NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Solution 7

File System & Unix I/O

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman
Daniel Hansen

Handout: 10.04.2023

Introduction

In this exercise, we are focussing again on the persistent storage in our system. We will explore disk characteristics, filesystems and also quickly tap into Unix-style I/O.

1 Accessing Files on a Harddsik

Assume the numbers given for the Barracuda Disk in Figure 37.5 in chapter 37.4[1] for the following exercises.

1. One way to improve the throughput of a disk and make sure we are using the full capacity of a disk is to run a compaction procedure. The compaction procedure reorders the files on the disk so that all the files are now contiguous and there are no holes between them. Now assume that this means we are required to move 128 8 KB files on the disk. How long does it take to complete the compact step? What is the average throughput of the operation? How does it compare to the max throughput shown in Figure 37.5?

   Answer: *The time taken for a single file read consists of the average seek time (9 ms), the average rotational delay ($(60s * 0.5)/7200 = 4.1ms$, we are taking into account that on average the disk needs to do half a rotation), and the transfer time of 0.0744 ms (8KB / (105 MB/s)). This yields a total read time of 13.2411 ms. Since we need to write the file back to disk, we now need to seek again (find where to write) and transfer the data back to disk, so we double the previous result for a single file. The overall time taken per file thus is 26.4821 ms. Now we do this for 128 files, so the overall time taken is 3.389 seconds. Since we moved the data from and to the disk, we moved 2 MB (2 * 128 * 8 KB) of data, so our average throughput for the operation is 0.59 MB/s (2MB / 3.389 s). This is a lot slower (factor 200) than the max throughput of 105 MB/s due to the seek time added for every single file.*

2. Now lets assume we only have to copy a single file of size 1 MB. How long does it take, how is the average throughput and how does it compare to the previous exercise?

   Answer: *Again, we start with 9 ms seek time and 4.16 average rotational delay before we can start transferring data. The transfer time is calculated analogous to the previous task and results in 9.52 ms. The total read time is thus 22.68 ms. As before, we need to write it back to disk, so we double that number to get the overall time for a single file (45.36 ms). Since we are only transferring one file, that is all we need to do and thus the entire operation takes only 45.36 ms, compared to the previous 2.323 s, while still moving the same amount of data. The throughput is thus 44.09 MB/s, which is still only roughly half of the reported max throughput due to seek overhead.*

3. Even though SSDs do not really benefit from being compacted, lets assume we are copying the same amount of files as in exercise 1 and 2 above. You find the expected speeds for a selected range of SSD models in chapter 44 of the ostep book, page 18. Explain in which case an SSD would be much faster than the HDD and give some background on how the SSD achieves these high throughput numbers.

   Answer: *SSDs outperform regular HDDs mainly when comparing the random access speeds. The sequential speed is also higher, but the difference there is a lot less pronounced, as HDDs tend to also be able to reach rather high speeds when transferring data sequentially. The main reason for the performance improvement is that SSDs don't require physical movement and are actually true random-access devices, meaning that access to any block is the same speed. The differences in random access vs contiguous accesses for SSDs is debatable, as accesses to any location should take roughly the same amount of time. The difference often stems from how big an accessed block is, thus the speed is often limited in logic further up (e.g. there might be a maximum of outstanding accesses).*

[1] https://pages.cs.wisc.edu/~remzi/OSTEP/file-disks.pdf, page 7

2 File Systems

In this part we will solve a few exercises on file systems and learn how they organize disk space that is available to us.

1. Which abstractions does a filesystem provide us with?

   ☐ The filesystem depends on the hardware layout of the persistent storage and abstracts from the hardware-specific mechanisms to a standard file interface

   ☐ It unifies all system information in a single namespace

   ☒ Gives us a logical layer providing directories and files above a persistent storage device

   ☐ Tracks the metadata of files (e.g. last edited, created at, file type, ...)

   ☐ Provides an abstract interface to read, write, and execute files that are stored on disk

2. Given the following outline of an inode structure, what is the biggest file you could store on such a filesystem? Assume a block size of 4 KB, while having 8 direct pointers, 4 single-indirect pointers and 2 double-indirect pointers, with a disk-address size of 64 bits.

