Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2007

Quiz III Solutions

There are 19 questions and 23 pages in this quiz booklet. Answer each question according to the instructions given. You have **90 minutes** to answer the questions.

Most questions are multiple-choice questions. Next to each choice, circle the word **True** or **False**, as appropriate. A correct choice will earn positive points, a wrong choice may earn negative points, and not picking a choice will score 0. The exact number of positive and negative points for each choice in a question depends on the question and choice. The maximum score for each question is given near each question; the minimum for each question is 0. Some questions are harder than others and some questions earn more points than others—you may want to skim all questions before starting.

If you find a question ambiguous, be sure to write down any assumptions you make. **Be neat and legible.** If we can't understand your answer, we can't give you credit!

Write your name in the space below AND at the bottom of each page of this booklet.

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
NO PHONES, NO COMPUTERS, NO LAPTOPS, NO PDAS, ETC.

CIRCLE your recitation section number:

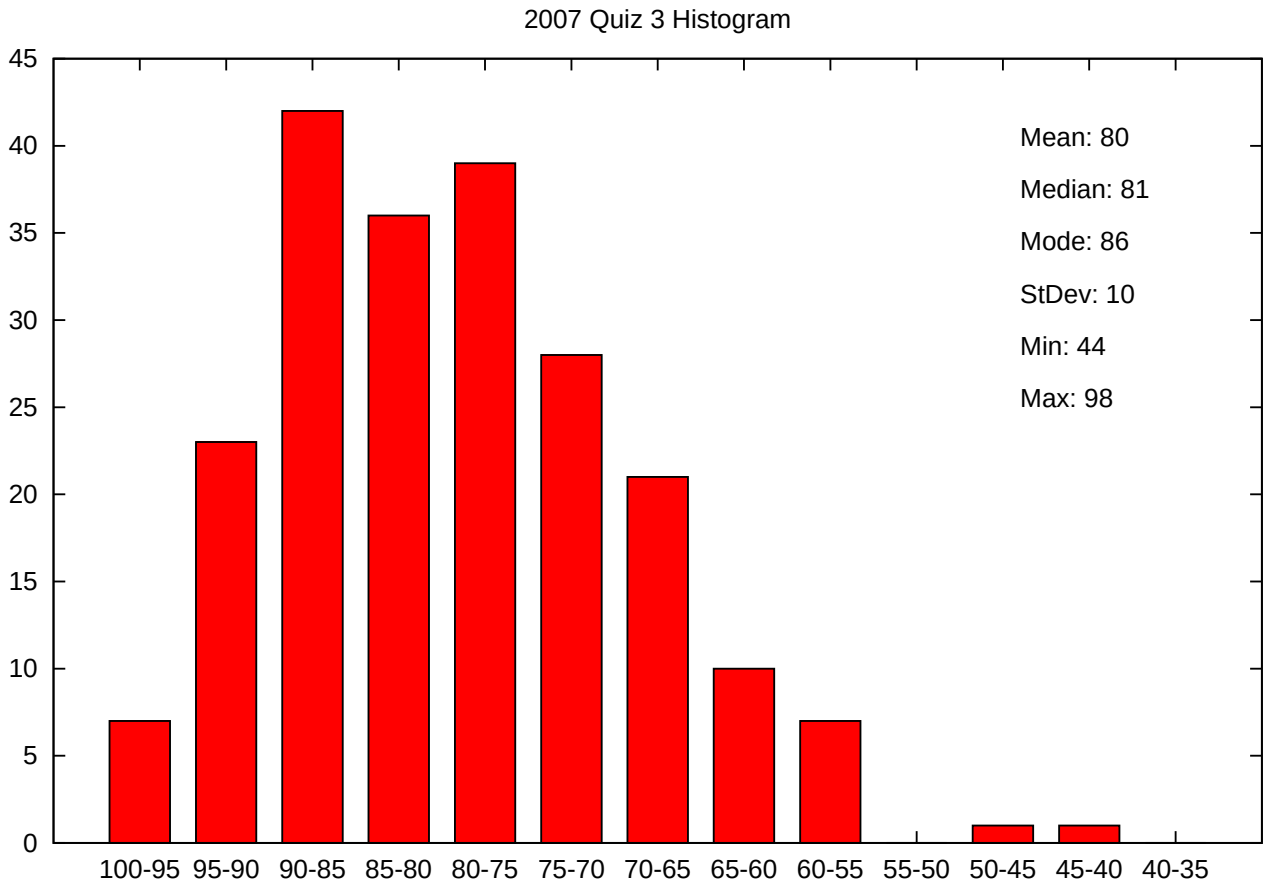
10:00	1. Madden/Komal		
11:00	2. Madden/Zhang	3. Katabi/Komal	10. Yip/Chachulski
12:00	4. Yip/Zhang	5. Katabi/Chachulski	
1:00	6. Ward/Shih	7. Girod/Schultz	
2:00	8. Ward/Schultz	9. Girod/Shih	

Do not write in the boxes below

1-5 (xx/21)	6-10 (xx/34)	11-15 (xx/25)	16-19 (xx/20)	Total (xx/100)

Name:

I Quiz 3 Statistics



Name:

II Short Reading Questions

II.1 Buffer overruns

1. [4 points]: For each of the following approaches, indicate whether it will completely eliminate buffer-overrun vulnerabilities in a user's program:

(Circle True or False for each choice.)

