7 File Systems [12 points]

Describe how each of these following techniques is intended to improve I/O performance, and then describe one case where the technique can be expected to have little or no effect on performance.

1. Disk scheduling

   Disk scheduling aims to reduce seek time by using techniques such as shortest seek first scheduling.

   It will have little benefit if there are very few requests in the scheduler queue at a time.

2. File system block caching

   With file system block caching, disk blocks are cached in memory. Accesses to these cached blocks require memory access rather than much slower disk access.

   This technique will have little benefit if the program accesses the file randomly (no spatial locality) or accesses the file sequentially only once (no temporal locality).

3. File block placement

   The logical file blocks are placed sequentially on disk to reduce seek and rotation times.

   This technique will have little benefit if the program accesses the file randomly (no spatial locality).

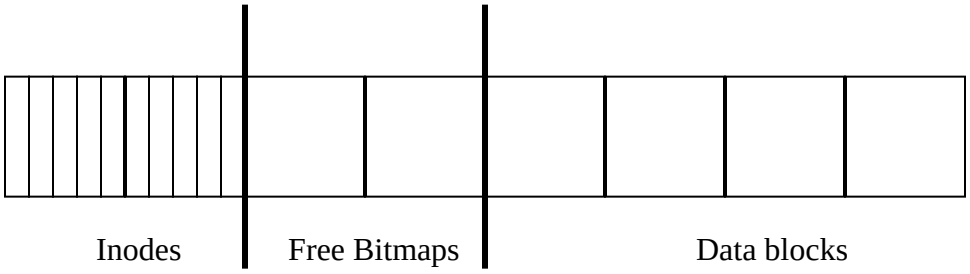
4. Increasing file-system block size

   Increasing the block size reduces seek and rotation time associated with accessing multiple smaller blocks.

   This technique will have little benefit if say the first byte of each larger block is accessed by a program (no spatial or temporal locality) or if the average file size is much smaller than the block size.

7. My Data is Corrupted@!@ [10 marks]

You have just bought a smart phone and have decided to port OS161 to it. To keep things simple, you design a file system without directories. The disk is physically partitioned into three regions: an inode list, a free list, and a collection of 4K data blocks, much like the UNIX file system. Unlike in the UNIX file system, each inode contains the name of the file it corresponds to, as well as a bit indicating whether or not the inode is in use. Like the UNIX file system, the inode also contains a list of blocks that compose the file, as well as metadata about the file, including permission bits, its length in bytes, and modification and creation timestamps. The free list is a bitmap, with one bit per data block indicating whether that block is free or in use. There are no indirect blocks in your file system. The following figure illustrates the basic layout of your file system:



The file system provides six primary calls: CREATE, OPEN, READ, WRITE, CLOSE, and UNLINK. You implement all six in a straightforward way, as shown in the next page. All updates to the disk are synchronous; that is, when a call to write a block of data to the disk returns, that block is definitely installed on the disk. Individual block writes are atomic.

```
CREATE(filename) {
  scan all non-free inodes to avoid duplicate filenames;
  return error if duplicate;
  find a free inode in the inode list;
  update the inode with 0 data blocks, mark it as in use,
    write inode to disk;
}

OPEN(filename) { // returns a file handle
  scan non-free inodes looking for filename;
  if found, allocate and return a file handle fh
    that refers to that inode;
}

WRITE(fh, buf, len) {
  look in file handle fh to determine inode of the file,
    read inode from disk;
  if there is free space in last block of file, write to it;
  determine number of new blocks needed, n;
  for i = 1 to n {
    use free list to find a free block b;
    update free list to show b is in use,
      write free list to disk;
    add b to inode, write inode to disk;
    write appropriate data for block b to disk;
  }
}

READ(fh, buf, len) {
  look in file handle fh to determine inode of the file,
    read inode from disk;
  read len bytes of data from the current location
    in file into buf;
}

CLOSE(fh) {
  remove fh from the file handle table
}

UNLINK(filename) {
  scan non-free inodes looking for filename,
    mark that inode as free;
  mark data blocks used by file as free in free list;
  write inode to disk;
  write modified free list blocks to disk;
}
```

A) [4 marks] You notice that your file system works correctly, even if multiple processes access files concurrently, as long as the system is up and running. However, if you remove the batteries from your phone while the following application is running, and then replace the batteries and reboot your system, then the file this application created exists but contains unexpected data that you never wrote to the file.

```
CREATE(filename);
fh = OPEN(filename);
// app_data is some data to be written
WRITE(fh, app_data, LENGTH (app_data));
CLOSE(fh);
UNLINK(filename);
```

Which of the following are possible explanations for this behavior? (Assume that the disk controller never writes partial blocks.) Circle the correct answers.

