Storage - take your time

Johan Montelius and Amir H. Payberah

1 Introduction

In this experiment we will look at secondary storage i.e. drives that we have connected to our computer to provide persistent storage. We will do some performance measurements to try to estimate how long time read and write operations actually take. You can do these experiments using your own computer and preferably should since you then have a more controlled environment.

There is a difference in performance between a hard-disk drive and a solid-state drive. If possible run these experiments on different machines to see the difference. Do the experiments in a group with different machines to see the difference. We will use the term *drive* to mean the physical device, hard-disk or solid-state, not to confuse with *driver* that is the operating system module that is responsible for talking to the drive.

2 File operations

In the benchmarks that we will implement, we will use the file operations that operate on *file descriptors*. These are low level operations that the operating system provide and will give us more control over what we are doing. The alternative is using the `stdio` library but then our benchmark would be running on top of a layer that buffers our operations to be more efficient. The efficiency is gained by doing write operations in the background, something that often is fine but it will not tell us when the data is actually on the drive.

open, read and write

We will only use three primitives: open, read and write. Look up these function in the man pages; you have state that you want to look at the system calls since there are also shell commands with the same names and these will show up by default.

```
> man 2 open
   :
   int open(const char *pathname, int flags, mode_t mode)
```

To open a file we specify the *path name* i.e. the name of the file, some *flags* that specify how the file should be opened and the *mode* of the file if we should create it. In our benchmarks we will open the file for reading and

writing and specify that it, if it is created it should have user read and write permissions.

```
int flag = O_RDWR | O_CREAT | O_DSYNC;
int mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
int fd = open(name, flag, mode);
```

There is one important flag here, the `O_DSYNC` flag. This flag specify that we do not want a write operation to return unless the thing that we have written is actually pushed to the drive. We will not accept any cheating, the operating system should have pushed the data to the drive and received an acknowledgment.

read, write and lseek

We will only read and write dummy values in our files using the procedures `read()` and `write()`. Both of these procedures work the same, they take a file descriptor, the address of a buffer and the number of bytes that they should read or write.

```
int buffer = 42;
write(fd, &buffer, sizeof(int));
```

```
int buffer;
read(fd, &buffer, sizeof(int));
```

Both procedures will operate on the file given the *current position*, i.e., an index that tells the operating system where in the file we want to read or write. This position is mostly handled automatically, if you read from the beginning of a file you just read and the position will move forward as you read. We however want to read and write at random locations so we will set this position explicitly. This is done using the procedure `lseek()` as shown in the examples below.

```
int pos = 512;
lseek(fd, pos, SEEK_SET);
 :
int step 64;
lseek (fd, step, SEEK_CUR);
 :
```

The third argument to `lseek()` decides if the value, given as the second argument, should be interpreted as an absolute value (512 in the example above) or as an offset to the current position (64 in the example).

3 Before we go

We will start by generating a large file so that we have something to work with. You could of course take any file that you happen to have but it will have to be large enough for our experiments. We will generate a file that is 512 MiBytes so if you have a movie on your drive you could use it but the code that you write will be reused in writing the benchmark so you might as well generate a new file.

Create a file called `generate.c` and include some good to have include files. Also define the macros `BLOCKS` and `SIZE` that describes that we will generate a million blocks of size 512 bytes each.

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

#define BLOCKS 1024*1024
#define SIZE 512
```

Start by finding the file name from the command line. Open the file and make sure that we actually managed to open it. When we generate the file we do not have to give the `O_DSYNC` flag since we're not interested in how long time each write operation took.

```
int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("usage: generate <file name>/n");
    return -1;
  }

  const char *name = argv[1];
  int mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
  int flag = O_RDWR | O_CREAT;
  int fd = open(name, flag, mode);
  assert(fd != -1);
```

Then we loop over the blocks and make sure that we write in each block. We are a bit paranoid here as it would be sufficient to just write at the last position, but we don't want the operating system nor the drive to do clever things behind our backs.

```
  int buffer;

  for(int b=0; b < BLOCKS; b++) {
    lseek (fd, SIZE*b, SEEK_SET);
    buffer = b;
    write(fd, &buffer, sizeof(int));
  }
  close(fd);
  printf("done\n");
  return 0;
}
```

Save the file, compile and generate a file `foo.txt`. Check if the file is actually 512 MiByte in size using the `ls -l` command. If everything looks OK, we're ready to go.

4 Read performance

Our first benchmark will be to examine the read performance. Make a copy of your file and call it `read.c` and make the following updates. We now only list the changes so if you've used an existing file you will have to go through the previous section. We will use `clock_gettime()` to do time measurement so we need the header file `time.h`. We will also do a number of read operations separated by `STEP` bytes and we will read from `COUNT` number of blocks.

```
#include <time.h>

#define STEP 64
#define COUNT 1000
```

The idea is that we will choose a block by random and then read the first bytes of this block, then the bytes at position 64, 128 etc. The question is if there is any difference in reading the first entry in a block and reading the second or third entry. We will gather statistics and present the different quartiles in execution time. The easiest way to do this is to collect the execution times and sort them and to do this we need a function that compare to entries.

