TDT4225 - Exercise 2

Håkon Ødegård Løvdal

October 2015

1 I/O Programming

Comments and results

For running this code I used a MacBook Pro (Retina, 13-inch, Late 2013). This machine has the following spesifications:

Processor 2,4 GHz Intel Core i5

Memory 8 GB 1600 MHz DDR3

Hard Drive Apple SSD with 250 GB, Trim Enabled

File System HFS Plus (OS X Extended), journaled

As listed above, the machine which I was running this code used the HFS Plus (journaled) file system. On my system, the default blocksize for this filesystem is set to 4096 KB. Example output from running the program code:

```
HFS Plus  Throughput  Time
---------------------------------
 1 GB  297 MB/s   3359 ms
 2 GB  345 MB/s   5794 ms
 4 GB  343 MB/s  11651 ms
 8 GB  348 MB/s  22951 ms
16 GB  339 MB/s  47065 ms
32 GB  352 MB/s  90876 ms
```

In my program I perform a sequential write to the disk. I made some attempts to create both a parallell and sequential write, but did not succeed to increase the overall performance. Therefore the final code is the sequential version.

2 Various questions on file systems

a)

The disk map contains 13 pointers, were the first ten (0-9) are direct pointers to data blocks. Thereafter the 11th block is a indirect block with 256 pointers to data blocks. The 12th block is a double indirect block. This means it contains 256 pointers to indirect blocks. The last, 13th, block is a tripple indirect block. This means it cointains 256 pointers to double indirect blocks. Given this pre requisites and the fact that a data block is 1 KB we can calculate the max file size:

- 10 pointers to disk blocks = 10 KB
- Indirect block: 256 pointers to disk blocks = 256 KB
- Double indirect block: 256 pointers to indirect blocks = $256 * 256$ KB = 65536 KB = 64 MB
- Triple indirect block: 256 pointers to double indirect block = $256 * 65536$ KB = 16777215 KB = 16384 MB = **16 GB**

b)

The fact that S5SF stores inodes on disk is not a problem due to a implementation secific detail. Unlike the representation of free disk blocks (in a linked list), free inodes are marked free or not on the disk. The superblock contains a cache holding indices of free inodes. So in a situation where the a system call to create or delete a file is made, the system will allocate an inode at the index of the first free inode found in the cache (and opposite for deletion). In such an event, where all inodes are marked as allocated, the i-list will be scanned to find available inodes. This way of implementing inodes exploits the fact that inodes are allocated and freed much more rarely than data blocks. Also, in the event of an system crash, the information about inode allocation is stored on disk, and not in the primary memory, which is lost in such an event.

c)

The main attempts for FFS to improve weaknesses with S5SF were to reduce seek time and to use a larger block size. The reason for this were to increase the overall disk thorughput during read/write. Also, since the processors became faster, the bottleneck in file systems needed to be coped with.

Block size

FFS implements a larger block size than S5SF. The small block size in S5SF yields that files often will be spread across the disk. This exploits the "principle of locality" in a bad manner. In FFS, with its larger block size, files are often closer in terms of locality. But the larger block size ain't perfect either. When the block size increase, the internal fragmentation also increases.

Seek time

See the section about extent-based allocation and cylinder groups for a deeper elaboration about this.

Block interleaving

FFS also implements a technique called block interleaving. Ths technique places blocks of a file with some spacing. Rather than placing them in contiguous disk locations, it places the next block a few block locations after the first. The primary reason for using this technique is the fact that when the read/write head has read a block, the processor must start the next operation when it receives the interupt message. By the time this operation starts, the read/write head is near the end of the next block, and therefore it must rotate one revolution to read the start of the block.

d)

Both soft updates and journaling has the goal of taking a file system (in the non-volatile memory) from one consistent state to another, with transactions that obey the "ACID" rules. They both can not guarantee that no data will be lost during an unexpected event, but they will ensure that the file system went from one consistent state to another. In an event of system failure, all data in a volatile memory is lost.

Journaling

A journaling file system keeps a log of all actions that are not yet commited in the file system. This is to ensure that if an event, such as a system crash, occurs, it will be possible to bring the file system back to the consistent, correct state it had before the crash. Typically this journaling is managed by a journal thread (often called journald in *nix systems). In many ways this can be compared to a database, holding information about the file system state. The journaling approach guarantees atomicity in all meta-data operations.