1. The free list entry for a data page allocated by the call to WRITE was written to disk, but neither the inode nor the data page itself was written.
2. The inode allocated to your application previously contained a (since deleted) file with the same name. If the system crashed during the call to CREATE, it may cause the old file to reappear with its previous contents.
3. The free list entry for a data page allocated by the call to WRITE as well as a new copy of the inode were written to disk, but the data page itself was not.
4. The inode allocated to your application was updated by the call to UNLINK but the free list was not.

B) [6 marks] Your trusted friend, S. Geek, is an OS expert. She checks your file system code and says that it is possible for the same block to be in use by multiple files after a crash. Is she correct? If so, show a scenario that causes this situation. Otherwise, argue why this is not possible.

The unlink call to file A can mark an inode as free and marks data blocks of file A as free in its free list. Then it can unlock the free list (it can reacquire a lock on the free list when the free list is written to disk without introducing races). A write can allocate a freed block of file A from the free list to another file B. Before the unlink call writes inode to disk, a crash occurs. On reboot, file A will not appear to be deleted and its block will be allocated to file B also.

Alternatively, if a lock on the free list is acquired for the entire duration of the unlink operation, then a block cannot be in use by multiple files after a crash. The reason is that an unlink removes blocks from inode and then marks blocks as free in free list, while a write marks blocks in use in the free list before updating inode. So a crash anytime in between cannot cause a block to be linked to multiple inodes.

8. Logging [15 marks]

You have decided that your smart phone file system is not very reliable and have decided to reimplement it. Your insight is that you store little data in your file system, e.g., your friend's addresses, and so you can keep your file system entirely in memory. You still need to deal with battery failures and other system crashes and have decided to use logging. On any error, the system crashes immediately, and you plan to reconstruct the in-memory file system state from the log.

Your file system operations uses transactions, each identified by a unique transaction ID. The visible effect of a completed file system transaction is confined to changes to global variables in memory, whose WRITE operations are logged. The log will contain entries recording the following operations:

```
BEGIN(tid) // start a new transaction, whose unique ID is tid
COMMIT(tid) // commit a transaction
ABORT(tid) // abort a transaction
// write a global variable, specifying previous & new values.
WRITE(tid, variable, old_value, new_value)
```

In addition, your system provides a `READ (tid, variable)` call which returns the current value of a variable. Reading an unwritten variable returns ZERO.

In your answers below, please justify your answer in one or two lines. Longer answers will be penalized.

A) [2marks] Assume that only one transaction is active at a time. You need to reset variables to committed values after a crash. What value would you use for variable v? Circle one or more that apply:

1. 0
2. new_value from the last logged `WRITE(tid, v, old_value, new_value)` or ZERO if unwritten.
3. new_value from the last logged `WRITE(tid, v, old_value, new_value)` that is not followed by an `ABORT(tid)`, or ZERO if unwritten.
4. new_value from the last logged `WRITE(tid, v, old_value, new_value)` that is followed by a `COMMIT(tid)`, or ZERO otherwise.
5. Either old_value or new_value from the last logged `WRITE(tid, v, old_value, new_value)`, depending on whether or not that `WRITE` is followed by a `COMMIT` on the same tid, or ZERO if unwritten.

B) [2 marks] You now try running concurrent transactions on your system. Accesses to the log are serialized by your disk driver. Consider the following two transactions:

`BEGIN (t1)`
`t1x = READ (x)`
`t1y = READ (y)`
`WRITE (t1, x, t1x, t1y + 1)`
`COMMIT (t1)`
	`BEGIN (t2)`
	`t2x = READ (x)`
	`t2y = READ (y)`
	`WRITE (t2, y, t2y, t2x + 2)`
	`COMMIT (t2)`

The initial values of x and y are ZERO, as are all uninitialized variables in the system. Here the `READ` primitive simply returns the most recently written value of a variable from memory, ignoring and `COMMIT`.

In the absence of locks or other synchronization mechanism, will the result necessarily correspond to some serial execution of the two transactions?

1. Yes.
2. No, since the execution might result in x = 3, y = 3.
3. (3.) No, since the execution might result in x = 1, y = 3.
4. No, since the execution might result in x = 1, y = 2.

C) [5 marks] You are considering using locks, and automatically adding code to each transaction to guarantee before-or-after atomicity. You would like to maximize concurrency, without introducing deadlocks. For each of the following proposals, fill the appropriate circle below if the approach a) yields semantics consistent with before-or-after atomicity (fill Ⓐ), and b) introduces potential deadlocks (fill Ⓑ).

