Trace event cases in slow path of memory reclaiming

August, 2017, Beijing

Joey Lee
SUSE Labs Taipei

Agenda

- Definition of Terms
- Slow path in page reclaim
- Trace event in kswapd
- Trace event in direct reclaim
- Trace event in compaction context
- IO congestion [Gary]
- Two cases
- Q&A

Definition of Terms

Anonymous and File-backed

- Anonymous mapping maps an area of the process's virtual memory not backed by any file.
- File-backed mapping maps an area of the process's virtual memory to files; i.e. reading those areas of memory causes the file to be read. It is the default mapping type. [6]

Anonymous and File-backed (cont.)

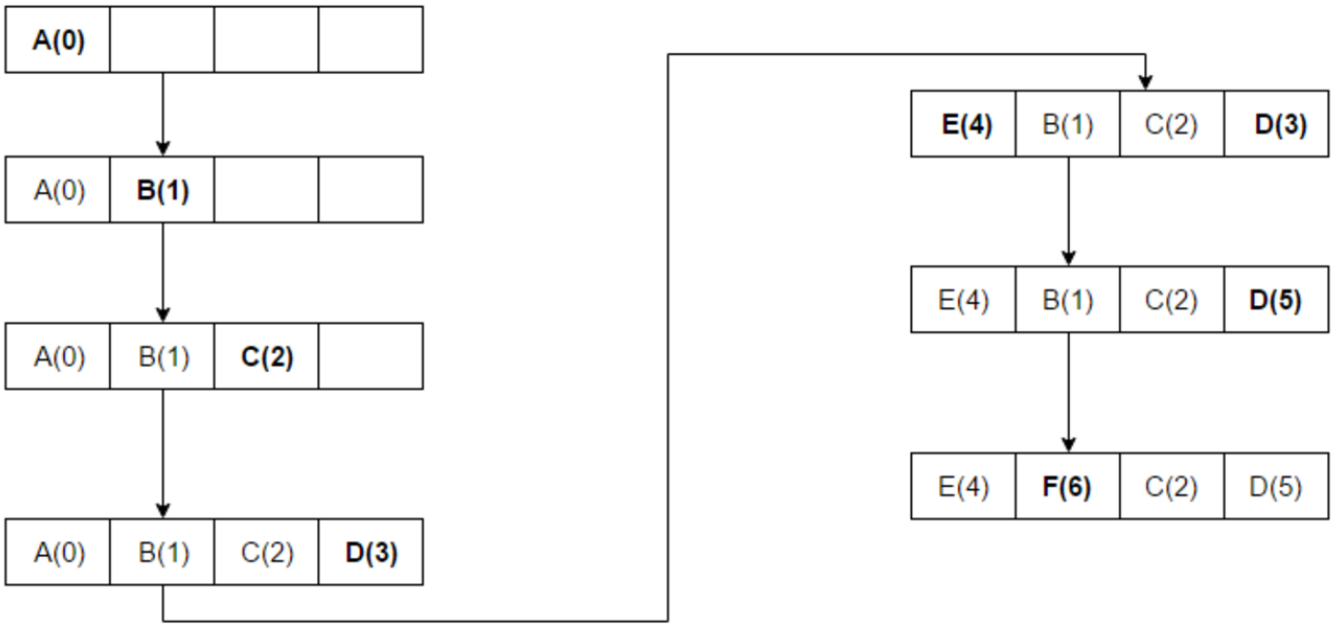
- Anonymous Private
  - stack
  - malloc()
  - mmap(): MAP_ANONYMOUS + MAP_PRIVATE
  - brk()/sbrk()
- Anonymous Shared
  - mmap(): MAP_ANONYMOUS + MAP_SHARED
- File-backed Private
  - mmap(): fd + MAP_PRIVATE
  - Binary/shared libraries
- File-backed Shared
  - mmap(): fd + MAP_SHARED

Least Recently Used (LRU)

- Discards the least recently used items first. ...General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. [7]
- For the page replacement policy, pages that may be swapped out will exist on either the active_list or the inactive_list declared in page_alloc.c . This is the list head for these LRU lists. These two lists are discussed in detail in Chapter 10; [4]

Least Recently Used (Cont.)

The access sequence for the below example is A B C D E D F.



In the above example once A B C D gets installed in the blocks with sequence numbers (Increment 1 for each new Access) and when E is accessed, it is a miss and it needs to be installed in one of the blocks. According LRU Algorithm, since A has the lowest Rank(A(0)), E will replace A.