Soft updates

Soft updates encounters the preserving of a consistent state by guaranteeing that blocks are written to disk in an valid order. By maintaining constraints and dependency between data it will ensure a consistent state. During file creation, the system much ensure that the new inode reaches disk before the directory that references it does. This is the key point of soft updates. By keeping information about this, the directory dependent of the inode is written after the inode. The soft updates approach guarantees that the ordering constraints is followed, in an asynchronoulsy way.

e)

Extent-based allocation

Extent-based allocation exploits the idea that if you access a file, you would most likely read files close to it in locality. This makes it acheive good performance for sequential file access. In an extent-based allocation we only need to read the start block number and the length of the file. Thereafter we read all the data blocks in that extent. This means that we only need to read the meta-data once. This approach gives little meta-data for large files. One problem with extent-based allocation is that extents are of variable size. This opens the possibility for external fragmentation to be a problem.

Cylinder groups

Cylinder groups exploits the idea that data that belonging together should be put together. A cylinder group is space big enough to store its collection of inodes, data blocks and indirect blocks (including meta-data). By doing this one reduces seek time and latency for retreiving files. Another point to mention is that this may reduce fragmentation because a directory's content is not spread across the disk.

f)

RAID 0 All the data is spread throughout all disks in the RAID. This gives no redundancy, so if one of the disks fails, all the data is lost. Though, RAID 0 will increase the overall performance in most cases.

RAID 1 This level is simple disk mirroring. Two identical copies of the file system is kept on seperate disks. Write calls are done on both disks at the same time. Read calls is done on one of the disks.

RAID 2 This level uses bit-level stripping with ECC/Hamming-code parity checks. The data is synchronized in such a way that the each sequential bit is on a different drive. The parity bits are stored on at least one parity drive.

RAID 3 This level is an improvment of level 2. It uses a dedicated parity disk for error. Like level 1, data is striped in such a way that each sequential byte is on a different disk. Parity is calculated as XOR of the data bits and stored on the parity disk.

RAID 4 Level 4 is just like level 3, but has block-level striping rather than bit-level. Since all data is not spread across all the disks, level 4 opens for parallel read/write operations.

RAID 5 Level 5 fixes the problem with check disks, since here all disks hold check blocks. If one disk fail, the reads missing can be calculated from the distributed parity bits. No data loss.

RAID 6 Extends the RAID 5 with another parity block. All disks has this parity blocks.

3 Source code

```c
#include <sys/types.h>
#include <sys/stat.h> /* struct stat */
#include <sys/time.h> /* gettimeofday etc */
#include <fcntl.h> /* fcntl, fflush etc */
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h> /* printf etc */
#include <math.h> /* Pow operator */

/* For preprocessoring */
#define ONE_GIGABYTE pow(1024, 3) /* Size of one GB in B */
#define FILENAME "data.out"

/* Global variables */
struct stat fi;
blksize_t BLOCKSIZE; /* Keeps the blocksize for the current filesystem */

/* Functions */
void print_time_for_interval(struct timeval *start, struct timeval *end, int gb);
void seq_write(int gb);

int main(int argc, char** argv) {
  stat("/", &fi); /* System stat struct for root */
  BLOCKSIZE = fi.st_blksize; /* Set filesystem blocksize */

  printf("HFS Plus\t Throughput\t Time\n");
  printf("----------------------------------------\n");

  for (int i = 0; i < 6; i++) {
    seq_write(pow(2, i));
  }

  unlink(FILENAME); /* Delete the file after all files written, since we are done with it */
}

void seq_write(int gb) {
  struct timeval start, end;
  FILE *fp;
  char block[BLOCKSIZE];
  memset(block, '#', BLOCKSIZE);

  int length = (gb * ONE_GIGABYTE) / BLOCKSIZE;

  fp = fopen(FILENAME, "w+");
  gettimeofday(&start, NULL);
  for (int i = 0; i < length; i++) {
    fwrite(block, BLOCKSIZE, 1, fp);
  }
  gettimeofday(&end, NULL);
  fclose(fp);

  print_time_for_interval(&start, &end, gb);
}

void print_time_for_interval(struct timeval *start, struct timeval *end, int gb) {
  double time_in_ms;
  long start_t = start->tv_sec * 1000000 + start->tv_usec;
  long end_t = end->tv_sec * 1000000 + end->tv_usec;;
  time_in_ms = ((end_t - start_t) / 1000000.0) * 1000.0 ;
  printf("%2d GB\t %3d MB/s\t %5d ms\n", gb, (int)(gb * (1000*1000) / (time_in_ms)), (int)time_in_ms);
}
```