15-440 Distributed Systems
Homework 2

Due: November 10, 11:59:59 PM

Name:	
Andrew: ID	

December 1, 2010

1. Fix the code below so that the numbers print in order, using mutexes and conditional variables.

```
#include <pthread.h>
#include <stdio.h>

pthread_cond_t c1, c2;
pthread_mutex_t m1, m2;

void* thread_one (void* args)
{
  pthread_mutex_lock(&m1);
  printf("1");
  pthread_cond_signal(&c1);
  pthread_cond_wait(&c1, &m1);
  printf("5");
  pthread_cond_signal(&c1);
  pthread_mutex_unlock(&m1);
  return NULL;
}

void* thread_two (void* args)
{
  pthread_mutex_lock(&m2);
  printf("2");
  pthread_cond_signal(&c2);
  pthread_cond_wait(&c2, &m2);
  printf("4");
  pthread_cond_signal(&c2);
  pthread_mutex_unlock(&m2);
  return NULL;
}

int main()
{
  pthread_t tid_one;
  pthread_t tid_two;
  pthread_mutex_init(&m1, NULL);
  pthread_mutex_init(&m2, NULL);
  pthread_cond_init(&c1, NULL);
  pthread_cond_init(&c2, NULL);
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
  pthread_create(&tid_one, NULL, thread_one, NULL);
  pthread_create(&tid_two, NULL, thread_two, NULL);
  pthread_cond_wait(&c1, &m1);
  pthread_cond_wait(&c2, &m2);
  printf("3");
  pthread_cond_signal(&c2);
  pthread_cond_wait(&c2, &m2);
  pthread_cond_signal(&c1);
  pthread_cond_wait(&c1, &m1);
  pthread_join(tid_one, NULL);
  pthread_join(tid_two, NULL);
  printf("6\n");
}
```

2. In this problem you're going to compare the performance of disk writes both with and without write-ahead logging (WAL). For all of these sub-problems assume that the disk is the performance bottleneck, that the disk has the following performance parameters, and that all specifications are given in SI units:

Capacity	1000 GB
Seek latency	5 ms
Rotational delay	2 ms
Transfer speed	100 MB/s

Assume that a disk address consists of a 48-bit page address plus a 16-bit offset specifying the location within the disk page. All log records contain the 64-bit disk address for the transaction's previous log entry, a 32-bit transaction ID, and an 8-bit record type (104 bits total). The log supports the `BEGIN`, `COMMIT`, `ROLLBACK`, and byte-level `UPDATE` record types as described in class, as well as a new `WRITE` record described below.

The `BEGIN`, `COMMIT`, and `ROLLBACK` records contain just the 104-bit header described above. The `UPDATE` record contains the 104-bit header plus the disk address being updated (64 bits), the length in bytes of the update (16 bits), plus both the old and new values of the location being updated (184 bits total, plus twice the length of the update to store the old and new values).

The `WRITE` record is used for a disk update that does not need to be undone if a transaction fails to commit. (For instance, a write to an otherwise unused block does not need to be erased if the transaction rolls back.) It is the same as the `UPDATE` record except that it does not contain the old value of the location being updated (184 bits total, plus the length of the update to store the new value).

Finally, assume that the operating system implements an in-memory page cache for disk pages. When a transaction commits, *some* persistent write must occur – either to the log (if logging is used) or to the disk page itself (if logging is not used). The system can buffer some disk pages in the page cache and write just the in-memory pages if logging is being used.

(a) Suppose you copy a large, single 4 GB file from a friend's computer, writing the file to your single local disk. Without logging, how long will this take? Clearly state any assumptions you make.

Solution: Because the disk is the performance bottleneck, the network speed is not relevant. To write the 4 GB file, your system must select a location to write, seek the disk arm to that location (5 ms), wait for the location to rotate under the head (2 ms), and write the 4 GB file. Assuming that 4 GB can be written continuously to the disk at the 100 MB/s transfer rate, the write itself will require 40 seconds, for a total of 40.007 s.

(b) How long will writing the same 4 GB file take on your computer, using WAL?

Solution: To write a single 4 GB file with WAL, the workload will likely consist of a single `BEGIN` record, some number of `WRITE` records, and a single `COMMIT` record. `WRITE` records are more likely than `UPDATE` records because the OS will probably not need to undo the initial write of the file data to undo the write of the file; it will likely just need to unlink the file's inode and mark its data blocks as free. A single `WRITE` record will not suffice because the data length for a `WRITE` record is limited to 16 bits, or at most a 64KB length. If the logging system imposes no other limits on a record size, the whole 4 GB file will require 61036 `WRITE` records (totaling 4.0014 GB of data).

