UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

FINAL EXAMINATION, December, 2013
Fourth Year – Electrical
ECE454H1 - Computer Systems Programming
Calculator Type: 2
Exam Type: X
Examiner – D. Yuan

Please Read All Questions Carefully! Please keep your answers as concise as possible. You do not need to fill the whole space provided for answers.

The key assumptions for each question are provided in the question statement. If you feel you need to make more assumptions, write them down.

*There are **21** total numbered pages, **8** Questions.*
You have 2.5 hours. Budget your time carefully!

Please put your FULL NAME, UTORid, Student ID on THIS page only.

Name: _____

UTORid: _____

Student ID: _____

Grading Page

	Total Marks	Marks Received
Question 1	12	
Question 2	15	
Question 3	12	
Question 4	13	
Question 5	16	
Question 6	9	
Question 7	11	
Question 8	12 (and 4 bonus marks)	
Total	100	

Question 1 (12 marks, 2 marks each): True or False *(No explanations needed)*

(a) `gprof` can generate exact counts of the number of times each statement is executed.
□True □False

(b) Spatial locality, instead of temporal locality, is the main reason for Facebook to use Memcached to cache the MySQL data.
□True □False

(c) With transactional memory, programmers write code like if they are using fine-grained lock, and enjoy the good performance as if coarse-grained locks were used.
□True □False

(d) Latency to hard disk is still shorter than network latency within today's data center.
□True □False

(e) On 64-bit Linux, even if malloc returns a pointer `p` that is 16-byte aligned, the physical address of p might not be 16-byte aligned because Linux sometimes maps an aligned virtual address to an unaligned physical address.
□True □False

(f) The mark and sweep garbage collection algorithm is efficient because it scans the entire heap only once to perform "mark" and "sweep".
□True □False

Question 2 (15 marks): Dynamic memory

Consider the implementation of malloc and free as we discussed in the lecture.

(a)(3 marks) How you know how much memory to free given just a pointer?

(b)(2 marks) On a 32-bit machine, how many bytes do you need to store a header or a footer block? Why?

(c)(2 marks) Why do you need the boundary tag (i.e., footer block)?

(d)(2 marks) What is the problem with only using implicit list? How does explicit list solve it?

For segregated-list, one implementation is to keep all blocks in a list to have the same size N. Now with this design, answer the following questions:

(e)(2 marks) The advantage of this approach is fast allocation when compared to a segregated-list design where each list can hold blocks with different sizes. Why?

(f)(2 marks) What is the disadvantage of this approach, compared to a segregated-list design where each list can hold blocks with different sizes? Why?

(g)(2 marks) How do you choose the difference size of Ns for the fastest allocation time?

Question 3 (12 marks): False sharing

(a)(3 marks) What is false sharing?

(b)(3 marks) Why it is important to avoid false sharing on multicore machines?

(c)(3 marks) How to avoid false sharing?

(d)(3 marks) Will false sharing still cause problem when running on a single core machine? Assume the cache write policy is “write-back” with write-allocation.

Question 4 (13 marks): Optimizations in practice

Consider the following code pattern that was discussed during the guest lecture:

```
/* Version 1 */
int DEBUG_FLAG; // global variable

void main(int argc, char *argv) {
  DEBUG_FLAG = atoi(argv[1]);
  .. ..
  if (DEBUG_FLAG) {
    printf(“some debug log messages”);
  }
  ..
}
```

“DEBUG_FLAG” here is to control whether the program is running in debug mode. If so, it will print additional debug messages in many code locations. This flag is provided as a command-line argument. This pattern is quite common in many real-world applications to enable debug mode -- the only difference is that instead of reading the flag from command line, real-world applications often read this value from a configuration file.

However, in most production settings, the users will always run the program with DEBUG_FLAG set to 0. After taking ECE454, you know that you could further optimize this program by converting DEBUG_FLAG from a global variable into a macro, and rewrite the above code as the one below:

```
/* Version 2 */
void main(int argc, char *argv) {
  .. ..
#ifdef DEBUG_FLAG
    printf("some debug log messages");
#endif
  ..
}
```

Now, this DEBUG_FLAG is enabled/disabled at compile time. For example, with GCC, if you want to compile a debug build with DEBUG_FLAG enabled, you can compile the program as follow:

```
gcc -D DEBUG_FLAG program.c
```

If you want to compile a production build and disable the DEBUG_FLAG, simply invoke GCC without the "-D DEBUG_FLAG".

