Linux Scripting

Core Skills That Every Roboticist Must Have

Alex Litoiu
alex.litoiu@yale.edu

Topics Covered

- Linux Intro
  - Basic Concepts
  - File system
- Bash Scripting Basics
  - Basic Syntax
  - Basic commands
  - Additional Syntax
- Advanced Bash Scripting
  - Job scheduling

1 Linux Intro

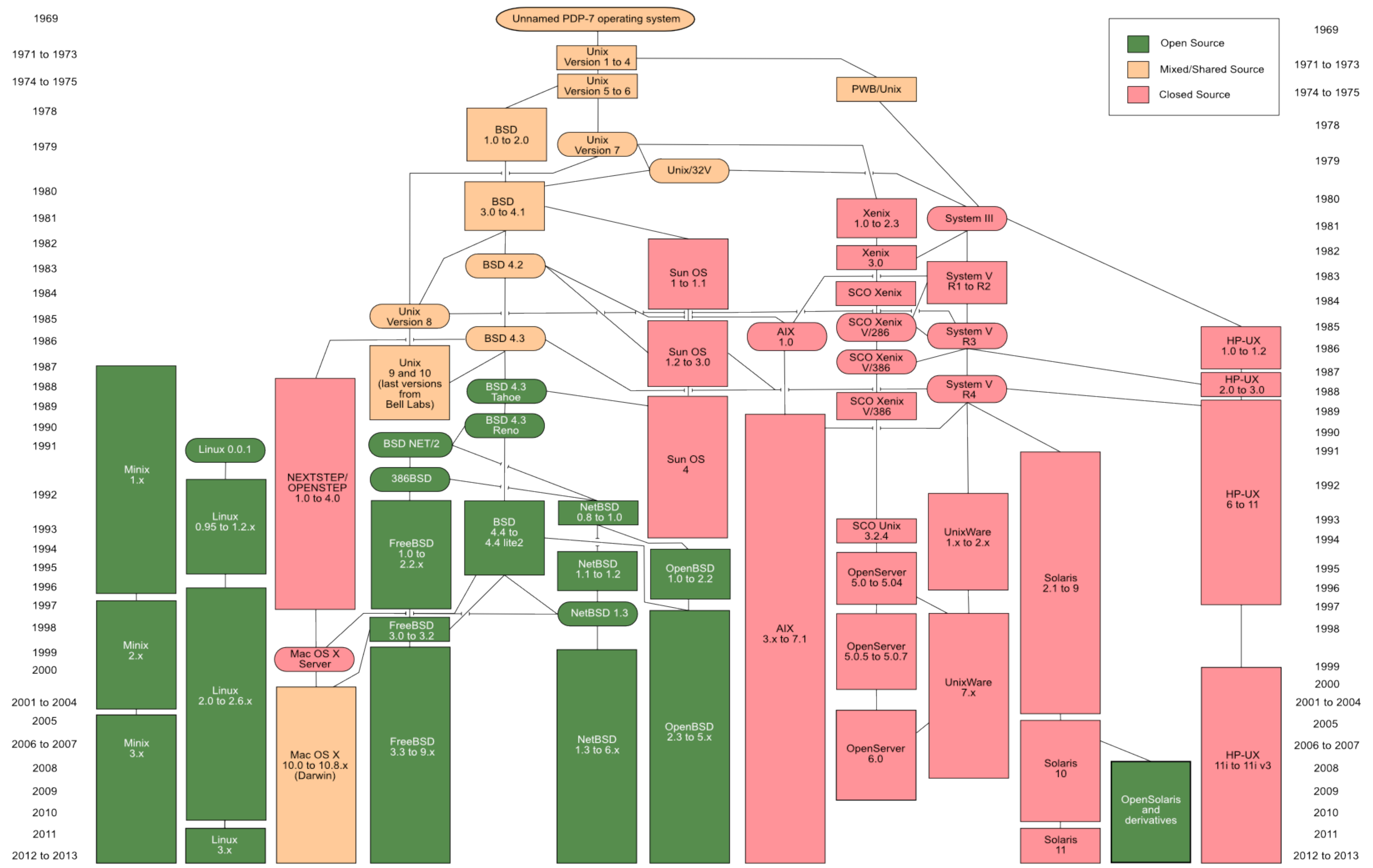
Why Linux?

- Free
- Well-designed
- Flexible
- Standard in academia
- The best technology firms use it
- Used in 92% of 500 world's fastest computers

Linux History - Unix Growth and Fragmentation

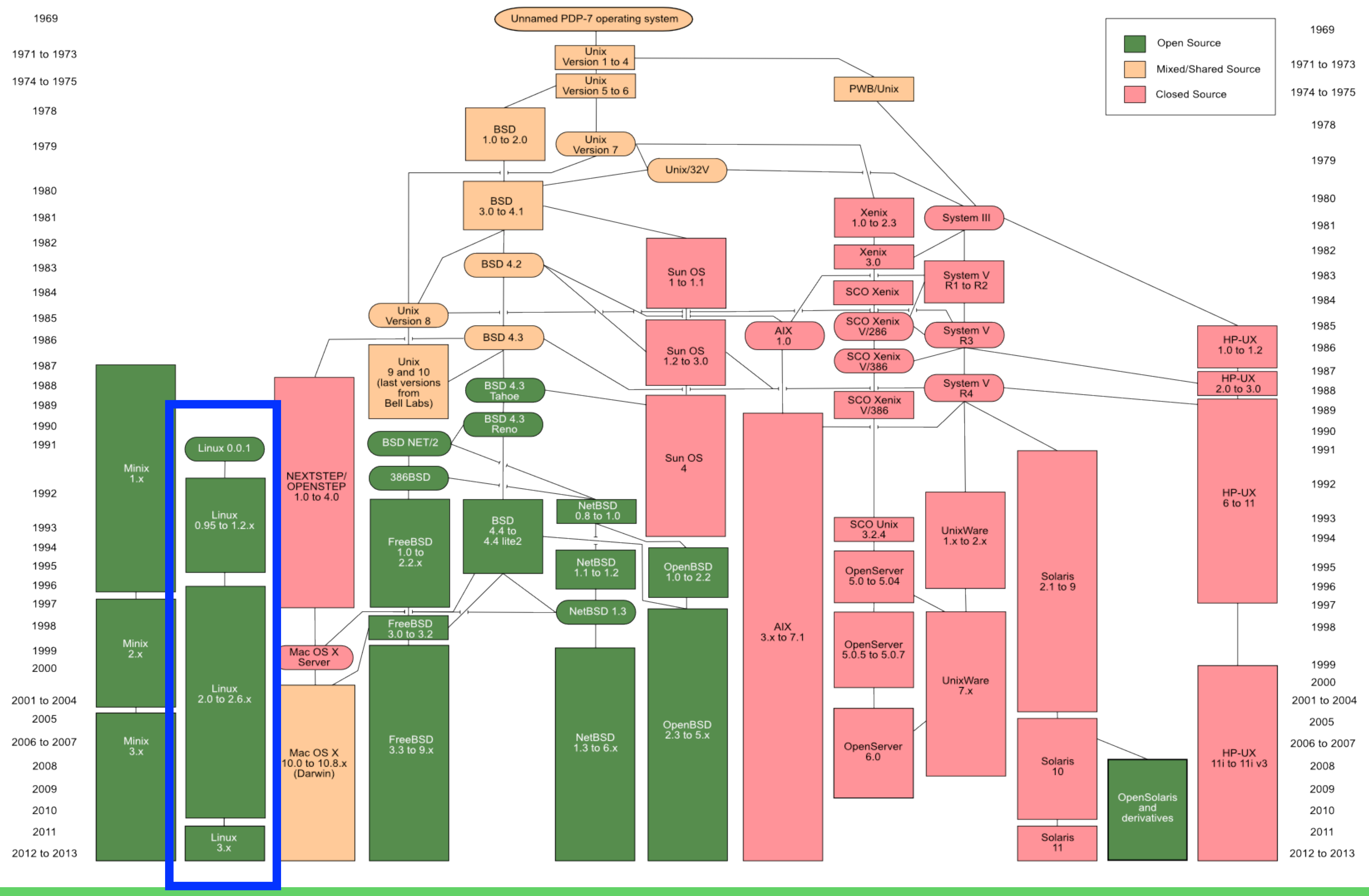
- Unix created in 1969 at Bell Laboratories (Ken Thompson and Dennis Ritchie)
- First operating system ported to C (Thompson and Ritchie)
- Led to it being the first portable OS
- Became very popular but fragmented, as vendors spun off their own Unix versions, optimized to their own hardware

