Name:_____

1. (Concurrency)

   (a) Suppose we have an algorithm for finding the second-smallest element in a binary search tree. The code is somewhat complicated to account for all possible tree shapes, but it always descends to the correct node without making any modifications to the tree. Suppose the code is in a method `secondSmallestMidterm`. Would it be correct for two threads to execute `secondSmallestMidterm` concurrently? Explain briefly.

   (b) Here is a simpler algorithm for finding the second smallest element in a binary search tree in terms of some other operations:

```
int deleteMin() { ... } // guarded by mutex
int findMin() { ... } // guarded by mutex
void insert(int x) { ... } // guarded by mutex
int secondSmallestFinal() {
  int min = this.deleteMin();
  int ans = this.findMin();
  this.insert(min);
  return ans;
}
```

   Notice `deleteMin`, `findMin`, and `insert` are synchronized methods, acquiring the lock when they are called and releasing it after they return.

      i. Suppose two threads call `secondSmallestFinal` concurrently. Demonstrate how one of them can get the wrong answer.

      ii. Does the code above have any *data races*? Explain briefly.

      iii. What is the easiest way to fix `secondSmallestFinal`?

Name:_____

Solution:

(a) Yes, `secondSmallestMidterm` does no writes to shared memory, so multiple concurrent executions would not interfere. (Interleaving with other tree operations like `insert` would be a problem, but that is not the question here.)

(b) i. If thread 1 deletes the minimum element and then thread 2 runs `secondSmallestFinal`, it will actually get the 3rd smallest element (or raise an exception if there were only 2 elements to begin with).

ii. No, there is never unsynchronized access to the same field by two threads because all field access is by synchronized helper methods.

iii. Make it guarded by the mutex as well.

2. (Concurrency) You are designing a new social-networking site to take over the world. To handle all the volume you expect, you want to support multiple threads with a fine-grained locking strategy in which each user's profile is protected with a different lock. At the core of your system is this simple class definition:

```
class UserProfile {
private:
   static int id_counter;
   int id; // unique for each account
   int *friends = new int[9999]; // horrible style
   int numFriends;
   Image *embarrassingPhotos = new Image[9999];
   mutex mtx;
public:
   UserProfile() { // constructor for new profiles
     id = id_counter++;
     numFriends = 0;
   }
   void makeFriends(UserProfile *newFriend) {
      mtx.lock();
      newFriend->mtx.lock();
      if(numFriends == friends.length
         || newFriend->numFriends == newFriend->friends.length)
          throw TooManyFriendsException();
        friends[numFriends++] = newFriend->id;
        newFriend->friends[newFriend->numFriends++] = id;
      }
   }
   void removeFriend(UserProfile frenemy) {
      ...
   }
}
```

(a) The constructor has a concurrency error. What is it and how would you fix it? A short English answer is enough – no code or details required.

Name:_____

(b) The `makeFriends` method has a concurrency error. What is it and how would you fix it? A short English answer is enough – no code or details required.

Solution:

(a) There is a data race on `id_counter`. Two accounts could get the same `id` if they are created simultaneously by different threads. Or even stranger things could happen. You could use a lock for `id_counter` to prevent this.

(b) There is a potential deadlock if there are two objects `obj1` and `obj2` and one thread calls `obj1.makeFriends(obj2)` when another thread calls `obj2.makeFriends(obj1)`. The fix is to acquire locks in a consistent order based on the `id` fields, which are unique.

3. (Integer representation) Please complete the following table of 4-bit integer values (use a two's complement representation for signed values). Tables of powers of two and multiple of 16 are provided in case you should fine them useful.

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
1	2	4	8	16	32	64	128

16×	2	3	4	5	6	7	8	9	10	11	12	13	14	15
=	32	48	64	80	96	112	128	144	160	176	192	208	224	240

binary	**unsigned decimal integer**	**unsigned hexadecimal integer**	**signed decimal integer**
			-4
		0xE	
0010			
1111			
			-8

Solution:

binary	**unsigned decimal integer**	**unsigned hexadecimal integer**	**signed decimal integer**
1100	12	0xC	-4
1110	14	0xE	-2
0010	2	0x2	2
1111	15	0xF	-1
1000	8	0x8	-8

4. (x86-64 assembly)

Name:_____

(a) For each of the following C snippets, give a series of assembly instructions equivalent to that snippet. Relevant register values are given in each case. You can omit width specifiers on instructions (e.g., `mov` instead of `movq`). Note that register `%rax` is used as the return value, so the assembly for any snippet with `return` must ensure the return value is stored in `%rax` when the `ret` instruction executes.

