

```

1 Semaphore writePending=1, readBlock=1, mutex1=1, mutex2=1;
2 Int readCount=0, writeCount=0;
3
4 Reader(){
5   while (TRUE) {
6     other stuff;
7     P(writePending);
8     P(readBlock);
9     P(mutex1);
10    readCount = readCount +1;
11    if(readCount == 1)
12      P(writeBlock);
13    V(mutex1);
14    V(readBlock);
15    V(writePending);
16    access resource;
17    P(mutex1);
18    readCount = readCount -1;
19    if(readCount == 0)
20      V(writeBlock);
21    V(mutex1); }}
22
23 Writer(){
24   while(TRUE){
25     other stuff;
26     P(mutex2);
27     writeCount = writeCount +1;
28     if (writeCount == 1)
29       P(readBlock);
30     V(mutex2);
31     P(writeBlock);
32     access resource;      V(writeBlock);
33     P(mutex2);
34     writeCount - writeCount - 1;
35     if (writeCount == 0)
36       V(readBlock);
37     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:

- Writer1 arrives: 31
- Reader2 arrives: 8

When Reader1 exits its “access resource” line:

- What line must Reader1 reach before another thread is allowed to proceed?: 20
- Which thread so proceeds?: Writer1

```

39 //-----
40 // ListElement::ListElement
41 //     Initialize a list element, so it can be added somewhere on a list.
42 //
43 //     "itemPtr" is the item to be put on the list. It can be a pointer
44 //     to anything.
45 //     "sortKey" is the priority of the item, if any.
46 //-----
47
48 ListElement::ListElement(void *itemPtr, int sortKey)
49 {
50     item = itemPtr;
51     key = sortKey;
52     next = NULL; // assume we'll put it at the end of the list
53 }
54
55 //-----
56 // List::SortedInsert
57 //     Insert an "item" into a list, so that the list elements are
58 //     sorted in increasing order by "sortKey".
59 //
60 //     Allocate a ListElement to keep track of the item.
61 //     If the list is empty, then this will be the only element.
62 //     Otherwise, walk through the list, one element at a time,
63 //     to find where the new item should be placed.
64 //
65 //     "item" is the thing to put on the list, it can be a pointer to
66 //     anything.
67 //     "sortKey" is the priority of the item.
68 //-----
69
70 void
71 List::SortedInsert(void *item, int sortKey)
72 {
73     ListElement *element = new ListElement(item, sortKey);
74     ListElement *ptr;          // keep track
75
76     if (IsEmpty()) { // if list is empty, put
77         first = element;
78         last = element;
79     } else if (sortKey < first->key) {
80         // item goes on front of list
81         element->next = first;
82         first = element;
83     } else {
84         // look for first elt in list bigger than item
85         for (ptr = first; ptr->next != NULL; ptr = ptr->next) {
86             if (sortKey < ptr->next->key) {
87                 element->next = ptr->next;
88                 ptr->next = element;
89                 return;
90             }
91         }
92         last->next = element;          // item goes at end of list
93         last = element;
94     }
95 }
96

```

```

96
97 //-----
98 // List::SortedRemove
99 //   Remove the first "item" from the front of a sorted list.
100 //
101 // Returns:
102 //   Pointer to removed item, NULL if nothing on the list.
103 //   Sets *keyPtr to the priority value of the removed item
104 //   (this is needed by interrupt.cc, for instance).
105 //
106 //   "keyPtr" is a pointer to the location in which to store the
107 //   priority of the removed item.
108 //-----
109
110 void *
111 List::SortedRemove(int *keyPtr)
112 {
113     ListElement *element = first;
114     void *thing;
115
116     if (IsEmpty())
117         return NULL;
118
119     thing = first->item;
120     if (first == last) {         // list had one item, now has none
121         first = NULL;
122         last = NULL;
123     } else {
124         first = element->next;
125     }
126     if (keyPtr != NULL)
127         *keyPtr = element->key;
128     delete element;
129     return thing;
130 }

```

### 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=1Ie->[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.