Linux History - Family Tree



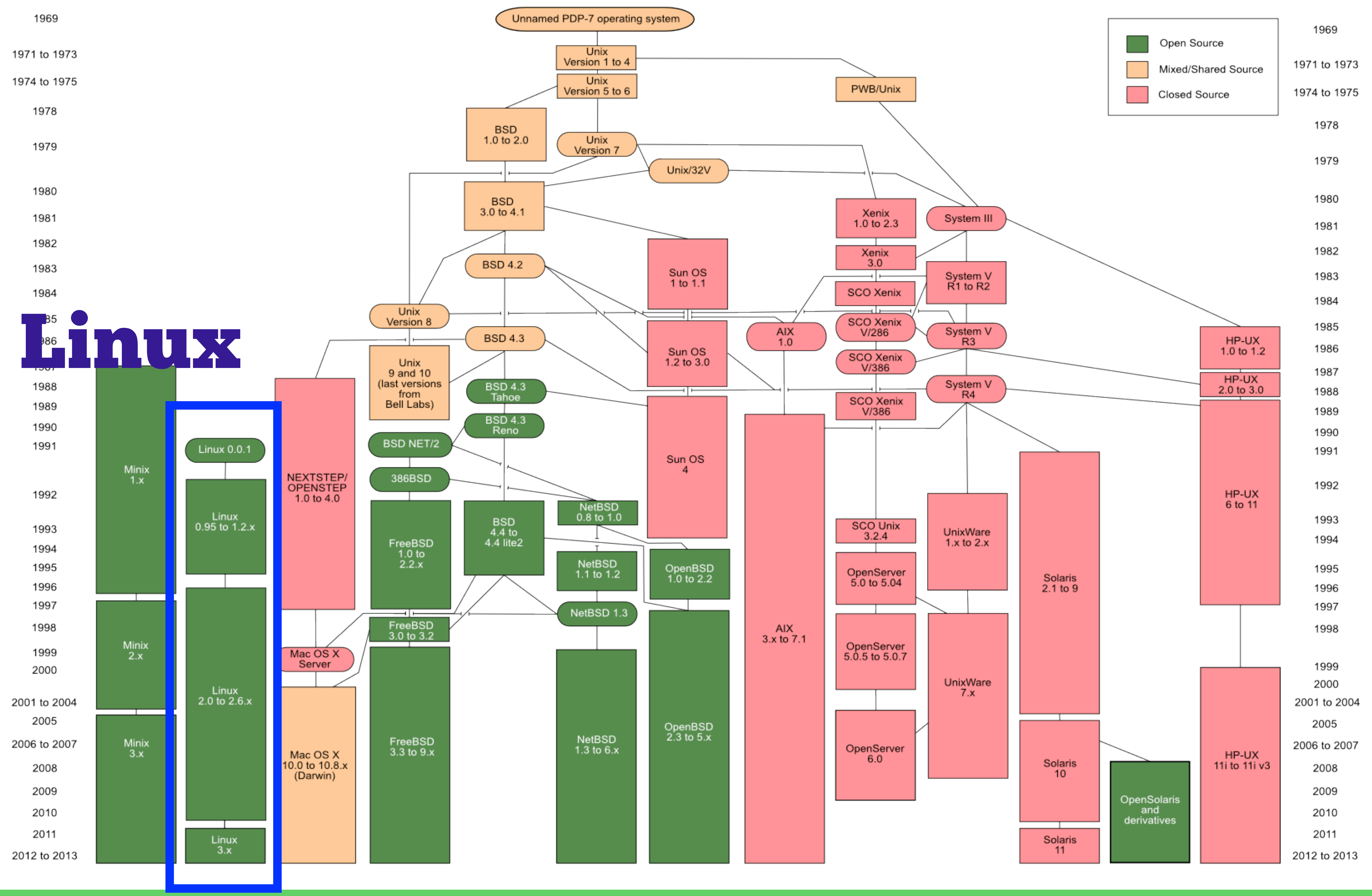
Scazlab

Linux History - Family Tree



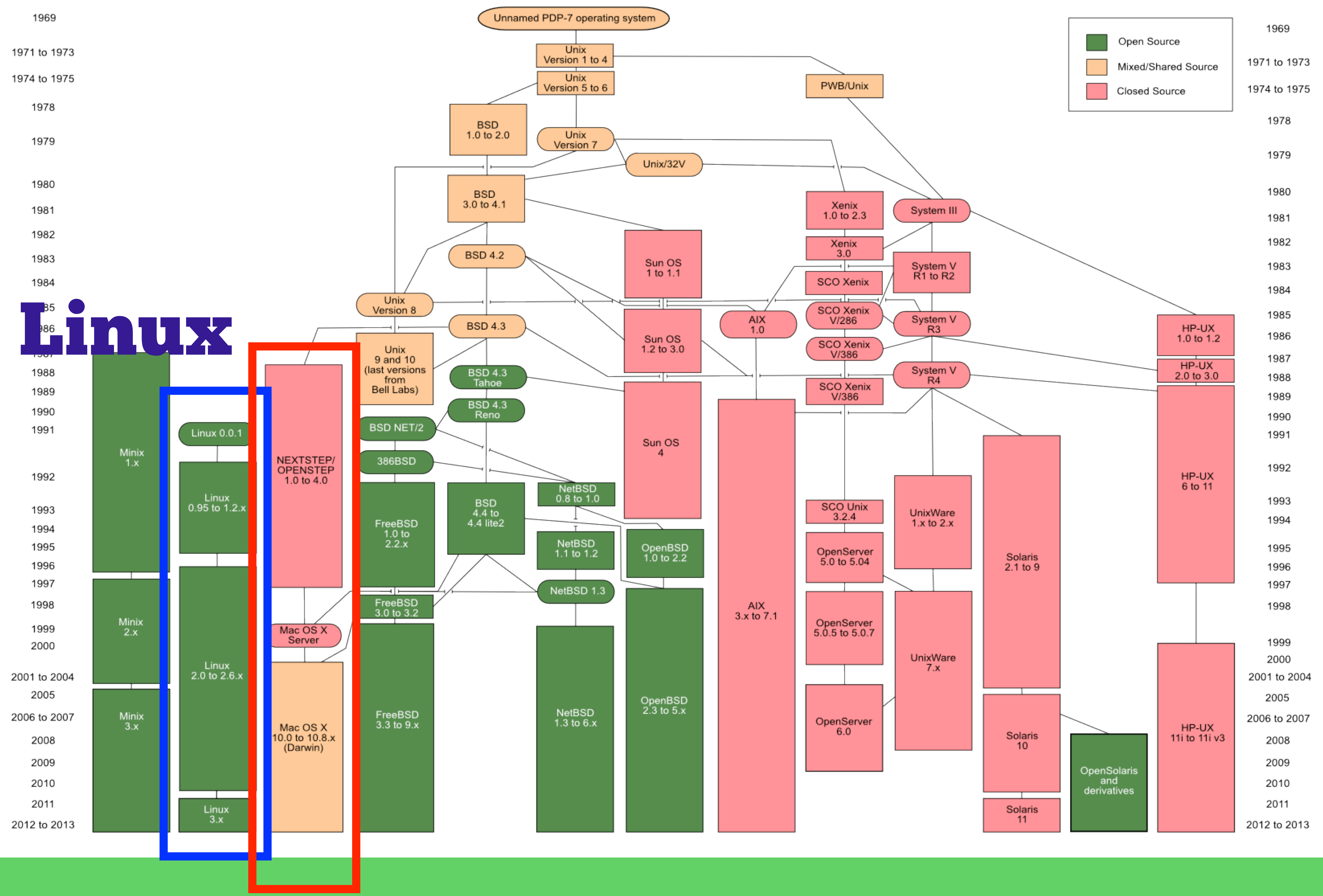
Thursday, November 14, 13

Linux History - Family Tree



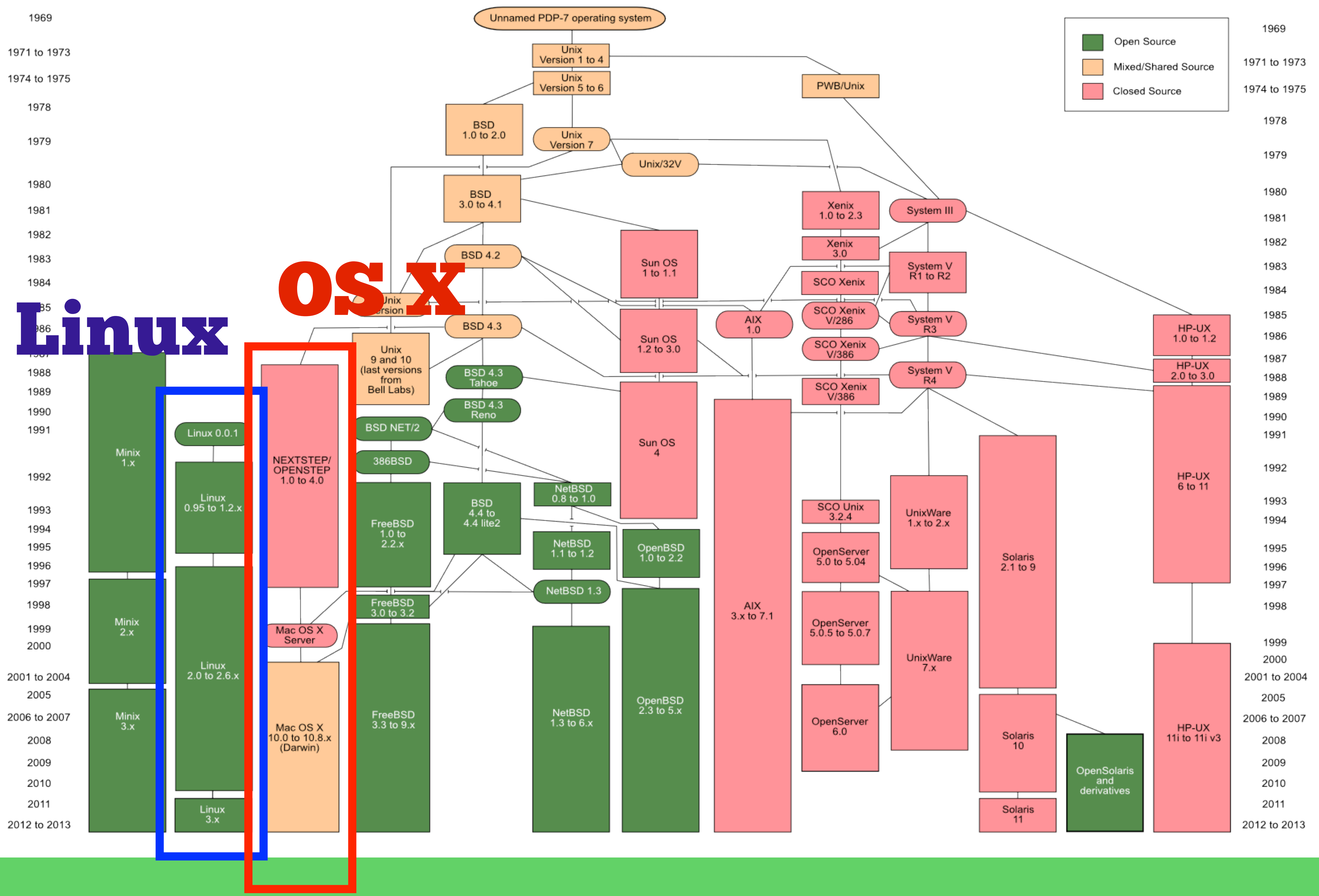
Scazlab

Linux History - Family Tree



Thursday, November 14, 13

