Linux System Programming

Linux System Programming

Introduction to Linux

- In the early 1990s, Torvalds became interested in a freeware product called Minix were written by Andrew S. Tanenbaum. Developed by Andrew S.Tanenbaum, Minix was a clone of the commercial UNIX operating system.

- Linux version 0.02, released on October 5, 1991, consisted of only the Linux kernel and three utilities:
  - bash : a command-line interface
  - update : a utility for flushing file system buffers
  - gcc : a C++ compiler

- Today used on 7-10 million computers with 1000's of programmers working to enhance it around the world.

Introduction to Linux

- GNU Project: Richard Stallman on September 27th 1983.

- The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system.

- GNU's kernel isn't finished, so GNU is used with the kernel Linux. The combination of GNU and Linux is the GNU/Linux operating system, now used by millions.

What is Linux

- A fully-networked 32/64-Bit Unix-like Operating System
  - Unix Tools Like sed, awk, and grep (explained later)
  - Compilers Like C, C++, Fortran, Smalltalk, Ada
  - Network Tools Like telnet, ftp, ping, traceroute
- Multi-user, Multitasking, Multiprocessor
- Has the X Windows GUI
- Coexists with other Operating Systems
- Runs on multiple platforms
- Includes the Source Code

Distribution

- Today there are hundreds of different distributions available popular Linux distributions include:
  - SUSE Linux
  - Red Hat Enterprise Linux
  - Debian Linux
  - CentOS
  - Ubuntu

Functionality of Linux

- **Application Platform**: An operating system provides applications with a platform where they can run, managing their access to the CPU and system memory.
- **Hardware Moderator**: The operating system also serves as a mediator between running applications and the system hardware. Most applications are not written to directly address a computer's hardware.
- **Security**: The operating system is responsible for providing a degree of security for the data it hosts.
- **Connectivity**: The operating system manages connectivity between computer systems using a variety of network media and interfaces, including infrared, Ethernet, and wireless.

Roles of Linux

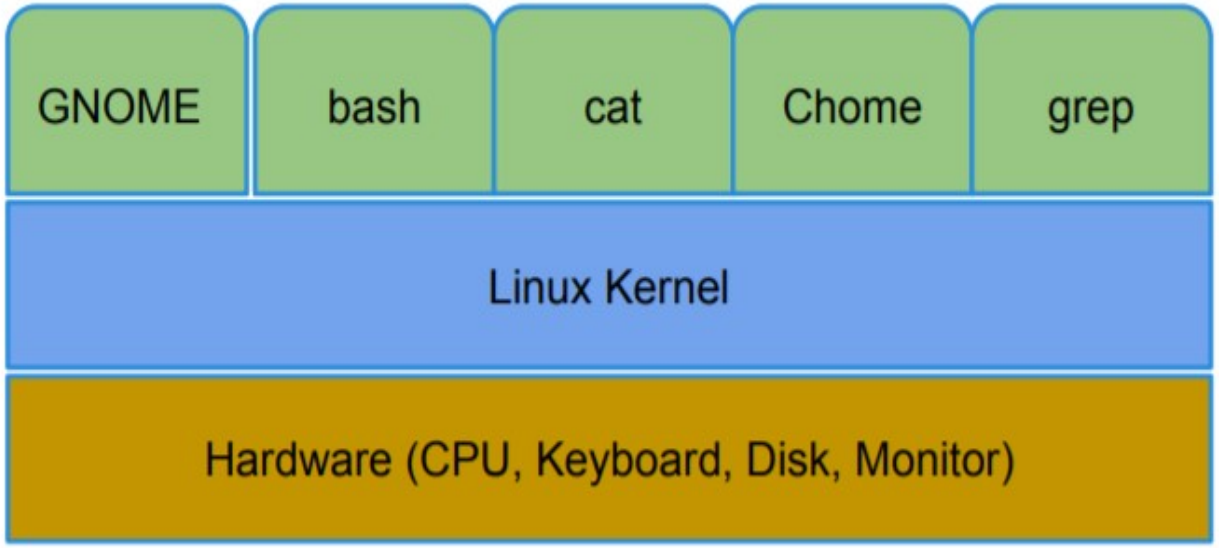
- Desktop
- Server
  - File Server
  - Print Server
  - Database Server
  - Web Server
  - E-Mail Server
- Firewall

Component of Linux

- The Linux Kernel
- Libraries
- Utilities
- User Interface

Linux Kernel

Very simple answer: it's a program that makes your hardware look and feel like an OS to other programs



Linux Kernel

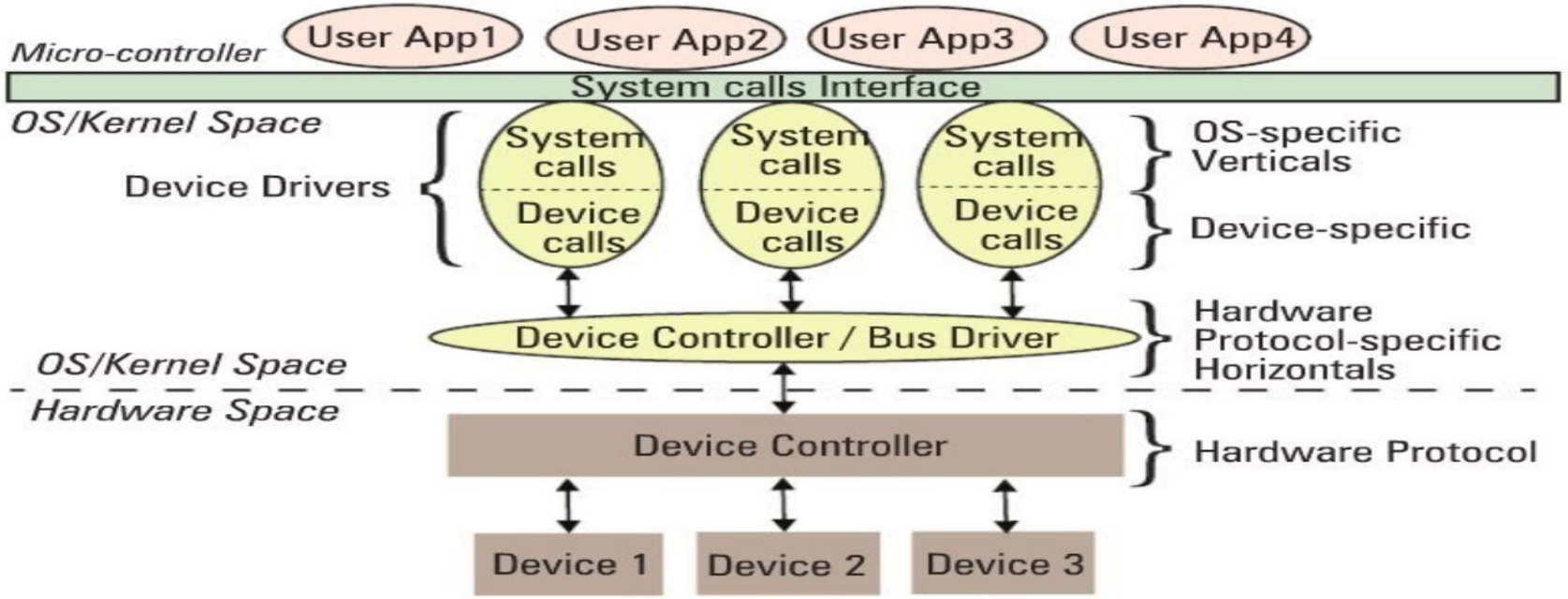
- **Libraries** are pre-written code “pieces” that application programmers use in their programs.
- **Utilities** maintaining the file system, editing text files, managing running processes, and installing new software packages.
- **User Interface** command-line interface (CLI) and a graphical user interface (GUI).

Linux Command Line Interfaces

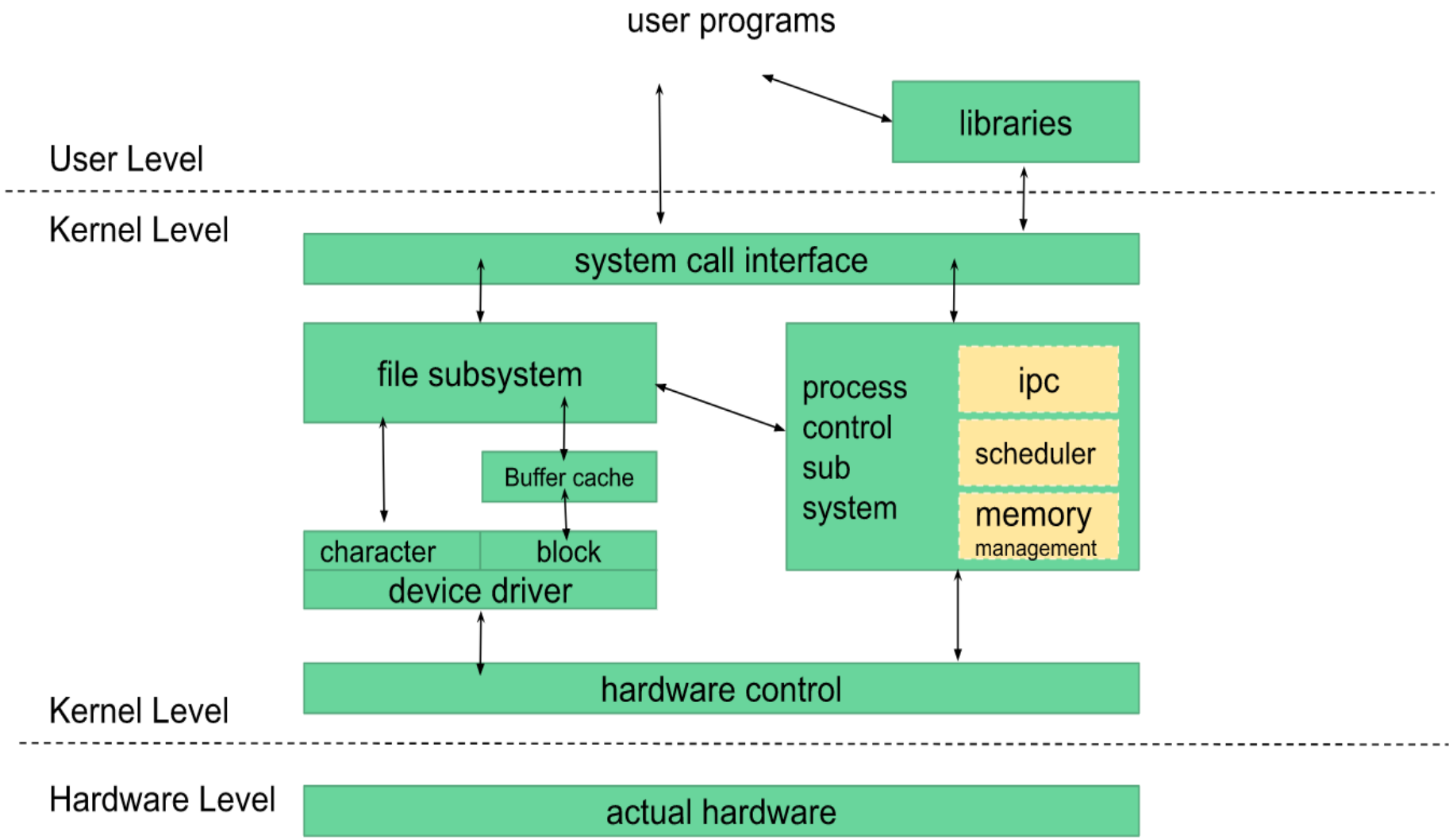
- **Linux shells**: A shell is a command interpreter that allows you to type commands from the keyboard to interact with the operating system kernel.
  - **sh (Bourne Shell):** The sh shell was the earliest shell, being developed for UNIX back in the late 1970s.
  - **bash (Bourne-Again Shell):** The bash shell is an improved version of the sh shell and is one of the most popular shells today. It's the default shell used by most Linux distributions.
  - **csh (C Shell):** The csh shell was originally developed for BSD UNIX. It uses a syntax that is very similar to C programming.
  - **tsch :** The tsch shell is an improved version of the C Shell. It is the default shell used on FreeBSD systems.
  - **zsh (Z Shell):** The Z Shell is an improved version of the bash shell.

Kernel Space and User Space

- The Concept of kernel space and user space is all about memory and access rights.
- It is a feature of modern CPU, allowing it to operate either in privileged or unprivileged mode.
- One ,may consider kernel to be privileged and user apps are restricted.

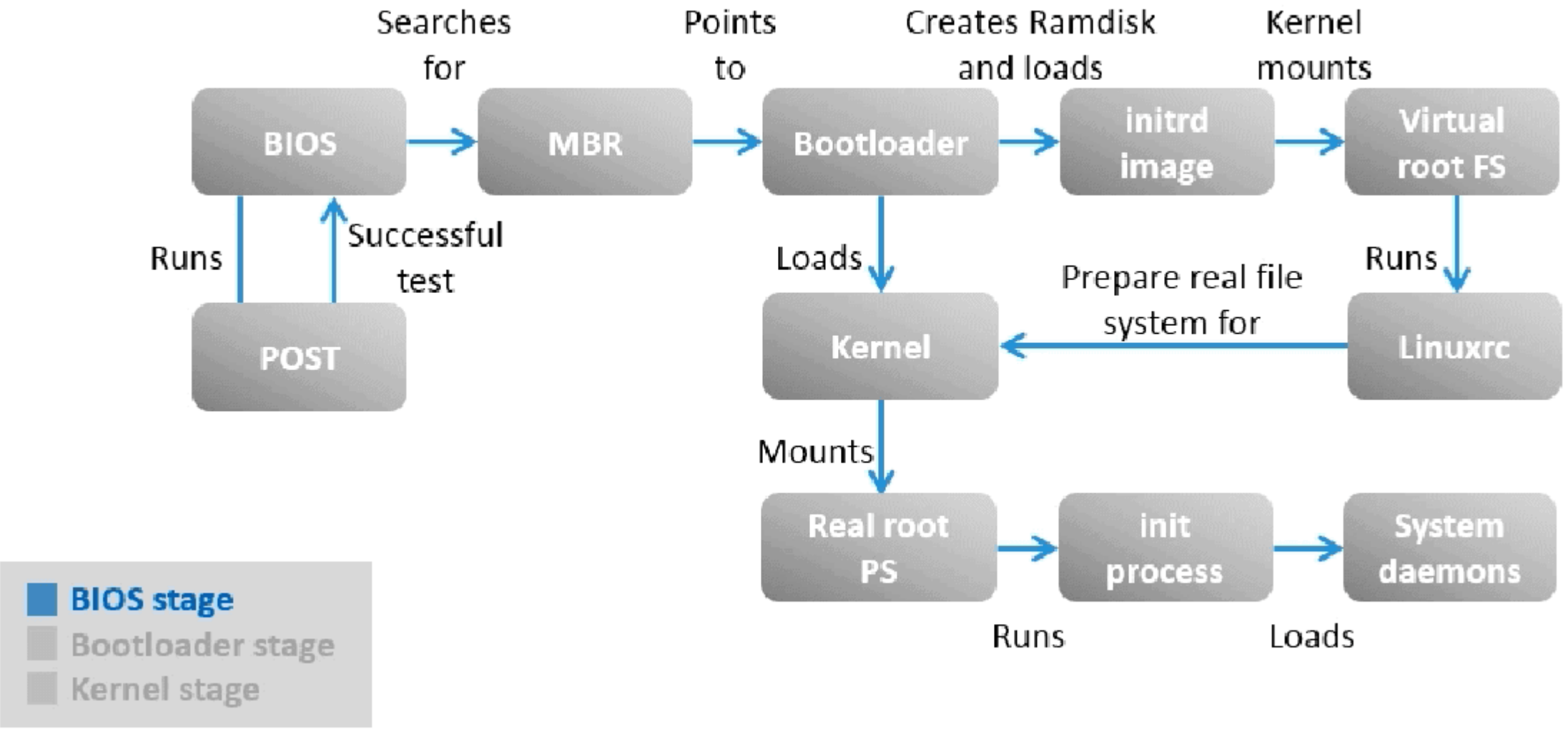


Kernel Space and User Space



Block Diagram of the System Kernel

Linux Bootup Sequence



Linux Bootup Sequence

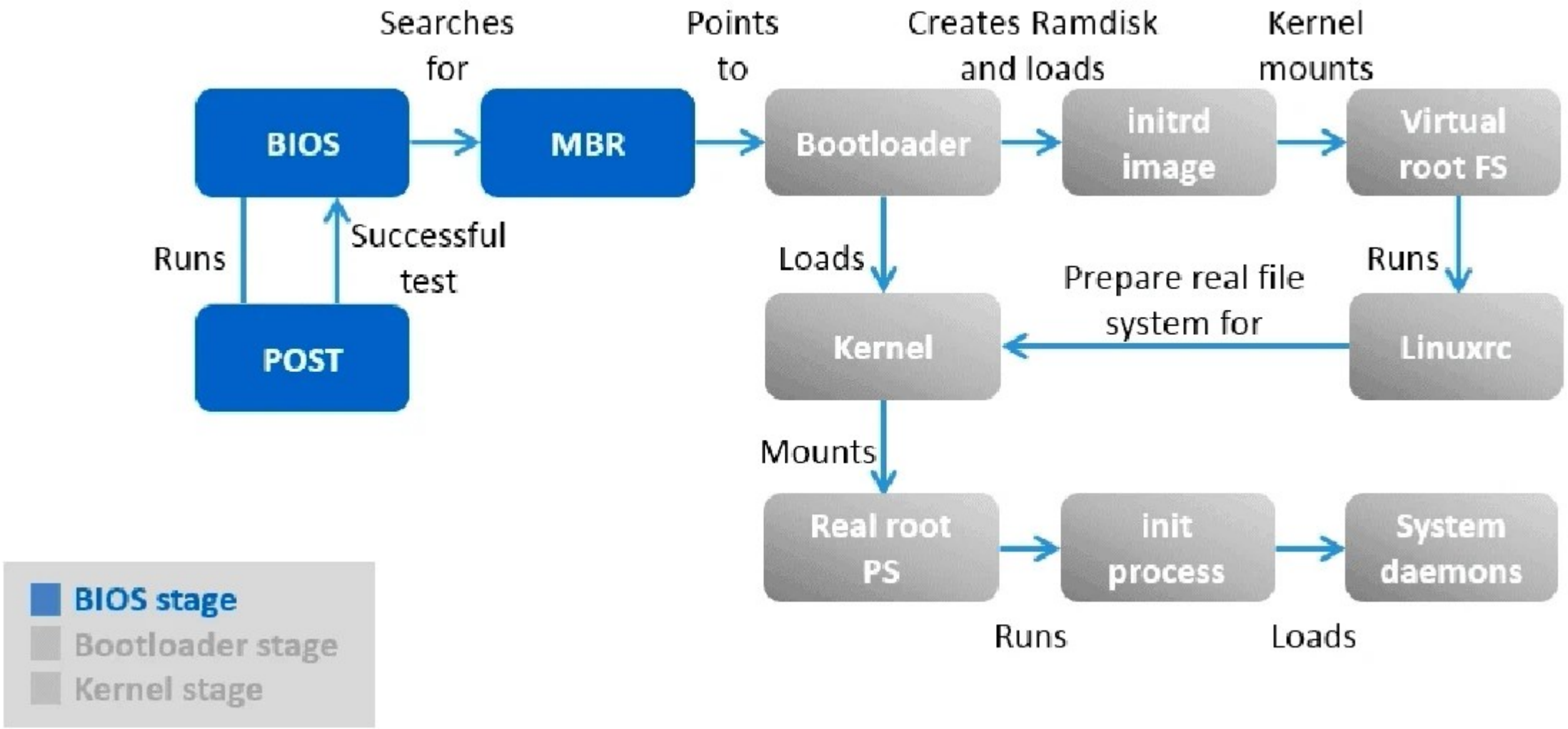
Stages of Linux Boot Process:

- BIOS Stage
- Boot Loader Stage
- Kernel Stage

Linux Bootup Sequence - BIOS Stage

- When the machine is powered on BIOS is the first one to be called to verify if the hardware is present in the machine and if it is functioning.
- This is done by performing a Power On Self Test (POST)
- In a multi-processor or multi-core system one CPU is dynamically chosen to be the bootstrap processor (BSP) that runs all of the BIOS and kernel initialization code, others are called application processors(AP)
- After a successful test, BIOS checks the MBR (Master Boot Record) in the hard disk to check if it refers to the location of the boot loader.

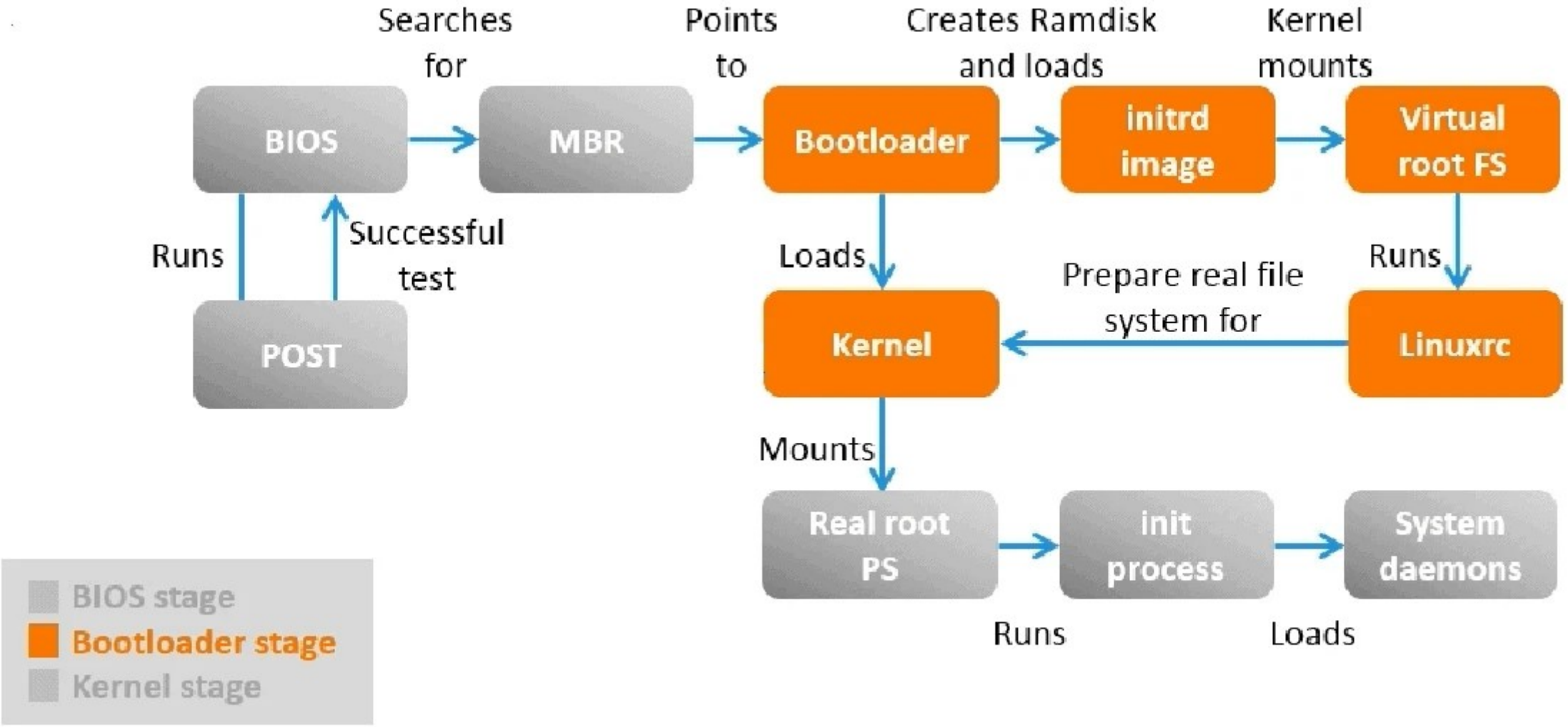
Linux Bootup Sequence – BIOS Stage



Linux Bootup Sequence - Boot loader Stage

- The boot loader(GRUB) will present the user with a list of menu entries, each of which corresponds to different operating system
- The boot loader(GRUB) will then start to boot the operating system
- When you select the the option to start Linux, it decompresses the Linux kernel in memory
- After that Linux kernel (which you selected to boot from) loads initrd (Initial ramdisk)
- The initrd is used by the Linux kernel as a temporary filesystem in the memory
- It contains tools and kernel modules which will continue the boot process including mounting a virtual root file system temporarily.
- Instead of using initrd, some Linux filesystem will also use initramfs.
- It serves the same purpose of initrd, it is just that it is a successor of initrd
- linuxrc is an executable file that is next spawn, it probes the mass storage hardware and finds a suitable kernel module to drive the mass storage hardware
- This is required to prepare the real root filesystem to be mounted by the Linux kernel

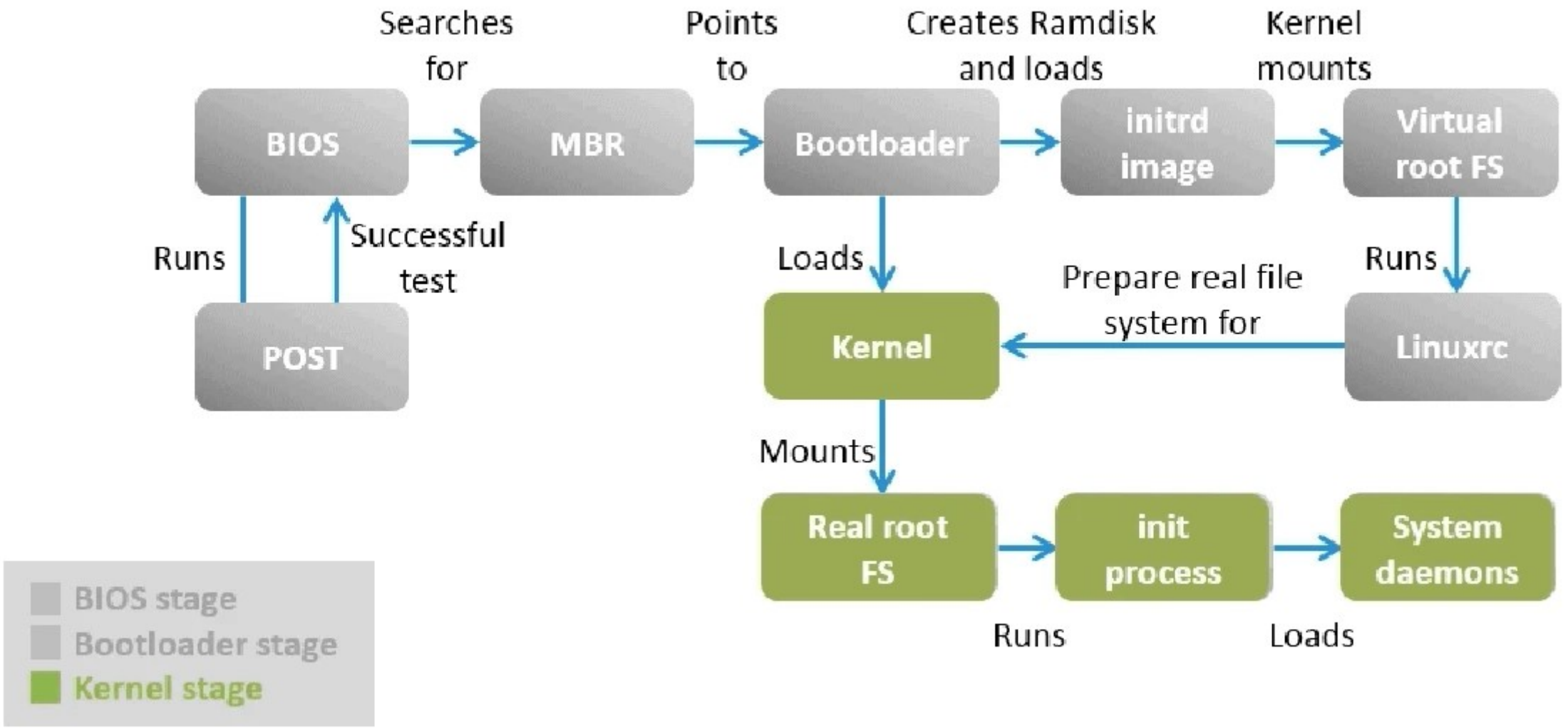
Linux Bootup Sequence – Boot Loader Stage



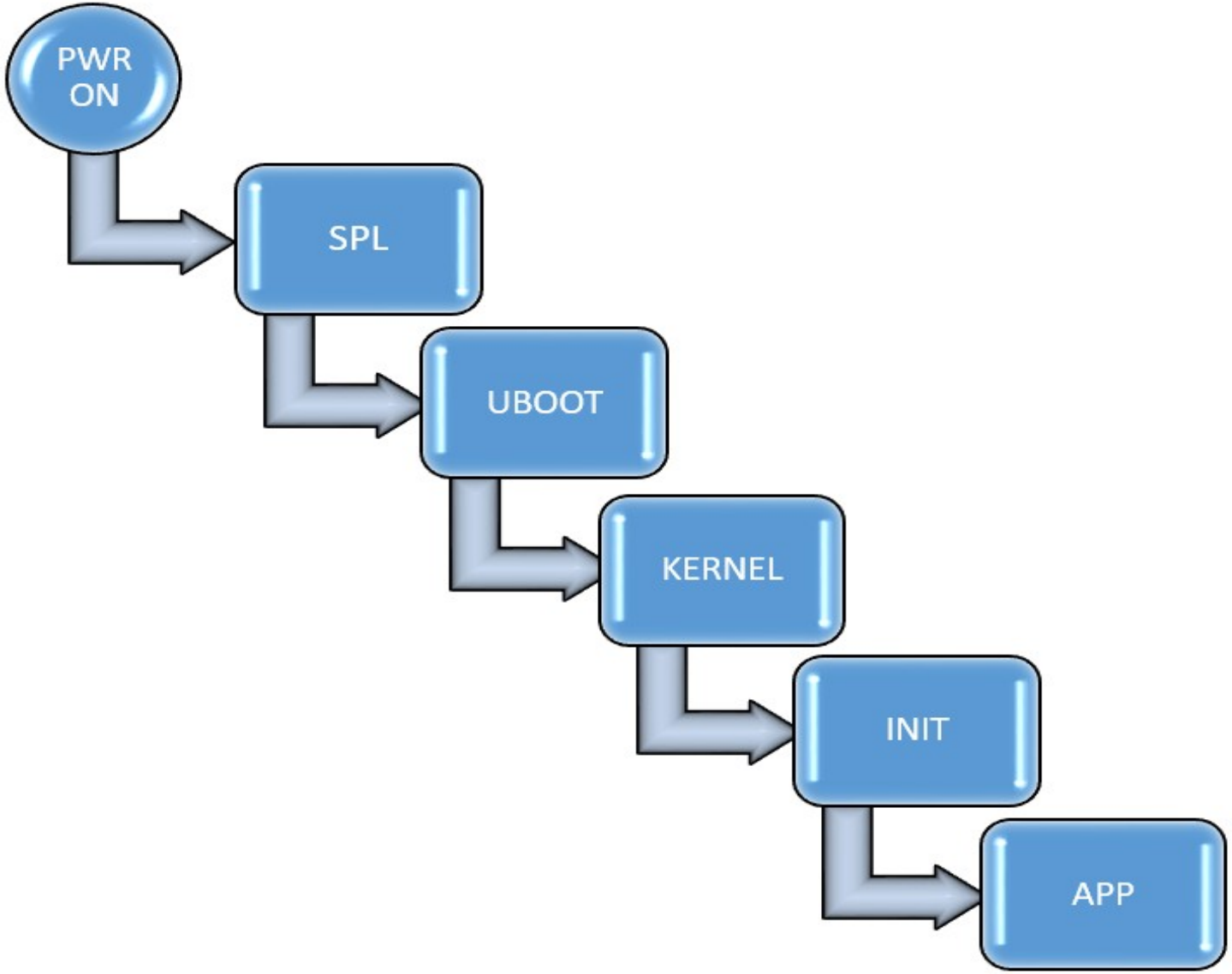
Linux Bootup Sequence - Kernel Stage

- In the kernel stage of the Linux boot sequence, the Linux kernel based on the result of linuxrc can then mount the real root file system
- The real root file system in Linux is referenced as "/" and it is where all other sub directory and files visible when Linux is running exist
- The kernel will then spawn the init process, this process always has the process identifier (PID) as "1" because it is the first background process or daemon started by the kernel upon boot
- All other background daemons are spawned from the init process
- So the init process will load other system daemons depending upon the configuration of different runlevel

Linux Bootup Sequence – Kernel Stage



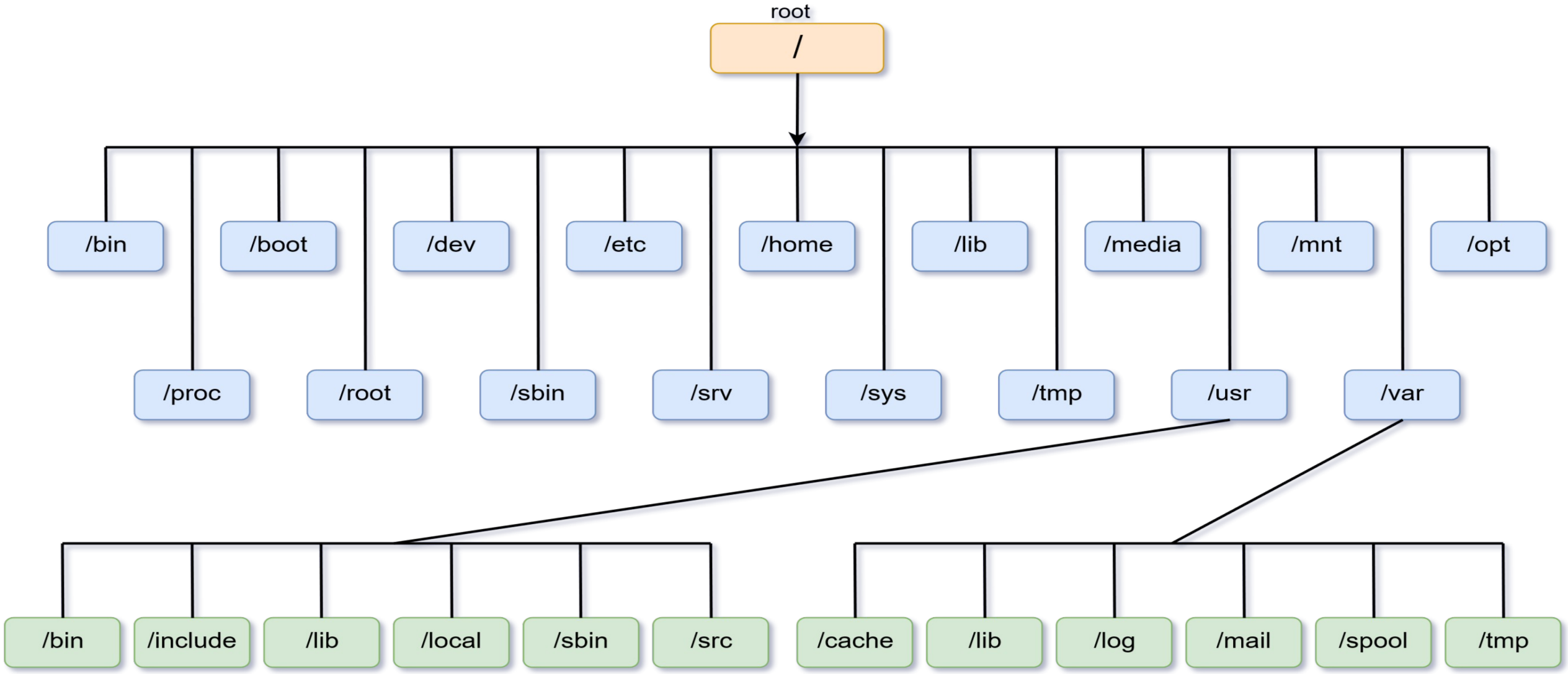
Linux Bootup Sequence – ARM UBOOT



Linux Filesystem Hierarchy

- The Linux File Hierarchy Structure or the Filesystem Hierarchy Standard (FHS) defines the directory structure and directory contents in Unix-like operating systems. It is maintained by the Linux Foundation.
  - In the FHS, all files and directories appear under the root directory /, even if they are stored on different physical or virtual devices.
  - Some of these directories only exist on a particular system if certain subsystems, such as the X Window System, are installed.
  - Most of these directories exist in all UNIX operating systems and are generally used in much the same way; however, the descriptions here are those used specifically for the FHS and are not considered authoritative for platforms other than Linux.

Linux Filesystem Hierarchy



Linux Filesystem Hierarchy

- **/ (root)**
  - Primary hierarchy root and is the first directory and the root directory of the entire file system hierarchy.
  - It contains all other directories i.e. the sub directories.
  - Only the root user has the permissions to write here.
  - This is not the home directory for the root user
- **/bin (user binaries)**
  - It contains binary executables.
  - Binary executables of common linux commands used by all users in single-user mode are located in this directory.
  - Some files present in this directory are: ls, cp, grep, ping, cat, etc.

Linux Filesystem Hierarchy

- **/boot (boot loader files)**
  - It contains boot loader files.
  - kernel, initrd, grub, and other files and directories are located in this directory.
  - e.g. vmlinux-2.7.31.25-generic, initrd.img-2.3.32.24-generic, etc.
- **/dev (device files)**
  - It contains the essential files related to the devices attached to the system.
  - This includes terminal devices, USB, network devices and any other devices that are attached to the system.
  - e.g. /dev/usbmon0 , /dev/tty1 , /dev/null , etc.

Linux Filesystem Hierarchy

- **/etc (configuration files)**
- etc stands for ‘edit to config’
- This directory contains the configuration files that are required by the installed programs.
- The files are host-specific and system-wide configurations needed for the proper functioning of the system.
- This directory also contains shell scripts for system startup and system shutdown that are used to start or stop individual programs.
- The files in this directory should not be edited without proper knowledge of system configuration as improper configuration could brick the system.
- e.g. /etc/passwd , /etc/shadow , /etc/group , /etc/resolv.conf , etc.

Linux Filesystem Hierarchy

- **/home (home directories)**
  - This directory contains user's home directories, containing saved files and personal settings.
  - Each user will have an separate directory with their username under this directory except the root user because every time a new user is created, a directory is created in the name of the user within the home directory.
  - e.g. /home/user , /home/sage , /home/guest , etc.
- **/lib (system libraries)**
  - This directory contains libraries that are essential for the binaries in /bin and /sbin
  - Library filenames are either ld* or lib*.so.*
  - e.g. ld-2.11.1.so , etc.
- **/media (removable media devices)**
  - Temporary mount directory for removable media such as CD-ROM.
  - e.g. /media/cdrom for CD-ROM ; /media/floppy for floppy drives ; /media/cdrecorder for CD writer ; etc.

Linux Filesystem Hierarchy

- **/mnt (mount directory)**
  - Temporary mount directory where system administrator can mount file systems.
- **/opt (optional application software packages)**
  - This directory contains add-on applications from individual vendors.
- **/proc (process information)**
  - This is a virtual filesystem providing process and kernel information. This files in this directory are automatically generated, populated and deleted by the system. In Linux, corresponds to a procfs mount.
  - This directory contains information about the processes running in the system.
  - This directory also contains text information about running processes. e.g. /proc/uptime
  - e.g. /proc/{pid} directory contains information about the process with that particular pid that will be mentioned within the brackets.

Linux Filesystem Hierarchy

- **/root (root directory)**
  - This is the home directory for the root user.
- **/sbin (system binaries)**
  - This directory contains essential system binaries.
  - The linux commands that are located in this directory are used by system administrator, for system maintenance and configuration purpose.

  e.g. fsck , reboot , fdisk , ifconfig , init , etc.

Linux Filesystem Hierarchy

- **/root (root directory)**
  - This is the home directory for the root user.
- **/sbin (system binaries)**
  - This directory contains essential system binaries.
  - The linux commands that are located in this directory are used by system administrator, for system maintenance and configuration purpose.

  e.g. fsck , reboot , fdisk , ifconfig , init , etc.
- **/srv (service data)**
  - This directory contains site-specific data served by the system, such as data and scripts for web servers, data offered by FTP servers, and repositories for version control systems i.e. server specific services related data.

  e.g. /srv/cvs contains CVS related data , etc.

Linux Filesystem Hierarchy

- **/sys (system)**
  - This directory contains information about devices, drivers, and some kernel features.
- **/tmp (temporary files)**
  - This directory contains temporary files created by system and the users that will be rebooted when the system is rebooted.
- **/usr (user programs)**
  - This directory contains read-only user data like binaries, libraries, documentation and source-code for second level programs like user utilities and applications.
  - /usr/bin → contains binary files for user programs. If you can't find a user binary under /bin, then we should look under /usr/bin.
  - /usr/include → contains standard include files.

Linux Filesystem Hierarchy

- /usr/lib → contains libraries for the binaries in /usr/bin and /usr/sbin

- /usr/local → tertiary hierarchy for local data. contains users programs that you install from source. e.g., when you install apache, it goes under /usr/local/apache2

- /usr/sbin → /usr/sbin contains binary files for system administrators. If you can't find a system binary under /sbin, then you should look under /usr/sbin. It also contains non-essential system binaries. e.g. daemons for network-services.

