Concurrency, Caching, Storage Allocation, Architecture, and Memory Consistency

Name: ________________________________

CSAIL username: __________________________

Instructions

• DO NOT open this quiz booklet until you are instructed to do so.

• This quiz booklet contains 20 pages, including this one. 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 and the Master Method card handed out in lecture.

• When the quiz begins, please write your name and CSAIL username on this coversheet. In addition, write your name on the top of each page, since the pages may be separated for grading.

• Some of the questions are true/false, and some are multiple choice. You are not required to explain these answers unless you are explicitly asked to do so, although including some justification for your answers maximizes your potential for partial credit. Since incorrect answers for these kinds of questions will be penalized, do not guess unless you are reasonably sure.

• 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>0</td>
<td>20</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1 True or False (11 parts, 22 points)

Indicate whether each statement below is true or false by circling the correct answer. Remember, you need not explain your answers, but since wrong answers will be penalized, do not guess unless you are reasonably sure.

1.1
Peterson’s algorithm for 2-thread mutual exclusion ensures freedom from starvation.

True  False

1.2
Any mutex for \( n \) threads which is implemented using a Compare-and-Swap (CAS) atomic operation requires \( \Omega(n) \) space.

True  False

1.3
Suppose that a Cilk program behaves deterministically when executed on a single processing core. If each concurrent access to shared data in the program is protected by the same mutex, then the program continues to behave deterministically when executed in parallel.

True  False
1.4
Explicit memory fences are unnecessary on a processor that provides sequentially consistent memory.

True  False

1.5
A memory fence is expensive, because it prohibits all other processing cores in the system from using the memory system until the core executing the fence has committed all of its outstanding stores.

True  False

1.6
If each worker executing a Cilk program allocates from and frees to a local heap, then the maximum space blowup relative to a serial execution is bounded by $P$, where $P$ is the number of workers.

True  False

1.7
An advantage of a parallel memory allocator using only local heaps is that it guarantees no false sharing.

True  False
Professors de Lancie and Barker are working independently on fixed-size serial heap allocation. Professor de Lancie implements a linked list which supports two $O(1)$-time operations: adding an element to the tail, and removing an element from the head. Professor Barker implements a linked list which also supports two $O(1)$-time operations: adding an element to the head, and removing an element from the head. The professors use their respective data structures as free lists in their respective allocators.

1.8
Professor Barker’s implementation will likely run faster than Professor de Lancie’s implementation due to increased temporal locality.

True  False

1.9
Professor de Lancie’s implementation will likely use less space than Professor Barker’s implementation due to decreased external fragmentation.

True  False

1.10
Running a garbage collector that uses reference counting will generally free more space than a garbage collector that uses marking with breadth-first search.

True  False

1.11
Autotuning a program by training on less data reduces the risk of overfitting, although performance may sometimes suffer.

True  False
2  Multiple Choice (5 parts, 25 points)

Circle the letters of all correct solutions to each question, and cross out the letters of all incorrect solutions with an X. For full credit, each letter should be either circled or X’ed. Remember (last warning), wrong answers will be penalized.

2.1

The x86 memory-consistency model (TSO) has the following properties:
A  Allows a load to address B to be reordered before a load to a different address A.
B  Allows a load to address B to be reordered before a store to a different address A.
C  Allows a store to address B to be reordered before a load to a different address A.
D  Requires that all stores are committed in program order, regardless of address.
E  Requires that all stores occurring earlier in program order be committed prior to issuing any Compare-and-Swap (CAS) instruction.

