Practice Quiz 3

Name: ____

Instructions

- DO NOT open this quiz booklet until you are instructed to do so.
- This quiz booklet contains 14 pages, including this one. You have 80 minutes to earn 80 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 on this coversheet, and 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 need not explain these answers unless you wish to receive partial credit if your answer is wrong. For these kinds of questions, **since incorrect answers will be penalized, do not guess unless you are reasonably sure**.
- Good luck!

Number	**Question**	**Parts**	**Points**	**Score**	**Grader**
0	Name on Every Page	14	1		
1	True or False	10	20		
2	Caching in Matrix Multiplication	3	12		
3	Dining Philosophers	4	16		
4	Caching for Karatsuba	4	17		
5	Lock-free FIFO Queue	1	14		
	Total		80		

1 True or False (10 parts, 20 points)

Incorrect answers will be penalized, so do not guess unless you are reasonably sure. You need not justify your answer, unless you want to leave open the possibility of receiving partial credit if your answer is wrong. Comments will have no impact on a correct answer.

1.1

Parallel pull algorithms are generally slower than parallel push algorithms because they require state updates to be atomic.

True **False**

1.2

Streaming writes avoid the latency of reading cache blocks into the cache.

True **False**

1.3

Suppose that a program runs on a machine with a fully associative cache of size $\mathcal{M}$ and incurs n conflict misses. Then we must have $n > \mathcal{M}$.

True **False**

1.4

In the ideal-cache model, an $n \times n$ submatrix of an underlying matrix stored in row-major order always fits in a cache of size $\Theta(n^2)$.

True **False**

1.5

Suppose that a parallel Cilk program exhibits poor speedup when run on a multicore machine with P processing cores. If the cause is insufficient memory bandwidth, it can often be detected by executing P copies of the serial elision of the program simultaneously and comparing the runtime with that of a single serial elision.

True **False**

1.6

Concurrent programs that are data-race free and obey atomicity constraints are deterministic.

True **False**

1.7

A yielding mutex lock should generally be preferred over a spinning mutex lock when a critical section is large.

True **False**

1.8

Hardware can reorder memory operations by allowing a store to bypass (move earlier in the execution order) a load to a different location in order to compensate for memory-system latency.

True **False**

1.9

Running on a machine with a relaxed-consistency memory model can cause Peterson's algorithm for mutual exclusion to operate incorrectly unless memory fences are used.

True **False**

1.10

The storage cost of a Compressed Sparse Rows (CSR) representation of a sparse graph with n vertices and m edges is $\Theta(m + n)$.

True **False**

2 Caching in Matrix Multiplication (3 parts, 12 points)

Ben Bitdiddle writes the following code to multiply two matrices. Ben's machine has a 32KB (32,768 byte) L1 data cache with 64-byte cache lines. Assume that Ben's machine has a fully associative cache with least-recently-used (LRU) replacement and code executes on a single processor.

```
void matrix_multiply(int32_t *a, int32_t *b, int32_t *c, size_t n) {
  for (size_t row = 0; row < n; row++)
    for (size_t col = 0; col < n; col++)
      for (size_t i = 0; i < n; i++)
        c[row*n + col] += a[row*n + i] * b[i*n + col];
}
```

Estimate the number of L1-cache misses each of the following three calls incurs.

2.1

```
matrix_multiply(a, b, c, 64);
```

A $(64 \cdot 3)/16 = 12$

B $64 \cdot 3 = 192$

C $(64^2 \cdot 3)/16 = 768$

D $64^2 + (64^2 \cdot 2)/16 = 4,608$

2.2

```
matrix_multiply(a, b, c, 256);
```

A $(256^2 \cdot 3)/16 = 12{,}288$

B $(256^3 + 256^2 \cdot 2)/16 = 1{,}056{,}768$

C $(256^3 \cdot 3)/16 = 3{,}145{,}728$

D $256^3 + 256^2 + 256^2/16 = 16{,}846{,}848$

2.3

```
matrix_multiply(a, b, c, 1024);
```

A $(1024^2 \cdot 3)/16 = 196{,}608$

B $(1024^3 + 1024^2 \cdot 2)/16 = 67{,}239{,}936$

C $(1024^3 \cdot 2)/16 + 1024^2 = 135{,}266{,}304$

D $1024^3 + (1024^3 + 1024^2)/16 = 1{,}140{,}916{,}224$

3 Dining Philosophers (4 parts, 16 points)

In the Dining Philosophers problem introduced in lecture, each of n philosophers needs the two chopsticks on either side of his/her plate to eat his/her noodles. Consider the pseudocode for philosopher i below, which uses fair mutexes:

```
while (1) {
    think();
    lock(&chopstick[i].L);
    lock(&chopstick[(i+1)%n].L);
    eat();
    unlock(&chopstick[i].L);
    unlock(&chopstick[(i+1)%n].L);
}
```

We saw in lecture that a deadlock may occur with this code.