```
int compare(const void *a, const void *b) {
  return (int)(*(long *)a - *(long*)b);
}
```

We will record the execution time in nano-seconds so we also provide a function that takes two `timespec` entries and returns the difference as `long`.

```
long n_sec(struct timespec *start, struct timespec *stop) {
  long diff_sec = stop->tv_sec - start->tv_sec;
  long diff_nsec = stop->tv_nsec - start->tv_nsec;
  long wall_nsec = (diff_sec *1000000000) + diff_nsec;
  return wall_nsec;
}
```

We have what we need to implement the benchmark. We start as before by opening the file that is given to us on the command line. We then allocate some space to hold a table where we will record the execution time. The table holds an array of time-stamps for each of the indexes that we will access i.e. 0, 64, 128, ... Each of these arrays are `COUNT` long.

```
  int entries = SIZE / STEP;
  long **table = malloc(sizeof(long*) *entries);

  for (int i = 0; i < entries; i++) {
    table[i] = malloc(sizeof(long) *COUNT);
  }
```

Now for the actual benchmark, we will access `COUNT` blocks and generate a radium number from 0 to `BLOCKS` minus one. We set the *current position* to the start of this block and then perform a number of read operations. We clock each read operation, store the result and forward the position by `SIZE` steps.

```
  for (int c=0; c < COUNT; c++) {
    int b = rand() % BLOCKS;
    lseek(fd, b*SIZE, SEEK_SET);

    for(int e = 0; e < entries; e++) {
      struct timespec t_start, t_stop;
      int buffer;
      clock_gettime(CLOCK_MONOTONIC, &t_start);
      read(fd, &buffer, sizeof(int));
      clock_gettime(CLOCK_MONOTONIC, &t_stop);
      lseek (fd, STEP - sizeof(int), SEEK_CUR);
      table[e][c] = n_sec(&t_start, &t_stop);
    }
  }
```

The only thing that is left is to collect the statistics and we do this one row at a time. Note that we also print the 90 percentile since the maximum value can differ very much.

```
printf("#N\tMin\tQ1\tMed\tQ3\tD9\tMax\n");
for (int e = 0; e < entries; e++) {
  qsort(table[e], COUNT, sizeof(long), compare);
  long min = table[e][0];
  long q1  = table[e][COUNT/4];
  long med = table[e][COUNT/2];
  long q3  = table[e][3*(COUNT/4)];
  long d9  = table[e][9*(COUNT/10)];
  long max = table[e][COUNT-1];
  printf("%d\t%ld\t%ld\t%ld\t%ld\t%ld\t%ld\n", e*STEP, min, ... etc);
}
```

That's it, run you first benchmark as find out how fast you hard drive is. Hmm, not very bad is it? Just for fun, change the `COUNT` value to one, run the benchmark and write down the execution time. Now turn your computer off, turn it back on again and run the same test again - hmm, run it again. That was probably only some random operating system thing - repeat, is it reproducible?

The page cache

When a file is read from disk, the operating system will place the read blocks in what is called *the page cache*. The idea is of course that if we read form the file once it is very likely that we will read from it again so we keep a copy of the blocks that we read in a cache. If you use the following command you will see how the memory is used by the operating system. There is one entry called `buff/cache` and this is the area used for cached files.

```
> free
              total        used        free      shared  buff/cache   available
Mem:        8089288     1321116     5221168      943912     1547004     5476420
Swap:       8301564        6644     8294920
```

We can direct the operating system to drop files from the cache by running the command below. This will run a shell in supervisor mode and let this shell write a `1` to the file `drop_caches`. This is an example of how we can interact with the kernel by using the regular file operations. All of the files under `/proc` directory are special files that allow us to interact with the kernel.

```
> sudo sh -c 'echo 1 > /proc/sys/vm/drop_caches'
```

Check if the size of the cache has been decreased. Run the benchmarks again and see if there is a difference when we clear the cache.

Evaluation

What do the numbers mean? Is there a difference in access time, why? Are there more caches involved, does the hard-drive have a cache of its own? Many questions and you will probably not be able top answer them all but you should be able to do a nice graph to go with your findings. You can quickly generate a nice graph using gnuplot, everything is prepared so we only need to save the data in a file and run some gnuplot commands.

```
> ./read foo.txt > foo.dat
```

Now start gnuplot and give the following commands:

```
> gnuplot
  :
gnuplot> set xrange[-30:500]
   :
gnuplot> set xtics 64
   :
gnuplot> set boxwidth 10
   :
gnuplot> plot 'foo.dat' u 1:3:2:6:5 with candlesticks
   :
```

Create a file `read.p` where you write a gnuplot script that includes the commands above. In the beginning of the file you also specify that you would like to generate a .png file and that we don't really need the legend. Set the `xlabel`, `ylabel` and adjust the `yrange` to start from 0 (`[0:*]`). Now you have a nice graph to show you friends when you explain the difference in read execution time.