2.2

What advantages does a set-associative cache have over a direct-mapped cache of the same total capacity?
A  Fewer cold misses.
B  Fewer capacity misses.
C  Less false sharing.
D  Fewer conflict misses.
E  Lower latency.
2.3

Which phenomena could cause the following code snippet to use the L1-cache ineffectively on a 12-core machine?

```c
#define NUM_CHUNKS 12
int sums[NUM_CHUNKS];

void compute_sum(int chunk_id) {
    sums[chunk_id] = 0;
    for (int i = 0; i < 5000; i++) {
        sums[chunk_id] += i;
    }
}

int main() {
    cilk_for (int i = 0; i < NUM_CHUNKS; i++) {
        compute_sum(i);
    }
    return 0;
}
```

A  True sharing.
B  False sharing.
C  Capacity misses.
D  Conflict misses.
E  TLB misses.
2.4

Professor Harrison writes an application that only allocates and frees 1040-byte objects. He has a choice between two allocators. The fixed-sized allocator uses a free list of 1040-byte blocks. The variable-sized allocator uses binned free lists with blocks that are exact powers of 2. What are the likely advantages of the fixed-size allocator over the variable-sized allocator?

A Allocating and freeing are faster.
B Less internal fragmentation.
C Less external fragmentation.
D Less false sharing.
E Fewer TLB misses.

2.5

Consider a dual-core x86 processor with two processing cores, named Albert and Bernard. The two cores concurrently execute the following instructions, operating on volatile variables \( x \) and \( y \), which are both initialized to 0.

\[
\begin{align*}
\text{Albert} & \quad \text{if} \ (x == \ 0) \\
& \quad \quad \quad y = x + 1;
\end{align*}
\]

\[
\begin{align*}
\text{Bernard} & \quad \text{if} \ (y == \ 0) \\
& \quad \quad \quad x = y + 1;
\end{align*}
\]

After Albert and Bernard have both finished executing all of their instructions, which of the following final values of \( x \) and \( y \) are possible?

A \( x = 0, \ y = 1 \).
B \( x = 0, \ y = 2 \).
C \( x = 1, \ y = 1 \).
D \( x = 1, \ y = 2 \).
E \( x = 2, \ y = 2 \).
3 Cache Complexity (5 parts, 25 points)

For this problem, assume that code executes serially on a processor with an ideal cache of size $M$ and cache-block size $B$. Thus, the cache is fully associative, uses the optimal replacement policy, and has sufficient capacity to store $M$ int values. In addition, each cache block can store $B$ int values, and the cache obeys the tall cache assumption: $M = \Omega(B^2)$.

The recursive procedure $\text{binary_search()}$ shown below performs a binary search on the array $\text{bins[]}$, which is given sorted in ascending order. In particular, the procedure returns the index $x$ of the largest element in $\text{bins[]}$ such that $\text{bins}[x] \leq \text{key}$ and $\text{low} \leq x < \text{high}$.

```c
int binary_search(int key, int *bins, int low, int high) {
    if (low <= low + 1) {
        return low;
    } else {
        int mid = (low+high) >> 1;
        if (key < bins[mid]) {
            return binary_search(key, bins, low, mid);
        } else {
            return binary_search(key, bins, mid, high);
        }
    }
}
```

The procedure $\text{histogram()}$ shown below uses $\text{binary_search()}$ to create a histogram of the elements in the array $\text{data[]}$. The $m$ elements in $\text{data[]} are independent random numbers drawn from an unknown distribution. The output array $\text{count[]} is computed as

$$
\text{count}[i] = \begin{cases} 
\{j \in \{0,1,...,n-1\}: \text{bins}[i] \leq \text{data}[j] < \text{bins}[i+1]\} & \text{if } 0 \leq i < n-1, \\
\{j \in \{0,1,...,n-1\}: \text{bins}[i] \leq \text{data}[j]\} & \text{if } i = n-1.
\end{cases}
$$

```c
void histogram(int *data, int m, int *count, int *bins, int n) {
    for (int i = 0; i < m; i++)
        count[binary_search(data[i], bins, 0, n)]++;
}
```
3.1
Give a tight asymptotic upper bound on the work of `histogram(data, m, count, bins, n)`. Express your answer using O-notation.

3.2
For simplicity, assume that m and n are constrained to be powers of 2 and that m > n > M > lg n. How many cache misses does an execution of `histogram(data, m, count, bins, n)` incur in the worst case? Circle the letter of the best answer.

