Lightweight Deferred Updates for Linux Kenrel Scalability

Abstract

Introduction

Background and Problem

LDU Design

Log-based Concurrent updates

Approach

The LDU Algorithm

LDU logical update

LDU Physical update

LDU Correctness

PLDU design

Approach

The PLDU Algorithm

PLDU logical update

PLDU Physical update

PLDU Correctness

Concurrent updates for Linux kernel

Case study:reverse mapping

anon vma

* LDU.
  * PLDU.

file mapping

* LDU.
  * PLDU.

Implementation

Evaluation

This section answers the following questions experimentally:

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

Experimental setup

To evaluate the performance of LDU, we use well-known three benchmarks:AIM7 Linux scalability benchmark, Exim email server in MOSBENCH and lmbench. We selected these three benchmarks because they are fork-intensive workloads and exhibit high reverse mapping lock contentions. Moreover, AIM7 benchmark has widely been used in practical area not only for testing the Linux but also for improving the scalability. To evaluate LDU for real world applications, we use Exim which is the most popular email server. A micro benchmark, Lmbench, has been selected to focus on Linux fork operation-intensive fine grained evaluations. Finally, we wanted to focus on Linux fork performance and scalability;therefore, we selected lmbench, a micro benchmark.

In order to evaluate Linux scalability, we used four different experiment settings. First, we used the stock Linux as the baseline reference. Second, we used ordered Harris lock-free list while we apply unordered Harris lock-free list for the third setting (see section 6). Finally, we used combination of unordered Harris lock-free list for anonymous mapping and our LDU for file mapping. Since we cannot obtain detailed implementation of

```
function logical_insert(obj, root):
  If xchg(obj.del_node.mark, 0) ≠ 1:
    BUG(obj.add_node.mark)
    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 xchg(obj.add_node.mark, 0) ≠ 1:
    BUG(obj.del_node.mark)
    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
    If !xchg(node.mark, 0):
      physical_update(node.op, obj, node.root)
    clear_bit(node.op, obj.used)
    If !xchg(node.mark, 0):
      physical_update(node.op, obj, node.root)

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.

Oplog, we could not include comparison between LDU and Oplog in this paper.

We compare our LDU implementation to a concurrent non-blocking Harris linked list [**?**];therefore, we implement the Harris The code refers from sysnchrobench [**?**] and ASCYLIB [**?**], and we convert their linked list to Linux kernel style. Because both synchrobench and ASCYLIB leak memory, we implement additional garbage collector for the Linux kernel using Linux's work queues and lock-less list.

In order to further improve performance, we move their ordered list to unordered list. A feature of the Harris linked list is all the nodes are ordered by their key. Zhang [**?**] implements a lock-free unordered list algorithm, whose list is each insert and remove operation appends an intermediate node at the head of the list;these approach is practically hard to implement. Indeed, Linux does not require contains operation because the Linux data structures such as list, tree and hash table not depended on search key;they depend on their unique object. This feature can eliminates the ordered list in Harris linked list. Therefore, we perform each insert operation appends an intermediate node at the first node of the list;on the other hand, each remove operation searches from head to their node.

We ran the three benchmarks on Linux 3.19.rc4 with stock Linux with the automatic NUMA balancing feature disabled because the Harris linked list has the iteration issue [**?**]. All experiments were performed on a 120 core machine with 8-socket, 15-core Intel E7-8870 chips equipped with 792 GB DDR3 DRAM.

AIM7

AIM7 forks many processes, each of which concurrently runs. We used AIM7-multiuser, which is one of workload in AIM7. The multiuser workload is composed of various workloads such as disk-file operations, process creation, virtual memory operations, pipe I/O, and arithmetic operation. To minimize IO bottlenecks, the workload was executed with tmpfs filesystems, each of which is 10 GB. To increase the number of users during our experiment and show the results at the peak user numbers, we used the crossover.

The results for AIM7-multiuser are shown in Figure 5, and the results show the throughput of AIM7-multiuser with four different settings. Up to 60 core, the stock Linux scales linearly while serialized updates in Linux kernel become bottlenecks. However, up to 120core, unordered harris list and our LDU scale well because these workloads can run concurrently updates and can reduce the locking overheads due to reader-writer semaphores(`anon_vma`, `file`). The combination of LDU

