Welcome gks-017 ! [home] [download] [exercises] [introduction] [status]

GridKA 2012 - IO/Data Management-Exercises Material

Exercises:

[back to top]

1.1 Time Measurements - exercise 1

Measure the execution time of the command

```
sleep 1
```

in the bash shell! This might sound **silly** but you will get a feeling that one can do big mistakes in trivial measurements. The answer is the command you issue in a bash prompt!

Hint

Use the time command in front of **sleep 1**.

Solution

You just run

```
time sleep 1
```

.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.1 Time Measurements - exercise 2

We are going to do IO measurements. Which type(s) of time measurements are more relevant in the context of IO measuremts:

1. REAL (real-time)
2. SYS (system time)
3. CPU (user cpu time)

You can answer REAL SYS CPU or any combination.

(answered)

Hint

Many IO operations are reflected in an increase of system time - therefore it is

interesting to measure the system time. In most cases we want to calculate IO rates like MB/s. What is **per second** in that case?

Solution

Certainly we need to do realtime measurements to compute IO rates. Nevertheless system time is interesting because the number of IO operations and IO bandwidth the OS can do is limited and these are reflected by system time measurements.

Answer provided:

done

[back to top]

1.1 Time Measurements - exercise 3

Why is the real-time of the process not exactly 1 second?

1. The **time** command itself consumes a considerable amount of realtime and changes the measurement result
2. The OS has to find the executable and libraries to run sleep - this adds few ms to the realtime
3. That is wrong - most of the time we measure exactly one second!

Hint

Try to think, how this time measurment is made in the OS. **Attention:** there are two time commands. If you run the bash shell you get the one implemented in bash, but there is also the GNU time command /usr/bin/time which shows the time with less resolution but can give you even more information than just cpu and real-time measurements. See the man page ...

Solution

```
[csc30 /home/peters/ > time sleep 1

real    0m1.002s
user    0m0.000s
sys     0m0.001s
```

The sleep is implemented as:

```
[pid 15971] nanosleep({1, 0}, NULL)     = 0
[pid 15971] exit_group(0)               = ?
```

But the time command measures also the preparation of the sleep executable - mapping the binary/loading shared libraries etc. - hence the small overhead. You can follow the timing using

```
 strace -ttt sleep 1
```

and see all the system calls with nanosecond resolution before the nanosleep system call actually executes. To measure the time implemented in bash (and not the GNU time /usr/bin/time) you must actually use:

```
strace -ttt bash -c "time sleep 1"
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.1 Time Measurements - exercise 4

Give a lower boundary in **ms** for the precision of a realtime measurement using time in the bash shell!

Hint

What is the smallest time you could measure? Do not consider the systematic shift of few **ms** you measure!

Solution

The bash **time** command prints 3 significant digits, therefore it cannot be more precise than the last digit. Internally time measurements have **ns** resolution.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.1 Time Measurements - exercise 5

What is the nature of the error which is given by the startup phase of the program we want to measure? How do you call such an error?

Hint

Errors can be of statistic or systematic nature. Moreover it can be proportional or independent of the value you measure, therefore your answer should contain some of these key words:

- statistic
- systematic
- proportional

Solution

The startup of the program to measure always adds few ms to the time we measure. This systematic shift is independent from the fact how long the program actually runs. On the contrarity the precision of the measurements given by the resolution of the **time** command (**1 ms**) is important for very short measurements of few **ms** while it can be neglected for long measurements e.g. seconds, minutes or hours.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.1 Time Measurements - exercise 6

How would you do a more precise realtime measurement if you implement the sleep command yourself as a compiled program? Here you cannot give a trivial answer. Try to imagine, then get a confirmation or not by the hint and solution and you terminate the exercise by typing **done**.

Hint

There is a system call to provide a **sleep 1** and there are system calls to measure times with the precision of **µs** or **ns**! How does your program looks like?

Solution

gettimeofday measures realtime in micro seconds, **clock_gettime** in nanoseconds. Do the measurement **inside** the program to measure. Program Structure:

```
gettimeofday(start)
sleep(1)
gettimeofday(stop)
calculate (stop - start)
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.2 top/vmstat - exercise 1

Identify the process consuming most of your physical memory sorting the output of the **top** command by memory consumption! How do you do that?

Hint

Execute **top**, then press **M**. Just in case: you can leave top with **q**.

Solution

The top process in the displayed list is now the one with the highest physical (resident) memory consumption.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.2 top/vmstat - exercise 2

Switch back to the process CPU view and change the update frequency of the top window to 2 Hz! Which keys do you press?

Hint

You can always get information via **man top**. You sort again by CPU consumption via **P** and you can set the update frequency by pressing **d** and the time in seconds between updates.

Solution

You just have to type the key sequence **Pd0.5** or **Ps0.5**!

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.2 top/vmstat - exercise 3

Switch the top window to 1000Hz update rate. Get familiar with the command **vmstat** and run it in a second window with a 1Hz update frequency. How much CPU time is used by your **top** window?

Hint

You start top, then press the key sequence **s0.001[ENTER]**. You run vmstat with 1Hz like **vmstat -n 1**.

Solution

You can see rougly the CPU usage by comparing the CPU idle column of the vmstat output (3rd last) while the **top** window runs and after you stopped it by pressing **q** or Control-C. This is roughly few percent! However it is less if the window is invisble. This is just a warning that a measurement tool like **top** or the particuliary useful command **strace** can indeed influence time measurements.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.3 iostat/dd/strace - exercise 1

Run a **vmstat** and an **iostat** (use -x) command with 1 Hz update frequency in separate windows. Use now the **dd** command to create a 1MB file /tmp/1MB. Use a blocksize of 1 byte and repeat the measurement few times to verify the result. What is the IO rate you measure? Put a number in kB/s as answer.

