CS2106 Operating Systems

Semester 1 2021/22

Tutorial 11 Solution

1. **(Putting it together)** This question is based on a "simple" file system built from the various components discussed in lecture 12.

Partition information (free space information):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
1	0	1	0	1	0	1	0	1	1	1	0	0	1	0	0

Directory Structure + File Information:

- Directory structures are stored in 4 "directory" blocks. Directory entries (both files and subdirectories) of a directory are stored in a single directory block.

Directory entry:

- For File: Indicates the first and last data block number.
- For Subdirectory: Indicates the directory structure block number that contains the subdirectory's directory entries.
- The "/" root directory has the directory block number 0.

0	1	2	3
y |Dir | 3	g |File| 0| 31	k |File| 6| 6	i |File| 1| 3
f |File| 12| 2	z |Dir | 2		h |File|27|28
x |Dir | 1			

File Data:

- Linked list allocation is used. The first value in the data block is the "next" block pointer, with "-1" to indicate the end of data block.
- Each data block is 1 KB. For simplicity, we show only a couple of letters/numbers in each block.

0	1	2	3	4	5	6	7
11 AL	9 TH	-1 S!	-1 ND	23 GS	-1 SO	-1 :)	10 TE
8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
31 RE	3 EE	28 M:	31 OH	19 SE	13 AH	4 IN	17 NO
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
30 YE	2 OU	1 ON	17 RI	26 EV	14 AT	21 DA	7 YS
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**
-1 HO	18 ME	0 AL	30 OP	-1 -(	5 LO	21 ER	-1 A!

a. (Basic Info) Give:
   - The current free capacity of the disk.
   - The current user view of the directory structure.

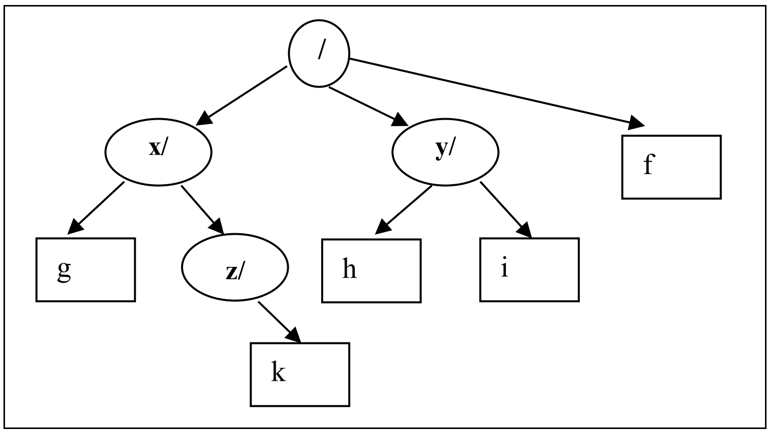
b. (File Paths) Walkthrough the file path checking for:
   - "/y/i"
   - "/x/z/i"

c. (File access) Access the entire content for the following files:
   - "/x/z/k"
   - "/y/h"

d. (Create file) Add a new file "/y/n" with 5 blocks of content. You can assume we always use the free block with the smallest block number. Indicate all changes required to add the file.

Ans:

a. ~12KB free space (12 '1' in Bitmap, each data block is 1KB). Due to the linked list file allocation overhead, the actual capacity is a little bit smaller.



b. Only the directory block number is indicated:
"/y/i" : 0, 3 (successful)
"/x/z/i": 0, 1, 2 (failed)

c. The file data block numbers are indicated, with the "content" pieced together.
"/x/y/k": 6, content = ":)"
"/y/h": 27, 30, 21, 14, 4, 23, 7, 10, 28, content="OPERATINGSYSTEM:-("

d. Bitmaps updated: Bit 5, 8, 13, 15, 16 changed to 0
Directory block 3 (for 'y') updated: "n | File | 5, 16" added
Data Blocks 5, 8, 13, 15, 16 (next block pointer changed, with -1 in block 16).

2. **(File System Overhead)** Let us find out the overhead of FAT16 and ext2 file systems. To have a meaningful comparison, we assume that there is a total of 2^{16} 1KB data blocks. Let us find out how much bookkeeping information is needed to manage these data blocks in the two file systems. Express the overhead in terms of number of data blocks.

   a. Overhead in FAT16: Give the size of the two copies of file allocation table.
   b. Overhead in ext2 is a little more involved. For simplicity, we will ignore the super block and group descriptors overhead.

      Below are some known restrictions:
      - The two bitmaps (data block and inode) each occupies a single disk block.
      - There are **184** inodes per block group.