❑ **/var (variable files)**

- This directory contains files whose content is expected to continually change during normal operation of the system—such as logs, spool files, and temporary e-mail file.
- /var/log → contains system log files.

Linux Filesystem Hierarchy

Mounting debugfs,proc,sysfs:

```
# sudo mkdir /tmp/{proc,sys,debug}
# sudo mount -t proc nondev /tmp/proc/
# sudo mount -t sysfs nondev /tmp/sys/
# sudo mount -t debugfs nondev /tmp/debug/
```

Linux Command – Basic Commands - ls

ls is a Linux shell command that lists directory contents of files and directories.

- **ls -t** : To sorts the file by modification time, showing the last edited file first.
- **ls -1** : To Display One File Per Line
- **ls -l** : To show long listing information about the file/directory.
- **ls -lh**: To display file size in easy to read format. i.e i.e M for MB, K for KB, G for GB.
- **ls -ld**: To display the information about directory.

Linux Command – Basic Commands - ls

ls is a Linux shell command that lists directory contents of files and directories.

- **ls -r :** To sort Files in Reverse Order.
- **ls -a :** To show all the hidden files in the directory.
- **ls -A :** To show the hidden files, but not the '.' and '..'
- **ls -R :** To show all the files recursively.
- **ls -i :** Tod display inode number.

Linux Command – Basic Commands - ls

ls is a Linux shell command that lists directory contents of files and directories.

- **ls -q :** To print question mark instead of the non graphics control characters.

- **ls -n :** To Display File UID and GID.

- **ls -F :** To Classify files with different special character.
  - / – directory.
  - nothing – normal file.
  - @ – link file.
  - * – Executable file

Linux Command – Basic Commands - dir

dir command in Linux is used to list the contents of a directory.

Syntax:

dir [OPTION] [FILE]

- **dir -a :** To display all the hidden files.

- **dir -A :** To display all files excluding . and ..

- **dir -l :** To display Author information.

Linux Command – Basic Commands - dir

- **dir -F :** To Classify files with different special character.
  - / – directory.
  - nothing – normal file.
  - @ – link file.
  - * – Executable file

Linux Command – Basic Commands - cd

cd command in linux known as change directory command. It is used to change current working directory.

Syntax:

```
$ cd [directory]
```

Linux Command – Basic Commands - echo

- You can use echo command with various options.

```
echo [ OPTION(S) ] [ STRING ]
```

Options:

Option	Description
-n	Omit newline from the output.
-e	Enable the function of backslash(/) character.
-E	Disable the function of backslash(/) character.
–version	Display the version information
–help	Display help messages related to the uses of this command

Linux Command – Basic Commands - echo

- When you use ‘echo’ command without any option then a newline is added by default.
- ‘-n’ option is used to print any text without new line and
- ‘-e’ option is used to remove backslash characters from the output.

```
#!/bin/bash
echo "Printing text with newline"
echo -n "Printing text without newline"
echo -e "\nRemoving \t backslash \t characters\n"
```

Linux Command – Basic Commands - echo

- ‘echo’ command is used with ‘-E’ option in the following script. This option disables the function of backslash(/). The new line(\n) used in the text will not work for the following command.‘echo’ command is used with ‘-E’ option in the following script. This option disables the function of backslash(/). The new line(\n) used in the text will not work for the following command.

```
$ echo -E "I\nlike\nbash\programming"
```

Linux Command – File Commands - Head

- The head command, as the name implies, print the top N number of data of the given input. By default, it prints the first 10 lines of the specified files. If more than one file name is provided then data from each file is preceded by its file name.

Syntax:

head [OPTION]... [FILE]...

Short Options	Long Options
-n	--lines
-c	--bytes
-q	--quiet
-v	--verbose

Linux Command – File Commands - Tail

- The tail command, as the name implies, print the top N number of data of the given input. By default, it prints the last 10 lines of the specified files. If more than one file name is provided then data from each file is preceded by its file name.

Syntax:

tail [OPTION]... [FILE]...

Short Options	Long Options
-n	--lines
-c	--bytes
-q	--quiet
-v	--verbose

Linux Command – File Commands - cat

Cat(concatenate) command reads data from the file and gives their content as output. It helps us to create, view, concatenate files.

- **cat filename :** view single file.
- **cat file1 file2 :** to view multiple files.
- **cat -n filename :** to view of file with line number.
- **cat > newfile :** create new file.
- **cat oldfile > new file :** copy old file to new file.

Linux Command – File Commands - cat

- **cat -s geeks.txt :** suppress repeated empty lines in output.
- **cat file1 >> file2 :** append data to file.
- **cat -E filename :** Highlight end of line.
- **cat file1 file2 > file3 :** merge files.

Linux Command – File Commands - wc

wc stands for word count. As the name implies, it is mainly used for counting purpose.

It is used to find out number of lines, word count, byte and characters count in the files specified in the file arguments.

By default it displays four-columnar output.

First column shows number of lines present in a file specified, second column shows number of words present in the file, third column shows number of characters present in file and fourth column itself is the file name which are given as argument.

Syntax:

```
wc [OPTION]... [FILE]...
```

Linux Command – File Commands - wc

- **wc -l :** prints the number of words present in a file.
- **wc -w :** prints the number of words present in a file.
- **wc -c :** printfs the number of bytes present in a file.
- **wc -L :** prints out length of longest line in a file.

Linux Command – File Commands - Sort

SORT command is used to sort a file, arranging the records in a particular order. By default, the sort command sorts file assuming the contents are ASCII. Using options in the sort command can also be used to sort numerically.

- SORT command sorts the contents of a text file, line by line.
- sort is a standard command-line program that prints the lines of its input or concatenation of all files listed in its argument list in sorted order.
- The sort command is a command-line utility for sorting lines of text files. It supports sorting alphabetically, in reverse order, by number, by month, and can also remove duplicates.
- The sort command can also sort by items not at the beginning of the line, ignore case sensitivity, and return whether a file is sorted or not. Sorting is done based on one or more sort keys extracted from each line of input.
- By default, the entire input is taken as the sort key. Blank space is the default field separator.

Linux Command – File Commands - Sort

- **-o Option:** Writes output to filename.
- **-r Option:** Sorting in Reverse Order.
- **-n Option:** Sort file numerically.
- **-c option :** check if file is sorted.
- **-u option:** sort and remove duplicates.
- **-M option:** sort by month.

Linux Command – File Commands - touch

The ***touch*** command is a standard command used in UNIX/Linux operating system which is used to create, change and modify timestamps of a file.

Linux Command – File Commands - rm

rm stands for remove here. rm command is used to remove objects such as files, directories, symbolic links and so on from the file system like UNIX. To be more precise, rm removes references to objects from the filesystem, where those objects might have had multiple references (for example, a file with two different names). By default, it does not remove directories.

This command normally works silently and you should be very careful while running rm command because once you delete the files then you are not able to recover the contents of files and directories.

Syntax:

```
rm [OPTION]... FILE...
```

Linux Command – File Commands - redirection

- In Linux (and in other OSs also). It's possible to send output to file or to read input from a file

  For e.g.

  $ *ls* command gives output to screen;

- to send output to file of ls command give command

  ***ls* > filename**

  It means put output of ***ls*** command to filename.

Linux Command – File Commands - redirection

There are three main redirection symbols: >,>>,<

> Redirector Symbol
Syntax:
Linux-command > filename

To output Linux-commands result to file. Note that if the file already exist, it will be overwritten else a new file will be created.

For e.g.

To send output of ***ls*** command give
$ *ls* > myfiles

if '**myfiles**' exist in your current directory it will be overwritten without any warning.

Linux Command – File Commands - redirection

>> Redirector Symbol

Syntax:

Linux-command >> filename

To output Linux-commands result to the END of the file. if file exist: it will be opened new information/data will be written to the **END** of the file, without losing previous information/data, if file does not exist, a new file is created.

For e.g.

To send output of date command to already exist file give command

$ *date* >> myfiles

Linux Command – File Commands - redirection

< Redirector Symbol

Syntax:

Linux-command < filename

To provide input to Linux-command from the file instead of standart input (key-board).

For e.g. To take input for cat command give

$ *cat* < myfiles

Linux Command – Process management - ps

Linux provides us a utility called ps for viewing information related with the processes on a system which stands as abbreviation for “Process Status”. ps command is used to list the currently running processes and their PIDs along with some other information depends on different options. It reads the process information from the virtual files in /proc file-system. /proc contains virtual files, this is the reason it’s referred as a virtual file system.

ps provides numerous options for manipulating the output according to our need.

Syntax

```
ps [options]
```

Linux Command – Process management - ps

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output
       specifiers (header "STAT" or "S") will display to describe the state of
       a process.
       D    Uninterruptible sleep (usually IO)
       R    Running or runnable (on run queue)
       S    Interruptible sleep (waiting for an event to complete)
       T    Stopped, either by a job control signal or because it is being
            traced.
       W    paging (not valid since the 2.6.xx kernel)
       X    dead (should never be seen)
       Z    Defunct ("zombie") process, terminated but not reaped by its
            parent.

       For BSD formats and when the stat keyword is used, additional
       characters may be displayed:
       <    high-priority (not nice to other users)
       N    low-priority (nice to other users)
       L    has pages locked into memory (for real-time and custom IO)
       s    is a session leader
       l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
       +    is in the foreground process group
```

Linux Command – Process management - top

top command is used to show the Linux processes. It provides a dynamic real-time view of the running system. Usually, this command shows the summary information of the system and the list of processes or threads which are currently managed by the Linux Kernel.

As soon as you will run this command it will open an interactive command mode where the top half portion will contain the statistics of processes and resource usage. And Lower half contains a list of the currently running processes. Pressing q will simply exit the command mode.

Linux Command – Process management - kill

kill command in Linux (located in /bin/kill), is a built-in command which is used to terminate processes manually. kill command sends a signal to a process which terminates the process. If the user doesn't specify any signal which is to be sent along with kill command then default TERM signal is sent that terminates the process.

- **Kill –l :** Display all the available signals.

Linux Command – Process management - killall

Killall command would kill the process by name.

Linux Command – Process management - pidof

pidof command is used to find out the process IDs of a specific running program. It is basically an identification number that is automatically assigned to each process when it is created.

Syntax:

pidof [options] program1 program2 ... programN

Linux Command – disk management- df

The **df** command (short for disk free), is used to display information related to file systems about total space and available space.

Syntax :

df [OPTION]... [FILE]...

Linux Command – disk management- du

du command, short for disk usage, is used to estimate file space usage.

The du command can be used to track the files and directories which are consuming excessive amount of space on hard disk drive.

Syntax :

du [OPTION]... [FILE]...

Linux Command – disk management- dd

dd is a command-line utility for Unix and Unix-like operating systems whose primary purpose is to convert and copy files.

On Unix, device drivers for hardware (such as hard disk drives) and special device files (such as /dev/zero and /dev/random) appear in the file system just like normal files.

dd can also read and/or write from/to these files, provided that function is implemented in their respective drivers

As a result, dd can be used for tasks such as backing up the boot sector of a hard drive, and obtaining a fixed amount of random data.

The dd program can also perform conversions on the data as it is copied, including byte order swapping and conversion to and from the ASCII and EBCDIC text encodings.

 # dd if = /dev/sda of = /dev/sdb

Linux Command – disk management- free

free command which displays the total amount of free space available along with the amount of memory used and swap memory in the system, and also the buffers used by the kernel.

This is pretty much what free command does for you.

Syntax:

$free [OPTION]

Linux Command – network command - ifconfig

ifconfig(interface configuration) command is used to configure the kernel-resident network interfaces. It is used at the boot time to set up the interfaces as necessary. After that, it is usually used when needed during debugging or when you need system tuning. Also, this command is used to assign the IP address and netmask to an interface or to enable or disable a given interface.

Syntax:

ifconfig [...OPTIONS] [INTERFACE]

Linux Command – network command - route

route command in Linux is used when you want to work with the IP/kernel routing table. It is mainly used to set up static routes to specific hosts or networks via an interface. It is used for showing or update the IP/kernel routing table.

Linux Command – network command - route

route command in Linux is used when you want to work with the IP/kernel routing table. It is mainly used to set up static routes to specific hosts or networks via an interface. It is used for showing or update the IP/kernel routing table.

Linux Command – network command - ip

ip command in Linux is present in the net-tools which is used for performing several network administration tasks. IP stands for Internet Protocol. This command is used to show or manipulate routing, devices, and tunnels. It is similar to ifconfig command but it is much more powerful with more functions and facilities attached to it. ifconfig is one of the deprecated commands in the net-tools of Linux that has not been maintained for many years. ip command is used to perform several tasks like assigning an address to a network interface or configuring network interface parameters.

It can perform several other tasks like configuring and modifying the default and static routing, setting up tunnel over IP, listing IP addresses and property information, modifying the status of the interface, assigning, deleting and setting up IP addresses and routes.

Syntax:

ip [OPTIONS] OBJECT { COMMAND | help }

Linux Command – network command - ip

 # ip address show

#ip link (get link layer information)

 # ip –s link

#ip route

 # ip a add 192.168.214.1/24 dev enp3s0

 # ip a del 192.168.214.1/24 dev enp3s0

 # ip link set enp3s0 up

 # ip monitor

#ip neighbour

Linux Command – network command - ping

PING (Packet Internet Groper) command is used to check the network connectivity between host and server/host. This command takes as input the IP address or the URL and sends a data packet to the specified address with the message “PING” and get a response from the server/host this time is recorded which is called latency. Fast ping low latency means faster connection. Ping uses ICMP(Internet Control Message Protocol) to send an ICMP echo message to the specified host if that host is available then it sends ICMP reply message. Ping is generally measured in millisecond every modern operating system has this ping pre-installed.

Linux Command – network command - nslookup

Nslookup (stands for “Name Server Lookup”) is a useful command for getting information from DNS server. It is a network administration tool for querying the Domain Name System (DNS) to obtain domain name or IP address mapping or any other specific DNS record. It is also used to troubleshoot DNS related problems.

Syntax:

nslookup [option]

Linux Command – network command – traceroot

traceroute command in Linux prints the route that a packet takes to reach the host.

 # traceroute google.com

Linux Command – network command – arp

arp command manipulates the System's ARP cache. It also allows a complete dump of the ARP cache. ARP stands for Address Resolution Protocol. The primary function of this protocol is to resolve the IP address of a system to its mac address, and hence it works between level 2(Data link layer) and level 3(Network layer).

Syntax:

arp [-v] [-i if] [-H type] -a [hostname]

Linux Command – network command – arping

The **arping** command is a command-line tool used in the Linux operating system to discover and detect hosts on the network. The arping uses the ARP protocol in order to ping hosts with the Layer 2 ARP requests. The ARP protocol is used to resolve IP addresses to the MAC addresses in order to communicate inside a LAN. The arping will try to resolve all network addresses into the MAC addresses and the hosts will provide the ARP reply for the ARP request with their MAC addresses. The arping name consists of arp+ing where the arp refers to the ARP protocol and ing or ping refers to the ping command.

Linux Command – remote connection - ssh

ssh stands for “Secure Shell”. It is a protocol used to securely connect to a remote server/system. ssh is secure in the sense that it transfers the data in encrypted form between the host and the client. It transfers inputs from the client to the host and relays back the output. ssh runs at TCP/IP port 22.

Syntax:

ssh user_name@host(IP/Domain_name)

Linux Command – remote connection - scp

scp (secure copy) command in Linux system is used to copy file(s) between servers in a secure way. The SCP command or secure copy allows secure transferring of files in between the local host and the remote host or between two remote hosts. It uses the same authentication and security as it is used in the Secure Shell (SSH) protocol. SCP is known for its simplicity, security and pre-installed availability.

Syntax:

scp [-346BCpqrTv] [-c cipher] [-F ssh_config] [-i identity_file] [-l limit] [-o ssh_option] [-P port] [-S program] [[user@]host1:]file1 … [[user@]host2:]file2

Linux Command – remote connection - rsync

rsync or remote synchronization is a software utility for Unix-Like systems that efficiently sync files and directories between two hosts or machines. One of them being the source or the local-host from which the files will be synced, the other one being the remote-host, on which synchronization will take place. There are basically two ways in which rsync can copy/sync data:

Copying/syncing to/from another host over any remote shell like ssh, rsh.

Copying/Syncing through rsync daemon using TCP.

Rsync is famous for its delta-transfer algorithm, in which it copies only the differences between the source files present in the local-host and the existing files in the destination or the remote host.

 # rsync local-file user@remote-host:remote-file

Linux Command – Command pipes

A pipe is a way to connect the output of one program to the input of another program without any temporary file.

Definition

"*A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands Multiple commandsfrom same command line.*"

Syntax:

command1 | command2

Linux Command – Command pipes

Command using Pipes	Meaning or Use of Pipes
$ ***ls*** **|** ***more***	Output of ***ls*** command is given as input to the command ***more*** So output is printed one screen full page at a time.
$ ***who*** **|** ***sort***	Output of ***who*** command is given as input to sort command So it will print sorted list of users
$ ***who*** **|** ***sort*** **> user_list**	Same as above except output of sort is send to (redirected) the file named user_list
$ ***who*** **|** ***wc*** **-l**	***who*** command provides the input of ***wc*** command So it will count the users logged in.
$ ***ls*** **-l | *wc* -l**	***ls*** command provides the input of ***wc*** command So it will count files in current directory.
$ ***who*** **|** ***grep*** **raju**	Output of ***who*** command is given as input to ***grep*** command So it will print if particular user is logged in. Otherwise nothing is printed

Linux Configuration Files - /etc/hosts

The /etc/hosts file contains information regarding the known hosts on the network. For each host, a single line should be present with the following information:

internet_address official_host_name aliases

Items are separated by any number of spaces or tabs, or both; however, spaces or tabs aren't allowed before the IP address. A # indicates the beginning of a comment — any characters after a #, up to the end of the line, aren't interpreted by routines that search the file.

If you're using a name server, this file provides a backup when the server isn't running. In this case, you should include only a few addresses in this file. These include addresses for the local interfaces that ifconfig needs at boot time and a few machines on the local network.

Linux Configuration Files - /etc/hosts

Network addresses are specified in the conventional “.” (dot) notation using the inet_addr() routine from the internet address manipulation functions. Hostnames may contain any printable character other than a field delimiter, newline, or comment character.

Linux Configuration Files - /etc/resolv.conf

The /etc/resolv.conf file defines how the system uses DNS to resolve host names and IP addresses. This file usually contains a line specifying the search domains and up to three lines that specify the IP addresses of DNS server. The following entries from /etc/resolv.conf configure two search domains and three DNS servers:

```
search us.mydomain.com mydomain.com
nameserver 192.168.154.3
nameserver 192.168.154.4
nameserver 10.216.106.3
```

If your system obtains its IP address from a DHCP server, it is usual for the system to configure the contents of this file with information also obtained using DHCP.

Interrupt

- Interrupts, asynchronous, from external devices, not related to code running
  - Maskable interrupts
  - Nonmaskable interrupts (NMI): hardware error

- Exceptions, synchronous, raised by CPU
  - *Processor-detected* exceptions:
    - *Faults* — correctable; offending instruction is *retried*
    - *Traps* — often for debugging; instruction is *not* retried
    - *Aborts* — major error (hardware failure), EIP wrong
  - *Programmed* exceptions:
    - Requests for kernel intervention (software intr/syscalls)

Interrupt

Faults

- Instruction would be illegal to execute
- Examples:
  - Writing to a memory segment marked ‘read-only’
  - Reading from an unavailable memory segment (on disk) ➔ page fault
  - Executing a ‘privileged’ instruction

- Detected *before* incrementing the IP
- The causes of ‘faults’ can often be ‘fixed’
- If a ‘problem’ can be remedied, then the CPU can just resume its execution-cycle

Interrupt

Traps

- A CPU might have been programmed to automatically switch control to a ‘debugger’ program after it has executed an instruction

- That type of situation is known as a ‘trap’

- It is activated *after* incrementing the IP

Interrupt

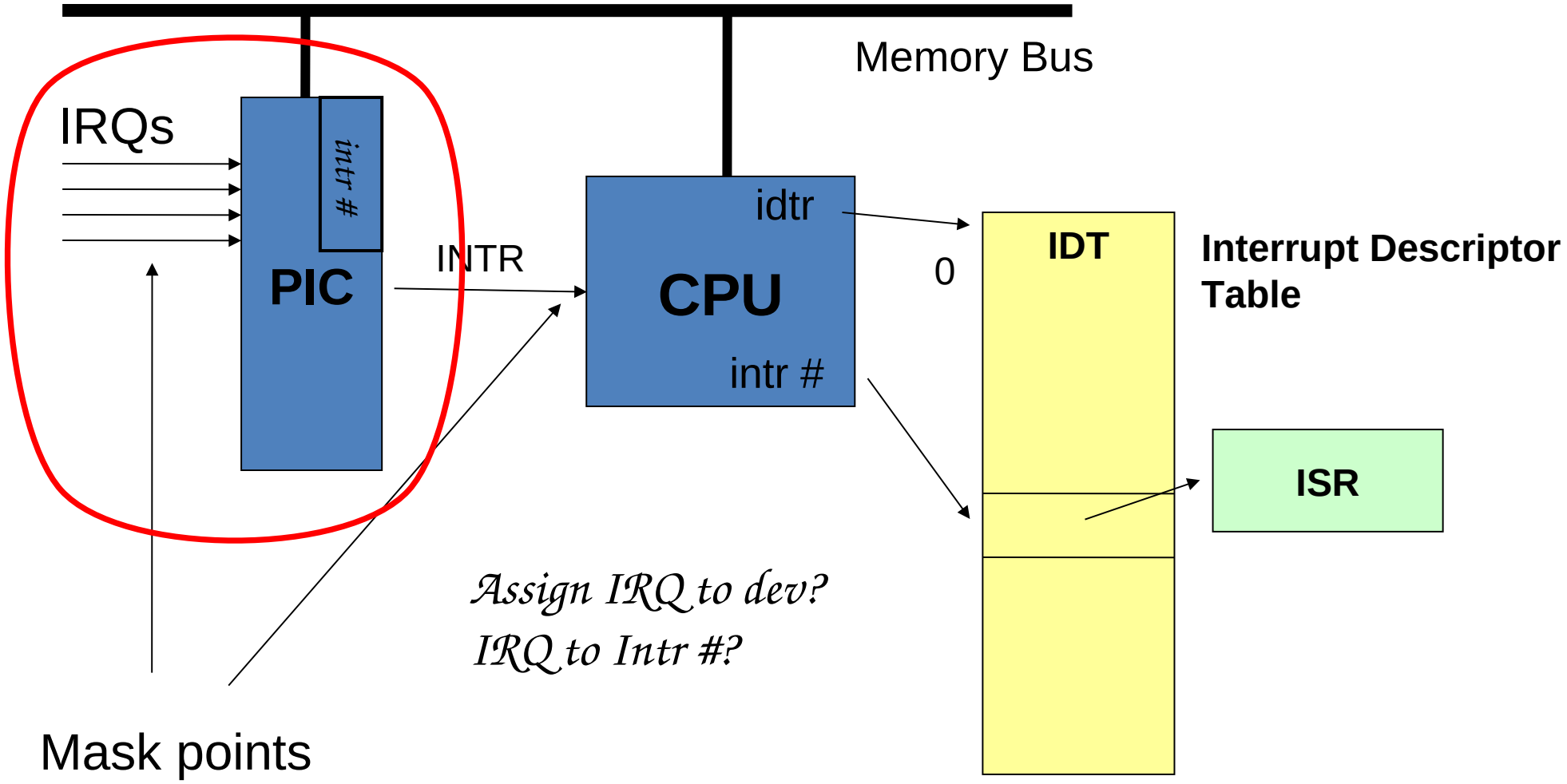
Handling Exceptions

- Most error exceptions — divide by zero, invalid operation, illegal memory reference, etc. — translate directly into signals
- This isn't a coincidence. . .
- The kernel's job is fairly simple: send the appropriate signal to the current process
  - **force_sig(sig_number, current);**
- That will probably kill the process, but that's not the concern of the exception handler

- One important exception: page fault

- An exception can (infrequently) happen in the kernel
  - **die(); // kernel oops**

Interrupts in Linux



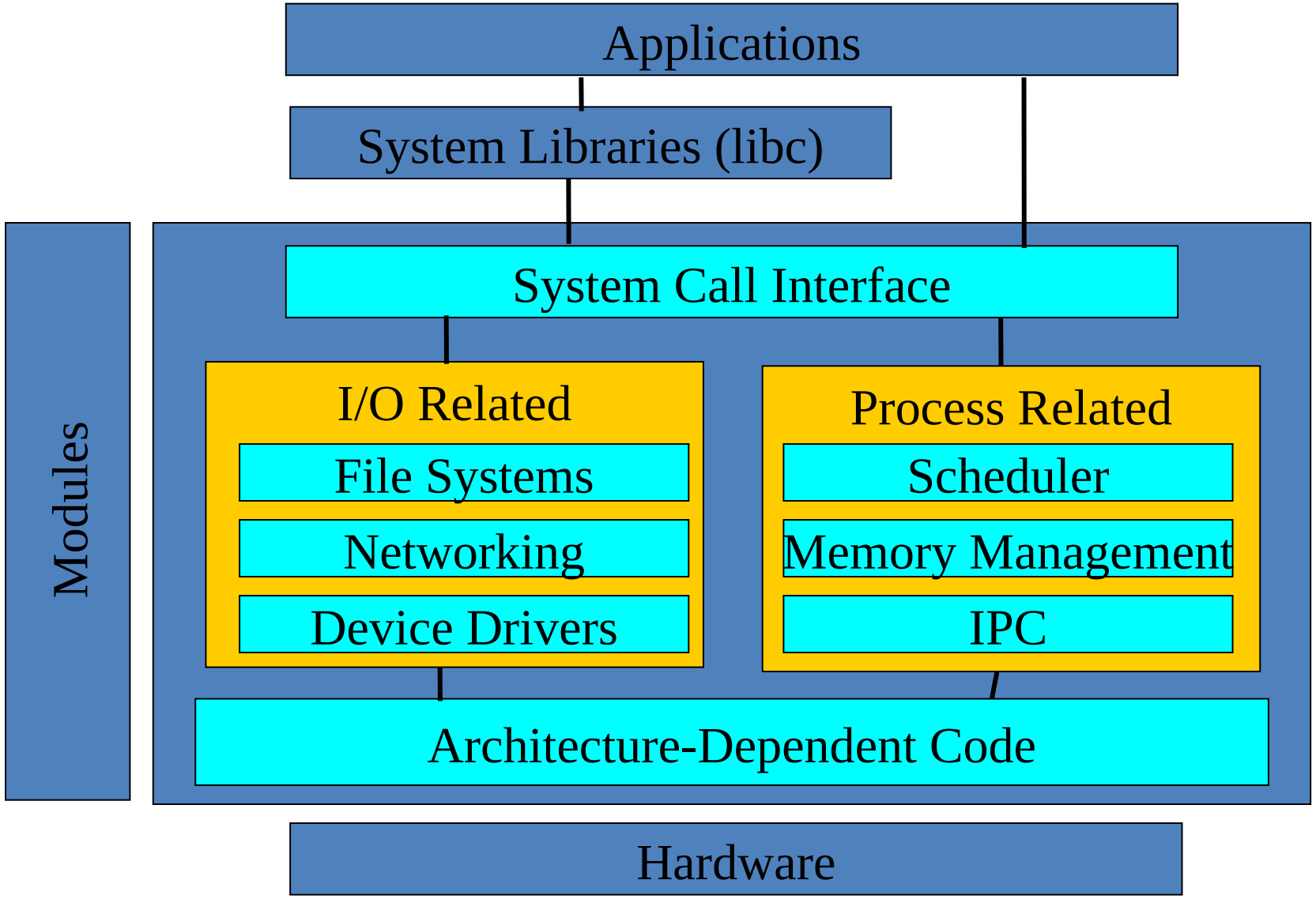
Loading an Interrupt handler

- *Hardware* locates the proper gate descriptor for this interrupt vector, and locates the new context
- Verifies Current Privilege Level (CPL) <= Descriptor Privilege level (DPL)
- Load a new stack pointer if needed
- Hw saves old IP, etc on new stack
- Set IP, etc to interrupt handler = invoke handler
  - disable interrupt by unsetting IF bit in eflags register
- Handler saves old CPU state on new stack

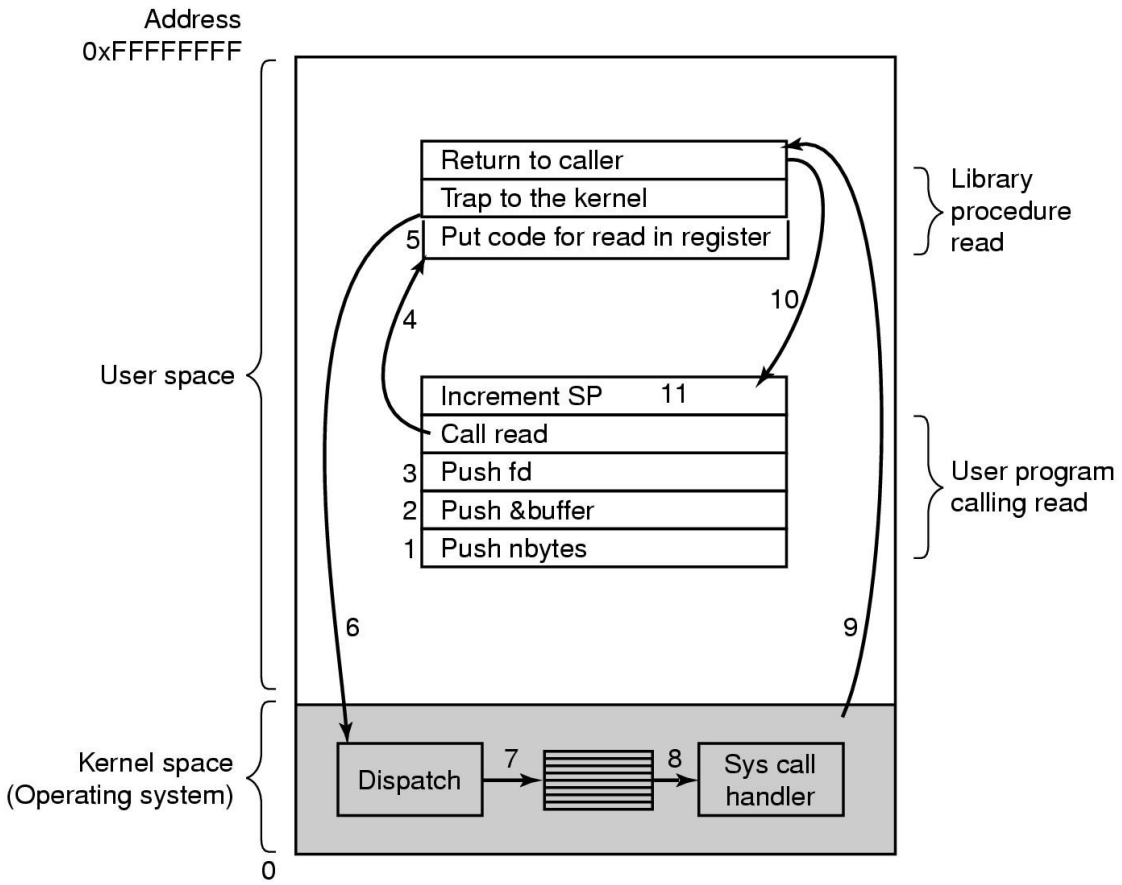
Syscall

System calls: The mechanism used by an application program to request service from the operating system. System calls often use a special machine code instruction which causes the processor to change mode (e.g. to "supervisor mode" or "protected mode"). This allows the OS to perform restricted actions such as accessing hardware devices or the memory management unit.

Syscall


Applications
System Libraries (libc)
Modules
System Call Interface
I/O Related
File Systems
Networking
Device Drivers
Process Related
Scheduler
Memory Management
IPC
Architecture-Dependent Code
Hardware

Steps in Making a System Call



There are 11 steps in making the system call

read (fd, buffer, nbytes)

Some System Calls For Process Management

Process management

Call	Description
pid = fork()	Create a child process, identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Some System Calls For File Management

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Some System Calls For Directory Management

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Some System Calls For Miscellaneous Tasks

Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Creating Processes in Linux

- The simple way: using *system* function.

- The flexible, secure, complex way: using *fork* and *exec*

- By using *system* you can create a subprocess running the standard Bourne shell (/bin/sh) and execute a command in it.

- By using *fork* function you can create a child process which is an exact copy of it's parent.

- By using *exec* family of functions, you can replace the current process image with a new one.

DOS and Windows API use *spawn* family instead of *fork* & *exec*

Creating Processes in Linux

- The *system* function, uses a shell to invoke the desired program.

- It has the same features, limitations, and security flaws of the system's shell.

```
int system (const char * command )
```

system will return the exit status of the command (see *wait*).

127: shell can not be run

-1: any other errors

system will call this command by calling *"/bin/sh -c command"*

Process Management - System

Creating Processes in Linux

Program_1

```
Here goes the program_1 code
.
.
system(program_2)
.
.
.
```

Prgoram_2

```
Program_2 will run as
a command in Bourne
Shell
```

The remaining code will continue executing.

Creating Processes in Linux

- The *fork* function creates child process which only differ in its PID with his parent.

- Return value in parent process is PID of child and in child is 0.

```
pid_t fork ( void )
```

fork will return a PID (PID of child or zero) on success and -1 on failure.

fork does not need any additional arguments. It just creates the same process as the parent.

Process Management - Fork

fork()

- A process calling `fork()` spawns a child process.
- The child is almost an identical *clone* of the parent:
  - Program Text (segment .text)
  - Stack (ss)
  - PCB (eg. registers)
  - Data (segment .data)

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Process Management - Fork

Creating a Process – fork()

pid_t fork() creates a duplicate of the calling process:

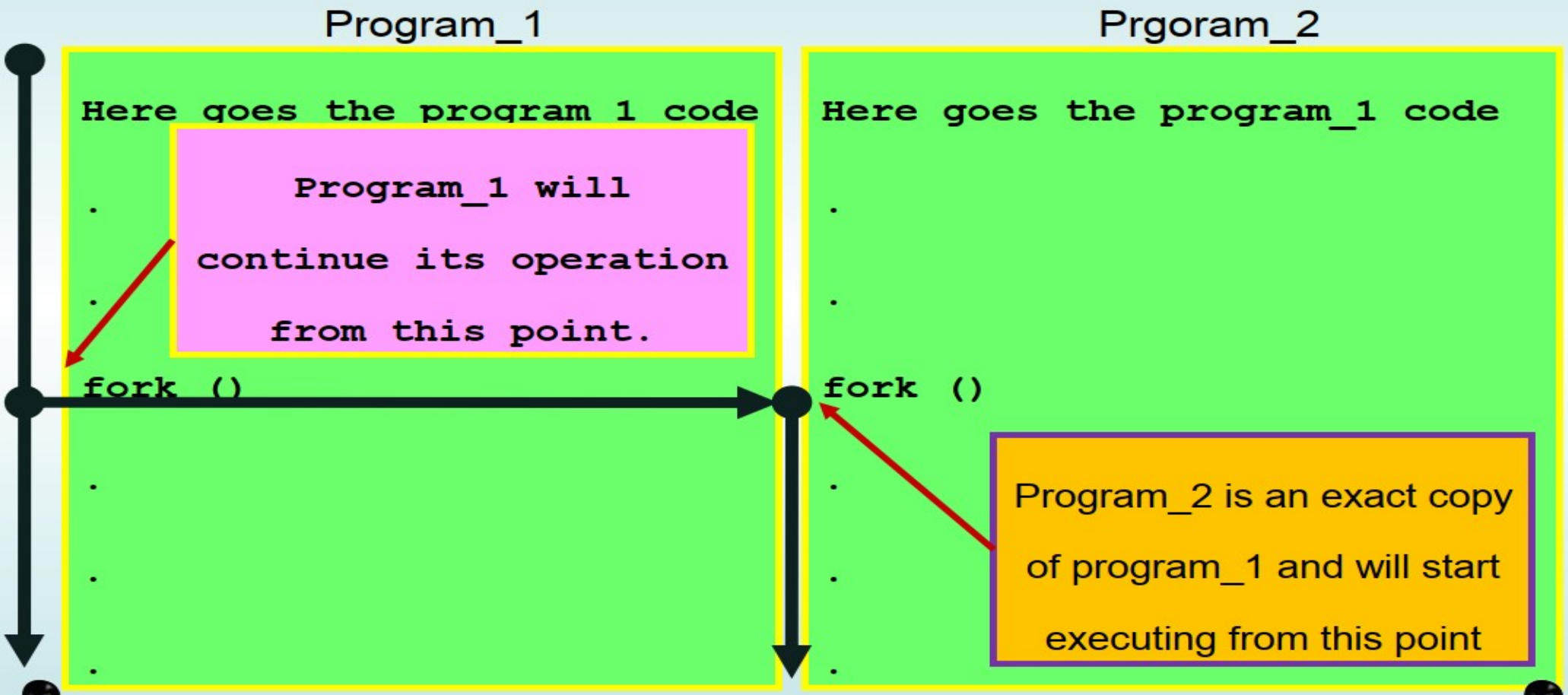
- Both processes continue with *return from fork()*
- Child gets exact copy of code, stack, file descriptors, heap, globals, and program counter

fork() returns

- -1 if fork fails
- 0 in child process
- child's PID in parent process

Process Management - Fork

Creating Processes in Linux



Process Management - Fork

Process creation with fork()

```
Pid_t pid;

Main()
{
   pid = fork();
   printf("\n Process ID : %d" , getpid() ) ;
}
```

FORK

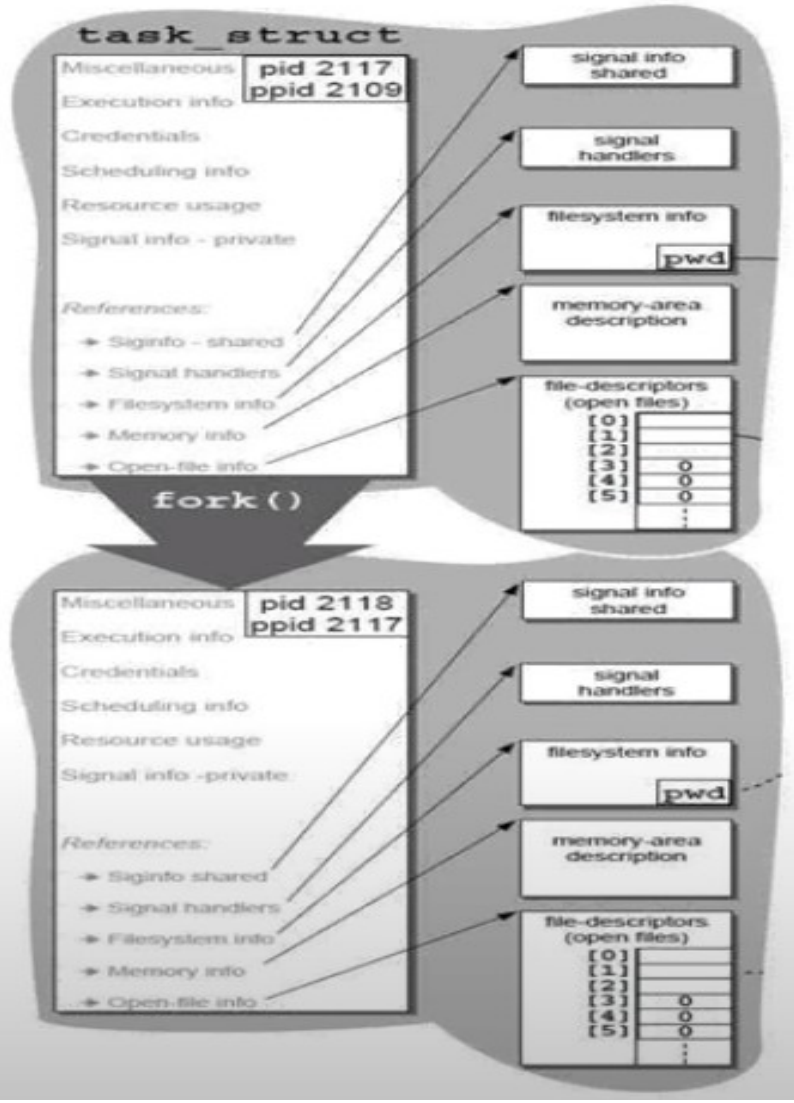
```
Pid_t pid;

Main()
{
   pid = fork();
   printf("\n Process ID : %d" , getpid() ) ;
}
```

PID =10
PPID = 200

PID = 11
PPID = 10

Process Management - Fork

Process creation with fork()



fork → sys_fork → do_fork ($SRC_HOME/kernel/fork.c)

- ✓ fail if #processes for this user exceeded
- ✓ increment #processes for this user
- ✓ fail if #processes in system exceeds maximum
  nr_threads >= max_threads
- ✓ Allocate new ***task_struct*** instance
- ✓ determine pid for child
- ✓ clear pending signals for child
- ✓ Copy the satellite structures for the child process.
  - copy ***files_struct***
  - copy ***fs_struct***
  - copy ***signal_struct***
  - copy ***sighand_struct***
  - copy ***mm_struct***
- ✓ copy ***cpu-registers*** from parent to child; fill **eip** of child with other continuation address
- ✓ set child's state to '***running***', add to runqueue and force process-switch to child
- ✓ Return PID of child to parent

Process Management - PID

Process Identification

UNIX identifies processes via a unique Process ID

Each process also knows its parent process ID since each process is created from a parent process.

Root process is the ‘init’ process

‘*getpid*’ and ‘*getppid*’ – functions to return process ID (PID) and parent process ID (PPID)

Example 1

