Lab 05 - I/O Monitoring (Linux)

“Every thing is a file”, is a very famous **Linux** philosophy. There is a reason for this philosophy. It's because, Linux operating system considers and works with most of its devices, by the same way a file is opened or closed.

- Block devices (Hard-disks, Compact Disk, Floppy, Flash Memory)
- Character devices or serial devices (Mouse, keyboard)
- Network Devices

Objectives

- Offer an introduction to I/O monitoring.
- Get you acquainted with a few linux standard monitoring tools and their outputs, for monitoring the impact of the I/Os on the system.
- Gives an intuition to be able to compare two relatively similar systems, but I/O different.

Contents

Tasks

Introduction

Disk I/O subsystems are the slowest part of any Linux system. This is mainly due to their distance from the CPU and for the old HDD the fact that disk requires physics to work (rotation and seek). If the time taken to access disk as opposed to memory was converted into days and minutes, it is the difference between 7 days and 7 minutes. As a result, it is essential that the Linux kernel minimises the amount of I/O operations it generates on a disk.

The following subsections describe the different ways the kernel processes data I/O from disk to memory and back.

01. Reading and Writing Data - Memory Pages

The Linux kernel breaks disk I/O into pages. The default page size on most Linux systems is **4K**. It reads and writes disk blocks in and out of memory in 4K page sizes. You can check the page size of your system by using the time command in verbose mode and searching for the page size:
 # getconf PAGESIZE

02. Major and Minor Page Faults

Linux, like most UNIX systems, uses a **virtual memory layer** that maps into physical address space. This mapping is **“on-demand”** in the sense that when a process starts, the kernel only maps what is required. When an application starts, the kernel searches the CPU caches and then physical

memory. If the data does not exist in either, the kernel issues a **Major Page Fault** (MPF). A MPF is a request to the disk subsystem to retrieve pages of the disk and buffer them in RAM.
Once memory pages are mapped into the buffer cache, the kernel will attempt to use these pages resulting in a **Minor Page Fault** (MnPF). A MnPF saves the kernel time by reusing a page in memory as opposed to placing it back on the disk.
To find out how many MPF and MnPF occurred when an application starts, the time command can be used:

```
# /usr/bin/time –v evolution
```

As an alternative, a more elegant solution for a specific pid is:

```
# ps -o min_flt,maj_flt ${pid}
```

03. The File Buffer Cache

The **file buffer cache** is used by the kernel to **minimise MPFs and maximise MnPFs**. As a system generates I/O over time, this buffer cache will continue to grow as the system will leave these pages in memory until memory gets low and the kernel needs to "**free**" some of these pages for other uses. The result is that many system administrators see low amounts of free memory and become concerned when in reality, the system is just making good use of its caches 😉

04. Types of Memory Pages

There are **3** types of memory pages in the Linux kernel:

- **Read Pages** – Pages of data read in via disk (MPF) that are read only and backed on disk. These pages exist in the Buffer Cache and include **static files**, **binaries**, and **libraries** that do not change. The Kernel will continue to page these into memory as it needs them. If the system becomes short on memory, the kernel will "steal" these pages and place them back on the free list causing an application to have to MPF to bring them back in.
- **Dirty Pages** – Pages of data that have been modified by the kernel while in memory. These pages need to be synced back to disk at some point by the pdflush daemon. In the event of a memory shortage, kswapd (along with pdflush) will write these pages to disk in order to make room in memory.
- **Anonymous Pages** – Pages of data that do belong to a process, but do not have any file or backing store associated with them. They can't be synchronised back to disk. In the event of a memory shortage, kswapd writes these to the swap device as temporary storage until more RAM is free ("swapping" pages).

05. Writing Data Pages Back to Disk

Applications themselves may choose to write **dirty pages** back to disk immediately using the **fsync()** or **sync()** system calls. These system calls issue a direct request to the **I/O scheduler**. If an application does not invoke these system calls, the pdflush kernel daemon runs at periodic intervals and writes pages back to disk.

Monitoring I/O

Certain conditions occur on a system that may create I/O bottlenecks. These conditions may be identified by using a standard set of system monitoring tools. These tools include **top**, **vmstat**, **iostat**, and **sar**. There are some similarities between the outputs of these commands, but for the most part, each offers a unique set of output that provides a different aspect on performance. The following subsections describe conditions that cause **I/O bottlenecks**.