(a) **True / False** Have the operating system mark stack memory pages as non-executable.
False. For example, a buffer overrun can still use techniques like arc injection to attack the system.

(b) **True / False** Have the operating system mark heap memory pages as non-executable.
False. For example, a buffer overrun can still use techniques like stack smashing to attack the system.

(c) **True / False** Save an extra copy of a function call's return address (e.g., on a separate call and return stack) and make sure the two copies match before jumping.
False. For example, a buffer overrun can still use function pointers to execute arbitrary code

(d) **True / False** Re-write all programs (including libraries) in Java (or another type safe language with array bounds checking).
True. Bounds-checking prevents buffer overruns in the user's program. There may still be vulnerabilities, but they aren't due to unintentional overwriting of data via overruning a buffer.

II.2 Subversion

Alice and Bob are collaborating on the Java code that implements their DP2 design. Using SVN (Subversion), they maintain the following set of files that compile into a working implementation of DP2.

```
trunk/
        Makefile
        tree.java
        server.java
        peer.java
        tracker.java
```

2. [4 points]: Which of the following statements are true?

(Circle True or False for each choice.)

(a) **True / False** If Alice and Bob concurrently edit different methods which are on different lines of `tree.java` on their respective up-to-date repositories (i.e., they ran SVN up before either of them started editing), then neither Alice nor Bob will be able to commit directly, without encountering an error message.

False. The first one to commit will see no problems. The second one will receive a file-out-of-date error when they attempt to commit.

(b) **True / False** If Alice and Bob concurrently edit different methods of `tree.java` on their respective up-to-date repositories (i.e., they ran SVN up before either of them started editing), neither Alice or Bob will need to manually resolve a conflict.

True. SVN's merge function takes care of resolving situations where people edited different sections of the file.

(c) **True / False** If Alice and Bob concurrently attempt to edit `tree.java` on their respective up-to-date repositories, SVN will prevent Alice and Bob from concurrently editing `tree.java`.

False. SVN uses an optimistic concurrency control scheme, not a pessimistic one.

(d) **True / False** Alice edits `peer.java` and Bob edits `server.java`. Both Alice and Bob verify that the project builds correctly after their edits, and they commit without error. When their sleeping teammate Charles wakes up from a nap and updates, his repository is certain to build without error.

False. Suppose server.java defines a constant that originally only the server code uses. Alice changes the peer code to reference the constant. Bob deletes the constant. Charlie build will generate an error.

Name:

II.3 Worms

Ben decides to start an ISP. He buys a /8 address space (i.e., 2^{24} addresses) that has never been used before. A few days after he buys this address space, a worm is launched. The worm targets a buffer overflow in the FOO server, which listens on UDP port 5044. Ben monitors all traffic sent to his address space on that port and plots the number of packets versus time in Figure 1 below.

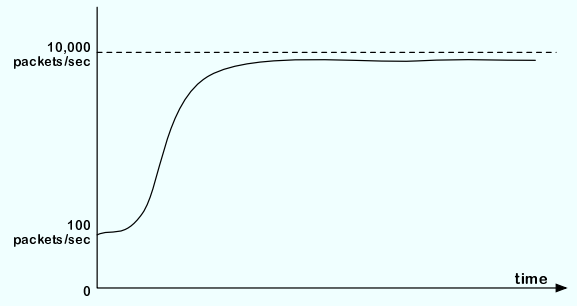


Figure 1: Packets per second from the point that the worm starts.

Assume the worm spreads by probing random IP addresses, and that its pseudo-random number generator is bug-free and generates a complete permutation of the integers before revisiting any integer. Ben learns from a security analyst that each infected machine sent 100 packets/second.

3. [4 points]: Give an estimate of the total number of machines that run the FOO server?
(Circle the BEST answer)

(a) 100 machines

(b) 7.2×10^{18} machines

(c) 25,600 machines

Answer: 10,000 packets / sec divided by 100 packets / sec / machine is 100. The telescope only sees 1/256 of the actual traffic, thus the total is 256 times 100 or 25,600.

(d) 8,000 machines

Name:

4. [5 points]: Ben thinks that the worm used a hit list of vulnerable addresses (i.e., addresses of FOO servers). Do you agree? If you do, what is the best estimate for the number machines contained in the hit list?