Another complication is the size of the page cache. In practice, the system will write both the log and the actual data to the page cache, and need to occasionally flush both the log and the data to disk when the page cache is full. The log and the data each might be written sequentially, but the disk will likely need to seek between writes to the log and writes to the file. Without knowing the size of the page cache and the algorithm used to write pages to disk, we cannot accurately predict the number of seeks needed to flush the page cache to disk. For reasonable modern page cache sizes, however, the cache will need to be flushed only a small number of times, so the total aggregate time of these extra seeks will likely be small (¡ 1 s).

Ignoring the cost of extra seeks while flushing the page cache and assuming the disk can write the entire log sequentially and the data itself sequentially, this sequence will require 7 ms for the initial seek, 40.014 s to write the log data, 7 ms for a seek to the file itself, and 40 s to write the file data. We can therefore lower-bound the time needed to be 80.028 s, acknowledging that the actual write-time will likely be higher than this.

(c) Suppose that instead of a single 4 GB write, you overwrite the contents of a single 4000-byte file 1 million times. How long will this take without logging? (Hint: When overwriting the same file repeatedly, what causes the delay between disk writes?)

Solution: A 4000-byte file will fit within a single disk track on modern hardware, so the disk will not need to seek between writes to the file. The disk will, however, need to wait for the beginning of the file to rotate under the disk head before writing each time. After the initial 5 ms seek, the disk will be able to complete one 4000-byte write per 2 ms disk rotation, for a total time of 2000.005 s. Note that the write-time for the 4000-byte file does not influence the answer! As long as the 4000-byte file fits in a single track, the time the disk spends writing the file will overlap the rotation time. If the file were slightly longer (but still within a single track), the disk would need to wait less time for the start of the file to rotate back under the disk head to begin the next write to the file.

(d) How long would part (c) take, with WAL? (As always, state any assumptions you make.)

Solution: The answer here depends on how the workload is implemented. If implemented as 1 million transactions the analysis is much as in (c). The disk page for the file itself can always be kept in the page cache, and only log will need to be written to disk for each transaction commit. These log records will each require 8069 bytes (the 23-byte `BEGIN` record, a 23-byte header plus twice 4000 bytes data for the `UPDATE` record, and a 23-byte `COMMIT` record). Here, though, the disk will need to seek back to *end* of the current log for each commit record, so the write-time will not overlap the rotation time as in (c). For the 1 million records, the disk therefore will spend 2000 s in rotation delay and 80.69 s actually writing the disk, for a total of 2080 s.

If the workload is implemented as a single transaction (one `BEGIN`, 1 million `UPDATE` s, and one `COMMIT`) then the log will not need to be flushed to disk after each write to the file. The file itself can again be stored in the page cache, and the log could then be written sequentially when the transaction finally committed (or when the page cache was full), for a total write-time of 80.69s.

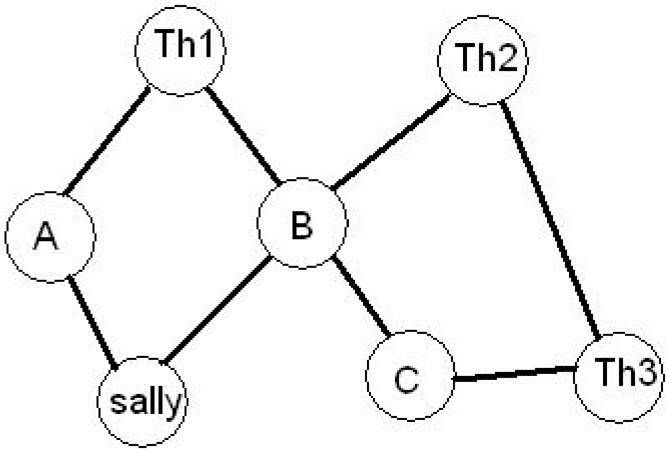
An aside: Many journaling filesystems take an intermediate approach, using WAL only for the filesystem metadata (such as the directory structure, free block list, etc) and not journaling the file data itself. This allows the filesystem to guarantee the ACID properties for some operations but not others. For instance, with metadata journaling, a file move operation will always result in the file existing at exactly one of its new or old locations, even if a power failure occurs. Editing an existing file, however, could result in a non-sensical mix of new and old data, because the file edits themselves would not be journaled.