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


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

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
    If !xchg(node.mark, 0):
      physical_update(node.op, obj, node.root)
    clear_bit(node.op, obj.used)
    If !xchg(node.mark, 0):
      physical_update(node.op, obj, node.root)

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.

with unordered harris list has best performance and scalability outperforming stock Linux by 1.7x and unordered harris list by 1.1x. While the unordered harris list has 19% idle time(see Table 1), stock Linux has 51% idle time waiting to acquire both `anon_vma's rwsem` and `file's i_mmap_rwsem`. We can notice that although LDU has 23% idle time, the throughput is higher than unordered harris list. In this benchmark, the ordered harris list has the lowest performance and scalability because their `CAS` fails frequently.

Exim

To measure the performance of Exim, shown in Figure 6, we used default value of MOSBENCH to use tmpfs for spool files, log files, and user mail files. Clients run on the same machine and each client sends to a different user to prevent contention on user mail file. The Exim was bottlenecked by per-directory locks protecting file creation in the spool directories and by forks performed on different cores [**?**]. Therefore, although we eliminate the fork problem, the Exim may suffer from contention on spool directories.

Results shown in Figure 6 show that Exim scales well for all methods up to 60 core but not for higher core counts. The stock Linux shows performance degradation for more than 60 core. Both unordered harris list and our LDU do not suffer from performance loss because they do not acquire the `anon_vma` semaphore and `i_mmap` semaphore in fork. LDU performs better due to the fact that it uses both update-side absorbing and lock-less list, outperforming stock Linux by 1.6x and unordered harris list by 1.1x. Even though we applied scalable solution, Exim shows limitation on scalability improvement since the main bottleneck is per-directory lock contention on spool directories. The unordered harris list has 31% idle time, whereas LDU has 37% idle time due to the their efficient concurrent updates.

lmbench

lmbench has various workloads including process creation workload(fork, exec, sh -c, exit). This workload is used to measure the basic process primitives such as creating a new process, running a different program, and context switching. We configured process create workload to enable the parallelism option which specifies the number of benchmark processes to run in parallel [**?**]; we used 100 processes.

The results for lmbench are shown in Figure 7, and the results show the execution times of the fork microbenchmark in lmbench with four different methods. Three methods outperform stock Linux by 2.2x at 120 cores;however, before 30 core, two harris list have lower performance due to their execution overheads. While stock Linux has 90% idle time, other methods have approximately 50% idle time since stock Linux waits to acquire reverse mapping locks such as `anon_vma's rwsem` and `mapping's i_mmap_rwsem`.

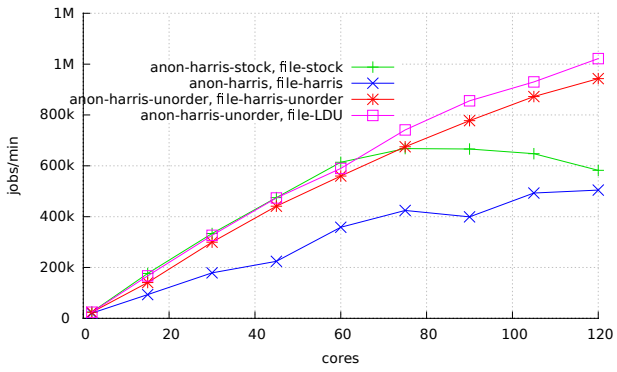

Figure 5: Scalability of AIM7 multiuser for different method. The combination LDU with unordered harris list scale well;in contrast, up to 60 core, the stock Linux scale linearly, then it flattens out.

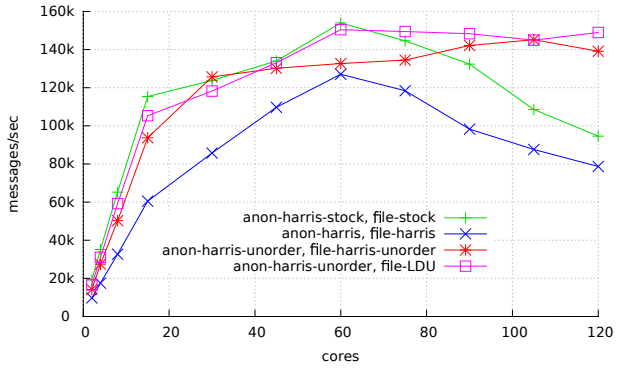

Figure 6: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both unordered harris list and our LDU flatten out.

AIM7	user	sys	idle
Stock(anon, file)	2487 s	1993 s	4647 s(51%)
H(anon, file)	1123 s	3631 s	2186 s(31%)
H-unorder(anon, flie)	3630 s	2511 s	1466 s(19%)
H-unorder(anon), L(file)	3630 s	1903 s	1662 s(23%)

EXIM	user	sys	idle
Stock(anon, file)	41 s	499 s	1260 s(70%)
H(anon, file)	47 s	628 s	1124 s(62%)
H-unorder(anon, file)	112 s	1128 s	559 s(31%)
H-unorder(anon), L(file)	87 s	1055 s	657 s(37%)

lmbench	user	sys	idle
Stock(anon, file)	11 s	208 s	2158 s(91%)
H(anon, file)	11 s	312 s	367 s(53%)
H-unorder(anon, file)	11 s	292 s	315 s(51%)
H-unorder(anon), L(file)	12 s	347 s	349 s(49%)

Table 1: Comparison of user, system and idle time at 120 cores.

Discussion

related work

In order to improve Linux scalability, researchers have been optimized memory management in Linux by finding and fixing scalability bottlenecks.

Shared address spaces in multithreaded applications easily become scalability bottlenecks since kernel operations including `mmap` and `munmap` system calls and `page faults` handling require per-process locks for synchronization. Multithreaded application, for example, can become bottleneck by kernel operations on their shared address space, whose operations are the `mmap` and `munmap` system calls and `page faults`. These operations are synchronized by a single per-process lock. BonsaiVM [**?**] solved this address space problem by using the RCU; RadixVM [**?**] created a new VM using refcache and radix tree, which enable `munmap`, `mmap`, and `page fault` on non-overlapping memory regions to scale perfectly. Alternatively, to avoid contention caused by shared address space locking, system programmers change their multithreaded applications to use processes [**?**].