(Circle the BEST answer)

(a) no hit list

(b) 100 machines

(c) 256 machines

Answer: The fact that the graph starts with a jump indicates that at the beginning the worm used a hit list. If the network telescope sees 100 packets/second while observing 1/256 of the address space, there must be 256 infected machines at the beginning.

(d) 25600 machines

(e) 80 machines

II.4 Privacy

5. [4 points]: The 4th Amendment of the Constitution, as interpreted in the Olmstead vs. U.S. ruling in 1928, established that the boundary of a person's property defines the area in which he or she can expect privacy protection ("a man's house is his castle"). According to Daniel Weitzner's lecture, which of the following technologies have led to changes in the definition and boundaries of privacy protection in the United States legislative and legal systems?

(Circle True or False for each choice.)

(a) **True / False** telephone booth

True. Katz v. United States. 389 U.S. 347 (1967)

(b) **True / False** email

True. ECPA: email gets status of 1st class mail vs. 3rd party business records

(c) **True / False** community websites like www.facebook.com

False. Not yet, at least.

(d) **True / False** public waste management systems

True or False. Your trash was clarified as non-private in a court case. Depending on whether this is considered a change or not, the answer is either true or false.

Name:

III The Bitdiddler

Ben Bitdiddle is designing a file system for a new handheld computer, the Bitdiddler, which is designed to be especially simple, for, as he likes to say, "people who are just average, like me."

In keeping with his theme of simplicity and ease of use for average people, Ben decides to 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, as in 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 Unix, it also contains a list of blocks that compose the file, and 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 Ben's file system. The basic layout of the Bitdiddler file system is shown in Figure 2.

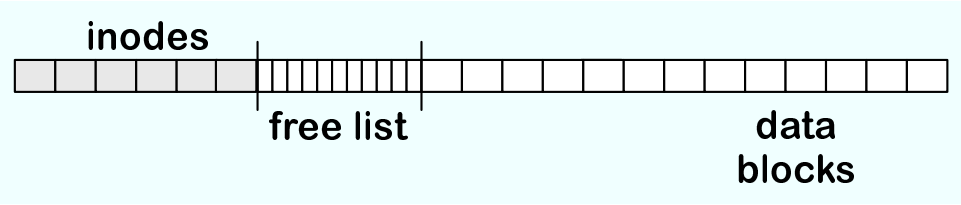


Figure 2: The Bitdiddler's file system layout. The free list indicates which data blocks are not in use, and the inode list indicates the data blocks that compose each file.

III.1 Bitdiddler with synchronous writes

The file system includes six primary API calls (`create`, `open`, `read`, `write`, `close`, `unlink`), which Ben implements correctly and in a straightforward way. They are shown on the next page. All updates to the disk in this pseudocode are synchronous – that is, after a call to write a block of data to disk returns, that block is definitely installed on the disk. Individual block writes are atomic.

Name:

procedure CREATE($filename$)
  scan all non-free inodes to ensure this is not a duplicate file (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 it to disk;

procedure OPEN($filename$)
  // returns file handle
  scan non-free inodes to ensure the file exists;
  if so, allocate and return a file handle that references the inode;

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

procedure READ(fh, buf, len)
  lookup the file handle to determine inode of the file, read it;
  read len bytes of data from the current location in file into buf;

procedure CLOSE(fh)
  remove fh from the file handle table;

procedure UNLINK($filename$)
  scan the disk to find the inode for the file, mark its inode as free;
  write inode to disk;
  mark data blocks used by file as free in free list;
  write modified free list blocks to disk;

Figure 3: File system calls and their implementation in the Bitdiddler. The implementation of these calls is straightforward.

Name:

Ben writes the following simple application for the Bitdiddler:

```
create(filename);
fh = open(filename);
write(fh, appData, length(appData)); //appData contains some data to be written
close(fh);
```

6. [8 points]: Ben notices that if he pulls the batteries out of the Bitdiddler while running his application and then replaces the batteries and reboots the machine, the file his application created exists but contains unexpected data that he didn't write into the file. Which of the following are possible explanations for this behavior? (Assume that the disk controller never partially writes blocks.)

(Circle ALL that apply)

(a) **Yes / No** 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 were written.

No. Without writing the inode, the allocated blocks aren't part of the file. This results in a leak of blocks off the free list but not garbage data in files.

(b) **Yes / No** The inode allocated to Ben's 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.

No. Without partial block writes, this can't happen because the inode is marked with 0 data blocks before being written to disk.

(c) **Yes / No** 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.

Yes. The inode is written out before the contents of the data block. Thus a crash between when the inode is written and when the data block is written will result in the file containing a block with data from a deleted file.

(d) **Yes / No** The free list entry for a data page allocated by the call to write() as well as the data page itself were written to disk, but the new inode was not.

No. The ordering is explicit: allocate off the free list, update inode, then write data. Without asynchronous write-back, this situation cannot occur.