Hint

You have to define the inputfile and outputfile via arguments. Use **/dev/zero** as input device, the local file as output device. Then you set the blocksize with additonal arguments: to write 1MB with 1 byte blocks you add the options **bs=1 count=1048576**.

Solution

Start **vmstat -n 1** and **iostat -x 1** .

```
for i in `seq 1 10`; do dd if=/dev/zero of=/tmp/1M bs=1 count=1MB; done

1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.16222 s, 528 kB/s
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.14328 s, 534 kB/s
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.18032 s, 527 kB/s
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.17776 s, 529 kB/s
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.19217 s, 531 kB/s
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.19899 s, 528 kB/s
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.20996 s, 529 kB/s
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.16939 s, 531 kB/s
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.15997 s, 532 kB/s
```

```
1048576+0 records in
1048576+0 records out
1048576 bytes (1.0 MB) copied, 2.17969 s, 529 kB/s
```

The average write speed is relativly low: only few hundred kB/s.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.3 iostat/dd/strace - exercise 2

Look at the output of vmstat and iostat during the previous exercise. How is it possible that it takes close to a second to write a small file? Shouldn't a harddisk be much faster? What is the limiting factor? Consider to use **strace** to inspect how our **dd** command is implemented. Where is the bottleneck?

Hint

The meaning of **bs=1** is revealed when you run **strace dd ...**

Solution

A blocksize of 1 byte means for the operating system one read system call and one write system call. To write 1MB/s we are roughly executing 2 Mio. system calls in a single second. This is reflected by a high system time usage. On the contrary our hard disk is staying more or less idle. To summarize: the rate of system calls is also a bootleneck in the IO system!

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.3 iostat/dd/strace - exercise 3

Inspect system calls using the strace command in front of the yes command. Compare the result when redirecting STDOUT to /dev/null (or a file). What is the difference? What happens during shell redirection?

Hint

To redirect STDOUT to a file you add to the command **> /tmp/myfile**. Compare the size of write system calls!

Solution

The size of the write calls changes with redirection from 2 to the page size of the operating system (4k). The buffering of STDOUT in libc is line-wise if it is connected to a terminal otherwise the default of the kernel buffer pipe buffer is 4k. You can verify this with by writing

```
strace yes 1234567890
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.3 iostat/dd/strace - exercise 4

Run the dd command creating a 10MB file for various blocksize: 1, 8, 16, 256, 4096, 256k, 1M. Draw **IO rate in MB/s vs blocksize**! When finished, give **done** as answer.

Hint

You just need to vary the **bs=..** parameter and make sure that the product **blocksize*count**=1MB.

Solution

```
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB bs=1 count=10M
10485760+0 records in
10485760+0 records out
10485760 bytes (10 MB) copied, 19.6962 seconds, 532 kB/s
root@gks-141:[~] $  dd if=/dev/zero of=/tmp/10MB bs=8 count=1280k
1310720+0 records in
1310720+0 records out
10485760 bytes (10 MB) copied, 2.58629 seconds, 4.1 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB bs=16 count=640k
655360+0 records in
655360+0 records out
10485760 bytes (10 MB) copied, 1.3099 seconds, 8.0 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB bs=256 count=40k
40960+0 records in
40960+0 records out
10485760 bytes (10 MB) copied, 0.10034 seconds, 105 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB bs=4k count=2560
2560+0 records in
2560+0 records out
10485760 bytes (10 MB) copied, 0.023658 seconds, 443 MB/s
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.3 iostat/dd/strace - exercise 5

Run the **yes** command and redirect the output into the file **/tmp/yes**. Inspect the vmstat and iostat windows while it runs and after it finishes. You can also try instead another tool called **dstat** which has simpler user interface. Which cache strategy is used by the OS when you write files:

1. no cache
2. write-through cache
3. write-back cache
   Cleanup the potentially huge file you created.

Hint

Pay attention to the block output value and the timing when the output to disk starts.

Solution

You observed, that the output to disk is delayed. The OS writes the output into the in-memory buffer cache and the disk writing starts with few seconds of delay. This implementation is called write-back. The write-back to the disk is asynchronous depending on the state of the cache and defined cache policies of the IO scheduler.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.4 cp - exercise 1

Use the strace command to inspect only open,read,write,close,stat calls for the following commands:

1. cp /etc/goup /tmp/group
2. cp /usr/bin/vim /tmp/vim Which IO pattern is used to do the copy e.g. what is the copy unit?

Hint

The strace command restricting to IO operations is **strace -e file,read,write**. Try to find the return value of the open command of the input file. This is the file descriptor used for that file. Then look for read calls on that file descriptor and the length specified for the read call.

Solution

The output of the strace command contains lines like that:

```
read(3, "L.count(value) -> integer -- ret"..., 4096) = 4096
write(4, "L.count(value) -> integer -- ret"..., 4096) = 4096
```

The **cp** command does 4k transactions e.g. it reads 4k, then writes 32k aso. ... 4k is the native page size of the Linux operatins system. On some more recent systems 32k are used as block size, on Mac OSX it is 64k.

```
bash-3.2$ getconf PAGESIZE
4096
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.4 cp - exercise 2

Give the realtime overhead in percent (answer just the plain number) comparing the second copy command in the previous exercise induced by running strace.

Hint

Measure the two numbers and express the overhead in percent e.g. if you measure something like 5s compared to 1s the overhead is 400 percent.

Solution