Linux History - Family Tree



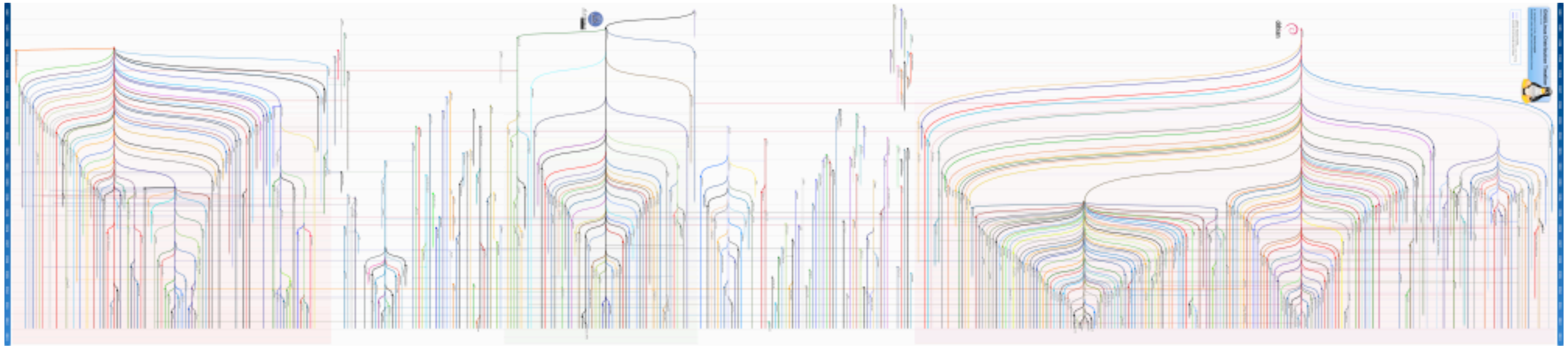
Linux History - Some Consolidation

- In 1985, POSIX (Portable Operating System Interface) standard came about, allowing a program to run on any POSIX systems
  - API to Kernel
  - Shells bundled with OS
  - Utility interfaces
