# Linux kernel page management and lru lock optimization

Alibaba 时奎亮

# Speaker Bis

## 时奎亮

Alex Shi

Alibaba 基础软件部高级专家

LPC, kernel 峰会邀请专家

前Linaro Stable Kernel 维护者

前Intel LKP测试系统 维护者

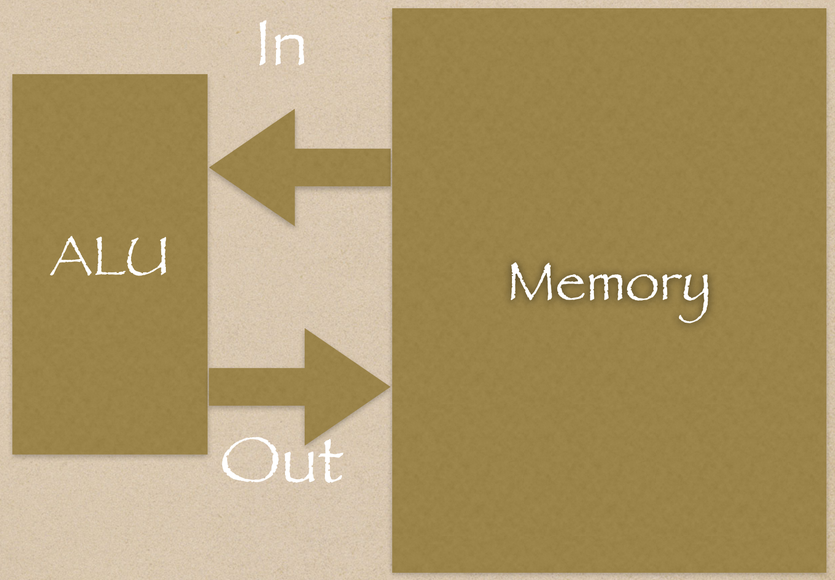
# Agenda

- Linux memory management
- Page mangement in Linux kernel
- Memcg and lru lock optimization
- Further chances
- Upstreaming status
- Questions?

# Linux memory management

## Von Neumann Arch

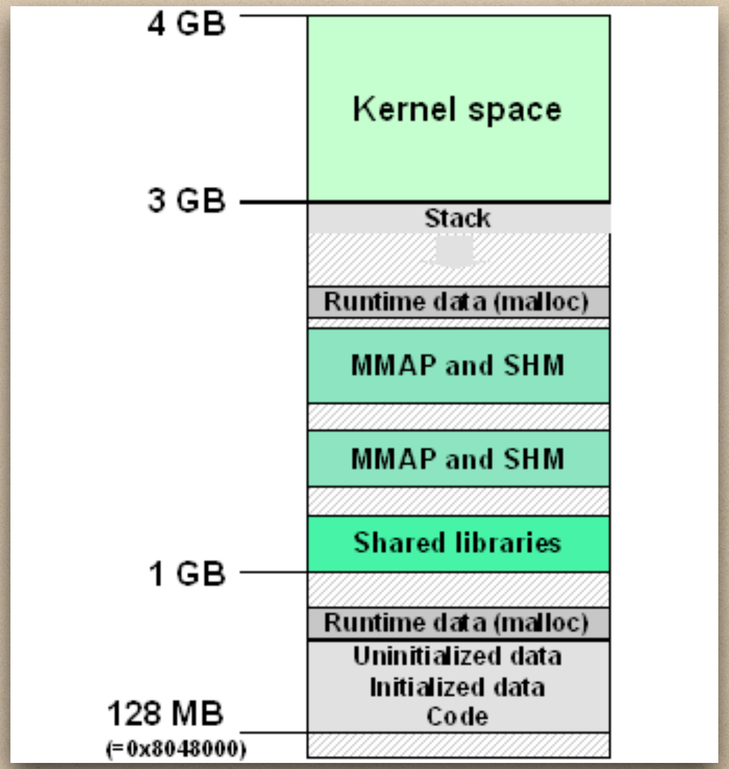
### Basic concept



# Linux memory management

Single address

- CPU: Arm, x86, PPC etc
- Portable C program expect flat memory
- Hardware mapped in address space