```
root@gks-141:[~] $ time cp /usr/bin/vim /tmp/1

real0m0.009s
user0m0.000s
sys0m0.009s
root@gks-141:[~] $ time strace cp /usr/bin/vim /tmp/1 2>/dev/null

real0m0.050s
user0m0.017s
sys0m0.033s
```

The overhead is huge but varies slightly with every run. In this case it is rougly 550 percent! So, do not measure real-time performance parameters in combination with strace.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

1.5 strace - exercise 1

strace is a very valuable tool to debug IO related problems in executables. Moreover it is usefull to understand binary programs without having source code at hand (e.g. you can attach strace to a running program like you do with a debugger).
Try to investigate with strace and top or vmstat what the execution of the mysterious program **/data/dm/bin/bash** does. Be careful not to loose track of it! Write some keywords of your observation as the answer (the answer is not evaluated automatically).

Hint

Run the command immedeatly with strace, you will understand that it is quite impossible to attach to it afterwards. Add also the **-f** switch to follow forks. Look for connect statements, file open and forks. Best practice is to redirect the output into a file and then step through it with an editor like vim,emacs or pico.

Solution

To trace this command you have to give the follow switch (strace -f), otherwise you don not see that a process detaches from the process group several times and continues to run: **strace -ttt -f /data/dm/bash >& /tmp/trace.log** a) It does CPU consuming computation. You can see that with top. b) It changes the PID/PGRP very quickly. 1020 different PIDs during runtime. c) It executes **wget of the index page in intervals of few seconds from google.com. It copies the /etc/passwd file into the /tmp/ directory. Both happens 10 times.**

Answer

Provide an answer for this exercise. You can just terminate the exercise by typing done or some text explaining your observation. Your response is not automatically evaluated

Finish exercise

[back to top]

2.1 Performance-Monitoring-Measurement-Caching - exercise 1

DISK WRITING
As you learned in the first exercise you can write a file F of length N with blocksize B containing only zeros like that:

```
dd if=/dev/zero of=F bs=B count=N divided by B
e.g. dd if=/dev/zero of=/tmp/1MB bs=32k count=64
```

DISK READING
Can you imagine how to read back the existing file /tmp/1MB using the dd command dumping the output into the **/dev/null** device? Can you do the same with a **cat** command? Investigate quickly if there is a difference in the IO pattern using strace on both commands. Just put **done** as the answer when you finished the exercise.

Hint

You can specify a file for reading with the **if=FILENAME** option. You can dump a file to STDOUT with the **cat** command and redirect the output to **/dev/null**.

Solution

```
dd if=/tmp/1MB of=/dev/null bs=4k count=512
cat /tmp/1MB > /dev/null
strace -e read,write dd if=/tmp/1MB of=/dev/null bs=4k count=512
..
read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
..
strace -e read,write cat /tmp/1MB > /dev/null
..
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
..
```

There is no difference in the IO pattern. **cat** also results in 32k read/writes.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.1 Performance-Monitoring-Measurement-Caching - exercise 2

Write sequential files using dd with 4k IO blocks with a size of 1MB, 10MB, 100MB, 1GB, 4GB into the /tmp/ directory with their size as filename. Does the real-time scale linear? Explain what you observe!

Hint

With every file we increase the size by a factor of ten. Linear scaling means, that also the real-time increases by a factor of 10 e.g. the performance stays the same.

Solution

```
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/1MB bs=4k count=256
```

```
256+0 records in
256+0 records out
1048576 bytes (1.0 MB) copied, 0.002632 seconds, 398 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/19MB bs=4k count=2560
2560+0 records in
2560+0 records out
10485760 bytes (10 MB) copied, 0.022512 seconds, 466 MB/s
root@gks-141:[~] $  dd if=/dev/zero of=/tmp/10MB bs=4k count=2560
2560+0 records in
2560+0 records out
10485760 bytes (10 MB) copied, 0.022477 seconds, 467 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/100MB bs=4k count=25600
25600+0 records in
25600+0 records out
104857600 bytes (105 MB) copied, 0.233277 seconds, 449 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/1GB bs=4k count=256000
256000+0 records in
256000+0 records out
1048576000 bytes (1.0 GB) copied, 3.22581 seconds, 325 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/4GB bs=4k count=1000000
1000000+0 records in
1000000+0 records out
4096000000 bytes (4.1 GB) copied, 73.4428 seconds, 55.8 MB/s
```

For 1M, 10M, 100M we measure pure memory copy to buffer cache performance, for files close to the memory size we asymptotically measure the pure disk performance. These values are very machine dependent (RAM, FS, harddisk, memory pressure etc.)

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.1 Performance-Monitoring-Measurement-Caching - exercise 3

Run first the command **dropcache**. Measure the time to read back the previously generated 5 files **two** times each using 4k blocksize with **dd**! How would you explain a large variation from the 1st to 2nd execution? Are all results compatible with the performance of a single hard disk? (The answer is not evaluated automatically).

Hint

How does the file size impact the performance? How big is your machine memory?

Solution

```
root@gks-141:[~] $ dropcache
root@gks-141:[~] $  time dd if=/tmp/1MB of=/dev/null bs=4k count=256
256+0 records in
256+0 records out
1048576 bytes (1.0 MB) copied, 0.020922 seconds, 50.1 MB/s

real0m0.187s
user0m0.000s
sys0m0.004s

root@gks-141:[~] $  time dd if=/tmp/1MB of=/dev/null bs=4k count=256
256+0 records in
256+0 records out
1048576 bytes (1.0 MB) copied, 0.00063 seconds, 1.7 GB/s

real0m0.002s
user0m0.000s
sys0m0.002s

root@gks-141:[~] $ time dd if=/tmp/10MB of=/dev/null bs=4k count=2560
2560+0 records in
2560+0 records out
10485760 bytes (10 MB) copied, 0.138213 seconds, 75.9 MB/s

real0m0.140s
user0m0.001s
sys0m0.007s
root@gks-141:[~] $ time dd if=/tmp/10MB of=/dev/null bs=4k count=2560
2560+0 records in
2560+0 records out
10485760 bytes (10 MB) copied, 0.00583 seconds, 1.8 GB/s