Cache replacement policies#LRU [7]

Active and Inactive lists

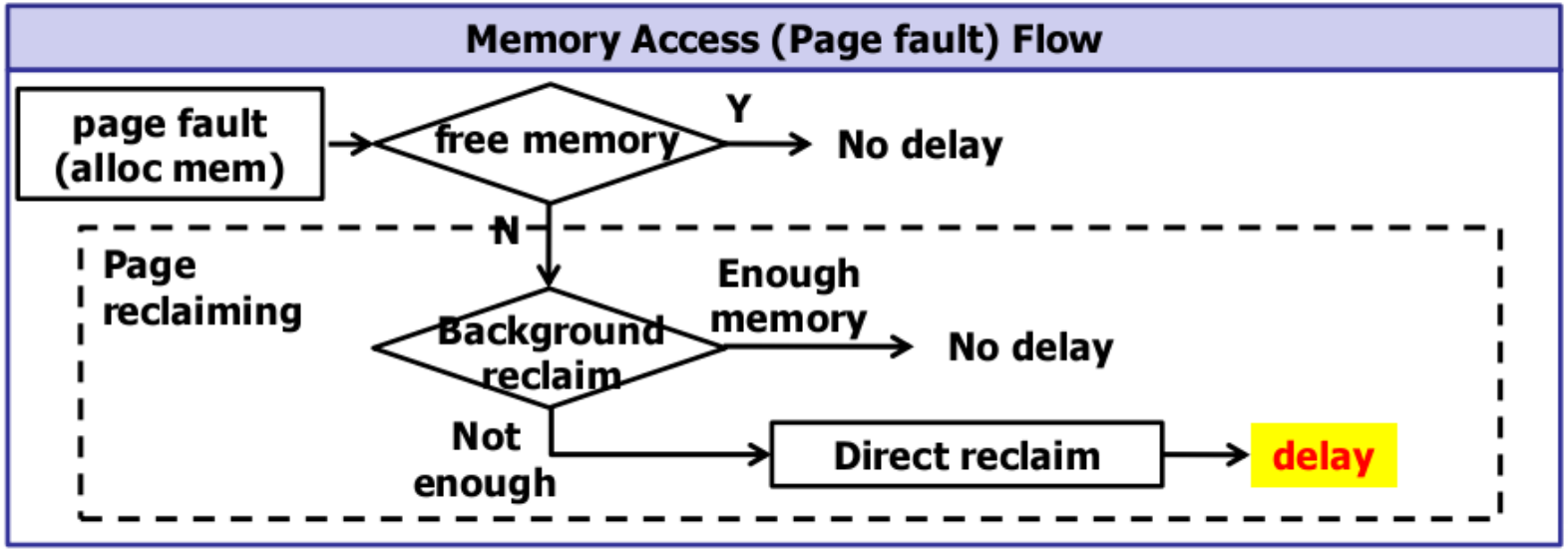
- During discussions the page replacement policy is frequently said to be a Recently Used (LRU) -based Least algorithm but this is not strictly speaking true as the lists are not strictly maintained in LRU order.
- The LRU in Linux consists of two lists called the active_list and inactive_list .
- The objective is for the active_list working set [Den70] of all processes and the inactive_list to contain to contain the reclaim canditates. [4]

Slow path in page reclaim

Page reclaim

- If Linux runs short of memory, it reclaims used pages and then allocate new pages [1]
- There are 2 type of reclaim
  - Background reclaim (kswapd)
    - The kernel may reclaim anon pages (swapout/swapin)
  - Foreground reclaim (direct reclaim)
    - The kernel reclaims memory and then allocates memory (direct reclaim)
    - Reclaim pages in process's context [1]

Page reclaim flow



•Satoru Moriya [1]

May slow I/O in slow path

- 2 issues for memory access latency
  - Reclaim in page alloc path (direct reclaim)
    - It takes some time
    - May need I/O
  - swapout/swapin
    - Put out anon pages to disk
    - Need I/O to read data from disk at next access [1]

Page allocation context

```
mm/page_alloc.c unsigned long get_zeroed_page(gfp_t gfp_mask)
                        /* Common helper functions. */
    mm/page_alloc.c     unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
        include/linux/gfp.h     static inline struct page *alloc_pages(gfp_t gfp_mask, unsigned int order)

include/linux/gfp.h     #define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
    include/linux/gfp.h     static inline struct page *alloc_pages(gfp_t gfp_mask, unsigned int order)
                        /* alloc_pages_current - Allocate pages. */
        mm/mempolicy.c  struct page *alloc_pages_current(gfp_t gfp, unsigned order)
                                /* This is the 'heart' of the zoned buddy allocator. */
            mm/page_alloc.c     struct page *__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
                                                                    int preferred_nid, nodemask_t *nodemask)
                mm/page_alloc.c static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order,
                                                                    struct alloc_context *ac)
```