# Linux memory management

## Single address

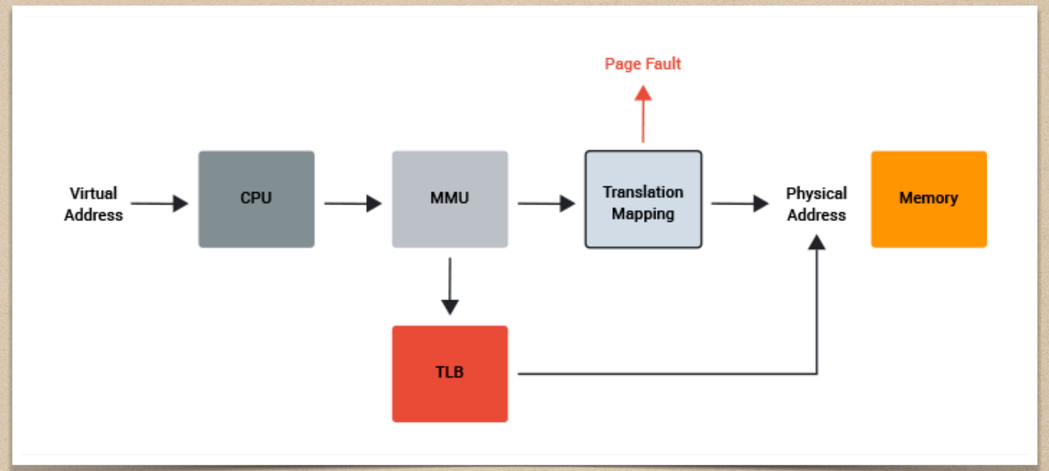
- CPU: Arm, x86, PPC
- Portable C program expect flat memory
- Hardware mapped in address space

