Started on	Wednesday, 17 March 2021, 11:49 AM PDT
State	Finished
Completed on	Wednesday, 17 March 2021, 2:48 PM PDT
Time taken	2 hours 59 mins
Grade	**99.00** out of 100.00

Question **1**

Complete

15.00 points out of 15.00

Assume your machine uses a disk specified below:

- **# of surfaces: 32**
- **# of sectors for each cylinder: 8192**
- **# of tracks for each surface: 2048**
- **# of bytes for each sector: 1024**

(a) How many platters does this disk have? (3 points)

(b) How many sectors per track? (6 points)

(c) What is the total size of this disk? Please show your reasoning. (6 points)

(a) There are 32 surfaces and 2 surfaces per platter so the total is 16 platters.

(b) There are 8192 sectors per cylinder. There are 32 surfaces per cylinder as well, but a cylinder is very thin and only encompasses 1 track per surface. Thus, there are 32 tracks per cylinder. Then, we can see that there are 8192/32 = 256 sectors per track.

(c) There are 1024 bytes per sector, and from part (b), 256 sectors per track. There are also 2048 tracks per surface and 32 surfaces. Thus, there are 1024 * 256 * 2048 bytes per surface, and 32 surfaces, for a total of 2^34 bytes. This is 16GB.

Question **2**

Complete

15.00 points out of 15.00

We use a file system where each i-node contains 10 direct pointers, 1 indirect pointer, 1 double-indirect pointer, and 1 triple-indirect pointer. The size of each disk block is 4K bytes and each pointer to a disk block also requires 4 bytes.

(a) What is the largest file that can be supported? Show your calculation steps (i.e., you don't need to calculate the final numeric answer!) (8 points)

(b) Assume that the OS has already read the i-node for your file into memory (i.e., the file buffer cache). How many disk reads are required to read data block number 800 into memory? Please describe your reasoning. (7 points)

(a) 10 direct pointers, 1 indirect pointer, 1 double-indirect pointer, 1 triple-indirect pointer.

10 direct pointers all point to disk blocks of size 4K bytes for a total of 10 * 4K bytes

1 indirect pointer points to a disk block of size 4K that stores pointers of size 4 bytes. Thus, there are 4K / 4 = 1K = 1024 pointers on that indirect block. each of those points to a disk block of size 4K bytes for a total of 1024 * 4K bytes.

1 double-indirect pointer points to a disk block of size 4K that stores pointers of size 4 bytes, leading to 1024 pointers on that double-indirect block. Each of those points to an indirect block that can contain 1024 * 4K bytes. So the total storage is 1024 * 1024 * 4K bytes

1 triple-indirect pointer points to a disk block of size 4K that stores pointers of size 4 bytes, leading to 1024 pointers on that triple-indirect block. Each of those points to a double-indirect block that can contain 1024 * 1024 * 4K bytes. So the total storage is 1024 * 1024 * 1024 * 4K bytes.

The total storage is therefore 10 * 4K + 1024 * 4K + 1024^2 * 4K + 1024^3 * 4K. (which is about 4TB I think?)

Answer: 10 * 4K + 1024 * 4K + 1024^2 * 4K + 1024^3 * 4K bytes

(b) If we have already read the inode for the file into memory, there is no need to read it again. The problem comes down to finding whether block number 800 lies in direct, indirect, double-indirect, or triple-indirect. We know that there are 10 direct pointers, and 1024 blocks pointed to by the single-indirect block, so block 800 must fall into single-indirect. This takes a total of 2 disk accesses. One to the indirect block, and one to the data block itself. (The inode has already been read).

Answer: 2 disk accesses

Answer: 2 disk accesses.

Question 3

Complete

15.00 points out of 15.00

Consider the following program that uses a set of queues and moves an item from a queue (i.e., "source") to another queue (i.e., "destination"). Each queue can be both a source and a destination.