```
#include <stdio.h>
#include <unistd.h>
int main (void) {
    printf(“I am  process %ld\n”, (long)getpid());
    printf(“My parent id is %ld\n”, (long)getppid());
    return 0;
}
```

Creating Processes in Linux

- The *exec* family, vary slightly in their capabilities and the way of calling:

  - Functions containing the letter '*p*' in their name, accept a program name and search it in current execution PATH.
  - Those who contain the letter '*v*' or '*l*' in their name, accept the argument list as an array or list for the new program.
  - Those who contain the letter '*e*' in their name, accept an array of environment variables.

exec replaces the calling process with another one, so it will never return a value on success but on failure, returns -1.

Process Management - exec

Creating Processes in Linux

Program_1

```
Here goes the program_1 code
.
.
exec (program_2)
.
.
.
```

Prgoram_2

Now, the program_2 will be replaced with progaram_1 and run till end.

The remaining program code, will never execute if *'exec'* finishes successfully.

Creating Processes in Linux

- All of the *exec* family of functions, use just one system call: *execve()*

- execl() functions are variadic functions.

- When calling exec, remember that almost all Linux applications, use argv[0] as their binary image name.

- When using *exec* family, the new process does not have the previous' signal handlers and other stuff.

- The new process has the same values for its PID, PPID, priority and permissions.

Process Management - exit

Process Termination

Normal exit (voluntary)

- Returning zero from main()
- exit(0)

Error exit (voluntary)

- exit(1)

Fatal error (involuntary)

- Divide by 0, seg fault, exceeded resources

Killed by another process (involuntary)

- Signal: kill(procID)

Process Management - exit

Process Termination

Normal exit (voluntary)

- Returning zero from main()
- exit(0)

Error exit (voluntary)

- exit(1)

Fatal error (involuntary)

- Divide by 0, seg fault, exceeded resources

Killed by another process (involuntary)

- Signal: kill(procID)

Process Management - exit

Process Operations: Termination

When a child process terminates:

- Open files are flushed and closed
- tmp files are deleted
- Child's resources are de-allocated
  - File descriptors, memory, semaphores, file locks, …
- Parent process is notified via signal SIGCHLD
- Exit status is available to parent via wait()

wait(), waitpid() System Calls

pid_t wait(int *status);

wait() causes parent process to wait (block) until *some* child finishes

wait() returns child's pid and exit status to parent

waitpid() waits for a specific child

errno	Cause
ECHILD	Caller has no unwaited-for children
EINTR	Function was interrupted by signal
EINVAL	Options parameter of waitpid was invalid

wait(), waitpid() System Calls

pid_t wait(int *status);

In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie“ state.

If a child has already terminated, then the call returns immediately. Otherwise it blocks until either a child terminates or a signal handler interrupts the call. A child that has terminated and which has not yet been waited upon by this system call (or waitpid) is termed waitable.

wait(), waitpid() System Calls

If status is not NULL, wait() stores status information in the int to which it points. This integer can be inspected with specific macros (see man pages):

WIFEXITED(status)

returns true if the child terminated normally, that is, by calling exit, or by returning from main().

WEXITSTATUS(status)

returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.

wait() & “zombie”

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child.

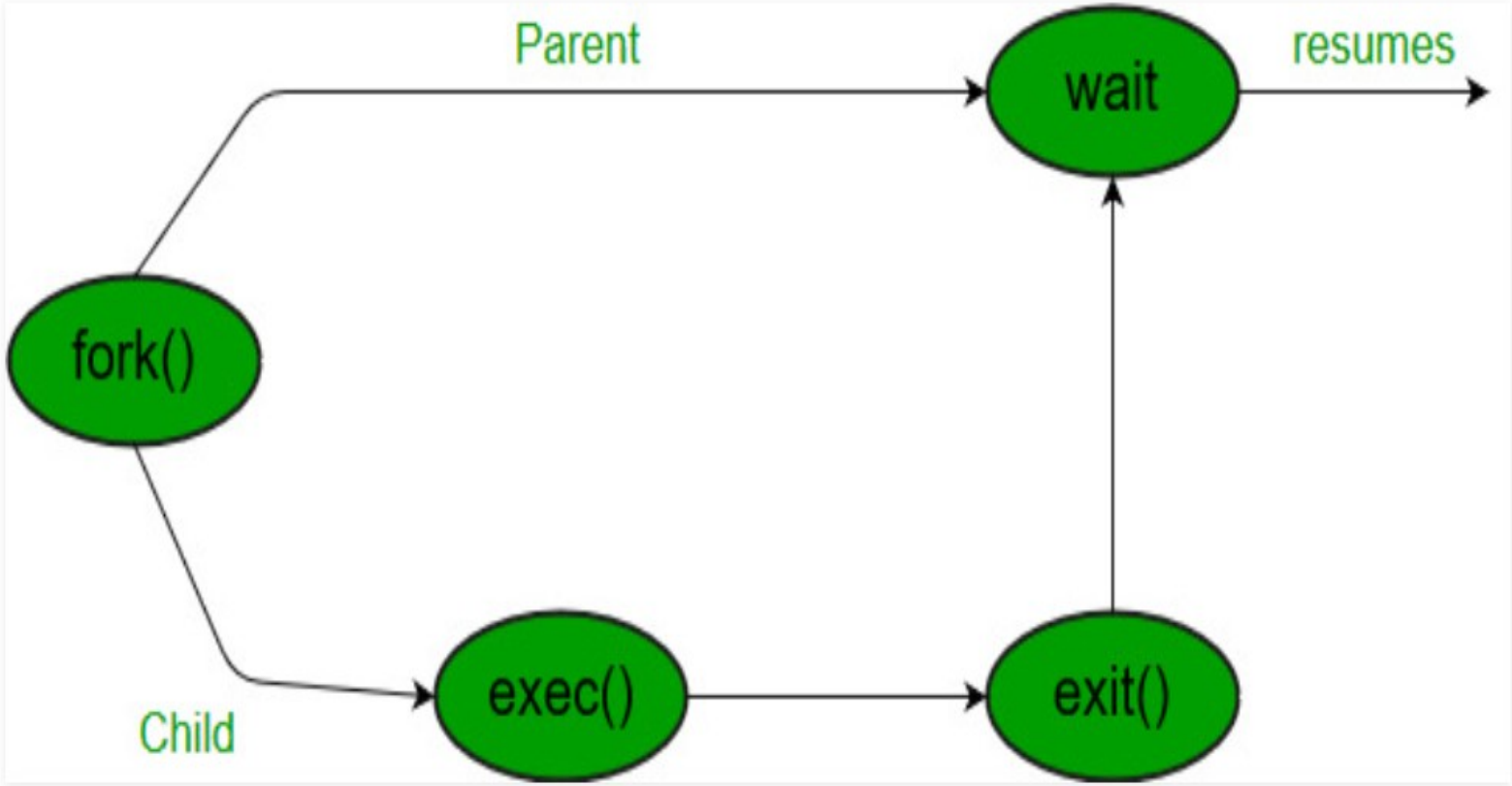
As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes.

If a parent process terminates, then its "zombie" children (if any) are adopted by init(8), which automatically performs a wait to remove the zombies.

Waiting for a child to finish

```
#include <errno.h>
#include <sys/wait.h>

pid_t childpid;

childpid = wait(NULL);
if (childpid != -1)
    printf("waited for child with pid %ld\n", childpid);
```

Process Management - wait



vfork

- Creates a new processes with the express purpose of exec-ing a new program
- New child process runs in parent's address space until exec or exit is called
- vfork guarantees that the child will run before the parent until exec or exit call is reached

Process Management - Vfork

Differences between fork() and vfork()

	fork()	vfork()
Address space	Both the child and parent process will have different address space	Both child and parent process share the same address space
Modification in address space	Any modification done by the child in its address space is not visible to parent process as both will have separate copies	Any modification by child process is visible to both parent and child as both will have same copies
CoW(copy on write)	This uses copy-on-write.	Vfork doesn't use CoW
Execution summary	Both parent and child executes simultaneously	Parent process will be suspended until child execution is completed.
Outcome of usage	Behaviour is predictable	Behaviour is not predictable

Process Management - Clone

- **Clone process** is created using primitive “clone,” by duplicating its parent process
  - Allows both processes to share same segment of code and data
  - Modification of one is visible to other, which is unlike classical processes
- Ability to clone processes brings possibility of implementing servers in which several threads may be executing

Threads

- Threads are mechanisms to do more than one job at a time.

- Threads are finer-grained units of execution.

- Threads, unlike processes, share the same address space and other resources.

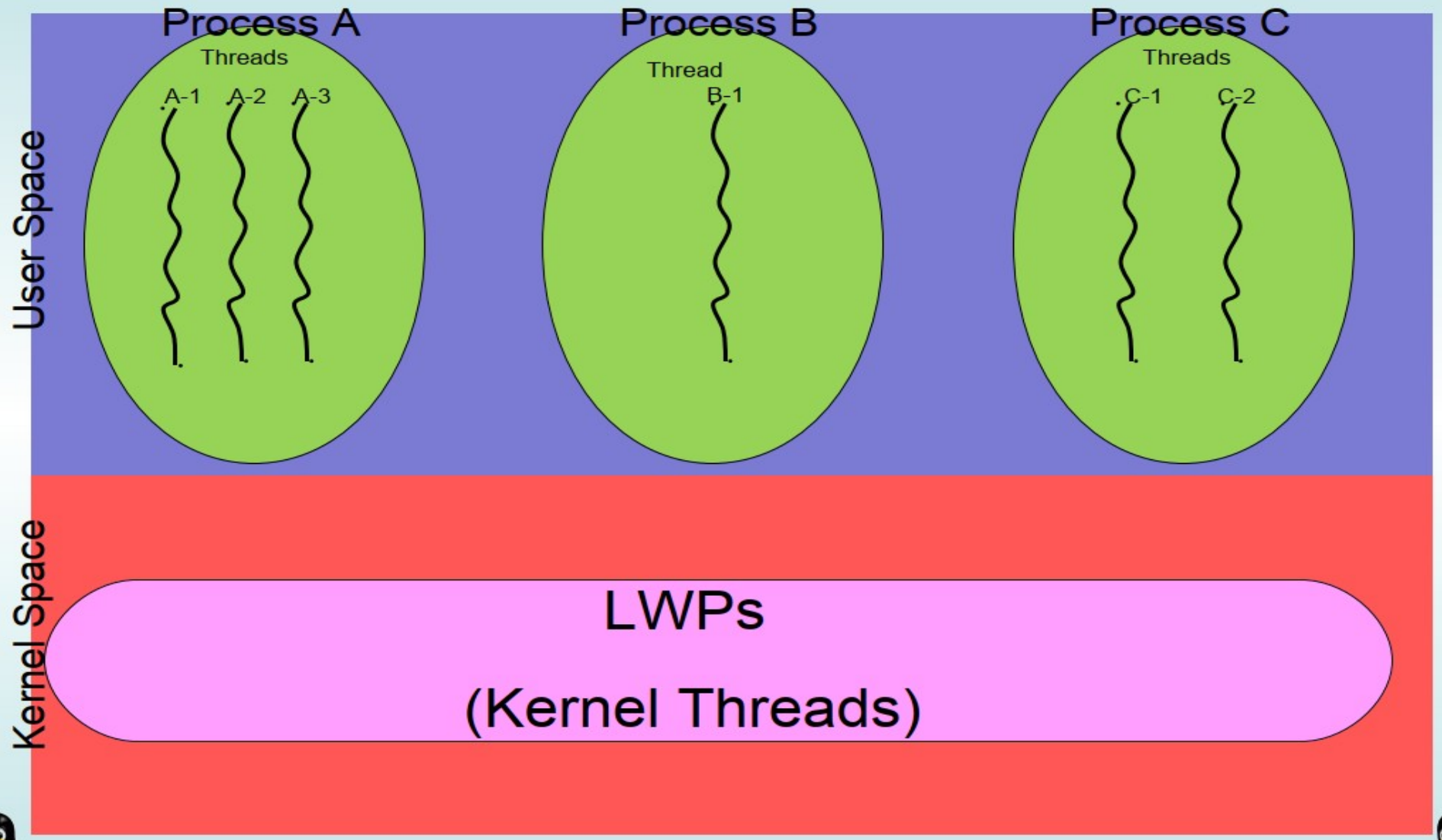
- POSIX standard thread API is not included in standard C library, they are in *libpthread.so.*

- In Linux, threads are handled by LWPs.

Process Management - Threads

- A Thread is an independent stream of instructions that can be schedule to run as such by the OS.
- Think of a thread as a “procedure” that runs independently from its main program.
- Multi-threaded programs are where several procedures are able to be scheduled to run simultaneously and/or independently by the OS.
- A Thread exists within a process and uses the process resources.
- Threads only duplicate the essential resources it needs to be independently schedulable
- A thread will die if the parent process dies.
- A thread is “lightweight” because most of the overhead has already been accomplished through the creation of the process.

Process Management - Threads



Why Use Pthreads

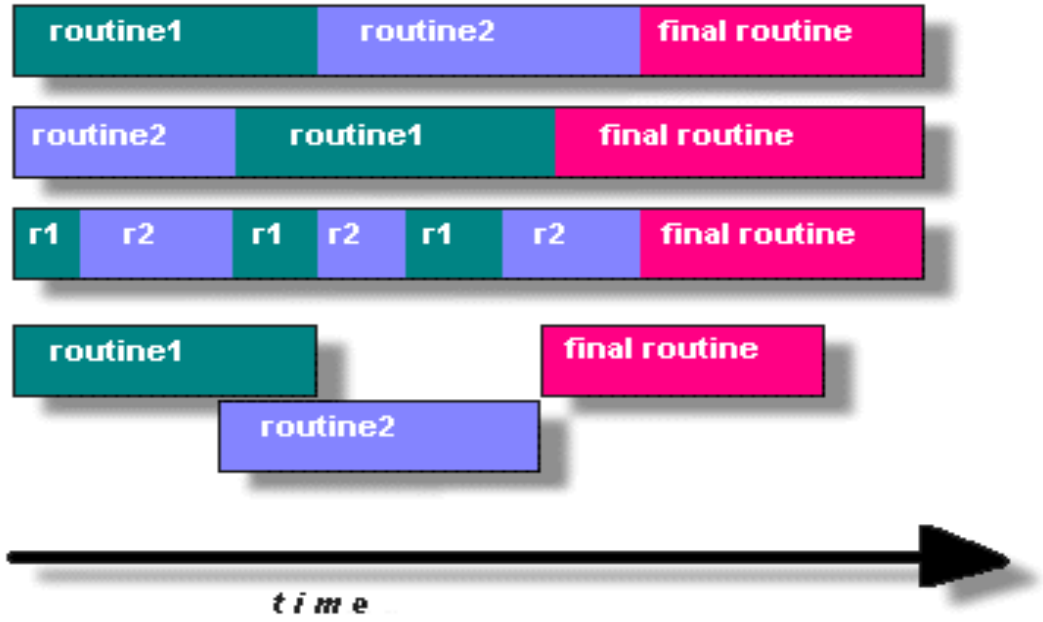
- The primary motivation behind Pthreads is improving program performance.
- Can be created with much less OS overhead.
- Needs fewer system resources to run.
- View comparison of forking processes to using a pthreads_create subroutine. Timings reflect 50,000 processes/thread creations.

Threads vs Forks

PLATFORM	fork()			pthread_create()		
	REAL	USER	SYSTEM	REAL	USER	SYSTEM
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Designing Pthreads Programs

- Pthreads are best used with programs that can be organized into discrete, independent tasks which can execute concurrently.
- Example: routine 1 and routine 2 can be interchanged, interleaved and/or overlapped in real time.



Creating threads

- Like processes, each thread has its own Thread-ID of type *pthread_t*.

- You can create a thread bye calling the *pthread_create* function.

Will return zero on success.

Thread ID of type pthread_t

Joinable or what?

```
int pthread_create( thread_id, attribute,
    thread start routine, routine arg )
```

What should this thread do?

Argument to thread function.

Creating threads

- *pthread_create* returns immediately and the specified thread will do its job separately.

- If one of the threads in a program, call *exec* the whole process image will be replaced.

- The argument passed to the thread routine is a *void* * .

- You can pass more data in a structure of type *void* *.

Joining threads

- You can wait for a thread to finish its job using *pthread_join*.

- *pthead_join* is something similar to *wait* function in processes.

- Using *pthread_join*, you can also take the return value of a thread.

- A thread, can not call *pthread_join* to wait for itself, you can use *pthread_self* function to get the TID of running thread and deciding what to do.

Joining threads

- Like processes, you can wait for a thread to finish its job…

```
int pthread_join( pthread_t thread_id, void ** return_value )
```

Will return zero on success.

Thread ID which you want to wait for.

The return value of thread will be put here.

Process Management - Threads

Pthread Management – Terminating Threads

- Several ways to terminate a thread:
  - The thread is complete and returns
  - The pthread_exit() method is called
  - The pthread_cancel() method is invoked
  - The exit() method is called
- The pthread_exit() routine is called after a thread has completed its work and it no longer is required to exist.
- If the main program finishes before the thread(s) do, the other threads will continue to execute if a pthread_exit() method exists.
- The pthread_exit() method does not close files; any files opened inside the thread will remain open, so cleanup must be kept in mind.

Thread attributes

- Second parameter in *pthread_create* is the thread attribute.

- Most useful attribute of a thread is *joinability*.

- If a thread is *joinable*, it is not automatically cleaned up.

- To clean up a *joinable* like a child process, you should call *pthread_join* .

- A *detached* thread, is automatically cleaned up.

- A joinable thread may be turned into a detached one, but can not be made joinable again.

- Using *pthread_detach* you can turn a joinable thread into detached.

Thread attributes

- If you do not clean up the joinable thread, it will become something like zombie.

- To assign an attribute to a thread, you should:

  - Create a *pthread_attr_t* object.
  - Call *pthread_attr_init* to initialize the attribute object.
  - Modify the attributes.
  - Pass a pointer to *pthread_create.*
  - Call *pthread_attr_destroy* to release the attribute object.

Thread cancelation

- A thread might be terminated by finishing its job or calling *pthread_exit* or by a request from another thread.

- The latter case is called “Thread Cancelation”.

- You can cancel a thread using *pthread_cancel*.

- If the canceled thread is not detached, you should join it after cancelation, otherwise it will become zombie.

- You can disable cancelation of a thread using *ptherad_setcancelstate()*.

Thread cancelation

- There are two cancel state:

- **PTHREAD_CANCEL_ASYNCHRONOUS:** Asynchronously cancelable (cancel at any point of execution)

- **PTHREAD_CANCEL_DEFERRED:** Synchronously cancelable (thread checks for cancellation requests)

- There are two cancelation types:

- **PTHREAD_CANCEL_DISABLE** and **PTHREAD_CANCEL_ENABLE.**

- It's a good idea to set the state to *Uncancelable* when entering critical section…

Process Scheduling

Although Linux is a preemptively multitasked operating system, it also provides a system call that allows processes to explicitly yield execution and instruct the scheduler to select a new process for execution:

```
#include <sched.h>

int sched_yield (void);
```

A call to `sched_yield( )` results in suspension of the currently running process, after which the process scheduler selects a new process to run, in the same manner as if the kernel had itself preempted the currently running process in favor of executing a new process. Note that if no other runnable process exists, which is often the case, the yielding process will immediately resume execution. Because of this uncertainty, coupled with the general belief that there are generally better choices, use of this system call is not common.

Process Scheduling

Linux provides several system calls for retrieving and setting a process' nice value. The simplest is `nice( )`:

```
#include <unistd.h>

int nice (int inc);
```

A successful call to `nice( )` increments a process' nice value by `inc`, and returns the newly updated value. Only a process with the `CAP_SYS_NICE` capability (effectively, processes owned by root) may provide a negative value for `inc`, decreasing its nice value, and thereby increasing its priority. Consequently, nonroot processes may only lower their priorities (by increasing their nice values).

On error, `nice( )` returns `-1`. However, because `nice( )` returns the new nice value, `-1` is also a successful return value. To differentiate between success and failure, you can zero out `errno` before invocation, and subsequently check its value. For example:

Process Scheduling

A preferable solution is to use the `getpriority( )` and `setpriority( )` system calls, which allow more control, but are more complex in operation:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

These calls operate on the process, process group, or user, as specified by `which` and `who`. The value of `which` must be one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, in which case `who` specifies a process ID, process group ID, or user ID, respectively. If `who` is `0`, the call operates on the current process ID, process group ID, or user ID, respectively.

A call to `getpriority( )` returns the highest priority (lowest numerical nice value) of any of the specified processes. A call to `setpriority( )` sets the priority of all specified processes to `prio`. As with `nice( )`, only a process possessing `CAP_SYS_NICE` may raise a process' priority (lower the numerical nice value). Further, only a process with this capability can raise or lower the priority of a process not owned by the invoking user.

Like `nice( )`, `getpriority( )` returns `-1` on error. As this is also a successful return value, programmers should clear `errno` before invocation if they want to handle error conditions. Calls to `setpriority( )` have no such problem; `setpriority( )` always returns `0` on success, and `-1` on error.

Process Scheduling

Processes can manipulate the Linux scheduling policy via `sched_getscheduler( )` and `sched_setscheduler( )`:

```
#include <sched.h>

struct sched_param {
        /* ... */
        int sched_priority;
        /* ... */


};

int sched_getscheduler (pid_t pid);

int sched_setscheduler (pid_t pid,
                        int policy,
                        const struct sched_param *sp);
```

A successful call to `sched_getscheduler( )` returns the scheduling policy of the process represented by `pid`. If `pid` is `0`, the call returns the invoking process' scheduling policy. An integer defined in `<sched.h>` represents the scheduling policy: the first in, first out policy is `SCHED_FIFO`; the round-robin policy is `SCHED_RR`; and the normal policy is `SCHED_OTHER`. On error, the call returns `-1` (which is never a valid scheduling policy), and `errno` is set as appropriate.

Thread priority

pthread_attr_setschedparam()

Set a thread's scheduling parameters attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedparam(
            pthread_attr_t * attr,
            const struct sched_param * param );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see *pthread_attr_init()*.

param

A pointer to a `sched_param` structure that defines the thread's scheduling parameters.

Thread priority

pthread_attr_getschedparam()

Get thread scheduling parameters attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedparam(
                    const pthread_attr_t * attr,
                    struct sched_param * param );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see *pthread_attr_init()*.

param

A pointer to a `sched_param` structure where the function can store the current scheduling parameters.

Processor Affinity

```
#include <sched.h>

typedef struct cpu_set_t;

size_t CPU_SETSIZE;

void CPU_SET (unsigned long cpu, cpu_set_t *set);
void CPU_CLR (unsigned long cpu, cpu_set_t *set);
int CPU_ISSET (unsigned long cpu, cpu_set_t *set);
void CPU_ZERO (cpu_set_t *set);

int sched_setaffinity (pid_t pid, size_t setsize,
                       const cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize,
                       cpu_set_t *set);
```

Processor Affinity

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);        /* clear all CPUs */
CPU_SET (0, &set);      /* allow CPU #0 */
CPU_CLR (1, &set);      /* forbid CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
        perror ("sched_setaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
        int cpu;

        cpu = CPU_ISSET (i, &set);
        printf ("cpu=%i is %s\n", i,
                cpu ? "set" : "unset");
}
```

Processor Affinity

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
        perror ("sched_getaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
        int cpu;

        cpu = CPU_ISSET (i, &set);
        printf ("cpu=%i is %s\n", i,
                cpu ? "set" : "unset");
}
```

Linux Performance - RSS

- Contemporary NICs support multiple receive and transmit descriptor queues (multi-queue). On reception, a NIC can send different packets to different queues to distribute processing among CPUs. The NIC distributes packets by applying a filter to each packet that assigns it to one of a small number of logical flows. Packets for each flow are steered to a separate receive queue, which in turn can be processed by separate CPUs. This mechanism is generally known as **Receive-side Scaling (RSS)**.

- The goal of RSS and the other scaling techniques is to increase performance uniformly.

Linux Performance - RSS

- The filter used in RSS is typically a hash function over the network and/or transport layer headers-- for example, a 4-tuple hash over IP addresses and TCP ports of a packet. The most common hardware implementation of RSS uses a 128-entry indirection table where each entry stores a queue number. The receive queue for a packet is determined by masking out the low order seven bits of the computed hash for the packet (usually a Toeplitz hash), taking this number as a key into the indirection table and reading the corresponding value.

- Some advanced NICs allow steering packets to queues based on programmable filters. For example, webserver-bound TCP port 80 packets can be directed to their own receive queue. Such tuple filters can be configured from ethtool (--config-ntuple).

==== RSS Configuration

The driver for a multi-queue capable NIC typically provides a kernel

Linux Performance - RSS

RSS Configuration

- The driver for a multi-queue capable NIC typically provides a kernel module parameter for specifying the number of hardware queues to configure. In the bnx2x driver, for instance, this parameter is called **num_queues**. A typical RSS configuration would be to have one receive queue for each CPU if the device supports enough queues, or otherwise at least one for each memory domain, where a memory domain is a set of CPUs that share a particular memory level (L1, L2, NUMA node, etc.).

- The indirection table of an RSS device, which resolves a queue by masked hash, is usually programmed by the driver at initialization. The default mapping is to distribute the queues evenly in the table, but the indirection table can be retrieved and modified at runtime using ethtool commands (--show-rxfh-indir and --set-rxfh-indir). Modifying the indirection table could be done to give different queues different relative weights.

Linux Performance - RSS

RSS IRQ Configuration

- Each receive queue has a separate IRQ associated with it. The NIC triggers this to notify a CPU when new packets arrive on the given queue. The signaling path for PCIe devices uses message signaled interrupts (MSI-X), that can route each interrupt to a particular CPU. The active mapping of queues to IRQs can be determined from /proc/interrupts. By default, an IRQ may be handled on any CPU. Because a non-negligible part of packet processing takes place in receive interrupt handling, it is advantageous to spread receive interrupts between CPUs. To manually adjust the IRQ affinity of each interrupt see Documentation/IRQ-affinity.txt. Some systems will be running irqbalance, a daemon that dynamically optimizes IRQ assignments and as a result may override any manual settings.

Linux Performance - RSS

When to Use RSS

- RSS should be enabled when latency is a concern or whenever receive interrupt processing forms a bottleneck. Spreading load between CPUs decreases queue length. For low latency networking, the optimal setting is to allocate as many queues as there are CPUs in the system (or the NIC maximum, if lower). The most efficient high-rate configuration is likely the one with the smallest number of receive queues where no receive queue overflows due to a saturated CPU, because in default mode with interrupt coalescing enabled, the aggregate number of interrupts (and thus work) grows with each additional queue. Per-cpu load can be observed using the mpstat utility, but note that on processors with hyperthreading (HT), each hyperthread is represented as a separate CPU. For interrupt handling, HT has shown no benefit in initial tests, so limit the number of queues to the number of CPU cores in the system.

Linux Performance - RPS

- Receive Packet Steering (RPS) is logically a software implementation of RSS. Being in software, it is necessarily called later in the datapath. Whereas RSS selects the queue and hence CPU that will run the hardware interrupt handler, RPS selects the CPU to perform protocol processing above the interrupt handler. This is accomplished by placing the packet on the desired CPU backlog queue and waking up the CPU for processing.

- RPS has some advantages over RSS:
  - it can be used with any NIC,
  - software filters can easily be added to hash over new protocols,
  - it does not increase hardware device interrupt rate (although it does introduce inter-processor interrupts (IPIs)).

Linux Performance - RPS

- RPS is called during bottom half of the receive interrupt handler, when a driver sends a packet up the network stack with netif_rx() or netif_receive_skb(). These call the get_rps_cpu() function, which selects the queue that should process a packet.

- The first step in determining the target CPU for RPS is to calculate a flow hash over the packets addresses or ports (2-tuple or 4-tuple hash depending on the protocol). This serves as a consistent hash of the associated flow of the packet. The hash is either provided by hardware or will be computed in the stack. Capable hardware can pass the hash in the receive descriptor for the packet; this would usually be the same hash used for RSS (e.g. computed Toeplitz hash). The hash is saved in skb->hash and can be used elsewhere in the stack as a hash of the packets flow.

Each receive hardware queue has an associated list of CPUs to which RPS may enqueue packets for processing. For each received packet, an index into the list is computed from the flow hash modulo the size

Linux Performance - RPS

- Each receive hardware queue has an associated list of CPUs to which RPS may enqueue packets for processing. For each received packet, an index into the list is computed from the flow hash modulo the size of the list. The indexed CPU is the target for processing the packet, and the packet is queued to the tail of that CPUs backlog queue. At the end of the bottom half routine, IPIs are sent to any CPUs for which packets have been queued to their backlog queue. The IPI wakes backlog processing on the remote CPU, and any queued packets are then processed up the networking stack.

Linux Performance - RPS

RPS Configuration

- RPS requires a kernel compiled with the CONFIG_RPS kconfig symbol (on by default for SMP). Even when compiled in, RPS remains disabled until explicitly configured. The list of CPUs to which RPS may forward traffic can be configured for each receive queue using a sysfs file entry:

/sys/class/net/<dev>/queues/rx-<n>/rps_cpus

- This file implements a bitmap of CPUs. RPS is disabled when it is zero (the default), in which case packets are processed on the interrupting CPU.

== Suggested Configuration

For a single queue device, a typical RPS configuration would be to set

the rps_cpus to the CPUs in the same memory domain of the interrupting

Linux Performance - RPS

Suggested Configuration

- For a single queue device, a typical RPS configuration would be to set the rps_cpus to the CPUs in the same memory domain of the interrupting CPU. If NUMA locality is not an issue, this could also be all CPUs in the system. At high interrupt rate, it might be wise to exclude the interrupting CPU from the map since that already performs much work.

- For a multi-queue system, if RSS is configured so that a hardware receive queue is mapped to each CPU, then RPS is probably redundant and unnecessary. If there are fewer hardware queues than CPUs, then RPS might be beneficial if the rps_cpus for each queue are the ones that share the same memory domain as the interrupting CPU for that queue.

Linux Performance - RPS

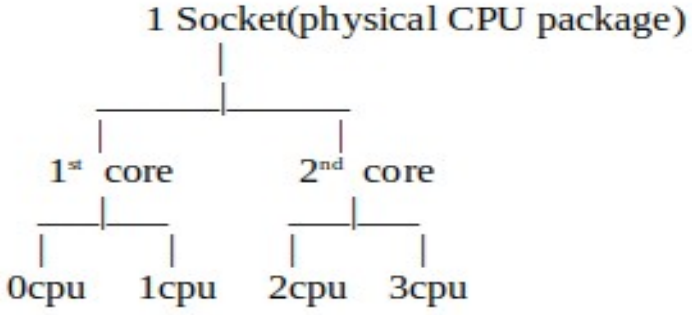
Suggested Configuration

- For a single queue device, a typical RPS configuration would be to set the rps_cpus to the CPUs in the same memory domain of the interrupting CPU. If NUMA locality is not an issue, this could also be all CPUs in the system. At high interrupt rate, it might be wise to exclude the interrupting CPU from the map since that already performs much work.

- For a multi-queue system, if RSS is configured so that a hardware receive queue is mapped to each CPU, then RPS is probably redundant and unnecessary. If there are fewer hardware queues than CPUs, then RPS might be beneficial if the rps_cpus for each queue are the ones that share the same memory domain as the interrupting CPU for that queue.

Linux Performance – CPU Isolation

- In Multi-processor architecture processors is directly get compared with processing power. Common notion is more processors means better performance but that is not always the case. In Most of Embedded chip-set multiple processing units are merged so one single chip can give you multiple processor functionality, processing units often referred as cores. To get an clear picture let's take an example. In general 4 core machine we can see below cpu arrangement.



- machine is having 2 cores and 4 logical processing unit which mean 4 parallel thread can run. When we say CPU Isolation then we are isolating one or more cpu.

Linux Performance – CPU Isolation

Scenario before CPU Isolation:

- When scheduler code get loaded on processor it put a spin lock and search for process which is ready to execute. Once it found it will load that process and remove lock form process table. Like wise on every processor scheduler code will get loaded

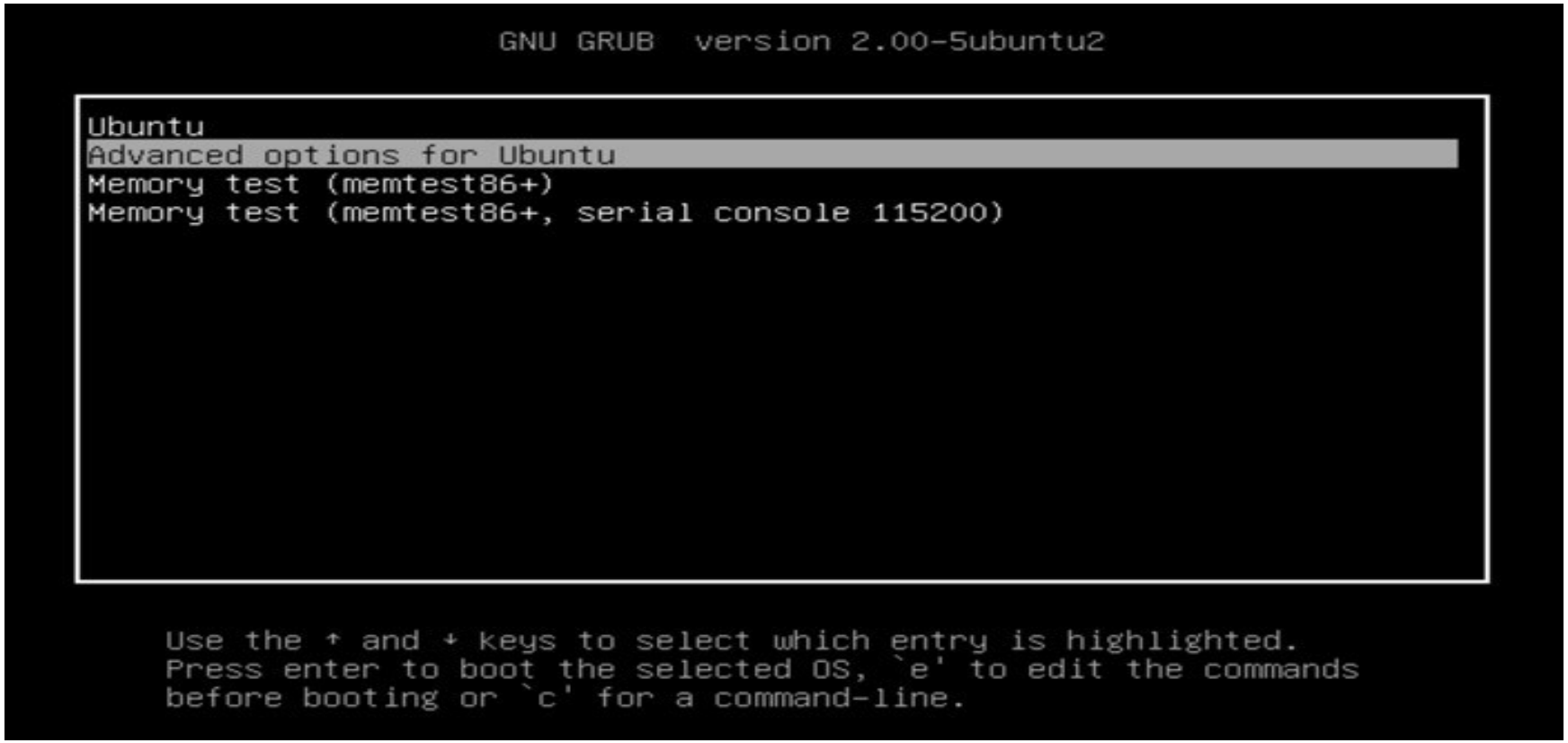
Scenario After CPU Isolation:

- If CPU isolation is activated for one of core then scheduler code will be never get loaded on isolated processor so no other process will get scheduled, until explicitly we tell to scheduler to load particular process on that processor no process will be loaded over it. Or in some architecture isolated CPU are removed/ Ignored in scheduler algorithms. it is basically Design dependent that how CPU is isolated from the process to be get loaded

Linux Performance – CPU Isolation

Steps to isolate CPU

❑To isolate cpu temporarily or for single boot we can do it by below steps,while Linux booting by grub loader user will see grub menu as shown in below picture.



Linux Performance – CPU Isolation

Steps to isolate CPU

❑To isolate cpu temporarily or for single boot we can do it by below steps,while Linux booting by grub loader user will see grub menu as shown in below picture.Here we can press ‘e’ to edit the kernel boot command. Once we press ‘e’ a new window will appear. Which is actual grub script to load application. There generally you will see line like below.

linux bootvmlinuz{kernel-version} root=UUID=.... quite splash isolcpus=2

above command loads kernel image there we are manually passing **“isolcpus=2”** parameter to isolate CPU number 2. after this change we can boot the kernel by pressing F10 or as mentioned in grub note. After this once system boots cpu2 will be isolated and no user/kernel process is loaded. Only few cpu bounded process will run over it.

Linux Performance – CPU Isolation

- To check CPU isolation is done or not. Fire below command

cat /sys/devices/system/cpu/isolated Isolated CPU number you will get as output. If No Cpu is isolated then you will get an empty Output

Linux Performance – CPU Affinity

- There are two types of CPU affinity:
- **soft affinity**: also called natural affinity, is the tendency of a scheduler to try to keep processes on the same CPU as long as possible.scheduler, which has poor CPU affinity. This behavior results in the ping-pong effect. The scheduler bounces processes between multiple processors each time they are scheduled and rescheduled.

- **Hard affinity:** on the other hand, is what a CPU affinity system call provides. It is a requirement, and processes must stick to a specified hard affinity. If a processor is bound to CPU zero, for example, then it can run only on CPU zero. It will not run on any processor Even if other CPUs are free and cpu0 is busy. It will wait until cpu0 gets free.

Linux Performance – CPU Affinity

Benefit:

❑CPU affinity is optimizing cache performance.when processes bounce between processors they constantly cause cache invalidation, and the data they want is never in the cache when they need it. Thus, cache miss rates grow very large. CPU affinity protects against this and improves cache performance.Scheduling that process to execute on the same processor improves its performance by reducing performance-degrading events such as cache misses.

- CPU isolation can help you to observe your code performance and process switch latency on different processor scenarios.
- Once CPU is isolated, Isolated CPU is completely free to execute specific Pinned process. When we will pin a specific process to isolated CPU then cpu will execute solely pinned process only. Doing this will reduce the latency for pinned process and throughput will increase as no other process is there to disturb or take cpu away. For critical and important process we can do this way which will help to reduce latency and increase through put.
- CPU affinity is optimizing cache performance

Linux Performance - NUMA

- Non-Uniform Memory Access or Non-Uniform Memory Architecture (NUMA) is a physical memory design used in SMP (multiprocessors) architecture, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors.

- In a NUMA system, processors, memory, and I/O are grouped together into cells. The latency and bandwidth characteristics of communication within a cell are “fast” while going outside a cell is “slow”. Since the memory in NUMA systems is physically distributed but logically shared, these systems offer better performance to applications that are optimized to use their features. For non-optimized applications, they still offer better performance since the default behavior is designed to be benign if not beneficial and they still have access to much larger shared resources of memory, CPUs, and disk space.

Linux Performance - SMP

- Symmetrical multiprocessing (SMP) denotes a multiprocessor architecture in which no CPU is selected as the Master CPU, but rather all of them cooperate on an equal basis, hence the name "symmetrical." As usual, we shall focus on Intel SMP architectures.

Synchronization mechanism

IPC

- Inter Process Communication (IPC) is transfer of data between processes.

- In Linux there are some methods of IPC:

  - Shared Memory.
  - Mapped Memory.
  - Pipe (Named and Unnamed).
  - Socket (Remote, Local).

Pipes

- A pipe is a communicational device that permits unidirectional communication.

- The first data written into pipe is the first one that is read.

- If the writer, writes faster than the reader and pipe is full, the writer blocks.

- If the reader reads tries to read an empty pipe, it blocks.

- You can create pipes using *pipe()*.

Synchronization mechanism - Pipe

Pipes

On success, 0 is returned and on error, -1 is returned and errno is set.

An array of size 2, which contains read and write file descriptors.

```
int  pipe ( int filedes[2] );
```

- *pipe()* stores the reading file descriptor in array position 0 and the writing file descriptor in position 1.

- Read and write file descriptors are available only in calling process and its children.

- You can use pipes to communicate between threads in a process.

Synchronization mechanism - Pipe

Pipes

The file descriptor related to created process stdin (or stdout).

The command you wish to execute.

Might be “w” or “r” as for writing or reading.

```
FILE * popen( const char * command, const char * type)
```

- You can use *popen()* to send data to or receive data from a program running in a subprocess.

- After closing the stream (using *pclos()*), *pclose()* waits for the child process to terminate.

Synchronization mechanism – Named Pipe

Named Pipes

0 on success and -1 on error.

Name and address of the pipe file in the system.

Permission flags.