      Calculate the following:
      i. Number of data block per block group.
      ii. Size of the inode table per block group.
      iii. Number of block groups in order to manage 2^{16} data blocks.
      iv. Combine (i – iii) to give the total overhead.

   c. Comment on the runtime overhead of the two file systems, i.e. how much memory space is needed to support file system operations during runtime.

Ans:

a. Each FAT table is 2^{16} x 16bit (2 bytes) = 2^{17} bytes.
   2 copies of FAT table = 2 x 2^{17} = 2^{18} bytes. This takes 2^{18} / 2^{10} = 2^{8} disk blocks.

   Note that due to the special codes (free, bad, eof etc), this setup actually handles less than 2^{16} data blocks. We ignore this fact for ease of comparison.

b.
   i. Each bitmap is in a 1KB block→ 8K bits → 8K data blocks per block group.
   ii. From restrictions, 184 inode x 128 bytes each / 1024 bytes data block = 23 disk blocks. Note that 184 inodes per block group is an arbitrary number.
   iii. From (i) 2^{16} blocks / 2^{13} per block group = 8 block groups.
   iv. Overhead per block group = 2 blocks of bitmaps + 23 blocks of inodes = 25 blocks. Total overhead = 8 x 25 blocks = 200 blocks.

c. FAT: The entire FAT table is in memory, i.e. 2^{17} bytes.
   Ext2: Nothing is needed. Though for efficiency, a couple of the recently accessed I-Node may be cached. As a comparison, with the overhead of FAT, we can cache 2^{17}/ 128 bytes = 2^{10} I-node in memory.

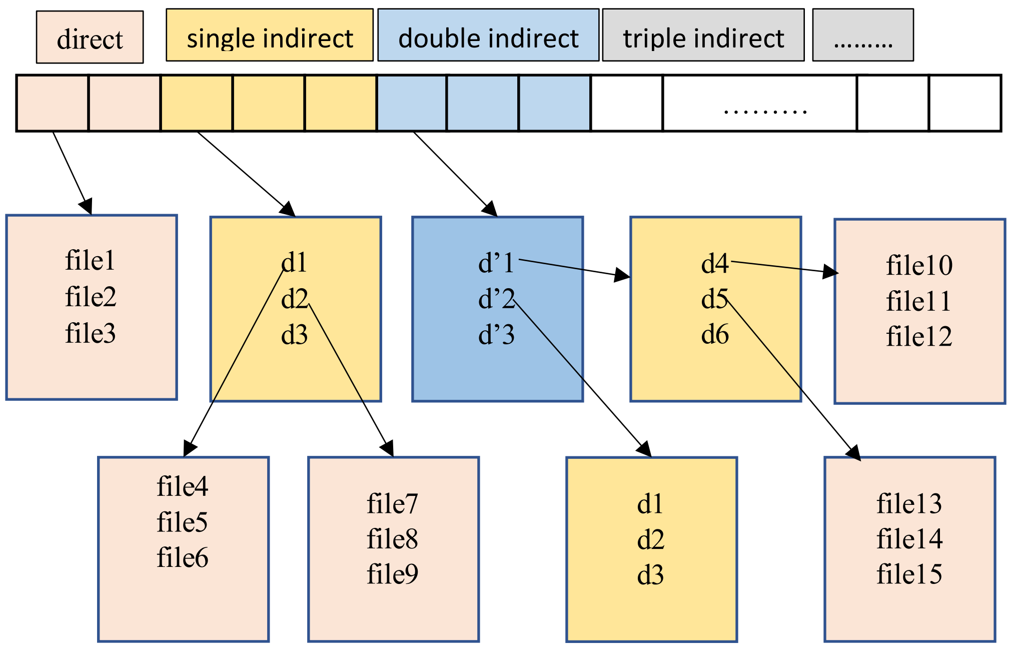
3. **(Adapted from AY 2019/20 S 1 exam)**
You are required to implement and optimize the storage and search of multiple fixed file sizes of 1MB. Each file has a unique numeric identifier, and it is saved on disk using ext2 file system. The number of files is unlimited (say, 10 million), but it cannot grow beyond the disk size (maximum 10TB). We want to optimize this file system to store and search fast for files by identifier. You should use some of the methods studied under file system implementation for your optimizations.

a. First, assume that all files are stored in a directory. What would be the main disadvantage when we want to add a new file? What would be the main disadvantage when searching for a file by identifier?
b. Assume that you do not have any restrictions in terms of how many directories you can use. How would you mitigate the disadvantages shown at point a. for storing a file? Explain your optimizations.
c. How would you mitigate the disadvantage(s) shown at point a. for searching a file by identifier? Explain your optimizations.
d. You are allowed to use additional data structures to speed up the storing and searching of files. What data structures would you use to implement your optimizations? Be specific about what you store in each data structure.
e. Explain how your suggestions (b, c, d) improve the storage and search times for a file. Estimate the overhead in terms of space and time of creating, maintaining, and using the additional data structures from point d.