Assume there are multiple threads that call AtomicMoveBetweenQueues() concurrently.

```
========================================
void AtomicMoveBetweenQueues (Queue *source, Queue *destination) {
  Item thing; /* thing being transferred */
  if (source == destination) {
    return; // same queue; nothing to move
  }
  source->lock.Acquire();
  destination->lock.Acquire();
  thing = source->Dequeue();
  if (thing != NULL) {
    destination->Enqueue(thing);
  }
  destination->lock.Release();
  source->lock.Release();
}
========================================
```

(a) Show a scenario with no more than three queues where AtomicMoveBetweenQueues () does not work correctly. (3 points)

(b) Modify the function to work correctly. (6 points)

(c) If a queue can be either a source or a destination, but not both, is the function working correctly? Why, or why not. If not, describe a scenario where the function (given at point (a)) does not work correctly. (6 points)

(a) Scenario: two threads one running AtomicMoveBetweenQueues(A, B) and another running AtomicMoveBetweenQueues(B, A). In this case, there will be a deadlock for the following reason: Assume the first thread runs first. It will try to acquire a lock for source, so basically a lock for queue A. Then, pretend that the second thread starts running now, and acquires a lock for its source, aka queue B. Now, when the first thread continues running, it can't acquire the lock for its destination, queue B, since the lock was already taken up by thread 2. So, it will wait there (or block). The second thread, however, ALSO can't run, because its destination, aka queue A, is also locked by thread 1. This is a deadlock and therefore AtomicMoveBetweenQueues() works incorrectly.

(b) The above example in part (a) showcases the example of a cycle. A, B vs B, A is a cycle, and when cycles exist, there can be a deadlock as described in class. Therefore, we need some way to prevent a cycle. In lecture, the professor talked about creating a lock ordering, to make sure that cycles don't exist. Pretend we made it so that A HAD to be locked before B, no matter if it was the source or destination, or what order it came in in the function arguments. Then, there would not be a deadlock because if thread 1 locks A, thread 2 came in in the function arguments. Then, there would not be a deadlock because if thread 1 locks A, thread 2 cannot lock B and cause a deadlock, since A has to come before B, and thread two will be prevented from locking A.

To create this sense of lock order, there are a couple things we can do. One is to somehow assign priorities to queues before we even begin moving items between queues (aka at the initialization stage of the queues), before we even begin calling AtomicMoveBetweenQueues().

Assigning priorities can be done by taking advantage of the data structure itself, (maybe setting the first element of each queue to signify the priority of it), or creating a data structure that maps every queue instantiated to a new priority number. The pseudocode for both cases is below:

Taking advantage of the data structure:

```
int priority = 0;
// before we fill in queue with any real data
for every queue created:
   queue.push(priority++)
```

and in AtomicMoveBetweenQueues, in place of

```
source->lock.Acquire();
destination->lock.Acquire();
```

we do:

```
if source.front() < destination.front() {
  source->lock.Acquire();
  destination->lock.Acquire();
} else {
  destination->lock.Acquire();
  source->lock.Acquire();
}
```

and

```
if source.front() < destination.front() {
  destination->lock.Release();
  source->lock.Release();
} else {
  source->lock.Release();
  destination->lock.Release();
}
```

However, this method takes up space in the queue. So instead, we can create a map from queue to priority number:

```
int priority = 0;
// before we fill in queue with any real data
for every queue created:
  priritymap[queue] = priority++
```

and in the function AtomicMoveBetweenQueues, we unlock and lock like this:

```
if prioritymap[source] < prioritymap[destination] {
  source->lock.Acquire();
  destination->lock.Acquire();
} else {
  destination->lock.Acquire();
  source->lock.Acquire();
}

if prioritymap[source] < prioritymap[destination] {
  destination->lock.Release();
  source->lock.Release();
} else {
  source->lock.Release();
  destination->lock.Release();
}
```

(c) I feel like if each queue can be only a source or a destination, and the type it is is determined beforehand during initialization, the function in part (a) would actually work. There won't be any deadlocks because a deadlock requires a cycle, that should look something like A, B; B, C; C, D; .... X, A. But A cannot be both source and destination. If there can't be a cycle, then one of the 4 conditions for a deadlock will not hold and there can't be a deadlock.

Question **4**

Complete

5.00 points out of 5.00

The BSD file system allocates a large file in a number of chunks. Each chunk is contiguously allocated but different chunks may be far apart. Assuming that a disk transfers at a peak rate of 200 MByte/s; a seek and rotation, combined, take a total of 20 milliseconds. In order to achieve 80% of peak transfer rate for large files when they are accessed sequentially, what is the minimum size of each chunk?

