g** is for groups
- **o** is for others

Flags for permissions

- **r** is for read permission
- **w** is for write permission
- **x** is for execute permission

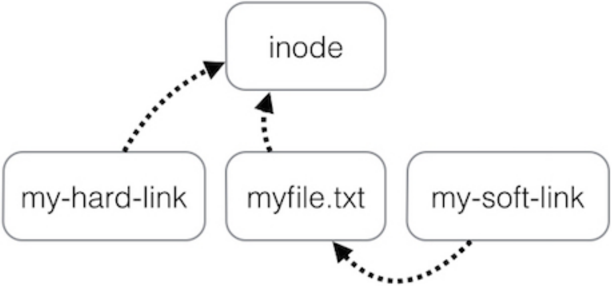
Managing permissions



```
sergey@sergey-Lenovo-B50-80:~$ ls -l loop.sh
-rwxrwxr-x 1 sergey sergey 61 янв  9 12:27 loop.sh
sergey@sergey-Lenovo-B50-80:~$ chmod -x loop.sh
sergey@sergey-Lenovo-B50-80:~$ ls -l loop.sh
-rw-rw-r-- 1 sergey sergey 61 янв  9 12:27 loop.sh
sergey@sergey-Lenovo-B50-80:~$ chmod u+x loop.sh
sergey@sergey-Lenovo-B50-80:~$ ls -l loop.sh
-rwxrw-r-- 1 sergey sergey 61 янв  9 12:27 loop.sh
sergey@sergey-Lenovo-B50-80:~$ chmod o=x
chmod: missing operand after ‘o=x’
Try 'chmod --help' for more information.
sergey@sergey-Lenovo-B50-80:~$ chmod o=x loop.sh
sergey@sergey-Lenovo-B50-80:~$ ls -l loop.sh
-rwxrw---x 1 sergey sergey 61 янв  9 12:27 loop.sh
sergey@sergey-Lenovo-B50-80:~$ █
```

Soft and Hard links

Hard link – is a file pointing into the **decriptor** of its target.
Soft (a.k.a. symbolic) link – is a file pointing to the **name** of its target.



Thus, **hard** link works only inside of OS-filesystem and soft link may lead to files out of filesystem (e.g. to some network HDD).

Soft and Hard links

Create two files:

```
$ touch Cat.txt
$ touch Dog.txt
```

Enter some data into them:

```
$ echo "Cat" > Cat.txt
$ echo "Dog" > Dog.txt
```

And as expected:

```
$cat Cat.txt; cat Dog.txt
Cat
Dog
```

Let's create hard and soft links:

```
$ ln Cat.txt Cat-hard
$ ln -s Dog.txt Dog-soft
```

Soft and Hard links

Let's see what just happened:

```
$ ls -l
Cat.txt
Cat-hard
Dog.txt
Dog-soft -> Dog.txt
```

Changing the name of Cat.txt does not matter:

```
$ mv Cat.txt Cat-new.txt
$ cat Cat-hard
Cat
```

Cat-hard points to the inode (file descriptor), the contents, of the file - that wasn't changed.

```
$ mv Dog.txt Dog-new
$ ls Dog-soft
Dog-soft
$ cat Dog-soft
cat: Dog-soft: No such file or directory
```

Copy data from outer internet

If you have some data at external server you can always use **wget**! Let's download NLOPT library during ssh-session:

```
wget https://github.com/stevengj/nlopt/archive/v2.6.2.tar.gz
```

Pack/Unpack archive

- Compress archive

  ```
  tar -zcvf file.tar.gz file1 file2 file3
  ```

- List of files

  ```
  tar -tf file.tar.gz
  ```

- Unpack tar-archive

  ```
  tar -xzvf file.tar.gz
  ```

- Append file into archive

  ```
  tar -rzvf file.tar.gz file7
  ```

- Remove files from archive

  ```
  tar -f file.tar --delete file1 file2
  ```

Pack/Unpack archive

NLOPT example compilation:

- Unpack nlopt2.6.2 archive
- Run the configure script
- cmake .
- make
- make install [2]
- -DCMAKE_INSTALL_PREFIX=[PATH TO LOCAL INSTALL]
- make install

and do the same for the remote host
Sometimes there are shortcuts e.g:

apt-cache search nlopt
apt-cache search nlopt | grep nlopt
sudo apt-get install libnlopt-dev

[2] fail

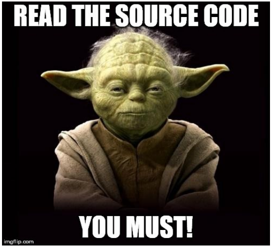
Git gentleman's set

- git clone
- git pull
- git add
- git commit
- git push[3]

[3] branching and synchronizing is more advanced, not for today lecture

C project with multiple source code files

Our model project should consist from 2 source code files and 1 main file.
How to organize work with it?