Calculating IOs Per Second

Every I/O request to a disk takes a certain amount of time. This is due primarily to the fact that a *disk must spin* and *a head must seek*. The spinning of a disk is often referred to as "**rotational delay**" (RD 😎) and the moving of the head as a "**disk seek**" (DS). The time it takes for each I/O request is calculated by adding DS and RD. A disk's RD is fixed based on the RPM of the drive. An RD is considered half a revolution around a disk.
Each time an application issues an I/O, it takes an average of 8MS to service that I/O on a 10K RPM disk. Since this is a fixed time, it is imperative that the disk be as efficient as possible with the time it will spend reading and writing to the disk. The amount of I/O requests is often measured in I/Os Per Second (IOPS). The 10K RPM disk has the ability to push 120 to 150 (burst) IOPS. To measure the effectiveness of IOPS, divide the amount of IOPS by the amount of data read or written for each I/O.

Random vs Sequential I/O

The relevance of KB per I/O depends on the workload of the system. There are two different types of workload categories on a system: sequential and random.
Sequential I/O - The **iostat** command provides information on IOPS and the amount of data processed during each I/O. Use the **–x** switch with **iostat** (*iostat –x 1*). **Sequential workloads** require large amounts of data to be read sequentially and at once. These include applications such as enterprise databases executing large queries and streaming media services capturing data. With sequential workloads, the KB per I/O ratio should be high. Sequential workload performance relies on the ability to move large amounts of data as fast as possible. If each I/O costs time, it is imperative to get as much data out of that I/O as possible.
Random I/O - Random access workloads do not depend as much on size of data. They depend primarily on the amount of IOPS a disk can push. Web and mail servers are examples of random access workloads. The I/O requests are rather small. Random access workload relies on how many requests can be processed at once. Therefore, the amount of IOPS the disk can push becomes crucial.

When Virtual Memory Kills I/O

If the system does not have enough **RAM** to accommodate all requests, it must start to use the **SWAP** device. As file system I/Os, writes to the SWAP device are just as costly. If the system is extremely deprived of RAM, it is possible that it will create a paging storm to the SWAP disk. If the SWAP device is on the same file system as the data trying to be accessed, the system will enter into contention for the I/O paths. This will cause a complete **performance breakdown** on the system. If pages can't be read or written to disk, they will stay in RAM longer. If they stay in RAM longer, the kernel will need to free the RAM. The problem is that the I/O channels are so clogged that nothing can be done. This inevitably leads to a kernel panic and crash of the system.

The following **vmstat** output demonstrates a system under memory distress. It is writing data out to the swap device:

```
 procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b    swpd   free  buff  cache   si   so    bi    bo   in cs   us sy id wa
17  0    1250   3248 45820 1488472   30 132   992     0 2437 7657 23 50  0 23
11  0    1376   3256 45820 1488888   57 245   416     0 2391 7173 10 90  0 0
12  0    1582   1688 45828 1490228   63 131  1348    76 2432 7315 10 90  0 10
12  2    3981   1848 45468 1489824  185 56   2300    68 2478 9149 15 12  0 73
14  2    10385 2400 44484 1489732    0 87   1112    20 2515 11620 0 12  0 88
14  2    12671 2280 43644 1488816   76 51   1812   204 2546 11407 20 45 0 35
```

The previous output demonstrates a large amount of read requests into memory (**bi**). The requests are so many that the system is short on memory (**free**). This is causing the system to send blocks to the swap device (**so**) and the size of swap keeps growing (**swpd**). Also notice a large percentage of WIO time (**wa**). This indicates that the CPU is starting to slow down because of I/O requests. Furthermore, **id** represents the time spent idle and it is included in **wa**

To see the effect the swapping to disk is having on the system, check the swap partition on the drive using **iostat**.

```
# iostat -x 1

avg-cpu: %user %nice %sys %idle
          0.00  0.00 100.00 0.00

Device: rrqm/s wrqm/s   r/s w/s rsec/s wsec/s    rkB/s    wkB/s   avgrq-sz  avgqu-sz   await svctm %util
/dev/sda  0.00  1766.67 4866.67 1700.00 38933.33 31200.00 19466.67 15600.00 10.68    6526.67 100.56 5.08
3333.33
/dev/sda1 0.00 933.33   0.00    0.00     0.00  7733.33     0.00  3866.67  0.00 20.00 2145.07 7.37 200.00
/dev/sda2 0.00 0.00  4833.33    0.00 38666.67   533.33  19333.33  266.67  8.11 373.33 8.07   6.90  87.00
/dev/sda3 0.00 833.33  33.33 1700.00   266.67 22933.33    133.33 11466.67   13.38   6133.33 358.46 11.35
1966.67
```

Both the swap device (*/dev/sda1*) and the file system device (*/dev/sda3*) are contending for I/O. Both have high amounts of write requests per second (*w/s*) and high wait time (*await*) to low service time ratios (*svctm*). This indicates that there is **contention** between the two partitions, causing both to **underperform**.