200 MB/s peak transfer rate

80% peak transfer rate = 0.8 * 200 = 160 MB/s transfer rate

transfer rate = total bytes transferred/amount of time needed.

assume x MB per chunk. assume large file needs y chunks. the total file is x * y MB.

now we calculate the total time needed to transfer the entire file:

all chunks require the same amount of time since they each require a seek+rotation, as well as a transfer time.

each chunk requires:

20 ms + x / 200 s. = x / 200 + 0.020 seconds

This is true for all y chunks, so the total time is xy/200 + 0.020y

Thus, x * y MB / (xy / 200 + 0.020y) seconds = x / (x / 200 + 0.020) >= 160. So, x >= 4/5 * x + 160 * 0.02. Thus, 1/5 * x >= 16/5 and x >= 16.

Each chunk has to be a minimum of 16 MB in order to get 80% peak transfer rate. As a check, let's plug in 10000 MB into the value of x for our equation: 10000 / (10000/200 + 0.020) = 200 MB/s, which makes sense as the peak transfer rate gets closer and closer to 200 as we increase the size of each chunk.

Answer = 16 MB.

Question 5

Complete

14.00 points out of 15.00

Now let's see just how large a file is handled by a set of file systems. Assume that the size of each block is 4 KBytes.

(a) Consider a simple filesystem, *simplefs*, where each inode only has 10 direct pointers, each of which can point to a single file block. Direct pointers are 32 bits in size (4 bytes). What is the maximum file size for *simplefs*? (3 points)

(b) Consider another fs *extentfs*, with a construct called an extent. Extents have a pointer (base address) and a length (in blocks). Assume the length field is 8 bits (1 byte). Assuming that an inode has exactly one extent. What is the maximum file size for *extentfs*? (3 points)

(c) Consider the third fs, *complexfs*, that uses direct pointers, but also adds indirect pointers and double-indirect pointers. Each inode within *complexfs* has 1 direct pointer, l indirect pointer, and 1 doubly-indirect pointer field. Pointers, as before, are 4 bytes (32 bits) in size. What is the maximum file size for *complexfs*? (3 points)

(d) Consider the fourth fs, *smallfs*, which saves as much space as possible within each inode. Thus, to point to files, it stores only one 32-bit pointer to the first block of the file. However, blocks within *smalltfs* store 4,092 bytes of user data and a 32-bit next field (like a linked list), and thus can point to a subsequent block (or to NULL). Draw a picture of an inode and the blocks of a file that is 10 KBytes in size, show how each part is linked together. You can draw on paper and upload a picture. (3 points)

(e) What is the maximum file size for *smallfs* (assuming no other restrictions)? (3 points)

(a) 10 direct pointers, each of which points to a single file block of size 4 KB. Thus, the max file size is 10 * 4KB = 40KB.

Note: File cannot be partially stored in inode, so final answer is still 40KB. (asked on piazza).

(b) Extents have a length field of 8 bits, so they can store up to 2^8 entries. Each entry can point to a block of size 4 KB, so the total size is 2^8 * 4KB = 1024KB = 1 MB.

(c) Calculation for this problem is very similar to calculation for problem 2. 1 direct pointer points to a single block of size 4 KB. 1 indirect pointer points to an indirect block that can hold 4 KB / 4 entries, or 1024 entries, each of which points to a single block of size 4 KB for a total space of 1024 * 4KB. 1 doubly-indirect pointer points to 1 doubly-indirect block which holds 1024 entries, each of which points to an indirect block that can hold a total space of 1024 * 4 KB, for a total storage space of 1024^2 * 4KB. Thus, the max file size for complexes is 4 KB + 1024 * 4 KB + 1024^2 * 4KB. This is about 2^32, or 4 GB of storage. Exact number is 4299165696 bytes.

(d) https://drive.google.com/file/d/1N6ti5YSZKuKT10SlMQELKyAtH6tkN8f5/view?usp=sharing

(e) Since the next field is 32 bits, there can be 2^32 possible next addresses aka 2^32 possible blocks. Each block is 4KB, so the max file size is 2^32 * 4KB. However, no useful data can be stored in the next pointer, so we have to subtract that amount of bytes, aka 2^32 * 4B. This comes out to 2^32 * (4KB - 4B), which is approximately 2^34 KB or 2^44 B = 16 TB, or exactly 16368 GB.

