Lightweight Deferred Updates for Linux Kenrel Scalability

Abstract

1 Introduction

2 Background

3 LDU Design

3.1 Log-based Concurrent updates

3.2 Approach

3.3 The LDU Algorithm

3.4 logical update

3.5 Physical update

4 PLDU design

4.1 Approach

4.2 The PLDU Algorithm

4.3 logical update

4.4 Physical update

5 Concurrent updates for Linux kernel

5.1 Case study:reverse mapping

5.2 anon vma

LDU.

PLDU.

5.3 file mapping

LDU.

PLDU.

6 Implementation

7 Evaluation

This section answers the following questions experimentally:

- Does LDU's design matter for applications?
- Why does LDU's scheme scale well?

7.1 Experimental setup

7.2 AIM7

8 Discussion

9 related work

10 Conclusion

11 Acknowledgments

```
function logical_insert(obj, root):
  If CAS(obj.del_node.mark, 1, 0) ≠ 1:
    obj.add_node.mark ← 1
    If test_and_set_bit(OP_INSERT, obj.exist) ≠ true:
      set_bit(OP_INSERT, obj.used):
      obj.add_node.op ← OP_INSERT
      obj.add_node.key ← obj
      obj.add_node.root ← root
      add_lock_less_list(obj.add_node)


function logical_remove(obj, root):
  If CAS(obj.add_node.mark, 1, 0) ≠ 1:
    obj.del_node.mark ← 1
    If test_and_set_bit(OP_REMOVE, obj.exist) ≠ true:
      set_bit(OP_REMOVE, obj.used):
      obj.del_node.op ← OP_REMOVE
      obj.del_node.key ← obj
      obj.del_node.root ← root
      add_lock_less_list(obj.del_node)
```

Figure 1: LDU logical update algorithm. `logical_insert` represents non-blocking insert function. It may be called by original insert position without locks. The fastpath is that when their object was removed by `logical_remove`, `logical_insert` just changes node's marking field.

```
function synchronize_ldu(obj, head):
  If (head.first = NULL):
    return;
  entry ← xchg(head.first, NULL);
  for each list node:
    obj ← node.key
    clear_bit(node.op, obj.exist)
    If !xchg(node.mark, 0):
      physical_update(node.op, obj, node.root)
    clear_bit(node.op, obj.used)

function physical_update(op, obj, root):
  If op = OP_INSERT :
    call real insert function(obj, root)
  Else If op = OP_REMOVE :
    call real remove function(obj, root)
```

Figure 2: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

```
function logical_insert(obj, root):
  If CAS(obj.del_node.mark, 1, 0) ≠ 1:
    obj.add_node.mark ← 1
    If test_and_set_bit(OP_INSERT, obj.exist) ≠ true:
      set_bit(OP_INSERT, obj.used):
      obj.add_node.op ← OP_INSERT
      obj.add_node.key ← obj
      obj.add_node.root ← root
      add_lock_less_list(obj.add_node)


function logical_remove(obj, root):
  If CAS(obj.add_node.mark, 1, 0) ≠ 1:
    obj.del_node.mark ← 1
    If test_and_set_bit(OP_REMOVE, obj.exist) ≠ true:
      set_bit(OP_REMOVE, obj.used):
      obj.del_node.op ← OP_REMOVE
      obj.del_node.key ← obj
      obj.del_node.root ← root
      add_lock_less_list(obj.del_node)
```

Figure 3: LDU logical update algorithm. `logical_insert` represents non-blocking insert function. It may be called by original insert position without locks. The fastpath is that when their object was removed by `logical_remove`, `logical_insert` just changes node's marking field.

```
function synchronize_ldu(obj, head):
  If (head.first = NULL):
    return;
  entry ← xchg(head.first, NULL);
  for each list node:
    obj ← node.key
    clear_bit(node.op, obj.exist)
    If !xchg(node.mark, 0):
      physical_update(node.op, obj, node.root)
    clear_bit(node.op, obj.used)

function physical_update(op, obj, root):
  If op = OP_INSERT :
    call real insert function(obj, root)
  Else If op = OP_REMOVE :
    call real remove function(obj, root)
```

Figure 4: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.