- *liblib.c* – source code
- *liblib2.c* – source code 2
- *liblib.h* – header file
- *main.c* – main number 1

more advanced VIM options

- Visual mode
- Multiple files
- Copy cut and paste tips
- Replacing strings

mode advanced VIM options

The *:substitute* command searches for a text pattern, and replaces it with a text string.
There are many options, but these are what you probably want[4]

- :%s/foo/bar/g
  Find each occurrence of 'foo' (in all lines), and replace it with 'bar'.
- :s/foo/bar/g
  Find each occurrence of 'foo' (in the current line only), and replace it with 'bar'.
- :%s/foo/bar/gc
  Change each 'foo' to 'bar', but ask for confirmation first.
- :term
- :!sh

[4] https://vim.fandom.com/wiki/Search_and_replace

Compiler

- Check the compiler and its version:
  which gcc
  gcc $--$ **version**
- Prepare the object files:
  -c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file

Ok. Let's do it

```
gcc -c liblib.c
gcc -c liblib2.c
gcc -c main.c
gcc main.c liblib.c liblib2.c -o EXAMPLE.exe
gcc main.c liblib.o liblib2.o main.o -o EXAMPLE.e
```

```
main.o: In function ‘main’:
main.c:(.text+0x0): multiple definition of ‘main’
/tmp/ccp6FWPX.o:main.c:(.text+0x0):
first defined here
collect2: error: ld returned 1 exit status
```

FAIL

ld

Our superhero here is **ld** – linker!
The corresponding environment variables are LD_LIBRARY_PATH LIBRARY_PATH

Ok. Next try

```
gcc -c liblib.c
gcc -c liblib2.c
gcc -c main.c
gcc main.c liblib.o liblib2.o -o EXAMPLE.exe
```

BETTER

Ok. Use ar to combine objectives

```
gcc -c liblib.c
gcc -c liblib2.c
gcc -c main.c
ar rc minilib.a liblib.o
gcc main.c minilib.a -o EXAMPLE.exe
gcc main2.c minilib.a -o EXAMPLE2.exe
```

BEST

minilib.a is a static library!

And don't forget optimization or debug!

```
gcc -c liblib.c -O3
gcc -c liblib2.c -g
gcc -c main.c
ar rc minilib.a liblib.o
gcc main.c minilib.a -o EXAMPLE.exe
gcc main2.c minilib.a -o EXAMPLE2.exe
```

gdb – is GNU debugger for C/C++/Fortran (when printf is not enough for debug)

Dynamic linking

```
gcc -c -Wall -Werror -fpic foo.c
gcc -shared -o libfoo.so foo.o
gcc -Wall -o test main.c -lfoo
/usr/bin/ld: cannot find -lfoo
collect2: ld returned 1 exit status
gcc -L$PWD/foo  -o test main.c -lfoo
echo $LD_LIBRARY_PATH
gcc -L$PWD/foo -Wl,-rpath=/home/username/foo
      -o test main.c -lfoo
```

* PIC – Position Independent Code

Thee Pillars of Makefile

target: dependencies
[tab] command

Makefile example

Download sources from host.

wget http://spring.inm.ras.ru/html/vtm/Example.tar

and unpack them:

tar -xvf Example.tar

Simplest way for compilation:

```
g++ main.cpp hello.cpp factorial.cpp -o hello
```

Assume you have 100 sources... Is it normal?

Preraration of Makefile is a solution. Simplest Makefile will be

```
all:
        g++ main.cpp hello.cpp factorial.cpp -o hello
```

Still strange...

```
all: hello
hello: main.o factorial.o hello.o
        g++ main.o factorial.o hello.o -o hello
main.o: main.cpp
        g++ -c main.cpp
factorial.o: factorial.cpp
        g++ -c factorial.cpp
hello.o: hello.cpp
        g++ -c hello.cpp
clean:
        rm -rf *.o hello
```

Continue...