Name:

7. [8 points]: Ben decides to fix inconsistencies in the Bitdiddler's file system by scanning its data structures on disk every time the Bitdiddler starts up. Which of the following inconsistencies can be identified using this approach (without modifying the Bitdiddler implementation)?

(Circle True or False for each choice.)

(a) **True / False** In use blocks on the free list.

True. By following pointers from allocated inodes, all in use blocks can the discovered and checked against the free list.

(b) **True / False** Unused blocks not on the free list.

True. After discovering in-use blocks as above, all remaining blocks are unused and thus should be on the free list.

(c) **True / False** In-use blocks containing data from previously unlinked files.

False. This is the case examined in the previous problem (6.c). There is no way to determine if the data in a file is correct.

(d) **True / False** Blocks used in multiple files.

True. If a block is reachable from multiple inodes, it's used in multiple files.

III.2 Logging Bitdiddler

Alyssa points out Ben's Bitdiddler with synchronous block writes doesn't guarantee that file system calls (e.g., write, close, etc.) provide all-or-nothing atomicity (e.g., they happen completely or don't happen at all). Instead, she suggests that Ben use a logging approach to help provide all-or-nothing atomicity for each file system call.

She proposes that the file system synchronously write a log record before every create, write, or unlink call. Each log record contains the type of operation performed, the name of the file, the old and new value of the data being written (for write), and the offset where the new data will be written (for write). The system ensures that log record writes are atomic.

Ben modifies the Bitdiddler code to perform these logging operations to a separate log file on a separate disk before doing the create, write, or unlink operations themselves. He then implements a crash recovery protocol that scans the log after a crash and uses it to ensure all-or-nothing atomicity.

8. [5 points]: Which of the following crash recovery protocols ensures that file system calls are all-or-nothing (assuming there was at most one file system call running when the system crashed)?
(Circle True or False for each choice.)

(a) **True / False** Scan the log from the beginning to the end; re-apply each logged operation to the specified file in forward-scan order.
True. Replaying the log in forward order will result in every file having its most recent contents. Since the log is keyed on filename, a create that returns a different inode than when it was originally executed will not cause a problem (file descriptors are volatile and thus do not persist through a crash). With this method, the disk could even be wiped clean before recovery and the recovery operation would work correctly.

(b) **True / False** Scan the log from the end to the beginning; re-apply each logged operation to the specified file in reverse-scan order.
False. Replaying the log in reverse order will result in all files having no contents (the earliest log record relating to each file is a create record, which when replayed sets the number of data blocks in the file to 0.

(c) **True / False** Read the last log record and re-apply it.
True. Since file system operations are synchronous and only one operation is in progress at a time, there can be only one operation interrupted by a crash. Thus, reapplying the most recent log record is sufficient.

(d) **True / False** Scan the log from the beginning to end and identify all the files that should have been created but don't exist (e.g., don't have an inode and were not deleted). Then, scan the log from beginning to end, re-doing creates and writes for those files in forward-scan order.
False. This process only recovers newly created files, ignoring any changes to pre-existing files.

(e) **True / False** Scan the log from the beginning to end and identify all the files that should have been created but don't exist (e.g., don't have an inode and were not deleted). Then, scan the log from the end to the beginning, re-doing creates and writes for those files in reverse-scan order.
False. As in part d, this doesn't recover old files with recent writes, and moreover it doesn't recover new files (they all end up empty).

Name:

III.3 High-Performance Logging Bitdiddler

Ben observes that synchronous writes slow down the performance of his file system. To improve performance with this logging approach, Ben modifies the Bitdiddler to include a large file system cache. `write`, `create`, or `unlink` update blocks in the cache. To maximize performance, the file system propagates these modified blocks to disk asynchronously, in an arbitrary order, and at a time of its own choosing. Ben's file system still writes log records synchronously to ensure that these are on disk before executing the corresponding file system operation.

9. [5 points]: Which of the following crash recovery protocols ensures that file system calls are all-or-nothing in this high performance version of the Bitdiddler (assuming there was at most one file system call running when the system crashed)?

(Circle True or False for each choice.)

(a) **True / False** Scan the log from the beginning to the end; re-apply each logged operation to the specified file in forward-scan order.

True. Replaying the log in forward order will result in every file having its most recent contents. Since the log is keyed on filename, a create that returns a different inode than when it was originally executed will not cause a problem (file descriptors are volatile and thus do not persist through a crash). A full replay takes care of any asynchrony issues.

(b) **True / False** Scan the log from the end to the beginning; re-apply each logged operation to the specified file in reverse-scan order.

False. Replaying the log in reverse order results in disaster.

(c) **True / False** Read the last log record and re-execute it.

False. With asynchronous disk block writeback, the results of many previous updates may not have been written to disk when the system crashes. Thus, a full log replay is necessary.