1. A single, global lock which is ACQUIREd at the start of each transaction and RELEASEd at COMMIT. (Ⓐ) Ⓑ
2. A lock for each variable. Every READ or WRITE operation is immediately surrounded by an ACQUIRE and RELEASE of that variable's lock. Ⓐ Ⓑ

3. A lock for each variable that a transaction READS or WRITEs, acquired immediately prior to the first reference to that variable in the transaction; all locks are released at COMMIT. Ⓐ Ⓑ
4. A lock for each variable that a transaction READS or WRITEs, acquired in alphabetical order, immediately following the BEGIN. All locks are released at COMMIT. Ⓐ Ⓑ
5. A lock for each variable a transaction WRITEs, acquired, in alphabetical order, immediately following the BEGIN. All locks are released at COMMIT. Ⓐ Ⓑ

D) [2 marks] You would like to avoid having to read the entire log during crash recovery. You propose periodically adding a CHECKPOINT entry to the log, reading the log backwards from the end, and restoring committed values to your file system in memory. The backwards scan should end as soon as committed values have been restored for all the variables. Each CHECKPOINT entry in the log contains current values of all variables and a list of uncommitted transactions at the time of the CHECKPOINT. What portion of the log must be read to properly restore values committed at the time of the crash?

1. All of the tape; checkpoints don't help.
2. Enough to include the STARTED record from each transaction that was uncommitted at the time of the crash.
3. Enough to include the last CHECKPOINT, as well as the STARTED record from each transaction that was uncommitted at the time of the crash.
4. Enough to include the last CHECKPOINT, as well as the STARTED record from each transaction that was uncommitted at the time of the last checkpoint.

E) [4 marks] Since the log stores all values, you decide to simplify your implementation even further by completely removing the in-memory file system implementation and thus any need for in-memory locks. Instead, the `READ(tid, var)` primitive returns an appropriate value of `var` by simply examining the log. You implement `READ` so that each transaction A "sees" the values of global variables at the time of `BEGIN(A)`, as well as any changes made within A.

Suppose that a transaction only reads variables and does not write any variable. The new `READ` scheme:

1. Provides atomicity and no deadlocks
2. Provides atomicity, but deadlocks are possible
3. Does not provide atomicity, but no deadlocks are possible
4. Doesn't work at all even with no concurrency

Suppose that a transaction reads and writes variables. To ensure atomicity, one would need to modify:

1. None of the other operations.
2. The WRITE operation.
3. The COMMIT operation.
4. The WRITE and COMMIT operations.
5. The COMMIT and ABORT operations.

Question 9. File Systems [9 MARKS]

Consider a Unix-like file system that maintains a unique index node (inode) for each file in the system. Each index node includes 6 direct pointers, a single indirect pointer, and a double indirect pointer. The file system block size is **B** bytes, and a block pointer occupies **P** bytes.

Part (a) [3 MARKS] Write an expression for the maximum file size that can be supported by this index node, in terms of **B** and **P**.

6 direct data blocks, each of size B -> 6B
1 indirect block of size B, holding B/P pointers to data blocks of size B -> B*B/P
1 double indirect block of size B holding B/P pointers to indirect blocks, each with B/P pointers to data blocks of size B
max file size = 6B + B*B/P+B*B*B/P

Part (b) [3 MARKS] How many disk operations will be required if a process reads data from the N^{th} block of a file? Assume that the file is already open, the buffer cache is empty, and each disk operation reads a single file block. Your answer should be given in terms of **N**, **B**, and **P**. *Hint: Your answer can include multiple expressions to cover different values of N.*

#reads =
1 if N <= 6
2 if 6 < N <= 6 + B/P
3 if N > 6 + B/P

Part (c) [3 MARKS] In Unix systems, you are not allowed to create a hard link where the source file is on one file system, and the destination file is on a different file system. Give two specific reasons why this is not allowed. Would your explanation change if the second file system is mounted within the first file system.

A hard link records the inode # of the file being linked to. But each file system has its own inodes (and hence inode numbers), so the inode number alone is not sufficient to identify the file that is being linked to. For example, inode 42 on filesystem1 could be a different file than inode 42 on filesystem 2. No difference if filesystem is mounted or not.

Question 7. File Systems [8 MARKS]

For each of the following techniques, briefly explain how it is intended to improve I/O performance, and then describe one case where the technique can be expected to have little or no effect on performance.

Part (a) [2 MARKS] Disk scheduling

Disk scheduling reduces seek time, which is a dominant factor when accessing disks, by using algorithms such as SCAN (or elevator).