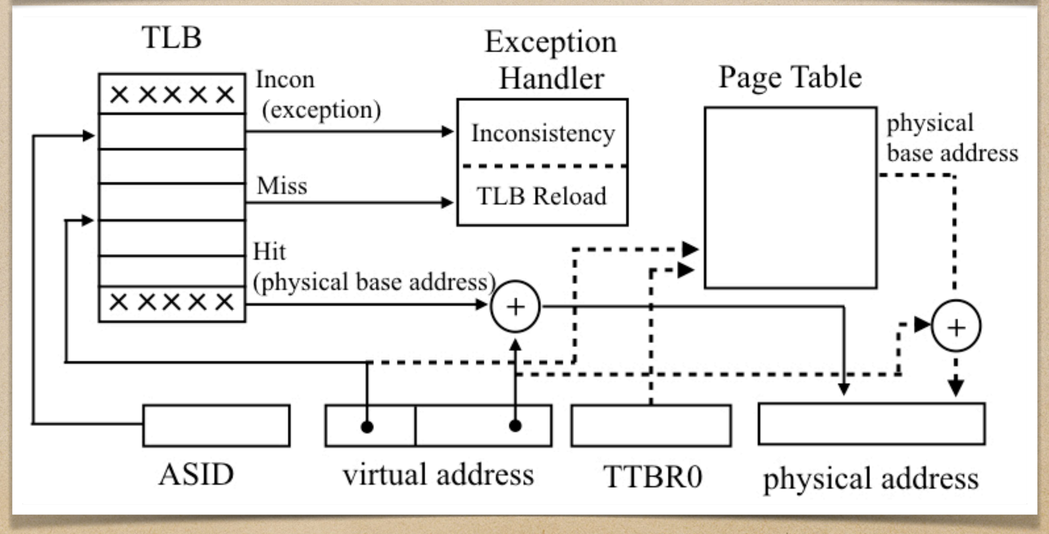
```
root@aliy80 linux-next]# cat /proc/1/maps
560b56a54000-560b56bb7000 r-xp 00000000 fc:01 660866                     /usr/lib/systemd/systemd
560b56db7000-560b56dda000 r--p 00163000 fc:01 660866                     /usr/lib/systemd/systemd
560b56dda000-560b56ddb000 rw-p 00186000 fc:01 660866                     /usr/lib/systemd/systemd
560b57027000-560b571c5000 rw-p 00000000 00:00 0                          [heap]
7f62b0000000-7f62b0029000 rw-p 00000000 00:00 0
7f62b0029000-7f62b4000000 ---p 00000000 00:00 0
7f62b8000000-7f62b8029000 rw-p 00000000 00:00 0
7f62b8029000-7f62bc000000 ---p 00000000 00:00 0
7f62be5a1000-7f62be5a2000 ---p 00000000 00:00 0
7f62be5a2000-7f62beda2000 rw-p 00000000 00:00 0
7f62beda2000-7f62beda3000 ---p 00000000 00:00 0
7f62beda3000-7f62bf5a3000 rw-p 00000000 00:00 0
7f62bf5a3000-7f62bf5a7000 r-xp 00000000 fc:01 673708                     /usr/lib64/libuuid.so.1.3.0
7f62bf5a7000-7f62bf7a6000 ---p 00004000 fc:01 673708                     /usr/lib64/libuuid.so.1.3.0
....
7f62c1b57000-7f62c1d56000 ---p 00024000 fc:01 657609                     /usr/lib64/libselinux.so.1
7f62c1d56000-7f62c1d57000 r--p 00023000 fc:01 657609                     /usr/lib64/libselinux.so.1
7f62c1d57000-7f62c1d58000 rw-p 00024000 fc:01 657609                     /usr/lib64/libselinux.so.1
7f62c1d58000-7f62c1d5a000 rw-p 00000000 00:00 0
7f62c1d5a000-7f62c1d7c000 r-xp 00000000 fc:01 657405                     /usr/lib64/ld-2.17.so
7f62c1f56000-7f62c1f60000 rw-p 00000000 00:00 0
7f62c1f79000-7f62c1f7b000 rw-p 00000000 00:00 0
7f62c1f7b000-7f62c1f7c000 r--p 00021000 fc:01 657405                     /usr/lib64/ld-2.17.so
7f62c1f7c000-7f62c1f7d000 rw-p 00022000 fc:01 657405                     /usr/lib64/ld-2.17.so
7f62c1f7d000-7f62c1f7e000 rw-p 00000000 00:00 0
7ffc2c833000-7ffc2c854000 rw-p 00000000 00:00 0                          [stack]
7ffc2c873000-7ffc2c877000 r--p 00000000 00:00 0                          [vvar]
7ffc2c877000-7ffc2c879000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

# Linux memory management

Virtual memory <—> physical memory

Memory Management Unit



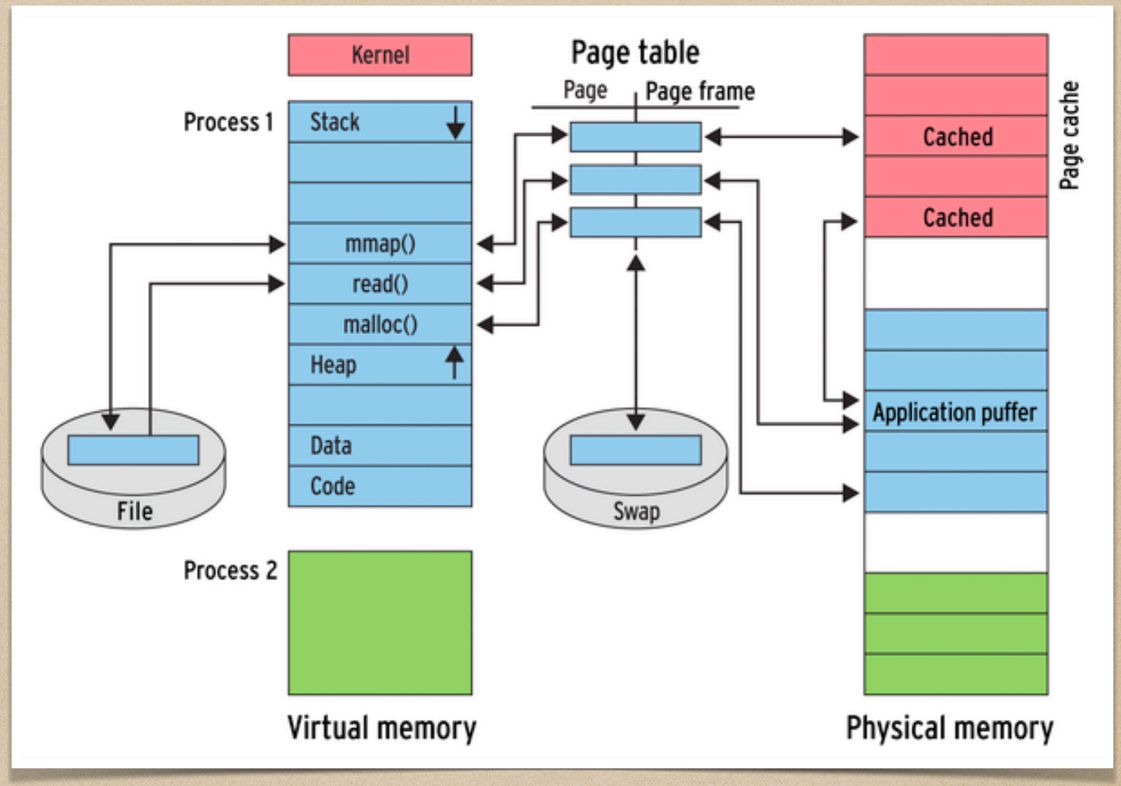



# Linux memory management

Virtual memory <—> physical memory

Page table. pgtable_types.h

_PAGE_BIT_PRESENT …



# Page management in Linux kernel

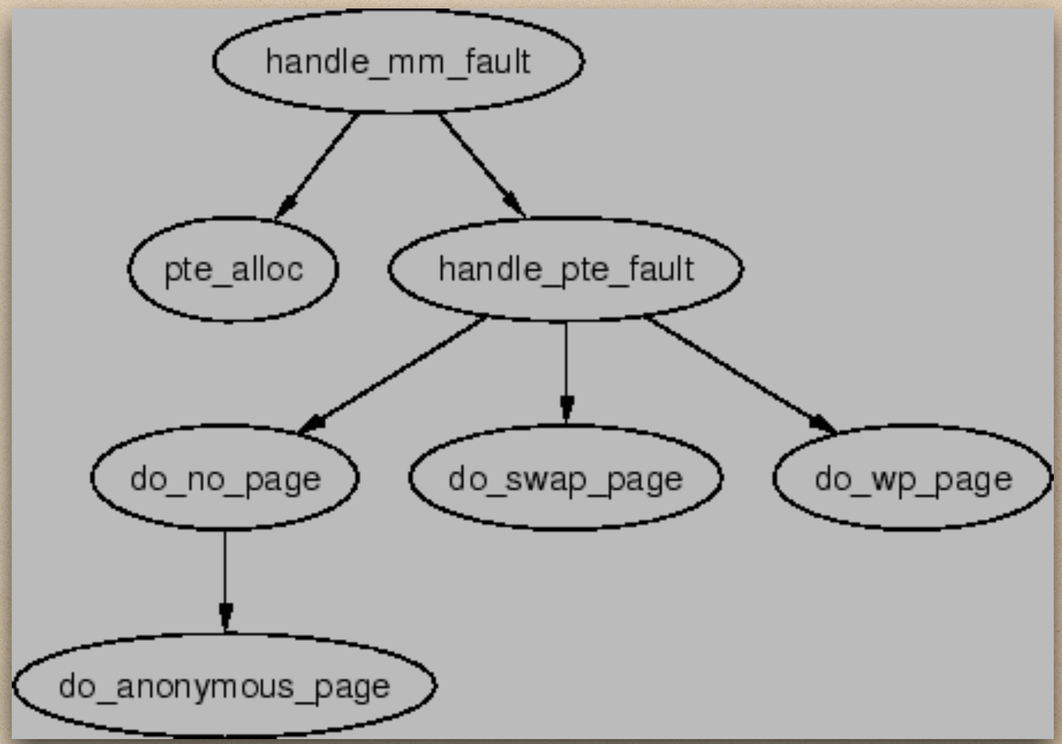
- Page fault
  - Lazy allocation
  - Major fault
    - IO involved
  - Minor fault