Now you find that version 2 actually executes much faster than version 1 during production setting (where debug mode is disabled).

(a)(6 marks) List two reasons that likely contribute the most to the huge speed-up achieved in version 2.

(b)(3 marks) Why the compiler cannot do this optimization for you (i.e., remove the "`if (DEBUG_FLAG)`" in version 1)?

(c)(4 marks) Despite the potential performance saving, many real-world, widely-used programs actually use the code pattern as in version 1 to enable/disable debug mode. This is a case where sometimes other properties are more important than performance. Can you think of any practical advantages of using version 1 over version 2 for enable/disable debug mode?

Question 5 (16 marks): Parallel architecture

Now it's time to expand your consulting firm "OptsRus". Your job now is to meet with different clients and build computer systems that best suit their need. The main task is to choose between two processors: a 48-core Intel-Xeon and a 48-core AMD-Opteron similar to the ones we discussed in the lecture.

Below are the specifications for these two processors:

	AMD Opteron	Intel Xeon
# of dies	6	6
Cores per die	8	8
Local L1 latency	3 cycles	5 cycles
Local L2 latency	15 cycles	11 cycles
RAM access latency	136 cycles	215 cycles

The table below shows the latencies (cycles) of the cache coherence to perform load/store/test-and-set on a cache line depending on the MESI state and the distance.

System	AMD Opteron			Intel Xeon		
State\Hops	same die	one hop	two hops	same die	one hop	two hops
loads						
Modified	81	172	252	109	289	400
Exclusive	83	175	254	92	273	383
Shared	83	176	254	44	223	334
Invalid	136	237	327	335	492	601
stores						
Modified	83	191	273	115	320	431
Exclusive	83	191	271	115	315	425
Shared	246	286	296	116	318	428
test-and-set						
Modified	110	216	296	120	324	430

Note that in this table, if the cache line is in "Modified" or "Exclusive" state, the owner of this cache line is always the remote core (with the distance being on the same die, one hop, or two hops). A hop is the interconnect hop between dies. The read latencies for "Shared" state indicates the cache line is not present in the local cache, but is present as "Shared" state in a remote cache. The write latency on "Shared" state indicates the local cache contains a copy of the cache line that is shared with other remote cores (either only within the same die or also with cores on other dies).

Assume all the other specifications that are not listed above are the same between the two processors, including CPU clock speed, L1/L2 cache size, ISA, cache-line size,

cost-per-core, etc. Assume there are only two levels of cache. Your job is only to choose between processors, and you can assume other components, such as motherboard, RAM, disk, etc., are the same.

Now given each type of the workload below, which one of the two processors is a better choice? Briefly explain each of your answer (no marks if there is no explanation on your choice). Note: assume the programmers are aware of the underlying architecture, and appropriate optimizations are applied (such as pinning threads to the cores on the same die).

(a)(2 marks) Single thread streaming application that linearly scans a large data-structure in the memory. Each byte is used only once (i.e., no temporal locality).

(b)(2 marks) Single thread matrix multiplication, where the entire matrices fit in L1 cache.

(c)(3 marks) Twitter workload, where a tweet is read once by each thread. In particular, assume the following pattern: thread 1 first reads the tweet, then thread 2, 3, .. N all read the same tweet one by one. Since the tweets update very frequently, very soon there will be a tweet available in RAM, and the threads repeat this read-sequence. No synchronization is used. Assume the number of thread, N, is 8.

(d) (3 marks) Assume the same twitter workload as in (c), only now that N equals to 16.

(e) (3 marks) Shared counter: N threads sharing a counter, where all threads will periodically read the counter's value. Quite frequently, one thread will increment the counter. Assume no synchronization is used (bad programming practice though). Assume counter reads occur more frequently than the increment. Assume N is 8.

(f) (3 marks) Assume the same workload as in (e), only now that N equals to 16. Assume that after a counter update, the first read is always from a core on the same die.