If disk requests arrive in SCAN order then disk scheduling will not improve performance. Similarly, disk scheduling will not improve the performance of a sequential scan of the disk.

Part (b) [2 MARKS] File system block caching

File system blocks are cached in memory and hence reading or writing these blocks more than once can be performed at memory speed rather than disk speed.

If file system blocks are accessed (read or written) once, e.g., playing a large media file, then caching these blocks will not improve performance.

Part (c) [2 MARKS] File block placement (allocation)

File blocks are placed sequentially, improving sequential read and write performance.

Sequential placement will not improve the performance of random access.

Part (d) [2 MARKS] Increasing the file-system block size

If the average size of a file is greater than the file-system block size, then increasing the block size will improve the performance of accessing the whole file. Also, it reduces the amount of file-system metadata (e.g., indirect blocks) that needs to be stored and accessed.

If the file size is smaller than the block size, then increasing the block size will not improve performance for accessing this file.

Question 8. File System Design [12 MARKS]

The Unix file system uses several types of indirect blocks (e.g., singly indirect, doubly indirect, etc.) for storing metadata information for large files. Thus, accessing a large file requires accessing several additional indirect blocks.

You decide to implement an extent-based file system, *efs*, to reduce metadata storage and access overhead for large files. An extent consists of a contiguous sequence of sectors on disk. Your file system uses the following representation for an inode:

```
#define NEXTENT 12

struct efs_inode {
        uint64_t filesize;
        uint16_t type;
        uint16_t nlink;
        uint32_t extent_size;
        uint32_t extents[NEXTENT];
}
```

The `extent_size` is a 32-bit unsigned number that indicates the size of each extent, in terms of the number of consecutive 512-byte sectors. All extents of a file are the same size. The `extents` array holds the starting disk sector number for each extent.

Files are created with an initial `extent_size` of 1 (that is, each extent is a single 512-byte sector). Whenever the file grows larger than the maximum file size supported by the current extent-size, the extent-size is doubled.

Part (a) [3 MARKS] What is the maximum file size supported by efs? Express your answer as $n * 2^m$ and explain your reasoning.

The maximum file size is $12 * 2^9 * extent\_size$. The efs file system stores 12 extents. Each extent has a size of $512 * extent\_size$. Since the maximum value of extent_size is 2^{32}, the maximum file size is $12 * 2^{41}$.

Part (b) [3 MARKS] Assuming that the inode is already in memory, how many disk accesses are required to read the byte at offset 1,000,000 in a file? *Briefly* explain your answer.

One disk access. The file system uses simple calculations based on the extent locations stored in the `extents` array and the extent size to find the block associated with the byte at offset 1,000,000 (or any offset) in the file.

Part (c) [4 MARKS] What operations are required when the file grows larger than the maximum file size supported by the current extent-size (in addition to doubling the extent-size field in the inode)?

All current extents have to be doubled. So contiguous extents that are twice as large will need to be allocated, the data copied from the old extents to the new extents, and the old extents need to be deallocated.

Part (d) [2 MARKS] Describe two drawbacks of the efs design over a traditional Unix file system.

1. Extent resizing for a file is very expensive, as described above.
2. Allocations of large contiguous extents will cause more severe external fragmentation.

Question 9. Journaling File System [6 MARKS]

Your friend has designed a new file system called the Confused Journaling File System (CJFS). CJFS tries to keep all of its meta-data consistent through the use of journaling. CJFS updates the file system when an application has appended a single block to a file as follows:

1 - write begin transaction to journal
2 - write journal descriptor to journal (describing the update)
3 - write data bitmap, inode, data block to journal
4 - write end transaction to journal
5 - write data bitmap, inode, data block to their final in-place locations
6 - free transaction in journal

However, CJFS has to wait for various I/Os to complete in order to work correctly. For example, if CJFS issues I/Os 1 and 2, waits for them to complete, and then issues I/O 3, it knows that I/Os 1 and 2 have completed before 3.

The following are some suggestions as to where to place such waits. Your job is to figure out whether the suggested placements work. Circle whether CJFS is either too slow (because it spends too much time waiting for I/Os to complete), broken (because it doesn't actually guarantee that meta-data is kept consistent), or just fine, for the following options:

(a) Wait between 3 and 4:	Too slow	Broken	Just Fine
(b) Wait between 3 and 4, Wait between 5 and 6:	Too slow	Broken	Just Fine
(c) Wait between 3 and 4, Wait between 4 and 5, Wait between 5 and 6:	Too slow	Broken	Just Fine
(d) Wait between 1 and 2, Wait between 3 and 4, Wait between 4 and 5, Wait between 5 and 6:	Too slow	Broken	Just Fine
(e) Wait between 1 and 2, Wait between 5 and 6:	Too slow	Broken	Just Fine
(f) Wait between 3 and 4, Wait between 4 and 5	Too slow	Broken	Just Fine

1. : Wait between 3-4: CJFS needs to wait before issuing end transaction (commit) so that all previous blocks are on disk. Otherwise, the commit block may reach before the file system updates are in the journal.
2. Wait between 4-5: CJFS needs to ensure that the commit block is on disk before writing the blocks to their final locations. This is required for write-ahead-logging.
3. Wait between 5-6: Before the transaction is freed from the journal, all the blocks need to be written to their final locations.