- make
- make clean
- make main
- make hello

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
        $(CC) $(LDFLAGS) $(OBJECTS) -o $@
hello.o: hello.cpp
        $(CC) $(OPTIMIZATION) $(CFLAGS) $< -o $@
.cpp.o:
        $(CC) $(CFLAGS) $< -o $@
clean:
        rm -f *.o hello
```

Magic Symbols

- $@ is a macro that refers to the target
- $ < is a macro that refers to the first dependency
- $^ is a macro that refers to all dependencies
- % is a macro to make a pattern that we want to watch in both the target and the dependency

Try it!

- make OPTIMIZE=-O3
- make clean
- make hello.o
- make clean
- make hello.o OPTIMIZE=-O3

Configuration files. ed example.

$ wget https://ftp.gnu.org/gnu/ed/ed-1.2.tar.gz
Let's unpack archive and run configuration file. And then make it.

Great! We have compiled total "alien"!

Matmul extension

- Call Blas/Cblas
- numpy.show_config()

Special credits

- Artyom Nikitin
- Marina Munkhoeva
- Anton Maliutin
- Daniil Stefonishin

Research Reproducibility

Minimum Requirements

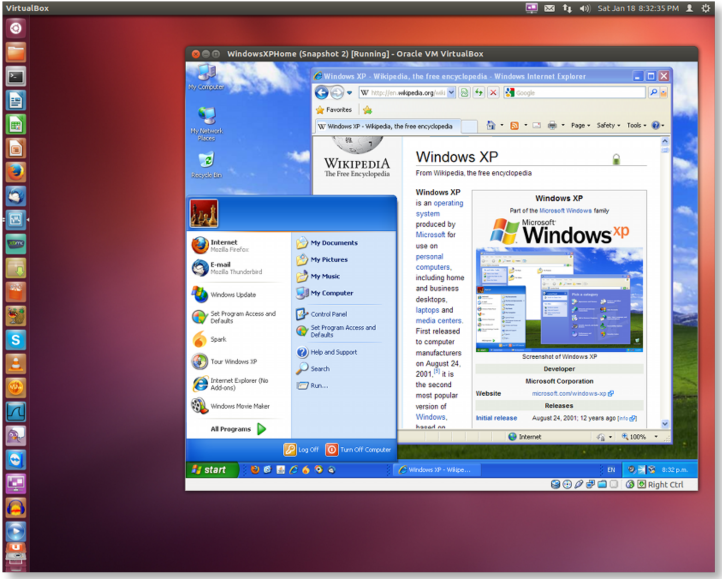
- Available source code
- Thorough documentation
- Comments
- Hardware used
- Software / libraries used
- Step-by-step installation / run instructions

Research Reproducibility

~~Minimum~~ Requirements

- Available source code
- Thorough documentation
- Comments
- Hardware used
- ~~Software / libraries used~~
- ~~Step-by-step installation / run instructions~~
- **ONE-click build / run**

Virtualization



Windows XP

From Wikipedia, the free encyclopedia

Windows XP is an operating system produced by Microsoft for use on personal computers, including home and business desktops, laptops and media centers. First released to computer manufacturers on August 24, 2001,[5] it is the second most popular version of Windows, based on

Windows XP

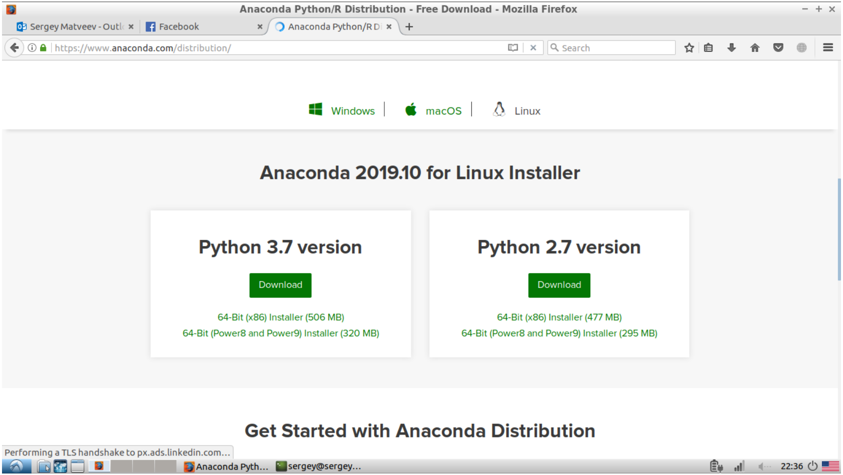
