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 (dynamic mem.)	27
Question 3	12 (false share, parallel)	39
Question 4	14 (compiler optimize)	53
Question 5	16 (parallel arch)	69
Question 6	8 (MR)	77
Question 7	11 (FB)	88
Question 8	12 (and 5 bonus marks)	100
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 do you know how much memory to free given just a pointer?
Answer: use a header block to store the size of each allocated memory chunk. The header block locates right before the allocated memory chunk.

Grading: full marks if they mention "header" block, or the block before "p" -- the returned pointer from malloc. The key here is this block needs to locate before the "p". Still full mark if they mention "footer" block or "boundary tag", but gave the clear explanation that this block is the one before "p". Otherwise 0 marks.

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

Answer: we need 4 bytes per header/footer block. The reason is that the heap may be as large as close to 4GB (size of virtual memory), therefore we need 4 bytes.

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

Answer: footer block is used when coalescing with the previous block --- from the current block we can easily locate the boundary tag of the previous block, which can take us to the start of the block and we can coalesce.

(d)(2 marks) What is the problem with only using implicit list? How does explicit list solve it?
Answer: the problem is malloc requires linear scan of the entire heap in the worst case. Explicit list is to speed up malloc by "indexing" all the free-blocks with a doubly linked list.

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?

Answer: no search is needed within a list. Since all the blocks in a list have the same size, we can take the first block in a list and return it. If the blocks can have different sizes, we need to scan through all the blocks to find a best fit.

(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?
Answer: the disadvantage is internal fragmentation -- waste of memory.

(g)(2 marks) How do you choose the difference size of Ns for the fastest allocation time?
Answer: choose the N be the multiple of 2 -- 2, 4, 8, 16, … Now given a malloc request with size M, simply bit-shift M to find its most significant bit "1" to determine the fitting N.

Question 3 (12 marks): False sharing

(a)(3 marks) What is false sharing?
Answer: multiple threads's non-shared data colocate in the same cache line (64 bytes). Although different threads only access its own data-structure, but since they're in the same cache line, they were effectively as shared data.

(b)(3 marks) Why is it important to avoid false sharing on multicore machines?
Answer: because they cause unnecessary coherence traffic and cache misses. Any write performed by one thread on its own data from one core will effectively invalidate the other thread's cache line.

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

Answer: introduce padding between the two adjacently-declared variables if they would be used by different threads, or move their declarations apart from each other.

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

Answer:
No. Because there is only one cache. If one thread writes its data, it will not invalidate the cache line (since it's write-back), so during a context-switch, the other thread will still see the shared cache line in the cache.

Question 4 (14 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.
Answer:

1. branch prediction cost. In version 1, every time it hits the “if (DEBUG_FLAG)”, it’s a branch instruction. If the hardware predicts it wrongly, the penalty can be expensive. In addition, branch prediction works by using a history buffer to store the prediction results of the recent branches. By adding more branches, it can increase the pressure of this buffer and reduce its effectiveness on other branches.

2. Increase the code cache size, and can cause code-cache misses.

Grading: 3 marks each. For 1, give 1 mark if they only mentioned the additional overhead branch, but not the branch misprediction. A correctly predicted branch instruction won’t add much overhead.

(b)(4 marks) Why the compiler cannot do this optimization for you (i.e., remove the “`if (DEBUG_FLAG)`” in version 1)?

Answer: because the value of DEBUG_FLAG is from the input, and compiler has no way to know that it’s always set to false in production mode (and it will be dangerous for the compiler to perform such optimization).

(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?

Answer: the advantage is that if there is a software failure on the user site, the user can simply rerun the program and enable the debug mode without recompiling the code. If we use version 2, if a user encounters a failure, the software vendor needs to send them a new version of the compiled program, ask them to install it (which can be non-trivial) to collect the debug-mode log. In fact, most of the widely-used software use version 1’s model. This is another example where performance sometimes is not the most important thing.

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 cache line 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. You are only to choose between processors, and you can assume other components, such as motherboard, RAM, disk, etc., can be 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).

Answer: RAM latency will be the dominant factor -- AMD opteron because of the shorter RAM access.

(b)(2 marks) Single thread matrix multiplication, where the entire matrices fit in L1 cache.
Answer: AMD opteron because of the short L1 latency.