   Answer: *Let b be the block size. This gives us $(2 \times (b/8)^2 + 4 \times (b/8) + 8) \times b = 526344b$. This corresponds to roughly a file size of 2 GB.*

3. In the previous exercise: How much overhead do we have for storing the largest possible file?

   Answer: *For indirect nodes, we use additional blocks from the data blocks to keep the pointers. So for the double indirect blocks, we will allocate $((b/8) + 1)$ blocks each and for the single indirect block, we will allocate 1 block each. No additional blocks are allocated for the direct mapped nodes. This gives us a total of 1030 additional blocks allocated or roughly 4 MB of overhead to store a 2GB file in that structure.*

4. You are building a system where you mostly expect large files to be stored. Propose another way of organising files on the disk in order to reduce the storage overhead and maximize the amount of storage used for useful data.

   Answer: *Instead of storing simple pointers on the inode, we use the idea of extent-based approaches, where we point to where a file begins and additionally store the number of consecutive blocks from there on. To increase the flexibility of the solution (e.g. a file doesn't fit in any of the free lists superblocks), we add a flag indicating if this is the entire file or not. If not, the last few bytes will contain another pointer, a block number count and a flag if that is the last thing of the file in the list. This approach makes it a bit slower to seek within the file (since it is partially a linked list and we need to traverse the list of larger blocks), but that only ever affects large files that can't be fit inside one contiguous free space. This could also be resolved by regularly compacting the disk again, and reordering the blocks of the files in order to not have any indirection.*

5. Describe a technique to make your filesystem described above more resilient to a system crash while updating a huge file. What guarantees are needed by the hardware to achieve higher resiliency?

   Answer: *One possible solution is using a write-ahead log, where we log which operation we will perform next before we start the operation. That way, when we crash during the operation, we can go back to the log and check what operation we need to re-run, or at least know which block might be damaged. Once the writes have completed, we can checkpoint them by updating the corresponding metadata, and finally write the TxE block to the journal. The hardware needs to guarantee us a certain block size to be either written or not (and not to crash halfway through). That way we can organise our log in blocks no larger than that atomic block. This guarantees that we either will see the TxE or not.*

3 Unix I/O

While file systems are generally independent of the OS, Unix played a crucial role in today's file-system interfaces we are working with. Due to this special role, we will also dive into the interface and abstractions of Unix.

1. Unix has the open syscall. Assume we would remove that syscall. Could one still operate on a file? How could such a system probably look like?

   Answer: *Without modification it would not necessarily be possible, as many other APIs rely on being handed over file-descriptors. However, if we modify those APIs and syscalls, we could update them to take a file name or inode number instead. The system would then have to look up the file whenever an access is made. This might still be possible, but the file-descriptor abstraction provides a way of keeping and scheduling I/O operations a lot easier.*

2. Can you always assume that your file descriptor is private to your process? If yes, how does the system ensure that? If no, what would you need to do to prevent race conditions?

   Answer: *No, by forking, both child and parent, share the same file descriptor, and both of them are valid. Thus modifications to the file could be overlapped if processes are forked. One way of preventing race conditions is to map a shared memory page in order to use this as a lock for reading/writing/modifying the file and its descriptor. Another option is to keep track of all descriptors and reopen all of them in the child, thus they are now private again.*
   If you answered yes, a description could look something like that: When we copy a process, we already pass through the entire pagetable in order to install the mappings in the child process. We could follow a similar logic and just go ahead and make a copy of the file descriptors.

3. We talked a lot about handling I/O in this exercise sheet, but the focus resided a lot with the filesystem and persistent storage. How does Unix model other types of I/O devices? How are they accessed and where are they exposed?