3.1

In a first attempt to solve this problem, Ben Bitdiddle sets a time limit t. If a philosopher holds his or her first chopstick for t seconds without acquiring the second chopstick, the philosopher puts down the first chopstick, waits another t seconds, and then restarts the chopstick-acquisition protocol.

Which of the following may happen?

A	A critical-section violation may occur	**Yes**	**No**
B	Deadlock may occur	**Yes**	**No**
C	Starvation may occur	**Yes**	**No**
D	Livelock may occur	**Yes**	**No**

3.2

In a second attempt to solve the Dining Philosophers problem, Alyssa P. Hacker decides to keep track of how many philosophers are attempting to eat, and only allow at most $n - 1$ philosophers to attempt eating at the same time. Alyssa uses the atomic compare-and-swap operation `CAS(address, old, new)`.

```
volatile int count = 0;
while (1) {
  think();
  int c = count;
  if (c < n-1 && CAS(&count, c, c+1)) {
    lock(&chopstick[i].L);
    lock(&chopstick[(i+1)%n].L);
    eat();
    unlock(&chopstick[i].L);
    unlock(&chopstick[(i+1)%n].L);
    c = count;
    while (!CAS(&count, c, c-1)) {
      c = count;
    }
  }
}
```

Which of the following may happen?

A A critical-section violation may occur **Yes** **No**

B Deadlock may occur **Yes** **No**

C Starvation may occur **Yes** **No**

D Livelock may occur **Yes** **No**

In a third attempt to solve the Dining Philosophers problem, Professor Kung writes the following code:

```
while (1) {
    think();
    if (i & 1) {
        lock(&chopstick[i].L);
        lock(&chopstick[(i+1)%n].L);
    } else {
        lock(&chopstick[(i+1)%n].L);
        lock(&chopstick[i].L);
    }
    eat();
    unlock(&chopstick[i].L);
    unlock(&chopstick[(i+1)%n].L);
}
```

3.3

Which of the following may happen?

A A critical-section violation may occur **Yes** **No**

B Deadlock may occur **Yes** **No**

C Starvation may occur **Yes** **No**

D Livelock may occur **Yes** **No**

3.4

Assume that n is an even number. Which of the following may happen?

A A critical-section violation may occur **Yes** **No**

B Deadlock may occur **Yes** **No**

C Starvation may occur **Yes** **No**

D Livelock may occur **Yes** **No**

4 Caching for Karatsuba (4 parts, 17 points)

For this problem, assume an ideal cache model (fully associative with an optimal or LRU replacement policy, as appropriate) with cache size $\mathcal{M}$ and cache-line length $\mathcal{B}$.

Recall that a ***polynomial*** in the variable x is a formal sum

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ ,$$

where the values $a_0, a_1, \ldots, a_{n-1}$ are the ***coefficients*** of the polynomial and $n-1$ is its ***degree***. Any integer strictly greater than the degree of a polynomial is a ***degree-bound*** of that polynomial.

Computing the product of two polynomials with degree-bound n takes $\Theta(n^2)$ work if the normal grade-school multiplication algorithm is used. In 1962, however, Karatsuba invented a faster way based on a clever scheme for multiplying two linear polynomials $a_1x + a_2$ and $b_1x + b_2$. The product is $a_1b_1x^2 + (a_1b_2 + a_2b_1)x + a_2b_2$, which normally involves four multiplications of coefficients. Karatsuba's algorithm accomplishes it in three:

$$\begin{aligned} m_1 &= a_1 \cdot b_1 \ , \\ m_2 &= a_2 \cdot b_2 \ , \\ m_3 &= (a_1 + a_2) \cdot (b_1 + b_2) \ . \end{aligned}$$

The product of the two polynomials is then $m_1x^2 + (m_3 - m_1 - m_2)x + m_2$.