3. Sally has a very very very large map. She's really excited about finding all the movie theaters on the map, and how long it would take to get to each one (she collects movie tickets from various theaters, it's quite an impressive collection). Sally will need your help to find the shortest path from her house to every movie theater using MapReduce.

   Assume the following:

   - There are arbitrary points on the graph that represent intersections, known as A, B, C, etc.
   - All edges on the graph are undirected.

   Here is an example of small map:



   Clearly, on a graph of smaller size, Sally can use Dijkstra's to find the information she needs, but because this graph is so large, she wants to use MapReduce, where the MapReduce will effectively be another graph traversal algorithm.

   (a) Suppose that your input for the problem looks like this:

```
Sally A 3
B Th2 4
C B 1
...
```

   Set up a possible solution to this problem. Tell us how many MapReduce phases there would be, and what each Mapper and each Reducer would do. The same phase can be run more than once, so make sure you explain how and why. Give examples of input and output for each map and reduce of a phase.

   The general format can be:
   General idea:

   Phase 1: Map:
   Phase 1: Reduce:
   Run Phase 1 X times.

   Solution: Without Map/Reduce, we would run Dijkstra's to solve this problem. With Map/Reduce, we can use parallelized BFS instead.

```
Phase 1 would do the following: (run this phase once)
map:
input: <node1, (node2, distance)>
output: <node1, (node2, distance)>
```

```
<node2, (node1, distance)>

reduce:
input: <node, {(node1, dist1), (node2, dist2), (node3, dist3)}>
output:
if(node is sally):
<node, {0, (node1, dist1), (node2, dist2), (node3, dist3)}>
else:
<node, {inf, (node1, dist1), (node2, dist2), (node3, dist3)}>

Phase 2: (run this phase until convergence, at most N times, where N is the number of nodes)
map:
input: <node, {currdist, (node1, dist1), (node2, dist2), (node3, dist3)}>
output:
<node, {currdist, (node1, dist1), (node2, dist2), (node3, dist3)}>
if(currdist is inf):
ignore
  else:
<node1, {(currdist+dist2)}>
<node2, {(currdist+dist2)}>
<node3, {(currdist+dist3)}>

reduce:
input: <node, {(currdist1), (node1, dist1), (node2, dist2), (node3, dist3)},
  {(currdist2)},
  {(currdist3)}>
output:
mindist = min(currdist1, currdist2, currdist3)
<node, {mindist, (node1, dist1), (node2, dist2), (node3, dist3)}>


This will get you the distances to the theaters. If you want to keep track of the path also, y

A sample run is:
------------------------------------
input to Phase 1 map:
Sally Th1 7
Th1 A 3
Th1 E 2
E Th2 4
A Th2 3
A B 4
Sally A 2
Sally B 5
B Th3 6

output of map:
<Sally, (Th1, 7)>
<Th1, (Sally, 7)>
<Th1, (A, 3)>
<A, (Th1, 3)>
```

```
<Th1, (E, 2)>
<E, (Th1, 2)>
<E, (Th2, 4)>
<Th2, (E, 4)>
...

input to Phase 1 reduce:
<Sally, {(Th1, 7), (A, 2), (B, 5)}>
<E, {(Th1, 2), (Th2, 4)}>
...

output of reduce:
<Sally, {0, (Th1, 7), (A, 2), (B, 5)}>
<E, {inf, (Th1, 2), (Th2, 4)}>
-----------------------------------
input to Phase 2 map:
(the output of the reducer of phase 1)

output:
<Sally, {0, (Th1, 7), (A, 2), (B, 5)}>
<Th1, {7}>
<A, {2}>
<B, {5}>
<E, {inf, (Th1, 2), (Th2, 4)}>
<Th3, {inf, (B, 6)}>
...

input to Phase 2 reduce:
<A, {2}, {inf, (Sally, 7), (Th1, 3), (B, 4), (Th3, 3)}>
<B, {5}, {inf, (Th3, 6), (A, 4), (Sally, 5)}>
...

output of reduce:
<A, {2, (Sally, 7), (Th1, 3), (B, 4), (Th3, 3)}>
<B, {5, (Th3, 6), (A, 4), (Sally, 5)}>
...

And so on, run Phase 2 again and again and again.
```

4. If the real-time clocks in a group of workstations can drift 15 seconds max per day, how frequently should the clocks be synchronized for each approach to keep them within 1 second of each other using a) Christian's Algorithm and b) The Berkely Algorithm?

5. You have set up a fault-tolerant banking service. Based upon an examination of other systems, you've decided that the best way to do so is to use Paxos to replicate log entries across three servers, and let one of your employees handle the issue of recovering from a failure using the log.