Question 8. Feature Full File System (FFFS) [20 MARKS]

You have just joined a storage startup that has designed a high-end, one petabyte (1024 TB = 2^{50} bytes) storage array for a modern data center. You are in charge of designing a modern, feature-full file system (FFFS) for this storage system. To do so, you start with the traditional Unix file system (UFS) and address many of its limitations.

Part (a) [3 MARKS] Assume the standard UFS supports an 8KB block size, and an inode has 12 direct pointers, one single indirect block pointer, and one double indirect block pointer. Assume also that each pointer is 32 bits in size. What is the approximate maximum file size supported in UFS? State your number in the most appropriate human readable unit (e.g., MB, TB).

Block size = 8 KB.
Nr of pointers in each block = 2^13 / 4 (bytes per pointer) = 2^11 = 2048.
Maximum file size = 8 KB * (12 + 2048 + 2048 * 2048)
Approximately, 8 KB * 2048 * 2048 = 2^13 * 2^11 * 2^11 = 2^35 = 2^5 GB = 32 GB.

Part (b) [3 MARKS] In FFFS, we want to support files that can be as large as 128 TB. What would be the minimal changes you would make to UFS to support these huge files.

128 TB = 2^7 * 2^40 = 2^47
Unfortunately, since the pointer size is 32 bits, and block size is 13 bits, the maximum file size that could be supported with 8 KB blocks is 2^45. So first, we need to increase the block size to 15 bits (32 KB).

Now the maximum file size that can be supported is 32 KB * (2048 * 4) * (2048 * 4) = 2^15 * 2^13 * 2^13 = 2^41.

This is insufficient, so we also need to add a triple indirect block pointer.

Some answers mentioned adding two triple indirect block pointers. We accepted that answer although it doesn't solve the problem associated with the limits of the pointer size * block size.

Part (c) [2 MARKS] In UFS, the number of inodes that can be allocated is fixed when the file system is first created. Describe why it is fixed (i.e., describe how the allocation/deallocation of inodes is managed).

Inodes are allocated from a fixed size linear inode table (an array). As a result, the inode table size cannot be increased after the file system is created.

Part (d) [6 MARKS] In FFFS, we want to support growing the number of inodes over time. Describe the changes you would make to UFS to implement this feature, as precisely as possible.

We could create a multi-level inode table similar to page tables or an inode pointing to blocks using a multi-level scheme. A multi-level inode table can be implemented in may ways. One way would be to have all inode blocks be allocated from the dynamically allocated blocks. A pointer in the super block would point to a single block, which is the first level inode table. The first-level inode table would point to multiple second-level inode blocks, that would be allocated from the dynamically allocated region as more inodes are needed. This scheme can be extended to an arbitrary level inode table.

Some answers mentioned that the inode table should be made really big. That was not an acceptable answer.

Some answers mentioned that new inode tables could be allocated dynamically and linked together in a linked list manner. We accepted that answer although allocating an inode table dynamically may not work if the file system is fragmented.

Part (e) [6 MARKS] In UFS, the file system is created in a fixed size partition or disk. In FFFS, we would like to add new 1 PB storage arrays to the file system dynamically (e.g., starting with a 1 PB file system, grow it to a 8 PB file system over time). Describe the changes you would make to UFS to implement this feature, as precisely as possible.

One way to allow adding disks to UFS while making minimal changes to UFS is to assume that each disk has the same format as UFS. Files can be stored on one disk only (otherwise, block pointers would have to contain a tuple consisting of disk#, block #).

There would be one change required to the disk format: the super block would contain a pointer to the super block on the next disk in the chain.

inode numbers in directory entries could refer to the inode tables on other disks (there would be no change to the inode number format). By caching the super blocks of all the disks, it would be easy to convert an inode number to the inode table on the appropriate disk.

Question 7. Sparse Files [8 MARKS]

A file system stores files that are mostly empty as *sparse files* on disk. For example, if the length of a file is 1 MB, but the file only contains data in its first and last blocks, then the file system only stores these two data blocks. When an application reads a sparse file, the file system converts the empty blocks into ”real” blocks filled with zero bytes, without the application being aware of this conversion. Answer the following questions.