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

Answer: The first read by thread 1 will be from RAM (latency: Opteron 136 vs. Xeon 215). The second read is from the “Exclusive” cache line owned by thread 1 (Opteron 83 vs. Xeon: 92). The next 6 reads are from “Shared” cache line (Opteron 83 vs. Xeon: 44).

So if we use Opteron, the total read latency of the first read for each thread on a news update is: 136 + 83 + 6*83 = 717.

If we use Xeon, total read latency will be:
215 + 92 + 6*44 = 571

So we choose Intel Xeon.

(d) (3 marks) Assume the same workload as in (c), only now that N equals to 16.
Answer: similar solution as above: first read is from RAM, 2nd is from “Exclusive” state, 3-8th reads are from Shared state in the same die, the 9th read is from “Shared” state on a remote die (one-hop), and 10-16th reads are again from “Shared” state on the same die.

If we use Opteron, the latency:
136 + 83 + 6*83 + 176 + 7*83 = 1474

If we use Xeon, the latency:
215 + 92 + 6*44 + 223 + 7*44 = 1102

So we should choose Intel Xeon.

(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.

Answer: consider a cycle of operations: increment by thread 1, and all other threads read it. The increment will be a write request on a shared state. In this case, since N = 8, this cache line is shared within the die. Then each thread will eventually read the new data, the first subsequent read will be on “modified” state, where the later reads are on “shared” states.

We ignore the later reads on cache line that already in the local cache (since the latency is too small compared to the latency of coherence traffic).

The total latency for this cycle:
Opteron: 246 (store on shared) + 81 (read on modified) + 6*83 (read on shared) = 825.
Xeon: 116 (store on shared) + 109 (read on modified) + 6*44 = 489.

We should choose intel Xeon.

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

Answer: the only differences from (e) are: A) the write is on shared state with remote cores; B) among the 15 subsequent reads from other threads, the first read is from a

core on the same die, but one of the read will be from a remote die on a Shared cache line.

The total latency for write-15 reads:
Opteron: 286 (store on one-hop shared) + 81 (read on modified) + 172 (read on Shared from different die) + 13*83 (read on shared) = 1618.
Xeon: 318 (store on one-hop shared) + 109 (read on modified) + 223 (read on Shared from one-hop die) + 13*44 (read on shared) = 1222.

We should choose Intel Xeon.

Question 6 (8 marks): MapReduce

Write a MapReduce program that computes the reverse web-link. The input of this program is the raw html webpages, and the output is 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 page rank.

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"), write (mode = "w"), or append (mode = "a"). Returns a file handler pointer.

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

`char* next_link (FILE* fp)`: return the next hyperlink URL in the webpage pointed by fp (assume `next_link` will 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 pair.

`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)(5 marks) Write your map and reduce program below.

```
map(char* url, char* path) {
 // key: webpage url
 // value: path to the webpage that
```

```
reduce(_____ _________, // key
       list<_______> l // value
      ){
```

```
//  is stored on the local file
//  system.
FILE * fp = fopen (path, "r");
char* target;
while (target = next_link(fp)) {
 emit_intermediate(target, url);
}
fclose(fp);









}
```

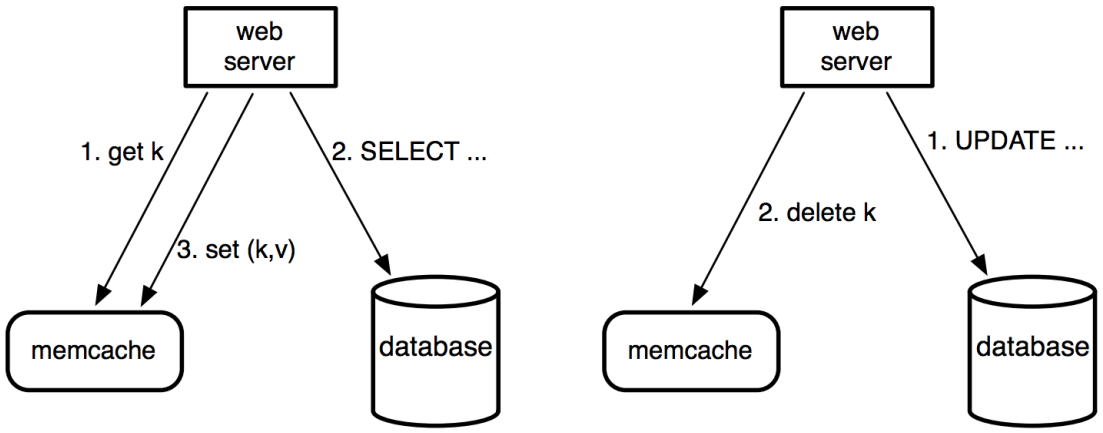
```
// key: intermediate keys from map
// value: a list of intermediate
//   values of the same key from map
      char* target, list<char*> l
foreach v in l {
// This loop is to iterate every
// value in the list l,
// you may or may not need this loop
// comment it out if you don't.

emit_output (target, l);



}
```