Question **6**

Complete

10.00 points out of 10.00

Consider four processes C1, C2, C3, and C4, and two resources, D1, and D2, respectively. Each resource has two instances. Furthermore:

- C1 allocates an instance of D2, and requests an instance of D1;
- C2 allocates an instance of D1, and doesn't need any other resource;
- C3 allocates an instance of D1 and requires an instance of D2;
- C4 allocates an instance of D2, and doesn't need any other resource.

(a) Draw the resource allocation graph; you can draw it on paper and upload a picture. (3 points)

(b) Is there a cycle in the graph? If yes name it. (2 points)

(c) Is the system in deadlock? If yes, explain why. If not, give a possible sequence of executions after which every process completes. (5 points)

(a) https://drive.google.com/file/d/1uNsmxRFoOmSFWBotaHwXhELhUpZCTsPt/view?usp=sharing

(b) Yes, there is a cycle: C1 waits for D1, D1 is held by C3, C3 is waiting for D2, and D2 is holding C1. The other two instances of D1 and D2 are being held by C2 and C4. The cycle is C1, D1, C3, D2.

(c) No, the system is not in deadlock. C2 and C4 are holding, and not waiting, so they can first run to completion.

Sequence of executions:

- C4 runs to completion, and releases D2
- C2 runs to completion and releases D1
- C1 can now get an instance of D1
- C1 runs to completion and releases an instance of both D1 and D2
- C3 can get an instance of D2
- C3 runs to completion and releases an instance of both D1 and D2
- all processes are completed and all resources have been freed.

Question **7**

Complete

10.00 points out of 10.00

Consider the following implementation of a reader writer lock. A reader writer lock allows either multiple readers to have access to a critical section or a single writer.

=====================================

```
struct rwlock
{
  sem_t *sem;
  int readers;
  int writers;
};
void rwlock_init(struct rwlock *lock)
{
   sem_init(&lock->sem, 1);
  lock->readers = 0; lock->writers = 0;
}
void readlock(struct rwlock *lock)
{
  while(1)
  {
    sem_wait(lock->sem);
    if(lock->writers == 0) { lock->readers++; break; }
    sem_post(lock->sem);
  }
}
void writelock(struct rwlock *lock)
{
   while(1)
   {
    sem_wait(lock->sem);
    if(lock->readers == 0 && lock->writers == 0) { lock->writers = 1; break; }
    sem_post(lock->sem);
   }
}
void unlock(struct rwlock *lock)
{
  sem_wait(lock->sem);
  if(lock->readers > 0) lock->readers--;
  else lock->writers--;
  sem_post(lock->sem);
}
====================================
```

(a) What is the problem with this implementation? (5 points)

(b) Starvation is a problem where one thread is unable to acquire a resource. After fixing the problem, is starvation possible? How? (5 points)

(a) The point of sem_wait and sem_post is to protect the critical sections in each function. However, in writelock and readlock, not all paths of execution are protected by both a wait and a post. For example, take a look at readlock. If a reader were to call readlock, it would first call sem_wait, which would decrement the semaphore. Then, since there are no writers yet, it will enter the if statement, and increase the readers to 1. Afterwards, it will break the while loop and exit the function without posting the semaphore. If another reader were to try and read, and call readlock, it would just get stuck at sem_wait forever since the semaphore has never been posted aka released.

This is a problem in both readlock and writelock. To fix this, we need to put a sem_post before the break statement inside the if statement for both writelock and readlock. This way, all code execution paths will be protected on both sides.

Something like this, except for both readlock and writelock.

```
void readlock(struct rwlock *lock)
{
  while(1)
  {
    sem_wait(lock->sem);
    if(lock->writers == 0) { lock->readers++; sem_post(lock->sem); break; }
    sem_post(lock->sem);
  }
}
```

(b) Starvation is when one thread is unable to acquire a resource. Assuming each thread is either a reader or a writer, starvation is definitely possible. If we have a pattern like RRRW, then the writers will actually never be able to write. Writers can only write when the number of readers is 0, so if there are a consistent nonzero number of readers at a time, a writer thread will never be able to write, and will be starved. Fixing the problem identified in A will fix all the major execution problems, but it will not prevent starvation.

Question **8**

Complete

15.00 points out of 15.00

As discussed in our lectures, creating a new file in a directory needs to update 4 blocks under Linux ext2: the inode bitmap, the file inode, the directory inode, and the directory's data block. Assume the directory inode and the file inode are in different on-disk blocks. (3 points for each question)

Part I: Assume we perform neither journaling nor FSCK. What would happen if a crash occurs after only updating the following block(s)? Choose from the following answers and explain the reasons:

- no inconsistency
- wasted data block/inode
- multiple file paths may point to the same inode
- point to garbage data
- multiple problems listed above.