A  $O(m/B)$.
B  $O(m\lg n)$.
C  $O(m\lg(n/M))$.
D  $O((m/B)\lg(n/M))$.
E  $O((m/B)\lg n)$.
F  $O(m)$.
G  None of the above.
The procedure `merge_sort()` sorts the input `A[]` and stores the result in `T[]`.

```c
void merge(int *A, int len_A, int *B, int len_B, int *dest) {
    int *a = A;
    int *b = B;
    int *d = dest;
    while (a < A + len_A && b < B + len_B)
        *d++ = (*a < *b) ? *a++ : *b++;
    while (a < A + len_A)
        *d++ = *a++;
    while (b < B + len_B)
        *d++ = *b++;
}

void merge_sort(int *A, int *T, int low, int high) {
    if (high <= low + 1) {
        *T = *A;
    } else {
        int mid = (low+high) >> 1;
        merge_sort(A, T+mid, mid, high);
        merge_sort(A, A+mid, low, mid);
        merge(A+mid, mid-low, T+mid, high-mid, T);
    }
}
```

3.3

Give a recurrence on the total work `T(n)` of `merge_sort(A, T, 0, n)`. Give a tight asymptotic upper bound (`O`-notation) on the solution to the recurrence.
3.4

How many cache misses does an execution of $\text{merge_sort}(A, T, 0, n)$ incur in the worst case? Circle the letter of the best answer. (*Hint: You may wish to draw a recursion tree.*)

A $O(n/B)$.
B $O(n \lg n)$.
C $O(n \lg (n/M))$.
D $O((n/B) \lg (n/M))$.
E $O((n/B) \lg n)$.
F $O(n)$.
G None of the above.
The procedure `histogram_with_sort()` is functionally equivalent to `histogram()` except that it uses `merge_sort()` as a preprocessing step prior to finding the histogram of the elements in `data[]`.

```c
void histogram_with_sort(int *data, int m,
                         int *count, int *bins, int n) {
    int *sorted_chunk = (int *) malloc(sizeof(int) * n);
    for (int i = 0; i < m/n; i++) {
        merge_sort(data + i*n, sorted_chunk, 0, n);
        for (int j = 0; j < n; j++)
            count[binary_search(sorted_chunk[j], bins, 0, n)]++;
    }
}
```

3.5

For simplicity, assume that \( m \) and \( n \) are constrained to be powers of 2 and that \( m > n > M > \lg n \).

How many cache misses does an execution of `histogram_with_sort(data, m, count, bins, n)` incur in the worst case? Circle the letter of the best answer.

A \( O(m/B) \).
B \( O(m\lg n) \).
C \( O(m\lg(n/M)) \).
D \( O((m/B)\lg(n/M)) \).
E \( O((m/B)\lg n) \).
F \( O(m) \).
G None of the above.
4 Concurrent Hash Table (3 parts, 15 points)

This problem considers the potential hazards of concurrent accesses to a hash table which uses locks to protect shared data from data races. The hash table in question supports resizing when it becomes too full by doubling the number of slots. Thus, as the number of entries in the table grows, the number of slots also grows by successive doubling, and the density of the table is held within constant bounds. We shall present a (buggy) implementation of such a hash table below, and you will analyze it for various concurrency bugs.

Data types

We begin with the definition of two data types:

```c
typedef struct entry_t {
    void *ptr_to_user_data;
    struct entry_t *next;
    key_t key;
} entry_t;

typedef struct hashtable_t {
    entry_t **table;
    size_t lg_size;
    lock_t lock;
    lock_t *slotlock;
} hashtable_t;
```

The hash table stores items of type entry_t, which is a struct with a key and a pointer to user data. In addition, an entry_t includes a next pointer of type entry_t * on line 3, which is used to form a linked list at each slot in the table.