```
#ps -eo min_flt,maj_flt,cmd
 MINFL  MAJFL CMD
 24357     64 /usr/lib/systemd/systemd --switched-root --system --deserialize 22
    93      0 /sbin/auditd
   126      0 /sbin/audispd
   210      0 /usr/sbin/sedispatch
   101      0 /sbin/rpcbind -w
  1180      0 /usr/sbin/abrtd -d -s
   439      0 /sbin/rngd -f
   789      1 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --
  1213      1 /usr/lib/polkit-1/polkitd --no-debug
  2883      0 /usr/lib/systemd/systemd-logind
   340      0 avahi-daemon: running [aliy80.local]
   151      0 /usr/bin/lsmd -d
   172      0 /usr/sbin/gssproxy -D
    31      0 avahi-daemon: chroot helper
   664      0 /usr/sbin/ModemManager
   503      0 /usr/sbin/smartd -n -q never
   285      0 /usr/sbin/chronyd
137110      0 /bin/bash /usr/sbin/ksmtuned
    71      0 /sbin/dhclient -1 -q -lf /var/lib/dhclient/dhclient--eth0.lease -pf /var/
th0
 53208      0 /usr/bin/python2 -Es /usr/sbin/tuned -l -P
   583     13 /usr/sbin/cupsd -f
   175      0 /usr/bin/rhsmcertd
258526    918 /usr/sbin/rsyslogd -n
  5907     63 /usr/sbin/aliyun-service
  6160    388 /usr/bin/containerd
  8621      1 /usr/sbin/libvirtd
 24708    744 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
   156      0 /sbin/agetty --noclear tty1 linux
  3983      0 /usr/sbin/crond -n
   187      0 /usr/sbin/atd -f
   151      0 /sbin/agetty --keep-baud 115200,38400,9600 ttyS0 vt220
  1358      6 /usr/local/aegis/aegis_update/AliYunDunUpdate
   890      0 /usr/sbin/sshd -D
```

# Page management in Linux kernel

## Page fault

- cr2
- arch/x86/mm/fault.c
- mm/memory.c



# Page management in Linux kernel

How user application get a page:

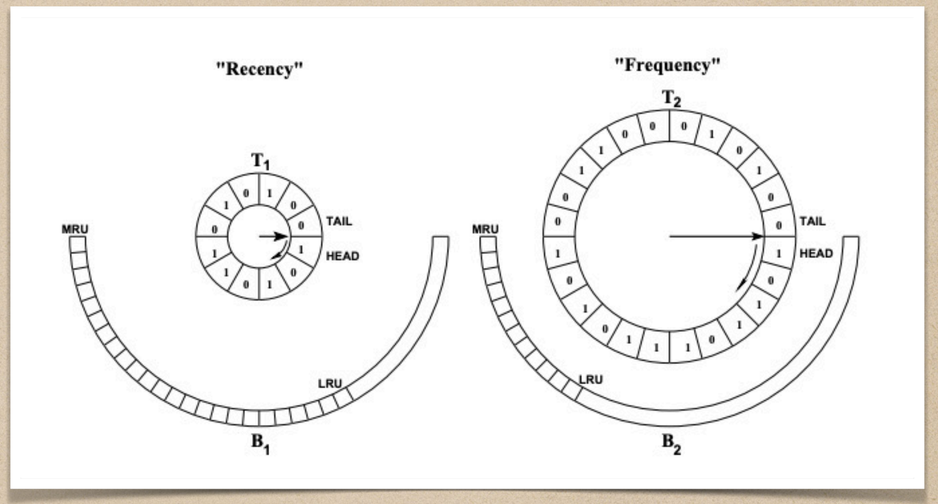
Steps of page swap in

1, virtual address to physical address
2, not in TLB, not in Page Table —> page fault
3, PTE filled w/o present bit
4, Is it in swapcache?
5, get a page frame, read it from swap device
6, Add it into swap cache
7, Add it into LRU
8, Add it into PageTable
9, charge it in memcg
10, Add_anon_rmap
11, update mmu_cache — optional

# Page mangement in Linux kernel

https://linux-mm.org/AdvancedPageReplacement

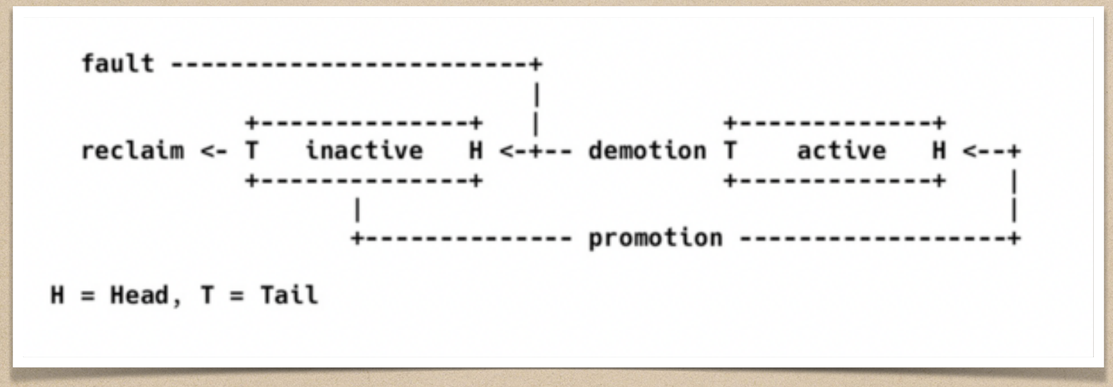
ARC
Clock-Pro
CAR
LRU



# Page management in Linux Kernel

Last Recent Used

- lru_lock
- PG_lru
- Lruvec.lists[NR_LRU_LISTS]