(a) Bitmap

(b) Directory inode and Directory data

Part II:

Let us add a simple implementation of data (including data and metadata) journaling to our file system and perform the same file creation as Part I. Assume each transaction on the journal starts with a header block and finishes at a commit block. If the system crashes after the following number of blocks have been synchronously written to disk (including the journal and real FS), what state will the FS be in after the reboot? What can we do to recover?

(a) 1 disk write (hint: just the transaction header block is written to the journal)

(b) 8 disk writes (hint: transaction header, plus 4 blocks, plus commit block to journal, plus two data blocks to the real FS)

(c) 11 disk writes (hint: you also have to clear the transaction after writes to the real FS finish)

Part I:

(a) If a crash occurs only after updating bitmap, we are essentially turning the 0 bit for an inode to a 1, without actually creating the file inode or updating anything else. This will cause a wasted file inode, as future file creations will not be able to use that inode for a file. Instead, they will read that the inode is allocated (since the bit in the bitmap is set to 1), and move on to find a 0 bit (for a free inode to use). The reason the other two bullet points aren't correct is because there cannot be a pointer to garbage data, as we haven't created any pointers. There also cannot be multiple file paths pointing to the same inode because we haven't updated any file paths or directory contents to even create file paths. There isn't even a file yet!

Answer:

- wasted data block/inode

(b) If a crash occurs only after updating the directory inode and directory data, a couple of serious problems can occur. The directory inode contains metadata such as # of files in the directory, total size, date of last modification, etc. In addition, the directory inode contains pointers to all the file inodes since Unix inodes contain pointers. If we change this, and change the total size and # of files, but do not actually create the file inode or update the inode bitmap and make the actual file, the directory inode could have a pointer to garbage data.

In addition, what's dangerous is that the inode hasn't been created. Additionally, the inode bitmap hasn't even been updated! It is totally possible that a new file can correctly be created and use that inode as it sees that it is free. Now, we have multiple file paths pointing to the same file. :(

Answer:

- point to garbage data

- multiple file paths may point to the same inode

Part II.

How journaling works it that according to the textbook, there are 4 steps: Journal Write, Journal commit, Checkpoint, and Free. Journal write essentially writes the transaction header as well as the data blocks in question (4 blocks under Linux ext2). Then, the Journal commit writes the transaction commit block. Then, the checkpoint copies all the 4 data blocks into the file system at their final locations. Finally, the transaction is marked free in the journal after the checkpoint is complete.

(a) 1 disk write: just the transaction header block is written. we are still in the journal write stage, and the computer does not know where the crash happened in the midst of that stage, so the journal entry is deemed garbage and the action does not go through. We do not go through the recover process because there is no journal commit block. The FS will be in the exact same state as before

(b) 8 disk writes: in this case, 1 + 4 + 1 = 6. 8 > 6, so we are in the middle of the checkpoint stage. If the crash happens here, the FS data is wrong. It only has 2/4 of the correct blocks written to the FS. However, the journal has been committed, so to recover we rerun the action written out by the journal and re-attempt the checkpoint phase given the journal entry. If this succeeds, the FS will be in the correct state.

(c) 11 disk writes: in this case, 1 + 4 + 1 = 6, and 6 + 4 = 10. After 10 disk writes, we will have finished the checkpoint stage. However, we still need to free the journal entry. This can be as simple as moving the old/new pointers located in the journal superblock to the right place. This is assuming we use the circular log described in the textbook, where there are start and end pointers pointing to the start and end of the journal entries that are still active. To free an entry, all we do is move the start pointer up one entry. This will take 1 disk write, as we have to access the journal superblock.

NOTE: the hint is kind of confusing and it is entirely possible that the problem wants us to figure out what happens if it crashes before the free is complete (this could happen if the free step takes up more than 1 disk write, such as if the superblock AND the journal entry have to be written to for a total of 2 disk writes). In this case, if the free step is not complete, the journal has no way of knowing if the checkpoint phase has truly completed, in which it will have to recover and redo the entire journal entry action again upon boot up.

◀ Project 4C

Jump to...