Takeaways

- Any time the **CPU is waiting** on I/O, the **disks are overloaded**.
- Calculate the amount of **IOPS** your disks can sustain.
- Determine whether your applications require **random** or **sequential** disk access.
- Monitor slow disks by comparing **wait times** and **service times**.
- Monitor the swap and file system partitions to make sure that **virtual memory** is not contending for **filesystem I/O**.

Tasks

01. [10p] Rotational delay - IOPS calculations

Every disk in your storage system has a maximum theoretical IOPS value that is based on a formula. Disk performance and IOPS is based on three key factors:

- **Rotational speed**. Measured in RPM, mostly 7,200, 10,000 or 15,000 RPM. A higher rotational speed is associated with a higher-performing disk.
- **Average latency**. The time it takes for the sector of the disk being accessed to rotate into position under a read/write head.
- **Average seek time**. The time (in ms) it takes for the hard drive's read/write head to position itself over the track being read or written.
- **Average IOPS**: Divide 1 by the sum of the average latency in ms and the average seek time in ms (1 / (average latency in ms + average seek time in ms).

To calculate the **IOPS range** divide 1 by the sum of the average latency in ms and the average seek time in ms. The formula is:

```
average IOPS = 1 / (average latency in ms + average seek time in ms).
```

Let's calculate the Rotational Delay - RD for a 10K RPM drive:

- Divide 10000 RPM by 60 seconds: **10000/60 = 166 RPS**
- Convert 1 of 166 to decimal: **1/166 = 0.006 seconds per Rotation**
- Multiply the seconds per rotation by 1000 milliseconds (6 MS per rotation).
- Divide the total in half (RD is considered half a revolution around a disk): **6/2 = 3 MS**
- Add an average of 3 MS for seek time: **3 MS + 3 MS = 6 MS**
- Add 2 MS for latency (internal transfer): **6 MS + 2 MS = 8 MS**
- Divide 1000 MS by 8 MS per I/O: **1000/8 = 125 IOPS**

[10p] Task A - Calculate rotational delay

Calculate the rotational delay (RD) for a 5400 RPM drive.

02. [10p] Iostat

Parameteres for iostat:

- -x for extended statistics
- -d to display device stastistics only
- -m for displaying r/w in MB/s

```
$ iostat -xdm
```

Use iostat with -p for specific device statistics:

```
$ iostat -xdm -p sda
```

[10p] Task A - Monitoring the behaviour

- Run *iostat -x 1 5*.
- Considering the last two outputs provided by the previous command, calculate **the efficiency of IOPS** for each of them. Does the amount of data written per I/O **increase** or **decrease**?

How to do:

- Divide the kilobytes read (*rkB/s*) and written (*wkB/s*) per second by the reads per second (*r/s*) and the writes per second (*w/s*).
- If you happen to have quite a few loop devices in your **iostat** output, find out what they are exactly:

```
$ df -kh /dev/loop*
```

03. [10p] Iotop

Iotop is an utility similar to top command, that interfaces with the kernel to provide per-thread/process I/O usage statistics.

Debian/Ubuntu Linux install iotop

$ sudo apt-get install iotop

How to use iotop command

$ sudo iotop OR $ iotop

Supported options by iotop command:

Options	**Description**
–version	show program's version number and exit
-h, –help	show this help message and exit
-o, –only	only show processes or threads actually doing I/O
-b, –batch	non-interactive mode
-n NUM, –iter=NUM	number of iterations before ending [infinite]
-d SEC, –delay=SEC	delay between iterations [1 second]
-p PID, –pid=PID	processes/threads to monitor [all]
-u USER, –user=USER	users to monitor [all]
-P, –processes	only show processes, not all threads
-a, –accumulated	show accumulated I/O instead of bandwidth
-k, –kilobytes	use kilobytes instead of a human friendly unit
-t, –time	add a timestamp on each line (implies –batch)
-q, –quiet	suppress some lines of header (implies –batch)

[10p] Task A - Monitoring the behaviour

- Run iotop (install it if you do not already have it) in a separate shell showing only processes or threads actually doing I/O.
- Inspect the script code (dummy.sh) to see what it does.
- Monitor the behaviour of the system with iotop while running the script.
- Identify the PID and PPID of the process running the dummy script and kill the process using command line from another shell (sending SIGINT signal to both parent & child processes).
- Hint - How to get parent PID of a given process in GNU/Linux from command line?

04. [20p] Monitor I/O with vmstat and iostat

We said in the beginning that the disk I/O subsystems are the slowest part of any system. This is why the I/O monitoring is so important, maximizing the performance of the slowest part of a system resulting in an improvement of the performance of the entire system.