```
[root@aliy80 ~]# cat /proc/meminfo
MemTotal:       196149696 kB
MemFree:        189579036 kB
MemAvailable:   193416976 kB
Buffers:           335480 kB
Cached:           4457020 kB
SwapCached:             0 kB
Active:           1564632 kB
Inactive:         3435256 kB
Active(anon):         376 kB
Inactive(anon):    204940 kB
Active(file):     1564256 kB
Inactive(file):   3230316 kB
Unevictable:            0 kB
Mlocked:                0 kB
SwapTotal:        2097148 kB
SwapFree:         2097148 kB
Dirty:                180 kB
Writeback:              0 kB
AnonPages:         207468 kB
Mapped:            187208 kB
Shmem:               2688 kB
KReclaimable:      412120 kB
Slab:              656360 kB
SReclaimable:      412120 kB
SUnreclaim:        244240 kB
KernelStack:        17008 kB
PageTables:         10668 kB
```

# Memcg and lru lock optimization

Lru lock protected objects:

A, PG_lru
B, List integrity
C, PG_mlocked, munlock/split
D, Memcg->move_lock
E, l_pages lock
F, Page Idle

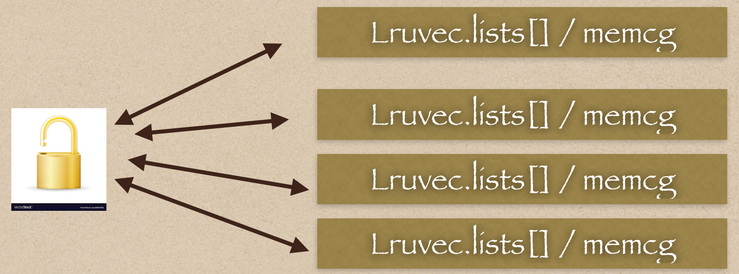
Lru lock timing:

A, Add page into lru lists
B, Delete it from lru lists
C, Moving pages between lru lists
D, Isolation pages
  D1, reclaim
  D2, compaction
  D3, migrations
  D4, munlock
E, Put page back

# Memcg and lru lock optimization

Before Mem cgroup


After Mem cgroup



# Memcg and lru lock optimization

Per node Lru lock


Per memcg lru lock

# Memcg and lru lock optimization

Is that Simple?

# Memcg and lru lock optimization

To guard page's memcg change:

A, relock on lru_lock

B, lock_page_memcg

C, TestClearPageLRU

