6.172 Fall 2014

Quiz 2

Given: In class on Thursday, November 20th, 2014

Name: _____________________________________________

CSAIL username: ____________________________________

Instructions

• DO NOT open this quiz booklet until you are instructed to do so.
• This quiz booklet contains 16 pages, including the x86 quick reference at the end.
• You have 80 minutes to earn 100 points.
• This quiz is closed book, but you may use one handwritten, double-sided 8 1/2” ×11” crib sheet.
• When the quiz begins, please write your name and username on this coversheet, and write your name on the top of each page, since the pages may be separated for grading.
• Good luck!

<table>
<thead>
<tr>
<th>Question</th>
<th>Parts</th>
<th>Points</th>
<th>Score</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. True/False</td>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Garbage</td>
<td>3</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Locks</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Looping Algorithm</td>
<td>2</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Divide-and-conquer Algorithm</td>
<td>10</td>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Section 1: True/False

Question 1.1 Programs written in languages such as Java and Python have more predictable runtime program performance because of garbage collection.

T / F

Explain:

False, The effects of garbage collection on runtime is not predictable.

Question 1.2 Ben Bitdiddle created a new function, int matrix_magic(Matrix m) for his company’s top-grossing big data processing package. As matrix_magic is performance critical, Ben wanted to autotune a bunch of parameters in the function using an autotuner. For that, he found the biggest matrix available, and used it as the input to the autotuner. His autotuned matrix_magic function will always be faster than the original matrix_magic without autotuning.

T / F

Explain:

False, Overfitting
Question 1.3 The following code would benefit from a readers-writer lock in place of regular mutexes. (A readers-writer lock allows multiple readers, but requires there be no readers or writers for a write lock to be acquired, thus reducing contention.)

```c
const int TABLE_SIZE = 1007;
const long PRIME = 100000007L;
struct HashTable {
    Mutex mutexes[TABLE_SIZE];
    LinkedList *buckets[TABLE_SIZE];
};
HashTable table;

void hashtable_init() {
    // initialize mutexes and buckets
}

void hashtable_insert(int value) {
    int entry = (value * PRIME) % TABLE_SIZE;
    lock(&mutexes[entry]);
    buckets[entry]->insert(value);
    unlock(&mutexes[entry]);
}

void hashtable_destroy() {
    // destroy mutexes; free buckets
}

int main() {
    hashtable_init();
    cilk_for (int i = 0; i < 1000000; i += 15) {
        hashtable_insert(i * 3 + 7);
    }
    hashtable_destroy();
    return 0;
}
```

T / F

Explain:

False, there are only writes to the hashtable, so a readers-writer lock would be equivalent to a normal lock
Question 1.4 The following code has a concurrency bug.

```c
bool blah(int *list, int len, int p, int x) {
    int *fizz = (int *)malloc(p * sizeof(int));
    cilk_for (int i = 0; i < len; i++) {
        fizz[i % p]++;
    }
    bool r = fizz[x] > 0;
    free(fizz);
    return r;
}
```

T  /  F

If true explain the bug and if false explain how he concurrency works without a problem:

```
False, there is a benign race, but that doesn't affect the correctness. If anything writes to a specific slot, it will be at least one, which is what the condition checks.
```
Question 1.5 There is some input to handle_transactions below that will cause a deadlock.

```c
struct Bank {
    int num_accounts;
    int *balances; // one for each account
    Mutex *account_mutexes; // one for each account
}
Bank bank;

struct Transaction {
    int from_index;
    int to_index;
    int amount;
}

void handle_transaction(Transaction *trans) {
    lock(&bank.account_mutexes[trans->from_index]);
    lock(&bank.account_mutexes[trans->to_index]);

    bank.balances[trans->from_index] -= trans->amount;
    bank.balances[trans->from_index] += trans->amount;

    unlock(&bank.account_mutexes[trans->to_index]);
    unlock(&bank.account_mutexes[trans->from_index]);
}

void handle_trasactions(Transaction *transactions, int len) {
    cilk_for (int i = 0; i < len; i++) {
        handle_transaction(&transactions[i]);
    }
}
```

T / F

If true, explain why and how to fix it. If false, explain:

Answer: True, there is no global ordering for taking locks here. For example, the input 
[(1, 2, $50), (2, 1, $50)] could result in two threads each acquiring a lock 
(the first grabbing lock 1, the second grabbing lock 2), and then waiting forever 
to acquire the next lock. To fix this, lock first on the minimum (or maximum) of 
the two accounts, and then lock on the second.
Section 2: Storage Allocation

Question 2.1 Garbage collection using reference counting has the following advantages. (select all that apply)

A. It does not incur instruction overhead
B. It does not incur memory overhead
C. It can handle data that forms cycles
D. Objects are reclaimed as soon as their count drops to zero.