Question 6 (9 marks): MapReduce

Write a MapReduce program that computes the reverse web-link. The input of this program is the crawled html webpages, given as the key-value pairs: <URL, filesystem path> where the key is the URL of a crawled webpage, and the value is the file system path to the crawled html webpage. Output should be in the format of <target, *list*(source)> where the each source contains a link to the target (both target and source are URLs).

For example, if there are three webpages, nba.com, espn.com, wikipedia.org, and both espn.com and wikipedia.org link to nba.com, then you should output a key-value pair: `<nba.com, (espn.com, wikipedia.org)>`

Reverse web-link is the building block for Google's pagerank.

Now, given the semantic of map and reduce as we discussed in the lecture, please write a C-style MapReduce program to compute the reverse web-link. Assume you can use the following libraries:

`FILE *fopen (char* filename, char* mode)`: open a file for read (mode = "r"), or write (mode = "w"). Returns a file handler pointer.

`void fclose (FILE *fp)`: close the opened file.

`char* next_link (FILE* fp)`: parse and return the next hyperlink URL in the webpage pointed by fp (assume `next_link` will automatically allocate the output char* in the heap and you do not need to worry about freeing it). `null` is returned if no more hyperlinks can be found.

`void emit_intermediate (void* key, void* value)`: this is to be used by the map function, to emit the intermediate key-value pairs.

`void emit_output (char* key, list<char*> value)`: this is to be used by the reduce function, to emit the final reverse web-link output: <target, list(source)>.

You don't need to worry about the error returns from the above functions.

Assume that map will be invoked on every input key-value pair by the underlying system.

(a)(6 marks) Write your map and reduce program below.

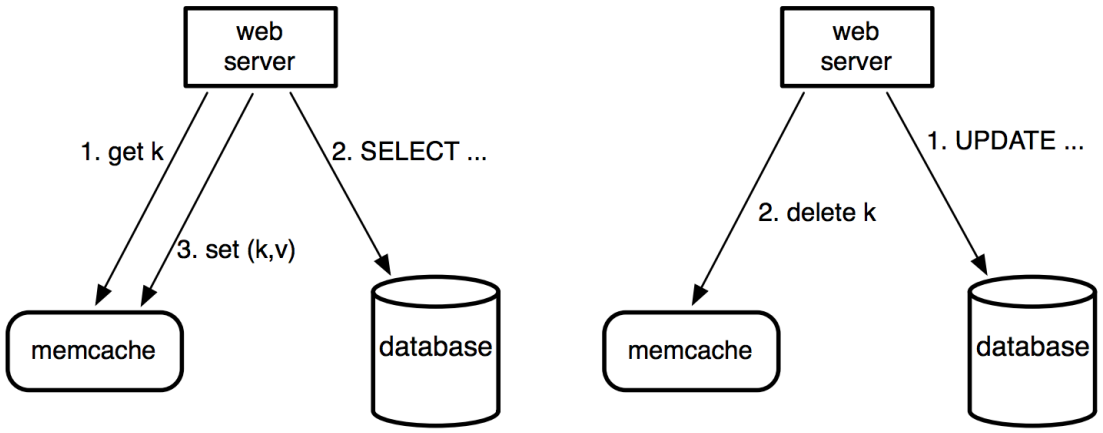
```
map(char* url, char* path) {
/* key: webpage url
 * value: path to the webpage that
 *  is stored on local filesystem.
 */









}
```

```
reduce(_____ _________, // key
       list<_______> l // value
      ){
/* key: intermediate keys from map's
 *      output
 * value: the list containing all
 *      the values of the same
 *      key from map's output
 * fill in the blank above to
 * complete the function declaration
 */



 foreach v in l {
// This loop is to iterate through
// every value in the list l,
// you may or may not need this loop
// comment it out if you don't.









}
```

(b)(3 marks) If there is one mapper machine that is extremely slow (called straggler), how will that impact the overall performance of the entire mapreduce job? How does Google's MapReduce system handle such cases?

Question 7(11 marks): Memcache@Facebook

The figure below shows how Facebook implements memcache.



The left half illustrates the sequence of actions taking place on a memcache miss. The right half illustrates the sequence of actions for a write.