[10p] Task A - Script

Write a script that reads the data into memory and generates a text file 500 times larger, by concatenating the contents of the following novel olivertwist.txt to itself.

[10p] Task B - Monitoring behaviour

Now we want to analyze what is happening with the I/O subsystem during an expensive operation. Monitor the behavior of the system while running your script using **vmstat** and **iostat**.

Understanding vmstat IO section:

- **bi** - column reports the number of blocks received (or "blocks in") from a disk per second.
- **bo** - column reports the number of blocks sent ("blocks out") to a disk per second.

05. [20p] Multitool Comparison

Now we will see in a slightly different approach, as for more special situations, a classic tool is not enough as long as we want to have a more detailed analysis of the behavior of I/O mechanisms.

[10p] Task A - Different tool, same I/O

An example would be to create an infinite loop that **copies endlessly** (simulation of a demanding process for a long time) a **large file** (use one of the files obtained previously).
Put the command below in a script:

```
$ while true; do cp original_file1 copied_file2; done
```

Use several special monitoring tools for I/O to investigate the state of the system.

Choose 3-4 from here: https://www.golinuxcloud.com/monitor-disk-io-performance-statistics-linux/

[10p] Task B - Plot of Comparison

Plot a graph with the results obtained with iostat, iotop and one of the previously chosen tools. Interpret the graph and the values obtained using those tools.

- basic plot
- the values used should represent the same metric (eg: kb written per second)
- you can take the values manually or automatically
- standardize the values (kb/s)
- preferably use the matplotlib module in python

Fill the data you obtained into the python3 script in plot.zip.
Make sure you have **python3** and **python3-matplotlib** installed.

06. [30p] RAM disk

Linux allows you to use part of your RAM as a block device, viewing it as a hard disk partition. The advantage of using a RAM disk is the **extremely low latency** (even when compared to SSDs). The disadvantage is that all contents will be lost after a reboot.

There are two main types of RAM disks:

- **ramfs** - cannot be limited in size and will continue to grow until you run out of RAM. Its size can not be determined precisely with tools like **df**. Instead, you have to estimate it by looking at the “cached” entry from **free**'s output.
- **tmpfs** - newer than **ramfs**. Can set a size limit. Behaves exactly like a hard disk partition but can't be monitored through conventional means (i.e. **iostat**). Size can be precisely estimated using **df**.

[15p] Task A - Create RAM Disk

Before getting started, let's find out the file system that our root partition uses. Run the following command (T - print file system type, h - human readable):

```
$ df -Th
```

The result should look like this:

```
Filesystem     Type      Size  Used Avail Use% Mounted on
udev           devtmpfs  1.1G     0  1.1G   0% /dev
tmpfs          tmpfs     214M  3.8M  210M   2% /run
/dev/sda1      ext4      218G  4.1G  202G   2% / <- root partition
tmpfs          tmpfs     1.1G  252K  1.1G   1% /dev/shm
tmpfs          tmpfs     5.0M  4.0K  5.0M   1% /run/lock
tmpfs          tmpfs     1.1G     0  1.1G   0% /sys/fs/cgroup
/dev/sda2      ext4      923M   73M  787M   9% /boot
/dev/sda4      ext4      266G   62M  253G   1% /home
```

From the results, we will assume in the following commands that the file system is **ext4**. If it's not your case, just replace with what you have:

```
$ sudo mkdir /mnt/ramdisk
$ sudo mount -t tmpfs -o size=1G ext4 /mnt/ramdisk
```

If you want the RAM disk to persist after a reboot, you can add the following line to */etc/fstab*. Remember that its contents will still be lost.

```
tmpfs    /mnt/ramdisk    tmpfs    rw,nodev,nosuid,size=1G    0  0
```

That's it. We just created a 1Gb **tmpfs** ramdisk with an **ext4** file system and mounted it at */mnt/ramdisk*. Use **df** again to check this yourself.

[15p] Task B - Pipe View & RAM Disk

As we mentioned before, you can't get I/O statistics regarding **tmpfs** since it is not a real partition. One solution to this problem is using **pv** to monitor the progress of data transfer through a pipe. This is a valid approach only if we consider the disk I/O being the bottleneck.

Next, we will generate 512Mb of random data and place it in */mnt/ramdisk/file* first and then in */home/student/file*. The transfer is done using **dd** with 2048-byte blocks.

```
$ pv /dev/urandom | dd of=/mnt/ramdisk/rand  bs=2048 count=$((512 * 1024 * 1024 / 2048))
$ pv /dev/urandom | dd of=/home/student/rand bs=2048 count=$((512 * 1024 * 1024 / 2048))
```

Look at the elapsed time and average transfer speed. What conclusion can you draw?