i. C code: `*p = *p + 1`
`p` has type `int*`

Register state:

register	value
`%rdi`	location of `p`

x86-64 assembly:

ii. C code: `return arr[i] + arr[i+1]`
`arr` has type `int*` `i` has type `int`

Register state:

register	value
`%rdi`	location of `arr`
`%rsi`	value of `i`

x86-64 assembly:

iii. C code: `node->next = node->next->next`
`node` has type `struct node*`
`struct node` is a structure representing a linked list node that contains a string:

```
struct node {
    char* word;
    struct node *next;
}
```

Register state:

register	value
`%rdi`	location of `node`

x86-64 assembly:

Name:_____

(b) For each of the following series of assembly instructions, give the equivalent snippet of C. Relevant register values and variable names are given in each case.

i. x86-64 assembly:

```
add  %rsi, %rdx
imul %rdi, %rdx
mov  %rdx, %rax
ret
```

Register state:

register	value
%rdi	value of variable a that has type int
%rsi	value of variable b that has type int
%rdx	value of variable c that has type int

C code:

ii. x86-64 assembly:

```
mov (%rsi,%rdx,8),%rbx
mov %rbx,(%rdi)
```

Register state:

register	value
%rdi	location of variable s that has type char*
%rsi	location of variable argv that has type char**
%rdx	value of variable i that has type int

C code:

Solution:

(a) i.
```
add $1,(%rdi)
```

ii.
```
mov (%rdi, %rsi, 4), %rax
add 4(%rdi, %rsi, 4), %rax
ret
```

iii.
```
mov 8(%rdi), %rbx
mov 8(%rbx), %rcx
mov %rcx, 8(%rdi)
```

Name:_____

(b) i. `return a * (b + c)`

ii. `s = argv[i]`

Name:_____

Reference

This is an incomplete list. **Just because a command or option is documented here doesn't mean there is a question that uses it.**

Bash

```
wc [OPTION]... [FILE]...
    Print  newline, word, and byte counts for each FILE, and a total line if
    more than one FILE is specified.
    -c, --bytes
          print the byte counts
    -l, --lines
          print the newline counts
    -w, --words
          print the words counts

shell scripting
$(cmd)      substitute with the stdout from running cmd
$n          nth argument ($0 is the command itself)
$#          number of arguments (does not include $0)
$@          a list of all the arguments (does not include $0)
$?          the exit status of the most recent command
shift       discard the first argument ($1) and move the remaining arguments
            down ($2 becomes $1 and so on, this affects $# and $@).

for item in list_of_things
do
    ...
done

if TEST
then
    ...
fi

tests:
    [ -d file ]      true if file exists and is a directory
    [ -e file ]      true if file exists, regardless of type
    [ -f file ]      true if file exists, and is a regular file
    [ -n string ]    true if length of string is nonzero
    [ -z string ]    true if length of string is zero
    [ s1 = s2 ]      true if the strings s1 and s2 are identical.
    [ s1 != s2 ]     true if the strings s1 and s2 are not identical.
    [ n1 -eq n2 ]    true if integer n1 is equal to integer n2.
                     similarly for not equal (-ne), greater than (-gt),
                     and less than (-lt)
```

Name:_____

x86-64 assembly

instructions:

instruction	**effect**	**description**
`mov` S, D	$D \leftarrow S$	move
`add` S, D	$D \leftarrow D + S$	add
`sub` S, D	$D \leftarrow D - S$	subtract
`imul` S, D	$D \leftarrow D * S$	multiply
`ret`	returns, uses value in `%rax` as return value	return

operand forms:

type	**form**	**operand value**
immediate	$\$D$	D
register	$\%R$	value stored in register R
memory	D	value at memory location D
memory	$(\%R)$	value at memory location given by value stored in register R
memory	$D(\%R_b)$	value at memory location $D + R_b$
memory	$(\%R_b, \%R_i)$	value at memory location $R_b + R_i$
memory	$D(\%R_b, \%R_i)$	value at memory location $D + R_b + R_i$
memory	$(, \%R_i, s)$	value at memory location $R_i \cdot s$
memory	$D(, \%R_i, s)$	value at memory location $D + R_i \cdot s$
memory	$(\%R_b, \%R_i, s)$	value at memory location $R_b + R_i \cdot s$
memory	$D(\%R_b, \%R_i, s)$	value at memory location $D + R_b + R_i \cdot s$

The scaling factor s must be either 1, 2, 4, or 8.