Page fault handler context

```
arch/x86/mm/fault.c    dotraplinkage void notrace do_page_fault(struct pt_regs *regs, unsigned long error_code)
                       /* This routine handles page faults. */
    arch/x86/mm/fault.c    static noinline void __do_page_fault(struct pt_regs *regs, unsigned long error_code, unsigned long address)
                           /* 64-bit: Handle a fault on the vmalloc area */
        arch/x86/mm/fault.c    static noinline int vmalloc_fault(unsigned long address)
                       /* handle the page fault from userland good area */
            mm/memory.c    int handle_mm_fault(struct vm_area_struct *vma, unsigned long address, unsigned int flags)
                mm/memory.c    static int __handle_mm_fault(struct vm_area_struct *vma, unsigned long address, unsigned int flags)
                    mm/memory.c    static int handle_pte_fault(struct vm_fault *vmf)
                        mm/memory.c    int do_swap_page(struct vm_fault *vmf)
                                       /* swap in pages in hope we need them soon */
                            mm/swap_state.c     struct page *swapin_readahead(swp_entry_t entry, gfp_t gfp_mask,
                                                                              struct vm_area_struct *vma, unsigned long addr)
                                       /* Locate a page of swap in physical memory, reserving swap cache space and
                                        * reading the disk if it is not already cached. */
                                mm/swap_state.c struct page *read_swap_cache_async(swp_entry_t entry, gfp_t gfp_mask,
                                                                                   struct vm_area_struct *vma, unsigned long addr,
                                                                                   bool do_poll)
                                    mm/swap_state.c struct page *__read_swap_cache_async(swp_entry_t entry, gfp_t gfp_mask,
                                                                                         struct vm_area_struct *vma, unsigned long addr,
                                                                                         bool *new_page_allocated)
                                        include/linux/gfp.h     #define alloc_page_vma(gfp_mask, vma, addr)                   \
                                                                        alloc_pages_vma(gfp_mask, 0, vma, addr, numa_node_id(), false)
                                                                /* alloc_pages_vma - Allocate a page for a VMA. */
                                            mm/mempolicy.c      struct page *alloc_pages_vma(gfp_t gfp, int order, struct vm_area_struct *vma,
                                                                                             unsigned long addr, int node, bool hugepage)
                                                                /* This is the 'heart' of the zoned buddy allocator. */
                                                mm/page_alloc.c struct page *__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
                                                                                                    int preferred_nid, nodemask_t *nodemask)
                                                    mm/page_alloc.c static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order,
                                                                                                                       struct alloc_context *ac)
```

__alloc_pages_slowpath

```
static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order, struct alloc_context *ac)
  retry_cpuset:
        wake_all_kswapds(order, ac);
        /* The adjusted alloc_flags might result in immediate success, so try that first */
        page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
                if (page)       goto got_pg;
        /* For costly allocations, try direct compaction first, as it's likely that we have enough base pages and don't need to reclaim. */
        page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flags, ac, INIT_COMPACT_PRIORITY, &compact_result);
                if (page) goto got_pg;
  retry:
        /* Ensure kswapd doesn't accidentally go to sleep as long as we loop */
        wake_all_kswapds(order, ac);
        /* Attempt with potentially adjusted zonelist and alloc_flags */
        page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
                if (page)       goto got_pg;
        /* Try direct reclaim and then allocating */
        page = __alloc_pages_direct_reclaim(gfp_mask, order, alloc_flags, ac, &did_some_progress);
                if (page)       goto got_pg;
        /* Try direct compaction and then allocating */
        page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flags, ac, compact_priority, &compact_result);
                if (page)       goto got_pg;
        /* Checks whether it makes sense to retry the reclaim to make a forward progress for the given allocation request. */
        if (should_reclaim_retry(gfp_mask, order, ac, alloc_flags, did_some_progress > 0, &no_progress_loops))
                goto retry;
        /* Deal with possible cpuset update races before we start OOM killing */
        if (check_retry_cpuset(cpuset_mems_cookie, ac))
                goto retry_cpuset;
        /* Reclaim has failed us, start killing things *//* OOM */
        page = __alloc_pages_may_oom(gfp_mask, order, ac, &did_some_progress);
                if (page)       goto got_pg;
        /* Deal with possible cpuset update races before we fail */
        if (check_retry_cpuset(cpuset_mems_cookie, ac))
                goto retry_cpuset;
  got_pg:
        return page;
```