Answer: D. It maintains reference count of each heap-allocated object, which is memory overhead. A cycle is never garbage collected.

Question 2.2 After analyzing the memory request trace of a program, Ben Bitdiddle implements a fixed-size memory allocator that allocates and frees 128-byte objects. His allocator takes a 4096-byte page of memory and splits it into blocks of size 128 bytes. It uses 96 bits at the beginning of the block for bookkeeping. To keep track of which blocks are free, it uses a bitmap placed in the bookkeeping area at the beginning of the page.

Ben now decides to adapt his allocator to work in a multithreaded environment with exactly two threads. He does this by splitting the blocks on the page into two sets such that each thread allocates from its own half. He also splits the bitmap in half. He pads each half of the bitmap to 64 bits so that the two threads can update the two halves independently. Nevertheless, Ben’s allocator suffers from poor performance. Which of the following explanations are most likely the reasons for Ben’s poor performance? (Select all that apply)

A. Poor space utilization
B. External fragmentation
C. True sharing of the bitmap
D. False sharing of the bitmap
E. None of the above

Answer: B, D, the bitmaps are on the same cache line.
Question 2.3 After reasoning about the problems with his previous approach, Ben decides to move to a version of the Hoard allocator, where each thread owns its own heap. He runs a set of benchmarks using a single thread, and observes good performance. When run with two threads, however, one of the benchmarks shows poor space utilization. That benchmark contained a predefined list of allocations and frees that are performed in a synchronized manner. He observed that most data allocated from one thread will be freed by the other. Which of the following explanations is most likely to be the cause of poor space utilization? (select all that apply)

A. Internal fragmentation
B. External fragmentation
C. Memory drift
D. The superblock size is poorly chosen.
E. None of the above

Answer: C
Section 3: Locks

Question 3.1 Multiple threads execute the following code. Assume that the compiler is unable to optimize away the inner loop. Choose the type of lock that would yield the best performance:
(Select the best answer)

```c
void incr(int * x) {
    for (int i = 0; i < 100000; i++) {
        lock();
        *x++;
        unlock();
    }
}
```

A. Hybrid Lock  
B. Spin Lock  
C. Reentrant Lock  
D. Yielding Lock

Answer B

Question 3.2 Multiple threads execute the following code. Assume that the compiler is unable to optimize away the inner loop. Choose the type of lock that would yield the best performance:
(Select the best answer)

```c
void incr(volatile int * x) {
    lock();
    for (int i = 0; i < 1000000000; i++) { // 1 billion
        *x++;
    }
    unlock();
}
```

A. Hybrid Lock  
B. Spin Lock  
C. Reentrant Lock  
D. Yielding Lock

Answer D
Question 3.3 Multiple threads execute the following code. Assume that the compiler is unable to optimize away the inner loop. Choose the type of lock that would yield the best performance:
(Select the best answer)

```c
void incr(int * x) {
    int y = random_bit() // returns 1 or 0 with equal probability
    int limit = 100;
    if (y == 0)
        limit = 1000000000 // one billion
    lock();
    for (int i = 0; i < limit; i++) {
        *x++;
    }
    unlock();
}
```

A. Hybrid Lock  
B. Spin Lock  
C. Reentrant Lock  
D. Yielding Lock

Answer A
Caching, Cache-Efficient Algorithms, Cache-Oblivious Algorithms

In this problem, we shall explore the “least-weight subsequence” problem, which is defined as follows. Suppose that we have a weight function \( w(i,k) \) on pairs of integers \( i \) and \( k \). For a sequence \( i_0, i_1, \ldots, i_{t-1} \) of \( t \) strictly increasing integers, the weight of the sequence is the sum of the weights of the adjacent pairs, that is, \( \sum_{s=1}^{t-1} w(i_{s-1}, i_s) \). The least-weight subsequence problem is to find the subsequence of \( 0, 1, \ldots, n - 1 \) starting at 0 and ending at \( n - 1 \) with minimum weight.