```
int mkfifo (const char * pathname, mode_t mode)
```

- You can access a named pipe like an ordinary file.

- One program must open it for writing and another for reading.

Synchronization mechanism – Named Pipe

- In computing, a named pipe (also known as a **FIFO**) is one of the methods for intern-process communication.
  - It is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.
  - A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
  - Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.

Synchronization mechanism – Named Pipe

- A FIFO special file is entered into the filesystem by calling mkfifo() in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Synchronization mechanism – Message Queue

Message Queue

- Another IPC mechanism in Linux.

- Implemented both SYS V type and POSIX type and it's structure is somehow like pipes (FIFO).

- A process initializes a message queue and itself or other processes can put messages in this queue, knowing the MSQID of this message queue.

- After finishing the job, one process should deallocate the queue.

- If you don't remove a message queue it will remain even after process termination.

- You can use *ipcs* command to view the current message queues.

Synchronization mechanism – Message Queue

Message Queue

Message queue ID on success and -1 on error

The flags (and permissions) to use for the action (IPC_CREAT, IPC_EXCL, …)

```
int msgget (key_t key, int msgflg);
```

The message queue Key to create or to connect to. All key features are same as other IPC mechanisms

- You can connect to a queue and also create one, using *msgget()*.

Synchronization mechanism – Message Queue

Message Queue

Zero on success and -1 on error

A pointer to the struct which you are going to send

Size of the payload (mtext member of your struct)

```
int msgsnd (int msqid, const void * msgp, size_t msgsz,
            int msgflg);
```

The message queue ID to send messages through

Some flags related to the message and IPC actions

- The message you are going to send should be a struct containing two members: m_type and m_text.

Synchronization mechanism – Message Queue

Message Queue

Zero on success and -1 on error

A pointer to the struct which you are going to send

Size of the payload (mtext member of your struct)

```
int msgrcv (int msqid, const void * msgp, size_t msgsz,
            long msgtyp, int msgflg);
```

The message queue ID to send messages through

Second member of the sent message struct or 0 or a negative number

Some flags related to the message and IPC actions

- If the message has size bigger than what is said here, the truncated message could be lost (MSG_NOERROR flag)

Synchronization mechanism – Message Queue

Message Queue

Zero or msqid or index on success and -1 on error

A pointer to a struct you want to put returned data in it

```
int msgctl (int msqid, int cmd, struct msqid_ds * buf);
```

The message queue ID to send messages through

The command you want to perform on this msgQ (like other IPC commands)

- The msqid_ds type is described in sys/msg.h and contains information about desired message queue.

Synchronization mechanism – Posix Message Queue

- POSIX supports asynchronous, indirect message passing through the notion of message queues
- A message queue can have many readers and many writers
- Priority may be associated with the queue
- Intended for communication between processes (not threads)
- Message queues have attributes which indicate their maximum size, the size of each message, the number of messages currently queued etc.
- An attribute object is used to set the queue attributes when the queue is created

Synchronization mechanism – Posix Message Queue

- Message queues are given a name when they are created
- To gain access to the queue, requires an mq_open name
- mq_open is used to both create and open an already existing queue (also mq_close and mq_unlink)
- Sending and receiving messages is done via mq_send and mq_receive
- Data is read/written from/to a character buffer.
- If the buffer is full or empty, the sending/receiving process is blocked unless the attribute O_NONBLOCK has been set for the queue (in which case an error return is given)
- If senders and receivers are waiting when a message queue becomes unblocked, it is not specified which one is woken up unless the priority scheduling option is specified

Synchronization mechanism – Posix Message Queue

- A process can also indicate that a signal should be sent to it when an empty queue receives a message and there are no waiting receivers
- In this way, a process can continue executing whilst waiting for messages to arrive or one or more message queues
- It is also possible for a process to wait for a signal to arrive; this allows the equivalent of selective waiting to be implemented
- If the process is multi-threaded, each thread is considered to be a potential sender/receiver in its own right

Synchronization mechanism – Shared Memory

Shared Memory

- Is the fastest form of interprocess communication and is also called: Fast Local Communication.

- It allows two or more processes to access the same memory.

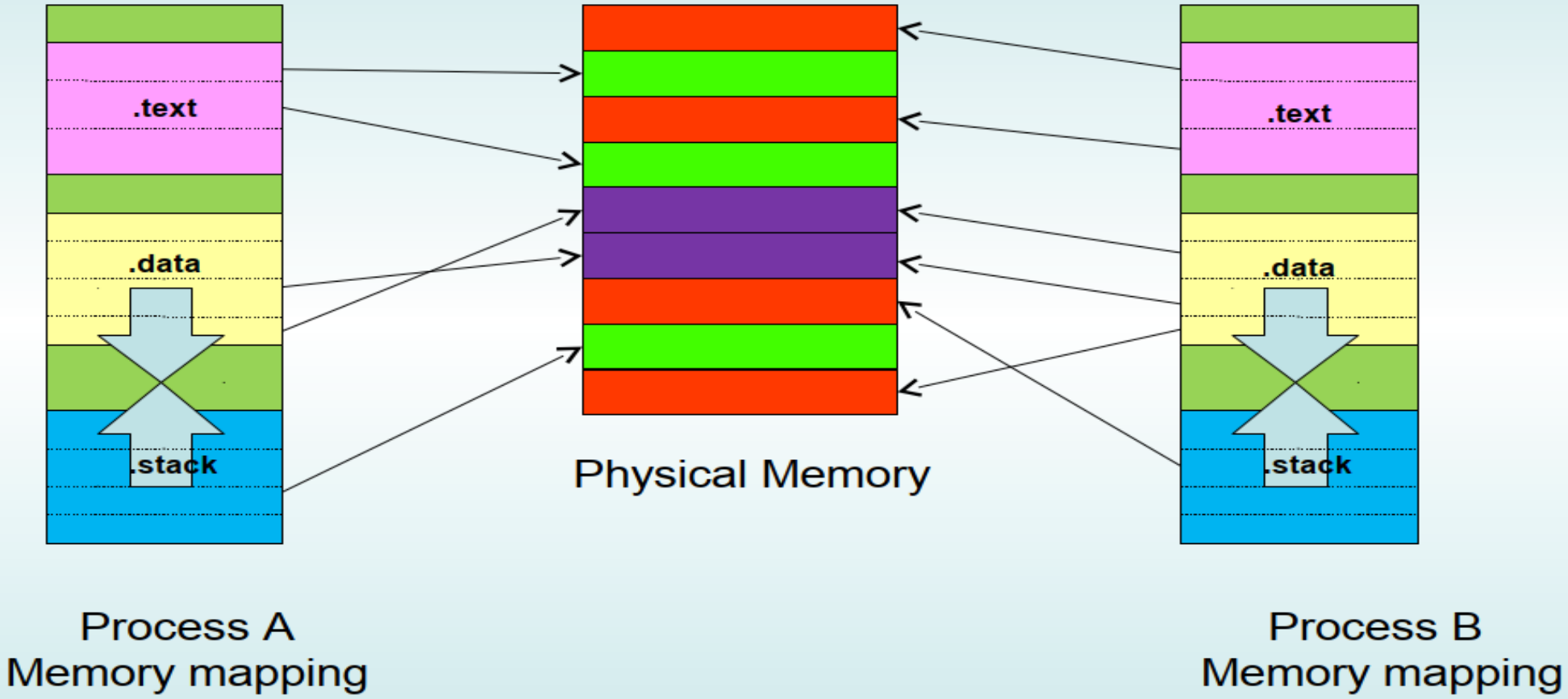
- Linux kernel does not take care of synchronization between processes to access the shared memory.

- Process semaphores are a suitable way of synchronization.

- Using shared memory for each process is like calling *malloc.*

Synchronization mechanism – Shared Memory

Shared Memory



Using Shared Memory

- To use shared memory in Linux, one process must allocate the segment.
- Each process desiring to access the segment, must attach the segment and after finishing its job must detach the segment.
- One process at the end must deallocate the segment.
- Allocating a new shared memory segment, causes virtual memory pages to be created.

Using Shared Memory

- Allocating an existing segment, does not create new pages, but returns and identifier to the existing ones.

- All shared memory segments are allocated as integral multiples of the system's page size.

- On Linux systems, page size is 4KB.

- You should obtain the page size by calling the *getpagesize()* function.

Using Shared Memory

- You can use *ipcs -m* command to see currently assigned shared segments.

- Using *ipcrm -m* command, you can remove unused shared segments left behind by processes.

- There are some limitations on using shared memory in linux.

- *SHMALL, SHMMAX, SHMMIN, SHMMNI* are corresponding values to these limitations which are located under /proc/sys/kernel/.

Synchronization mechanism – Shared Memory

Using Shared Memory

- SHMALL: system-wide maximum of shared memory pages.

- SHMMAX: maximum size in bytes for a shared memory segment.

- SHMMIN: minimum size in bytes for a shared memory segment.

- SHMMNI: system-wide maximum number of shared segments.

If the minimum size of a shared memory segment is equal to *page size*, then what is *SHMMIN*?

Using Shared Memory

- A process may allocate a shared memory segment using *shmget()*.

- Its first argument is a key specified to the shared segment.

- Other processes can access to the same shared memory segment using the same key.

- To ensure that the key is not previously used, you can use the special constant IPC_PRIVATE.

Synchronization mechanism – Shared Memory

Using Shared Memory

```
int shmget ( key_t key, size_t size, int shmflg )
```

A valid segment identifier on success and -1 on error.

The key you wish to specify for the shared segment.

The segment size (will be rounded up to the multiple of page size).

Permission and other specifications of the shared segment.

Synchronization mechanism – Shared Memory

Using Shared Memory

- shmflg is the logical OR of flags. The most useful flags are:

  - IPC_CREAT: Is used to create a new shared segment.
  - IPC_EXCL: Is used with IPC_CREAT to ensure failure if the segment already exists.
  - Mode flags (see the manual page of stat.h for details).

If IPC_CREAT is used without IPC_EXCL, and the segment key already exists, the existing segment's id will be returned and no error will occur.

Using Shared Memory

- Permission flags are:

Mode bit	Meaning
S_IRWXU	R, W, X by owner
S_IRUSR	Read by owner
S_IWUSR	Write by owner
S_IXUSR	Execute by owner
S_IRWXG	R, W, X by group
S_IRGRP	Read by group
S_IWGRP	Write by group
S_IXGRP	Execute by group
S_IRWXO	R, W, X by other
S_IROTH	Read by other
S_IWOTH	Write by other
S_IXOTH	Execute by other

Synchronization mechanism – Shared Memory

Using Shared Memory

- To use a shared memory segment, a process must attach it.

- *shmat()* is used to attach to a shared memory segment with given segment identifier.

- You can tell *shmat()* where in your process address space to map the shared memory.

- If you call *fork()* the child will inherit the shared memory.

- When you finished with shared memory, you can detach it using *shmdt()*.

Synchronization mechanism – Shared Memory

Using Shared Memory

```
void * shmat ( int shmid, const void * shmaddr, int
shmflg )
```

On success, will return the address of attached shared memory. On error, -1 is returned and errno is set.

The address in your process address space in which you want the shared memory be mapped.

The segment ID (returned by *shmget()*).

Could be ***SHM_RND***, ***SHM_RDONLY***, ***SHM_REMAP*** *(Linux specific)*

Synchronization mechanism – Shared Memory

Using Shared Memory

Any process who knows the key, can

access to this shared memory segment

(regarding to permission flags)

Using Shared Memory

- The *shmctl()* returns information about a shared memory segment and can modify it.

- Using *shmctl()* you can also deallocate a shared memory segment.

- Each shared memory segment should be deallocated explicitly.

- The *shmctl()* fills the *shmid_ds* type structure.

Synchronization mechanism – Shared Memory

Using Shared Memory

```
int shmctl ( int id, int cmd, struct shmid_ds * buf )
```

On success depends on cmd. On error, -1 is returned and errno is set.

The segment ID (returned by *shmget()*).

IPC_STAT, IPC_SET, IPC_RMID, RPC_INFO *(Linux specific)*, ***SHM_INFO*** *(Linux specific)*, ***SHM_STAT*** *(Linux specific)*, ***SHM_LOCK*** *(Linux specific)*, ***SHM_UNLOCK*** *(Linux specific)*.

An structure which contains the information you want to be set or to be read.

Synchronization mechanism – Semaphore

Process Semaphore

- Processes must coordinate access to shared memory.

- Process semaphores like thread semaphores are kind of counter with two operations: *POST* & *WAIT*.

- Process semaphores come in sets.

- The last process using a semaphore set, must explicitly remove it.

- Unlike shared memory, removing a semaphore set causes Linux to deallocate immediately.

Synchronization mechanism – Semaphore

Process Semaphore

- To use a semaphore set, you should first allocate it using *semget()*.

- The *semget()* system call, will return a semaphore ID regarding to the key you gave it before.

- After allocating the semaphore, you should initialize it using *semctl()* .

- After initializing the semaphore set, you can do POST or WAIT on it using *semop()* system call.

- The last process must invoke *semctl()* to remove the semaphore.

Process Semaphore

- Allocating an existing semaphore will return it's semaphore ID.

- *semget()* flags behave the same way as *shmget()* does.

- You can use *ipcs –s* command to view information about existing semaphore sets.

- Using *ipcrm –s* you may remove the semaphore sets.

- Each semaphore in a set has the following associated values:

```
unsigned short semval;      /* semaphore value */
unsigned short semzcnt;     /* # waiting for zero */
unsigned short semncnt;     /* # waiting for increase */
pid_t sempid;               /* PID that did last OP */
```

Synchronization mechanism – Semaphore

Allocating a semaphore set

```
int semget ( key_t key, int nsems, int semflg )
```

A valid semaphore identifier on success and -1 on error.

The key you wish to specify for the semaphore set.

The number of semaphores you wish to have in this set.

Permission and other specifications of the semaphore set.

Initializing Semaphores

- To initialize a semaphore set, you must do:
  - Set the semaphore value of all members to desired values.
  - Set the last change time of all members.
  - Set other specifications of semaphore set members.
- To do so, you can use *semctl()* function.
- As mentioned in *semctl()* manual page, the calling program must define a *semun* union.

Synchronization mechanism – Semaphore

Initializing Semaphores

```
int semctl ( int semid, int semnum, int cmd, union semun args )
```

On success depends on cmd. On error, -1 is returned and errno is set.

Number of desired semaphore in set.

Use of fourth argument is depended on CMD (might be ignored).

The semaphore ID (returned by *semget()*).

IPC_STAT, IPC_SET, IPC_RMID, IPC_INFO *(Linux specific)*, ***SEM_INFO*** *(Linux specific)*, ***SEM_STAT*** *(Linux specific)*, ***GETALL, SETALL, GETVAL, SETVAL, GETNCNT, GETZCNT, GETPID***

Initializing Semaphores

- Depending on CMD, you may need to provide the fourth argument.

- If so, you must define the *union semun* yourself like below:

```
union semun
{
        int val;      /* Value for SETVAL */
        struct semid_ds * buf;     /* Buffer for IPC_STAT, IPC_SET */
        unsigned short int * array;   /* Array for GETALL, SETALL */
        struct seminfo * __buf; /* Buffer for IPC_INFO (linux specific) */
};
```

semid_ds and *ipc_perm* (an structure in *semid_ds*) are filled with information about semaphores

Wait and post operation

- To do *wait* and *post* on a semaphore in a set, you can use *semop()*.

- *semop()* performs desired operation on selected semaphores in a set.

- *semop()* takes an array of operation structure.

- Each operation structure is related to one semaphore in a set.

- Operation structure is of type *sembuf* and contains:

```
unsigned short sem_num;   /* Semaphore number */
short sem_op;    /* Semaphore operation */
short sem_flg;    /* Semaphore flags */
```

Semaphore Deallocation

- The semaphore operations are performed atomically.

- If the *SEM_UNDO* flag is set in operation structure, the action will be undone when process terminates.

- You must deallocate the semaphore set when you finnished.

- Unlike shared memory segments, removing a semaphore set causes Linux to deallocate immediately.

Synchronization mechanism – Critical Section

Critical Section

- The ultimate cause of most bugs involving threads is that they are accessing the same data at the same time.

- The section of code which is responsible to access the shared data, is called *Critical Section* .

- A critical section is part of code that should be executed completely or not at all (a thread should not be interrupted when it is in this section)

- If you do not protect the *Critical Section*, your program might crash because of *Race Condition.*

Synchronization mechanism – Race Condition

Race Condition

- Race Condition is a condition in which threads are racing each other to change the same data structure.

- Because there is no way to know when the system scheduler will interrupt one thread and execute the other one, the buggy program may crash once and finish regularly next time.

- To eliminate race conditions, you need a way to make operations *atomic* (uninterruptible).

Synchronization mechanism – Threads

- Counting Semaphores
  - Permit a limited number of threads to execute a section of the code
- Binary Semaphores - Mutexes
  - Permit only one thread to execute a section of the code
- Condition Variables
  - Communicate information about the state of shared data

Synchronization mechanism – PosixSemaphore

Data type

Semaphore is a variable of type **sem_t**

Include **<semaphore.h>**

Atomic Operations

```
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);

Initialize an semaphore

Returns

0 on success

-1 on failure, sets **errno**

Parameters

sem:

Target semaphore

pshared:

0: only threads of the creating process can use the semaphore

Non-0: other processes can use the semaphore

value:

Initial value of the semaphore

You cannot make a copy of a semaphore variable!!!

Synchronization mechanism – PosixSemaphore

- Sharing semaphores between threads within a process is easy, use **pshared==0**
- A non-zero **pshared** allows any process that can access the semaphore to use it Places the semaphore in the global (OS) environment Forking a process creates copies of any semaphore it has

Synchronization mechanism – PosixSemaphore

sem_init can fail

On failure

sem_init returns -1 and sets **errno**

errno	cause
EINVAL	**Value > sem_value_max**
ENOSPC	Resources exhausted
EPERM	Insufficient privileges

```
sem_t semA;

if (sem_init(&semA, 0, 1) == -1)
  perror("Failed to initialize semaphore semA");
```

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_destroy(sem_t *sem);

Destroy an semaphore

Returns

0 on success

-1 on failure, sets **errno**

Parameters

sem:

Target semaphore

Notes

Can destroy a **sem_t** only once

Destroying a destroyed semaphore gives undefined results

Destroying a semaphore on which a thread is blocked gives undefined results

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_post(sem_t *sem);

Unlock a semaphore - same as signal

Returns

0 on success

-1 on failure, sets **errno** (**== EINVAL** if semaphore doesn't exist)

Parameters

sem:

Target semaphore

sem > 0: no threads were blocked on this semaphore, the semaphore value is incremented

sem == 0: one blocked thread will be allowed to run

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_wait(sem_t *sem);

Lock a semaphore

Blocks if semaphore value is zero

Returns

0 on success

-1 on failure, sets **errno** (**== EINTR** if interrupted by a signal)

Parameters

sem:

Target semaphore

sem > 0: thread acquires lock

sem == 0: thread blocks

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_trywait(sem_t *sem);

Test a semaphore's current condition

Does not block

Returns

0 on success

-1 on failure, sets **errno** (**== AGAIN** if semaphore already locked)

Parameters

sem:

Target semaphore

sem > 0: thread acquires lock

sem == 0: thread returns

Synchronization mechanism – Mutex

A typical sequence in the use of a mutex

1. Create and initialize **mutex**
2. Several threads attempt to lock **mutex**
3. Only one succeeds and now owns **mutex**
4. The owner performs some set of actions
5. The owner unlocks **mutex**
6. Another thread acquires **mutex** and repeats the process
7. Finally **mutex** is destroyed

Synchronization mechanism – Mutex

Creating a mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Initialize a pthread mutex: the mutex is initially unlocked

Returns

0 on success

Error number on failure

EAGAIN: The system lacked the necessary resources; ENOMEM: Insufficient memory ; EPERM: Caller does not have privileges; EBUSY: An attempt to re-initialise a mutex; EINVAL: The value specified by attr is invalid

Parameters

mutex: Target mutex

attr:

NULL: the default mutex attributes are used

Non-NULL: initializes with specified attributes

Synchronization mechanism – Mutex

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Destroy a pthread mutex

Returns

0 on success

Error number on failure

EBUSY: An attempt to re-initialise a mutex; EINVAL: The value specified by attr is invalid

Parameters

mutex: Target mutex

Synchronization mechanism – Mutex

Locking/unlocking a mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Returns

0 on success

Error number on failure

EBUSY: already locked; EINVAL: Not an initialised mutex; EDEADLK: The current thread already owns the mutex; EPERM: The current thread does not own the mutex

Synchronization mechanism – Conditional Variable

- Used to communicate information about the state of shared data
  - Execution of code depends on the state of
    - A data structure or
    - Another running thread
- Allows threads to synchronize based upon the actual value of data
- Without condition variables
  - Threads continually poll to check if the condition is met

Synchronization mechanism – Conditional Variable

- Signaling, not mutual exclusion
  - A mutex is needed to synchronize access to the shared data
- Each condition variable is associated with a single mutex
  - Wait atomically unlocks the mutex and blocks the thread
  - Signal awakens a blocked thread

Synchronization mechanism – Conditional Variable

Similar to pthread mutexes

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Synchronization mechanism – Conditional Variable

Waiting

- Block on a condition variable.
- Called with **mutex** locked by the calling thread
- Atomically release **mutex** and cause the calling thread to block on the condition variable
- On return, **mutex** is locked again

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const
    struct timespec *abstime);
```

Synchronization mechanism – Conditional Variable

<u>Signaling</u>

int pthread_cond_signal(pthread_cond_t *cond);

unblocks at least one of the blocked threads

int pthread_cond_broadcast(pthread_cond_t *cond);

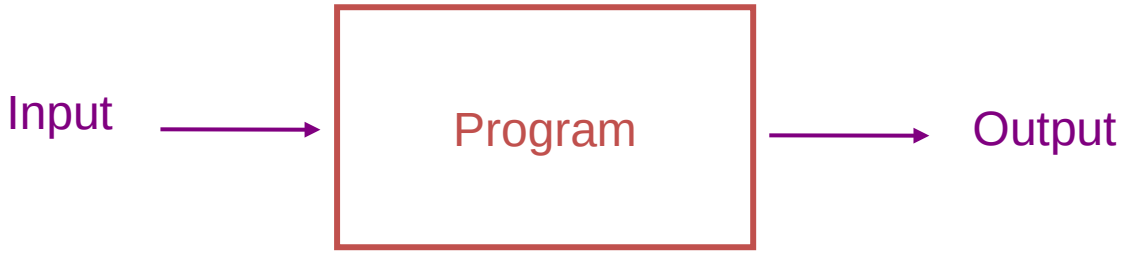
unblocks all of the blocked threads

Signals are not saved

Must have a thread waiting for the signal or it will be lost

Profiling – Gprof

- Timing, Instrumenting, Profiling
- How slow is the code?
  - How long does it take for certain types of inputs?
- Where is the code slow?
  - Which code is being executed most?
- Why is the code running out of memory?
  - Where is the memory going?
  - Are there leaks?
- Why is the code slow?
  - How imbalanced is my hash table or binary tree?



Profiling – Gprof

- Gather statistics about your program's execution
  - e.g., how much time did execution of a function take?
  - e.g., how many times was a particular function called?
  - e.g., how many times was a particular line of code executed?
  - e.g., which lines of code used the most time?
- Most compilers come with profilers
  - e.g., **gprof**
- Gprof (GNU Performance Profiler)
  - **gcc –pg –o mymath mymath.c**
- Running the code (e.g., testmath)
  - Produces output file gmon.out containing statistics
- Printing a human-readable report from gmon.out
  - gprof testmath > gprofreport

Profiling – ftrace

Ftrace is a tracing framework for the Linux kernel. It was added to the kernel back in 2008 and has evolved a lot since then.

Ftrace stands for function tracer and basically lets you watch and record the execution flow of kernel functions. It was created by Steven Rostedt, derived from two other tools called latency tracer from Ingo Molnar and Steven’s logdev utility.

Profiling – ftrace

With ftrace you can really see what the kernel is doing. You can trace function calls and learn a lot about how the kernel works. You can find out which kernel functions are called when you run a user space application. You can profile functions, measure execution time and find out bottlenecks and performance issues. You can identify hangs in kernel space. You can measure the time it takes to run a real-time task and find out latency issues. You can measure stack usage in kernel space and find out possible stack overflows. You can really do a lot of things to monitor and find bugs in the kernel!

Profiling – ftrace

And this is the filesystem interface provided by ftrace:

```
# ls /sys/kernel/tracing/
README                      set_ftrace_filter
available_events            set_ftrace_notrace
available_filter_functions  set_ftrace_pid
available_tracers           set_graph_function
buffer_size_kb              set_graph_notrace
buffer_total_size_kb        snapshot
current_tracer              stack_max_size
dyn_ftrace_total_info       stack_trace
enabled_functions           stack_trace_filter
events                      timestamp_mode
free_buffer                 trace
function_profile_enabled    trace_clock
hwlat_detector              trace_marker
instances                   trace_marker_raw
max_graph_depth             trace_options
options                     trace_pipe
per_cpu                     trace_stat
printk_formats              tracing_cpumask
saved_cmdlines              tracing_max_latency
saved_cmdlines_size         tracing_on
saved_tgids                 tracing_thresh
set_event                   uprobe_events
set_event_pid               uprobe_profile
```

Profiling – ftrace

We can print the list of available tracers:

```
# cat available_tracers
hwlat   blk      function_graph   wakeup_dl   wakeup_rt
wakeup  irqsoff  function         nop
```

There are function tracers (*function*, *function_graph*), latency tracers (*wakeup_dl*, *wakeup_rt*, *irqsoff*, *wakeup*, *hwlat*), I/O tracers (*blk*) and many more!

To enable a tracer, we just have to write its name to *current_tracer*:

```
# echo function > current_tracer
```

Profiling – ftrace

And we can read the trace buffer with the *trace* or *trace_pipe* file:

```
# cat trace
# tracer: function
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [001] d...    23.695208: _raw_spin_lock_irqsave <-hrtimer_next_event_wi...
          <idle>-0     [001] d...    23.695209: __hrtimer_next_event_base <-hrtimer_next_event...
          <idle>-0     [001] d...    23.695210: __next_base <-__hrtimer_next_event_base
          <idle>-0     [001] d...    23.695211: __hrtimer_next_event_base <-hrtimer_next_event...
          <idle>-0     [001] d...    23.695212: __next_base <-__hrtimer_next_event_base
          <idle>-0     [001] d...    23.695213: __next_base <-__hrtimer_next_event_base
          <idle>-0     [001] d...    23.695214: _raw_spin_unlock_irqrestore <-hrtimer_next_eve...
          <idle>-0     [001] d...    23.695215: get_iowait_load <-menu_select
          <idle>-0     [001] d...    23.695217: tick_nohz_tick_stopped <-menu_select
          <idle>-0     [001] d...    23.695218: tick_nohz_idle_stop_tick <-do_idle
          <idle>-0     [001] d...    23.695219: rcu_idle_enter <-do_idle
          <idle>-0     [001] d...    23.695220: call_cpuidle <-do_idle
          <idle>-0     [001] d...    23.695221: cpuidle_enter <-call_cpuidle
```

Profiling – ftrace

The function graph tracer is an alternative function tracer that traces not only the function entry but also the return of the function, allowing you to create a call graph of the function flow and output the trace data in a C-like style with information about the duration of each function.

```
# echo function_graph > current_tracer

# cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)   1.000 us    |                } /* idle_cpu */
 0)               |                tick_nohz_irq_exit() {
 0)               |                  ktime_get() {
 0)   1.000 us    |                    clocksource_mmio_readl_up();
 0)   7.666 us    |                  }
 0) + 14.000 us   |                }
 0)               |                rcu_irq_exit() {
 0)   1.334 us    |                  rcu_nmi_exit();
 0)   7.667 us    |                }
 0) ! 556.000 us  |              } /* irq_exit */
 0) # 1187.667 us |            } /* __handle_domain_irq */
 0) # 1194.333 us |          } /* gic_handle_irq */
 0)   <========== |
 0) # 8197.333 us |        } /* cpuidle_enter_state */
 0) # 8205.334 us |      } /* cpuidle_enter */
[...]
```

Profiling – ftrace

```
# cd /sys/kernel/tracing/
# echo 0 > trace
# echo nop > current_tracer
# echo 1 > events/syscalls/sys_enter_mkdir/enable
# echo 1 > events/syscalls/sys_enter_fork/enable
# echo 1 > tracing_on ; mkdir ~/fooo; echo 0 > tracing_on
# cat trace
```

Profiling – kprobes

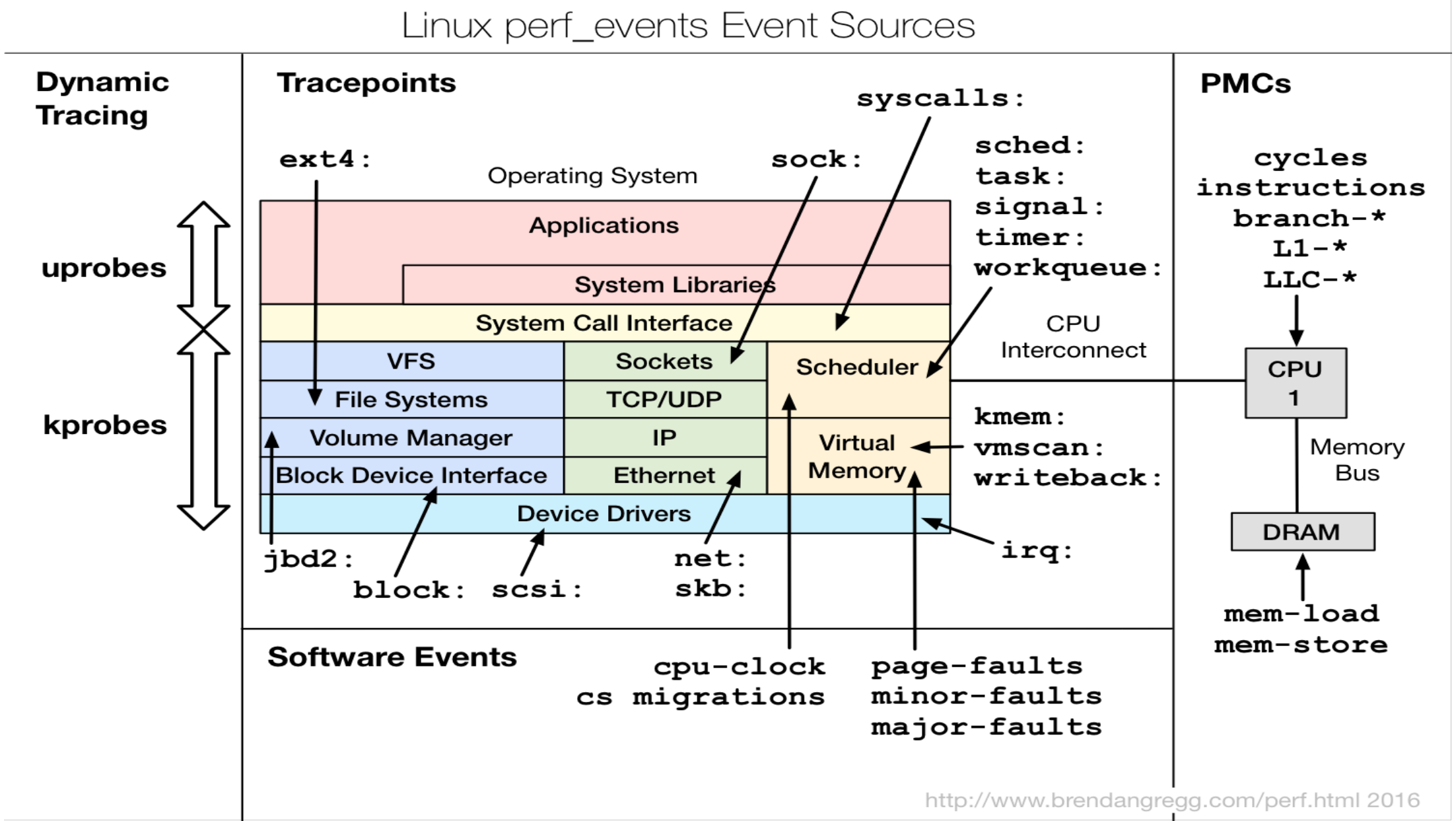
```
# cd /sys/kernel/tracing/
# echo 'p:myprobewargs do_mkdirat pathname=%si mode=%dx' >> /sys/kernel/debug/tracing/kprobe_events
# echo 'p:myprobenoargs do_mkdirat' >> /sys/kernel/debug/tracing/kprobe_events
# echo 1 > /sys/kernel/debug/tracing/events/kprobes/myprobenoargs/enable
# echo 1 > /sys/kernel/debug/tracing/events/kprobes/myprobewargs/enable
# echo 0 > trace
# echo 1 > tracing_on ; mkdir ~/fooo; echo 0 > tracing_on
# cat trace
```

Profiling – perf

perf_events is an event-oriented observability tool, which can help you solve advanced performance and troubleshooting functions. Questions that can be answered include:

- ❑Why is the kernel on-CPU so much? What code-paths?
- ❑Which code-paths are causing CPU level 2 cache misses?
- ❑Are the CPUs stalled on memory I/O?
- ❑Which code-paths are allocating memory, and how much?
- ❑What is triggering TCP retransmits?
- ❑Is a certain kernel function being called, and how often?
- ❑What reasons are threads leaving the CPU?

Profiling – perf



Profiling – perf

Perf Subcommands

- **Perf stat**: collects and display event data (performance counters) during a command execution.

- **Perf record**: run a command, store its profiling (sampling mode) in output file (perf.data) (no output is produced).

- **Perf report**: display data previously recorded in output file (perf.data)

- **Perf diff**: diff between perf.data files

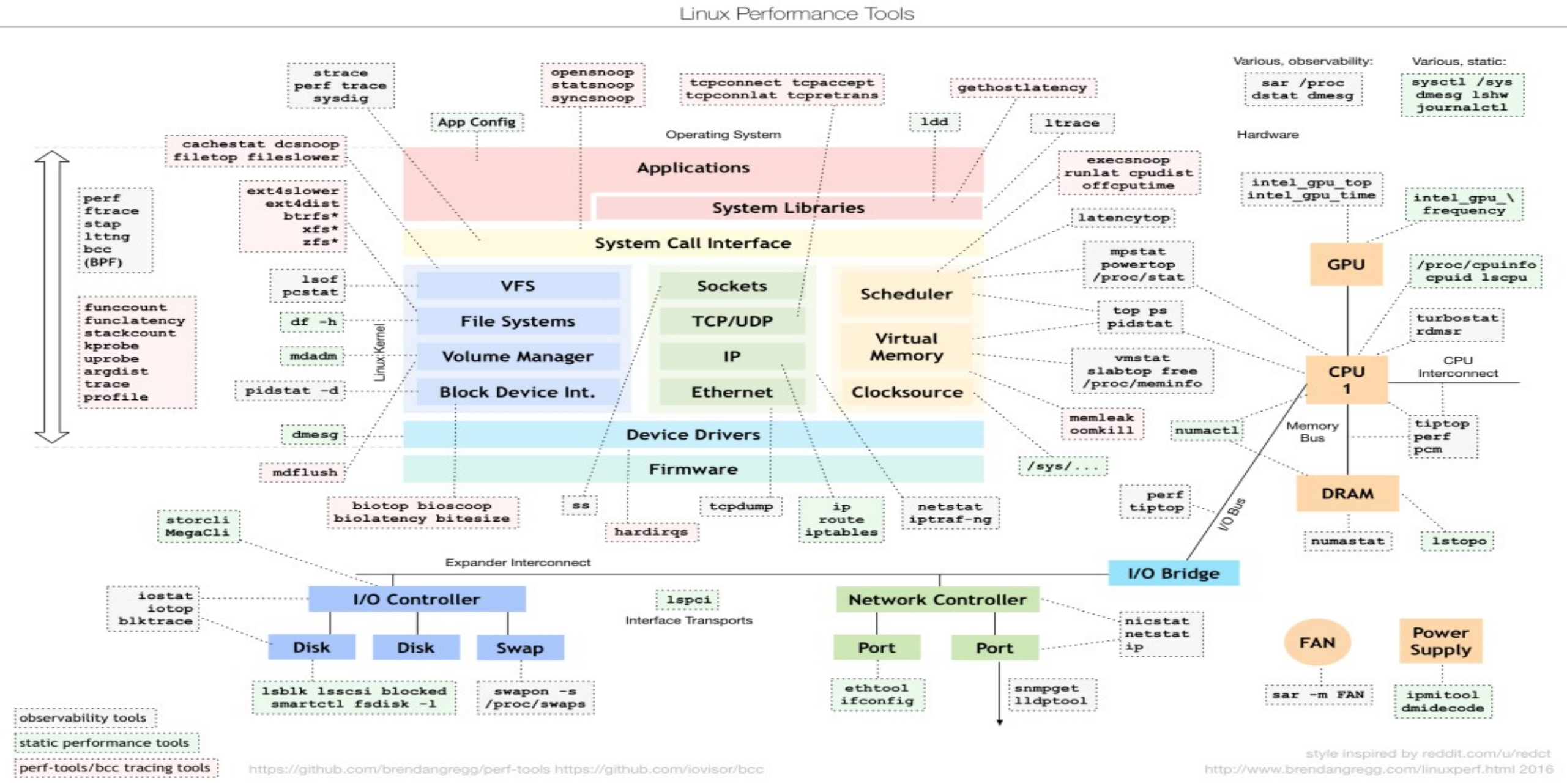
- **Perf top**: performance counter profile in realtime (live)

- **Perf probe**: define dynamic trace points.

Profiling – perf

```
#  sudo perf top
# sudo perf record -e probe:* -aRg /bin/sh ./command.sh
# sudo perf report
```

Debugging


Linux Performance Tools
strace
perf trace
sysdig
opensnoop
statsnoop
syncsnoop
tcpconnect tcpaccept
tcpconnlat tcpretrans
gethostlatency
Various, observability:
sar /proc
dstat dmesg
Various, static:
sysctl /sys
dmesg lshw
journalctl
App Config
ldd
ltrace
Operating System
Hardware
cachestat dcsnoop
filetop fileslower
Applications
execsnoop
runlat cpudist
offcputime
perf
ftrace
stap
lttng
bcc
(BPF)
ext4slower
ext4dist
btrfs*
xfs*
zfs*
System Libraries
intel_gpu_top
intel_gpu_time
intel_gpu_\
frequency
latencytop
System Call Interface
mpstat
powertop
/proc/stat
GPU
/proc/cpuinfo
cpuid lscpu
lsof
pcstat
VFS
Sockets
Scheduler
funccount
funclatency
stackcount
kprobe
uprobe
argdist
trace
profile
df -h
File Systems
TCP/UDP
top ps
pidstat
turbostat
rdmsr
mdadm
Volume Manager
IP
Virtual
Memory
vmstat
slabtop free
/proc/meminfo
CPU
1
CPU
Interconnect
Linux Kernel
pidstat -d
Block Device Int.
Ethernet
Clocksource
memleak
oomkill
numactl
Memory
Bus
tiptop
perf
pcm
dmesg
Device Drivers
mdflush
Firmware
/sys/...
perf
tiptop
I/O Bus
DRAM
biotop bioscoop
biolatency bitesize
ss
tcpdump
ip
route
iptables
netstat
iptraf-ng
hardirqs
storcli
MegaCli
numastat
lstopo
Expander Interconnect
I/O Bridge
iostat
iotop
blktrace
I/O Controller
lspci
Interface Transports
Network Controller
nicstat
netstat
ip
Disk
Disk
Swap
Port
Port
FAN
Power
Supply
lsblk lsscsi blocked
smartctl fsdisk -l
swapon -s
/proc/swaps
ethtool
ifconfig
snmpget
lldptool
sar -m FAN
ipmitool
dmidecode
observability tools
static performance tools
perf-tools/bcc tracing tools
https://github.com/brendangregg/perf-tools https://github.com/iovisor/bcc
style inspired by reddit.com/u/redct
http://www.brendangregg.com/linuxperf.html 2016

Debugging - Syslog

- Syslog is a known standard for message logging. Most times, the system that does the logging and the software that gets to generate them tend to interfere during processes. But syslog helps separate the software generating the logs from the system that stores the logs, thereby making the process of logging less complicated and stressful.
  - **Syslog Daemon**: It is a daemon that listens for logs and writes them to a specific location. The location(s) is defined in the configuration file for the daemon. rsyslog is the Syslog daemon shipped with most of the distros.
  - **Syslog Message Format**: It refers to the syntax of Syslog messages. The syntax is usually defined by a standard (for eg RFC5424).
  - **Syslog Protocol**: It refers to the protocol used for remote logging. Modern Syslog daemons can use TCP and TLS in addition to UDP which is the legacy protocol for remote logging.

Debugging - Syslog

Benefits of syslog

- Helps analyze the root cause for any trouble or problem caused
- Reduce overall downtime helping to troubleshoot issues faster with all the logs
- Improves incident management by active detection of issues
- Self-determination of incidents along with auto resolution
- Simplified architecture with different level of severity like error,info,warning etc

Debugging - Syslog

Display syslogs with the ls command

Listing the contents of /var/log for an Ubuntu 20.04 machine using the ls command:

$ sudo ls /var/log