Part (a) [1 MARK] Briefly explain how sparse files might be implemented in a Unix file system.

An easy way to implement sparse files is to ensure that a block pointer is null/zero for an empty block (block is not allocated).

Part (b) [3 MARKS] Say you wanted to upload the sparse files on your local disk to a file hosting service (such as Dropbox), and the file service supports sparse files as well. The problem will applications being unaware of sparse files is that your file synchronization client needs to read the entire sparse file, to determine whether any block contains all zeroes, before sending the sparse file to the file service. Show the function prototype of a new system call that you would implement that will not require reading the entire sparse file but just the blocks that are non-empty. How is the system call used? Are there any restrictions or limitations on the usage of this system call?

```
char buffer[4096];
int offset, next_offset;
loop:
  error = read_sparse(fd, offset, buffer, &next_offset);
  offset = next_offset;
```

In this interface, reads must be at block granularity. offset is the block or byte offset in the file at which to read a block. If the block at that offset is empty, the call should return an error. next_offset returns the next offset in the file at which a block is allocated. The main limitation of this interface is that the application must know about the file system block size (e.g., 4096 bytes).

This question continues on the following page.

Part (c) [2 MARKS] Briefly describe how a file system would implement your new system call.

read_sparse would be implemented similar to read. the main difference is that the next_offset would need to be returned. This will require traversing the file metadata blocks, e.g., inode, indirect block, etc., until the next allocated block is found (non-null block pointer).

Part (d) [2 MARKS] State two reasons why your file synchronization client will be more efficient when it uses your new system call.

1) The client will issue fewer read_sparse system calls than read system calls
2) The client does not need to parse a block and see if it has all zeroes to detect empty blocks.

Question 8. File System Extents [7 MARKS]

In an extent-based file system, storage is allocated in variable-sized, contiguous disk sectors called *extents*. Assume that extents are sized in multiples of blocks (e.g., 1 block, 13 blocks, etc.). In a Unix-style file system, suppose the only change you make to the on-disk format is that all block pointers are replaced with extent pointers. Each extent pointer is 8 bytes long, and contains a block location on disk, and the length of the extent in blocks. Answer the following questions. If you make any assumption, state them.

Part (a) [3 MARKS] State three advantages of using an extent-based file system design compared to a block based design.

1) a large file that is allocated sequentially will need to store data for one extent only, so there is lower metadata storage overhead.
2) there will be fewer accesses for metadata blocks for the same reason as above, improving performance.
3) files could be preallocated large extents, allowing for file growth without fragmentation.

Part (b) [3 MARKS] How would you change the block-based Unix file system code so that the extent-based file system will generally perform much better than the block-based design. Clearly explain your answer.

The file system could preallocate large extents when files are created, allowing for file growth. This may however lead to internal fragmentation.

Another option would be for the file system to periodically copy one or more small logically contiguous extents into a large single extent.

Part (c) [1 MARK] Give two examples where the extent-based file system design would be less efficient than a block-based design. Explain why.

The inode size is much bigger (close to twice the size), leading to performance overhead, because more inode blocks have to be read (to read the same number of files).
If each extent is a single block, the file metadata is doubled (8 bytes per extent, versus 4 bytes per block), leading to more storage/performance overhead, and smaller maximum file size.

Question 9. File System Consistency [10 MARKS]

Part (a) [4 MARKS] **Performance Optimization:** Suppose that you decide to improve the performance of the Berkeley fast file system (FFS) by removing all synchronous writes to disk. In your modified file system, all writes to disk are delayed 30 seconds (or until the kernel evicts blocks from the buffer cache). This allows two successive writes to the same inode, directory block, or other metadata structures to only require a single write to disk. Moreover, your kernel uses a disk scheduler that orders the writes to minimize disk seeks. Describe a way in which, after a power failure, a directory block may contain, instead of directory entries, data blocks from an existing user file. Make sure to show the sequence of system calls made before the power failure that cause this problem.

1. delete file f

2. create new directory d

say the kernel reuses a block that used to belong to f to store the directory data for d

3. crash

if all the metadata indicating the f has been deleted (inode bitmap block for f, inode of f, block containing directory entry for f, block bitmap for all blocks of f that are deallocated) did not reach disk before the crash, while all the metadata blocks indicating that d is created (inode bitmap block for d, inode of d, block containing directory entry for d) did reach disk before crash, then both file f and directory d will point to the same blocks

This question continues
on the following page.