- In 1991, Linus Torvalds released Linux, which has steadily become the most popular open-source descendant of Unix

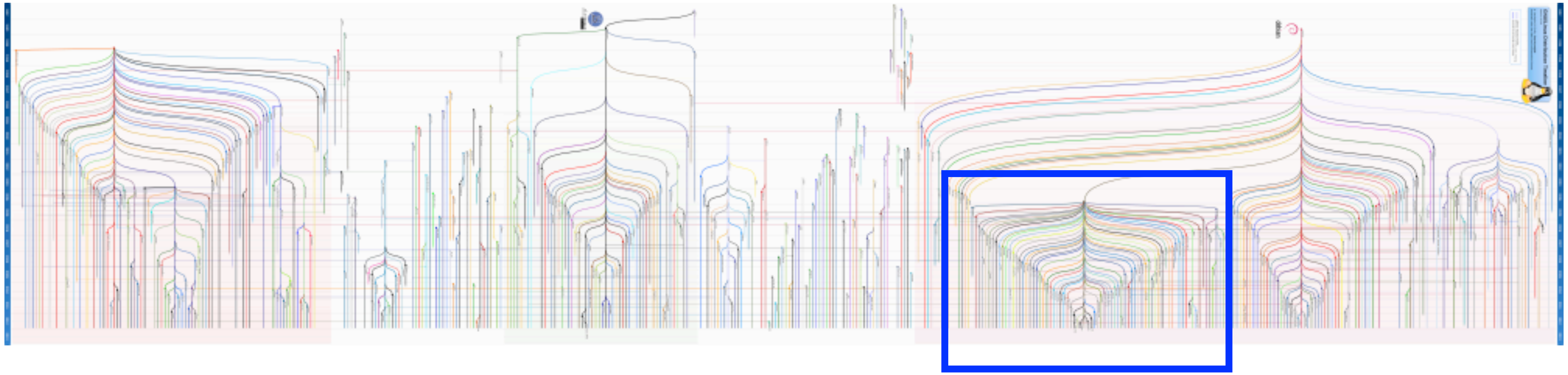
Linux Distributions

- Even within Linux, many different distributions
- **Same:**
  - Linux Kernel
- **Different:**
  - Package manager
  - Windowing system
  - Packages included