```
test@test-VirtualBox:~$ sudo ls /var/log
[sudo] password for test:
alternatives.log        auth.log.3.gz   dmesg             journal             syslog.5.gz
alternatives.log.1      auth.log.4.gz   dmesg.0           kern.log            syslog.6.gz
alternatives.log.2.gz   boot.log        dmesg.1.gz        kern.log.1          syslog.7.gz
apport.log              boot.log.1      dmesg.2.gz        kern.log.2.gz       ubuntu-advantage.log
apport.log.1            boot.log.2      dmesg.3.gz        kern.log.3.gz       ubuntu-advantage.log.1
apport.log.2.gz         boot.log.3      dmesg.4.gz        kern.log.4.gz       ubuntu-advantage.log.2.gz
apport.log.3.gz         boot.log.4      dpkg.log          lastlog             unattended-upgrades
apport.log.4.gz         boot.log.5      dpkg.log.1        openvpn             vboxadd-install.log
apport.log.5.gz         boot.log.6      dpkg.log.2.gz     private             vboxadd-setup.log
apport.log.6.gz         boot.log.7      faillog           speech-dispatcher   vboxadd-setup.log.1
apport.log.7.gz         bootstrap.log   fontconfig.log    syslog              vboxadd-setup.log.2
apt                     btmp            gdm3              syslog.1            vboxadd-setup.log.3
auth.log                btmp.1          gpu-manager.log   syslog.2.gz         vboxadd-setup.log.4
auth.log.1              cups            hp                syslog.3.gz         vboxadd-uninstall.log
auth.log.2.gz           dist-upgrade    installer         syslog.4.gz         wtmp
test@test-VirtualBox:~$
```

Debugging - Syslog

View system logs in Linux using the tail command

Using the tail command you can view the last few logs. Adding the -f option lets you watch them in real time.

$ sudo tail -f /var/log/syslog

Similarly, the tail command can be used to view kernel logs (kern.log), boot logs (boot.log), etc .

Debugging - Syslog

Syslog Configuration

The rules for which logs go where are defined in the Syslog daemon’s configuration file.

$ sudo vi /etc/rsyslog.conf

$ sudo vi /etc/rsyslog.d/50-default.conf

$ logger -p local1.info -t Testing "How are you"

```
#  Default rules for rsyslog.
#
#                       For more information see rsyslog.conf(5) and /etc/rsyslog.conf

#
# First some standard log files.  Log by facility.
#
auth,authpriv.*                 /var/log/auth.log
*.*;auth,authpriv.none          -/var/log/syslog
#cron.*                         /var/log/cron.log
#daemon.*                       -/var/log/daemon.log
kern.*                          -/var/log/kern.log
#lpr.*                          -/var/log/lpr.log
mail.*                          -/var/log/mail.log
#user.*                         -/var/log/user.log
local1.*                        -/var/log/test41.log
```

Debugging - Strace

strace is a powerful command line tool for debugging and trouble shooting programs in Unix-like operating systems such as Linux. It captures and records all system calls made by a process and the signals received by the process.

Trace Linux Command System Calls

You can simply run a command with strace like this, here we are tracing of all system calls made by the free command.

 # strace free

Debugging - Strace

Trace Linux Process PID

If a process is already running, you can trace it by simply passing its PID as follows; this will fill your screen with continues output that shows system calls being made by the process, to end it, press [Ctrl + C]

 # sudo strace -p 3569

Get Summary of Linux Process

Using the -c flag, you can generate a report of total time, calls, and errors for each system call, as follows.

 # strace –c free

Debugging - Strace

Print Instruction Pointer During System Call

The -i option displays the instruction pointer at the time of each system call made by the program.

 # sudo strace -i free

Show Time of Day For Each Trace Output Line

You can also print the time of day for each line in the trace output, by passing the -t flag.

 # sudo strace –t free

Debugging - Strace

Print Command Time Spent in System Calls

To shows the time difference between the starting and the end of each system call made by a program, use the -T option.

 # sudo strace –T free

Trace Only Specific System Calls

In the command below, trace=write is known as a qualifying expression, where trace is a qualifier (others include signal, abbrev, verbose, raw, read, or write). Here, write is the value of the qualifier.

Debugging - Strace

The following command actually shows the system calls to print df output on standard output.

 # sudo strace –e trace=write free

 # sudo strace -e trace=open,close free

 # sudo strace -e trace=open,close,read,write free

 # sudo strace -e trace=all free

Debugging - Strace

Trace System Calls Based on a Certain Condition

Let’s look at how to trace system calls relating to a given class of events. This command can be used to trace all system calls involving process management.

 # sudo strace -e trace=process free

Debugging - Strace

Trace System Calls Based on a Certain Condition

Let’s look at how to trace system calls relating to a given class of events. This command can be used to trace all system calls involving process management.

 # sudo strace -e trace=process free

Next, to trace all system calls that take a filename as an argument, run this command.

$ sudo strace -q -e trace=file free

Debugging - Strace

To trace all system calls involving memory mapping, type.

$ sudo strace -q -e trace=memory free

You can trace all network and signals related system calls.

$ sudo strace -e trace=network free

$ sudo strace -e trace=signal free

Debugging - Strace

Redirect Trace Output to File

To write the trace messages sent to standard error to a file, use the -o option. This means that only the command output is printed on the screen as shown below.

$ sudo strace -o df_debug.txt free

strace a program and threads

To trace both program and it threads using option -f

 # strace -f ./program

Debugging - Strace

strace a program and print strings

This will print the first 80 characters of every string.

 # strace -s 80 -f ./program

Debugging - GDB

Debuggers are programs which allow you to execute your program in a controlled manner, so you can look inside your program to find a bug.

gdb is a reasonably sophisticated text based debugger. It can let you:

Start your program, specifying anything that might affect its behavior.

Make your program stop on specified conditions.

Examine what has happened, when your program has stopped.

Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

SYNOPSIS

gdb [prog] [core|procID]

Debugging - GDB

GDB is invoked with the shell command gdb.

Once started,it reads commands from the terminal until you tell it to exit with the GDB command quit.

The most usual way to start GDB is with one argument or two, specifying an executable program as the argument:

 # gdb program

You can also start with both an executable program and a core file specified:

 # gdb program core

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

 # gdb program 1234

would attach GDB to process 1234

Debugging - GDB

To use gdb best, compile your program with:

gcc –g –c my_math.c

gcc –g –c sample.c

gcc –o sample my_math.o sample.o

or:

gcc –o sample -g my_math.c sample.c

That is, you should make sure that –g option is used to generate the .o files.

This option tells the compiler to insert more information about data types, etc., so the debugger gets a better understanding of it.

Debugging - GDB

Here are some of the most frequently needed GDB commands:

b(reak) [file:]function	Set a breakpoint at function (in file).
r(un) [arglist]	Start program (with arglist, if specified).
bt or where	Backtrace: display the program stack; especially useful to find where your program crashed or dumped core.
print expr	Display the value of an expression.
c	Continue running your program (after stopping, e.g. at a breakpoint).
n(ext)	Execute next program line (after stopping); step over any function calls in the line.

Debugging - GDB

s(tep) Execute next program line (after stopping); step into any function calls in the line.

help [name] Show information about GDB command name, or general information about using GDB.

q(uit) Exit from GDB.

l(ist) print the source code

Debugging – GDB commandline

 # gdb --args executablename arg1 arg2 arg3

Or

$ gdb executablename

(gdb) r arg1 arg2 arg3

Debugging – GDB Fork

set follow-fork-mode mode

Set the debugger response to a program call of fork or vfork. A call to fork or vfork creates a new process. The mode argument can be:

parent

The original process is debugged after a fork. The child process runs unimpeded. This is the default.

child

The new process is debugged after a fork. The parent process runs unimpeded.

show follow-fork-mode

Print state of follow-fork-mode

Debugging - GDB

set detach-on-fork mode

Tells gdb whether to detach one of the processes after a fork, or retain debugger control over them both.

on

The child process (or parent process, depending on the value of follow-fork-mode) will be detached and allowed to run independently. This is the default.

off

Both processes will be held under the control of GDB. One process (child or parent, depending on the value of follow-fork-mode) is debugged as usual, while the other is held suspended.

Debugging - GDB

(gdb) info inferior

(gdb) inferior

Debugging – GDB Thread

(gdb) info thread

(gdb) thread

Debugging – GDB coredump

You can also start with both an executable program and a core file specified:

 # gdb program core

Debugging – GDB Variable

(gdb) print variable

(gdb) print func::variable

print variable in different format

x

Regard the bits of the value as an integer, and print the integer in hexadecimal.

d

Print as integer in signed decimal.

u

Print as integer in unsigned decimal.

Debugging – GDB Variable

o

Print as integer in octal.

t

Print as integer in binary. The letter `t' stands for "two". (1)

a

Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

(gdb) p/a 0x54320

$3 = 0x54320 <_initialize_vx+396>

Debugging – GDB Variable

c

Regard as an integer and print it as a character constant.

f

Regard the bits of the value as a floating point number and print using typical floating point syntax.

Debugging – GDB Variable

(gdb) set var variable=90

(gdb) print &a

Type **info variables** to list "All global and static variable names".

Type **info locals** to list "Local variables of current stack frame" (names and values), including static variables in that function.

Type **info args** to list "Arguments of the current stack frame" (names and values).

Debugging – GDB Variable

(gdb) condition <breakpoint> <condition>

Example:

(gdb) break 28

(gdb) condition 2 i>5

- **watch**: gdb will break when a *write* occurs
- **rwatch**: gdb will break wnen a *read* occurs
- **awatch**: gdb will break in *both cases*

Debugging – Address Sanitizer

AddressSanitizer (ASan) is an instrumentation tool created by Google security researchers to identify memory access problems in C and C++ programs.

When the source code of a C/C++ application is compiled with AddressSanitizer enabled, the program will be instrumented at runtime to identify and report memory access errors.

Debugging – Address Sanitizer

Memory access errors and AddressSanitizer

C and C++ are very insecure and error-prone languages. And one of the main sources of problems is memory access errors.

Different kind of bugs in the source code could trigger a memory access error, including:

- **Buffer overflow or buffer overrun** occurs when a program overruns a buffer's boundary and overwrites adjacent memory locations.
- **Stack overflow** is when a program crosses the boundary of function's stack.
- **Heap overflow** is when a program overruns a buffer allocated in the heap.
- **Memory leak** is when a program allocates memory but does not deallocate.
- **Use after free (dangling pointer)** is when a program uses memory regions already deallocated.

Debugging – Address Sanitizer

- **Uninitialized variable** is when a program reads a memory location before it is initialized.

All these errors are due to programming bugs. They could prevent the application from executing, cause invalid results or expose a vulnerability that could be exploited by a malicious actor. They are usually very hard to reproduce, debug and fix.

- **Heap overflow** is when a program overruns a buffer allocated in the heap.
- **Memory leak** is when a program allocates memory but does not deallocate.
- **Use after free (dangling pointer)** is when a program uses memory regions already deallocated.

- **Uninitialized variable** is when a program reads a memory location before it is initialized.

All these errors are due to programming bugs. They could prevent the application from executing, cause invalid results or expose a vulnerability that could be exploited by a malicious actor. They are usually very hard to reproduce, debug and fix.

Debugging – Address Sanitizer

-fsanitize=address

Debugging – Thread Sanitizer

ThreadSanitizer is a tool that detects data races. It consists of a compiler instrumentation module and a run-time library. Typical slowdown introduced by ThreadSanitizer is about 5x-15x. Typical memory overhead introduced by ThreadSanitizer is about 5x-10x.

-fsanitize=thread

Debugging – Valgrind

Valgrind is a multipurpose code profiling and memory debugging tool for Linux when on the x86 and, as of version 3, AMD64, architectures. It allows you to run your program in Valgrind's own environment that monitors memory usage such as calls to malloc and free (or new and delete in C++). If you use uninitialized memory, write off the end of an array, or forget to free a pointer, Valgrind can detect it. Since these are particularly common problems, this tutorial will focus mainly on using Valgrind to find these types of simple memory problems, though Valgrind is a tool that can do a lot more.

 # valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 ./test

Debugging – Valgrind

Callgrind

valgrind Callgrind is a program that can profile your code and report on its resources usage. It is another tool provided by Valgrind, which also helps detect memory issues.

 # valgrind --tool=callgrind program-to-run program-arguments

 # callgrind_annotate --auto=yes callgrind.out.pid

Debugging – Valgrind

helgrind

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.

The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Debugging – Valgrind

helgrind

Helgrind can detect three classes of errors, which are discussed in detail in the next three sections:

- Misuses of the POSIX pthreads API.
- Potential deadlocks arising from lock ordering problems.
- Data races -- accessing memory without adequate locking or synchronisation.

 # valgrind --tool=helgrind program_name

Debugging – Valgrind

massif

Massif is a heap profiler. It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s), although it does not do so by default.

Debugging – Valgrind

massif

Heap profiling can help you reduce the amount of memory your program uses. On modern machines with virtual memory, this provides the following benefits:

- It can speed up your program -- a smaller program will interact better with your machine's caches and avoid paging.

- If your program uses lots of memory, it will reduce the chance that it exhausts your machine's swap space.

 # valgrind --tool=massif --time-unit=B prog

 # ms_print massif.out.pid

Bash Introduction

What is Shell?

- The shell is a command interpreter.
- It is the layer between the operating system kernel and the user.

Bash Setup

- On the first line
  - #!/bin/bash
  - Tells the operating system that the following will be a script and not a regular text file
  - You add - to the end to prevent users from spoofing the interpreter
- Change the file to give it execute permissions
  - chmod +x test.sh

Bash working

- **How scripts work**

#!/bin/bash

echo ”Hello”

ls -al

 # /bin/bash echo ”Hello”

 # /bin/bash ls -al

Bash Comments

- ‘#’ symbol is used to add single line comment in bash script.

```
#!/bin/bash

# Add two numeric value
((sum=25+35))

#Print the result
echo $sum
```

- You can use multi line comment in bash in various ways. Use : ' to open and ' to close.

```
#!/bin/bash
: '
The following script calculates
the square value of the number, 5.
'

((area=5*5))
echo $area
```

Bash Command - Echo

- You can use echo command with various options.

```
echo [ OPTION(S) ] [ STRING ]
```

Options:

Option	Description
-n	Omit newline from the output.
-e	Enable the function of backslash(/) character.
-E	Disable the function of backslash(/) character.
–version	Display the version information
–help	Display help messages related to the uses of this command

Bash Command - Echo

When you use ‘echo’ command without any option then a newline is added by default.

‘-n’ option is used to print any text without new line and

‘-e’ option is used to remove backslash characters from the output.

```
#!/bin/bash
echo "Printing text with newline"
echo -n "Printing text without newline"
echo -e "\nRemoving \t backslash \t characters\n"
```

Bash Command - Echo

‘echo’ command is used with ‘-E’ option in the following script. This option disables the function of backslash(/). The new line(\n) used in the text will not work for the following command.‘echo’ command is used with ‘-E’ option in the following script. This option disables the function of backslash(/). The new line(\n) used in the text will not work for the following command.

```
$ echo -E "I\nlike\nbash\programming"
```

Bash Command - Printf

The “printf” command in bash scripting works the same way “printf ()” works in the C language.

```
printf <var> <formate> <arguments…>
```

<var> : It is optional to assign output to a variable.

<formate> : It is a string that may contain different format specifiers such as “%s”, “%b”, “%d”, “%f”, and backslash escape

<arguments…> : The arguments can be any value or variable

Specifier	Description
%%	Prints “%” symbol
%c	Takes arguments as a single character
%e and %E	Take argument in floating-point number and prints in exponential notation, **%e** for lower case letter and **%E** for capital letter
%g and %G	Take argument in floating-point number and prints in normal or exponential notation
%f	Takes argument as floating numbers
%d	Takes arguments as signed integers
%u	Takes argument as unsigned integers
%o	Takes argument as an unsigned octal number
%x and %X	Takes arguments as unsigned hexadecimal integers

Bash Read

Use to get input (data from user) from keyboard and store (data) to variable.

Syntax:

***read* variable1, variable2,...variable**

```
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Bash Read

To catch a line of input NAMEs and options are not required by read. When NAME is not specified, a variable named REPLY is used to store user input.

```
{
echo -n "Type something and press enter: ";
read;
echo You typed ${REPLY}
}
```

To catch a word of input, the -d option is required. In the case of a word we would set -d to a space, read ‘-d ‘. That is when the user presses the space bar read will load REPLY with the word.

```
{
echo -n "Type something and hit space: ";
read '-d ';
echo "";
echo "You typed ${REPLY}"
}
```

Bash Read

The -p option allows a message to be displayed before reading from standard input.

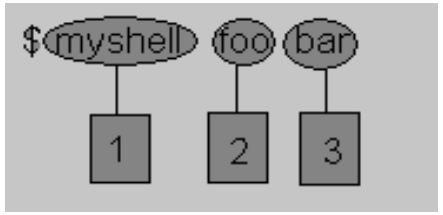
```
{
read -p 'Type something and hit space: ' '-d ';
echo "";
echo "You typed ${REPLY}"
}
```

When catching user input without it showing up int the terminal, the -s option comes in handy. read -s -p allows you to catch and hide user input

```
{
read -s -p 'Type something I promise to keep it a secret: '
echo "";
echo "Your secret is safe with me" ; unset REPLY ;
echo "${REPLY}"
}
```

Bash Arguments

$ myshell foo bar



1 Shell Script name i.e. **myshell**

2 First command line argument passed to **myshell** i.e. ***foo***

3 Second command line argument passed to **myshell** i.e. **bar**

In shell if we wish to refer this command line argument we refer above as follows

1 **myshell it is $0**

2 **foo it is $1**

3 **bar it is $2**

- Here **$#** (built in shell variable) will be 2 (Since ***foo*** and **bar** only two Arguments),
- Please note at a time such 9 arguments can be used from $1..$9,
- You can also refer all of them by using $* (which expand to `$1,$2...$9`).
- Note that $1..$9 i.e command line arguments to shell script is know as "*positional parameters*".

Bash Arguments

```
#!/bin/bash
#
# Script that demos, command line args
#
echo "Total number of command line argument are $#"
echo "$0 is script name"
echo "$1 is first argument"
echo "$2 is second argument"
echo "All of them are :- $* or $@
```

Bash Redirection

- In Linux (and in other OSs also) it's possible to send output to file or to read input from a file

For e.g.

$ *ls* command gives output to screen;

- to send output to file of ls command give command

  ***ls* > filename**

  It means put output of ***ls*** command to filename.

Bash Redirection

There are three main redirection symbols: >,>>,<

(1) > Redirector Symbol

Syntax:

Linux-command > filename

To output Linux-commands result to file. Note that if the file already exist, it will be overwritten else a new file will be created.

For e.g.

To send output of ***ls*** command give

$ *ls* > myfiles

if '**myfiles**' exist in your current directory it will be overwritten without any warning.

Bash Redirection

(2) **>>** Redirector Symbol

Syntax:

Linux-command >> filename

To output Linux-commands result to the END of the file.

if file exist: it will be opened new information/data will be written to the **END** of the file, without losing previous information/data, if file does not exist, a new file is created.

For e.g.

To send output of date command

to already exist file give command

$ *date* >> myfiles

Bash Redirection

(3) **<** Redirector Symbol

Syntax:

Linux-command < filename

To provide input to Linux-command from the file instead of standart input (key-board).

For e.g. To take input for cat command give

$ *cat* < myfiles

Bash Pipe

A pipe is a way to connect the output of one program to the input of another program without any temporary file.

Definition

"*A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands Multiple commandsfrom same command line.*“

Syntax:

command1 | command2

Bash Pipe

Command using Pipes	Meaning or Use of Pipes
$ ***ls | more***	Output of ***ls*** command is given as input to the command ***more*** So output is printed one screen full page at a time.
$ ***who | sort***	Output of ***who*** command is given as input to sort command So it will print sorted list of users
$ ***who | sort > user_list***	Same as above except output of sort is send to (redirected) the file named user_list
$ ***who | wc -l***	***who*** command provides the input of ***wc*** command So it will count the users logged in.
$ ***ls -l | wc -l***	***ls*** command provides the input of ***wc*** command So it will count files in current directory.
$ ***who | grep raju***	Output of ***who*** command is given as input to ***grep*** command So it will print if particular user is logged in. Otherwise nothing is printed

Bash if

A basic if statement effectively says, **if** a particular test is true, then perform a given set of actions. If it is not true then don't perform those actions. If follows the format below:

```
if [ <some test> ]
then
    <commands>
fi
```

Anything between **then** and **fi** (if backwards) will be executed only if the test (between the square brackets) is true.

Bash if

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [5 -ge 6]

Bash if

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

Bash if

Test	Meaning
-s file	Non empty file
-f file	File exists or is a normal file and not a directory
-d dir	Directory exists and not a file
-w file	file is a writeable file
-r file	file is a read-only file
-x file	file is executable

Bash if

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND
expression1 -o expression2	Logical OR

Bash if..else..fi

Sometimes we want to perform a certain set of actions if a statement is true, and another set of actions if it is false. We can accommodate this with the **else** mechanism.

```
if [ <some test> ]
then
    <commands>
else
    <other commands>
fi
```

Bash if..elif..fi

Sometimes we may have a series of conditions that may lead to different paths.

```
if [ <some test> ]
then
    <commands>
elif [ <some test> ]
then
    <different commands>
else
    <other commands>
fi
```

Bash case

Sometimes we may wish to take different paths based upon a variable matching a series of patterns. We could use a series of **if** and **elif** statements but that would soon grow to be unweildly. Fortunately there is a **case** statement which can make things cleaner. It's a little hard to explain so here are some examples to illustrate:

```
case <variable> in
<pattern 1>)
    <commands>
    ;;
<pattern 2>)
    <other commands>
    ;;
esac
```

Bash loops

Bash supports:

for loop

while loop

Note that in each and every loop,

(a) First, the variable used in loop condition

- must be initialized,
- then execution of the loop begins.

(b) A test (condition) is made

at the beginning of each iteration.

(c) The body of loop ends

with a statement modifies

the value of the test (condition) variable.

for Loop

Syntax:

for *{ variable name } in { list }*

do

execute one for each item in the list

until the list is not finished

(And repeat all statements between do and done)

done

while Loop

Syntax:

while *[condition]*

do

command1

command2

..

....

done

Bash until

The **until** loop is fairly similar to the while loop. The difference is that it will execute the commands within it until the test becomes true.

```
until [ <some test> ]
do
    <commands>
done
```

Bash break

The **break** statement tells Bash to leave the loop straight away. It may be that there is a normal situation that should cause the loop to end but there are also exceptional situations in which it should end as well. For instance, maybe we are copying files but if the free disk space get's below a certain level we should stop copying.

Bash Continue

The **continue** statement tells Bash to stop running through this iteration of the loop and begin the next iteration. Sometimes there are circumstances that stop us from going any further. For instance, maybe we are using the loop to process a series of files but if we happen upon a file which we don't have the read permission for we should not try to process it.

Bash Function

A Bash function is essentially a set of commands that can be called numerous times. The purpose of a function is to help you make your bash scripts more readable and to avoid writing the same code repeatedly. Compared to most programming languages, Bash functions are somewhat limited.

The first format starts with the function name, followed by parentheses. This is the preferred and more used format.

```
function_name () {
  commands
}
```

Bash Function

The second format starts with the reserved word function, followed by the function name.

```
function function_name {
  commands
}
```

Bash Function variable

Global variables are variables that can be accessed from anywhere in the script regardless of the scope. In Bash, all variables by default are defined as global, even if declared inside the function.

Local variables can be declared within the function body with the local keyword and can be used only inside that function. You can have local variables with the same name in different functions.

Bash function return value

Unlike functions in “real” programming languages, Bash functions don’t allow you to return a value when called. When a bash function completes, its return value is the status of the last statement executed in the function, 0 for success and non-zero decimal number in the 1 - 255 range for failure.

The return status can be specified by using the return keyword, and it is assigned to the variable $?. The return statement terminates the function. You can think of it as the function’s exit status .

To actually return an arbitrary value from a function, we need to use other methods. The simplest option is to assign the result of the function to a global variable.

Another, better option to return a value from a function is to send the value to stdout using echo or printf.

Bash function arguments

To pass any number of arguments to the bash function simply put them right after the function's name, separated by a space. It is a good practice to double-quote the arguments to avoid the misparsing of an argument with spaces in it.

- The passed parameters are $1, $2, $3 … $n, corresponding to the position of the parameter after the function's name.
- The $0 variable is reserved for the function's name.
- The $# variable holds the number of positional parameters/arguments passed to the function.
- The $* and $@ variables hold all positional parameters/arguments passed to the function.

SED

SED command in UNIX is stands for stream editor and it can perform lot's of function on file like, searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening it, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

Syntax:

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

SED

The essential command: s for substitution

Sed has several commands, but most people only learn the substitute command: s. The substitute command changes all occurrences of the regular expression into a new value. A simple example is changing "day" in the "old" file to "night" in the "new" file:

```
# echo "day day day" | sed 's/day/night/’
# echo "day day day" | sed 's/day/night/g'
```

SED

 # echo Sunday | sed 's/day/night/’

Delete a file line starting with & ending with a pattern

To delete a line starting with a particular string & ending with another string, use

 # sed -e 's/^danger.*stops$//g' testfile.txt

This will delete the line with ‘danger’ on start & ‘stops’ in the end & it can have any number of words in between , ‘.*’ defines that part.

SED

Replacing or substituting string : Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word “unix” with “linux” in the file.

$ sed 's/unix/linux/' sample.txt

Here the “s” specifies the substitution operation. The “/” are delimiters. The “unix” is the search pattern and the “linux” is the replacement string.

By default, the sed command replaces the first occurrence of the pattern in each line and it won’t replace the second, third…occurrence in the line.

SED

Replacing the nth occurrence of a pattern in a line : Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word “unix” with “linux” in a line.

$ sed 's/unix/linux/2' sample.txt

SED

Replacing all the occurrence of the pattern in a line : The substitute flag /g (global replacement) specifies the sed command to replace all the occurrences of the string in the line.

$ sed 's/unix/linux/g' sample.txt

SED

Replacing from nth occurrence to all occurrences in a line : Use the combination of /1, /2 etc and /g to replace all the patterns from the nth occurrence of a pattern in a line. The following sed command replaces the third, fourth, fifth… “unix” word with “linux” word in a line.

$ sed 's/unix/linux/3g' sample.txt

SED

The slash as a delimiter

The character after the s is the delimiter. It is conventionally a slash, because this is what ed, more, and vi use. It can be anything you want, however. If you want to change a pathname that contains a slash - say /usr/local/bin to /common/bin - you could use the backslash to quote the slash:

```
# sed 's/\/usr\/local\/bin/\/common\/bin/' <old >new

# sed 's_/usr/local/bin_/common/bin_' <old >new
# sed 's:/usr/local/bin:/common/bin:' <old >new
# sed 's|/usr/local/bin|/common/bin|' <old >new
```

SED

Using & as the matched string

Sometimes you want to search for a pattern and add some characters, like parenthesis, around or near the pattern you found. It is easy to do this if you are looking for a particular string:

```
# sed 's/abc/(abc)/' <old >new
```

```
# sed 's/[a-z]*/(&)/' <old >new
```

```
# echo "123 abc" | sed 's/[0-9]*/& &/’
```

```
# echo "123 abc" | sed 's/[0-9][0-9]*/& &/'
```

SED

Using \1 to keep part of the pattern

The escaped parentheses (that is, parentheses with backslashes before them) remember a substring of the characters matched by the regular expression. You can use this to exclude part of the characters matched by the regular expression. The "\1" is the first remembered pattern, and the "\2" is the second remembered pattern. Sed has up to nine remembered patterns.

If you wanted to keep the first word of a line, and delete the rest of the line, mark the important part with the parenthesis:

 # sed 's/\([a-z]*\).*/\1/’

"[a-z]*" matches zero or more lower case letters, and tries to match as many characters as possible. The ".*" matches zero or more characters after the first match. Since the first one grabs all of the contiguous lower case letters, the second matches anything else.

 # echo abcd123 | sed 's/\([a-z]*\).*/\1/'

SED

Parenthesize first character of each word : This sed example prints the first character of every word in parenthesis.

$ echo "Welcome To The Geek Stuff" | sed 's/\(\b[A-Z]\)/\(\1\)/g'

SED

Replacing string on a specific line number : You can restrict the sed command to replace the string on a specific line number. An example is

$ sed '3 s/unix/linux/' sample.txt

SED

Deleting lines from a particular file : SED command can also be used for deleting lines from a particular file. SED command is used for performing deletion operation without even opening the file

Examples:

1. To Delete a particular line say n in this example

Syntax:

$ sed 'nd' filename.txt

Example:

$ sed '5d' filename.txt

SED

- To Delete a last line

  Syntax:

  $ sed '$d' filename.txt

- To Delete line from range x to y

  Syntax:

  $ sed 'x,yd' filename.txt

  Example:

  $ sed '3,6d' filename.txt

SED

5. To Delete from nth to last line

Syntax:

$ sed 'nth,$d' filename.txt

Example:

$ sed '12,$d' filename.txt

6. To Delete pattern matching line

Syntax:

$ sed '/pattern/d' filename.txt

Example:

$ sed '/abc/d' filename.txt

SED

/p - print

By default, sed prints every line. If it makes a substitution, the new text is printed instead of the old one. If you use an optional argument to sed, "sed -n," it will not, by default, print any new lines. I'll cover this and other options later. When the "-n" option is used, the "p" flag will cause the modified line to be printed. Here is one way to duplicate the function of grep with sed:

 # sed -n 's/pattern/&/p' <file

SED

The sed has two buffers, the pattern space and the hold space. Both are initially empty. The following commands manipulate the hold space:

command	description
h	Replace the contents of the hold space with the contents of the pattern space.
H	Append a newline to the contents of the hold space, and then append the contents of the pattern space to that of the hold space.
g	Replace the contents of the pattern space with the contents of the hold space.
G	Append a newline to the contents of the pattern space, and then append the contents of the hold space to that of the pattern space.
x	Exchange the contents of the hold and pattern spaces.

SED

```
# printf '%s\n' one two  | sed --debug 'H;1h;$x;$s/\n/_/g’
```

SED

Here, to delete three lines, we have specified three separate commands with -e option.

```
# sed -e '1d' -e '2d' -e '5d' books.txt
```

```
# sed -n '
h;n;H;x
s/\n/, /
/Paulo/!b Print
s/^/- /
:Print
p' books.txt
```

AWK Introduction

AWK is a pattern-directed scanning and processing language. An AWK program consists of a set of actions to be taken against streams of textual data. AWK extensively uses regular expressions. It is a standard feature of most Unix-like operating systems.

AWK was created at Bell Labs in the 1977. Its name is derived from the family names of its authors – Alfred Aho, Peter Weinberger, and Brian Kernighan.

There are two major implementations of AWK. The traditional Unix AWK and the newer GAWK. GAWK is the GNU Project's implementation of the AWK programming language. GAWK has several extensions to the original AWK.

The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line.

Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then performs the associated actions.

AWK Introduction

WHAT CAN WE DO WITH AWK?

1. AWK Operations:

(a) Scans a file line by line

(b) Splits each input line into fields

(c) Compares input line/fields to pattern

(d) Performs action(s) on matched lines

2. Useful For:

(a) Transform data files

(b) Produce formatted reports

3. Programming Constructs:

(a) Format output lines

(b) Arithmetic and string operations

(c) Conditionals and loops

AWK Syntax

 # awk options program file

Awk can take the following options:

-F fs : To specify a file separator.

-f file: To specify a file that contains awk script.

-v var=value: To declare a variable.

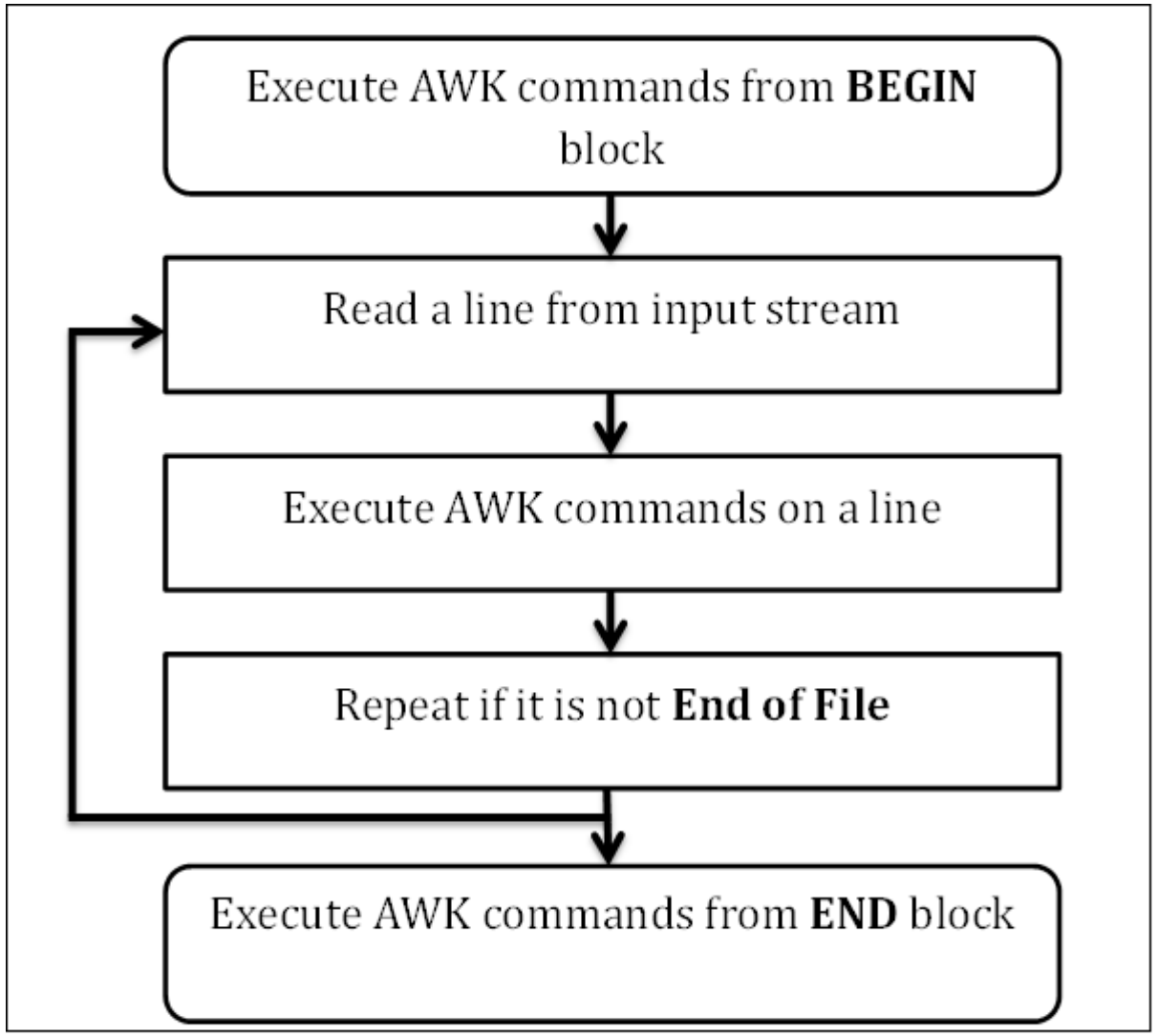
The structure of an AWK program has the following form:

pattern { action }

Example: awk '{print}' employee.txt

AWK Workflow

AWK follows a simple workflow − Read, Execute, and Repeat.



AWK Workflow

Read

AWK reads a line from the input stream (file, pipe, or stdin) and stores it in memory.

Execute

All AWK commands are applied sequentially on the input. By default AWK execute commands on every line. We can restrict this by providing patterns.

Repeat

This process repeats until the file reaches its end.

BEGIN Block

<u>BEGIN block</u>

The syntax of the BEGIN block is as follows:

Syntax

BEGIN {awk-commands}

The BEGIN block gets executed at program start-up. It executes only once. This is good place to initialize variables. BEGIN is an AWK keyword and hence it must be in upper-case. This block is optional.

BODY Block

Body Block

The syntax of the body block is as follows:

Syntax

/pattern/ {awk-commands}

The body block applies AWK commands on every input line. By default, AWK executes commands on every line. We can restrict this by providing patterns. Note that there are no keywords for the Body block.

END Block

END Block

The syntax of the END block is as follows:

Syntax

END {awk-commands}

The END block executes at the end of the program. END is an AWK keyword and hence it must be in upper-case. This block is optional.

BEGIN Example

```
# awk 'BEGIN{printf "Name\tRole\tDepartment\tSalary\n"} {print}' employee.txt
```

-f option

 # awk -f command.awk marks.txt

-v option

 # awk -f command.awk marks.txt

-F option

Sometimes the separator in some files is not space nor tab but something else. You can specify it using –F option:

 # awk -F: '{print $1}' /etc/passwd

AWK Variables

With awk, you can process text files. Awk assigns some variables for each data field found:

$0 for the whole line.

$1 for the first field.

$2 for the second field.

$n for the nth field.

The whitespace character like space or tab is the default separator between fields in awk.

 # awk '{print $1}’ employee.txt

Multiple commands

To run multiple commands, separate them with a semicolon like this:

```
# echo "Hello Tom" | awk '{$2="Adam"; print $0}'
```

Builtin Variables

Awk's built-in variables include the field variables:

NR: NR command keeps a current count of the number of input records. Remember that records are usually lines. Awk command performs the pattern/action statements once for each record in a file.

NF: NF command keeps a count of the number of fields within the current input record.

FS: FS command contains the field separator character which is used to divide fields on the input line. The default is "white space", meaning space and tab characters. FS can be reassigned to another character (typically in BEGIN) to change the field separator.

RS: RS command stores the current record separator character. Since, by default, an input line is the input record, the default record separator character is a newline.

Builtin Variables

OFS: OFS command stores the output field separator, which separates the fields when Awk prints them. The default is a blank space. Whenever print has several parameters separated with commas, it will print the value of OFS in between each parameter.

ORS: ORS command stores the output record separator, which separates the output lines when Awk prints them. The default is a newline character. print automatically outputs the contents of ORS at the end of whatever it is given to print.

;display line number along with entire line

 # awk '{print NR,$0}' employee.txt

;display first and last string

 # awk '{print $1,$NF}' employee.txt

Builtin Variables

OFS: OFS command stores the output field separator, which separates the fields when Awk prints them. The default is a blank space. Whenever print has several parameters separated with commas, it will print the value of OFS in between each parameter.

ORS: ORS command stores the output record separator, which separates the output lines when Awk prints them. The default is a newline character. print automatically outputs the contents of ORS at the end of whatever it is given to print.

;display line number along with entire line

 # awk '{print NR,$0}' employee.txt

;display first and last string

 # awk '{print $1,$NF}' employee.txt

Builtin Variables

;Display lines between 3 to 6

 # awk 'NR==3, NR==6 {print NR,$0}' employee.txt

Builtin Variables

ARGC

It implies the number of arguments provided at the command line.

```
# awk 'BEGIN {print "Arguments =", ARGC}' One Two Three Four
```

```
# awk 'BEGIN {
   for (i = 0; i < ARGC - 1; ++i) {
      printf "ARGV[%d] = %s\n", i, ARGV[i]
   }
}' one two three four
```

Builtin Variables

CONVFMT

It represents the conversion format for numbers. Its default value is %.6g.

 # awk 'BEGIN { print "Conversion Format =", CONVFMT }’

ENVIRON

It is an associative array of environment variables.

 # awk 'BEGIN { print ENVIRON["USER"] }'

Builtin Variables

FILENAME

It represents the current file name.

 # awk 'END {print FILENAME}’ employee.txt

FS

It represents the (input) field separator and its default value is space. You can also change this by using -F command line option.

 # awk 'BEGIN {print "FS = " FS}' | cat -vte

Builtin Variables

NF

It represents the number of fields in the current record. For instance, the following example prints only those lines that contain more than two fields.

```
# echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NF > 2’
```

NR

It represents the number of the current record. For instance, the following example prints the record if the current record number is less than three.

```
# echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NR < 3’
```

Builtin Variables

FNR

It is similar to NR, but relative to the current file. It is useful when AWK is operating on multiple files. Value of FNR resets with new file.

OFMT

It represents the output format number and its default value is %.6g.

 # awk 'BEGIN {print "OFMT = " OFMT}’

OFS

It represents the output field separator and its default value is space.

 # awk 'BEGIN {print "OFS = " OFS}' | cat -vte

Builtin Variables

ORS

It represents the output record separator and its default value is newline.

 # awk 'BEGIN {print "ORS = " ORS}' | cat -vte

RLENGTH

It represents the length of the string matched by match function. AWK's match function searches for a given string in the input-string.

 # awk 'BEGIN { if (match("One Two Three", "re")) { print RLENGTH } }'

Builtin Variables

RS

It represents (input) record separator and its default value is newline.

 # awk 'BEGIN {print "RS = " RS}' | cat -vte

RSTART

It represents the first position in the string matched by match function.

 # awk 'BEGIN { if (match("One Two Three", "Thre")) { print RSTART } }’

Builtin Variables

SUBSEP

It represents the separator character for array subscripts and its default value is \034.

```
# awk 'BEGIN { print "SUBSEP = " SUBSEP }' | cat -vte
```

$0

It represents the entire input record.

```
# awk '{print $0}' marks.txt
```

Builtin Variables

$n

It represents the nth field in the current record where the fields are separated by FS.

