Concurrent Linked-List

Concurrent Programming

Introduction

- Coarse-Grained Linked List
- Fine-Grained Linked List
- Optimistic Linked List
- Lock-Free Linked List (do it yourself)

Properties of our Linked List

- Single Linked-list
- Sorted by node's key
- Do not allows to insert a duplicated key
- Should supports *add, remove, contains* functions

Prepare

- Download *list_template.cpp*, and *workload & result* file from the Piazza Resource page

Coarse-Grained Linked List

- Only one thread can access the list at a time
- Prepare a giant lock

[coarse_grained.cpp]

```
29 // Giant lock
30 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Coarse-Grained Linked List

- Hold the giant lock and add a new node into the list

```
43 // add key into the list
44 bool list_add(int key) {
45     Node* node = (Node*)malloc(sizeof(Node));
46     node->key = key;
47
48     pthread_mutex_lock(&mutex);
49
50     Node* prev = head;
51     Node* curr = head->next;
52
53     while (curr->key < key) {
54         prev = curr;
55         curr = curr->next;
56     }
```

Coarse-Grained Linked List

```
58      if (curr->key == key) {
59          // This key is already in the list
60          pthread_mutex_unlock(&mutex);
61          return false;
62      }
63      prev->next = node;
64      node->next = curr;
65
66      pthread_mutex_unlock(&mutex);
67
68      return true;
69  }
```

- **Although it is a coarse-grained implementation, please keep compact your critical section**

Coarse-Grained Linked List

- Hold the giant lock and remove a node from the list

```
71 // remove key from the list
72 bool list_remove(int key) {
73     pthread_mutex_lock(&mutex);
74
75     Node* prev = head;
76     Node* curr = head->next;
77
78     while (curr->key < key) {
79         prev = curr;
80         curr = curr->next;
81     }
```

Coarse-Grained Linked List

```
82      if (curr->key != key) {
83          // This key is not in the list
84          pthread_mutex_unlock(&mutex);
85          return false;
86      }
87      prev->next = curr->next;
88      pthread_mutex_unlock(&mutex);
89      free(curr);
90
91      return true;
92  }
```

Coarse-Grained Linked List

- Hold the giant lock and see whether the list contains the key

```
 94 // check whether a key is in the list
 95 bool list_contains(int key) {
 96     pthread_mutex_lock(&mutex);
 97
 98     Node* prev = head;
 99     Node* curr = head->next;
100
101     while (curr->key < key) {
102         prev = curr;
103         curr = curr->next;
104     }
105     pthread_mutex_unlock(&mutex);
106
107     return (curr->key == key);
108 }
```

Did you notice the problem in this code?

Coarse-Grained Linked List

- Test your program with *workload.txt* file

```
jongbin@multicore-96:~/TA/Multicore/lab11$ g++ -o coarse_grained coarse_grained.cpp -lpthread
jongbin@multicore-96:~/TA/Multicore/lab11$ time ./coarse_grained
correct!

real    0m16.093s
user    0m17.792s
sys     0m0.852s
```

[8 threads / 10,000 operations per thread]

Fine-Grained Linked List

- Prepare a small lock for each node
- Access the list with Latch-Coupling

```
10 // list node structure
11 struct Node {
12     int key;
13     Node* next;
14     pthread_mutex_t mutex;
15 };
```

Fine-Grained Linked List

```
41 // add key into the list
42 bool list_add(int key) {
43     Node* node = (Node*)malloc(sizeof(Node));
44     node->key = key;
45     pthread_mutex_init(&node->mutex, NULL);
46
47     Node* prev = head;
48     pthread_mutex_lock(&prev->mutex);
49     Node* curr = head->next;
50     pthread_mutex_lock(&curr->mutex);
51
52     while (curr->key < key) {
53         pthread_mutex_unlock(&prev->mutex);
54         prev = curr;
55         curr = curr->next;
56         pthread_mutex_lock(&curr->mutex);
57     }
```

Fine-Grained Linked List