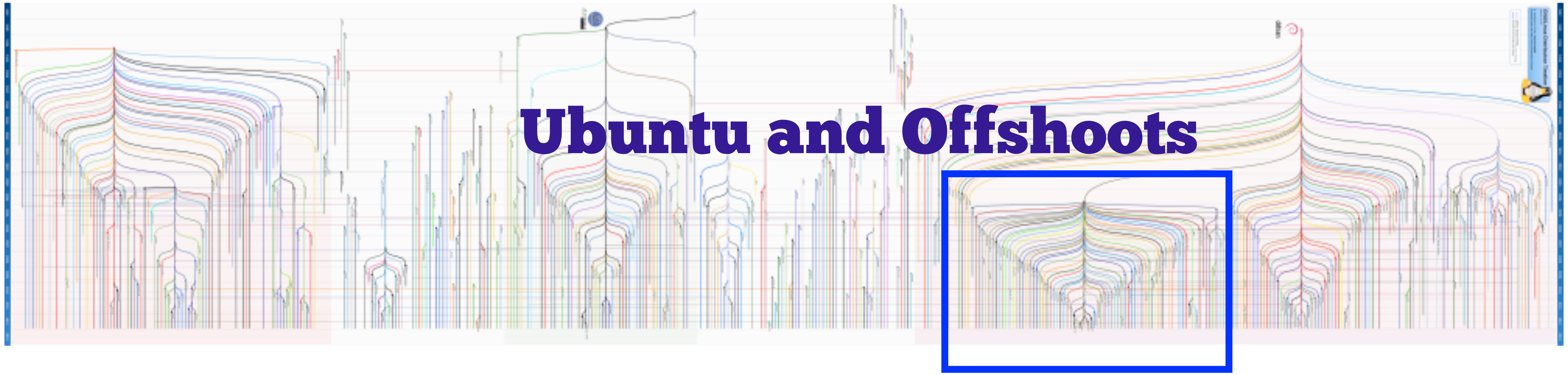
Linux Distributions Chart



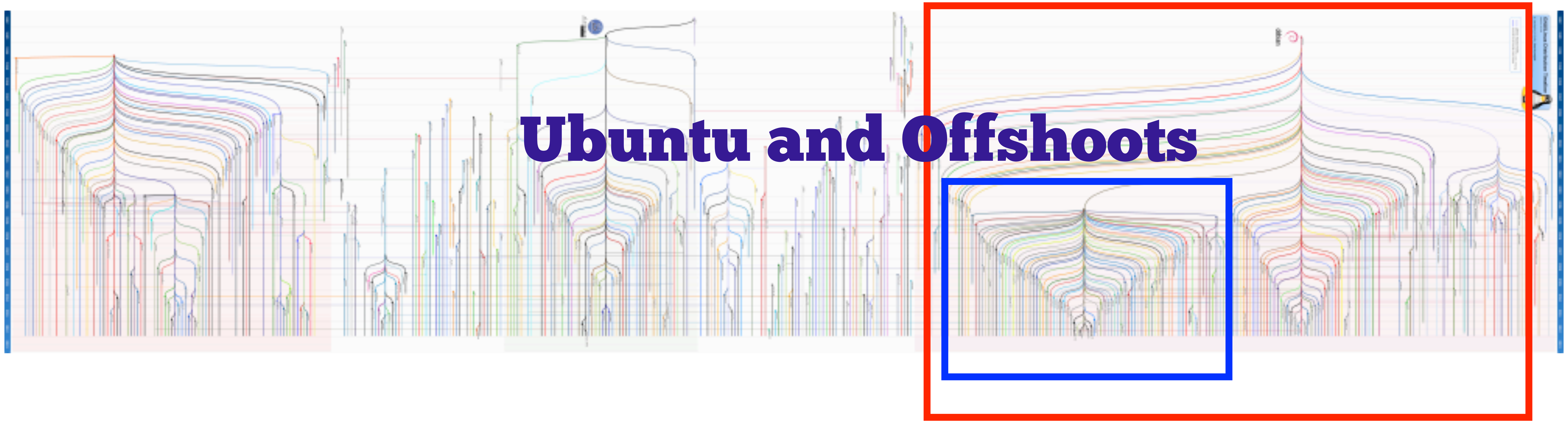
Linux Distributions Chart



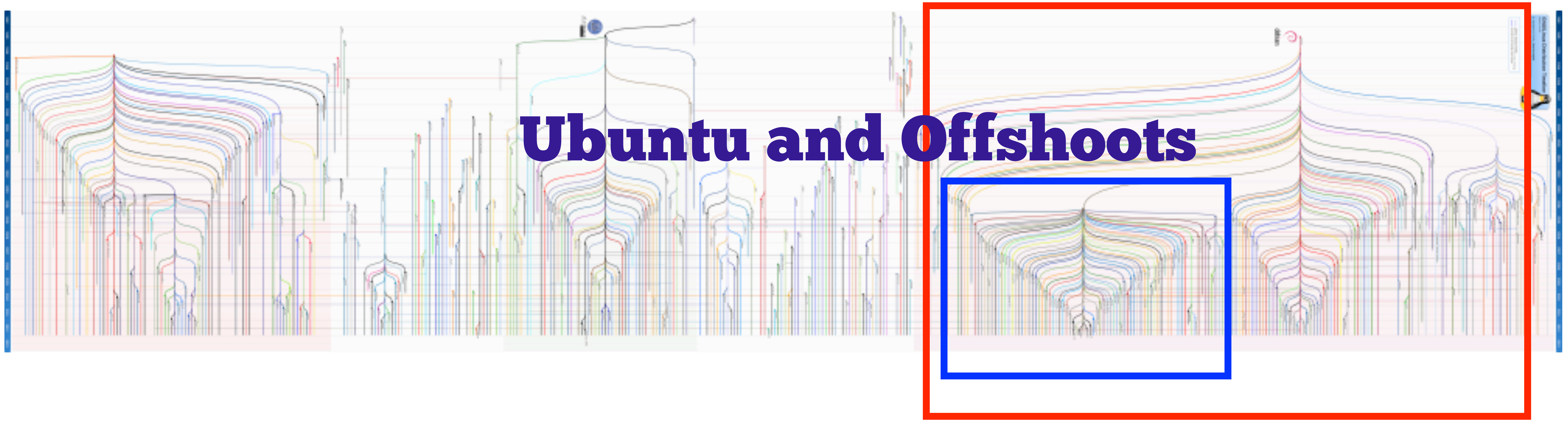
Linux Distributions Chart



Linux Distributions Chart



Linux Distributions Chart



Linux Basic Concepts

- Everything in Linux is a file (identified by a path) or a process (identified by a PID)
- **Examples of Processes:**
  - Bash Shell
  - Browser
- **Examples of Files:**
  - essay.txt (arbitrary data file)
  - /dev/ttyUSB0 (Unix special file - USB interface)
  - /tmp/.X11-unix/X0 (Unix special file - Socket File)

Linux File Structure - Binaries

/

- **/boot** - The startup files and the kernel, vmlinuz
- **/bin** - Common programs, shared by the system, all users
- **/sbin** - Programs for use by the system and the system administrator.
- **/usr** - Programs, libraries, documentation etc. for all user-related programs.
- **/lib** - Library files, includes files
- **/opt** - Typically contains extra and third party software
- ...

Linux File Structure - Config and System State

...

/etc - Most important system configuration files are in /etc
/tmp - Temporary space for use by the system, cleaned upon reboot
/var - Storage for all variable files and temporary files created by users, such as log files
...