Part of the Microsoft Windows family

Screenshot of Windows XP

Developer	
Microsoft Corporation	
Website	microsoft.com/windows-xp
Releases	
Initial release	August 24, 2001; 12 years ago [info]

Anaconda as alternative

To be honest it is not a true virtualization...



Anaconda Python/R Distribution - Free Download - Mozilla Firefox

Anaconda 2019.10 for Linux Installer

Python 3.7 version

Download

64-Bit (x86) Installer (506 MB)

64-Bit (Power8 and Power9) Installer (320 MB)

Python 2.7 version

Download

64-Bit (x86) Installer (477 MB)

64-Bit (Power8 and Power9) Installer (295 MB)

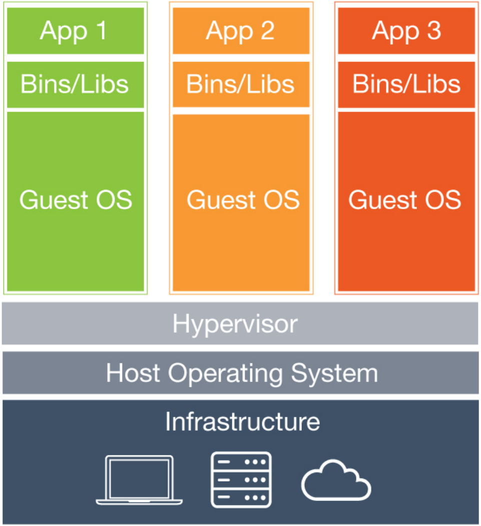
Get Started with Anaconda Distribution

Performing a TLS handshake to px.ads.linkedin.com...

Anaconda as alternative

- CHECK bashrc for modification of PATH and other variables
- **source activate**
- **source deactivate**

Virtualization: Hypervisor



App 1

Bins/Libs

Guest OS

App 2

Bins/Libs

Guest OS

App 3

Bins/Libs

Guest OS

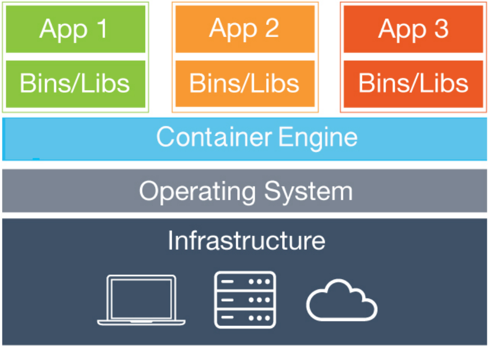
Hypervisor

Host Operating System

Infrastructure

Hypervisor-based Virtualization

Virtualization: Containers



Container virtualization

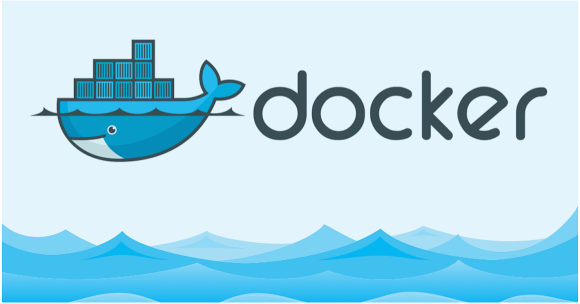
Virtualization: Containers

Hypervisor-based:

- Any OS on any OS
- Complete isolation of guest instances
- Worse performance

Container-based:

- Share kernel with host
- Worse security
- Almost native performance



- Convenient
- Efficient
- Open-source (split recently to CE and EE)
- Huge community

Docker

What you need to know…

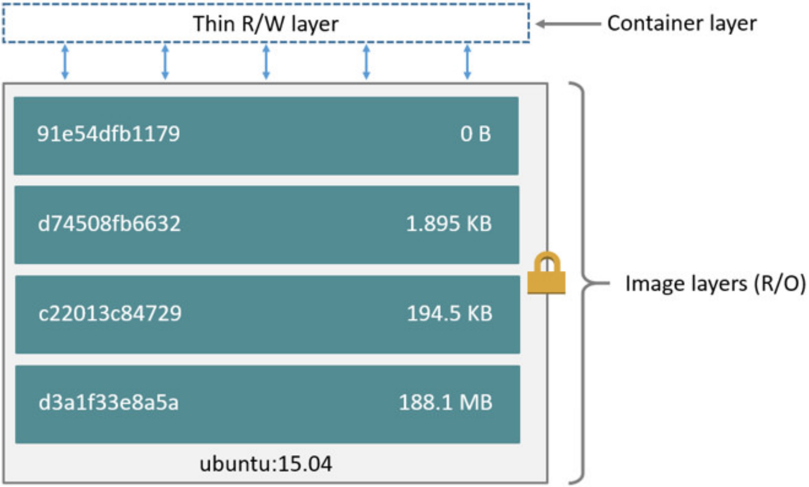
1. *Docker Image* - basis of a container.
    - OS distribution-specific code.
    - Your libraries, source code, data, etc.
2. *Docker Container.*
3. *Docker Engine* - creates, ships and runs docker containers on physical / virtual host.

Build and delete