```
+struct lruvec *lock_page_lruvec_irq(struct page *page,
+                                    struct pglist_data *pgdat)
+{
+       struct lruvec *lruvec;
+
+again:
+       rcu_read_lock();
+       lruvec = mem_cgroup_page_lruvec(page, pgdat);
+       spin_lock_irq(&lruvec->lru_lock);
+       rcu_read_unlock();
+
+       /* lruvec may changed in commit_charge() */
+       if (lruvec != mem_cgroup_page_lruvec(page, pgdat)) {
+               spin_unlock_irq(&lruvec->lru_lock);
+               goto again;
+       }
+
+       return lruvec;
+}
+
```

# Memcg and lru lock optimization

How user application get a page:

Steps of page swap in

1, virtual address to physical address
2, not in TLB, check Page Table
3, PTE show it is in swap device
4, Is it in swapcache?
5, get a page frame, read it from swap device
6, Add it into swap cache
7, Add it into LRU. Pending in pagevec
8, Add it into PageTable
9, Charge it in memcg
10, Add_anon_rmap
11, update mmu_cache

Move step 9 before step 7

# Memcg and lru lock optimization

commit 4c6355b25e
mm: memcontrol: charge swapin pages on instantiation

Right now, users that are otherwise memory controlled can easily escape their containment and allocate significant amounts of memory that they're not being charged for. That's because swap readahead pages are not being charged until somebody actually faults them into their page table. This can be exploited with MADV_WILLNEED, which triggers arbitrary readahead allocations without charging the pages.

There are additional problems with the delayed charging of swap pages:

1. To implement refault/workingset detection for anonymous pages, we need to have a target LRU available at swapin time, but the LRU is not determinable until the page has been charged.

2. To implement per-cgroup LRU locking, we need page->mem_cgroup to be stable when the page is isolated from the LRU; otherwise, the locks change under us. But swapcache gets charged after it's already on the LRU, and even if we cannot isolate it ourselves (since charging is not exactly optional).

The previous patch ensured we always maintain cgroup ownership records for swap pages. This patch moves the swapcache charging point from the fault handler to swapin time to fix all of the above problems.

```
+	/*
+	 * The swap entry is ours to swap in. Prepare the new page.
+	 */
+
+	__SetPageLocked(page);
+	__SetPageSwapBacked(page);
+
+	/* May fail (-ENOMEM) if XArray node allocation failed. */
+	if (add_to_swap_cache(page, entry, gfp_mask & GFP_KERNEL)) {
+		put_swap_page(page, entry);
+		goto fail_unlock;
+	}
+
+	if (mem_cgroup_charge(page, NULL, gfp_mask, false)) {
+		delete_from_swap_cache(page);
+		goto fail_unlock;
+	}
+
+	/* Caller will initiate read into locked page */
+	SetPageWorkingset(page);
+	lru_cache_add_anon(page);
+	*new_page_allocated = true;
+	return page;
```

# Memcg and lru lock optimization

## Memory Lock sequence change

```
2.6 Locking
-----------

-   lock_page_cgroup()/unlock_page_cgroup() should not be called under
-   the i_pages lock.
+Lock order is as follows:

-   Other lock order is following:
+  Page lock (PG_locked bit of page->flags)
+    mm->page_table_lock or split pte_lock
+      lock_page_memcg (memcg->move_lock)
+          mapping->i_pages lock
+          lruvec->lru_lock.