Though sufficient level of performance scalability has been achieved for reader intensive operations through RCU and Hazard pointer, solutions to scalability for update-heavy operations has not been satisfiable. A recent paper by Arbel and Attiya [**?**] shows a new design of concurrent search tree called the Citrus tree. The Citrus tree combines RCU and fine-grained locks, and it supports concurrent write operations that traverse the search tree by using RCU concurrently. When increasing the update rate, Citrus tree still suffers from bottlenecks. RLU [**?**] presents a new synchronization mechanism that allows unsynchronized sequences of reads to execute concurrently with updates. In high update rate, Oplog can achieve substantially multi-core scaling for update-heavy data structures. Our work focus on update-heavy data structures and uses non-blocking method to store the operation log instead of per-core processing.

One method for the concurrent update is using the non-blocking algorithms [**?**] [**?**] [**?**], which are based on CAS. In non-blocking algorithms, each core tries to read the values of shared data structures from its local location, but has possibility of reading obsolete values. CAS is performed at the time of reading values that are not the current values and CAS fails and requires retrials sometimes when the values have been overwritten. These algorithms execute optimistically as though

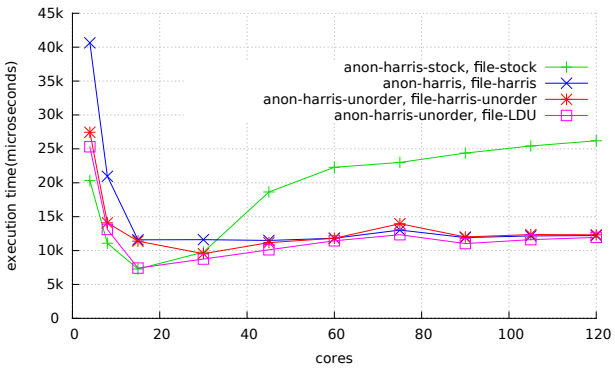

Figure 7: Execution time of lmbench's fork micro benchmark. The fork micro benchmark drops down for all methods up to 15 core but either flattens out or goes up slightly after that. At 15 core, the stock Linux goes up;the others flattens out

they read the value at location in their data structure;they may obtain stale data at the time. When they observed against the current value, they execute a CAS to compare the against value. The CAS fails when the value has been overridden, and they must be retried later on. Consequently, both repeated CAS operation and their iteration loop caused by CAS fails cause bottlenecks due to inter-core communication overheads [**?**]. Moreover, none of the non-blocking algorithms implements an iterator, whose data structure just consists of the insert, delete and contains operations [**?**]. The Linux, however, commonly uses the iteration to read, so when applying non-blocking algorithms to the Linux, they may meet this iteration problem. Petrank [**?**] solved this problem by using a consistent snapshot of the data structure; this method, however, may require a lot of effort to apply its sophisticated algorithms to Linux. For evaluation purposes, we implemented Harris linked list [**?**] to Linux, and we sometimes have failure where reading the pointer that had been deleted by updater concurrently result of the problem of the iteration.

MCS [**?**], a scalable exclusive lock, is used in the Linux kernel [**?**]. to avoid unfairness at high contention levels, so this scalable exclusive lock can be used for fine grained locking in Linux. However, in MCS, since only one thread may hold the lock at a time, it can cause low scalability in case of long critical regions. Reader-writer lock [**?**] allows either number of readers to execute concurrently or single writer to execute. Thus, readers-writer locks allow better scalability in case of read-mostly objects.

In read-mostly data structures, RCU [**?**] can be quite useful since it allows read operations to proceed without read locks, and delays freeing of data structures to avoid races. One drawback is that as the update rate increases, their performance and scalability decrease due to a single writer and their synchronization function. Consequently, scalable exclusive lock, reader-writer lock and RCU require serialization for updates and thus show significant limitation on scalability.

Conclusion

We propose a concurrent update algorithm, LDU, for update-heavy data structures scalable for many-core systems. To achieve the scalability during process spawning, we applied deferred log processing with global log queue and update-side absorbing to Linux reverse mapping. Evaluation results using the AIM7, Exim and lmbench reveal that LDU shows better performance up to 2.2 times compared to existing solutions. LDU is implemented on to Linux kernel 3.19 and available as open-source from `https://github.com/KMU-embedded/scalablelinux`.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (14-824-09-011, "Research Project on High Performance and Scalable Manycore Operating System")