(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?

Answer: one slow machine will stall the entire mapreduce job since reduce cannot start before all the mappers finish. The MapReduce handles such cases by spawning backup copies of map tasks near the end of map phase. Whichever one finishes first "wins"

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?
Answer: to reduce the page load latency. Without it, the objects need to be retrieved from database, which is slow.

(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?

Answer: consider two subsequent write requests on the same key. Now if we chose to update the value in memcache, the order of the two write requests received by the database might be different than the order received by memcache, resulting in inconsistency.

If we simply delete the key from memcache on a write, this problem won't matter because the order of delete doesn't matter.

(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).

Answer: a race can still occur between a read-miss and a write on the same key. Consider the following event sequence:

1). [read request] webserver reads key k
2). [read request] memcache doesn't have k (read miss)
3). [read request] webserver reads k from database
4). [write request] webserver receives a write request to k -- first update the database
5). [write request] webserver deletes k from memcache
6). [read request] webserver set k to the old value (read from 3) to memcache

Now there is an inconsistency: the database contains the newly written value for k, but the memcache still holds the old value for k.

Note that this inconsistency is far less probable than the one in (b).

Here is Facebook how to further solve this problem:
https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920

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?

Answer: no. The reason: two threads might both return from lock_acquire and thus enter the critical section. Consider this interleaving (each line is one cycle):

```
value of lock->held      Thread 1                      Thread 2
0                     lock_acquire                  lock_acquire
                  while(lock->held == 1): false  while(lock->held == 1): false
                  test_and_set(lock_held)
1                                                test_and_set(lock_held)
Now both threads return from lock_acquire!
```

(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?

Answer: yes. This code seemingly makes the mistake of only using one condition variable in the producer/consumer problem, but solves it by using broadcast to wake all waiting threads. Because each thread re-checks the condition when awoken, only those who should be able to make progress will do so. Of course, this can be inefficient, which is why we normally use two CVs and signal accordingly thus waking up only those who need to be awoken (solution to part (d)).

(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?

Answer: it no longer works. Consider this case: assume now multiple producers and consumers are waiting on the condition variable, and only one consumer is running. Now at this point, the buffer has only one item left, the consumer consumes it, and wakes up one blocking thread. But since the decision is on the scheduler, it might pick another consumer thread to be waken up. Now this thread wakes up, but find the buffer is empty, so goes back to waiting state. At this point, all threads are waiting, and we reach a deadlock situation.

Optional BONUS (5 marks)

Can you write an optimized version of 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);
}
```

Answer:
The problem we try to address is the inefficiency with only one condition variable (as described in the answer in part b). We solve it by using two condition variables: full and empty.

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


 while (true) {
  pthread_mutex_lock(&mutex);
  while (number_of_items(buffer) == MAX)
    pthread_cond_wait(&full, &mutex);
  insert_item(buffer);
  pthread_cond_signal(&empty);
   // wake up one blocking consumer!
  pthread_mutex_unlock(&mutex);
 }

}
```

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


 while(true) {
  pthread_mutex_lock(&mutex);
  while (number_of_items(buffer) == 0)
    pthread_cond_wait(&empty, &mutex);
  tmp = consume_item(buffer);
  pthread_cond_signal(&full);
  // wake up one blocking producer
  pthread_mutex_unlock(&mutex);
  printf("%d\n", tmp);
 }
}
```

Note: we can use signal now because a producer is guaranteed to wake up a consumer, and vice-versa.

Grading: 2 points for identifying the performance problem with the solution in (b). 1 point for using two condition variables. If used “broadcast” instead of “signal”, reduce 1 point.

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