The hash table resolves collisions by chaining, which means that entries hashing to the same slot form a linked list. The hash table has type hashtable_t, which is a struct comprised of four variables. The variable table[] on line 8 is an array of pointers to type entry_t. The element table[i] either points to the head of the linked list associated with the ith slot or is equal to NULL indicating that the slot is empty. The variable lg_size on line 9 stores the value \( \log S \) where \( S \) is the number of active slots in table[].

Locking

Mutual exclusion is provided by the lock() and unlock() functions, which operate on arguments of type lock_t. Each hash table of type hashtable_t has two types of locks: lock on line 10 and a bank of locks in the array slotlock[] on line 11. The purpose of the variable lock is to protect lg_size and table[] during resizing. The purpose of the ith lock slotlock[i] is to protect the linked list pointed to by table[i].
Resizing

The procedure hashtable_resize() is listed below:

```c
#define hash(key, ht) ((key) & ((1 << ht->lg_size) - 1))

void hashtable_resize(hashtable_t *ht) {
    lock(&ht->lock);
    int old_num_slots = 1 << ht->lg_size;
    ht->lg_size++;
    for (int i = old_num_slots; i < (1 << ht->lg_size); i++)
        ht->table[i] = NULL;
    for (int i = 0; i < old_num_slots; i++) {
        lock(&ht->slotlock[i]);
        lock(&ht->slotlock[i+old_num_slots]);
        entry_t *x = ht->table[i];
        ht->table[i] = NULL;
        while (x) {
            int slot = hash(x->key,ht);
            entry_t *next = x->next;
            x->next = ht->table[slot];
            ht->table[slot] = x;
            x = next;
        }
        unlock(&ht->slotlock[i]);
        unlock(&ht->slotlock[i+old_num_slots]);
    }
    unlock(&ht->lock)
}
```

In order to improve cache locality, the array table[] is preallocated for the largest possible size. The large preallocated array is not initialized, however, so that when hashtable_resize() doubles the number of slots, it is responsible for initializing the newly active slots — the upper half of the slots in the now-active region — on lines 20–21. Similarly, the lock array slotlock[] is also preallocated and initialized for the largest possible number of locks (i.e., slotlock[i] protects table[i] for all i).

Once the newly active entries in table[] have been initialized, hashtable_resize() walks all previously active slots. For each one, the slot is marked NULL on line 27, and the existing linked list is walked. Each entry in the linked list is inserted into the correct slot one by one on lines 29–33. Observe that the hash() function masks off the lg_size least-significant bits of the key. Hence, for every entry in the linked list of the i'th slot, the new slot (after resizing) can be either
i or \(i + 2^k\), where \(k\) is the value of \(\log_{10} \text{size}\) prior to the resize operation. Thus, grabbing both the \(i\)th and \((i + 2^k)\)th lock in slotlock[] suffices to migrate the linked list.

**Inserting entries**

The procedure `hashtable_insert()` is provided below:

```c
void hashtable_insert(hashtable_t *ht, entry_t *x) {
    lock(&ht->lock);
    int slot = hash(x->key, ht);
    unlock(&ht->lock);

    lock(&ht->slotlock[slot]);
    x->next = ht->table[slot];
    ht->table[slot] = x;
    unlock(&ht->slotlock[slot]);
}
```

4.1

Is it possible for concurrent calls to `hashtable_insert()` and/or `hashtable_resize()` to deadlock? Explain why or why not.
4.2

Consider an execution of hashtable_resize() which occurs concurrently with arbitrarily many other calls to hashtable_insert(). At the point just prior to calling unlock(&ht->lock) on line 38 in hashtable_resize(), is it possible for any entry in the hash table to be stored in the wrong slot (e.g., an entry e is stored in slot s, yet hash(e->key, ht) does not equal s)? Explain why or why not.