(d) **True / False** Scan the log from the beginning to end and identify all the files that should have been created but don't exist (e.g., don't have an inode and were not deleted). Then, scan the log from beginning to end, re-doing creates and writes for those files in forward-scan order.

False. Didn't work under the simple case, still doesn't work now.

(e) **True / False** Scan the log from the beginning to end and identify all the files that should have been created but don't exist (e.g., don't have an inode and were not deleted). Then, scan the log from the end to the beginning, re-doing creates and writes for those files in reverse-scan order.

False. Ditto.

Name:

10. [8 points]: Alyssa suggests that Ben might want to modify his system to periodically write checkpoints to make recovery efficient. Which of the following checkpoint protocols will allow Ben's recovery code to start recovering from the latest checkpoint while still ensuring all-or-nothing atomicity of each file system call in the high performance, asynchronous Bitdiddler?

(Circle ALL that apply)

(a) **Yes / No** Complete any currently running file system operation (e.g., open, write, unlink, etc.), stop processing new file system operations, write all modified blocks in the file system cache to disk, and then write a checkpoint record to the log containing a list of open files.

Yes. When there are no operations in progress and the file system disk cache contains no modified blocks, the state of the disk matches the state of the log. It is safe to write a checkpoint log at this point. There is no need to include open files in the checkpoint (open files do not persist across a crash), but it doesn't hurt.

(b) **Yes / No** Complete any currently running file system operation, stop processing new file system operations, write all modified blocks in the file system cache to disk, and then write a checkpoint record to the log containing no additional information.

Yes. When there are no operations in progress and the file system disk cache contains no modified blocks, the state of the disk matches the state of the log. It is safe to write a checkpoint log at this point.

(c) **Yes / No** Write all modified blocks in the file system cache to disk without first completing current file system operations, and then write a checkpoint record to the log containing a list of open files.

No. Since there are operations in progress, the state of the disk and the state of the log may be out of sync. Noting the list of open files will indicate which files might have issues, but there may be log records before the checkpoint log record that must be replayed to ensure all-or-nothing semantics.

(d) **Yes / No** Write a checkpoint record to the log (containing a list of open files), but do not write all modified blocks to disk.

No. With modified blocks in the cache, the state of the disk and the state of the log are out of sync, potentially by an arbitrary amount. Any log record before the checkpoint could possibly need to be replayed to ensure all-or-nothing semantics.

Name:

III.4 Transactional Bitdiddler

By now, Ben is really excited about his file system and decides to add some advanced features. From taking 6.033, he knows that transactions are a way to make multiple operations appear as though they are one isolated, all-or-nothing atomic action, and he decides he would like to make his file system transactional, so that programs can commit changes to several files as a part of one transaction, and so that concurrent users of the file system are isolated from seeing the effects of others' partially complete transactions. He adds three new function calls:

```
tid = beginTransaction();
commit(tid);
abort(tid);
```

Ben modifies the `open()` call to take a `tid` parameter that specifies the transaction that this file access will be a part of. In this model, one transaction can access multiple files, and those changes are only visible to other transactions after commit has been called. If the system crashes before a transaction commits, its actions are undone during recovery.

Ben decides to implement transactions using locking. He places a single exclusive lock on each file, and modifies open so that it attempts to acquire that lock before returning. If another transaction currently has the lock, it blocks until the lock is free.

The implementation of Ben's new `open` call is as follows:

```
FileHandle open(int tid, String file) {
    int lockingTid;
    do {
        lockingTid = testAndAcquireLockOnFile(file, tid);
    } while (lockingTid != tid);
    return doOpen(file);
}
```

`testAndAcquireLockOnFile()` tests to see if file is locked by a transaction, and if it is, returns the id of the locking transaction. If it is not locked, it acquires the lock on behalf of `tid`, and returns `tid`.

Ben modifies his logging code so that each log record includes the `tid` of the transaction it belongs to and adds `COMMIT` and `ABORT` records to indicate the outcome of transactions.

Name:

Ben is writing the code for the close and commit functions, and is trying to figure out when he should release the locks acquired by his transaction. His code is as follows:

```
close(FileHandle fh) {
     remove fh from file handle list
A:
}
commit(int tid) {
    FileHandle files[] = getFilesLockedByTid(tid);
    for (f in files) {
        if (isOpen(f)) close (f);
    }
B:
    log  a COMMIT record for tid ; //commit point
C:
}
```

Note that commit closes files before committing them, and that files may be closed before commit is called.

11. [5 points]: When can Ben's code release a lock on a file (or all files) while still ensuring that the locking protocol implements a serial-equivalent transaction schedule?

(Circle ALL that apply)

(a) **Yes / No** At the line labeled A:

No. Files may be closed before commit is called. Imagine T1 reads file A, closes and re-opens it, then writes to it. In the middle, a transaction T2 could write file A, resulting in a non-serializable schedule.