 # awk '{print $3 "\t" $4}' marks.txt

AWK Operators

AWK also provides a large set of operator:

- Arithmetic Operators
- Increment and Decrement Operators
- Assignment Operators
- Relational Operators
- Logical Operators
- Ternary Operator
- Unary Operators
- Exponential Operators
- String Concatenation Operator
- Array Membership Operator
- Regular Expression Operators

Arthematic Operators

Addition

It is represented by plus (+) symbol which adds two or more numbers.

 # awk 'BEGIN { a = 50; b = 20; print "(a + b) = ", (a + b) }’

Subtraction

It is represented by minus (-) symbol which subtracts two or more numbers.

 # awk 'BEGIN { a = 50; b = 20; print "(a - b) = ", (a - b) }'

Arthematic Operators

Multiplication

It is represented by asterisk (*) symbol which multiplies two or more numbers.

 # awk 'BEGIN { a = 50; b = 20; print "(a * b) = ", (a * b) }’

Division

It is represented by slash (/) symbol which divides two or more numbers.

 # awk 'BEGIN { a = 50; b = 20; print "(a / b) = ", (a / b) }’

Arthematic Operators

Modulus

It is represented by percent (%) symbol which finds the Modulus division of two or more numbers.

```
# awk 'BEGIN { a = 50; b = 20; print "(a % b) = ", (a % b) }'
```

Exponent Operators

There are two formats of exponential operators:

Exponential Format 1

It is an exponential operator that raises the value of an operand.

```
# awk 'BEGIN { a = 10; a = a ^ 2; print "a =", a }'
```

Exponential Format 2

It is an exponential operator that raises the value of an operand.

```
# awk 'BEGIN { a = 10; a = a**2; print "a =", a }'
```

Increment and Decrement Operators

AWK supports the following increment and decrement operators:

Pre-Increment

It is represented by ++. It increments the value of an operand by 1. This operator first increments the value of the operand, then returns the incremented value.

```
# awk 'BEGIN { a = 10; b = ++a; printf "a = %d, b = %d\n", a, b }’
```

Pre-Decrement

It is represented by --. It decrements the value of an operand by 1. This operator first decrements the value of the operand, then returns the decremented value.

```
# awk 'BEGIN { a = 10; b = --a; printf "a = %d, b = %d\n", a, b }'
```

Increment and Decrement Operators

Post-Increment

It is represented by ++. It increments the value of an operand by 1. This operator first returns the value of the operand, then it increments its value.

```
# awk 'BEGIN { a = 10; b = a++; printf "a = %d, b = %d\n", a, b }’
```

Post-Decrement

It is represented by --. It decrements the value of an operand by 1. This operator first returns the value of the operand, then it decrements its value.

```
# awk 'BEGIN { a = 10; b = a--; printf "a = %d, b = %d\n", a, b }'
```

Assignment Operators

AWK supports the following assignment operators:

Simple Assignment

It is represented by =.

```
# awk 'BEGIN { name = "Jerry"; print "My name is", name }'
```

Shorthand Addition

It is represented by +=.

```
# awk 'BEGIN { cnt = 10; cnt += 10; print "Counter =", cnt }'
```

Assignment Operators

Shorthand Subtraction

It is represented by -=.

```
# awk 'BEGIN { cnt = 100; cnt -= 10; print "Counter =", cnt }'
```

Shorthand Multiplication

It is represented by *=.

```
#  awk 'BEGIN { cnt = 10; cnt *= 10; print "Counter =", cnt }'
```

Assignment Operators

Shorthand Division

It is represented by /=.

```
# awk 'BEGIN { cnt = 100; cnt /= 5; print "Counter =", cnt }'
```

Shorthand Modulo

It is represented by %=.

```
# awk 'BEGIN { cnt = 100; cnt %= 8; print "Counter =", cnt }'
```

Assignment Operators

Shorthand Exponential

It is represented by ^=.

 # awk 'BEGIN { cnt = 2; cnt ^= 4; print "Counter =", cnt }'

Shorthand Exponential

It is represented by **=.

 # awk 'BEGIN { cnt = 2; cnt **= 4; print "Counter =", cnt }'

Assignment Operators

Shorthand Exponential

It is represented by ^=.

 # awk 'BEGIN { cnt = 2; cnt ^= 4; print "Counter =", cnt }'

Shorthand Exponential

It is represented by **=.

 # awk 'BEGIN { cnt = 2; cnt **= 4; print "Counter =", cnt }'

Relationship Operators

AWK supports the following relational operators:

Equal to

It is represented by ==. It returns true if both operands are equal, otherwise it returns false.

```
# awk 'BEGIN { a = 10; b = 10; if (a == b) print "a == b" }'
```

Not Equal to

It is represented by !=. It returns true if both operands are unequal, otherwise it returns false.

```
#  awk 'BEGIN { a = 10; b = 20; if (a != b) print "a != b" }'
```

Relationship Operators

Less Than

It is represented by <. It returns true if the left-side operand is less than the right-side operand; otherwise it returns false.

```
# awk 'BEGIN { a = 10; b = 20; if (a < b) print "a  < b" }'
```

Less Than or Equal to

It is represented by <=. It returns true if the left-side operand is less than or equal to the right-side operand; otherwise it returns false.

```
#  awk 'BEGIN { a = 10; b = 10; if (a <= b) print "a <= b" }'
```

Relationship Operators

Greater Than

It is represented by >. It returns true if the left-side operand is greater than the right-side operand, otherwise it returns false.

```
# awk 'BEGIN { a = 10; b = 20; if (b > a ) print "b > a" }'
```

Greater Than or Equal to

It is represented by >=. It returns true if the left-side operand is greater than or equal to the right-side operand; otherwise it returns false.

```
b >= a
```

Logical Operators

AWK supports the following logical operators:

Logical AND

It is represented by &&. Its syntax is as follows:

Syntax

expr1 && expr2

It evaluates to true if both expr1 and expr2 evaluate to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to true.

```
# awk 'BEGIN {
  num = 5; if (num >= 0 && num <= 7) printf "%d is in octal format\n", num
}'
```

Logical Operators

Logical OR

It is represented by ||. The syntax of Logical OR is:

Syntax

expr1 || expr2

It evaluates to true if either expr1 or expr2 evaluates to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to false.

```
# awk 'BEGIN {
   ch = "\n"; if (ch == " " || ch == "\t" || ch == "\n")
   print "Current character is whitespace."
}'
```

Logical Operators

Logical NOT

It is represented by exclamation mark (!). The Syntax of logical NOT is:

Syntax

! expr1

It returns the logical compliment of expr1. If expr1 evaluates to true, it returns 0; otherwise it returns 1.

 # awk 'BEGIN { name = ""; if (! length(name)) print "name is empty string." }'

Logical Operators

Logical NOT

It is represented by exclamation mark (!). The Syntax of logical NOT is:

Syntax

! expr1

It returns the logical compliment of expr1. If expr1 evaluates to true, it returns 0; otherwise it returns 1.

 # awk 'BEGIN { name = ""; if (! length(name)) print "name is empty string." }'

Ternary Operators

We can easily implement a condition expression using ternary operator.

Syntax

condition expression ? statement1 : statement2

When the condition expression returns true, statement1 gets executed; otherwise statement2 is executed.

 # awk 'BEGIN { a = 10; b = 20; (a > b) ? max = a : max = b; print "Max =", max}'

Uniary Operators

AWK supports the following unary operators:

Unary Plus

It is represented by +. It multiplies a single operand by +1.

```
# awk 'BEGIN { a = -10; a = +a; print "a =", a }'
```

Unary Minus

It is represented by -. It multiplies a single operand by -1.

```
# awk 'BEGIN { a = -10; a = -a; print "a =", a }'
```

String concatenation Operators

Space is a string concatenation operator that merges two strings.

```
# awk 'BEGIN { str1 = "Hello, "; str2 = "World"; str3 = str1 str2; print str3 }'
```

Array membership Operators

It is represented by **in**. It is used while accessing array elements.

```
# awk 'BEGIN {
   arr[0] = 1; arr[1] = 2; arr[2] = 3; for (i in arr) printf "arr[%d] = %d\n", i, arr[i]
}'
```

Regular Expression Operators

There two forms of regular expressions operators.

Match

It is represented as ~. It looks for a field that contains the match string.

 # awk '$0 ~ "manager"' employee.txt

Not Match

It is represented as !~. It looks for a field that does not contain the match string.

 # awk '$0 !~ "manager"' employee.txt

print

Each print statement makes at least one line of output. However, it isn’t limited to only one line. If an item value is a string containing a newline, the newline is output along with the rest of the string. A single print statement can make any number of lines this way.

 # awk '{print "Welcome to awk command tutorial "}’

 # awk 'BEGIN { print "line one\nline two\nline three" }’

 # awk '{ print $1, $2 }’ employee.txt

 # awk '{ print $1 $2 }’ employee.txt

 # awk ‘/manager/ {print $0}’ employee.txt

print

 # awk '{ print $2, $2 }’ employee.txt ;out of order printing

#awk ‘BEGIN{cnt=0} /manager/ {cnt++} END{print cnt}’ employee.txt

Length greater than 25

#awk 'length($0) > 25' employee.txt

printf

For more precise control over the output format than what is provided by print, use printf. With printf you can specify the width to use for each item, as well as various formatting choices for numbers (such as what output base to use, whether to print an exponent, whether to print a sign, and how many digits to print after the decimal point).

```
# awk '{ printf "%-10s %s\n", $1, $2 }' employee.txt
```

```
# awk 'BEGIN { print "Name      Role"
            print "----      ------" }
          { printf "%-10s %s\n", $1, $2 }' employee.txt
```

AWK Regular Expression

AWK is very powerful and efficient in handling regular expressions. A number of complex tasks can be solved with simple regular expressions. Any command-line expert knows the power of regular expressions.

Dot

It matches any single character except the end of line character. For instance, the following example matches fin, fun, fan etc.

```
# echo -e "cat\nbat\nfun\nfin\nfan" | awk '/f.n/'
```

AWK Regular Expression

Start of line

It matches the start of line. For instance, the following example prints all the lines that start with pattern The.

```
# echo -e "This\nThat\nThere\nTheir\nthese" | awk '/^The/'
```

End of line

It matches the end of line. For instance, the following example prints the lines that end with the letter n.

```
# echo -e "knife\nknow\nfun\nfin\nfan\nnine" | awk '/n$/'
```

AWK Regular Expression

Match character set

It is used to match only one out of several characters. For instance, the following example matches pattern Call and Tall but not Ball.

```
# echo -e "Call\nTall\nBall" | awk '/[CT]all/'
```

Exclusive set

In exclusive set, the carat negates the set of characters in the square brackets. For instance, the following example prints only Ball.

```
# echo -e "Call\nTall\nBall" | awk '/[^CT]all/'
```

AWK Regular Expression

Alteration

A vertical bar allows regular expressions to be logically ORed. For instance, the following example prints Ball and Call.

```
# echo -e "Call\nTall\nBall\nSmall\nShall" | awk '/Call|Ball/'
```

Zero or One Occurrence

It matches zero or one occurrence of the preceding character. For instance, the following example matches Colour as well as Color. We have made u as an optional character by using ?.

```
# echo -e "Colour\nColor" | awk '/Colou?r/'
```

AWK Regular Expression

Zero or More Occurrence

It matches zero or more occurrences of the preceding character. For instance, the following example matches ca, cat, catt, and so on.

```
# echo -e "ca\ncat\ncatt" | awk '/cat*/'
```

One or More Occurrence

It matches one or more occurrence of the preceding character. For instance below example matches one or more occurrences of the 2.

```
# echo -e "111\n22\n123\n234\n456\n222"  | awk '/2+/'
```

AWK Regular Expression

Grouping

Parentheses () are used for grouping and the character | is used for alternatives. For instance, the following regular expression matches the lines containing either Apple Juice or Apple Cake.

```
# echo -e "Apple Juice\nApple Pie\nApple Tart\nApple Cake" | awk
  '/Apple (Juice|Cake)/'
```

AWK Arrays

AWK has associative arrays and one of the best thing about it is – the indexes need not to be continuous set of number; you can use either string or number as an array index. Also, there is no need to declare the size of an array in advance – arrays can expand/shrink at runtime.

Its syntax is as follows:

Syntax

array_name[index] = value

Where array_name is the name of array, index is the array index, and value is any value assigning to the element of the array.

AWK Arrays

Creating Array

To gain more insight on array, let us create and access the elements of an array.

```
# awk 'BEGIN {
   fruits["mango"] = "yellow";
   fruits["orange"] = "orange"
   print fruits["orange"] "\n" fruits["mango"]
}'
```

AWK Arrays

Deleting Array Elements

For insertion, we used assignment operator. Similarly, we can use delete statement to remove an element from the array. The syntax of delete statement is as follows:

Syntax

delete array_name[index]

The following example deletes the element orange. Hence the command does not show any output.

```
# awk 'BEGIN {
  fruits["mango"] = "yellow";
  fruits["orange"] = "orange";
  delete fruits["orange"];
  print fruits["orange"]
}'
```

AWK Arrays

Multi-Dimensional arrays

AWK only supports one-dimensional arrays. But you can easily simulate a multi-dimensional array using the one-dimensional array itself.

Syntax

array["0,0"] = 100

Though we gave 0,0 as index, these are not two indexes. In reality, it is just one index with the string 0,0.

AWK Arrays

```
# awk 'BEGIN {
   array["0,0"] = 100;
   array["0,1"] = 200;
   array["0,2"] = 300;
   array["1,0"] = 400;
   array["1,1"] = 500;
   array["1,2"] = 600;

   # print array elements
   print "array[0,0] = " array["0,0"];
   print "array[0,1] = " array["0,1"];
   print "array[0,2] = " array["0,2"];
   print "array[1,0] = " array["1,0"];
   print "array[1,1] = " array["1,1"];
   print "array[1,2] = " array["1,2"];
}'
```

AWK Control Flow

AWK provides conditional statements to control the flow of a program.

If statement

It simply tests the condition and performs certain actions depending upon the condition. Given below is the syntax of if statement:

Syntax

if (condition)

action

AWK Control Flow

We can also use a pair of curly braces as given below to execute multiple actions:

Syntax

```
if (condition) {
   action-1
   action-1
   .
   .
   action-n
}
```

For instance, the following example checks whether a number is even or not:

```
# awk 'BEGIN {num = 10; if (num % 2 == 0) printf "%d is even number.\n", num }'
```

AWK Control Flow

If Else Statement

In if-else syntax, we can provide a list of actions to be performed when a condition becomes false.

The syntax of if-else statement is as follows:

Syntax

```
if (condition)
  action-1
else
  action-2
```

In the above syntax, action-1 is performed when the condition evaluates to true and action-2 is performed when the condition evaluates to false. For instance, the following example checks whether a number is even or not:

```
# awk 'BEGIN {
  num = 11; if (num % 2 == 0) printf "%d is even number.\n", num;
    else printf "%d is odd number.\n", num
}'
```

AWK Control Flow

If-Else-If Ladder

We can easily create an if-else-if ladder by using multiple if-else statements.

```
# awk 'BEGIN {
   a = 30;

   if (a==10)
   print "a = 10";
   else if (a == 20)
   print "a = 20";
   else if (a == 30)
   print "a = 30";
}'
```

AWK Loops

Loops are used to execute a set of actions in a repeated manner. The loop execution continues as long as the loop condition is true.

For Loop

The syntax of for loop is:

Syntax

for (initialization; condition; increment/decrement)

action

Initially, the for statement performs initialization action, then it checks the condition. If the condition is true, it executes actions, thereafter it performs increment or decrement operation. The loop execution continues as long as the condition is true. For instance, the following example prints 1 to 5 using for loop:

```
# awk 'BEGIN { for (i = 1; i <= 5; ++i) print i }'
```

AWK Loops

While Loop

The while loop keeps executing the action until a particular logical condition evaluates to true. Here is the syntax of while loop:

Syntax

while (condition)

action

AWK first checks the condition; if the condition is true, it executes the action. This process repeats as long as the loop condition evaluates to true. For instance, the following example prints 1 to 5 using while loop.

```
# awk 'BEGIN {i = 1; while (i < 6) { print i; ++i } }'
```

AWK Loops

Do-While Loop

The do-while loop is similar to the while loop, except that the test condition is evaluated at the end of the loop. Here is the syntax of do-whileloop:

Syntax

do

action

while (condition)

In a do-while loop, the action statement gets executed at least once even when the condition statement evaluates to false. For instance, the following example prints 1 to 5 numbers using do-while loop.

```
# awk 'BEGIN {i = 1; do { print i; ++i } while (i < 6) }'
```

AWK Loops

Break Statement

As its name suggests, it is used to end the loop execution. Here is an example which ends the loop when the sum becomes greater than 50.

```
# awk 'BEGIN {
  sum = 0; for (i = 0; i < 20; ++i) {
    sum += i; if (sum > 50) break; else print "Sum =", sum
  }
}'
```

AWK Loops

Continue Statement

The continue statement is used inside a loop to skip to the next iteration of the loop. It is useful when you wish to skip the processing of some data inside the loop. For instance, the following example uses continue statement to print the even numbers between 1 to 20.

```
# awk 'BEGIN {
  for (i = 1; i <= 20; ++i) {
    if (i % 2 == 0) print i ; else continue
  }
}'
```

AWK Loops

Exit Statement

It is used to stop the execution of the script. It accepts an integer as an argument which is the exit status code for AWK process. If no argument is supplied, exit returns status zero. Here is an example that stops the execution when the sum becomes greater than 50.

```
# awk 'BEGIN {
  sum = 0; for (i = 0; i < 20; ++i) {
    sum += i; if (sum > 50) exit(10); else print "Sum =", sum
  }
}'
```

Let us check the return status of the script.

```
# echo $?
```

User define Functions

AWK allows us to define our own functions. A large program can be divided into functions and each function can be written/tested independently. It provides re-usability of code.

Syntax

```
function function_name(argument1, argument2, ...) {
   function body
}
```

In this syntax, the **function_name** is the name of the user-defined function. Function name should begin with a letter and the rest of the characters can be any combination of numbers, alphabetic characters, or underscore. AWK's reserve words cannot be used as function names.

Functions can accept multiple arguments separated by comma. Arguments are not mandatory. You can also create a user-defined function without any argument.

function body consists of one or more AWK statements.

User define Functions

function.awk

```
# Returns minimum number
function find_min(num1, num2){
  if (num1 < num2)
  return num1
  return num2
}
# Returns maximum number
function find_max(num1, num2){
  if (num1 > num2)
  return num1
  return num2
}
# Main function
function main(num1, num2){
  # Find minimum number
  result = find_min(10, 20)
  print "Minimum =", result

  # Find maximum number
  result = find_max(10, 20)
  print "Maximum =", result
}
# Script execution starts here
BEGIN {
  main(10, 20)
}
```

Arithematic Functions

AWK has the following built-in arithmetic functions:

atan2(y, x)

It returns the arctangent of (y/x) in radians.

```
# awk 'BEGIN {
   PI = 3.14159265
   x = -10
   y = 10
   result = atan2 (y,x) * 180 / PI;

   printf "The arc tangent for (x=%f, y=%f) is %f degrees\n", x, y, result
}'
```

Arithematic Functions

cos(expr)

This function returns the cosine of expr, which is expressed in radians.

```
# awk 'BEGIN {
   PI = 3.14159265
   param = 60
   result = cos(param * PI / 180.0);

   printf "The cosine of %f degrees is %f.\n", param, result
}'
```

Arithematic Functions

exp(expr)

This function is used to find the exponential value of a variable.

```
# awk 'BEGIN {
   param = 5
   result = exp(param);

   printf "The exponential value of %f is %f.\n", param, result
}'
```

Arithematic Functions

int(expr)

This function truncates the expr to an integer value.

```
# awk 'BEGIN {
   param = 5.12345
   result = int(param)

   print "Truncated value =", result
}'
```

Arithematic Functions

log(expr)

This function calculates the natural logarithm of a variable.

```
# awk 'BEGIN {
   param = 5.5
   result = log (param)

   printf "log(%f) = %f\n", param, result
}'
```

Arithematic Functions

rand

This function returns a random number N, between 0 and 1, such that 0 <= N < 1. For instance, the following example generates three random numbers

```
# awk 'BEGIN {
   print "Random num1 =" , rand()
   print "Random num2 =" , rand()
   print "Random num3 =" , rand()
}'
```

Arithematic Functions

sin(expr)

This function returns the sine of expr, which is expressed in radians.

```
# awk 'BEGIN {
   PI = 3.14159265
   param = 30.0
   result = sin(param * PI /180)

   printf "The sine of %f degrees is %f.\n", param, result
}'
```

Arithematic Functions

sqrt(expr)

This function returns the square root of expr.

```
# awk 'BEGIN {
   param = 1024.0
   result = sqrt(param)

   printf "sqrt(%f) = %f\n", param, result
}'
```

Arithematic Functions

srand([expr])

This function generates a random number using seed value. It uses expr as the new seed for the random number generator. In the absence of expr, it uses the time of day as the seed value.

```
# awk 'BEGIN {
  param = 10

  printf "srand() = %d\n", srand()
  printf "srand(%d) = %d\n", param, srand(param)
}'
```

String Functions

AWK has the following built-in String functions:

gsub(regex, sub, string)

gsub stands for global substitution. It replaces every occurrence of regex with the given string (sub). The third parameter is optional. If it is omitted, then $0 is used.

```
# awk 'BEGIN {
  str = "Hello, World"
  print "String before replacement = " str

  gsub("World", "Jerry", str)
  print "String after replacement = " str
}'
```

String Functions

index(str, sub)

It checks whether sub is a substring of str or not. On success, it returns the position where sub starts; otherwise it returns 0. The first character of str is at position 1.

```
# awk 'BEGIN {
   str = "One Two Three"
   subs = "Two"
   ret = index(str, subs)

   printf "Substring \"%s\" found at %d location.\n", subs, ret
}'
```

String Functions

length(str)

It returns the length of a string.

```
# awk 'BEGIN {
   str = "Hello, World !!!"
   print "Length = ", length(str)
}'
```

String Functions

match(str, regex)

It returns the index of the first longest match of regex in string str. It returns 0 if no match found.

```
# awk 'BEGIN {
  str = "One Two Three"
  subs = "Two"
  ret = match(str, subs)

  printf "Substring \"%s\" found at %d location.\n", subs, ret
}'
```

String Functions

split(str, arr, regex)

This function splits the string str into fields by regular expression regex and the fields are loaded into the array arr. If regex is omitted, then FS is used.

```
# awk 'BEGIN {
   str = "One,Two,Three,Four"
   split(str, arr, ",")
   print "Array contains following values"

   for (i in arr) {
      print arr[i]
   }
}'
```

String Functions

strtonum(str)

This function examines str and return its numeric value. If str begins with a leading 0, it is treated as an octal number. If str begins with a leading 0x or 0X, it is taken as a hexadecimal number. Otherwise, assume it is a decimal number.

```
# awk 'BEGIN {
  print "Decimal num = " strtonum("123")
  print "Octal num = " strtonum("0123")
  print "Hexadecimal num = " strtonum("0x123")
}'
```

String Functions

sub(regex, sub, string)

This function performs a single substitution. It replaces the first occurrence of the regex pattern with the given string (sub). The third parameter is optional. If it is omitted, $0 is used.

```
# awk 'BEGIN {
  str = "Hello, World"
  print "String before replacement = " str

  sub("World", "Jerry", str)
  print "String after replacement = " str
}'
```

String Functions

substr(str, start, l)

This function returns the substring of string str, starting at index start of length l. If length is omitted, the suffix of str starting at index start is returned.

```
# awk 'BEGIN {
   str = "Hello, World !!!"
   subs = substr(str, 1, 5)

   print "Substring = " subs
}'
```

String Functions

tolower(str)

This function returns a copy of string str with all upper-case characters converted to lower-case.

```
# awk 'BEGIN {
  str = "HELLO, WORLD !!!"
  print "Lowercase string = " tolower(str)
}'
```

toupper(str)

This function returns a copy of string str with all lower-case characters converted to upper case.

```
# awk 'BEGIN {
  str = "hello, world !!!"
  print "Uppercase string = " toupper(str)
}'
```

Redirection

Redirection Operator

The syntax of the redirection operator is:

Syntax

print DATA > output-file

It writes the data into the output-file. If the output-file does not exist, then it creates one. When this type of redirection is used, the output-file is erased before the first output is written to it. Subsequent write operations to the same output-file do not erase the output-file, but append to it. For instance, the following example writes Hello, World !!! to the file.

```
# awk 'BEGIN { print "Hello, World !!!" > "/tmp/message.txt" }'
```

Redirection

Append Operator

The syntax of append operator is as follows:

Syntax

print DATA >> output-file

It appends the data into the output-file. If the output-file does not exist, then it creates one. When this type of redirection is used, new contents are appended at the end of file. For instance, the following example appends Hello, World !!! to the file.

```
# awk 'BEGIN { print "Hello, World !!!" >> "/tmp/message.txt" }'
```

VIM

vi filename

Creates a new file if it already does not exist, otherwise opens an existing file.

vi -R filename

Opens an existing file in the read-only mode.

view filename

Opens an existing file in the read-only mode.

VIM

```
$vi testfile
```

```
|
~
~
~
~
~
~
~
~
~
~
~
~
"testfile" [New File]
```

A tilde (~) on each line following the cursor. A tilde represents an unused line. If a line does not begin with a tilde and appears to be blank, there is a space, tab, newline, or some other non-viewable character present.

VIM – Operation Modes

While working with the vi editor, we usually come across the following two modes:

- **Command mode:** This mode enables you to perform administrative tasks such as saving the files, executing the commands, moving the cursor, cutting (yanking) and pasting the lines or words, as well as finding and replacing. In this mode, whatever you type is interpreted as a command.

- **Insert mode:** This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and placed in the file.

vi always starts in the command mode. To enter text, you must be in the insert mode for which simply type **i**. To come out of the insert mode, press the Esc key, which will take you back to the command mode.

Getting Out of vi

The command to quit out of vi is **:q**. Once in the command mode, type colon, and 'q', followed by return. If your file has been modified in any way, the editor will warn you of this, and not let you quit. To ignore this message, the command to quit out of vi without saving is **:q!.** This lets you exit vi without saving any of the changes.

The command to save the contents of the editor is :w. You can combine the above command with the quit command, or use :wq and return.

The easiest way to save your changes and exit vi is with the ZZ command. When you are in the command mode, **type ZZ**. The ZZ command works the same way as the :wq command.

If you want to specify/state any particular name for the file, you can do so by specifying it after the **:w**. For example, if you wanted to save the file you were working on as another filename called filename2, you would type **:w filename2** and return.

VIM

Moving within a File

Sr.No.	Command & Description
1	**k** Moves the cursor up one line
2	**j** Moves the cursor down one line
3	**h** Moves the cursor to the left one character position
4	**l** Moves the cursor to the right one character position

Editing Files

Sr.No.	Command & Description
1	**i** Inserts text before the current cursor location
2	**I** Inserts text at the beginning of the current line
3	**a** Inserts text after the current cursor location
4	**A** Inserts text at the end of the current line
5	**o** Creates a new line for text entry below the cursor location
6	**O** Creates a new line for text entry above the cursor location

Deleting Characters

Sr.No.	Command & Description
1	**x** Deletes the character under the cursor location
2	**X** Deletes the character before the cursor location
3	**dw** Deletes from the current cursor location to the next word
4	**d^** Deletes from the current cursor position to the beginning of the line
5	**d$** Deletes from the current cursor position to the end of the line
6	**D** Deletes from the cursor position to the end of the current line
7	**dd or ndd** Deletes the line the cursor is on

Change Commands

Sr.No.	Command & Description
1	**cc** Removes the contents of the line, leaving you in insert mode.
2	**cw** Changes the word the cursor is on from the cursor to the lowercase **w** end of the word.
3	**r** Replaces the character under the cursor. vi returns to the command mode after the replacement is entered.
4	**R** Overwrites multiple characters beginning with the character currently under the cursor. You must use **Esc** to stop the overwriting.
5	**s** Replaces the current character with the character you type. Afterward, you are left in the insert mode.
6	**S** Deletes the line the cursor is on and replaces it with the new text. After the new text is entered, vi remains in the insert mode.

Copy and Paste Commands

Sr.No.	Command & Description
1	**yy** Copies the current line.
2	**yw** Copies the current word from the character the lowercase w cursor is on, until the end of the word.
3	**p** Puts the copied text after the cursor.
4	**P** Puts the yanked text before the cursor.

Word and Character Searching

The vi editor has two kinds of searches: string and character. For a string search, the / and ? commands are used. When you start these commands, the command just typed will be shown on the last line of the screen, where you type the particular string to look for.

These two commands differ only in the direction where the search takes place:

- The / command searches forwards (downwards) in the file.
- The ? command searches backwards (upwards) in the file.

Set Commands

Sr.No.	Command & Description
1	**:set ic** Ignores the case when searching
2	**:set ai** Sets autoindent
3	**:set noai** Unsets autoindent
4	**:set nu** Displays lines with line numbers on the left side
5	**:set sw** Sets the width of a software tabstop. For example, you would set a shift width of 4 with this command — **:set sw = 4**
6	**:set ws** If *wrapscan* is set, and the word is not found at the bottom of the file, it will try searching for it at the beginning
7	**:set wm** If this option has a value greater than zero, the editor will automatically "word wrap". For example, to set the wrap margin to two characters, you would type this: **:set wm = 2**
8	**:set ro** Changes file type to "read only"
9	**:set term** Prints terminal type
10	**:set bf** Discards control characters from input

VIM

Running Commands

The vi has the capability to run commands from within the editor. To run a command, you only need to go to the command mode and type :! command.

For example, if you want to check whether a file exists before you try to save your file with that filename, you can type **:! ls** and you will see the output of ls on the screen.

You can press any key (or the command's escape sequence) to return to your vi session.

:r !program

:. ! program

Replacing Text

The substitution command (:%s/) enables you to quickly replace words or groups of words within your files. Following is the syntax to replace text −

:%s/search/replace

VIM

Starting a shell

: sh

Splitting window

: split filename

CTRL+W to shift to different window

Device Tree – Introduction

A device tree is a tree structure used to describe the physical hardware in a system. Each node in the tree describes the characteristics of the device being represented. The purpose of the device tree is to describe device information in a system that cannot necessarily be dynamically detected or discovered by a client program. For example, a PCI host may be able to probe and detect attached devices; and so a device tree node describing PCI devices may not be required. However, a device node is required to describe the PCI host bridge in the system, if that cannot be detected by probing.

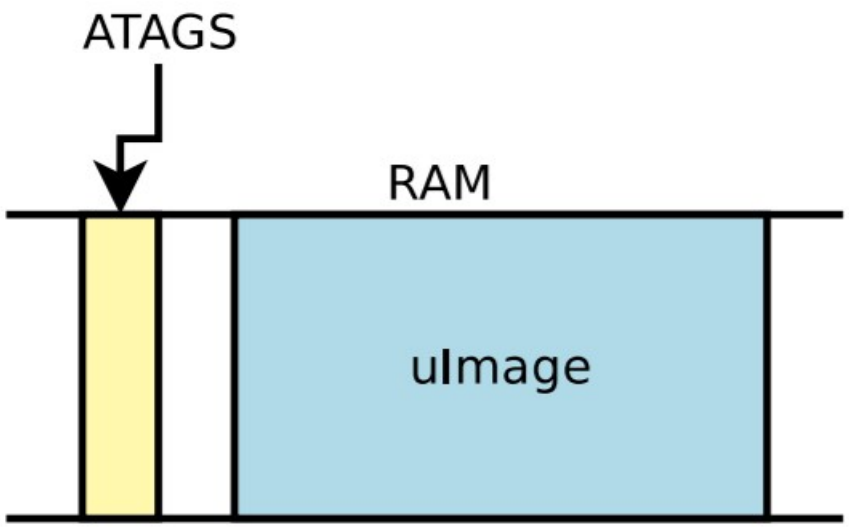
Device Tree – Before Device Tree

- The kernel contains the entire description of the hardware.
- The bootloader loads a single binary, the kernel image, and executes it.

  uImage or zImage
- The bootloader prepares some additional information, called ATAGS, which address is passed to the kernel through register r2 Contains information such as memory size and location, kernel command line, etc.
- The bootloader tells the kernel on which board it is being booted through a machine type integer, passed in register r1.

  U-Boot command: bootm <kernel img addr>

Device Tree – Before Device Tree



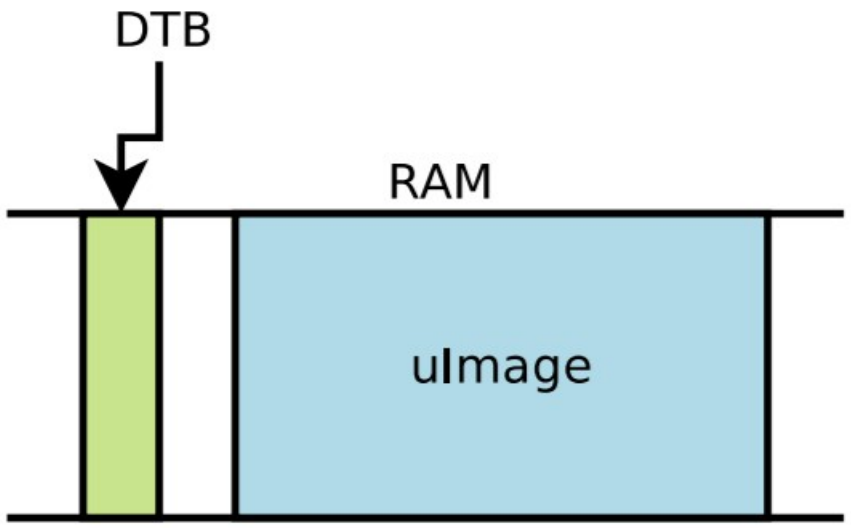

```
r1 = <machine type>
r2 = <pointer to ATAGS>
```

Device Tree – After Device Tree

- The kernel no longer contains the description of the hardware,it is located in a separate binary: the device tree blob
- The bootloader loads two binaries: the kernel image and the DTB
  - Kernel image remains uImage or zImage
  - DTB located in arch/arm/boot/dts, one per board
- The bootloader passes the DTB address through r2. It is supposed to adjust the DTB with memory information, kernel command line, and potentially other info.
- No more machine type.
- U-Boot command:

  bootm <kernel img addr> - <dtb addr>

Device Tree – After Device Tree




```
r1 = don't care
r2 = <pointer to DTB>
```

Device Tree – Sample

Node name

Unit address

Property name

Property value

Properties of node@0

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];

        child-node@0 {
            first-child-property;
            second-child-property = <1>;
            a-reference-to-something = <&node1>;
        };

        child-node@1 {
        };
    };

    node1: node@1 {
        an-empty-property;
        a-cell-property = <1 2 3 4>;

        child-node@0 {
        };
    };
};
```

Bytestring

A phandle
(reference to another node)

Label

Four cells (32 bits values)

Device Tree

- On ARM, all Device Tree Source files (DTS) are for now located in arch/arm/boot/dts
  - .dts files for board-level definitions
  - .dtsi files for included files, generally containing SoC-level definitions
- A tool, the Device Tree Compiler compiles the source into a binary form.
  - Source code located in scripts/dtc

```
dtb-$(CONFIG_ARCH_MVEBU) += armada-370-db.dtb \
        armada-370-mirabox.dtb \
...
```

- The Device Tree Blob is produced by the compiler, and is the binary that gets loaded by the bootloader and parsed by
- the kernel at boot time.
- arch/arm/boot/dts/Makefile lists which DTBs should be generated at build time.

Device Tree – Sample

```
auart0: serial@8006a000 {
```

Defines the "programming model" for the device. Allows the operating system to identify the corresponding device driver.

```
        compatible = "fsl,imx28-auart", "fsl,imx23-auart";
```

Address and length of the register area.

```
        reg = <0x8006a000 0x2000>;
```

Interrupt number.

```
        interrupts = <112>;
```

DMA engine and channels, with names.

```
        dmas = <&dma_apbx 8>, <&dma_apbx 9>;
        dma-names = "rx", "tx";
```

Reference to the clock.

```
        clocks = <&clks 45>;
```

The device is not enabled.