```
FROM ubuntu:14.04
WORKDIR /home/matseralex/TEST_TEACHING
RUN sudo apt-get update && apt-get install -y python
CMD lsb_release -a
```

```
docker build - < TEST_DOCKER
docker images
docker run IMAGEID
docker tag IMAGEID YOURTAG
docker run -it --entrypoint /bin/bash YOURTAG
docker rmi -f IMAGEID
```

Image: Structure



Thin R/W layer

Container layer

91e54dfb1179 0 B

d74508fb6632 1.895 KB

c22013c84729 194.5 KB

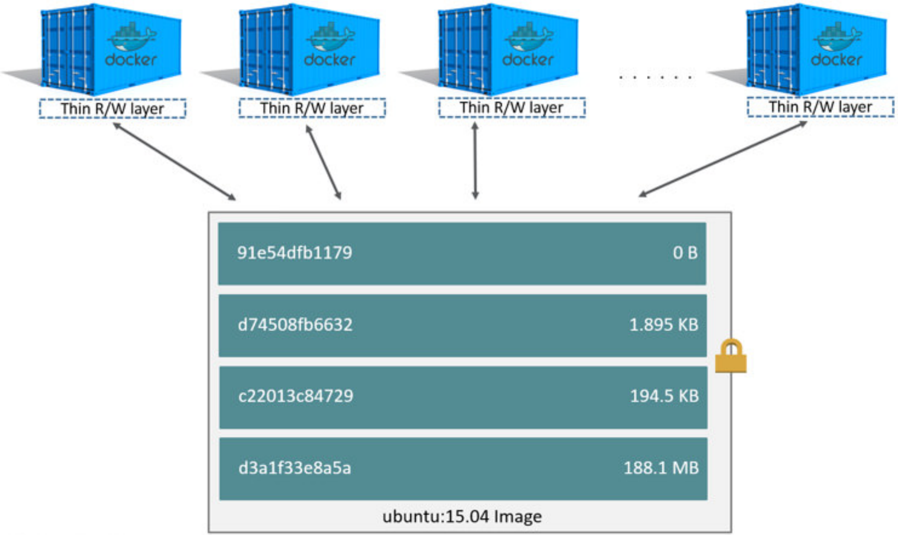
d3a1f33e8a5a 188.1 MB

ubuntu:15.04

Image layers (R/O)

Container
(based on ubuntu:15.04 image)

Image: Structure



91e54dfb1179	0 B
d74508fb6632	1.895 KB
c22013c84729	194.5 KB
d3a1f33e8a5a	188.1 MB

ubuntu:15.04 Image

What could be studied after it?

- More advanced use of Containers and their preparation (also Singularity)
- Details about virtualization layers and cgroups
- Many, many other things – these technologies are fresh!

At home

- Implement classical matrix multiplication and matrix by vector for C/C++ programming language (25%) [5]
- Use static linking, prepare Makefile, compile it with **-g** and **-O3** flags (25%)

  Measure timings for square matrices of size $N = 500, 512, 1000, 1024, 2000, 2048$ (25%)
- Compare timings for virtual box and real machine and within docker container (optional)
- And also... basic bash scripts, please... (25%)
- Additional score for running LINPACK test at your machine (+20%)
- Superbonus: for Strassen matrix multiplication (optional) (+20%)

[5] We will use it for the forthcoming Pagerank-basics and corresponding task