(b) **Yes / No** At the line labeled B:

No. Suppose transaction T1 got to point B, released its locks, then stalled for a while. In the meantime, another transaction T2 reads values that T1 produced and successfully commits. Then the system crashes. T1 will be aborted because it did not manage to write a commit record. However, T2's results did get written and they depended on results from an aborted transaction.

(c) **Yes / No** At the line labeled C:

Yes. Locks can only be released after the commit point.

Name:

Ben begins running his new transactional file system on the Bitdiddler. The Bitdiddler allows multiple programs to run concurrently, and Ben is concerned that he may have a bug in his implementation because he finds that sometimes some of his applications block forever waiting for a lock. Alyssa points out that he may have deadlocks.

Ben hires you to help him figure out whether there is a bug in his code or if applications are just deadlocking. He shows you several traces of file system calls from several programs; your job is to figure out out whether the operations indicate a deadlock, and if not, what one equivalent serial order of the transactions in the system could be (there may be several possible equivalent orders – you only need to show one.)

After each trace, assume that there are no more `read` or `open` calls but that transactions will commit if they have not deadlocked.

Alyssa helps out by analyzing and commenting on the first trace for you (you should read these traces as though time goes down the page – so the first one shows that the first action is `begin(T1)`, followed by `begin(T2)`):

Example Trace

```
Program 1:              Program 2:
begin(T1)
                        begin(T2)
fh=open(T1,'foo')
                        fh2=open(T2,'foo') //blocks waiting for T1
write(fh,'hi')
close(fh)
commit(T1)
                        write(fh2,'hello')
                        //Program 2 can now commit without deadlocking
```

Give an equivalent serial order, or write DEADLOCK: T1, T2

Trace 1

```
Program 1:              Program 2:              Program 3:
begin(T1)
                        begin(T2)
                                                begin(T3)
fh=open(T1,'foo')
                        fh2=open(T2,'bar')
                                                fh3=open(T3,'baz')
write(fh,'hi')
                        fh4=open(T2,'baz')
                                                fh5=open(T3,'foo')
close (fh)
commit (T1)
                        ...                     ...
```

12. [4 points]: Give an equivalent serial order, or write DEADLOCK: T1,T3,T2

Name:

Trace 2

```
Program 1:          Program 2:          Program 3:          Program 4:
begin(T1)
                    begin(T2)
fh=open(T1,'foo')
write(fh,'boo')
                    fh2=open(T2,'bar')
                    write(fh2,'car')
close(fh)
commit(T1)
                                        begin(T3)
                                                            begin(T4)
                                                            fh3=open(T4,'foo')
                                        fh4=open(T3,'bar')
                    fh5=open(T2,'foo')
                                                            fh6=open(T4,'bar')
                    ...                 ...                 ...
```

13. [4 points]: Give an equivalent serial order, or write DEADLOCK:DEADLOCK

T2 and T4 form a cycle in the happens-before graph. T2 depends on foo's value from T4 and T4 depends on bar's value from T2.

Trace 3

```
Program 1:          Program 2:          Program 3:          Program 4:
begin(T1)
                    begin(T2)
fh=open(T1,'foo')
write(fh,'boo')
                    fh2=open(T2,'bar')
                    write(fh2,'car')
close(fh)
commit(T1)
                                        begin(T3)
                                                            begin(T4)
                                                            fh3=open(T4,'foo')
                                        fh4=open(T3,'foo')
                    fh5=open(T2,'foo')
                                                            fh6=open(T4,'baz')
                    ...                 ...                 ...
```

14. [4 points]: Give an equivalent serial order, or write DEADLOCK:T1,T4,T3,T2 or T1,T4,T2,T3

Name:

III.5 Transactional, Distributed Bitdiddler

Ben begins to get really carried away. He decides that he wants the Bitdiddler to be able to access files of remote Bitdiddler's via a networked file system protocol, but he wants to preserve the transactional behavior of his system, such that one transaction can update files on several different computers. He remembers that one way to provide atomicity when there are multiple participating sites is to use the two-phase commit protocol.

The protocol works as follows: one site is appointed the coordinator. The program that is reading and writing files runs on this machine, and issues requests to begin and commit transactions and read and write files on both the local and remote file systems (the "workers").

When the coordinator is ready to commit, it uses the logging-based two-phase commit protocol we studied in class, which works as follows: First, the coordinator sends a *prepare* message to each of the workers. For each worker, if it is able to commit, it writes a log record indicating it is entering the *prepared* state and send a *yes* vote to the coordinator; otherwise it votes *no*. If all workers vote *yes*, the coordinator logs a *commit* record and sends a *commit* outcome message to all workers, which in turn log a *commit* record. If any worker votes *no*, the coordinator logs an *abort* record and sends an *abort* message to the workers, which also log *abort* records. After they receive the transaction outcome, workers send an *acknowledgment* message to the coordinator. Once the coordinator has received an acknowledgment from all of the workers, it logs an *end* record. Workers that have not learned the outcome of a transaction periodically contact the coordinator asking for the outcome. If the coordinator has not received an acknowledgment from some of the workers, it resends the outcome.