```
59      if (curr->key == key) {
60          // This key is already in the list
61          pthread_mutex_unlock(&prev->mutex);
62          pthread_mutex_unlock(&curr->mutex);
63          return false;
64      }
65      prev->next = node;
66      node->next = curr;
67
68      pthread_mutex_unlock(&prev->mutex);
69      pthread_mutex_unlock(&curr->mutex);
70
71      return true;
72  }
```

Fine-Grained Linked List

```
74 // remove key from the list
75 bool list_remove(int key) {
76     Node* prev = head;
77     pthread_mutex_lock(&prev->mutex);
78     Node* curr = head->next;
79     pthread_mutex_lock(&curr->mutex);
80
81     while (curr->key < key) {
82         pthread_mutex_unlock(&prev->mutex);
83         prev = curr;
84         curr = curr->next;
85         pthread_mutex_lock(&curr->mutex);
86     }
```

Fine-Grained Linked List

```
 87      if (curr->key != key) {
 88          // This key is not in the list
 89          pthread_mutex_unlock(&prev->mutex);
 90          pthread_mutex_unlock(&curr->mutex);
 91          return false;
 92      }
 93      prev->next = curr->next;
 94
 95      pthread_mutex_unlock(&prev->mutex);
 96      pthread_mutex_unlock(&curr->mutex);
 97
 98      free(curr);
 99
100      return true;
101  }
```

Fine-Grained Linked List

```
103 // check whether a key is in the list
104 bool list_contains(int key) {
105     Node* prev = head;
106     pthread_mutex_lock(&prev->mutex);
107     Node* curr = head->next;
108     pthread_mutex_lock(&curr->mutex);
109
110     while (curr->key < key) {
111         pthread_mutex_unlock(&prev->mutex);
112         prev = curr;
113         curr = curr->next;
114         pthread_mutex_lock(&curr->mutex);
115     }
116
117     bool ret = (curr->key == key);
118     pthread_mutex_unlock(&prev->mutex);
119     pthread_mutex_unlock(&curr->mutex);
120
121     return ret;
122 }
```

Fine-Grained Linked List

- Test your program with *workload.txt* file

```
jongbin@multicore-96:~/TA/Multicore/lab11$ g++ -o fine_grained fine_grained.cpp -lpthread
jongbin@multicore-96:~/TA/Multicore/lab11$ time ./fine_grained
correct!

real    0m13.482s
user    1m27.804s
sys     0m14.312s
```

[8 threads / 10,000 operations per thread]

Optimistic Linked List

- Prepare a small lock for each node
- Try to find the target node without latch-coupling, and hold the node's lock later

```
10 // list node structure
11 struct Node {
12     int key;
13     Node* next;
14     pthread_mutex_t mutex;
15 };
```

Optimistic Linked List

- Validate whether the neighbors are still in the list and they are still connected directly

```
41 // validate nodes
42 bool list_validate(Node* prev, Node* curr) {
43     Node* it = head;
44     while (it->key <= prev->key) {
45         if (it == prev) {
46             return (it->next == curr);
47         }
48         it = it->next;
49     }
50     return false;
51 }
```

Optimistic Linked List

- Find the target position without acquiring any lock

```
53 // add key into the list
54 bool list_add(int key) {
55     Node* node = (Node*)malloc(sizeof(Node));
56     node->key = key;
57     pthread_mutex_init(&node->mutex, NULL);
58
59     Node* prev;
60     Node* curr;
61
62     while (1) {
63         prev = head;
64         curr = head->next;
65         while (curr->key < key) {
66             prev = curr;
67             curr = curr->next;
68         }
```

Optimistic Linked List

- Hold locks later, and validate
- If fails, retry

```
70          pthread_mutex_lock(&prev->mutex);
71          pthread_mutex_lock(&curr->mutex);
72
73          if (!list_validate(prev, curr)) {
74              pthread_mutex_unlock(&prev->mutex);
75              pthread_mutex_unlock(&curr->mutex);
76          } else {
77              if (curr->key == key) {
78                  // This key is already in the list
79                  pthread_mutex_unlock(&prev->mutex);
80                  pthread_mutex_unlock(&curr->mutex);
81                  return false;
82              }
```

Optimistic Linked List

- Add a new node with the locks safely