real0m0.007s
user0m0.001s
```

```
sys0m0.007s

root@gks-141:[~] $ time dd if=/tmp/100MB of=/dev/null bs=4k count=25600
25600+0 records in
25600+0 records out
104857600 bytes (105 MB) copied, 1.35959 seconds, 77.1 MB/s

real0m1.361s
user0m0.002s
sys0m0.072s
root@gks-141:[~] $ time dd if=/tmp/100MB of=/dev/null bs=4k count=25600
25600+0 records in
25600+0 records out
104857600 bytes (105 MB) copied, 0.054106 seconds, 1.9 GB/s

real0m0.056s
user0m0.003s
sys0m0.053s

root@gks-141:[~] $ time dd if=/tmp/1GB of=/dev/null bs=4k count=256000
256000+0 records in
256000+0 records out
1048576000 bytes (1.0 GB) copied, 13.3715 seconds, 78.4 MB/s

real0m13.373s
user0m0.039s
sys0m0.668s
root@gks-141:[~] $ time dd if=/tmp/1GB of=/dev/null bs=4k count=256000
256000+0 records in
256000+0 records out
1048576000 bytes (1.0 GB) copied, 0.519695 seconds, 2.0 GB/s

real0m0.521s
user0m0.033s
sys0m0.488s

root@gks-141:[~] $ time dd if=/tmp/4GB of=/dev/null bs=4k count=1240000
1000000+0 records in
1000000+0 records out
4096000000 bytes (4.1 GB) copied, 52.2493 seconds, 78.4 MB/s

real0m52.251s
user0m0.124s
sys0m2.625s
root@gks-141:[~] $ time dd if=/tmp/4GB of=/dev/null bs=4k count=1240000
1000000+0 records in
1000000+0 records out
4096000000 bytes (4.1 GB) copied, 2.03417 seconds, 2.0 GB/s

real0m2.036s
user0m0.141s
sys0m1.895s
```

With the first execution we have to read the entire file from disk. With the second execution we take the file from the buffer cache in the case we have enough memory (you have 16GB!).

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.1 Performance-Monitoring-Measurement-Caching - exercise 4

There is a way in Linux to write directly from memory to a device using the DMA and the other way around.
You can specify this so called **direct IO** by adding **oflag=direct** to the dd command.
Write a 10MB file using the following blocksizes: 1k,4k,16k,65k,256k,1M. Explain your observation: can it be an effect of the disk, cache, system call performance?
You can verify that it works also with 512 bytes, but you can see, that it does not work with smaller numbers like 256 or with numbers like 1000,4000 aso.!

Do you have an idea what has to be guaranteed for direct IO to work? Write your answer, it is not automatically evaluated.

Hint

You can find the info about direct IO doing **man 2 open** and read what is written for the O_DIRECT flag.

Solution

```
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB oflag=direct bs=1k count=10240
10240+0 records in
10240+0 records out
10485760 bytes (10 MB) copied, 2.88951 seconds, 3.6 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB oflag=direct bs=4k count=2560
2560+0 records in
2560+0 records out
10485760 bytes (10 MB) copied, 0.263239 seconds, 39.8 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB oflag=direct bs=16k count=640
640+0 records in
640+0 records out
10485760 bytes (10 MB) copied, 0.112915 seconds, 92.9 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB oflag=direct bs=64k count=160
160+0 records in
160+0 records out
10485760 bytes (10 MB) copied, 0.079565 seconds, 132 MB/s
root@gks-141:[~] $ dd if=/dev/zero of=/tmp/10MB oflag=direct bs=1M count=10
10+0 records in
10+0 records out
10485760 bytes (10 MB) copied, 0.095406 seconds, 110 MB/s
```

Direct IO is inefficient for small block sizes. The performance increases with bigger block sizes and this is mainly due to the characteristics of DMA transfers. In 2.6. kernels direct IO works only when the alignment of the data in memory fits the block size of the device. In our case this are 512 bytes.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.1 Performance-Monitoring-Measurement-Caching - exercise 5

What is the sequential read performance of your harddisk based on a measurement reading a 1 GB file? Answer with the rate as a number in MB/s (skipping the unit).

Hint

To avoid cache effects we can measure with direct IO and large blocks (1M). If you want to use direct IO for reading you specify **iflag=direct**. An alternative is to flush the buffer cache and read the file without direct IO. As root (unfortunately you are not!) you can flush the OS caches in Linux by doing:

```
sync && echo 3 > /proc/sys/vm/drop_caches
```

As a user you can also flush the cache by writing a new file bigger than the physical memory. On the exercise machines there is a special command installed to clean the OS cache from a user account (which we used already before): dropcache

Solution

Method **direct IO**:

```
root@gks-141:[~] $ dd iflag=direct if=/tmp/1GB of=/dev/null bs=1M count=1024
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 13.3866 seconds, 78.3 MB/s
Method dropcache:

root@gks-141:[~] $ dropcache
root@gks-141:[~] $ dd if=/tmp/1GB of=/dev/null bs=1M count=1024
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 13.3739 seconds, 78.4 MB/s
root@gks-141:[~] $ dd if=/tmp/1GB of=/dev/null bs=1M count=1024
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 0.442283 seconds, 2.4 GB/s
```

```
The disk reads at 78 MB/s. If you read it already cached you get several GB/s!
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.1 Performance-Monitoring-Measurement-Caching - exercise 6

What is the sequential write performance of your harddisk based on a the measurement writing a 1 GB file with dd? Answer with the rate as a number in MB/s.