Part (b) [6 MARKS] **Journaling:** Recall that journaling file systems are designed to speed up crash recovery. Explain clearly how a journaling file system would avoid or recover from the inconsistency caused by the scenario that you have outlined above. To do so, make sure to describe the types of blocks that are written, allocated or deallocated, and show the precise order of operations using a simple diagram. You do not need to describe any blocks in your scenario that are not relevant to crash recovery. Note that your journaling file system only journals metadata blocks. It may help to write your answer as a set of bullet points.

The reason for the inconsistency is that the file f's deletion does not reach disk before directory d's creation reached disk. We can avoid this inconsistency by using journaling. Two options:

1. Journal "delete f, mkdir d" in one transaction, where all the metadata updates for each operation (described above) are logged to the journal. In this case, write-ahead logging will ensure that either both the operations are performed, or neither will be performed, avoiding any inconsistency.

2. Journal "delete f" in one transaction, "mkdir d" in next transaction. In this case, write-ahead logging will ensure one of the following: 1) none of them are performed, 2) "delete f" is performed, but "mkdir d" is not, 3) both operations are performed. In all cases, there is no inconsistency.

Question 8. Inode Lost [8 MARKS]

In this question, we consider a variant of the Unix file system, which instead of using inodes, instead stores relevant information about a file in the directory entry for that file, so no inode table exists in the file system. We call our new system the Inode-Lost File System or ILFS.

Part (a) [2 MARKS] Which of the following are found in a standard Unix inode? Circle all that apply:

1) Direct pointers to data blocks
2) The name of the file
3) Some statistics about when the file has been accessed, updated, etc.
4) The inode number of the file's parent directory
5) The current position of the file pointer

Part (b) [2 MARKS] In a standard Unix file system, what is the minimum number of disk reads it will take to read a single block from the file `/this/path/is/tooshort` from disk? Other than the superblock, you should assume nothing is cached, i.e., everything starts on disk. Indicate the number of reads of each type of block (e.g., # of directory blocks, etc.).

4 inode blocks, 3 directory blocks, 1 data block

Part (c) [2 MARKS] Now compare this to the minimum number of reads it would take to read a single block from the same file `/this/path/is/tooshort` in ILFS. Again, other than the superblock, assume nothing is cached, i.e., everything starts on disk. Indicate the number of reads of each type of block.

3 directory blocks, 1 data block

Part (d) [2 MARKS] Why do you think ILFS does not support hard links?

The inode allows the Unix file system to keep a reference count of the number of directory entries that point to the inode. This reference count is used to ensure that file contents are deleted only when the last name of the file is deleted. Without the inode, there is no area in which ILFS can keep the reference count, so it cannot support hard links.

Part (e) [2 MARKS] Say the contents of a directory are corrupted due to some errors on the disk. In which file system, ILFS or the Unix file system, would it be *easier* to recover the *contents* of the files in that directory? Why?

It would be easier in the Unix file system because the inodes are located in a well-known location on disk, and each inode helps locate the contents of a unique file or directory. In the Unix file system, a file system recovery program could traverse the file system and find out which inodes are not referenced by any directories. These are the lost files. In ILFS, the directory is corrupted, so we wouldn't know where to start looking for the contents of files.

Question 9. 2^{Files} [8 MARKS]

In a Unix-like file system called 4FS, the block size is 4K, and the block numbers are stored as 4 byte integers in the inode and in the indirect blocks.

Part (a) [2 MARKS] With three levels of indirect blocks, what is the maximum size of a file that can be supported in 4FS. Circle the closest answer below. Show your calculations.

1) 1 GB (2^{30} bytes) 2) 4 GB 3) 1 TB (2^{40} bytes) 4) 4 TB 5) 16 TB

Each block contains 2^10 pointers (4KB/4B). The third-level block allows accessing (2^10)^3 = 2^30 blocks. Each block is 2^12 bytes. So we can access 2^42 bytes = 4TB.

Part (b) [2 MARKS] Suggest a way by which you can increase the maximum file size without changing the number of levels of indirect blocks.

increase block size, e.g., 8KB, which increases number of pointers and block size.

Part (c) [2 MARKS] Now let's go back to the original 4FS file system. Say we store block numbers as 8 byte integers. Would that increase or decrease the maximum file size? Show your calculation.

Decreases maximum file size. Each block will contain 2^9 pointers. So maximum file size is 2^9^3 * 2^12 = 2^39.

Part (d) [2 MARKS] Now let's go back to the original 4FS file system. Say we use four levels of indirect blocks. What is the maximum size of a file that can be supported. Circle the closest answer below. Show your calculations.