This algorithm can be performed recursively using divide-and-conquer, similar to Strassen's algorithm for multiplying matrices. The code for Karatsuba polynomial multiplication is given on the following page, where each polynomial is represented as an array of its coefficients. (Do not spend time trying to understand the code. You only need to understand its performance characteristics.)

```c
// Compute the polynomial product C(x) = A(x) * B(x).
// A(x) and B(x) each have degree-bound n (and n elements).
// C(x) has degree-bound 2n-1 (and 2n-1 elements).
// Each polynomial is represented as an array of coefficients.

void karatsuba(double* A, double* B, int n, double* C) {
  assert((n & (-n)) == n);  // n must be an exact power of 2.

  if (n == 1) //degree 0
    C[0] = A[0] * B[0];
  else {
    double M1[n-1], M2[n-1], M3[n-1], AA[n/2], BB[n/2];

    double* A1 = A;
    double* A2 = A + n/2;
    double* B1 = B;
    double* B2 = B + n/2;

    karatsuba(A1, B1, n/2, M1);      // M1(x) = A1(x) * B1(x)

    karatsuba(A2, B2, n/2, M2);      // M2(x) = A2(x) * B2(x)

    for (int i = 0; i < n/2; i++)   // AA(x) = A1(x) + A2(x)
      AA[i] = A1[i] + A2[i];

    for (int i = 0; i < n/2; i++)   // BB(x) = B1(x) + B2(x)
      BB[i] = B1[i] + B2[i];

    karatsuba(AA, BB, n/2, M3);      // M3(x) = AA(x) * BB(x)

    for (int i = 0; i < n-1; i++) { // C(x) = M3(x) - M2(x) - M1(x)
      C[i] = M1[i];
      C[i + n] = M2[i];
    }
    C[n-1] = 0;
    for (int i = 0; i < n-1; i++)
      C[i + n/2] += M3[i] - M1[i] - M2[i];
  }
}
```

4.1

The work $T(n)$ required by this program to multiply two polynomials of degree-bound n satisfies the recurrence

$$T(n) = 3T(n/2) + \Theta(n) \ .$$

What is the solution to this recurrence?

A $\Theta(n)$

B $\Theta(n \lg n)$

C $\Theta(n^{\log_3 2})$

D $\Theta(n^{\lg 3})$

E None of the above

The worst-case number $Q(n)$ of cache misses incurred by the Karatsuba algorithm is given by

$$Q(n) = \begin{cases} \Theta(n/\mathcal{B}) & \text{if } n < c\mathcal{M}\text{, where } c \leq 1 \text{ is a sufficiently small constant,} \\ 3Q(n/2) + \Theta(n/\mathcal{B}) & \text{otherwise.} \end{cases}$$

4.2

Explain why $Q(n) = 3Q(n/2) + \Theta(n/\mathcal{B})$ if n is sufficiently large.

4.3

Explain why $Q(n) = \Theta(n/\mathcal{B})$ if $n < c\mathcal{M}$.

4.4

Sketch a recursion tree for the recurrence for $Q(n)$. Compute the height and the number of leaves of the recursion tree, and label the internal nodes and leaves of the tree with the corresponding number of cache misses incurred at them. Solve the recurrence, providing a tight asymptotic bound in simple form.

Height =

Number of leaves =

$Q(n)$ =

5 Lock-free FIFO Queue (1 part, 14 points)

Consider the following lock-free FIFO queue with blocking semantics for a single enqueuer and a single dequeuer.

```
_____________ int head = 0;
_____________ int tail = 0;
_____________ double *items = malloc(CAPACITY*sizeof(double));

void enq(double x) {
    while (tail - head == CAPACITY) {}
    _____________
    items[tail % CAPACITY] = x;
    _____________
    tail++;
}

void deq(double x) {
    while (tail - head == 0) {}
    _____________
    double x = items[head % CAPACITY];
    _____________
    head++;
    return x;
}
```

The queue is implemented as an array. Initially, the `head` and `tail` fields are equal, and the queue is empty. If `head` and `tail` differ by exactly `CAPACITY`, then the queue is full. The `enq()` function reads the `head` field, and if the queue is full, it repeatedly checks `head` until the queue is no longer full. It then stores the double `x` in the `items` array and increments the `tail` field. The `dec()` function works in a symmetric way.

For this problem, assume the x86-64 total-store-order memory model discussed in lecture.

Since the queue is blocking, removing an item from an empty queue or inserting an item into a full one causes the thread to block (or wait). The surprising thing about this queue is that it requires only atomic loads and stores and not an atomic read-modify-write operation. It may, however, require the use of volatile variables, a compiler fence (`asm volatile("": : :"memory")`), or a hardware fence (`asm volatile("mfence": : :"memory")`).

For each of the seven missing code snippets, choose one of the following such that the code is correct and has the highest performance:

- `/* Nothing */`
- `volatile`
- `COMPILER_FENCE();`
- `HARDWARE_FENCE();`

Line 1: ______________________________

Line 2: ______________________________

Line 3: ______________________________

Line 7: ______________________________

Line 9: ______________________________

Line 15: ______________________________

Line 17: ______________________________

MIT OpenCourseWare
https://ocw.mit.edu

6.172 Performance Engineering of Software Systems
Fall 2018

For information about citing these materials or our Terms of Use, visit: https://ocw.mit.edu/terms