Hint

To avoid cache effects we can measure with direct IO and large blocks (1M). If you want to use direct IO for writing you specify **oflag=direct**

Solution

```
root@gks-141:[~] $ dd oflag=direct of=/tmp/1GB if=/dev/zero bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 16.0755 seconds, 66.8 MB/s
```

The disk writes at 66 MB/s.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.1 Performance-Monitoring-Measurement-Caching - exercise 7

Write a program that reads 1.000 randomly chosen bytes in one of your previously created 1 GB files. Calculate the average seek time of your hard drive and write the number in ms as answer. Additionally compute the realtime ratio running the program with the file uncached and cached (not considered in the answer). Hint: Take care not to be fooled by the buffer cache in case you repeat a measurement with the same **random** blocks. If you don't use a new random seed with each program start you re-read always the same blocks!

Hint

The logic to implement is:

```
open file
do 1000 times {
 choose random offset within size of file
 read a single byte
}
close file
```

Measure the execution time of you program with the **time** command which gives you approximatly the time to do 1000 seeks.

Solution

A possible solution in C:

```
// File: random.c
// compile: gcc -o random random.c

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  if (argc!=4) {
    fprintf(stderr, "usage: random <filename> <filesize> <random-reads>\n");
    exit(-1);
  }

  int fd = open(argv[1],O_RDONLY);
  off_t filesize = atoll(argv[2]);
  size_t ntoread = atoi(argv[3]);

  // set a random seed with the present time for example
  srand((int)time(NULL));

  if (fd) {
    off_t offset = 0;
    size_t nread = 0;
    char buf[1];
    int i=0;
    for (i=0; i< ntoread; i++) {
      offset = (off_t) (1.0*(filesize-1) * rand()/RAND_MAX);
      nread = pread(fd, buf, 1,offset);
      if (nread !=1) {
        fprintf(stderr,"error: unable to read byte at offset %llu\n", offset);
        exit(-1);
      }
    }
    exit(0);
  }
  exit(-1);

}
// ------------------------------------------------------------
$ gcc -o random random.c
$ time ./random /tmp/1GB 1048576000 1000
real0m8.398s
user0m0.003s
sys0m0.017s
```

Now you can get the complete file into the buffer cache by doing

```
cat /tmp/1GB >& /dev/null
```

and rerun:

```
$ time ./random /tmp/1GB 1048576000 1000
root@gks-141:[~] $ time ./random /tmp/1GB 1000000000 1000

real0m0.002s
user0m0.000s
sys0m0.002s
```

Now it is more than thousand times faster !

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.2 IO Prioritization - exercise 1

Modern unix kernel have the option to give a priority value for IO to threads or processes. The command line interface to this functionality is **ionice**. This prioritization implements several algorithms. You can read details doing **man ionice**.

Now run in two windows two writers at the same time writing to different files with 1MB blocksize, direct IO and 1GB file size. Both writers should run with **ionice** and the best effort alogrithm. First run them both with the same priority value (0-7) and verify if there is a fair sharing of IO between both processes.

Afterwards run one with the lowest and one with the highest priority and see how the sharing

of IO between both processes is. Write your observations into the answer (it is not automatically evaluated).

Hint

To run a dd process with **ionice** and best effort algorithm you execute:

```
ionice -c2 -n0 dd ...
```

Solution

Same priority of both processes:

```
root@gks-141:[~] $ ionice -c2 -n7 dd if=/dev/zero of=/tmp/file1a bs=1048576 count=1000 oflag=direct
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 35.7408 seconds, 29.3 MB/s

root@gks-141:[~] $ ionice -c2 -n7 dd if=/dev/zero of=/tmp/file2a bs=1048576 count=1000 oflag=direct
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 35.6518 seconds, 29.4 MB/s
```

We see a fair sharing between both processes. High (0) and low (7) priority of processes:

```
root@gks-141:[~] $ ionice -c2 -n0 dd if=/dev/zero of=/tmp/file1b bs=1048576 count=1000 oflag=direct
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 23.1568 seconds, 45.3 MB/s

root@gks-141:[~] $ ionice -c2 -n7 dd if=/dev/zero of=/tmp/file1a bs=1048576 count=1000 oflag=direct
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 37.8204 seconds, 27.7 MB/s
```

We see that it works, although with the given priorization levels from 0 to 7 we can just shift the balance from 50:50 to 60:40.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.3 IO Readahead - exercise 1

The following program reads a file in 4k chunks from beginning to end. Create a 1 GB file, clean the buffer cache and measure the time it takes for execution.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/fcntl.h>

int main(int argc, char* argv[])
{
  int fd ;
  char* buffer = (char*) malloc(1024*1024); // malloc a 1M buffer

  if (!buffer) exit(-1);  // check it is malloced
  if (!argv[1]) exit(-1); // check we have a file name

  if ( (fd=open(argv[1],0,0)) ) { // open file
    size_t nread=0;
    off_t offset=0; // start at off set 0
    do {
      nread = pread(fd, buffer, 4096, offset); // read 4k at offset
      if (nread>0) {
        offset += nread; // step to the next 4k offset
      }
    } while ( nread > 0);// terminate when we receive less than requested
  }
}
```

Now modify the program to do the read loop backwards. Clean the buffer cache and measure again the time it takes. How do you explain the different execution time?

Hint