4.3

Is it possible for an entry to be lost due to the procedure hashtable_resize() initializing the upper half of ht->table[] on lines 20–21? Explain why or why not.
5 Memory System Performance (6 parts, 12 points)

The plot below summarizes the performance of six different programs, which can be found on pages 18 and 19. The running time is plotted on the y-axis and $\lg_N$ is plotted on the x-axis, where $\lg_N$ is an input parameter to each program equal to $\lg N$. In particular, each program iterates over one or more arrays of length $N = 2^{\lg N}$ and performs arithmetic on the elements. The number of trials is calibrated such that each data point in the graph corresponds to an execution with approximately the same amount of work. Thus, the difference in running time is primarily attributable to the differences in the cache, TLB, and memory-bandwidth effects. The programs were compiled with GCC using optimization level -O3, and the data was collected on cloud5.csail.mit.edu.
The following serial programs perform arithmetic on arrays of size $N = 2^{\lg N}$. The number of trials is determined such that each invocation does approximately the same amount of work for any value of $\lg N$. For instance, notice that frob1_serial performs 4 times as many trials as frob2_serial and frob3_serial, since these two programs each perform 4 times as much work per iteration as frob1_serial.

```c
void frob1_serial(int *a, size_t lg_N) {
    size_t trials = 1 << (31 - lg_N);
    for (size_t j = 0; j < trials; j++) {
        for (size_t i = 0; i < (1 << lg_N); i++)
            a[i] = a[i] + 5;
    }
}

void frob2_serial(int *a, int *b, int *c, size_t lg_N) {
    size_t N = 1 << lg_N;
    size_t trials = 1 << (29 - lg_N);
    for (size_t j = 0; j < trials; j++) {
        for (size_t i = 0; i < N; i++)
            c[i] = a[i] + b[i] + a[N-i-1] + b[N-i-1] + c[N-i-1];
    }
}

void frob3_serial(int *a, size_t lg_N) {
    int mask = (1 << lg_N) - 1;
    size_t trials = 1 << (29 - lg_N);
    for (size_t j = 0; j < trials; j++) {
        for (size_t i = 0; i < (1 << lg_N); i++) {
            int addr = ((i + 523)*253573) & mask;
            a[addr] = a[addr] + 5;
        }
    }
}
```
The following programs are parallel versions of the programs on page 18. The only difference is that the inner loop in each program is converted into a *cilk_for* loop with no other modifications (e.g., coarsening).

```c
void frob1_parallel(int *a, size_t lg_N) {
    size_t trials = 1 << (31 - lg_N);
    for (size_t j = 0; j < trials; j++) {
        cilk_for (size_t i = 0; i < (1 << lg_N); i++)
            a[i] = a[i] + 5;
    }
}

void frob2_parallel(int *a, int *b, int *c, size_t lg_N) {
    size_t N = 1 << lg_N;
    size_t trials = 1 << (29 - lg_N);
    for (size_t j = 0; j < trials; j++) {
        cilk_for (size_t i = 0; i < N; i++)
            c[i] = a[i] + b[i] + a[N-i-1] + b[N-i-1] + c[N-i-1];
    }
}

void frob3_parallel(int *a, size_t lg_N) {
    int mask = (1 << lg_N) - 1;
    size_t trials = 1 << (29 - lg_N);
    for (size_t j = 0; j < trials; j++) {
        cilk_for (size_t i = 0; i < (1 << lg_N); i++) {
            int addr = ((i + 523)*253573) & mask;
            a[addr] = a[addr] + 5;
        }
    }
}
```
Match the letter of the performance curve (i.e., A, B, ..., F) with the program that generated it. (*Hint: All of the serial codes use dashed lines, and all of the parallel codes use solid lines.*) The performance graph is included below for your convenience.

<table>
<thead>
<tr>
<th>5.1</th>
<th>frob1_serial =</th>
<th>5.2</th>
<th>frob1_parallel =</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>frob2_serial =</td>
<td>5.4</td>
<td>frob2_parallel =</td>
</tr>
<tr>
<td>5.5</td>
<td>frob3_serial =</td>
<td>5.6</td>
<td>frob3_parallel =</td>
</tr>
</tbody>
</table>