```
set terminal png
set output "read.png"

unset key
  :
```

Run the script from the command line by giving it as an argument to `gnuplot`. Now you have a nice graph to show you friends when you explain the difference in read execution time.

5 Write performance

So now that we know something about the read performance it's time to test the write performance. Make copy of your benchmark and call it `write.c`; there are only some small changes that we need to fix. Make sure that you

have the `O_DSYNC` flag set, this will prevent the operating system driver from pulling your leg.

```
int flag = O_RDWR | O_CREAT | O_DSYNC;
```

Then we simply replace the read operation with a write operation.

```
int buffer = e;
write(fd, &buffer, sizeof(int));
```

That's it, test it and see if there are any differences compared to the read benchmark.

Living on the edge

You might have discovered that writing takes a whole lot longer time than reading, but we can improve things dramatically by cheating. Remember that we set the `O_DSYNC` flag to make sure that the thing that we wrote was actually pushed out to the disk and not just written to a buffer waiting for better times. Do the same experiment but now without the `O_DSYNC` flag - any improvements?

The operating system will now allow *write back*, i.e., it will acknowledge that the write operation has been performed but the data is still in a buffer allocated by the operating system. The operating system will push the data to drive as soon as it finds a suitable moment, it might have done so already but it is not hanging around waiting for a confirmation. If you trust your power supply this is probably very good strategy but if your computer crashes, the data might not be on the drive.

Note - using `O_DSYNC` only means that the operating system has pushed the data to the drive. The drive might still have the data in it's own buffer but it is the responsibility of the drive to push it to the physical disk or flash memory.

6 Memory speed

While we're at it, we might as well do two more benchmarks. The first thing we should check is the read and write performance of the regular memory. To do this there are only a few changes we have to do to our benchmark program. Make a copy of `read.c` and call it `mread.c` and do the following changes. We will need to do more than one read operation to get an accurate answer so lets define a macro called `LOOP`.

```
#define LOOP 64  // can not be 0
```

Then we allocate a large array on the heap that of course has the size determined by the number of `BLOCKS` and their `SIZE`. We of course remove everything that is related to the opening of a file.

```
  int *heap = malloc(sizeof(int)*BLOCKS*SIZE);
```

Now we replace the read operation in the benchmark with a loop that reads from our array.

```
    for(int l = 0; l < LOOP; l++) {
      buffer = buffer + heap[b*SIZE+e+l];
    }
```

The final trick is to divide the calculated execution time by `LOOP` to obtain the execution time of one read operation. This is why `LOOP` can not be set to 0.

```
  table[e][c] = (n_sec(&t_start, &t_stop) / LOOP);
```

What is the read performance of main memory? Can you trust the number? Do a small change to the program, set `LOOP` to 0 and remove the division, how long time does it take to do nothing (or rather, measure the time)? Should we compensate for this? What if we add the following to our program:

```
#define RES ??  // the time it takes to do nothing
  :
  table[e][c] = ((n_sec(&t_start, &t_stop) - RES) / LOOP);
```

This gives us something more accurate but now we have hard coded the benchmark for this particular machine. You could let the benchmark first determine what the time is to do nothing when it starts but let's not over do things. Do some experiments and change the number of `COUNTS` between 10 and 10000, what is going on Change also the write benchmark in the same way and include for example the operation below in the loop. Run with a small number of counts - what is happening?

```
  heap[b*SIZE+e+l] = e+l;
```

Let's try to access the whole array before we run the benchmark loop. Add the following to your programs, both to `mread.c` and `mwrite.c`. Add it immediately after allocating the array - any difference?

```
for (int b = 0; b < BLOCKS; b++) {
  heap[b*SIZE] = b;
}
```

On my computer a write operation seams to be faster than a read operation, this is probably only a problem with the clock - right? (when I say something like this it is to make you think, there is nothing wrong with my clock)

7 A mix of both

This experiment is only possible if you have sudo access to the machine that you're running on. We will *mount* a new file system using something called `tmpfs`. This will create a drive that looks like a regular drive from the outside but is actually implemented as a segment of the physical DRAM memory. Things will be faster than any regular disk drive but once you turn off the power everything will be lost.

Start by creating a directory called for example `dram`. Then you run the following command in a shell. Your `user id` is probably 1000 but check this using the command `id`.

```
> sudo mount -t tmpfs -o mode=744,uid=1000,gid=1000 tmpfs ./dram
```

So far so good, what you have now done is created a virtual drive and mounted it using the directory `dram`. Anything we place in this directory will be allocated in physical memory (if you have enough of it). Start by copying the file `foo.txt` to this directory. Then run the benchmarks `read` and `write` using the copy as the target.

```
> ./read dram/foo.txt
  :
```

Any difference in read performance? How about write performance?

8 Summary

You should now have a better understanding of the performance of secondary storage i.e. your hard-disk or solid-state drive. We have looked at the time it takes to do a read or write operation to a random block in a large file

and made some observations. As an exercise write down the approximate execution time for the following operations:

- Reading or writing to a random location in memory.
- Reading or writing to a random location of a file that has been cached by the drive and/or the operating system.
- Reading or writing to several positions in the same file block.
- Reading or writing to a random location of a file from a cold start, i.e., not cached by the operating system nor drive.

The performance of a hard-disk drive differs from the performance of a solid-state drive - what is the difference and when does it matter?