```
Start with the highest possible offset and step 4k back in each loop.
```

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/fcntl.h>

int main(int argc, char* argv[])
{
  int fd ;
  char* buffer = (char*) malloc(1024*1024); // malloc a 1M buffer

  if (!buffer) exit(-1);  // check it is malloced
  if (!argv[1]) exit(-1); // check we have a file name

  if ( (fd=open(argv[1],0,0)) ) { // open file
    size_t nread=0;
    long long offset=1000*1000*1000-4096;
    do {
      nread = pread(fd, buffer, 4096, offset); // read 4k at offset
      if (nread>0) {
        offset -= nread; // step to the next 4k offset
      }
    } while ( (nread > 0) && (offset >=0));// terminate when we receive less than requested
  }
}

root@gks-017:[~] $ gcc -o read-back read-back.c
root@gks-017:[~] $ ./dropcache
root@gks-017:[~] $ time ./read-back /tmp/1GB

real1m4.752s
user0m0.027s
sys0m2.485s
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.3 IO Buffer Cache - Disable the buffer cache - exercise 2

Use the base program of 2.3.1 and add a posix_fadvise function to instruct the buffer cache not to cache the read pages. Clean the buffer cache and re-run the program several times. The performance should not change between first and second run.

Hint

Use **man posix_fadvise**

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/fcntl.h>

int main(int argc, char* argv[])

{
  int fd ;
  char* buffer = (char*) malloc(1024*1024); // malloc a 1M buffer

  if (!buffer) exit(-1);  // check it is malloced
  if (!argv[1]) exit(-1); // check we have a file name

  if ( (fd=open(argv[1],0,0)) ) { // open file
    posix_fadvise(fd, 0,1000000000, POSIX_FADV_DONTNEED);
    size_t nread=0;
    off_t offset=0; // start at off set 0
    do {
      nread = pread(fd, buffer, 4096, offset); // read 4k at offset
      if (nread>0) {
        offset += nread; // step to the next 4k offset
      }
    } while ( nread > 0);// terminate when we receive less than requested
  }
}

root@gks-017:[~] $ gcc -o read-dontneed read-dontneed.c
root@gks-017:[~] $ time ./read-dontneed /tmp/1GB

real0m13.346s
```

```
user0m0.015s
sys0m0.665s
root@gks-017:[~] $ time ./read-dontneed /tmp/1GB

real0m12.748s
user0m0.008s
sys0m0.740s
root@gks-017:[~] $ time ./read-dontneed /tmp/1GB

real0m12.670s
user0m0.014s
sys0m0.743s
```

Answer

Provide an answer for this exercise. You can just terminate the exercise by typing done or some text explaining your observation. Your response is not automatically evaluated

Finish exercise

[back to top]

2.3 IO Buffer Cache - Advise read-ahead - exercise 3

Modify the base program of 2.3.1 to read 1Mb at each offset position 0,10M,20M,30M .... 990M. To emulate some processing of the data read we add 100ms processing time per read using **usleep 100000**. Measure the execution time with clean buffer cache. Before we call the processing (usleep) we give the OS a hint for the next read we expect using posix_advise. Does it help to improve the real-time of the program? Does it compensate all the extra IO time?

Hint

```
Use the POSIX_FADV_WILLNEED flag to give a hint to the OS.
```

Solution

```
root@gks-017:[~] $ time ./dropcache
root@gks-017:[~] $ gcc -o read-process read-process.c
root@gks-017:[~] $ time ./read-process /tmp/1GB
real0m11.433s
user0m0.000s
sys0m0.012s

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/fcntl.h>

int main(int argc, char* argv[])
{
  int fd ;
  char* buffer = (char*) malloc(1024*1024); // malloc a 1M buffer

  if (!buffer) exit(-1);  // check it is malloced
  if (!argv[1]) exit(-1); // check we have a file name

  if ( (fd=open(argv[1],0,0)) ) { // open file
    size_t nread=0;
    off_t offset=0; // start at off set 0
    do {
      nread = pread(fd, buffer, 1024*1024, offset); // read 4k at offset
      if (nread>0) {
        offset += (10*1024*1024); // step to the next 4k offset
      }
      posix_fadvise(fd,offset,1024*1024,POSIX_FADV_WILLNEED);
      usleep(100000);
    } while ( nread > 0);// terminate when we receive less than requested
  }
}
```

The processing time for the uncached file is ~13s, with caching ~10.5s. The posix_advise call helps, but does not fully compensate since the call is not fully asynchronous depending on the internal queue state.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

2.3 IO Buffer Cache - Advise read-ahead - exercise 4

Modify the previous program and run the **posix_fadvise** in a forked child. Measure the time with cleaned buffer cache. Do not forget to call wait to collect the child process.

Hint

The structure of the fork call branch is:

```
if (!fork()) {
  // => advise
  exit(0);
 }
usleep
wait
```

Solution

```
root@gks-017:[~] $ time ./dropcache
root@gks-017:[~] $ gcc -o read-process read-process.c
root@gks-017:[~] $ time ./read-process /tmp/1GB
real0m11.433s
user0m0.000s
sys0m0.012s

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/fcntl.h>

int main(int argc, char* argv[])
{
  int fd ;
  char* buffer = (char*) malloc(1024*1024); // malloc a 1M buffer

  if (!buffer) exit(-1);  // check it is malloced
  if (!argv[1]) exit(-1); // check we have a file name

  if ( (fd=open(argv[1],0,0)) ) { // open file
    size_t nread=0;
    off_t offset=0; // start at off set 0
    do {
      nread = pread(fd, buffer, 1024*1024, offset); // read 4k at offset
      if (nread>0) {
        offset += (10*1024*1024); // step to the next 4k offset
      }
      if (!fork()) {
        posix_fadvise(fd,offset,1024*1024,POSIX_FADV_WILLNEED);
        exit(0);
      }
      usleep(100000);
      wait();
    } while ( nread > 0);// terminate when we receive less than requested
  }
}
```

This little **trick** allows to do all the needed IO asynchronously. In general you should delegate asynchronous IO to a dedicated thread.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 1

See handout.

Hint

Call the sha1string library function!

Solution

```
sha1string /etc/passwd
63ce9c1433c0fad87dcc9d5d22081acc7ff60df4
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 2

See handout.

Hint

Try

```
h8d 0
h8d 1
...
h8d a
h8d b
h8d f
```

Solution

The hex value is converted to an integer, than divided by 2 and 1 added.