Figure 4 shows a coordinator node issuing requests to begin a transaction and read and write files on several worker nodes.

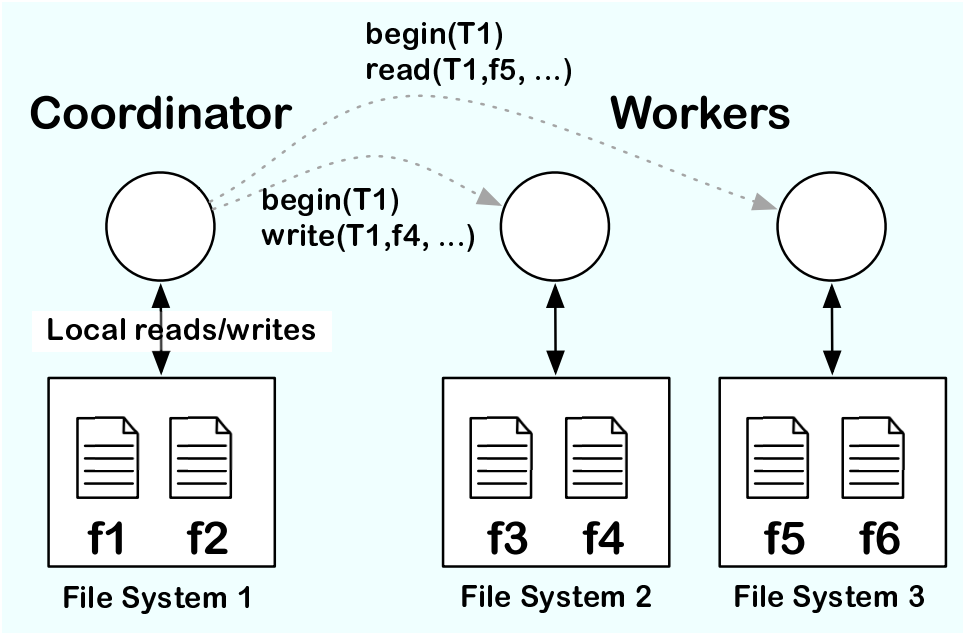


Figure 4: Coordinator issuing transactional reads and writes to several workers in the two-phase commit based distributed file system for the Bitdiddler.

Name:

Ben is having a hard time figuring out what to do when one of the nodes crashes in the two phase commit protocol. When a worker node recovers and finds log records for a transaction, it has several options:

W1. Abort the transaction by writing an ”abort” record

W2. Commit the transaction by writing a ”commit” record

W3. Resend its vote to the server and ask it for transaction outcome

Similarly, the coordinator has several options when recovering from a crash; it can:

C1. Abort the transaction by writing an “abort” record

C2. Do nothing

C3. Send commit messages to the workers

15. [8 points]: For each of the following situations, indicate the appropriate action that a worker or coordinator should take:

(Write in the BEST action (W1–W3 or C1–C3))

(a) The coordinator crashes, finds log records for a transaction but no commit record

Action C1

Since there is no commit record, it is safe for the coordinator to abort the transaction. If it sent prepare messages, the workers will check back and receive the abort notification.

(b) The coordinator crashes, finds a commit record for a transaction but no end record indicating the transaction is complete

Action C3

A commit record on the coordinator means the transaction did commit. By resending the commit, the works will acknowledge the commit and the coordinator will be able to write an end record.

(c) A worker crashes, finds a prepare record for a transaction

Action W3

The worker does not know if the transaction committed or aborted, and thus must ask the coordinator.

(d) A worker crashes, and finds log records for a transaction, but no prepare or commit records

Action W1

If the worker does not find a prepare or commit record in its log, it doesn’t know if it actually finished executing the transaction, thus it must abort it.

Name:

III.6 Space-efficient, confidential Bitdiddler

Ben uses the original Bitdiddler with synchronous writes. Ben stores many files in the file system on his handheld computer, and runs out of disk space quickly. He looks at the blocks on the disk and discovers that many blocks have the same content. To reduce space consumption he augments the file system implementation as follows:

A. The file system keeps a table in memory that records for each allocated block a 32-bit non-cryptographic hash of that block. (When the file system starts, it computes this table from the on-disk state.) Ben talks to a hashing expert, who tells Ben to use the b-bit (here $b = 32$) non-cryptographic hash function

$$H(\text{block}) = \text{block } mod\ P$$

where P is a large b-bit prime number that yields a uniform distribution of hash values throughout the interval $[0 \ldots 2^b - 1]$.