   The state on the replicas consists of a list of all bank account mutation operations that have been made, each with a unique request ID to prevent retransmitted requests, listed in the order they were committed.

   Assume that the replicas execute Paxos for every operation.[1] Each value that the servers agree on looks like "account action 555 transfers $1,000,000 from Mark Stehlik to David Andersen". When a server receives a request, it looks at its state to find the next unused action number, and uses Paxos to propose that value for the number to use.

   The three servers are S1, S2, and S3.

   At the same time:

   S1 receives a request to withdraw $500 from A. Carnegie.
   - S1 picks proposal number 101 (the n in Paxos)
   - S2 receives a request to transfer $500 from A. Carnegie to A. Mellon.
     - S2 picks proposal number 102

   Both servers lok at their lists of applied account actions and decide that the next action number is 15. So both start executing Paxos as a leader for action 15.

   Each sequence below shows one initial sequence of messages of a possible execution of Paxos when both S1 and S2 are acting as the leader. Each message shown is received successfully. The messages are sent one by one in the indicated order. No other messages are sent until after the last message shown is received.

   Answer three questions about the final outcomes that could result from each of these sequences:

   (a) Is it possible for the servers to agree on the withdrawal as entry 15?
   - Is it possible for the servers to agree on the transfer as entry 15?
   - For each of these outcomes, explain how it either could occur or how Paxos prevents it.

   To be clear on the message terminology: The PREPARE message is the leader election message. RESPONSE is the response to the prepare message. ACCEPT says "you've agreed that I'm the leader, now take a value."

   Sequence 1:

```
S1 -> S1 PREPARE(101)
S1 -> S1 RESPONSE(nil, nil)

S1 -> S2 PREPARE(101)
S2 -> S1 RESPONSE(nil, nil)

S1 -> S3 PREPARE(101)
S3 -> S1 RESPONSE(nil, nil)

S2 -> S1 PREPARE(102)
S2 -> S2 PREPARE(102)
S2 -> S3 PREPARE(102)
... the rest of the Paxos messages.
```

   Solution: Only the transfer can succeed. The higher-numbered PREPARE, which was received by all servers, will cause them to ignore the lower numbered proposals by S1.

[1]In practice, most systems use Paxos to elect a primary and let it have a lease on the operations for a while, but that adds complexity to the homework problem.

Sequence 2:

```
S1 -> S1 PREPARE(101)
S1 -> S1 RESPONSE(nil, nil)

S1 -> S2 PREPARE(101)
S2 -> S1 RESPONSE(nil, nil)

S1 -> S3 PREPARE(101)
S3 -> S1 RESPONSE(nil, nil)

S1 -> S3 ACCEPT(101, ‘‘withdraw...’’)

S2 -> S1 PREPARE(102)
S2 -> S2 PREPARE(102)
S2 -> S3 PREPARE(102)
... the rest of the Paxos messages.
```

Solution: Either can succeed. If S2 receives a prepare OK from S3 before hearing from S1 and S2, it will include the value ”withdraw” in its later messages. If it hears from S2 and S1 first, it will conclude (correctly) that it has a majority telling it it can do what it wants, and it will propose the transfer.

Sequence 3:

```
S1 -> S1 PREPARE(101)
S1 -> S1 RESPONSE(nil, nil)

S1 -> S2 PREPARE(101)
S2 -> S1 RESPONSE(nil, nil)

S1 -> S3 PREPARE(101)
S3 -> S1 RESPONSE(nil, nil)

S1 -> S3 ACCEPT(101, ‘‘withdraw...’’)
S1 -> S1 ACCEPT(101, ‘‘withdraw..’’)
S2 -> S1 PREPARE(102)
S2 -> S2 PREPARE(102)
S2 -> S3 PREPARE(102)
... the rest of the Paxos messages.
```

Solution: Only the withdraw can succeed. It has already established a majority. Any subsequent PREPARE OK responses must include at least one that says value = withdraw.

(b) Suppose one of the servers received an ACCEPT for a particular instance of Paxos (remember, each “instance” agrees on a value for a particular account event), but it never heard back about what the final outcome was. What steps should the server take to figure out whether agreement was reached and what the agreed-upon value was? Explain why your procedure is correct even if there are still active leaders executing this instance of Paxos.

Solution: The best solution: The server should send messages to all servers asking whether they accepted an ACCEPT value, and if so, what value. If a majority respond with the same value, then the server knows that agreement was achieved. The procedure is safe even with active leaders, because a majority is a majority – the leader MUST hear from at least one other node about the value that was agreed upon.