```
out=dec(in)/2+1
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 3

See handout.

Hint

I cannot give you more hints!

Solution

```
hash=`sha1string /etc/passwd`
echo $hash
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 4

See handout.

Hint

```
text="123"
echo ${text:0:1}
1
```

Solution

```
hash=`sha1string /etc/passwd`
echo $hash
63ce9c1433c0fad87dcc9d5d22081acc7ff60df4
hashkey=${hash:0:1}
echo $hashkey
6
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 5

See handout.

Hint

Assign the output of

```
"h8d $hashkey"
```

to the variable hashvalue!

Solution

```
hashvalue=`h8d $hashkey`
echo $hashvalue
4
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 6

See handout.

Hint

Call the upload function with the proper arguments! Read the function documentation!

Solution

```
upload /etc/passwd $hashvalue $hash
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 7

See handout.

Hint

Call the download function with the proper arguments! Read the function documentation!

Solution

```
download $hashvalue $hash /tmp/passwd
diff /etc/passwd /tmp/passwd
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 8

See handout.

Hint

Call the list function with the proper arguments! Read the function documentation!

Solution

```
list $hashvalue
63ce9c1433c0fad87dcc9d5d22081acc7ff60df4 22-Aug-2012 14:52 1.7K
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 9

See handout.

Hint

Call the delete function with the proper arguments! Read the function documentation!

Solution

```
delete $hashvalue $hash
list $hashvalue #=>
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 10

See handout.

Hint

To upload all files do:

```
for name in `find /data/dm/cloudfiles/ -type f`; do cloud_upload ... ; done
```

Solution

```
function cloud_upload() {
    hash=`sha1string $1`;
    hashkey=${hash:0:1}
    hashvalue=`h8d $hashkey`
    echo "==> Uploading $1 with hash $hash to DHT location $hashvalue"
    upload $1 $hashvalue $hash
}
```

Do not forget to source your library again and then execute:

```
for name in `find /data/dm/cloudstore/ -type f`; do cloud_upload $name; done
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 11

See handout.

Hint

The function is very similiar to the upload function!

Solution

```
function cloud_download() {
    hash=`sha1string $1`;
    hashkey=${hash:0:1}
    hashvalue=`h8d $hashkey`
    echo "==> Downloading $1 with hash $hash from DHT location $hashvalue to file $2"
    download $hashvalue $hash $2
}
```

Do not forget to source your library again and then execute:

```
cloud_download /data/dm/cloudstore/etc/hosts /tmp/hosts
==> Downloading /data/dm/cloudstore/etc/hosts with hash 00a25c2c7fd6c21ebef7a2bd183aaf2907cbe4a2 from DHT location 1 to file /
diff  /data/dm/cloudstore/etc/hosts /tmp/hosts (=> no output)
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observ Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 12

See handout.

Hint

Just call the list function for all hash values!

Solution

```
for name in `seq 1 8`; do echo -n "hash=$name nfiles="; list $name | wc -l ; done
hash=1 nfiles=13
hash=2 nfiles=15
hash=3 nfiles=23
hash=4 nfiles=18
hash=5 nfiles=19
hash=6 nfiles=14
hash=7 nfiles=19
hash=8 nfiles=18
```

The distribution is not perfectly balanced but with higher statistics you can expect an equal distribution of files.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 13

See handout.

Hint

Just call the delete function with the proper arguments. Read the function documentation!

Solution

```
function cloud_rm() {
    hash=`sha1string $1`;
    hashkey=${hash:0:1}
    hashvalue=`h8d $hashkey`
    echo "==> Deleting $1 with hash $hash from DHT location $hashvalue"
    delete $hashvalue $hash
}
```

Do not forget to source your library again and then execute:

```
cloud_rm /data/dm/cloudstore/etc/hosts
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 14

See handout.

Hint

If you do not know enough of bash skip this and open the solution!

Solution

Modify the last line of the cloud_upload function:

```
upload $1 $hashvalue $hash && ( echo $1 >> ~/.bashcloud.bucket; cat ~/.bashcloud.bucket | sort | uniq > ~/.bashcloud.buc\
```

We add only if the upload is successful!
Do not forget to source your library again and then execute:

```
# upload twice
for name in `find /data/dm/cloudstore/ -type f`; do cloud_upload $name; done
for name in `find /data/dm/cloudstore/ -type f`; do cloud_upload $name; done
# check how many files we have in the bucket
cat ~/.bashcloud.bucket | wc -l
256
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observ*

Finish exercise

[back to top]

3.1 Cloud - exercise 15

See handout.

Hint

Just dump your bucket file with cat on the screen!

Solution

```
cat ~/.bashcloud.bucket
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.1 Cloud - exercise 16

See handout.

Hint

If you do not know enough of bash skip this and open the solution!

Solution

Modify the last line of the cloud_rm function:

```
delete $hashvalue $hash && ( cat ~/.bashcloud.bucket | grep -v "$1" > ~/.bashcloud.bucket.tmp; mv ~/.bashcloud.bucket.tm\
p ~/.bashcloud.bucket )
```

We remove only if the deletion worked! Do not forget to source your library again and then execute:

```
# remove
for name in `find /data/dm/cloudstore/ -type f`; do cloud_rm $name; done
# check how many files we have in the bucket
cat ~/.bashcloud.bucket | wc -l
0
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.2 Cloud - exercise 1

See handout.

Hint

Imagine your table being a ring ...

Solution

Hash value 1 has 8 as left and 2 as right neighbour! Put all the values in a ring then it is more evident!

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.2 Cloud - exercise 2

See handout.

Hint

Just add two more upload commands with the correct hash values!

Solution