The least-weight subsequence problem can be solved using dynamic programming. For a given weight function, the weight \( lw(n) \) of the least-weight subsequence \( 0, 1, \ldots, n - 1 \) can be computed using the following recurrence relation:

\[
\begin{align*}
lw(i) &= 0 \text{ if } i = 0, \text{ and} \\
lw(i) &= \min_{0 \leq k < i} \{ lw(k) + w(k, i) \} \text{ otherwise.}
\end{align*}
\]

In this problem, we shall focus on computing the weights \( lw \). (To compute least-weight subsequence itself, you maintain back pointers while computing these weights. We omit that code here for simplicity.) We shall investigate the cache complexity of two algorithms - a simple looping algorithm, and a divide-and-conquer algorithm - that implement this dynamic program. Let \( M \) be the size of the cache, and let \( B \) be the size of a cache line.

The following codes store the array of \( n \) least-weight-subsequence weights in \( lw \), using the weight function \( w() \). Assume that \( w() \) has work \( \Theta(1) \) and cache complexity 0, that is, assume that \( w() \) does not access memory.
Section 4: Looping algorithm

The following code implements the looping algorithm to compute lw.

```c
int lw_looping(int *lw, int n) {
    lw[0] = 0;
    for (int r = 1; r < n; ++r) {
        lw[r] = lw[0] + w(r, 0);
        for (int i = 1; i < r; ++i) {
            if (lw[i] + w(r, i) < lw[r]) {
                lw[r] = lw[i] + w(r, i);
            }
        }
    }
    return lw[n-1];
}
```

**Question 4.1** Argue that the work of `lw_looping()` is \( \theta(n^2) \)

The inner loop executes \( r \) iterations, each of which performs a constant amount of work. The other loop executes \( n \) iterations of the inner loop. The total work is therefore \( \sum_{r=1}^{n} \sum_{i=1}^{r} \theta(1) = \theta(n^2) \)

**Question 4.2** Assume that \( n >> M \). Argue that the number of cache misses incurred by `lw_looping()` is \( \theta \left( \frac{n^2}{B} \right) \).
Section 5: Divide-and-conquer algorithm

The code for lw_dac() below implements a divide-and-conquer algorithm to compute lw. This lw_dac() code calls a recursive helper function, lw_dac_h_same(), which itself calls another recursive helper function, lw_dac_h_disjoint(). We shall analyze the cache complexity of lw_dac() by first analyzing lw_dac_h_disjoint() and then analyzing lw_dac_h_same().

Let us start by analyzing the cache complexity $Q_d(n)$ of lw_dac_h_disjoint(), the code for which is as follows.

```c
static void lw_dac_h_disjoint(int *lw,
   int start_l, int end_l,
   int start_h, int end_h) {
   if ((end_l - start_l) < BASECASE_SZ && (end_h - start_h) < BASECASE_SZ) {
      // Base case
      for (int r = start_h; r < end_h; ++r) {
         for (int i = start_l; i < end_l; ++i) {
            if (lw[i] + w(r, i) < lw[r]) {
               lw[r] = lw[i] + w(r, i);
            }
         }
      }
   } else {
      // Recursive calls
      int mid_l = start_l + (end_l - start_l) / 2;
      int mid_h = start_h + (end_h - start_h) / 2;
      lw_dac_h_disjoint(lw, start_l, mid_l, start_h, mid_h);
      lw_dac_h_disjoint(lw, mid_l, end_l, start_h, mid_h);
      lw_dac_h_disjoint(lw, start_l, mid_l, mid_h, end_h);
      lw_dac_h_disjoint(lw, mid_l, end_l, mid_h, end_h);
   }
}
```

Assume that $n >> M$ and that BASECASE_SZ < cM for some sufficiently small constant $c < 1$.

