Semaphore writePending=1, readBlock=1, mutex1=1, mutex2=1;
Int readCount=0, writeCount=0;

Reader()
while (TRUE) {
    other stuff;
P(writePending);
P(readBlock);
P(mutex1);
    readCount = readCount +1;
    if(readCount == 1)
        P(writeBlock);
V(mutex1);
V(readBlock);
V(writePending);
    access resource;
P(mutex1);
    readCount = readCount -1;
    if(readCount == 0)
        V(writeBlock);
V(mutex1);
}

Writer()
while(TRUE){
    other stuff;
P(mutex2);
    writeCount = writeCount +1;
    if (writeCount == 1)
        P(readBlock);
V(mutex2);
P(writeBlock);
    access resource; V(writeBlock);
P(mutex2);
    writeCount - writeCount - 1;
    if (writeCount == 0)
        V(readBlock);
V(mutex2);
}

Question 1:
Assume that one Reader has reached its “access resource” line, and remains there.
Assume each of the following actions occur in the specified order, each in a different
thread. For each one, give the line number where, if at all, each thread blocks:

a. Writer1 arrives:____31______
b. Reader2 arrives:___8_______
When Reader1 exits its “access resource” line:
c. What line must Reader1 reach before another thread is allowed to
    proceed?:_____20_______
d. Which thread so proceeds?:__Writer1____
// ListElement::ListElement
// Initialize a list element, so it can be added somewhere on a list.
// "itemPtr" is the item to be put on the list. It can be a pointer
to anything.
// "sortKey" is the priority of the item, if any.
ListElement::ListElement(void *itemPtr, int sortKey)
{
    item = itemPtr;
    key = sortKey;
    next = NULL; // assume we'll put it at the end of the list
}

// List::SortedInsert
// Insert an "item" into a list, so that the list elements are
// sorted in increasing order by "sortKey".
// Allocate a ListElement to keep track of the item.
// If the list is empty, then this will be the only element.
// Otherwise, walk through the list, one element at a time,
to find where the new item should be placed.
// "item" is the thing to put on the list, it can be a pointer to
// anything.
// "sortKey" is the priority of the item.
void List::SortedInsert(void *item, int sortKey)
{
    ListElement *element = new ListElement(item, sortKey);
    ListElement *ptr; // keep track

    if (IsEmpty()) { // if list is empty, put
        first = element;
        last = element;
    } else if (sortKey < first->key) {
        element->next = first;
        first = element;
    } else { // look for first elt in list bigger than item
        for (ptr = first; ptr->next != NULL; ptr = ptr->next) {
            if (sortKey < ptr->next->key) {
                element->next = ptr->next;
                ptr->next = element;
                return;
            }
        }
        last->next = element; // item goes at end of list
        last = element;
    }
}
// List::SortedRemove
// Remove the first "item" from the front of a sorted list.

// Returns:
// Pointer to removed item, NULL if nothing on the list.
// Sets *keyPtr to the priority value of the removed item
// (this is needed by interrupt.cc, for instance).
// "keyPtr" is a pointer to the location in which to store the
// priority of the removed item.

void *
List::SortedRemove(int *keyPtr)
{
    ListElement *element = first;
    void *thing;

    if (IsEmpty())
        return NULL;

    thing = first->item;
    if (first == last) {
        // list had one item, now has none
        first = NULL;
        last = NULL;
    } else {
        first = element->next;
    }

    if (keyPtr != NULL)
        *keyPtr = element->key;

    delete element;

    return thing;
}

Question 2:

a. Present an interleaved sequence of statements/instructions that result in a list that contains a ListElement object that has been deleted. (Where a SortedInsert() or SortedRemove() operation runs to completion in your sequence, just indicate the call, along with the item inserted or removed; if one of these primitives yields to another within the primitive, indicate the “call” arguments, and where the switch takes place, using line numbers for precision. The threads that call these primitives each generate N integers, and insert each on the list. Then each thread removes N items from the list.)

ONE POSSIBLE ANSWER: Notation: “T1 I(3)81” means Thread 1 inserts 3; but stops after line 81, “f=1e->[3]->[4]” means “first equals element in thread 1’s Insert method, which points to the ListElement containing 3. That element points to another ListElement containing 4, which points to NULL”

Sequence: T1 I(10), T2 I(5)81. List is: f=2Ie->[10]. T1 R(10).
List is 2Ie->[5]->[10](deleted), f=NULL. T2 I(5) finishes.
List is f->[5]->[10](deleted). This answers the question.
If T2 R(5), the list is f->[10](deleted), so both threads can complete inserting and deleting one element, leaving the list containing a deleted element.
b. Assume that, once deleted, an object’s memory space is filled with random data. Outline a program that could fairly reliably detect a list containing a deleted ListElement. Feel free to change the definitions of the List and ListElement structures. The detection program should NOT Segment Fault:

**POSSIBLE ANSWERS:**

When an object’s memory space is filled with random data, that space can be EXAMINED by referencing one of its fields: Assume Obj points to a deleted ListElement. Then “x=Obj->next”, for example, should work. But USING the referenced data as a pointer, as in Obj->next->next will likely give a SegFault.

To detect a deleted object: Add an int field “sig” to ListElement, which is initialized to hold a “magic number”, say 12345678. Before using x->next, check to be sure that x->sig==12345678. If not, report that element x is in the list, but has been deleted. This works with high probability, because it is unlikely that the random data placed in x’s memory will have exactly the same pattern of bits in field sig as does 12345678. Note that a Bool field makes detection far less reliable, since it takes on only 2 distinguishable values, say 0 and 1, and the random data has 50% probability of setting such a flag to the “wrong” value.

**Better answer:** Maintain a system data structure indexed by ListElement addresses, say, which indicates if the ListElement at the specified address is “valid”. This could be a hash table. You’d have to add and use a “destructor” method for a ListElement, that would update this table just before the element is deleted. A faster scheme would put an int field “tag” in each ListElement. When element x is constructed, the system array T is searched for an entry i such that T[i]=NULL. x->tag is set to i, and T[i] is set to x (the address of the element). To check, look at y->tag. If y->tag>=0, and y->tag<TMAX, and T[y->tag]=y, element y is not deleted. Again, the destructor for ListElement y should set T[y->tag]=NULL before y is deleted. This is faster than using a hash table, and both are very reliable. The “tag” scheme might run out of slots in the fixed-size array T, however.