Linux File Structure - Other

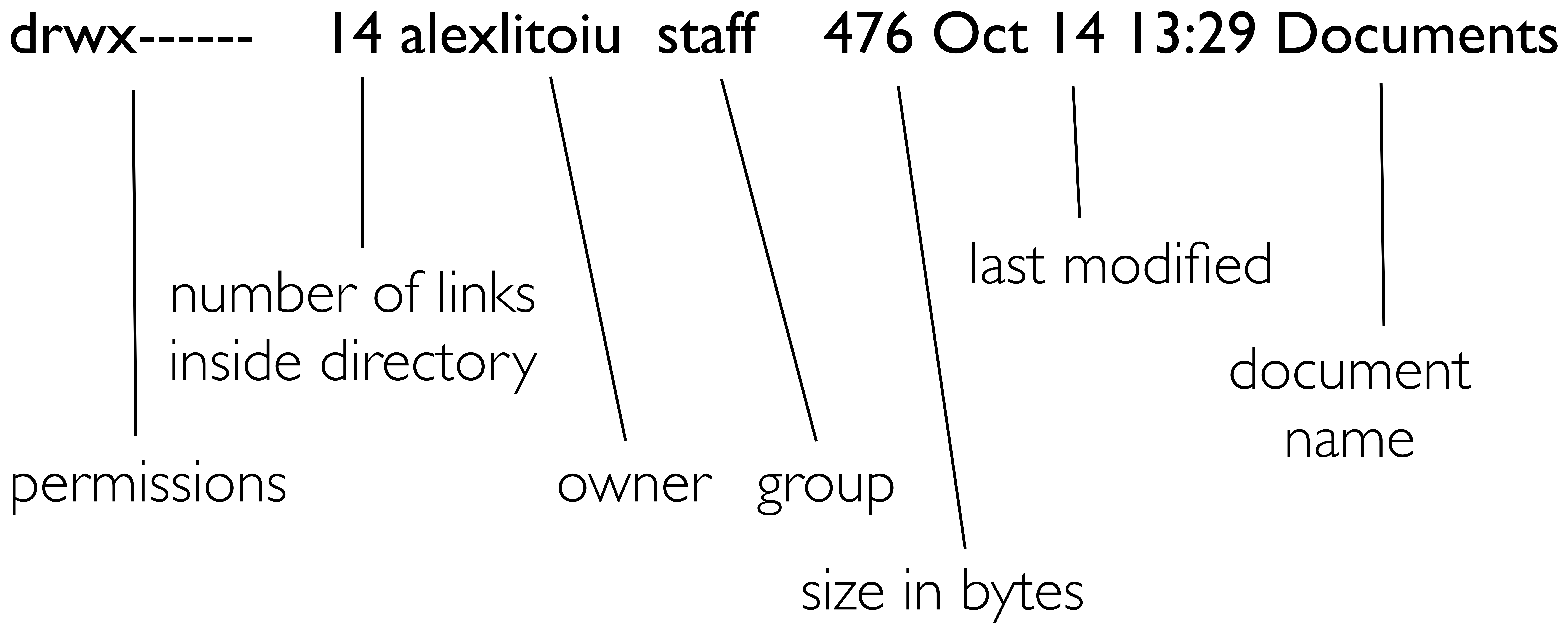
...

/home - Home directories of the common users

/root - The administrative user's home directory

/dev - Contains references to all the CPU peripheral hardware

Linux File Ownerships



```
drwx------    14 alexlitoiu  staff    476 Oct 14 13:29 Documents
```

Linux Change File Ownerships

- The owner of a file, or the administrator can change the owner of the file
- **$ chown new_owner file_name**
- Can also change the group using:
- **$ chgrp new_group file_name**

Manage a User's Groups

- **/etc/group** is the file that contains list of all groups, and the users in each one
- **$ groups user** to list the groups that a user is in

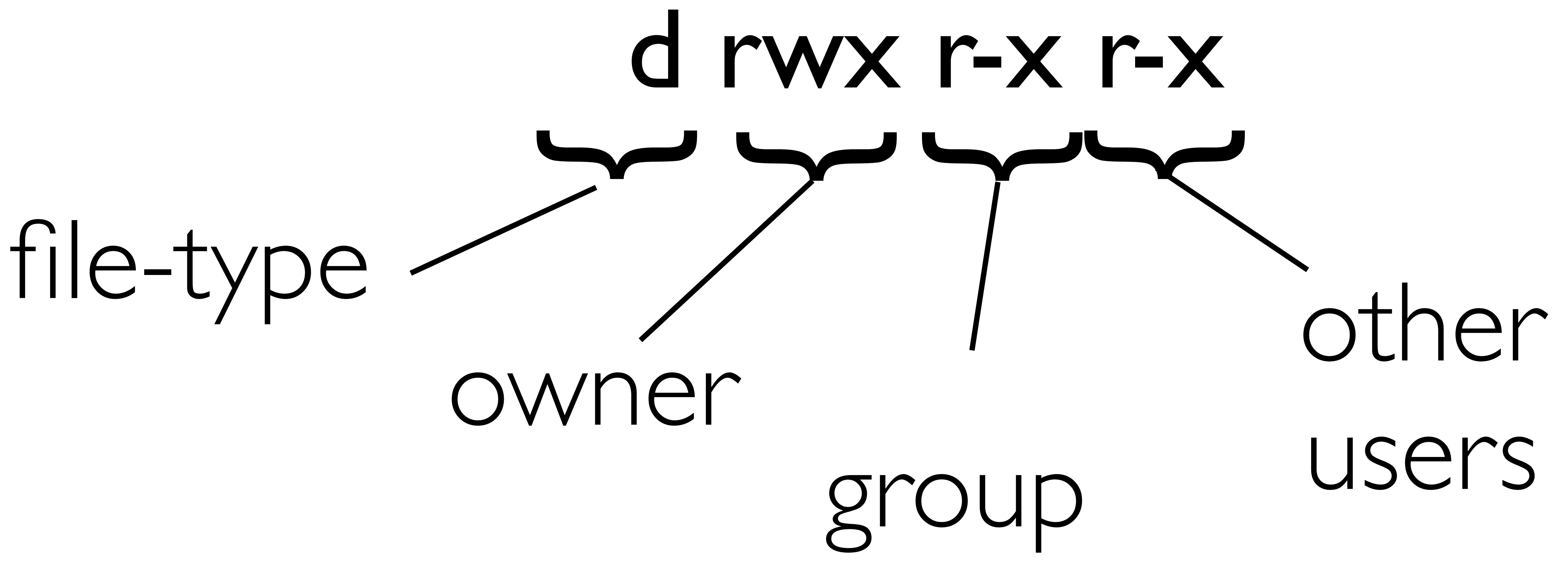
- **$ groupadd new_group** to add a new group to the system
- **$ groupdel old_group** to remove a group to the system

- **$ gpasswd -a user group** add user to group
- **$ gpasswd -d user group** delete user from group