(a)(3 marks) Why does Facebook need memcache?

(b)(4 marks) If Facebook chooses to update the key in the memcache on a write request, instead of simply deleting it as shown in the right half of the above figure, it might result in inconsistencies between memcache and the database. How can this occur, and why delete instead of update can solve it?

(c)(4 marks) Even with the implementation as shown in the above figure (where the key is deleted from memcache upon a write), the problem of inconsistent values between the database and memcache can still occur. How can this problem still occur? (assume there is only one data center).

Question 8 (12 marks): Threads and synchronization

(a)(4 marks) Consider the following implementation of “test_and_test_and_set” lock as we discussed in the lecture:

```
void lock_acquire (lock) {
   while (lock->held == 1)
      ; // spin
   test_and_set(lock->held); // test_and_set atomic instruction
}
void release (lock) {
   lock->held = 0;
}
```

Does it work? Why?

(b)(4 marks) Consider the producer/consumer problem: there are ***multiple*** producer threads and ***multiple*** consumer threads. All of them operate on a shared buffer. A producer inserts data into the buffer one item at a time, and a consumer removes data from the buffer one item at a time. The correctness criteria is that the producers cannot insert into a buffer that is full, and consumers cannot remove any items from an empty buffer.

Now consider the following code:

```
void *producer(void *arg) {
  while (true) {
    pthread_mutex_lock(&mutex);
    while (number_of_items(buffer) == MAX) // MAX is the buffer length.
      pthread_cond_wait(&cv, &mutex);
    insert_item(buffer); // inserts an item into shared buffer
    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mutex);
  }
}
```

```
void *consumer(void *arg) {
  while (true) {
    pthread_mutex_lock(&mutex);
    while (number_of_items(buffer) == 0)
      pthread_cond_wait(&cv, &mutex);
    tmp = consume_item(buffer); // removes an item from the shared buffer
    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

The semantics of pthread libraries are the same as we discussed in the lecture. In case you forgot, their semantics are:

`pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex):` Atomically releases the mutex and cause the calling thread to block on cond. Upon successful return, the mutex has been locked and is owned by the calling thread. This function should be called with mutex locked and owned by the calling thread.

`pthread_cond_signal(pthread_cond_t *cond):` Unblocks one thread waiting on cond. The scheduler decides which thread. If no thread waiting, then signal does nothing.

`pthread_cond_broadcast(pthread_cond_t *cond):` Unblocks all threads waiting on cond. If no thread waiting, then broadcast does nothing.

Now, does this code work correctly? Why?

(c)(4 marks) Now let's make a small change to the code above: replace "`pthread_cond_broadcast`" with "`pthread_cond_signal`". The modified code is as below:

```
void *producer(void *arg) {
  while (true) {
    pthread_mutex_lock(&mutex);
    while (number_of_items(buffer) == MAX) // MAX is the buffer length.
      pthread_cond_wait(&cv, &mutex);
    insert_item(buffer); // inserts an item into shared buffer
    pthread_cond_signal(&cv); // here!
    pthread_mutex_unlock(&mutex);
  }
}

void *consumer(void *arg) {
  while (true) {
    pthread_mutex_lock(&mutex);
    while (number_of_items(buffer) == 0)
      pthread_cond_wait(&cv, &mutex);
    tmp = consume_item(buffer); // removes an item from the shared buffer
    pthread_cond_signal(&cv); // here!
    pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

Does this code work correctly? Why?

(d) Optional BONUS (4 marks)

Can you write an optimized producer/consumer code that is both correct and faster than the code in (b) and (c)? *The synchronization libraries you can use are limited to the ones used above (i.e., "pthread_cond_wait", "pthread_cond_signal", "pthread_cond_broadcast", "pthread_mutex_lock/unlock"). No other synchronizations, such as semaphores, are allowed.* Write your code below and briefly explain why the performance is better:

```
void *producer(void *arg) {

 while (true) {









   insert_item(buffer);









}
```

```
void *consumer(void *arg) {

 while(true) {









  tmp = consume_item(buffer);









 printf("%d\n", tmp);
}
```

This page is blank. You can use it as an overflow page for your answers.