```
        status = "disabled";
};
```

Taken from arch/arm/boot/dts/imx28.dtsi

Device Tree – Inclusion

Definition of the AM33xx SoC

```
/ {
     compatible = "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               compatible = "ti,omap3-uart";
               reg = <0x44e09000 0x2000>;
               interrupts = <72>;
               status = "disabled";
          };
     };
};
                                   am33xx.dtsi
```

+

Definition of the BeagleBone board

```
#include "am33xx.dtsi"

/ {
     compatible = "ti,am335x-bone", "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               pinctrl-names = "default";
               pinctrl-0 = <&uart0_pins>;
               status = "okay";
          };
     };
};
                                   am335x-bone.dts
```

=

Compiled DTB

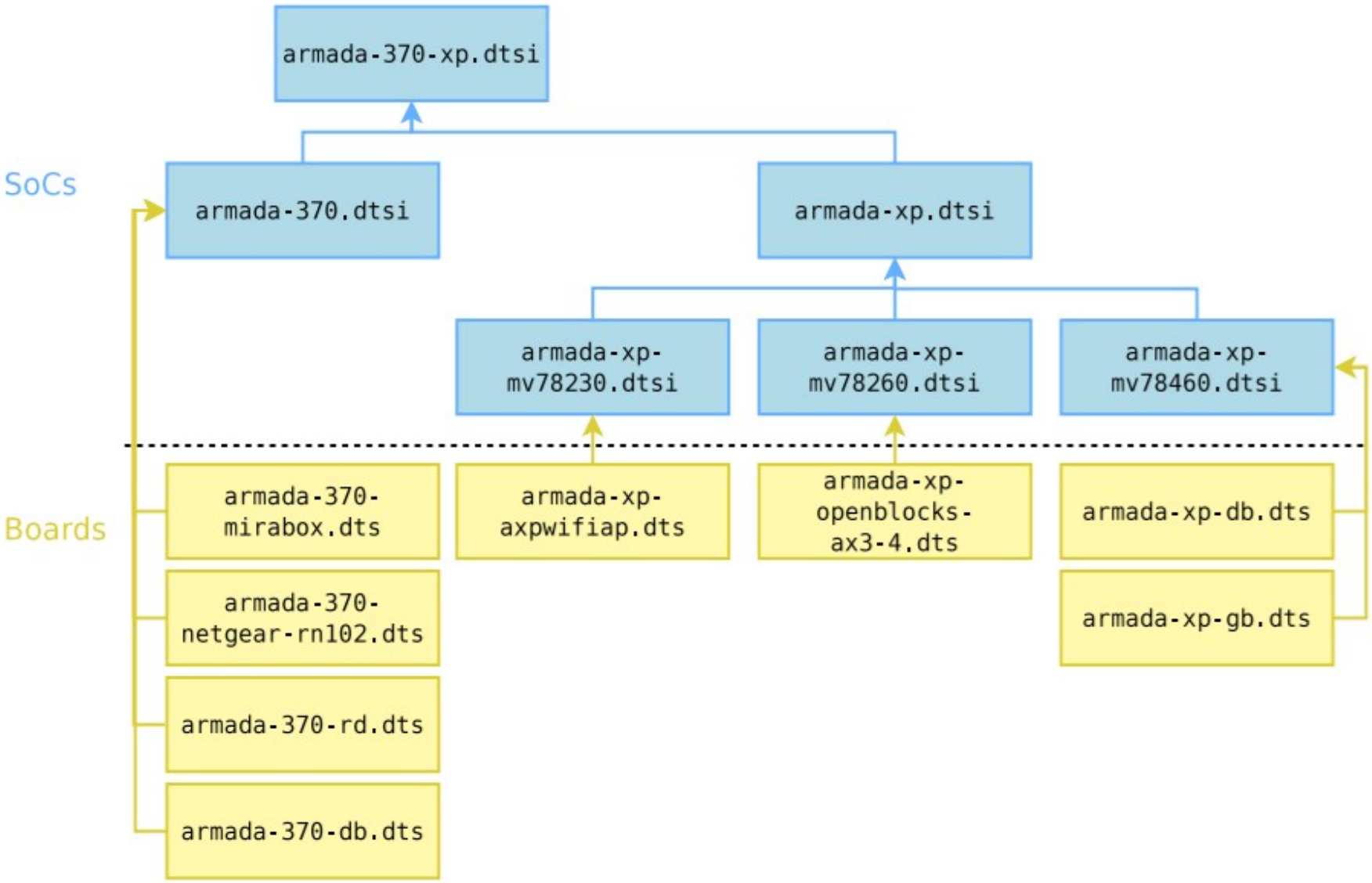
```
/ {
     compatible = "ti,am335x-bone", "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               compatible = "ti,omap3-uart";
               reg = <0x44e09000 0x2000>;
               interrupts = <72>;
               pinctrl-names = "default";
               pinctrl-0 = <&uart0_pins>;
               status = "okay";
          };
     };
};
                                   am335x-bone.dtb
```

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

Device Tree – After Device Tree

- Device Tree files are not monolithic, they can be split in several files, including each other.
- `.dtsi` files are included files, while `.dts` files are *final* Device Trees
- Typically, `.dtsi` will contain definition of SoC-level information (or sometimes definitions common to several almost identical boards).
- The `.dts` file contains the board-level information.
- The inclusion works by **overlaying** the tree of the including file over the tree of the included file.
- Inclusion using the DT operator `/include/`, or since a few kernel releases, the DTS go through the C preprocessor, so `#include` is recommended.

Device Tree – After Device Tree



Device Tree – top level nodes

Under the root of the Device Tree, one typically finds the following top-level nodes:

- A `cpus` node, which sub-nodes describing each CPU in the system.
- A `memory` node, which defines the location and size of the RAM.
- A `chosen` node, which defines *parameters chosen or defined by the system firmware at boot time*. In practice, one of its usage is to pass the kernel command line.
- A `aliases` node, to define shortcuts to certain nodes.
- One or more nodes defining the *buses* in the SoC.
- One or mode nodes defining on-board devices.

Device Tree – top level nodes

arch/arm/boot/dts/imx28.dtsi

```
/ {
        aliases { ... };
        cpus { ... };

        apb@80000000 {
                apbh@80000000 {
                        /* Some devices */
                };

                apbx@80040000 {
                        /* Some devices */
                };
        };

        ahb@80080000 {
                /* Some devices */
        };
};
```

Device Tree – top level nodes

arch/arm/boot/dts/imx28-evk.dts

```
/ {
        model = "Freescale i.MX28 Evaluation Kit";
        compatible = "fsl,imx28-evk", "fsl,imx28";

        memory {
                reg = <0x40000000 0x08000000>;
        };


        apb@80000000 {
                apbh@80000000 { ... };
                apbx@80040000 { ... };
        };

        ahb@80080000 { ... };

        sound { ... };
        leds { ... };
        backlight { ... };
};
```

Device Tree – top level nodes

Inside a bus, one typically needs to define the following properties:

- A `compatible` property, which identifies the bus controller (in case of I2C, SPI, PCI, etc.). A special value `compatible = "simple-bus"` means a simple memory-mapped bus with no specific handling or driver. Child nodes will be registered as *platform devices*.
- The `#address-cells` property indicate how many cells (i.e 32 bits values) are needed to form the base address part in the `reg` property.
- The `#size-cells` is the same, for the size part of the `reg` property.
- The `ranges` property can describe an *address translation* between the child bus and the parent bus. When simply defined as `ranges;`, it means that the translation is an identity translation.

Device Tree – top level nodes

```
apbh@80000000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        reg = <0x80000000 0x3c900>;
        ranges;

        [...]

        hsadc: hsadc@80002000 {
                reg = <0x80002000 0x2000>;
                interrupts = <13>;
                dmas = <&dma_apbh 12>;
                dma-names = "rx";
                status = "disabled";
        };

        [...]
};
```

Device Tree – top level nodes

```
i2c0: i2c@80058000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx28-i2c";
        reg = <0x80058000 0x2000>;
        interrupts = <111>;
        [...]

        sgtl5000: codec@0a {
                compatible = "fsl,sgtl5000";
                reg = <0x0a>;
                VDDA-supply = <&reg_3p3v>;
                VDDIO-supply = <&reg_3p3v>;
                clocks = <&saif0>;
        };

        at24@51 {
                compatible = "at24,24c32";
                pagesize = <32>;
                reg = <0x51>;
        };
};
```

Device Tree – Interrupt Handling

- `interrupt-controller;` is a boolean property that indicates that the current node is an interrupt controller.
- `#interrupt-cells` indicates the number of cells in the `interrupts` property for the interrupts managed by the selected interrupt controller.
- `interrupt-parent` is a *phandle* that points to the interrupt controller for the current node. There is generally a top-level `interrupt-parent` definition for the main interrupt controller.

Device Tree – Interrupt Handling

```
/ {
    interrupt-parent = <&icoll>;
    apb@80000000 {
        apbh@80000000 {
            icoll: interrupt-controller@80000000 {
                compatible = "fsl,imx28-icoll", "fsl,icoll";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = <0x80000000 0x2000>;
            };

            ssp0: ssp@80010000 {
                [...]
                interrupts = <96>;
            };
        };
    };
};
```

Device Tree – Interrupt Handling

```
/ {
    interrupt-parent = <&icoll>;
    apb@80000000 {
        apbh@80000000 {
            icoll: interrupt-controller@80000000 {
                compatible = "fsl,imx28-icoll", "fsl,icoll";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = <0x80000000 0x2000>;
            };

            ssp0: ssp@80010000 {
                [...]
                interrupts = <96>;
            };
        };
    };
};
```

Device Tree – Interrupt Handling

- The Device Tree is really a hardware description language.
- It should **describe the hardware layout**, and how it works.
- But it should **not describe which particular hardware configuration** you're interested in.
- As an example:
  - You may describe in the DT whether a particular piece of hardware supports DMA or not.
  - But you may not describe in the DT if you *want* to use DMA or not.

Device Tree – Interrupt Handling

- Since the DT is OS independent, it should also be stable.
- The original idea is that DTBs can be flashed on some devices by the manufacturer, so that the user can install whichever operating system it wants.
- Once a Device Tree binding is defined, and used in DTBs, it should no longer be changed anymore. It can only be extended.
- This normally means that **Device Tree bindings become part of the kernel ABI**, and it should be handled with the same care.
- However, kernel developers are realizing that this is really hard to achieve and slowing down the integration of drivers.
  - The ARM Kernel Mini-summit discussions have relaxed those rules.
  - There will be additional discussions during the Kernel Summit, with final conclusions published afterwards.

Device Tree – Handson

```
# ~/qemu/linux-5.1.16/scripts/dtc/dtc -I dts -O dtb -o test.dtb test.dts
# ~/qemu/linux-5.1.16/scripts/dtc/dtc -I dtb -O dts -o test.dts1 test.dtb
```

UIO Driver

For many types of devices, creating a Linux kernel driver is overkill. All that is really needed is some way to handle an interrupt and provide access to the memory space of the device. The logic of controlling the device does not necessarily have to be within the kernel, as the device does not need to take advantage of any of other resources that the kernel provides. One such common class of devices that are like this are for industrial I/O cards.

To address this situation, the userspace I/O system (UIO) was designed. For typical industrial I/O cards, only a very small kernel module is needed. The main part of the driver will run in user space. This simplifies development and reduces the risk of serious bugs within a kernel module.

UIO Driver

Devices that are already handled well by other kernel subsystems (like networking or serial or USB) are no candidates for an UIO driver. Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped. The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

UIO Driver

Devices that are already handled well by other kernel subsystems (like networking or serial or USB) are no candidates for an UIO driver. Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped. The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

UIO Driver Advantages

If you use UIO for your card's driver, here's what you get:

- only one small kernel module to write and maintain.
- develop the main part of your driver in user space, with all the tools and libraries you're used to.
- bugs in your driver won't crash the kernel.
- updates of your driver can take place without recompiling the kernel.

UIO Driver Working

Each UIO device is accessed through a device file and several sysfs attribute files. The device file will be called /dev/uio0 for the first device, and /dev/uio1, /dev/uio2 and so on for subsequent devices.

/dev/uioX is used to access the address space of the card. Just use mmap() to access registers or RAM locations of your card.

Interrupts are handled by reading from /dev/uioX. A blocking read() from /dev/uioX will return as soon as an interrupt occurs. You can also use select() on /dev/uioX to wait for an interrupt. The integer value read from /dev/uioX represents the total interrupt count. You can use this number to figure out if you missed some interrupts.

UIO Driver Working

For some hardware that has more than one interrupt source internally, but not separate IRQ mask and status registers, there might be situations where userspace cannot determine what the interrupt source was if the kernel handler disables them by writing to the chip's IRQ register. In such a case, the kernel has to disable the IRQ completely to leave the chip's register untouched. Now the userspace part can determine the cause of the interrupt, but it cannot re-enable interrupts. Another cornercase is chips where re-enabling interrupts is a read-modify-write operation to a combined IRQ status/acknowledge register. This would be racy if a new interrupt occurred simultaneously.

To address these problems, UIO also implements a write() function. It is normally not used and can be ignored for hardware that has only a single interrupt source or has separate IRQ mask and status registers. If you need it, however, a write to /dev/uioX will call the irqcontrol() function implemented by the driver. You have to write a 32-bit value that is usually either 0 or 1 to disable or enable interrupts. If a driver does not implement irqcontrol(), write() will return with -ENOSYS.

UIO Driver Working

To handle interrupts properly, your custom kernel module can provide its own interrupt handler. It will automatically be called by the built-in handler.

For cards that don’t generate interrupts but need to be polled, there is the possibility to set up a timer that triggers the interrupt handler at configurable time intervals. This interrupt simulation is done by calling uio_event_notify() from the timer’s event handler.

Each driver provides attributes that are used to read or write variables. These attributes are accessible through sysfs files. A custom kernel driver module can add its own attributes to the device owned by the uio driver, but not added to the UIO device itself at this time. This might change in the future if it would be found to be useful.

UIO Driver Attributes

The following standard attributes are provided by the UIO framework:

- name: The name of your device. It is recommended to use the name of your kernel module for this.
- version: A version string defined by your driver. This allows the user space part of your driver to deal with different versions of the kernel module.
- event: The total number of interrupts handled by the driver since the last time the device node was read.

These attributes appear under the /sys/class/uio/uioX directory. Please note that this directory might be a symlink, and not a real directory. Any userspace code that accesses it must be able to handle this.

UIO Driver Attributes

The following standard attributes are provided by the UIO framework:

- name: The name of your device. It is recommended to use the name of your kernel module for this.
- version: A version string defined by your driver. This allows the user space part of your driver to deal with different versions of the kernel module.
- event: The total number of interrupts handled by the driver since the last time the device node was read.

These attributes appear under the /sys/class/uio/uioX directory. Please note that this directory might be a symlink, and not a real directory. Any userspace code that accesses it must be able to handle this.

UIO Driver Memory mapping

Each UIO device can make one or more memory regions available for memory mapping. This is necessary because some industrial I/O cards require access to more than one PCI memory region in a driver.

Each mapping has its own directory in sysfs, the first mapping appears as /sys/class/uio/uioX/maps/map0/. Subsequent mappings create directories map1/, map2/, and so on. These directories will only appear if the size of the mapping is not 0.

UIO Driver Memory mapping

Each mapX/ directory contains four read-only files that show attributes of the memory:

- **name:** A string identifier for this mapping. This is optional, the string can be empty. Drivers can set this to make it easier for userspace to find the correct mapping.
- **addr:** The address of memory that can be mapped.
- **size:** The size, in bytes, of the memory pointed to by addr.
- **offset:** The offset, in bytes, that has to be added to the pointer returned by mmap() to get to the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by mmap() are always page aligned, so it is good style to always add this offset.

UIO Driver Memory mapping

Each mapX/ directory contains four read-only files that show attributes of the memory:

- **name:** A string identifier for this mapping. This is optional, the string can be empty. Drivers can set this to make it easier for userspace to find the correct mapping.
- **addr:** The address of memory that can be mapped.
- **size:** The size, in bytes, of the memory pointed to by addr.
- **offset:** The offset, in bytes, that has to be added to the pointer returned by mmap() to get to the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by mmap() are always page aligned, so it is good style to always add this offset.

UIO Driver PORT IO

The new directory /sys/class/uio/uioX/portio/ is added. It only exists if the driver wants to pass information about one or more port regions to userspace. If that is the case, subdirectories named port0, port1, and so on, will appear underneath /sys/class/uio/uioX/portio/.

Each portX/ directory contains four read-only files that show name, start, size, and type of the port region:

- **name:** A string identifier for this port region. The string is optional and can be empty. Drivers can set it to make it easier for userspace to find a certain port region.
- **start:** The first port of this region.
- **size:** The number of ports in this region.
- **porttype:** A string describing the type of port.

UIO Driver PORT IO

The new directory /sys/class/uio/uioX/portio/ is added. It only exists if the driver wants to pass information about one or more port regions to userspace. If that is the case, subdirectories named port0, port1, and so on, will appear underneath /sys/class/uio/uioX/portio/.

Each portX/ directory contains four read-only files that show name, start, size, and type of the port region:

- **name:** A string identifier for this port region. The string is optional and can be empty. Drivers can set it to make it easier for userspace to find a certain port region.
- **start:** The first port of this region.
- **size:** The number of ports in this region.
- **porttype:** A string describing the type of port.

UIO Driver Userspace

Once you have a working kernel module for your hardware, you can write the userspace part of your driver. You don’t need any special libraries, your driver can be written in any reasonable language, you can use floating point numbers and so on. In short, you can use all the tools and libraries you’d normally use for writing a userspace application.

UIO Driver Information

Information about all UIO devices is available in sysfs. The first thing you should do in your driver is check name and version to make sure your talking to the right device and that its kernel driver has the version you expect.

You should also make sure that the memory mapping you need exists and has the size you expect.

There is a tool called lsuio that lists UIO devices and their attributes. It is available here:

http://www.osadl.org/projects/downloads/UIO/user/

With lsuio you can quickly check if your kernel module is loaded and which attributes it exports. Have a look at the manpage for details.

The source code of lsuio can serve as an example for getting information about an UIO device. The file uio_helper.c contains a lot of functions you could use in your userspace driver code.

UIO Driver mmap

After you made sure you've got the right device with the memory mappings you need, all you have to do is to call mmap() to map the device's memory to userspace.

The parameter offset of the mmap() call has a special meaning for UIO devices.

UIO Driver mmap

After you made sure you've got the right device with the memory mappings you need, all you have to do is to call mmap() to map the device's memory to userspace.

The parameter offset of the mmap() call has a special meaning for UIO devices.

UIO Driver Interrupt

After you successfully mapped your devices memory, you can access it like an ordinary array. Usually, you will perform some initialization. After that, your hardware starts working and will generate an interrupt as soon as it’s finished, has some data available, or needs your attention because an error occurred.

/dev/uioX is a read-only file. A read() will always block until an interrupt occurs. There is only one legal value for the count parameter of read(), and that is the size of a signed 32 bit integer (4). Any other value for count causes read() to fail. The signed 32 bit integer read is the interrupt count of your device. If the value is one more than the value you read the last time, everything is OK. If the difference is greater than one, you missed interrupts.

You can also use select() on /dev/uioX.

UIO Driver code

```
#include  <stdlib.h>
#include  <stdio.h>
#include  <unistd.h>
#include  <fcntl.h>

int main()
{
        int fd;
        unsigned long nint;
        if ((fd = open("/dev/uio0", O_RDONLY)) < 0) {
        perror("Failed to open /dev/uio0\n");
        exit(EXIT_FAILURE);
        }
        fprintf(stderr, "Started uio test driver.\n");
        while (read(fd, &nint, sizeof(nint)) >= 0)
        fprintf(stderr, "Interrupts: %ld\n", nint);
        exit(EXIT_SUCCESS);
}
```

VFIO Driver

What is VFIO?

- A new user level driver framework for Linux
- **V**irtual **F**unction **I**/**O***
- Originally developed by Tom Lyon (Cisco)
- IOMMU-based DMA and interrupt isolation
- Full devices access (MMIO, I/O port, PCI config)
- Efficient interrupt mechanisms
- Modular IOMMU and device backends

VFIO Driver

- VFIO is a device driver
  - supports modular device driver backends
  - vfio-pci binds to non-bridge PCI devices
  - pci-stub available as “no access” driver
    - Allows admins to restrict access within a group
  - Users cannot attempt to use in-service host devices
  - Devices in use by users cannot be simultaneously claimed by other host drivers

VFIO Driver

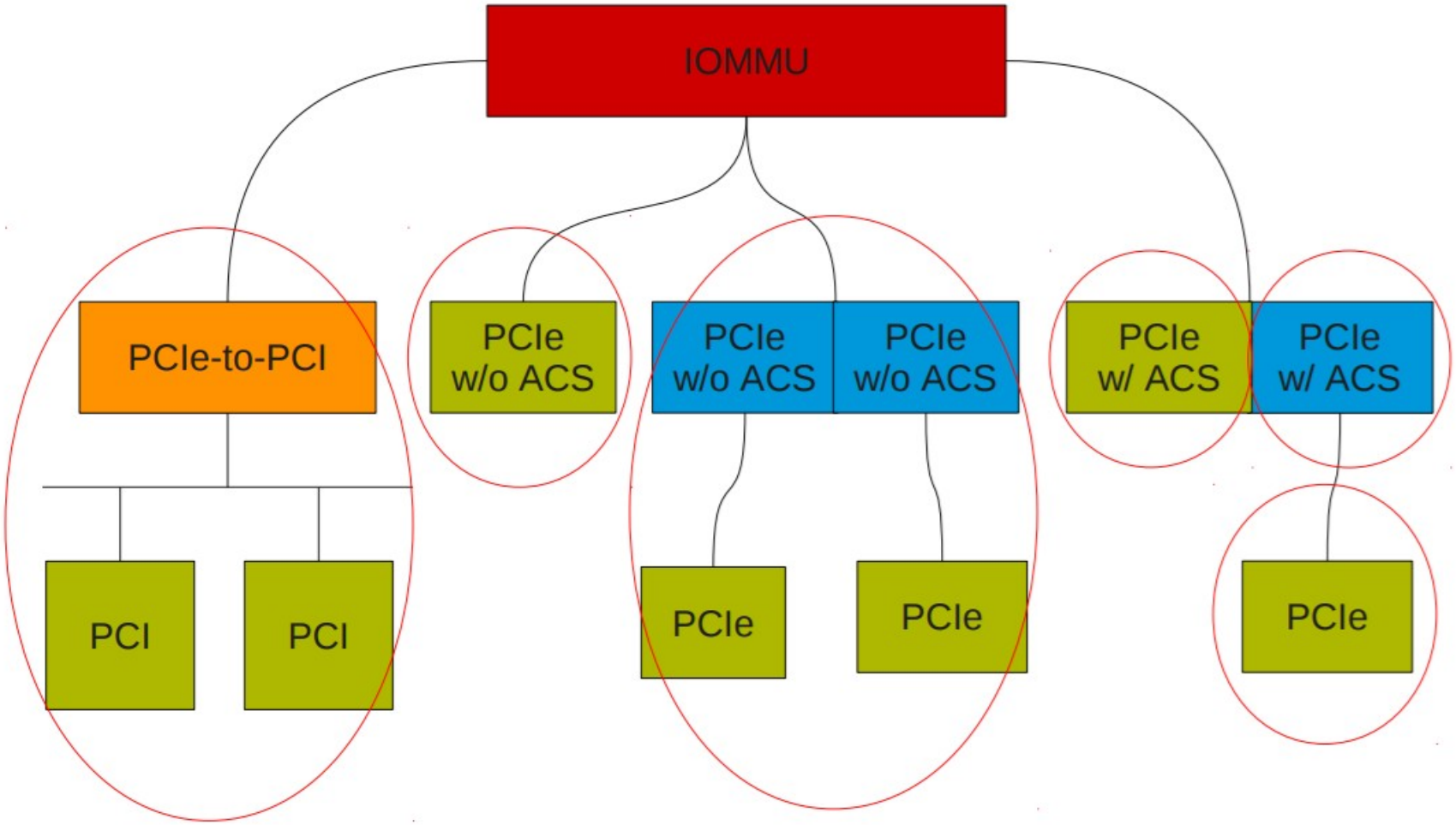
- VFIO device backends provide secure resource access
  - No device access without IOMMU isolation
  - Integral to the interface
    - Not outsourced to pci-sysfs
  - Virtualized access to PCI config space

VFIO Driver

- VFIO uses IOMMU groups
  - Allows the IOMMU driver to define both visibility and containment
  - Solves devices hidden by bridges
    - IOMMU cannot differentiate devices behind PCI bridge
  - Solves peer-to-peer back channels
    - All transactions required to reach IOMMU for translation
    - For PCIe, ACS (Access Control Services) indicates support
  - Result is better **security**

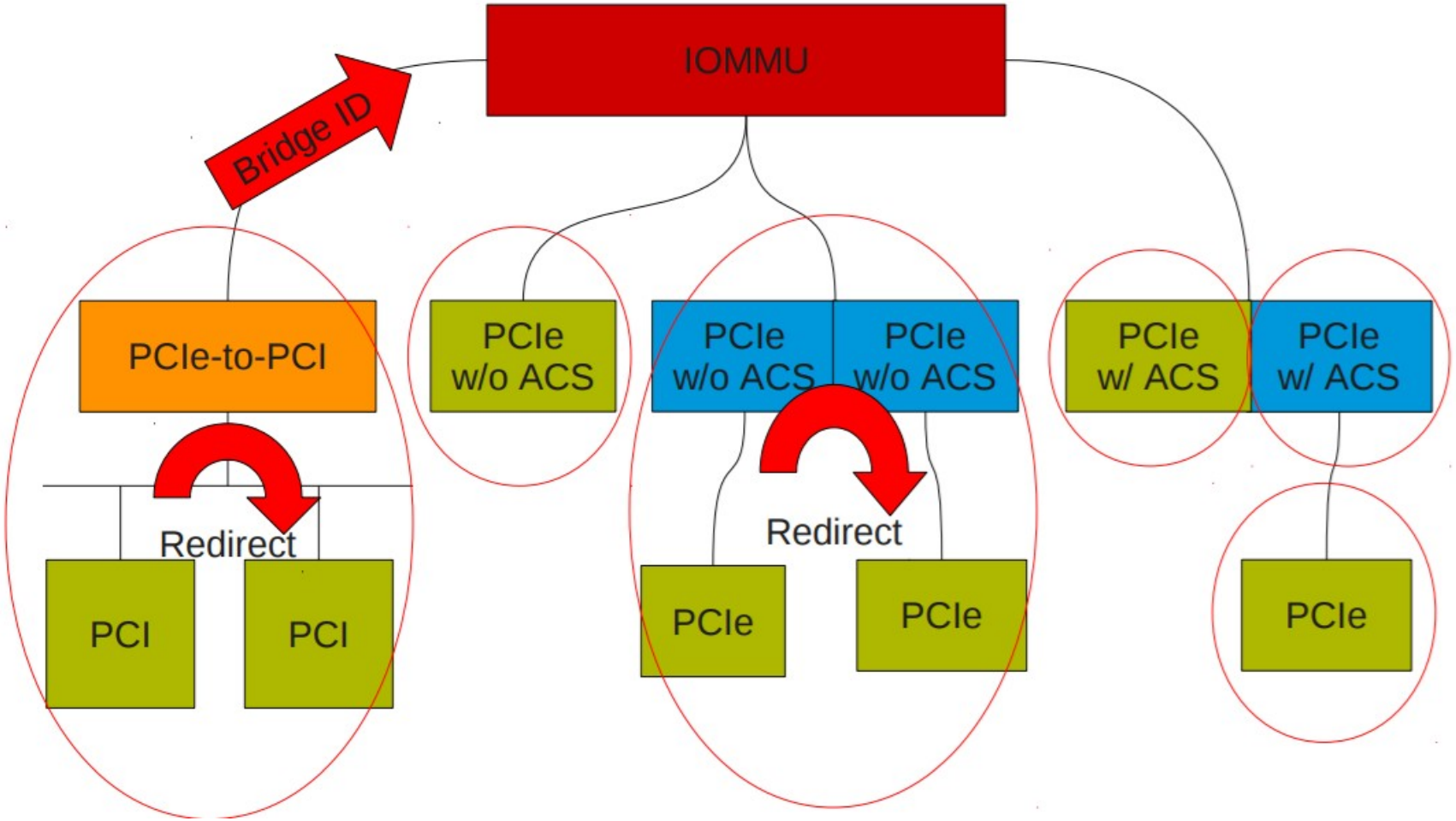
VFIO Driver

IOMMU Group examples



VFIO Driver

IOMMU Group examples



VFIO Driver

- VFIO moves ownership to the group level
  - Access to device file grants ownership
  - Ownership extends to all devices within the group
  - All accesses through VFIO
- VFIO supports a modular IOMMU interface
  - IOMMU API (type1) implemented
  - POWER (SPAPR) under development
- VFIO supports a modular device interface
  - PCI (vfio-pci) implemented
- VFIO has no KVM dependencies
  - Used only for acceleration
  - Non-x86 guests on x86 host work today
    - ppc g3beige – Big Endian driver test platform!
    - Any guest platform with PCI support

VFIO Driver

- VFIO moves ownership to the group level
  - Access to device file grants ownership
  - Ownership extends to all devices within the group
  - All accesses through VFIO
- VFIO supports a modular IOMMU interface
  - IOMMU API (type1) implemented
  - POWER (SPAPR) under development
- VFIO supports a modular device interface
  - PCI (vfio-pci) implemented
- VFIO has no KVM dependencies
  - Used only for acceleration
  - Non-x86 guests on x86 host work today
    - ppc g3beige – Big Endian driver test platform!
    - Any guest platform with PCI support

VFIO Driver

- AMD-Vi or Intel VT-d capable hardware
- Linux 3.6+ host
  - CONFIG_VFIO_IOMMU_TYPE1=m
  - CONFIG_VFIO=m
  - CONFIG_VFIO_PCI=m
  - modprobe vfio-pci

VFIO Driver

- Device to assign:

```
01:10.0 Ethernet controller: Intel Corporation 82576
Virtual Function (rev 01)
```

- Find the group:

```
$ readlink /sys/bus/pci/devices/0000:01:10.0/iommu_group
../../../../kernel/iommu_groups/15
```

- IOMMU Group = 15
- Check the devices in the group:

```
$ ls /sys/bus/pci/devices/0000:01:10.0/iommu_group/devices/
0000:01:10.0
```

VFIO Driver

- Unbind from device driver

```
$ echo 0000:01:10.0 | sudo tee \
/sys/bus/pci/devices/0000:01:10.0/driver/unbind
```

- Find vendor & device ID

```
$ lspci -n -s 01:10.0
01:10.0 0200: 8086:10ca (rev 01)
```

- Bind to vfio-pci

```
$ echo 8086 10ca | sudo tee \
/sys/bus/pci/drivers/vfio-pci/new_id
```

- Check

```
$ ls /dev/vfio
15  vfio
```

VFIO Driver

- Unbind from device driver

```
$ echo 0000:01:10.0 | sudo tee \
/sys/bus/pci/devices/0000:01:10.0/driver/unbind
```

- Find vendor & device ID

```
$ lspci -n -s 01:10.0
01:10.0 0200: 8086:10ca (rev 01)
```

- Bind to vfio-pci

```
$ echo 8086 10ca | sudo tee \
/sys/bus/pci/drivers/vfio-pci/new_id
```

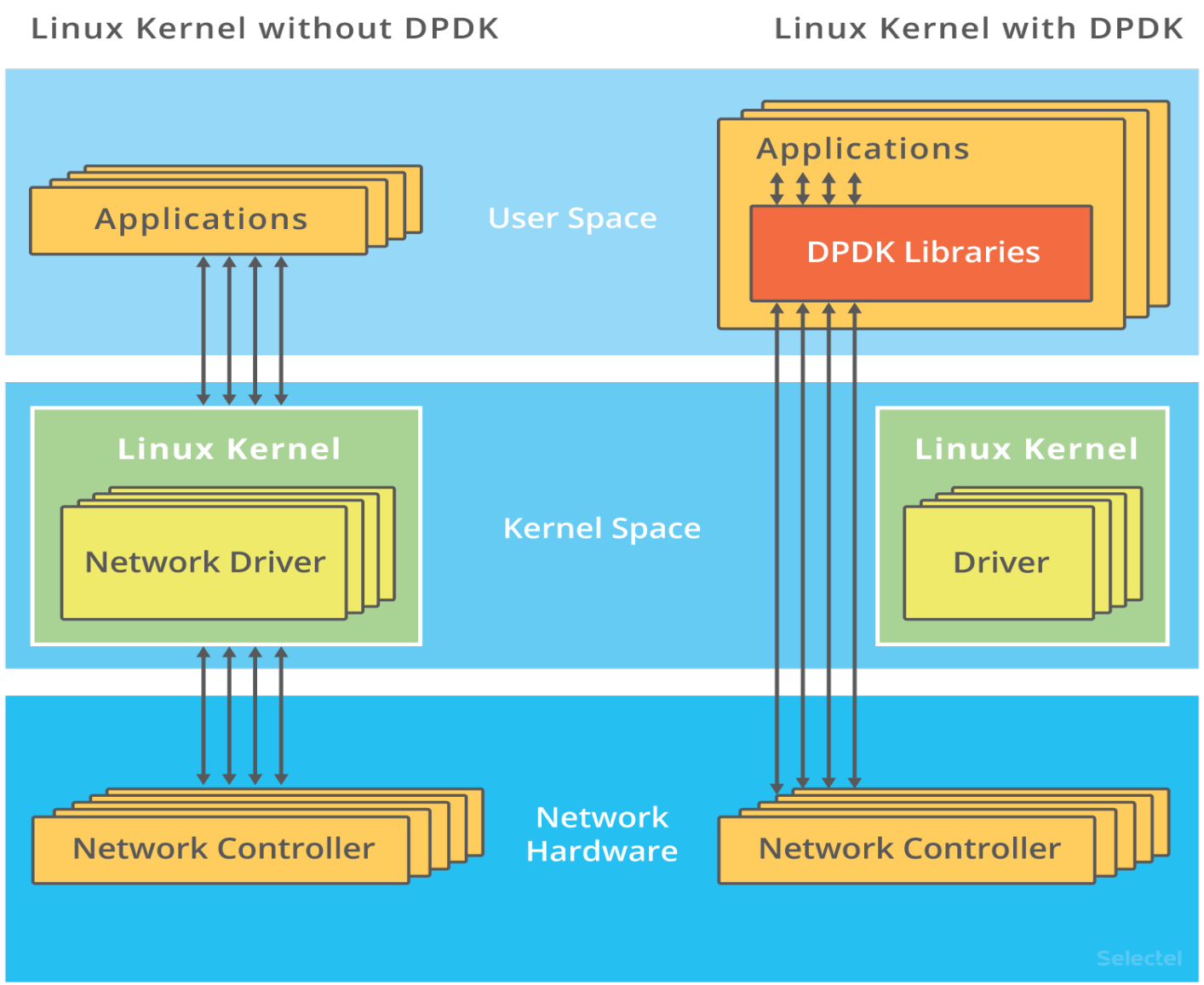
- Check

```
$ ls /dev/vfio
15  vfio
```

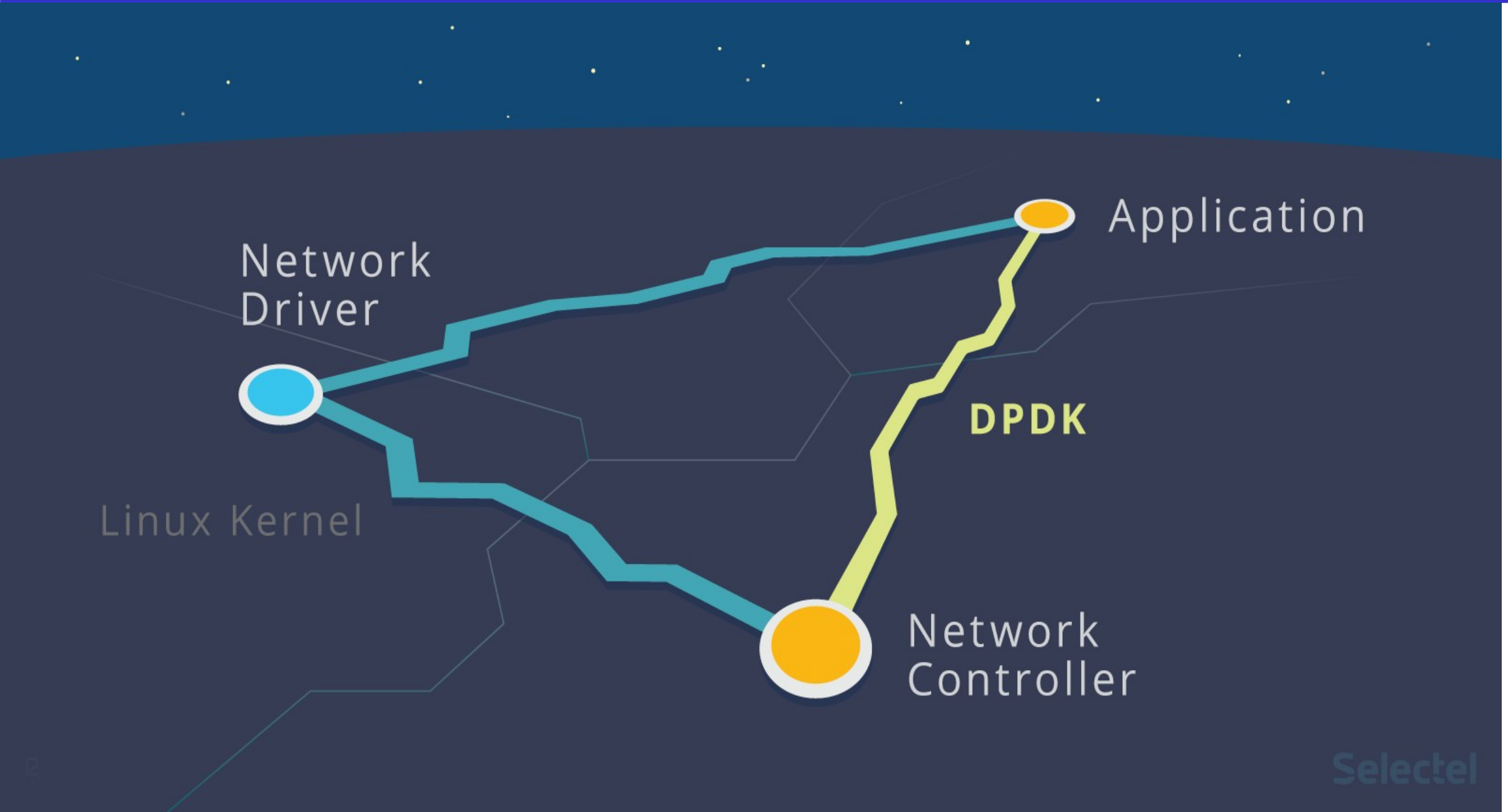
VFIO Driver

- The primary difference between uio and vfio is that vfio is capable of programming the platform's IOMMU, which is a critical piece of hardware for ensuring memory safety in user space drivers.
- In order for SPDK to take control of a device, it must first instruct the operating system to relinquish control. This is often referred to as unbinding the kernel driver from the device and on Linux is done by writing to a file in sysfs. SPDK then rebinds the driver to one of two special device drivers that come bundled with Linux - uio or vfio. These two drivers are "dummy" drivers in the sense that they mostly indicate to the operating system that the device has a driver bound to it so it won't automatically try to re-bind the default driver. They don't actually initialize the hardware in any way, nor do they even understand what type of device it is.

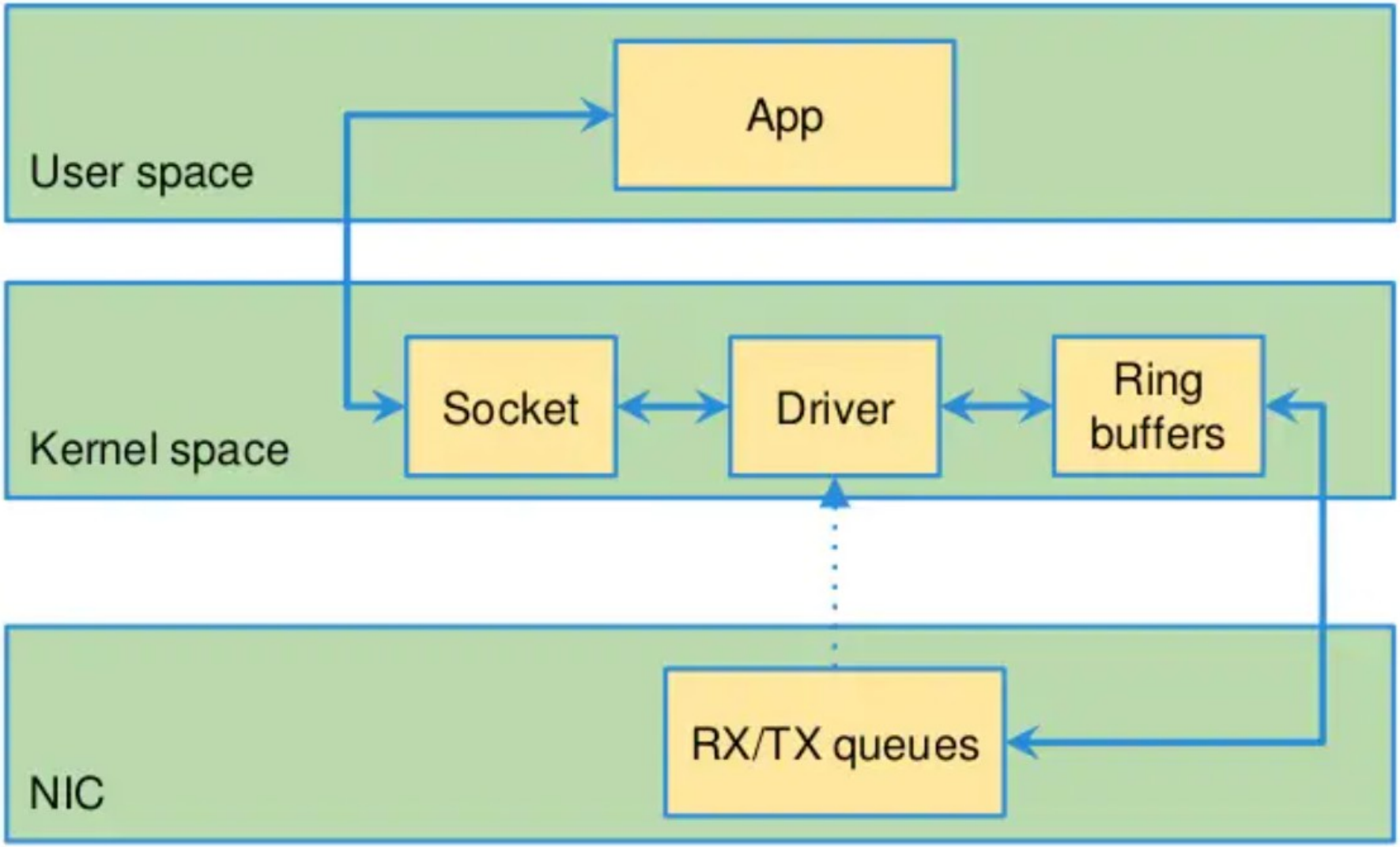
DPDK


Linux Kernel without DPDK
Linux Kernel with DPDK
Applications
User Space
Applications
DPDK Libraries
Linux Kernel
Network Driver
Kernel Space
Linux Kernel
Driver
Network Controller
Network Hardware
Network Controller
Selectel

DPDK


Application
Network
Driver
DPDK
Linux Kernel
Network
Controller
Selectel

DPDK


User space
App
Kernel space
Socket
Driver
Ring buffers
NIC
RX/TX queues

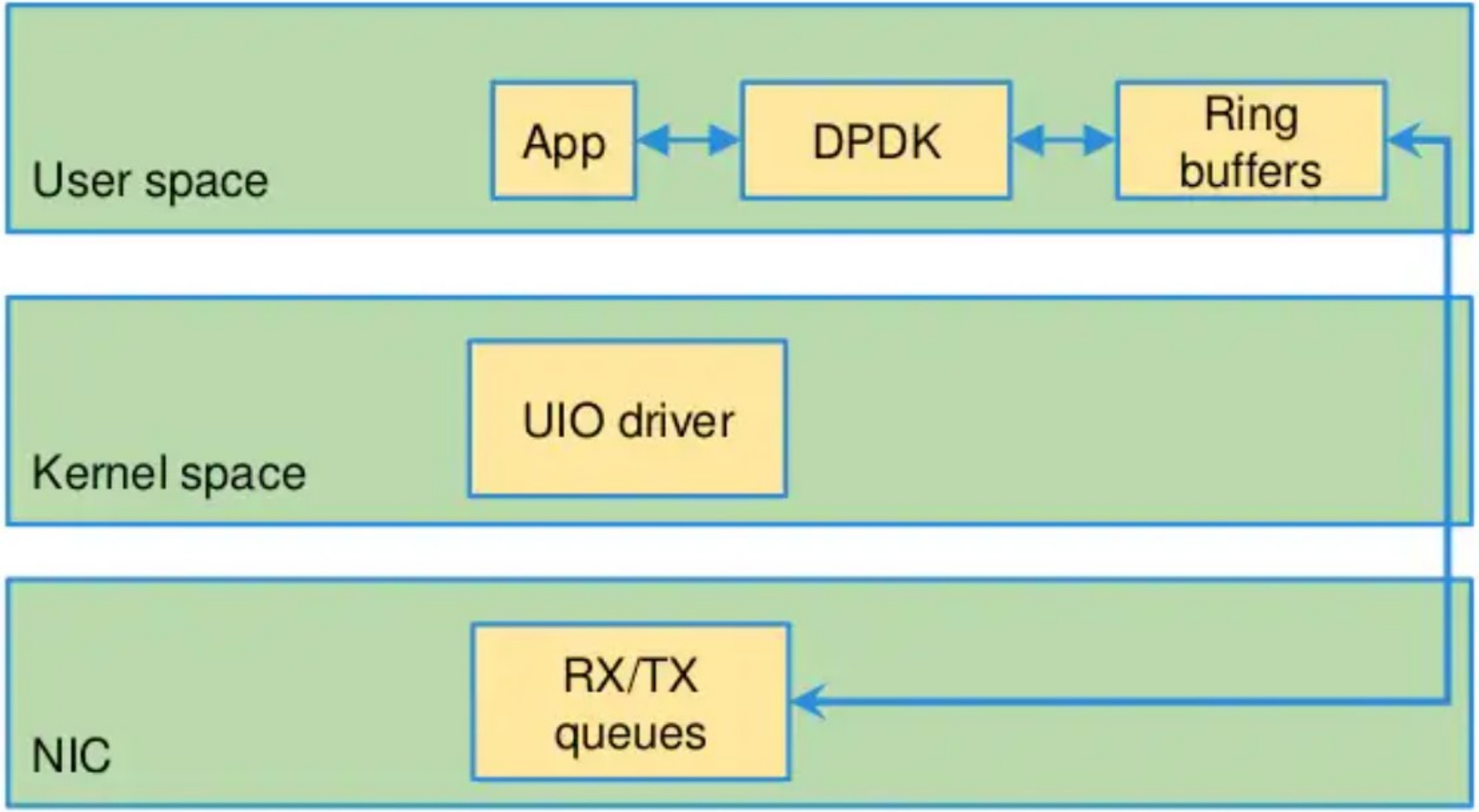
DPDK

System calls
Context switching on blocking I/O
Data copying from kernel to user space
Interrupt handling in kernel

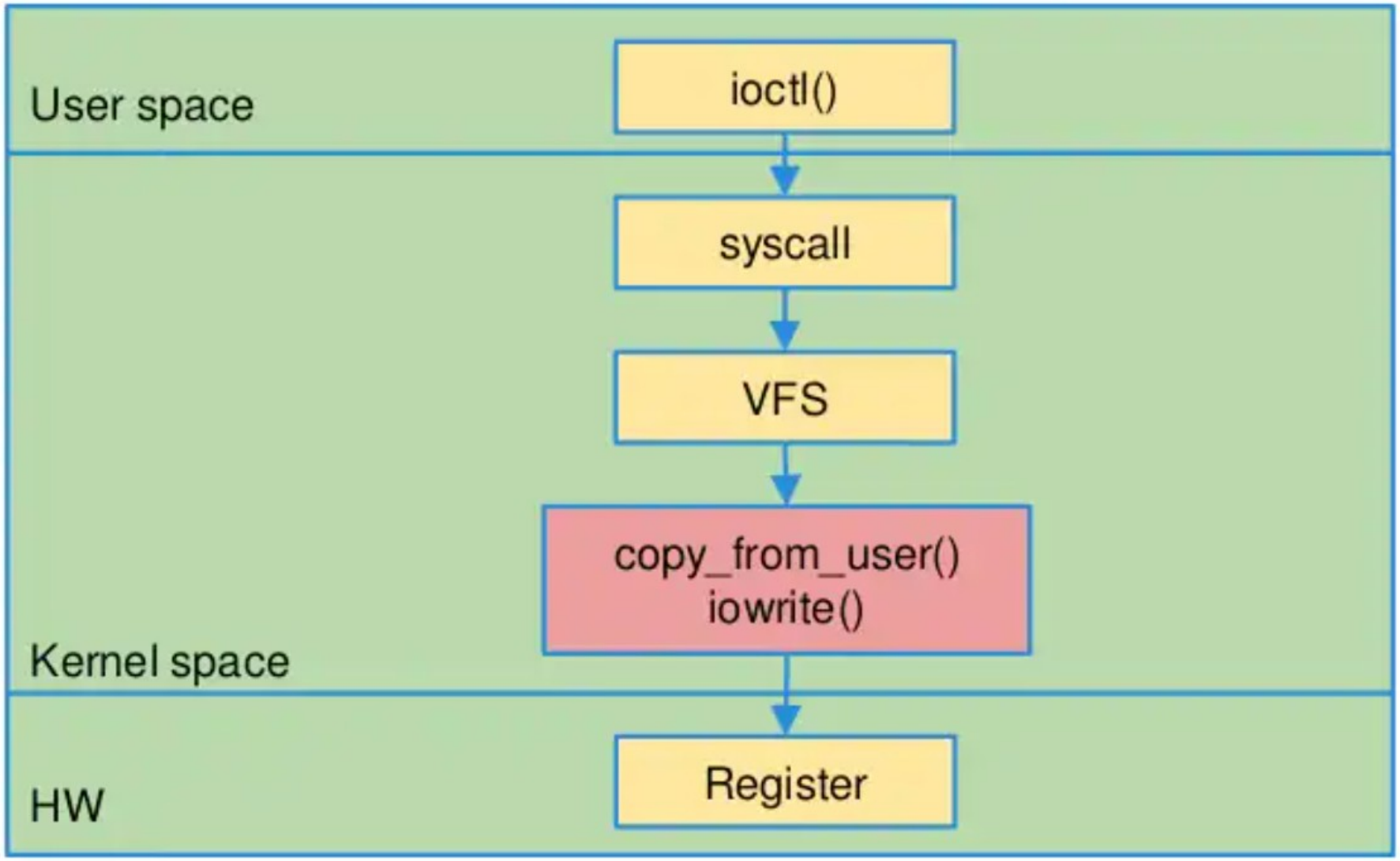
DPDK

Function	Activity	Time (ns)
sendto	system call	96
sosend_dgram	lock sock_buff, alloc mbuf, copy in	137
udp_output	UDP header setup	57
ip_output	route lookup, ip header setup	198
ether_otput	MAC lookup, MAC header setup	162
ixgbe_xmit	device programming	220
Total		**950**

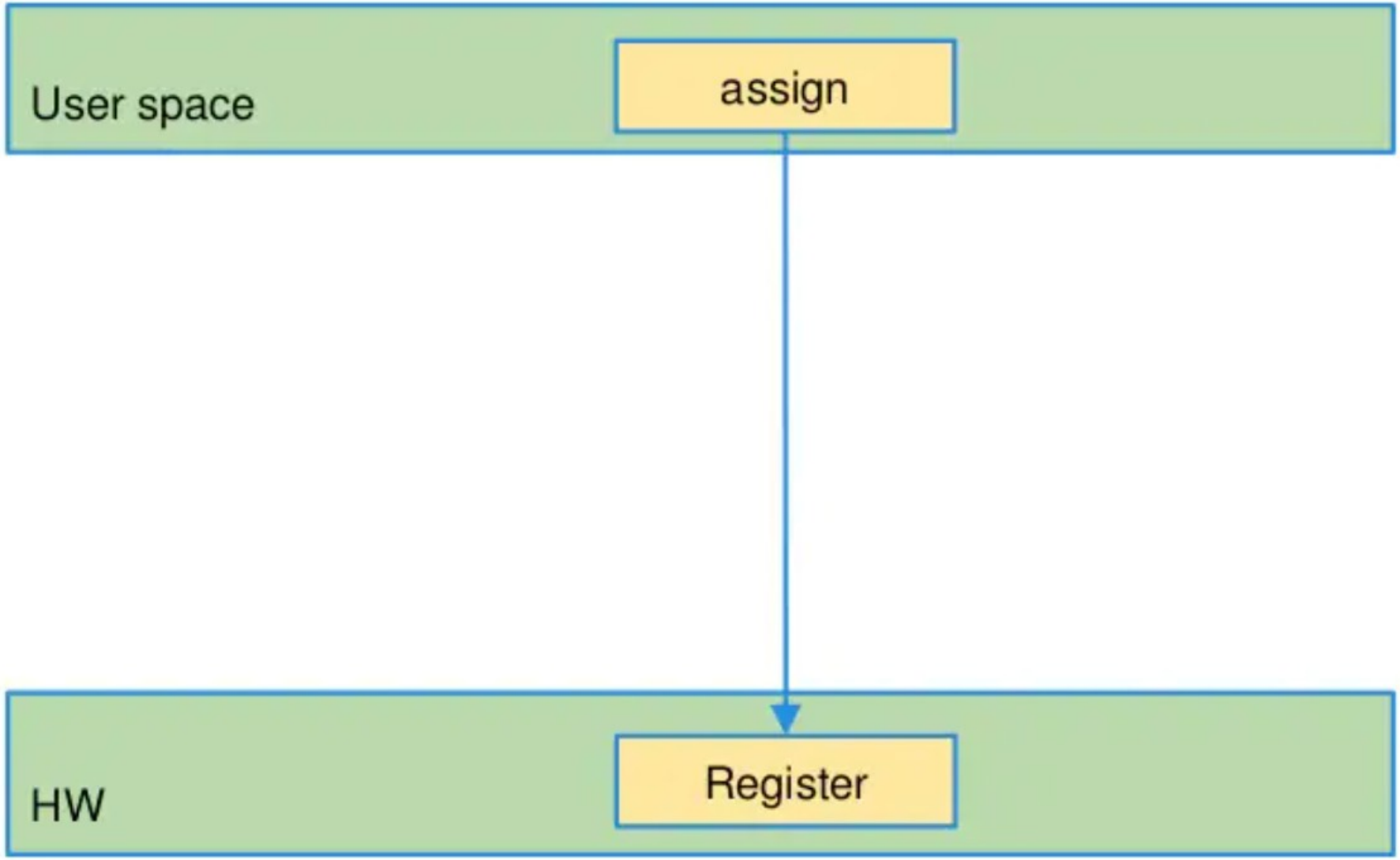
DPDK


User space
App
DPDK
Ring buffers
Kernel space
UIO driver
NIC
RX/TX queues

DPDK


User space
ioctl()
syscall
VFS
copy_from_user()
iowrite()
Kernel space
HW
Register

DPDK



DPDK

Uncompress DPDK and Browse Sources

First, uncompress the archive and move to the uncompressed DPDK source directory:

```
tar xJf dpdk-<version>.tar.xz
cd dpdk-<version>
```

The DPDK is composed of several directories:

lib: Source code of DPDK libraries
drivers: Source code of DPDK poll-mode drivers
app: Source code of DPDK applications (automatic tests)
examples: Source code of DPDK application examples
config, buildtools: Framework-related scripts and configuration

DPDK

Compiling and Installing DPDK System-wide

DPDK can be configured, built and installed on your system using the tools **meson** and **ninja**.

DPDK Configuration

To configure a DPDK build use:

meson <options> build

where “build” is the desired output build directory, and “<options>” can be empty or one of a number of meson or DPDK-specific build options, described later in this section. The configuration process will finish with a summary of what DPDK libraries and drivers are to be built and installed, and for each item disabled, a reason why that is the case. This information can be used, for example, to identify any missing required packages for a driver.

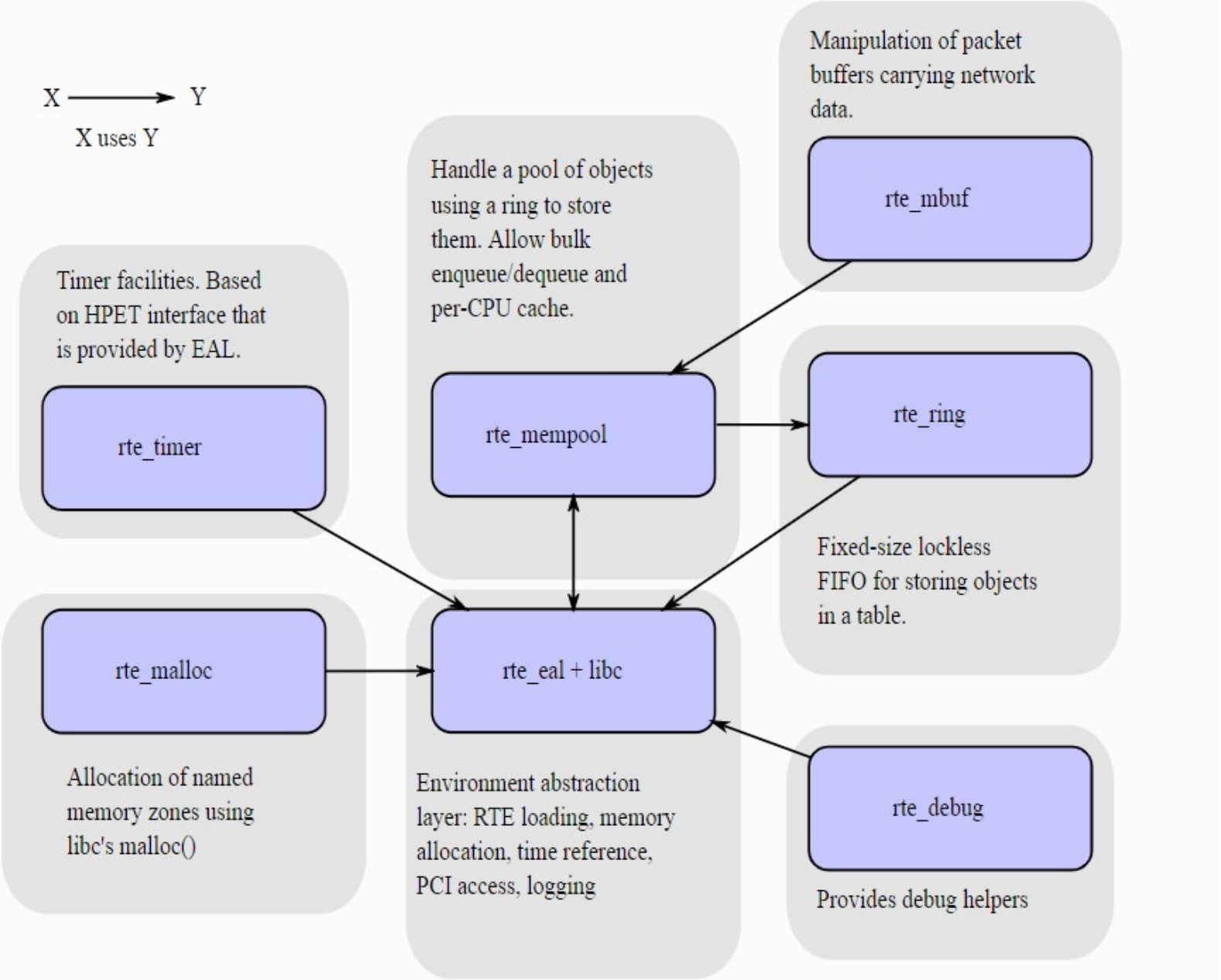
DPDK

Once configured, to build and then install DPDK system-wide use:

```
cd build
ninja
ninja install
ldconfig
```

The last two commands above generally need to be run as root, with the ninja install step copying the built objects to their final system-wide locations, and the last step causing the dynamic loader ld.so to update its cache to take account of the new objects.

DPDK


X → Y
X uses Y
Manipulation of packet buffers carrying network data.
rte_mbuf
Handle a pool of objects using a ring to store them. Allow bulk enqueue/dequeue and per-CPU cache.
Timer facilities. Based on HPET interface that is provided by EAL.
rte_timer
rte_mempool
rte_ring
Fixed-size lockless FIFO for storing objects in a table.
rte_malloc
rte_eal + libc
Allocation of named memory zones using libc's malloc()
Environment abstraction layer: RTE loading, memory allocation, time reference, PCI access, logging
rte_debug
Provides debug helpers

DPDK

Ring Manager (librte_ring)

The ring structure provides a lockless multi-producer, multi-consumer FIFO API in a finite size table. It has some advantages over lockless queues; easier to implement, adapted to bulk operations and faster. A ring is used by the Memory Pool Manager (librte_mempool) and may be used as a general communication mechanism between cores and/or execution blocks connected together on a logical core.

Memory Pool Manager (librte_mempool)

The Memory Pool Manager is responsible for allocating pools of objects in memory. A pool is identified by name and uses a ring to store free objects. It provides some other optional services, such as a per-core object cache and an alignment helper to ensure that objects are padded to spread them equally on all RAM channels.

DPDK

Network Packet Buffer Management (librte_mbuf)

The mbuf library provides the facility to create and destroy buffers that may be used by the DPDK application to store message buffers. The message buffers are created at startup time and stored in a mempool, using the DPDK mempool library.

This library provides an API to allocate/free mbufs, manipulate packet buffers which are used to carry network packets.

Timer Manager (librte_timer)

This library provides a timer service to DPDK execution units, providing the ability to execute a function asynchronously. It can be periodic function calls, or just a one-shot call. It uses the timer interface provided by the Environment Abstraction Layer (EAL) to get a precise time reference and can be initiated on a per-core basis as required.

DPDK

Ethernet* Poll Mode Driver Architecture

The DPDK includes Poll Mode Drivers (PMDs) for 1 GbE, 10 GbE and 40GbE, and para virtualized virtio Ethernet controllers which are designed to work without asynchronous, interrupt-based signaling mechanisms.

Packet Forwarding Algorithm Support

The DPDK includes Hash (librte_hash) and Longest Prefix Match (LPM,librte_lpm) libraries to support the corresponding packet forwarding algorithms.

librte_net

The librte_net library is a collection of IP protocol definitions and convenience macros. It is based on code from the FreeBSD* IP stack and contains protocol numbers (for use in IP headers), IP-related macros, IPv4/IPv6 header structures and TCP, UDP and SCTP header structures.

DPDK

Ethernet* Poll Mode Driver Architecture

The DPDK includes Poll Mode Drivers (PMDs) for 1 GbE, 10 GbE and 40GbE, and para virtualized virtio Ethernet controllers which are designed to work without asynchronous, interrupt-based signaling mechanisms.

Packet Forwarding Algorithm Support

The DPDK includes Hash (librte_hash) and Longest Prefix Match (LPM,librte_lpm) libraries to support the corresponding packet forwarding algorithms.

librte_net

The librte_net library is a collection of IP protocol definitions and convenience macros. It is based on code from the FreeBSD* IP stack and contains protocol numbers (for use in IP headers), IP-related macros, IPv4/IPv6 header structures and TCP, UDP and SCTP header structures.

DPDK

helloworld.c

IRQ

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. In simple word, Whenever a hardware such as a disk controller or ethernet card needs attention from CPU it sends a signal knows as Interrupt. The interrupt tells the CPU that something has happened and that the CPU should drop what it's doing to handle the event.

IRQ Request

In order to prevent multiple devices from sending the same interrupts, the IRQ system was established where each device in a computer system is assigned its own special IRQ so that its interrupts are unique. For example keyboard, USB, network card etc each have their own IRQ.

IRQ Affinity

IRQ affinity is an ability of Linux to assign (affinity) certain IRQs to specific processors (or groups of processors) i.e. particular IRQ will be handled only by a specific set of processors. It allows you to restrict or repartition the workload that your server must do so that it can more efficiently do its job.

IRQ Affinity

- The /proc/interrupts file lists the number of interrupts per CPU per I/O device. It displays the IRQ number, the number of that interrupt handled by each CPU core, the interrupt type, and a comma-delimited list of drivers that are registered to receive that interrupt.

- IRQs have an associated "affinity" property, smp_affinity, which defines the CPU cores that are allowed to execute the ISR for that IRQ. This property can be used to improve application performance by assigning both interrupt affinity and the application's thread affinity to one or more specific CPU cores. This allows cache line sharing between the specified interrupt and application threads.

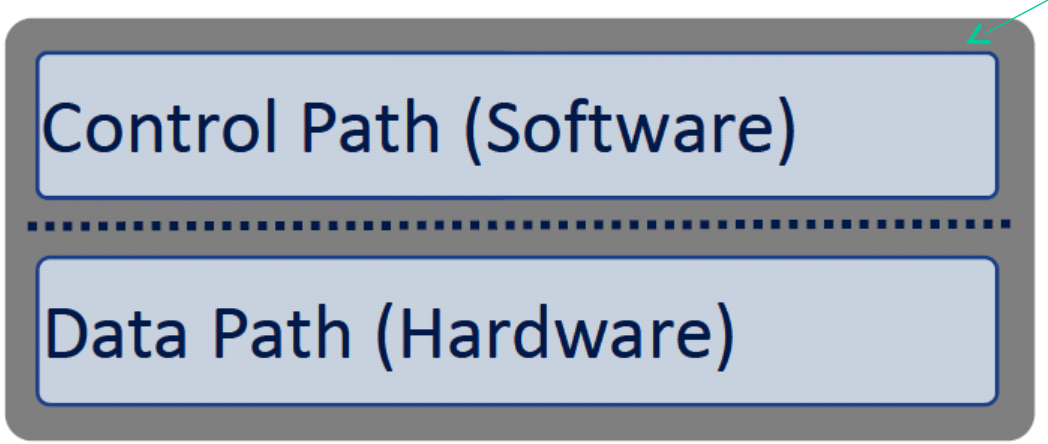
- The interrupt affinity value for a particular IRQ number is stored in the associated /proc/irq/IRQ_NUMBER/smp_affinity file, which can be viewed and modified by the root user. The value stored in this file is a hexadecimal bit-mask representing all CPU cores in the system.

SDN

- Software-defined networking (SDN) is an approach to computer networking that allows network administrators to manage network services through abstraction of lower-level functionality.
  - Abstractions for three problems: constrained forwarding model, distributed state, detailed configuration
- SDN is
  - Directly programmable: network control is programmable because it is decoupled from forwarding functions
  - Agile: administrator can dynamically adjust network-wide traffic flow to meet changing needs.
  - Centrally managed: network intelligence is logically centralized.
  - Programmatically configured
  - Open standards-based and vendor-neutral

SDN

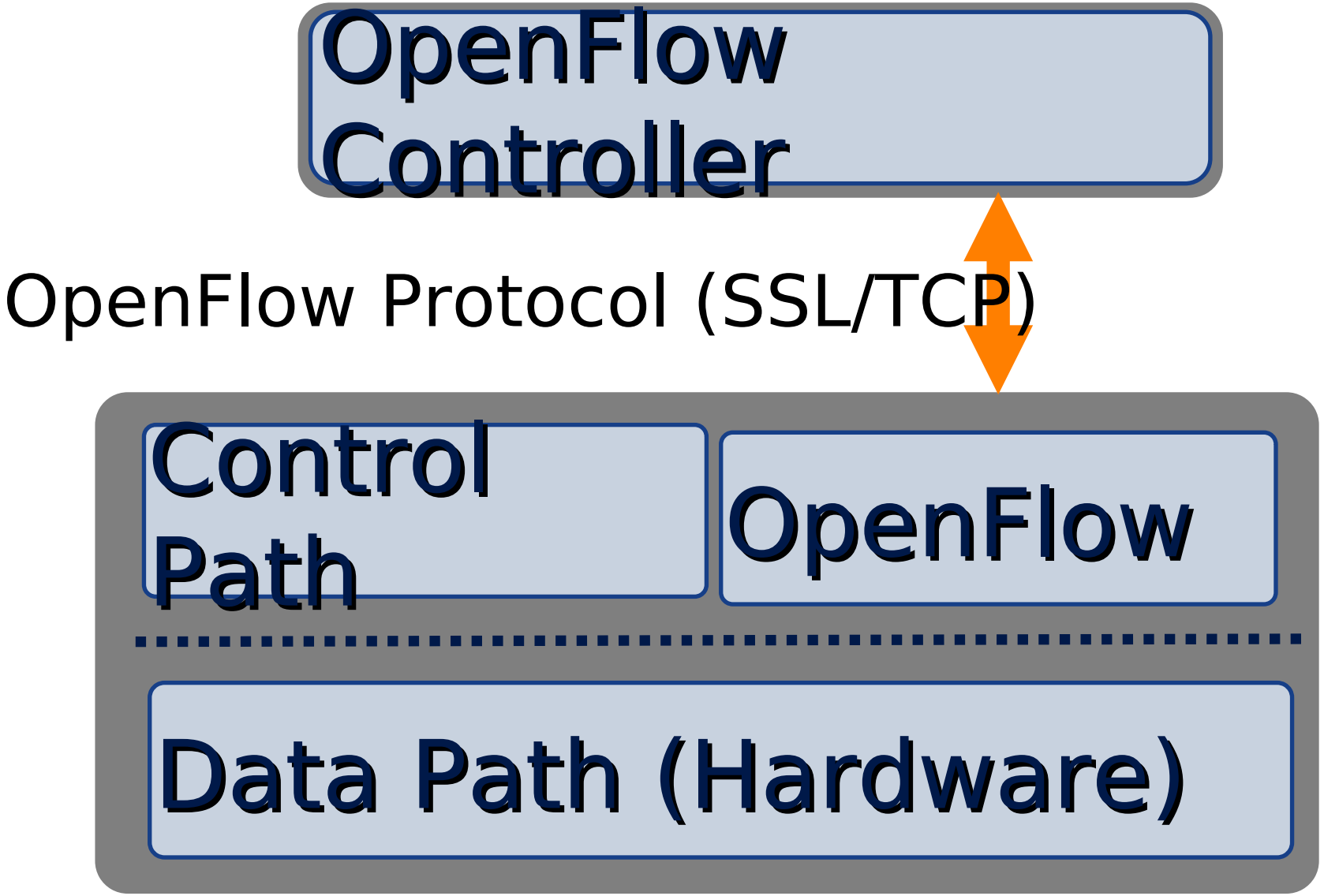
What sets the forwarding Table in Ethernet?



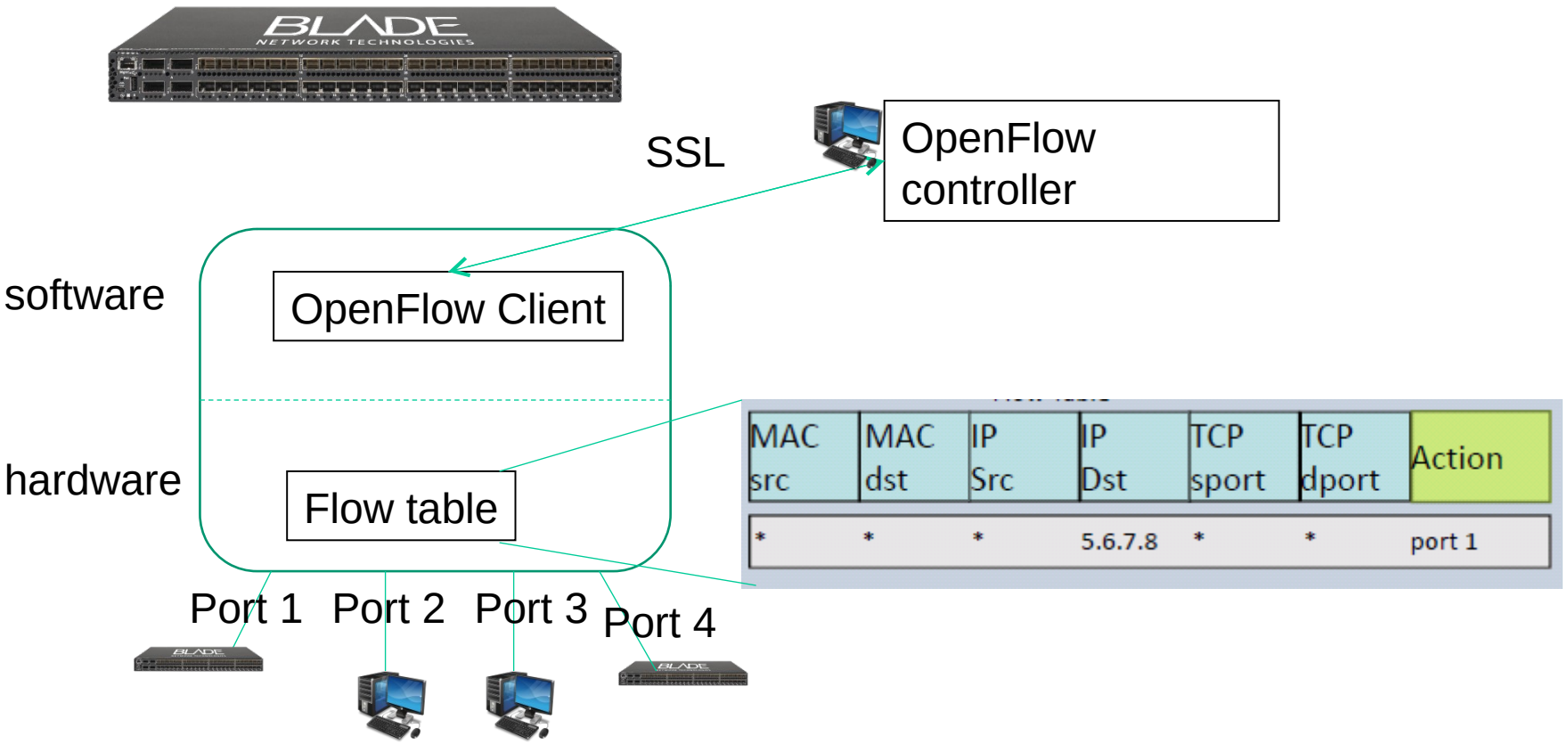
Forwarding table:

12:12:12:12:12:12	port 1
3f:13:33:ef:ff:ff	port 2

SDN


OpenFlow Controller
OpenFlow Protocol (SSL/TCP)
Control Path
OpenFlow
Data Path (Hardware)

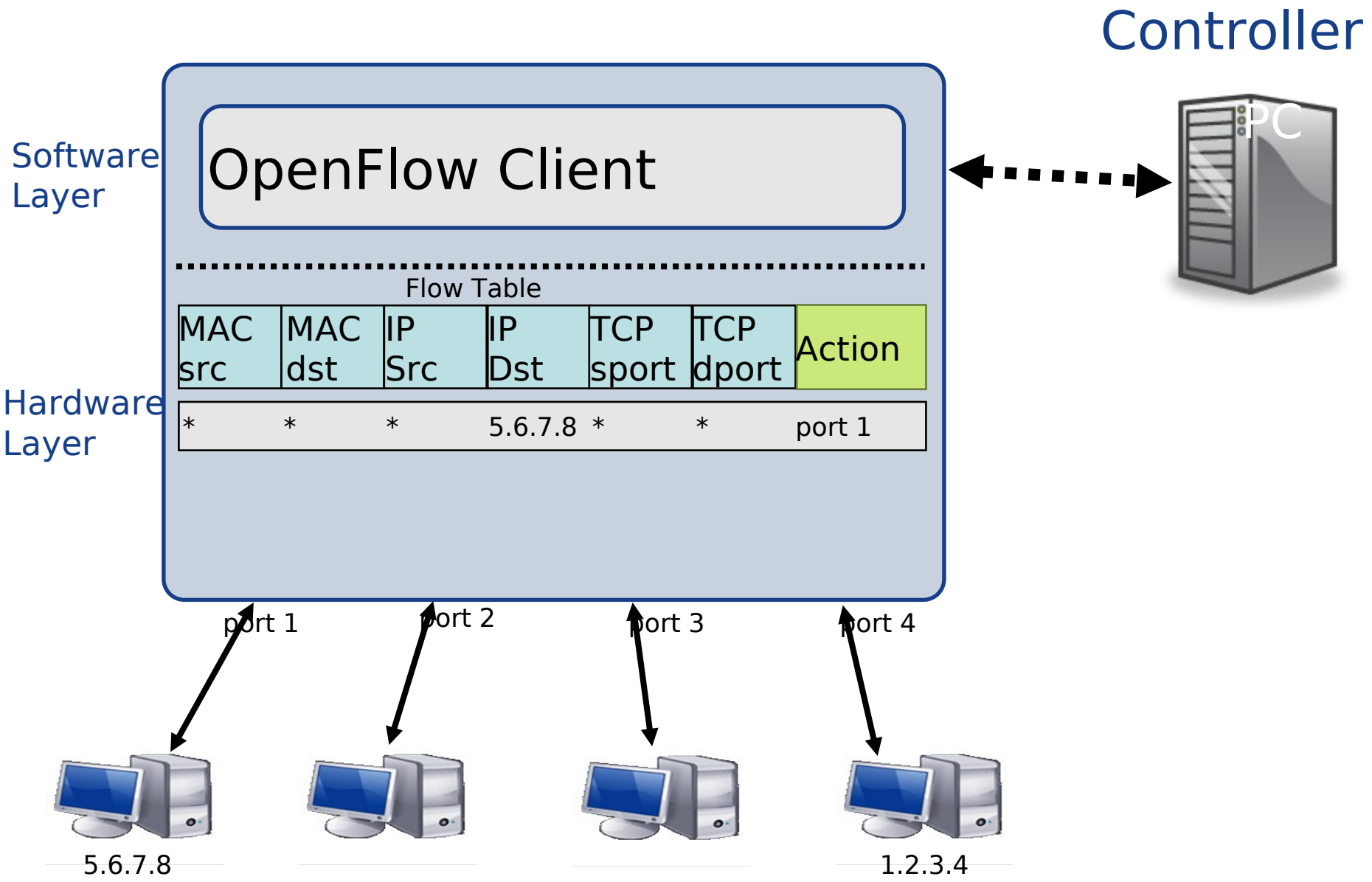
SDN


BLADE
NETWORK TECHNOLOGIES
SSL
OpenFlow controller
software
OpenFlow Client
hardware
Flow table
MAC src
MAC dst
IP Src
IP Dst
TCP sport
TCP dport
Action
* * * 5.6.7.8 * * port 1
Port 1
Port 2
Port 3
Port 4

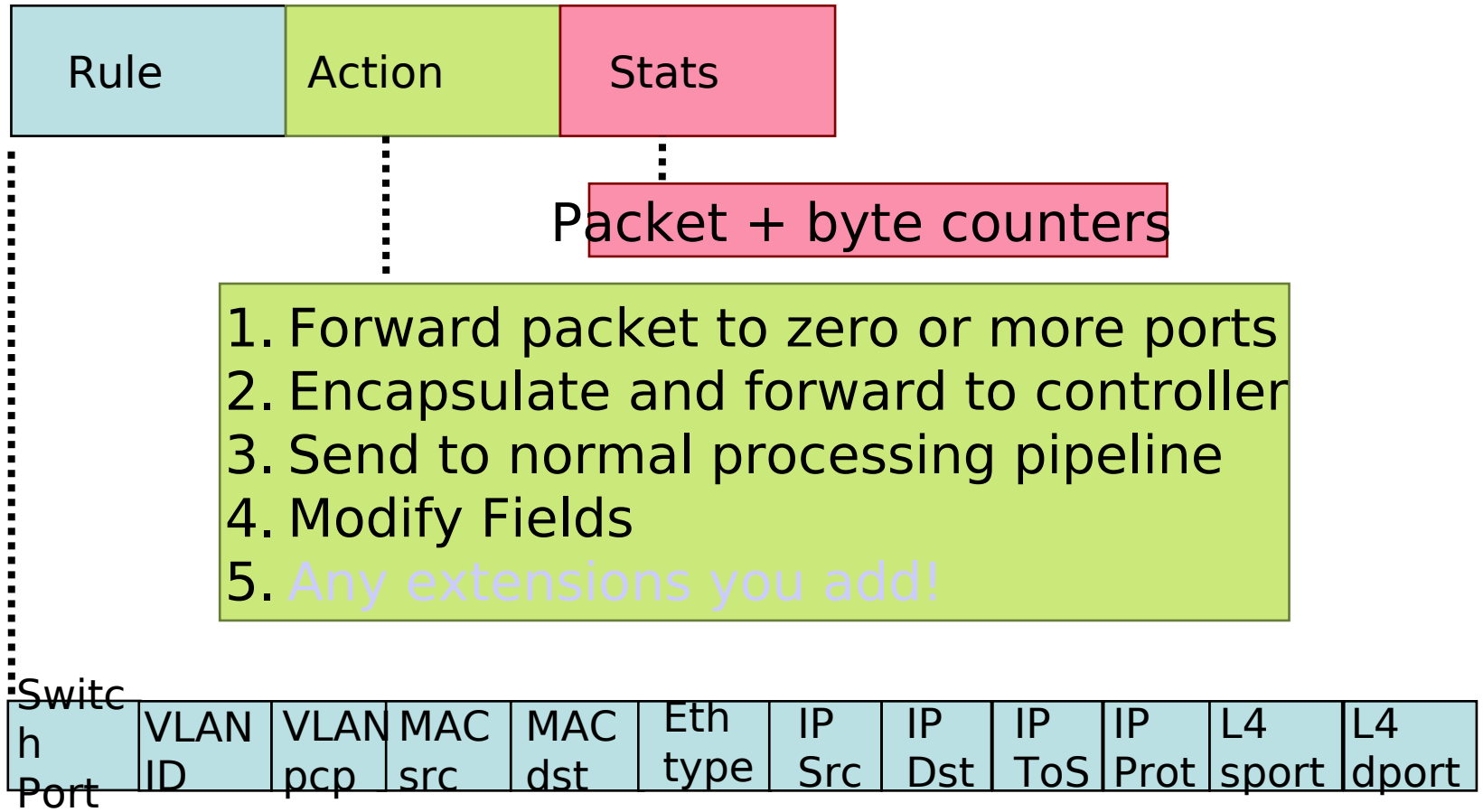
SDN

- An Openflow switch (Ethernet switch) has an internal flow table.
  - If a packet matches an entry in the flow table, perform the actions (e.g. forward to port 10) according to the flow table.
  - If a packet does not match any entry in the flow table. Send it to the Openflow controller
    - The controller will figure out what to do with such packet
    - The controller will then respond to the switch, informing how to handle such a packet so that the switch would know how to deal with such packets next time.
    - For each flow, ideally the controller will be queried once.
- Openflow defines the standard interface to add and remove flow entries in the table.

SDN


Controller
PC
Software Layer
OpenFlow Client
Flow Table
MAC src
MAC dst
IP Src
IP Dst
TCP sport
TCP dport
Action
Hardware Layer
* * * 5.6.7.8 * * port 1
port 1
port 2
port 3
port 4
5.6.7.8
1.2.3.4

SDN Rules



+ mask what fields to match

SDN

Switching

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	00:1f:..	*	*	*	*	*	*	*	port6

Flow Switching

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
port3	00:20..	00:1f..	0800	vlan1	1.2.3.4	5.6.7.8	4	17264	80	port6

Firewall

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	*	*	*	*	*	*	*	22	drop

SDN

Routing

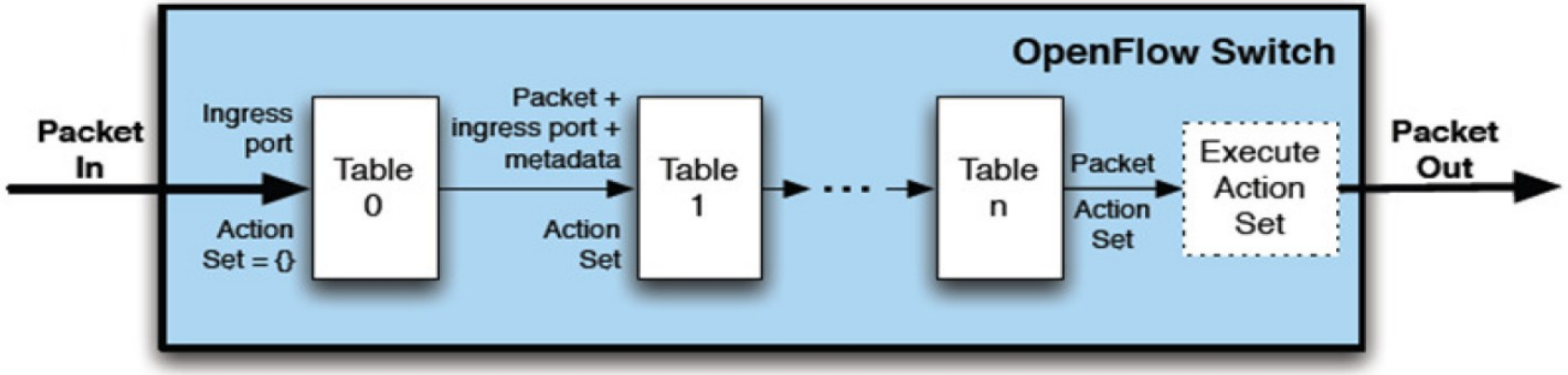
Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	*	*	*	*	5.6.7.8	*	*	*	port6

VLAN Switching

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	00:1f..	*	vlan1	*	*	*	*	*	port6, port7, port9

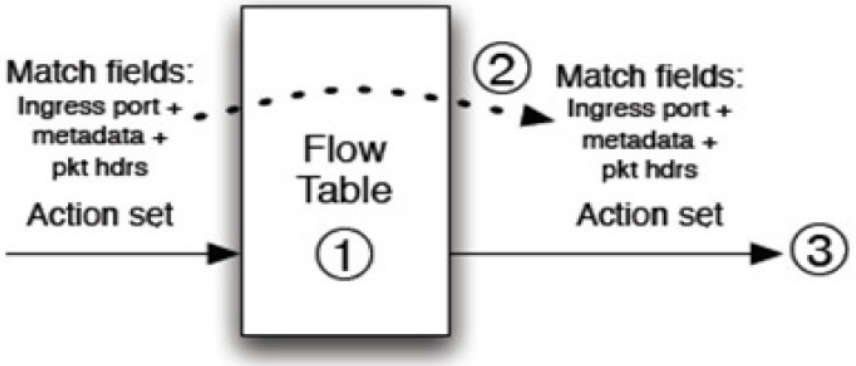
SDN

A switch can have multiple flow tables that are matched in a pipeline fashion.



(a) Packets are matched against multiple tables in the pipeline

SDN



① Find highest-priority matching flow entry

② Apply instructions:
- i. Modify packet & update match fields (apply actions instruction)
- ii. Update action set (clear actions and/or write actions instructions)
- iii. Update metadata

③ Send match data and action set to next table

(b) Per-table packet processing