   Answer: *Unix has the infamous notion of "everything is a file". This also means, that there are files listed in the filesystem, that are not stored on the disk, but represent either an I/O device (e.g. the keyboard, mouse, screen) or abstract concepts (e.g. processes). We can use the same mechanisms to read, write and enforce security policies on those files as with any normal file. That makes the handling of devices and abstract information very simple in Unix. Even though the abstraction means that it is easy to interact with it in terms of mechanisms used, it can sometimes result in a convoluted interface for device operations.*

4. Practical Exercise: You might got to know the `ls` tool in the labs (e.g. xv6 has one, Linux or MacOS also have a `ls` tool). In this exercise, you will be implementing your own version of the ls tool. The output should correspond to the output of `ls -l --color=never` (sample output below). You might find it useful to read up on the `opendir`, `readdir` and `fstat` functions.

```
$ ls
drwxr-xr-x 2 username groupname    4096 Apr  4 07:24 lab1
-rw-r--r-- 1 username groupname 3355648 Apr  4 07:26 lab1.pdf
drwxr-xr-x 2 username groupname    4096 Apr  4 07:24 lab2
-rw-r--r-- 1 username groupname 2843648 Apr  4 07:26 lab2.pdf
drwxr-xr-x 2 username groupname    4096 Apr  4 07:24 lab3
-rw-r--r-- 1 username groupname 4441088 Apr  4 07:27 lab3.pdf
drwxr-xr-x 2 username groupname    4096 Apr  4 07:24 lab4
-rw-r--r-- 1 username groupname 3738624 Apr  4 07:27 lab4.pdf
```

Listing 1: Sample output for the `ls` utility

```c
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <stdlib.h>
```

```c
#include <dirent.h>
#include <time.h>
#include <grp.h>
#include <pwd.h>
#include <unistd.h>

void print_entry(char *name, struct stat *s) {
    if (name[0] == '.')
        return; // hidden file

    // get user & group information
    struct passwd *uid = getpwuid(s->st_uid);
    struct group *gid = getgrgid(s->st_uid);

    // prepare permission string
    // if no permission, print -
    char permissions[11];
    memset(permissions, '-', 10);
    permissions[10] = '\0';

    // Get entry type
    char type = '-';
    switch(s->st_mode & S_IFMT){
    case S_IFBLK:
        type = 'b';
        break;
    case S_IFCHR:
        type = 'c';
        break;
    case S_IFDIR:
        type = 'd';
        break;
    case S_IFIFO:
        type = 'p';
        break;
    case S_IFLNK:
        type = 'l';
        break;
    case S_IFSOCK:
        type = 's';
        break;
    }
    permissions[0] = type;

    // extract owner permissions
    if (s->st_mode & S_IRUSR) permissions[1] = 'r';
    if (s->st_mode & S_IWUSR) permissions[2] = 'w';
    if (s->st_mode & S_IXUSR) permissions[3] = 'x';

    // extract group permissions
    if (s->st_mode & S_IRGRP) permissions[4] = 'r';
    if (s->st_mode & S_IWGRP) permissions[5] = 'w';
    if (s->st_mode & S_IXGRP) permissions[6] = 'x';

    // extract other permissions
    if (s->st_mode & S_IROTH) permissions[7] = 'r';
    if (s->st_mode & S_IWOTH) permissions[8] = 'w';
    if (s->st_mode & S_IXOTH) permissions[9] = 'x';
```

```
64
65      // last modified time
66      char mtime[32];
67      strftime(
68          mtime,
69          32,
70          "%Y-%m-%d %H:%M:%S",
71          localtime(&(s->st_mtim.tv_sec))
72      );
73
74      // print single entry
75      printf("%s\t%i\t%s\t%s\t%lu\t%s\t%s\n",
76          permissions, s->st_nlink, uid->pw_name,
77          gid->gr_name, s->st_size, mtime, name);
78  }
79
80  int main(int argc, char *argv[]) {
81      DIR *dir;
82      struct stat statbuf;
83      struct dirent *direntry;
84      char *dir_string;
85
86      // open directory (either current or argument)
87      if (argc > 1)
88      {
89          dir_string = argv[1];
90          dir = opendir(argv[1]);
91      }
92      else
93      {
94          dir_string = malloc(2*sizeof(char));
95          dir_string = ".";
96          dir = opendir(".");
97      }
98
99      // Opening dir failed - print message and exit
100     if (dir == NULL) {
101         printf("Couldn't open dir %s", dir_string);
102         exit(1);
103     }
104
105     // Loop over all entries in the directory
106     while ((direntry = readdir(dir)) != NULL) {
107         lstat(direntry->d_name, &statbuf);
108         print_entry(direntry->d_name, &statbuf);
109     }
110
111     exit(0);
112 }
```

Listing 2: Possible implementation of a simple `ls` tool