Ans:

a. For storing a new file: need to traverse the whole directory (all DE in the linked list)
For searching, possibly need to search through the whole linked list.

b. Store a limited number of entries (X) in each directory. Use a hierarchical structure similar to what EXT2 is using. Shown below is an illustration of an example hierarchical structure.



There are three types of folders. Direct folders contain files. Single indirect folders contain folders that contain files. Double indirect folders contain single indirect folders. This scheme can be extended to introduce more levels.

c. Use a hash table to map the file identifier to the directory in which it is located. The hash table can be stored in memory because its size is: #files * (file_id_size + pointer_to_directory_block). 10 million files would occupy only 160MiB of RAM. The idea is that the memory overhead of the hash table is affordable, thus this scheme is feasible.

d. Hash table + hierarchical structure.The directory structure can be similar to direct and indirect blocks in ext2:

    Direct folders contain X files
    Single indirect – contains X other folders, each with X files.
    Double indirect - …

e. Access time to a file depends on X and the number of indirect folders. The access time for a file is X ($\log_X N + 1$). If N=10 million, we can choose X to minimize the access time.

In this problem, it is not enough to use:

- A binary search tree as a hierarchical directory structure. Maintaining the binary search tree would have a huge overhead.

- Split the file in folders based on their first digit in the identifier. This structure can be hierarchical or linear. If it is hierarchical, the overhead of traversing such a structure is too high when you have very few files.

- Maintain the files in a sorted order in the directory as the overhead of insertion sort in the list of directory entries is too high

There is no guarantee the identifiers are distributed uniformly to bins (based on first digits).

4. **(Adapted from AY 2020/21 S 1 exam)**
In a particular ext2 filesystem, many data blocks belonging to different files have the same contents. We can implement a mechanism to reduce the space on disk occupied by such files.

   a. Define a new system call, `clone_file()`, that makes a copy of a file without allocating new blocks on disk. Give the parameters, return value and description for `clone_file()`, and explain how to use the system call.
   b. How would you allow for files that are cloned by `clone_file` to later be modified independently of each other? For example, if file A is cloned by `clone_file` to file B, and later on a write to file A is done, file B should still reflect the original contents. You should avoid duplication of blocks that are not yet modified.
   c. Explain how you would implement the mechanism in (b) in the ext2 filesystem. If changes to structures are required, explain those as well.

a. Many answers are possible. The following answer is based off the real Linux system call `ioctl(FICLONE)`, which performs the exact operation described above (although it is currently supported by few filesystems—currently the only popular filesystems supporting it are btrfs and XFS). There is also a newer system call `copy_file_range` that is a bit more versatile.

SYNOPSIS

```
int clone_file(int srcfd, int destfd);
```

DESCRIPTION

If a filesystem supports files sharing physical storage between multiple files, this operation can be used to make the data in the `srcfd` file appear in the `destfd` file by sharing the underlying storage, which is faster than making a separate physical copy of the data. Both files must reside within the same filesystem. If a file write should occur to a shared region, the filesystem will ensure that the changes remain private to the file being written. This behavior is commonly referred to as "copy on write".

(Above description adapted from manpage `ioctl_ficlone(2)`)

RETURN VALUE

On error, `-1` is returned, and `errno` is set to indicate the error.

Error codes can be one of, but not limited to, the following:

- **`EINVAL`**: `srcfd` or `destfd` are not regular files, or the underlying filesystem does not support this functionality.
- **`EXDEV`**: `srcfd` and `destfd` are not on the same filesystem.
- **`EBADF`**: `srcfd` is not open for reading, or `destfd` is not open for writing.
- Etc…

EXAMPLE FOR USAGE

```
int srcfd = open("srcfile", O_RDONLY);
int destfd = open("destfile", O_WRONLY);
clone_file(srcfd, destfd);
```

Other answers are possible, for example, passing paths instead of FDs to the syscall. (This is the answer that is in the suggested answers to the AY 2020/21 exam.)

b. Copy-on-write. The blocks should be marked as shared somehow (detailed in part c), and then when those blocks are modified, a copy of the block should be made for the file through which the block is being modified.

c. Various answers are possible here, with varying complexity and overhead.

- We can have a reference count for each block.
- Or, we can have a ‘shared’ bit for each block. If a block that has the ‘shared’ bit set is written to, copy-on-write is done. However, some way to detect that a block is no longer shared is still needed.