1) 1 TB (2^{40} bytes) 2) 4 TB 3) 16 TB 4) 1 PB (2^{50} bytes) 5) 4 PB

This is a little tricky. Four levels of indirect pointers allows 2^40 blocks but the block numbers are stored in 4 bytes, so we cannot address more than 2^32 blocks. Hence, the maximum file size would be limited to 2^32 * block_size = 2^32 * 2^12 = 2^44 = 16TB. Using four levels of indirect pointers fully requires using 8 byte block numbers.

Question 10. The Lies That Editors Tell [14 MARKS]

A modern, text editor claims that it updates files atomically, so that on a system crash, the file contents will either be the old value of the file, or the new value of the file, but not some garbled combination of the old and the new value.

You look at the following editor code for updating a File `f`. Note that `f.new` is a file with the string `''.new''` appended to the name of File `f`. It looks like the editor is using redo recovery. We will ignore any errors returned by these calls.

```
   update_file(File f)
   {
       char buffer[file_size];
1:     read(f, buffer);          // read File ''f'' into a buffer in memory
2:     update_buf(buffer);       // update buffer in memory
3:     write(f.new, buffer);     // write buffer into new File ''f.new''
4:     create(f.done);           // create a new file ''f.done''
5:     copy(f.new, f);           // copy the new version to File ''f''
6:     remove(f.done);           // remove the ''f.done'' file if it exists
7:     remove(f.new);            // remove the ''f.new'' file if it exists
   }
```

Part (a) [6 MARKS] When the text editor opens File `f` after a crash, it needs to run recovery code to ensure that the file is updated atomically. Unfortunately, there is no such code implemented in the editor. Please implement it. You may assume that a function called `exists(f)` is defined (in the editor code) that returns TRUE if File `f` exists in the file system. Hint: Be as brief as possible. It may help to read the question on the next page before writing your answer.

```
   recover_file(File f)
   {
        if (exists(f.done)) { /* do every thing in update_file after Line 5 */
           copy(f.new, f);
           remove(f.done);
        }
        remove(f.new); /* remove f.new if it exists */




   }
```

This question continues
on the following page.

Question 10. The Lies That Disks and Their Drivers Tell (CONTINUED)

For your convenience, we have listed the `update_file` code again.

```
   update_file(File f)
   {
       char buffer[file_size];
1:     read(f, buffer);          // read File ''f'' into a buffer in memory
2:     update_buf(buffer);       // update buffer in memory
3:     write(f.new, buffer);     // write buffer into new File ''f.new''
4:     create(f.done);           // create a new file ''f.done''
5:     copy(f.new, f);           // copy the new version to File ''f''
6:     remove(f.done);           // remove the ''f.done'' file if it exists
7:     remove(f.new);            // remove the ''f.new'' file if it exists
   }
```

Part (b) [8 MARKS] The code above appears to work but it doesn't have any disk flush commands. Recall that modern disks lie to improve performance. They use a DRAM disk buffer, and when a write is issued, they simply copy the data to the disk buffer, and claim the write is completed. Then, they copy the data in the disk buffer to the disk platters asynchronously (sometime later). The problem is that on a crash, the data in the disk buffer that has not been copied to the disk platters can be lost.

Fortunately, disks provide a flush command that forces all data in the disk buffer to be written durably to the disk platters. In the `update_file` code, *under* which lines would you place the disk flush command to ensure that a file is updated atomically? For example, if you say "Line 1", then you would add a flush between Lines 1 and 2 in the code above. Write the line numbers in the table below. Since disk flushes are expensive, use the minimum number of flush commands that ensure a correct atomic update. With each line, state one of the two problems that could happen if the flush was not issued at that point: 1) *before crash*: File `f` is written partially before a crash occurs and no recovery is performed, or 2) *after recovery*: File `f` is written partially when recovery is performed (after the crash). Hint: consider how your recovery code works.

Line Number	Before crash/After recovery
3	after recovery
4	before crash
5	before crash
6	after recovery

3: If f.new is not fully on disk but f.done is created on disk in Line 5, then recovery will copy a garbage version of f.new to f.

4: If f.done is not on disk but the copy operation in Line 6 has partially succeeded, then we will not perform the copy operation during recovery, and f could have partial contents of f.new.

5: If the copy operation is not complete but f.done is removed in Line 7, then similar to the above reasoning, we will not perform the copy operation during recovery, and f could have partial contents of f.new.

6: If f.done is not removed from disk, but f.new is removed in Line 8, then recovery is not directly affected for a single write to f. However, consider a second update to file f (no crash yet). Say a crash occurs after a partial write to f.new in Line 4. On recovery, since f.done exists, we would copy the partially written f.new to f.