Important Groups

Group	Files affected	Purpose
audio	/dev/audio, /dev/snd/*, /dev/rtc0	Direct access to sound hardware
disk	/dev/sda[1-9], /dev/sdb[1-9]	Access to block devices
optical	/dev/sr[0-9], /dev/sg[0-9]	Access to optical devices (CD/DVD)
video	/dev/fb/0, /dev/misc/agpgart	Access to video capture hardware
lp	/var/cache/cups, /var/spool/cups, /dev/parport[0-9]	Access to printer hardware

Linux File Permissions



File-type

-: regular file

d: directory

p: named pipe

s: socket

c: character device

b: block device

Permissions

r: read

w: write

x: execute

Linux File Permission Representations

Symbolic	Binary	Octal	English
`- --- --- ---`	0 000 000 000	0000	No permissions
`- --x --x --x`	0 001 001 001	0111	Execute
`- -w- -w- -w-`	0 010 010 010	0222	Write
`- -wx -wx -wx`	0 011 011 011	0333	Write, Execute
`- r-- r-- r--`	0 100 100 100	0444	Read
`- r-x r-x r-x`	0 101 101 101	0555	Read, Execute
`- rw- rw- rw-`	0 110 110 110	0666	Read, Write
`- rwx rwx rwx`	0 111 111 111	0777	Full

Changing File Permissions Symbolic Method

- **$ ls -l** to get the file permissions in your current directory
- **$ chmod mode file**
- Mode has 3 sections:
  - Access Class: a (all), u(user), g(group), o(others)
  - Operator: + (add access), -(remove access), = (set exact access)
  - Access Type: r (read), w (write), x (execute)

Examples:

$ chmod a+r lorem.txt (add read access to all users)

$ chmod og-xw lorem.txt (remove execute, write access to other and group)

Changing File Permissions Absolute Mode

- **$ chmod mode file**
- Mode is the octal representation of permissions

Examples:

$ chmod 0700 lorem.txt (set permissions to `- rwx --- ---`)

$ chmod 0644 lorem.txt (set permissions to `- rw- r-- r--`)

2

Bash Scripting

What Shell Am I Using?

- **$ echo $SHELL** to determine which shell you are using
- **$ cat /etc/shells** to list your system’s available shells
- **$ chsh -s shell username** to change your shell to
  - For example, **$ chsh -s /bin/ username** to change your shell to csh

What is Bash?

- Bash is a type of **Shell** - a process that:
  - displays a prompt
  - reads a command
  - process the given command
  - then execute the command
- Written in 1989 by Brian Fox as replacement for Bourne Shell (sh)
- Default shell on Linux and Mac OS X

Executing Path Binaries in Bash

- Example: **$ date "+DATE: %Y-%m-%d TIME: %H:%M:%S"**
  - DATE: 2013-11-14 TIME: 15:43:02
- Bash checks the directories in the **$PATH** variable for a binary named date
- Finds it in /bin/
- Executes /bin/date, with parameter "+DATE: %Y-%m-%d TIME: %H:%M:%S"

Executing Binaries - Absolute Path

- Can execute a binary using the absolute path of the file
- **$ /home/FredStevens/Documents/runExperiment "all trials"**
- /home/FredStevens/Documents/runExperiment is the full path to the binary
- "all trials" is parameter given to the program
- equivalent to **$ ~/Documents /runExperiment "all trials"**

Executing Binaries - Relative Path

- Can also use the relative path of the file
- **$ /home/FredStevens/Documents/runExperiment “all trials”**
- If you are in /home/FredStevens/ can use
  - **$ ./Documents/runExperiment “all trials”**
- If you are in /home/FredStevens/Documents/ can use
  - **$ ./runExperiment “all trials”**

Common Binaries

- **$ ls** list files in current directory
  - **$ ls directory_name** list files in directory directory_name
- **$ pwd** echo the current directory
- **$ echo string** print out the given string
- **$ rm filename** remove file
- **$ cp source_file dest_file** copy source_file to dest_file
- **$ mv source_file dest_file** move source_file to dest_file
- **$ mkdir directory_name** create directory directory_name
- **$ rmdir directory_name** removes the directory directory_name
- **$ kill pid** kill the process with PID number pid

Ways of Running Bash Code

Many ways to run bash code:

1. Type in some bash code, and press enter

Directly

Given a bash script file:

Using Script

2. Run script using **$ bash mybashscript.sh**
3. Run script like a binary, if the file has

   #! /bin/bash

   as the first line of the file

- Run script using **$ ./mybashscript.sh**

mybashscript.sh

```
echo "Hello World"
echo "Files in cur dir:"
ls
```

Bash Syntax - Variables

- Assign variables using **$ VARIABLE=”STRING”**

hello_world.sh

```
#!/bin/bash
 STRING="HELLO WORLD!!!"
 echo $STRING
```

execution

```
$ ./hello_world.sh
HELLO WORLD!!!
```

Bash Syntax - Local Variables

- Assign local variables using **$ local VARIABLE=”STRING”**

variables.sh

```
#!/bin/bash
VAR="global variable"
function locfunc {
    local VAR="local variable"
    echo $VAR
}
echo $VAR
locfunc
echo $VAR
```

execution

```
$ ./variables.sh
global variable
local variable
global variable
```

Bash Syntax - Exported Variables

- If you want a variable from your shell to also be known by sub-processes, use export
  - **$ export PYTHONPATH=/home/alexlitoiu/extra_python_libraries/**
  - **$ ./python**
  - The python process will now know to also look in that folder when looking for files

Bash Syntax - Passing Parameters

- Access parameters using **$1 $2** etc.

```
#!/bin/bash

echo $1 $2 $3
echo $@
echo #@
```

arguments.sh

execution

```
$ ./arguments.sh My three parameters
My three parameters
My three parameters
3
```

Bash Syntax - If Statements

- Use if, then, else, fi for if statements

```
#!/bin/bash
directory="./BashScripting"

# bash check if directory exists
if [[ -d $directory ]]; then
    echo "Directory exists"
else
    echo "Directory does not exist"
fi
```

if_then_else.sh

execution

```
$ ./if_then_else.sh
Directory does not exist
$ mkdir BashScripting
$ ./if_then_else.sh
Directory exists
```

Bash Syntax - Arithmetic Comparisons

C Operator	Bash Operator
`<`	`-lt`
`>`	`-gt`
`<=`	`-le`
`>=`	`-ge`
`==`	`-eq`
`!=`	`-ne`