To store a reference count:

- The simplest way is to have a new block refcount table, perhaps one byte per block, in each block group. This can in fact replace the block usage bitmap: a block with refcount 0 is free.
    - This scheme is simple, but does increase overhead quite a bit, perhaps unnecessarily, as shared blocks are likely to be the minority in a filesystem.
    - Some optimisations are possible: e.g. only have the refcount table in block groups that have shared blocks, or use a smaller field for the refcount e.g. only 4 bits instead of a full byte.
- Another way is to have a balanced search tree (such as a btree) mapping block indexes to refcounts, containing only those blocks whose refcounts are 2 or more. Each block group that contains shared blocks has a separate btree.
    - However, without other changes elsewhere, this means that every modification to a block in a block group containing a shared block will require a lookup in the btree, even if the block itself is not shared.

Using the reference count:

- When a new block is allocated normally, its reference count is initialised to 1
- When a file (inode) is deleted, all the blocks referenced by the inode should have their refcount decremented
- When a block is modified, its refcount should be checked; if the refcount is more than 1, then copy-on-write should be done, and the refcount of the original block decremented (so once there is only one inode referring to that block, that block can be modified normally again)
- If using the btree mechanism mentioned above, then the block is inserted into the btree when its refcount becomes 2 or more, and removed when its refcount is decremented to 1

Questions for your own exploration

5. **(Adapted from AY 2016/17 S 1 exam)**
Most OSes perform some higher-level I/O scheduling on top of just trying to minimize hard disk seeking time. For example, one common hard disk I/O scheduling algorithm is described below:
    - **a.** User processes submit file operation requests in the form of **operation(starting hard disk sector, number of bytes)**
    - b. OS sorts the requests by hard disk sector.
    - c. The OS merges requests that are nearby into a larger request, e.g. several requests asking for tens of bytes from nearby sectors merged into a request that reads several nearby sectors.
    - d. The OS then issues the processed requests when the hard disk is ready.

a. How should we decide whether to merge two user requests? Suggest two simple criteria.

b. Give one advantage of the algorithm as described.

c. Give one disadvantage of the algorithm and suggest one way to mitigate the issue.

d. Strangely enough, the OS tends to intentionally delay serving user disk I/O requests. Give one reason why this is actually beneficial using the algorithm in this question for illustration.

e. In modern hard disks, algorithms to minimise disk head seek time (e.g. SCAN variants, FCFS etc.) are built into the hardware controller. i.e. when multiple requests are received by the hard disk hardware controller, the requests will be reordered to minimise seek time. Briefly explain how the high-level I/O scheduling algorithm described in this question may conflict with the hard disk's built-in scheduling algorithm.

Ans:

a. Criterion 1: Requests are in the same or nearby sector (can mention cluster size).
Criterion 2: Requests are of the same type, read / write.
b. Advantage: Seeking latency is reduced.
c. Disadvantage: Potential starvation for user process if the request is not near to existing requests.
Mitigate: Take the request time into account and set certain deadline. Once the deadline is near, issue request regardless of whether it can be merged.
d. Reason to delay: Disk I/O request has very high latency. Delaying the user request for several hundred machine instructions (note that instruction execution time is in the ns range) will not increase the waiting time significantly. However, with more user requests pending, OS can optimize the I/O better. If we do not have enough I/O requests to choose from, merging will not be very effective.
e. Potential conflict: It may turn out that the hard disk controller schedules the requests differently. In the worst case, the scheduling decision by OS may be undone by the controller → time used for sorting / merging are wasted.

6. **This question is not in scope for the exam.**
(Cluster Allocation, Adapted from [SGG]) A common modification to the file allocation scheme is to allocate several contiguous blocks instead of a single disk block for every allocation. This variation is known as the disk block cluster in FAT file systems. The design of cluster can be one of the following:

   a. Fixed cluster size, e.g. one cluster = 16 disk blocks.
   b. Variable cluster size, e.g. one cluster = 1 to 32 disk blocks.
   c. Several fixed cluster size, e.g. one cluster = 2, 4, 8 or 16 disk blocks.

   Discuss the changes to the file information in order to support cluster. Briefly state the general advantage and disadvantage of the various cluster design.

   ANS:

   For (a), it is the same as the original single disk block approach, only that the disk block is now larger. So, there is no change to the file information.

   For (b), the cluster size needs to be stored as part of the file block information. In this case, at least 5 bits ($2^5 = 32$) is needed. This overhead can be quite substantial, e.g., in linked list allocation, a cluster must contain a pointer + the size for the next cluster.

   For (c), the cluster size needs to be stored as well. However, due to smaller number of choices, lesser number of bits is needed. In this case, 4 choices can be represented by 2 bits.

   In general:

   - Cluster reduce disk accesses (e.g. in the linked list case), reduce overhead (more obvious in scheme (a)) and improve access speed for consecutive blocks.
   - Having more cluster sizes can reduce the chance of external fragmentation.
   - Having less cluster sizes can increase the chance of internal fragmentation.