```
84                  node->next = curr;
85                  prev->next = node;
86
87                  pthread_mutex_unlock(&prev->mutex);
88                  pthread_mutex_unlock(&curr->mutex);
89                  break;
90              }
91          }
92
93          return true;
94      }
```

Optimistic Linked List

```
 96 // remove key from the list
 97 bool list_remove(int key) {
 98     Node* prev;
 99     Node* curr;
100
101     while (1) {
102         prev = head;
103         curr = head->next;
104
105         while (curr->key < key) {
106             prev = curr;
107             curr = curr->next;
108         }
```

Optimistic Linked List

```
110         pthread_mutex_lock(&prev->mutex);
111         pthread_mutex_lock(&curr->mutex);
112
113         if (!list_validate(prev, curr)) {
114             pthread_mutex_unlock(&prev->mutex);
115             pthread_mutex_unlock(&curr->mutex);
116         } else {
117             if (curr->key != key) {
118                 // This key is not in the list
119                 pthread_mutex_unlock(&prev->mutex);
120                 pthread_mutex_unlock(&curr->mutex);
121                 return false;
122             }
```

Optimistic Linked List

```
123                 prev->next = curr->next;
124                 pthread_mutex_unlock(&prev->mutex);
125                 pthread_mutex_unlock(&curr->mutex);
126
127                 // free(curr); // this line could make a problem. WHY?
128                 break;
129             }
130         }
131
132         return true;
133 }
```

- We do not care about freeing memory at this time

Optimistic Linked List

```
135 // check whether a key is in the list
136 bool list_contains(int key) {
137     bool is_contain = false;
138 
139     while (1) {
140         Node* prev = head;
141         Node* curr = head->next;
142 
143         while (curr->key < key) {
144             prev = curr;
145             curr = curr->next;
146         }
```

Optimistic Linked List

```
148         pthread_mutex_lock(&prev->mutex);
149         pthread_mutex_lock(&curr->mutex);
150
151         if (!list_validate(prev, curr)) {
152             pthread_mutex_unlock(&prev->mutex);
153             pthread_mutex_unlock(&curr->mutex);
154         } else {
155             if (curr->key == key) {
156                 is_contain = true;
157             }
158             pthread_mutex_unlock(&prev->mutex);
159             pthread_mutex_unlock(&curr->mutex);
160             return is_contain;
161         }
162     }
163 }
```

Optimistic Linked List

```
jongbin@multicore-96:~/TA/Multicore/lab11$ g++ -o optimistic optimistic.cpp -lpthread
jongbin@multicore-96:~/TA/Multicore/lab11$ time ./optimistic
correct!

real    0m4.716s
user    0m33.484s
sys     0m0.004s
```

[8 threads / 10,000 query per thread]

- It will show a better performance with a read-most workload

Lock-Free Linked List

- Implement the Lock-Free linked-list by yourself
- Reference your text book or lecture slides

```
jongbin@multicore-96:~/TA/Multicore/lab11$ g++ -o lock_free lock_free.cpp -lpthread
jongbin@multicore-96:~/TA/Multicore/lab11$ time ./lock_free
correct!

real    0m2.038s
user    0m13.416s
sys     0m0.000s
```

[8 threads / 10,000 query per thread]

Lock-Free Linked List

- Bit stealing

```
#define ISMARKED(p)     ((unsigned long long)p & 0x8000000000000000)
#define MARKED(p)       ((unsigned long long)p | 0x8000000000000000)
#define UNMARKED(p)     ((unsigned long long)p & 0x7FFFFFFFFFFFFFFF)
#define REFERENCE(p)    UNMARKED(p)
```

- Compare and Swap

```
bool __sync_bool_compare_and_swap(type *ptr, type oldval, type newval);
```

- If *value pointed by ptr* is *oldval*, than change it to *newval*
- Return *true* if success, *false* otherwise

Thank You