-   PG_locked.
-     mm->page_table_lock
-         pgdat->lru_lock
-          lock_page_cgroup.
-
-  In many cases, just lock_page_cgroup() is called.
-
-  per-zone-per-cgroup LRU (cgroup's private LRU) is just guarded by
-  pgdat->lru_lock, it has no lock of its own.
+Per-node-per-memcgroup LRU (cgroup's private LRU) is guarded by
+lruvec->lru_lock; PG_lru bit of page->flags is cleared before
+isolating a page from its LRU under lruvec->lru_lock.
```

```
 * Lock ordering in mm:
 *
 * inode->i_mutex      (while writing or truncating, not reading or faulting)
 *   mm->mmap_lock
 *     page->flags PG_locked (lock_page)   * (see huegtlbfs below)
 *       hugetlbfs_i_mmap_rwsem_key (in huge_pmd_share)
 *         mapping->i_mmap_rwsem
 *           hugetlb_fault_mutex (hugetlbfs specific page fault mutex)
 *           anon_vma->rwsem
 *             mm->page_table_lock or pte_lock
- *               pgdat->lru_lock (in mark_page_accessed, isolate_lru_page)
 *               swap_lock (in swap_duplicate, swap_info_get)
 *                 mmlist_lock (in mmput, drain_mmlist and others)
 *                 mapping->private_lock (in __set_page_dirty_buffers)
- *                   mem_cgroup_{begin,end}_page_stat (memcg->move_lock)
+ *                   lock_page_memcg move_lock (in __set_page_dirty_buffers)
 *                     i_pages lock (widely used)
+ *                       lruvec->lru_lock (in lock_page_lruvec_irq)
 *                 inode->i_lock (in set_page_dirty's __mark_inode_dirty)
 *                 bdi.wb->list_lock (in set_page_dirty's __mark_inode_dirty)
 *                   sb_lock (within inode_lock in fs/fs-writeback.c)
 *                   i_pages lock (widely used, in set_page_dirty,
 *                             in arch-dependent flush_dcache_mmap_lock,
 *                             within bdi.wb->list_lock in __sync_single_inode)
 *
 * anon_vma->rwsem,mapping->i_mutex      (memory_failure, collect_procs_anon)
 *   ->tasklist_lock
 *     pte map lock
```

# Memcg and lru lock optimization

To guard page's memcg change:

A, relock on lru_lock

B, lock_page_memcg

C, TestClearPageLRU

```
+struct lruvec *lock_page_lruvec_irq(struct page *page)
+{
+       struct lruvec *lruvec;
+       struct mem_cgroup *memcg;
+
+       memcg = lock_page_memcg(page);
+       lruvec = mem_cgroup_lruvec(memcg, page_pgdat(page));
+       spin_lock_irq(&lruvec->lru_lock);
+
+       return lruvec;
+}
```

# Memcg and lru lock optimization

To guard page's memcg change:

A, relock on lru_lock

B, lock_page_memcg

C, TestClearPageLRU

Currently lru_lock still guards both lru list and page's lru bit, that's ok. but if we want to use specific lruvec lock on the page, we need to pin down the page's lruvec/memcg during locking. Just taking lruvec lock first may be undermined by the page's memcg charge/migration. To fix this problem, we could clear the lru bit out of locking and use it as pin down action to block the page isolation in memcg changing.

So now a standard steps of page isolation is following:

```
  1, get_page();              #pin the page avoid to be free
+    if TestClearPageLRU()   #block other isolation like memcg change
  2, spin_lock on lru_lock; #serialize lru list access
-    ClearPageLRU();
  3, delete page from lru list;
```

The step 2 could be optimzed/replaced in scenarios which page is unlikely be accessed or be moved between memcgs.

This patch start with the first part: TestClearPageLRU, which combines PageLRU check and ClearPageLRU into a macro func TestClearPageLRU. This function will be used as page isolation precondition to prevent other isolations some where else. Then there are may !PageLRU page on lru list, need to remove BUG() checking accordingly.

There 2 rules for lru bit now:

1, the lru bit still indicate if a page on lru list, just in some temporary moment(isolating), the page may have no lru bit when it's on lru list. but the page still must be on lru list when the lru bit set.

2, have to remove lru bit before delete it from lru list.

# Memcg and lru lock optimization

Lru lock Protected object:

- A, PG_lru
  B, List integrity

- C, PG_mlocked, unlock/split
- D, Memcg->move_lock
- E, I_pages lock
- F, Page Idle

Lru lock normal usages:

A, Add page into lru lists
B, Delete it from lru lists
C, Moving pages between lru lists
D, Isolation pages
  D1, reclaim
  D2, compaction
  D3, migrations
  D4, munlock
E, Put page back

# Memcg and lru lock optimization

Testing result:

New test case:

Per memcg lru-file-readtwice

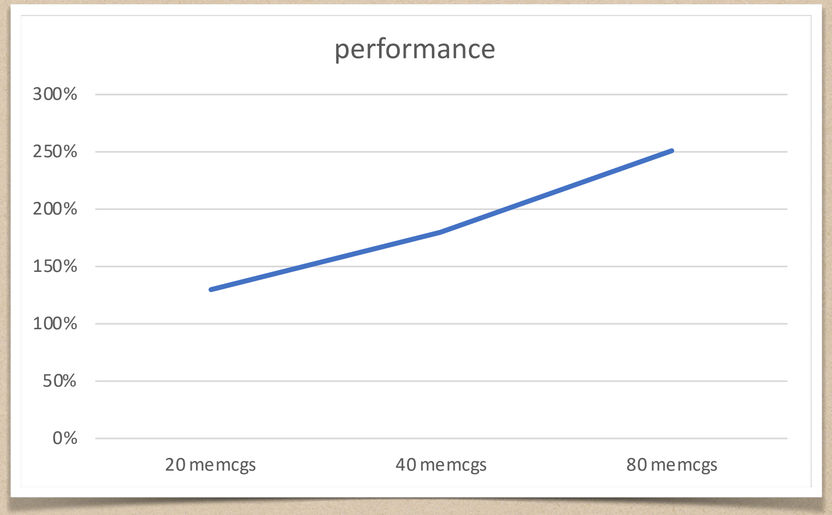
2 dd in a memcg

Dockerfile, vm-scalablity patch

https://lkml.org/lkml/2020/8/26/212

Daniel Jordan

https://lkml.org/lkml/2020/9/24/1054



# Memcg and lru lock optimization

Any shortages?

1, Extra atomic write when page not in lru
isolate_lru_page()

# Further chances

Further optimization:

- Sort lru lists before relocking
- Fairness locking issue

```
static void pagevec_lru_move_fn(struct pagevec *pvec,
        void (*move_fn)(struct page *page, struct lruvec *lruvec))
{
        int i;
        struct lruvec *lruvec = NULL;
        unsigned long flags = 0;

        for (i = 0; i < pagevec_count(pvec); i++) {
                struct page *page = pvec->pages[i];

                /* block memcg migration during page moving between lru */
                if (!TestClearPageLRU(page))
                        continue;

                lruvec = relock_page_lruvec_irqsave(page, lruvec, &flags);
                (*move_fn)(page, lruvec);

                SetPageLRU(page);
        }
        if (lruvec)
                unlock_page_lruvec_irqrestore(lruvec, flags);
        release_pages(pvec->pages, pvec->nr);
        pagevec_reinit(pvec);
}
```

# Further chances

Further optimization:

Per lru lock for each of lists

```
enum lru_list {
        LRU_INACTIVE_ANON = LRU_BASE,
        LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
        LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
        LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
        LRU_UNEVICTABLE,
        NR_LRU_LISTS
};

Struct lurvec {
        Struct list_head lists[NR_LRU_LISTS];
        spinlock_t       lru_lock;
```

# Upstreaming Status

First proposal:

Hugh Dickins & Konstantin Khlebnikov

https://fa.linux.kernel.narkive.com/9UwfrOeI/patch-0-10-mm-memcg-per-memcg-per-zone-lru-locking

Back in LKML:

Last Oct
Johannes Weiner, Feb, suggest TestClearPageLRU
Than Found memcg charge timing wrong,
V13 finished the main solution

Review:
Alexander Dukcy, reviewed 5 weeks in July 2020
Hugh Dickins reviewed 4 weeks in Sep 2020

Questions &
Thanks !

# Others

LKP found fio.readiops -30%

Reason:
    Qspinlock false sharing

```
diff --git a/include/linux/mmzone.h b/include/linux/mmzone.h
index a75e6d0effcb..58b21bffef95 100644
--- a/include/linux/mmzone.h
+++ b/include/linux/mmzone.h
@@ -272,9 +272,9 @@ enum lruvec_flags {
 };

 struct lruvec {
+       struct list_head                lists[NR_LRU_LISTS];
        /* per lruvec lru_lock for memcg */
        spinlock_t                      lru_lock;
-       struct list_head                lists[NR_LRU_LISTS];
        /*
         * These track the cost of reclaiming one LRU - file or anon -
         * over the other. As the observed cost of reclaiming one LRU


enum lru_list {
        LRU_INACTIVE_ANON = LRU_BASE,
        LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
        LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
        LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
        LRU_UNEVICTABLE,
        NR_LRU_LISTS
};
```