Bash Syntax - Arithmetic Comparisons

```
#!/bin/bash
num1=5
num2=7

if [[ $num1 -lt $num2 ]]; then
    echo "num1 < num2"
fi
```

comparison.sh

execution

```
$ ./comparison.sh
num1 < num2
```

Bash Syntax - String Comparisons

Bash Operator	In Words
`=`	`equals`
`!=`	`doesn't equal`
`>`	`greater than`
`<`	`less than`
`-n`	`not empty`
`-z`	`empty`

Bash Syntax - String Comparisons

```
#!/bin/bash
string1=”This is a non-empty string”

if [[ -n $string1 ]]; then
    echo $string1
fi
```

comparison.sh

execution

```
$ ./comparison.sh
This is a non-empty string
```

Bash Syntax - String Comparisons

```
#!/bin/bash
string1=”This is a non-empty string”
test=1

if [[ -n $string1 && ($test -eq 1) ]]; then
    echo $string1
fi
```

comparison.sh

execution

```
$ ./comparison.sh
This is a non-empty string
```

Bash Syntax - File Testing

Bash Operator	Tests For
`-d filename`	directory existence
`-e filename`	file or directory existence
`-f filename`	file existance
`-O filename`	file exists and owned by user
`-r filename`	file is readable
`-w filename`	file is writeable
`-X filename`	file is executable

Bash Syntax - For Loop

```
#!/bin/bash
for f in $( ls /var/ ); do
  echo $f
done
```

for.sh

execution

```
$ ./for.sh
agentx
at
audit
...
```

Bash Syntax - For Loop

while.sh

```
#!/bin/bash
COUNT=1
while [[ $COUNT -le 5 ]]; do
   echo $COUNT
   let COUNT=COUNT+1
done
```

execution

```
$ ./while.sh
1
2
3
4
5
```

Bash Syntax - Bash Functions

```
function afunc {
    echo "Inside afunc"
    for param in $@; do
        echo $param
    done
}

afunc a b c d
afunc
```

- Access parameters same way as to the bash script: **$1, $2, $@** etc.
- Call a function using **$ func_name param_1 param_2**

functions.sh

execution

```
$ ./functions.sh
Inside afunc
a
b
c
d
Inside afunc
```

Bash Syntax - Quotes

- Double Quotes, “”, allow $, ` and \ but no other special characters
  - **$ echo “$((5+3)) `whoami` ”**
  - Output: 8 alexlitoiu
- Single Quotes, ‘’, will not allow any special characters. Everything inside the quotes gets printed, literally
  - **$ echo ‘$((5+3)) `whoami` ’**
  - Output: ‘$((5+3)) `whoami` ’

Bash Syntax - Arithmetic

- Assign arithmetic result to a variable using “let” (note the lack of $ symbol)
  - **$ let VAR=VAR+3**
- Use arithmetic within a string, or expression using $((arithmetic))
  - **$ echo ‘VAR + 2 is $((5+2))’**
  - **“VAR + 2 is 7**

Bash Syntax - Data Streams

- Three standard streams
  - *Standard Input (stdin)* reads data
  - *Standard output (stdout)* outputs data
  - *Standard error (stderr)* outputs errors
- All three default to the terminal window (reading from it and writing to it)
- All three streams can be redirected

Bash Syntax - Redirecting STDOUT

- Output to a file using > or 1>(overwrites)
  - **$ ls > ls_file**
  - **$ ls 1> ls_file**
- Append to a file using >>
  - **$ ls >> ls_file**
- Both methods create the file if it doesn’t exist
- Silence output by outputting to /dev/null
  - **$ ls > /dev/null**

Bash Syntax - Redirecting STDERR

- Output stderr to a file using 2> (overwrites)
  - **$ error_prone_process 2> err_file**
- Output stderr to same source as stdout using 2>&1
  - **$ ls 1>output_file 2>&1**
- Silence stderr using
  - **$ ls 2> /dev/null**

Bash Syntax - Redirecting Both STDOUT and STDERR

- To redirect all output (both stdout and stderr) use &>
  - **$ my_process &> output_file**
- To silence a process, redirect both stdout and stderr to /dev/null
  - **$ yes &>/dev/null**

Bash Syntax - Chaining Output Using Pipes

- Use the output of one process as the input of another using |
  - $ ps -ef | grep “Chrome”
  - $ cat ~/Desktop/words.txt | sort | tail -n 1

3 Advanced Bash

Bash Jobs

- So far, we have seen the shell run one process at a time
- However, it’s possible to run multiple
- Key states that a process may be in
  - Running in Foreground (everything so far)
  - Running in Background
  - Suspended / Stopped
  - Terminated

Bash Jobs - Running in Foreground

- To run in foreground, run the process normally
  - **$ /usr/bin/firefox**
- At most one process may run in the foreground

Bash Jobs - Running in Background

- To run in background, run the process with an &
  - **$ /usr/bin/firefox &**
  - [1] 27070

$$\underbrace{[1]}_{\text{job number}}\ \underbrace{27070}_{\text{PID (Process ID)}}$$

Bash Jobs - Quitting Processes

- To quit foreground process use Ctrl+C or Ctrl+\ for additional core dump
- To quit background process use kill command in foreground
  - **$ kill %1** (kill job with job number 1)
  - **$ kill 27070** (kill job with PID 27070)

Bash Jobs - Suspending Processes

- To suspend foreground process use Ctrl+Z
- To suspend background process use kill command in foreground
  - **$ kill -20 %1** (suspend job with job number 1)
  - **$ kill -20 27070** (suspend job with PID 27070)

Bash Jobs - Changing Process States

1. Check the states of all of the processes (jobs)
   - **$ jobs**

```
[1]-  Running                 yes >&/dev/null &
[2]+  Stopped                  tail -f mod.sh
```

means it's running in the background

Bash Jobs - Changing Process States

```
[1]-  Running          yes >&/dev/null &
[2]+  Stopped           tail -f mod.sh
```

2. To move tail to background
   - **$ bg %2** or **$ bg +**
3. Check the state of your jobs again
   - **$ jobs**

```
[1]-  Running          yes >&/dev/null &
[2]+  Running          tail -f mod.sh &
```

Bash Jobs - Changing Process States

```
[1]-  Running                    yes >&/dev/null &
[2]+  Running                    tail -f mod.sh &
```

3. To move yes to foreground
   - **$ fg %1** or **$ fg %-** or **$ %1** or **$ %-**
4. Finally, yes is running in the foreground

Links

- More Bash Examples
  - http://linuxconfig.org/bash-scripting-tutorial
  - https://www.cac.cornell.edu/VW/Linux/
- Advanced Scripting Next Time
  - Awk
  - Sed
  - Cron
  - Advanced SSH

Thanks!

Questions?