**Question 5.1** Argue that $Q_d(n)$ satisfies the following recurrence:

$Q_d(n) = \Theta(N/B)$ if $n < cM$ for some sufficiently small constant $c < 1$, and

$Q_d(n) = 4 Q_d(n/2) + \Theta(1)$ otherwise.
For the recursion tree for $Q_d(n)$, please answer the following questions. (Select only one answer for the next 3 questions)

**Question 5.2 What is the height of the tree?**

A. $n$
B. $n - cM$
C. $\log_2 n$
D. $\log_2 \left(\frac{n}{cM}\right)$

*Answer D*

**Question 5.3 What is the total number of cache misses that occur in its leaves of the tree?**

A. $\Theta(n^2)$
B. $\Theta\left(\frac{n^2 M}{B}\right)$
C. $\Theta\left(\frac{n^2}{M^2}\right)$
D. $\Theta\left(\frac{n^2}{MB}\right)$

*Answer D*

**Question 5.4 What is the number of cache misses that occur at each of the intermediate levels of the tree?**

A. Level i incurs $\Theta(1)$ total misses
B. Level i incurs $4^i \cdot \Theta(1)$ total misses
C. Level i incurs $\Theta\left(\frac{n}{B}\right)$ total misses
D. Level i incurs $4^i \cdot \Theta\left(\frac{n}{B}\right)$ total misses

*Answer B*

**Question 5.5 Argue that the cache complexity $Q_d(n)$ of lw_dac_h_disjoint() is $\Theta\left(\frac{n^2}{MB}\right)$.**
Now we analyze the cache complexity $Q_d(n)$ of the $\text{lw_dac_h_same()}$ helper function, which calls $\text{lw_dac_h_disjoint()}$, and the cache complexity $Q(n)$ of the top-level $\text{lw_dac()}$ function, which calls $\text{lw_dac_h_same()}$. The codes for $\text{lw_dac_h_same()}$ and $\text{lw_dac()}$ follow.

```c
static void lw_dac_h_same(int *lw, int start, int end) {
    if ((end - start) < BASECASE_SZ) {
        // Base case
        for (int r = start; r < end; ++r) {
            for (int i = start; i < r; ++i) {
                if (lw[i] + w(r, i) < lw[r]) {
                    lw[r] = lw[i] + w(r, i);
                }
            }
        }
    } else {
        // Recursive calls
        int mid = start + (end - start) / 2;
        lw_dac_h_same(lw, start, mid);
        lw_dac_h_disjoint(lw, start, mid, mid, end);
        lw_dac_h_same(lw, mid, end);
    }
}

int lw_dac(int *lw, int n) {
    lw[0] = 0;
    for (int r = 1; r < n; ++r) {
        lw[r] = lw[0] + w(r, 0);
    }
    lw_dac_h_same(lw, 1, n);
    return lw[n-1];
}
```

Assume that $n >> M$ and that $\text{BASECASE_SZ} < cM$ for some sufficiently small constant $c < 1$. 
Question 5.6 Argue that $Q_s(n)$ satisfies the following recurrence:

$Q_s(n) = \Theta(n/B)$ if $n < cM$ for some sufficiently small constant $c < 1$, and

$Q_s(n) = 2 Q_s(n/2) + 2 Q_d(n/2) + \Theta(1)$ otherwise.

Answer the following three questions about the recursion tree for $Q_s(n)$. Select only one answer.

Question 5.7 What is the height of the tree?

A. $n$

B. $n - cM$

C. $\log_2 n$

D. $\log_2(n/cM)$

Answer D

Question 5.8 What is the total number of cache misses that occur in the tree’s leaves?

A. $\Theta(n)$

B. $\Theta(n/M)$

C. $\Theta(nM/B)$

D. $\Theta(n/B)$

Answer D

Question 5.9 What is the number of cache misses that occur at the intermediate level $i$, in terms of $Q_d(n)$?

A. Level $i$ incurs $\Theta(1) + Q_d(n/2^{i+1})$ total misses.

B. Level $i$ incurs $\Theta(n/B) + Q_d(n/2^{i+1})$ total misses.

C. Level $i$ incurs $2^i \cdot \left(\Theta(1) + Q_d(n/2^{i+1})\right)$ total misses.

D. Level $i$ incurs $\Theta(n/B) + 2^i \cdot Q_d(n/2^{i+1})$ total misses.

Answer C
**Question 5.10** Argue that the cache complexity $Q_s(n)$ of `lw_dac_h_same()` is $\theta \left( \frac{n^2}{MB} \right)$ and thus the cache complexity $Q(n)$ of `lw_dac()` is also $\theta \left( \frac{n^2}{MB} \right)$. 