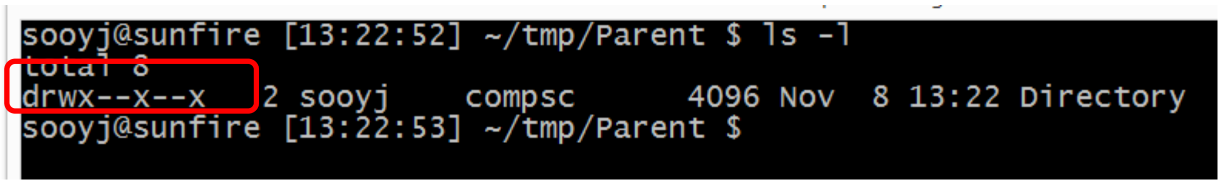
CS2106 Operating Systems
Tutorial 11 Suggested Solutions
File System Implementation

(Note to TA: Skip Q1 if you already did it in T10)

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



```
sooyj@sunfire [13:22:52] ~/tmp/Parent $ ls -l
total 8
drwx--x--x   2 sooyj    compsc      4096 Nov   8 13:22 Directory
sooyj@sunfire [13:22:53] ~/tmp/Parent $
```

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

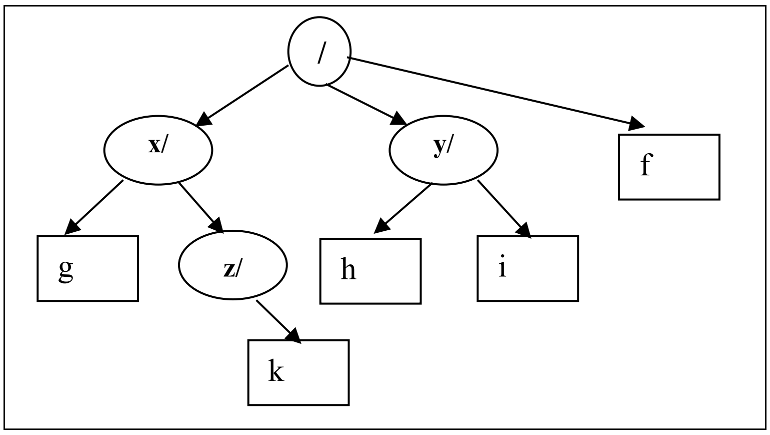
c. (File access) Access the entire content for the following files:
   - "`/x/z/k`"
   - "`/y/h`"

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. [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.

Question for your own Exploration

1. (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.

2. (File Implementations Comparison) Let us compare and contrast the various ways of implementing files in this question.

Below are the hardware parameters:

a. Number of disk blocks = 2^{16} = 65,536 blocks (i.e. each disk block number = 2 bytes).
b. Disk block size = 512 bytes.

Points of comparison:

- **Total Number of Data Blocks:** Number of disk blocks needed to store the file data.
- **Overhead**: Extra bookkeeping information in bytes. Focus only on information directly related to keep track of file data. You can ignore the space needed for file name and other metadata for this question.
- **Worst Case Disk Access:** The worst case number of disk accesses needed to get to a particular block in the file. May include accessing book keeping information if they are in disk. Specify the block which causes the worst case.

File data allocation schemes under consideration:

A. Contiguous
B. Linked list
C. File allocation table
D. Index block

Fill in the following table for the indicated file size. State assumptions for your calculation (if any).

File size: 100 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A			
B			
C			
D			

File size: 132,000 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A			
B			
C			
D			

File size: 33,554,432 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**

A			
B			
C			
D			

ANS:

Note that with different assumptions, you may need to adjust your answer accordingly.
Assumptions:
For A, the length of blocks is 2 bytes (i.e. can take all disk blocks if needed).
For B, we keep pointers to both first and last block.
For D, we use the "linked" scheme to chain up the index blocks when needed.

Basic Calculation:
For B, one disk block can store 512-2 (the "next" disk block pointer) = 510 bytes only.
Although the "next" disk block pointer overhead is already reflected by the larger number of disk blocks required, we still list them explicitly for learning purpose.
For C, the whole FAT cost $2^{16} * 2 = 2^{17}$.
For D, each index block can store 512 / 2 = 256 disk addresses. We use the last disk address to chain to the next index block, i.e. 255 disk addresses can be stored for file data blocks.

File size: 100 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A	**1**	**2 (start) + 2 (length)**	**1**
B	**1**	**2 (first) + 2 (last) + 2 (1 pointer)**	**1**
C	**1**	**2^{17} (FAT) + 2 (start)**	**1**
D	**1**	**512 (Index Block) + 2 (Location of Idx Blk)**	**1 (Index block) + 1**

File size: 132,000 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A	**258**	**2 (start) + 2 (length)**	**1**
B	**259**	**2 (first) + 2 (last) + 518 (259 pointers)**	**258 access for the 2nd last block.**
C	**258**	**2^{17} (FAT) + 2 (start)**	**1**
D	**258**	**1024 (2 × Index Block) + 2 (Location of Idx Blk)**	**2 (Index block) + 1 for any block beyond 255th.**

File size: 33,554,432 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**

A	**65,536**	**2 (start) + 2 (length)**	**1**
B	**Cannot store (Need 65,794)**	**---**	**---**
C	**65,536**	**2^{17} (FAT) + 2 (start)**	**1**
D	**Cannot store (Need 65,794)**	**---**	**---**