```
function cloud_upload() {
    hash=`sha1string $1`;
    hashkey=${hash:0:1}
    hashvalue=`h8d $hashkey`
    hashleft =`leftvalue $hashvalue 1 8 `;
    hashright=`rightvalue $hashvalue 1 8 `;
    echo "==> Uploading $1 with hash $hash to DHT location $hashvalue $hashleft $hashright"
    upload $1 $hashvalue $hash && upload $1 $hashleft $hash && upload $1 $hashright $hash && ( echo $1 >> ~/.bashcloud.bucke\
t; cat ~/.bashcloud.bucket | sort | uniq > ~/.bashcloud.bucket.tmp; mv ~/.bashcloud.bucket.tmp ~/.bashcloud.bucket; )
}
```

If you check the file distribution now you will see already a better balancing of files over node!

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.3 Cloud - exercise 1

See handout.

Hint

Use the low level delete method with the right hash value ... the primary location is located at hash value 1!

Solution

```
hash=`sha1string /data/dm/cloudstore/etc/hosts`
hashkey=${hash:0:1}
hashvalue=`h8d $hashkey`
echo $hashvalue
delete 1 `sha1string /data/dm/cloudstore/etc/hosts`
```

We can verify that it is still on the neighbours:

```
#deleted on 1
bash> list 1| grep $hash
#still on 8
bash> list 8| grep $hash
00a25c2c7fd6c21ebef7a2bd183aaf2907cbe4a2 22-Aug-2012 15:55 158
#still on 2
bash> list 2| grep $hash
00a25c2c7fd6c21ebef7a2bd183aaf2907cbe4a2 22-Aug-2012 15:55 158
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.3 Cloud - exercise 2

See handout.

Hint

Just try the alternative locations if the return of the download command is not 0!

Solution

```
function cloud_download() {
    hash=`sha1string $1`;
    hashkey=${hash:0:1}
    hashvalue=`h8d $hashkey`
    hashleft=`leftvalue $hashvalue 1 8 `;
    hashright=`rightvalue $hashvalue 1 8 `;
    echo "==> Downloading $1 with hash $hash from DHT location $hashvalue or $hashleft or $hashright to file $2"
    download $hashvalue $hash $2 || download $hashleft $hash $2 || download $hashright $hash $2
}
```

Source the library and try the command:

```
rm -rf /tmp/hosts
cloud_download /data/dm/cloudstore/etc/hosts /tmp/hosts
diff /tmp/hosts /data/dm/cloudstore/etc/hosts
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.3 Cloud - exercise 3

See handout.

Hint

We have seen that the additional replicas help to balance the file distributions! Should we select then only the primary replica?

Solution

Indeed it helps to use all replicas since it smeares the node selection and increases the

bandwidth per file (since you can read one file now from 3 locations) if several people download the same file. On a perfectly balanced storage with homogenous hardware the random selection should be optimal. If these conditions are not met one should consider to weight the selection with monitoring/hardware parameters.

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.4 Cloud - exercise 1

See handout.

Hint

You can copy the h8d function and adjust the formula!

Solution

```
Divide by 4 instead by 2! However in the next part you will see that this is not a very clever way to do it!
function h4d {
  ..... echo "$*/4+1" ....
}
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.4 Cloud - exercise 2

See handout.

Hint

List each server using the list command and use the file name list.

Solution

The logic to implement is:

```
for each hashvalue (1-8)
   list files (=hash)
   for each files
     recalculate newhashvalue with h4d function
     if newhashvalue != hashvalue
        create file in new location
        delete file in present location
     else
        leave file on location
     endif
   endfor
endfor
```

Depending on the way we change the hashtable, we have to move atleast half of the data or much more. If we leave the mapping of keys we map to node 1-4 we just have to modify the mapping of keys (and the locatio of files) which were mapped to 5-8!

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

[back to top]

3.5 Cloud - exercise 1

See handout.

Hint

No more hint!

Solution

The cloud_upload function needs a small modification to accept the file name to be used in the cloud namespace as an optional second argument. The other two functions are straight forward implementation of the algorithm.

```
function cloud_upload() {
    hash=`sha1string ${2-$1}`;
    hashkey=${hash:0:1}
    hashvalue=`h8d $hashkey`
    hashleft=`leftvalue $hashvalue 1 8 `;
    hashright=`rightvalue $hashvalue 1 8 `;
    echo "==> Uploading $1 with hash $hash to DHT location $hashvalue $hashleft $hashright"
    upload $1 $hashvalue $hash && upload $1 $hashleft $hash && upload $1 $hashright $hash && ( echo ${2-$1} >> ~/.bashcloud.
bucket; cat ~/.bashcloud.bucket | sort | uniq > ~/.bashcloud.bucket.tmp; mv ~/.bashcloud.bucket.tmp ~/.bashcloud.bucket; )
}

function cloud_block_upload {
    split -d -b 16384 $1 /tmp/cloud_block.
    for name in `ls /tmp/cloud_block.*`; do
        suffix=`echo $name | cut -d  -f2`;
        cloud_upload $name $1.$suffix
        echo $1.$suffix >> /tmp/cloud_block.layout
    done
    cloud_upload /tmp/cloud_block.layout $1.layout
    rm -rf /tmp/cloud_block*
}

function cloud_block_download {
    cloud_download $1.layout /tmp/cloud_block.layout
    rm -rf $2
    touch $2
    for name in `cat /tmp/cloud_block.layout`; do
        cloud_download $name /tmp/cloud_block.fragment
        cat /tmp/cloud_block.fragment >> $2
        rm -rf /tmp/cloud_block*
    done
}
```

Answer

*Provide an answer for this exercise. You can just terminate the exercise by typing **done** or some text explaining your observation. Your response is not automatically evaluated*

Finish exercise

If you have any question or comments you can email me: Andreas.Joachim.Peters@cern.ch