B. When the file system writes a block b for a file, it checks if the table contains a block number d whose block content on disk has the same hash value as the hash value for block b. If so, the file system frees b and inserts d into the file's inode. If there is no block d, the file system writes b to the disk, and puts b's block number and its hash in the table.

(You can ignore what happens when a user unlinks a file.)

16. [3 points]: Occasionally, Ben finds that his system has files with incorrect contents. He suspects hash collisions are to blame. These might be caused by:

(Circle True or False for each choice.)

(a) **True / False** Accidental collisions: different data blocks hash to the same 32-bit value.
True. 32 bits is rather small these days. You can easily have more than 2^{32} files on a disk, making accidently collisions likely.

(b) **True / False** Engineered collisions: adversaries can fabricate blocks that hash to the same 32 bit value.
True. By knowing P, it is easy for an adversary to construct another block that hashes to the same value.

(c) **True / False** A block whose hash is the same as its block number.
False. Blocks are indexed by their block number, not their hash, making collisions between the two unimportant for system correctness.

Name:

17. [5 points]: For each of the following proposed fixes, circle which of the problems listed in Question 16 (A, B, or C) it is likely to fix (if you circled False for any of the options in 16, you do not need to circle that option here):

(Circle ALL that apply)

(a) Use a $b = 160$-bit non-cryptographic hash in step A of the algorithm.

A B C

A. 160 bits is enough space that is very unlikely that accidental collisions will occur. However, the hash function is still not collision-resistant, thus allowing an adversary to construct another block with the same hash.

(b) Use a 160-bit cryptographic hash such as SHA-1 in step A of the algorithm.

A B C

A and B. A 160 bit cryptographic hash is large enough to avoid accidental collisions. Furthermore, a cryptographic hash (like SHA-1) is collision-resistant, meaning that it computationally impossible for an adversary to find a second block with the same hash.

(c) Modify step B of the algorithm so that when a matching hash is found, it compares the contents of the stored block to the data block and treats the blocks as different unless their contents match.

A B C

A and B. Actually checking the blocks for equality prevents both problems, at the cost of a little performance.

Name:

Ben decides he wants to encrypt the contents of the files on disk so that if someone steals his handheld computer, they cannot read the files stored on it. Ben considers two encryption plans:

- **User-key encryption:** One plan is to give each user a different key and use a secure block encryption scheme with no cryptographic vulnerabilities to encrypt the user's files. Ben implements this by storing a table of (user name, key) pairs, which the system stores securely on disk.
- **Hash-key encryption:** One problem with user-key encryption is that it doesn't provide the space saving if blocks in different files of different users have the same content. Instead, Ben proposes hash-key encryption, which encrypts a block using the cryptographic hash of the content of that block as a shared-secret key (i.e., ENCRYPT(*block*, HASH(*block*)). Ben reasons that since the output of the cryptographic hash is pseudo-random, this is just as good as choosing a fresh random key. Ben implements this scheme by modifying the file system to use the table of hash values as before, but now the file system writes encrypted blocks to the disk instead of plain-text ones. This way blocks are encrypted but Ben still gets the space savings for blocks with the same content. You can ignore how the table of block hash values is maintained; just assume that the file system maintains it securely and that the file system can find the hash values to decrypt blocks.

18. [6 points]: Which of the following statements are true of hash-key encryption?
(Circle True or False for each choice.)

(a) **True / False** If Alyssa can guess the contents of a block (by enumerating all possibilities, or by guessing based on the file metadata, etc), it is easy for her to verify whether her guess of a block's data is correct.

True. She can try encrypting her guess with the hash-key encryption function. If the result matches the block stored on disk, she knows her guess was correct.

(b) **True / False** If Alyssa can discover the 32-bit block numbers referenced by inodes in the file system, she can learn something about the contents of Ben's files.

True. If Alyssa notices that one of her files uses the same block(s) as one of Ben's files, Alyssa knows exactly the contents of the block(s).

(c) **True / False** The file system can detect when an adversary changes the content of a block on disk.

True. If the adversary changes a block on disk, when the file system decrypts the block, it can hash the resulting value and compare it with the original hash of the block. If they match, the block is authentic.

Name:

19. [6 points]: Which of the following statements are true of user-key encryption?
(Circle True or False for each choice.)

(a) **True / False** If Alyssa can guess the contents of a block but doesn't know Ben's key, it is easy for her to verify whether her guess of a block's data is correct.

False. She must try all possible keys to decrypt the block. This is not easy.

(b) **True / False** If Alyssa can discover the 32-bit block numbers referenced by inodes in the file system, she can learn something about the contents of Ben's files.

False. Knowing the block number gives no information about the content of the blocks in Ben's files, since those blocks are encrypted with Ben's key.

(c) **True / False** The file system can detect when an adversary changes the content of a block on disk.

False. The block will decrypt into some value, but the file system has no way of verifying that it is different that the value it originally encrypted.

Ben's brain has exploded. End of Quiz III. Enjoy!

Name: