Thread 1

```
int list_remove front(list *lp) {
	int num;
	list element *element;
	assert(!is_empty(lp));
	element = lp->first;
	num = lp->first->item;
	if (lp->first == lp->last) {
		lp->first = lp->last = NULL;
	} else {
		lp->first = element->next;
	}
	lp->num_in_list--;
	free(element);
	return num;
}
```

Thread 2

```
int list_remove front(list *lp) {
	int num;
	list element *element;
	assert(!is_empty(lp));
	element = lp->first;
	num = lp->first->item;
	if (lp->first == lp->last) {
		lp->first = lp->last = NULL;
	} else {
		lp->first = element->next;
	}
	lp->num_in_list--;
	free(element);
	return num;
}
```

Thread 1

1

```
int list_remove front(list *lp) {
        int num;
        list element *element;
        assert(!is_empty(lp));
        element = lp->first;
        num = lp->first->item;
        if (lp->first == lp->last) {
                lp->first = lp->last = NULL;
        } else {
                lp->first = element->next;
        }
        lp->num_in_list--;
        free(element);
        return num;
}
```

Thread 2

```
int list_remove front(list *lp) {
        int num;
        list element *element;
        assert(!is_empty(lp));
        element = lp->first;
        num = lp->first->item;
        if (lp->first == lp->last) {
                lp->first = lp->last = NULL;
        } else {
                lp->first = element->next;
        }
        lp->num_in_list--;
        free(element);
        return num;
}
```

Thread 1

1

```
int list_remove front(list *lp) {
        int num;
        list element *element;
        assert(!is_empty(lp));
        element = lp->first;
        num = lp->first->item;
        if (lp->first == lp->last) {
                lp->first = lp->last = NULL;
        } else {
                lp->first = element->next;
        }
        lp->num_in_list--;
        free(element);
        return num;
}
```

Thread 2

2

```
int list_remove front(list *lp) {
        int num;
        list element *element;
        assert(!is_empty(lp));
        element = lp->first;
        num = lp->first->item;
        if (lp->first == lp->last) {
                lp->first = lp->last = NULL;
        } else {
                lp->first = element->next;
        }
        lp->num_in_list--;
        free(element);
        return num;
}
```

Thread 1

```
1   int list_remove front(list *lp) {
        int num;
        list element *element;
        assert(!is_empty(lp));
        element = lp->first;
        num = lp->first->item;
        if (lp->first == lp->last) {
            lp->first = lp->last = NULL;
        } else {
3           lp->first = element->next;
        }
        lp->num_in_list--;
        free(element);
        return num;
    }
```

Thread 2

```
2   int list_remove front(list *lp) {
        int num;
        list element *element;
        assert(!is_empty(lp));
        element = lp->first;
        num = lp->first->item;
        if (lp->first == lp->last) {
            lp->first = lp->last = NULL;
        } else {
            lp->first = element->next;
        }
        lp->num_in_list--;
        free(element);
        return num;
    }
```

Thread 1

1

3

```
int list_remove front(list *lp) {
        int num;
        list element *element;
        assert(!is_empty(lp));
        element = lp->first;
        num = lp->first->item;
        if (lp->first == lp->last) {
                lp->first = lp->last = NULL;
        } else {
                lp->first = element->next;
        }
        lp->num_in_list--;
        free(element);
        return num;
}
```

Thread 2

2

```
int list_remove front(list *lp) {
        int num;
        list element *element;
        assert(!is_empty(lp));
        element = lp->first;
        num = lp->first->item;
        if (lp->first == lp->last) {
                lp->first = lp->last = NULL;
        } else {
                lp->first = element->next;
        }
        lp->num_in_list--;
        free(element);
        return num;
}
```

1. Both threads return the same value
2. Decrements num_in_list twice even though only one item has been removed
3. Dereferences an element that has been freed
4. Calls free() twice on the same element

Assume there is one item in the list.

Thread 1

```
int list_remove front(list *lp) {
    int num;
    list element *element;
    assert(!is_empty(lp));
    element = lp->first;
    num = lp->first->item;
    if (lp->first == lp->last) {
        lp->first = lp->last = NULL;
    } else {
        lp->first = element->next;
    }
    lp->num_in_list--;
    free(element);
    return num;
}
```

Thread 2

```
void list_append(list *lp, int new item) {
    list element *element =
        malloc(sizeof(list element));
    element->item = new item;
    assert(!is_in_list(lp, new item));
    if (is_empty(lp)) {
        lp->first = element;
        lp->last = element;
    } else {
        lp->last->next = element;
        lp->last = element;
    }
    lp->num in list++;
}
```

1

Thread 1

2

```
int list_remove front(list *lp) {
    int num;
    list element *element;
    assert(!is_empty(lp));
    element = lp->first;
    num = lp->first->item;
    if (lp->first == lp->last) {
        lp->first = lp->last = NULL;
    } else {
        lp->first = element->next;
    }
    lp->num_in_list--;
    free(element);
    return num;
}
```

Thread 2

1

```
void list_append(list *lp, int new item) {
    list element *element =
        malloc(sizeof(list element));
    element->item = new item;
    assert(!is_in_list(lp, new item));
    if (is_empty(lp)) {
        lp->first = element;
        lp->last = element;
    } else {
        lp->last->next = element;
        lp->last = element;
    }
    lp->num in list++;
}
```

Thread 1

2

```
int list_remove front(list *lp) {
    int num;
    list element *element;
    assert(!is_empty(lp));
    element = lp->first;
    num = lp->first->item;
    if (lp->first == lp->last) {
        lp->first = lp->last = NULL;
    } else {
        lp->first = element->next;
    }
    lp->num_in_list--;
    free(element);
    return num;
}
```

Thread 2

1

```
void list_append(list *lp, int new item) {
    list element *element =
        malloc(sizeof(list element));
    element->item = new item;
    assert(!is_in_list(lp, new item));
    if (is_empty(lp)) {
```

3

```
        lp->first = element;
        lp->last = element;
    } else {
        lp->last->next = element;
        lp->last = element;
    }
    lp->num in list++;
}
```

Thread 1

2

```
int list_remove front(list *lp) {
    int num;
    list element *element;
    assert(!is_empty(lp));
    element = lp->first;
    num = lp->first->item;
    if (lp->first == lp->last) {
        lp->first = lp->last = NULL;
    } else {
        lp->first = element->next;
    }
    lp->num_in_list--;
    free(element);
    return num;
}
```

Thread 2

1

```
void list_append(list *lp, int new item) {
    list element *element =
        malloc(sizeof(list element));
    element->item = new item;
    assert(!is_in_list(lp, new item));
    if (is_empty(lp)) {
```

3

```
        lp->first = element;
        lp->last = element;
    } else {
        lp->last->next = element;
        lp->last = element;
    }
    lp->num in list++;
}
```

Thread 1 removes the only item in the list after thread 2 checks if the list is empty.

As a result, thread 2 will try to dereference a NULL pointer (lp->last) when it resumes.