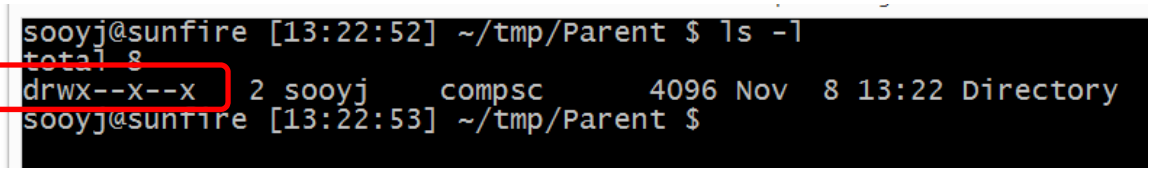
CS2106 Operating Systems
Semester 1 2020/2021

Week of 12th April 2021
Tutorial 11 Suggested Solutions
File System Implementation

1. [Deferred from T10 - Understanding directory permission] In *nix system, a directory has the same set of permission settings as a file. For example:



You can see that directory **`Directory`** has the read, write, execute permission for owner, but only execution permission for group and others. It is easy to understand the permission bits for a regular file (read = can see the file content, write = can modify, execute = can execute this file). However, the same cannot be said for the directory permission bits.

Let's perform a few experiments to understand the permission bits for a directory.

Setup:
- Unzip **`DirExp.zip`** on any *nix platform (Solaris, Mac OS X included).
- Change directory to the **`DirExp/`** directory, there are 4 subdirectories with the same set of files. Let's set their permission as follows:

`chmod 700 NormDir`	`NormDir` is a normal directory with read, write and execute permissions.
`chmod 500 ReadExeDir`	`ReadExeDir` has read and execute permission.
`chmod 300 WriteExeDir`	`WriteExeDir` has write and execute permission.
`chmod 100 ExeOnlyDir`	`ExeOnlyDir` has only execute permission.

Perform the following operations on each of the directory and note down the result. Make sure you are at the **`DirExp/`** directory at the beginning. DDDD is one of the subdirectories.

a. Perform "**`ls -l DDDD`**".
b. Change into the directory using "**`cd DDDD`**".
c. Perform "**`ls -l`**".
d. Perform "**`cat file.txt`**" to read the file content.
e. Perform "**`touch file.txt`**" to modify the file.
f. Perform "**`touch newfile.txt`**" to create a new file.

Can you deduce the meaning of the permission bits for directory after the above? Can you use the "directory entry" idea to explain the behavior?

ANS:

	ReadExeDir	WriteExeDir	ExeOnlyDir
a	ok	nope	nope
b	ok	ok	ok
c	ok	nope	nope
d	ok	ok	ok
e	ok	ok	ok
f	nope	ok	nope

The responses may seem arbitrary unless you apply the understanding that "Directory == the list of directory entries (file/subdir)".

Then, the permission can be understood as:

Read = Can you read this list? (impact: ls, <tab> auto-completion)

Write = Can you change this list? (impact: create, rename, delete file/subdir). Note the interaction with the execute permission bit.

Execute = Can you use this directory as your working directory? (impact: cd).

Since modifying the directory entries (i.e. with write permission) requires us to set the current working directory, execute bit is needed for the write-related operations under the target directory.

Some interesting scenarios:

a. Directory permission is independent from the file permission. So, you still can modify a file under a "read only" directory if the file allows write.

b. If you want to allow outsider to access a particular deeply nested file, e.g. A/B/C/D/file, you **only need** execute bit on A, B, C, D directory (i.e. read permission is not needed). This is a great way to hide the content of the directory and only allow access to specific file given the full pathname.

2. [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)

- Bitmap is used, with a total of 32 file data blocks (1 = Free, 0 = Occupied):

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

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`**"

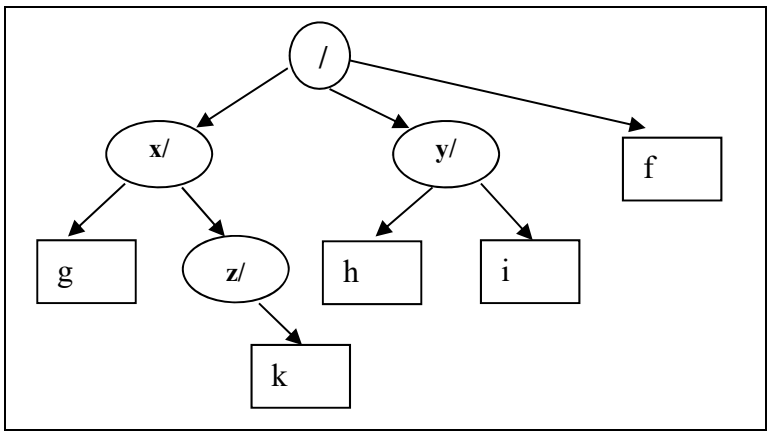
c. (File access) Access the entire content for the following files:
   - "**`/x/z/k`**"
   - "**`/y/h`**"

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.

- **`/x/z/k`**: block 6, content = "`:)`"
- **`/y/h`**: blocks 27, 30, 21, 14, 4, 23, 7, 10, 28, content = "`OPERATINGSYSTEM:-(`"

d.

Bitmap 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).

3. [Disk I/O Scheduling] The example used in the disk I/O scheduling section in lecture 12 assumes that all requests arrive at the same time. Specifically, all requests are assumed to arrive at time 0.

   If possible, give an ***arrival timeline*** for the same set of requests {13, 14, 2, 18, 17, 21, 15} that results in a different schedule under i) FCFS and ii) C-SCAN algorithm. For simplicity, assume we can service one I/O request per time unit.

   Note: this means that the **order of the requests remains unchanged**, but they may now arrive at different timing. One example timeline:

Arrival Time	T0	T0	T2	T4	T7	T9	T10
Request	13	14	2	18	17	21	15

ANS:
There is no answer for FCFS, as it is entirely dependent on the ordering of the requests.

There are many answers for C-SCAN, essentially just make the request arrives after the sweep, then it'll miss the schedule.

Assuming disk head starts at 13,
Original Schedule: 13, 14, 15, 17, 18, 21, 2
One possible answer:

Arrival Time	T0	T0	T2	T2	T6	T6	T10
Request	13	14	2	18	17	21	15

Schedule = 13, 14, 18, **2,** 17, 21, **15**

4. [AY16/17 Sem1 Exam Question] 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:
   i. User processes submit file operation requests in the form of
      ***operation*(starting hard disk sector, number of bytes)**
   ii. The OS sorts the requests by hard disk sector.
   iii. 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.
   iv. 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 schedule 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.