v4.12 mainline

__alloc_pages_slowpath (cont.)

```
static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order, struct alloc_context *ac)
  retry_cpuset:
        wake_all_kswapds(order, ac);
        /* The adjusted alloc_flags might result in immediate success, so try that first */
        page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
                if (page)       goto got_pg;
        /* For costly allocations, try direct compaction first, as it's likely that we have enough base pages and don't
        page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flags, ac, INIT_COMPACT_PRIORITY, &compact_result);
                if (page) goto got_pg;
  retry:
        /* Ensure kswapd doesn't accidentally go to sleep as long as we loop */
        wake_all_kswapds(order, ac);
        /* Attempt with potentially adjusted zonelist and alloc_flags */
        page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
                if (page)       goto got_pg;
        /* Try direct reclaim and then allocating */
        page = __alloc_pages_direct_reclaim(gfp_mask, order, alloc_flags, ac, &did_some_progress);
                if (page)       goto got_pg;
        /* Try direct compaction and then allocating */
        page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flags, ac, compact_priority, &compact_result);
                if (page)       goto got_pg;
```

Trace event in kswapd

Kswapd initialization context

```
mm/vmscan.c     module_init(kswapd_init)
mm/vmscan.c     static int __init kswapd_init(void)
                /* This kswapd start function will be called by init and node-hot-add.
                 * On node-hot-add, kswapd will moved to proper cpus if cpus are hot-added. */
    mm/vmscan.c int kswapd_run(int nid)
                    /* The background pageout daemon, started as a kernel thread from the init process. */
        mm/vmscan.c     static int kswapd(void *p)
            mm/vmscan.c     static void kswapd_try_to_sleep(pg_data_t *pgdat, int alloc_order, int reclaim_order,
                                                    unsigned int classzone_idx)
                                /* Prepare kswapd for sleeping. Returns true if kswapd is ready to sleep */
                mm/vmscan.c     static bool prepare_kswapd_sleep(pg_data_t *pgdat, int order, int classzone_idx)
                kernel/sched/core.c     asmlinkage __visible void __sched schedule(void)
                        /*
                         * For kswapd, balance_pgdat() will reclaim pages across a node from zones
                         * that are eligible for use by the caller until at least one zone is
                         * balanced.
                         *
                         * Returns the order kswapd finished reclaiming at.
                         *
                         * kswapd scans the zones in the highmem->normal->dma direction.  It skips
                         * zones which have free_pages > high_wmark_pages(zone), but once a zone is
                         * found to have free_pages <= high_wmark_pages(zone), any page is that zone
                         * or lower is eligible for reclaim until at least one usable zone is
                         * balanced.
                         */
            mm/vmscan.c static int balance_pgdat(pg_data_t *pgdat, int order, int classzone_idx)
                mm/vmscan.c     static bool allow_direct_reclaim(pg_data_t *pgdat)
```

Kswapd()
mm_vmscan_kswapd_wake

```
static int kswapd(void *p)
{
        unsigned int alloc_order, reclaim_order;
        unsigned int classzone_idx = MAX_NR_ZONES - 1;
        pg_data_t *pgdat = (pg_data_t*)p;
        struct task_struct *tsk = current;
        [...snip]
        for ( ; ; ) {
                [...snip]
kswapd_try_sleep:
                kswapd_try_to_sleep(pgdat, alloc_order, reclaim_order,
                                        classzone_idx);
                [...snip]
                /*
                 * Reclaim begins at the requested order but if a high-order
                 * reclaim fails then kswapd falls back to reclaiming for
                 * order-0. If that happens, kswapd will consider sleeping
                 * for the order it finished reclaiming at (reclaim_order)
                 * but kcompactd is woken to compact for the original
                 * request (alloc_order).
                 */
                trace_mm_vmscan_kswapd_wake(pgdat->node_id, classzone_idx,
                                                alloc_order);
                reclaim_order = balance_pgdat(pgdat, alloc_order, classzone_idx);
                if (reclaim_order < alloc_order)
                        goto kswapd_try_sleep;
        }
        [...snip]
        return 0;
}
```

kswapd_try_to_sleep()
mm_vmscan_kswapd_sleep

```
static void kswapd_try_to_sleep(pg_data_t *pgdat, int alloc_order, int reclaim_order,
                                unsigned int classzone_idx)
{
        long remaining = 0;
        DEFINE_WAIT(wait);
        [...snip]
        prepare_to_wait(&pgdat->kswapd_wait, &wait, TASK_INTERRUPTIBLE);

        /* Try to sleep for a short interval. Note that kcompactd will only be
         [...snip]
         */
        if (prepare_kswapd_sleep(pgdat, reclaim_order, classzone_idx)) {
                [...snip]
        }

        /* After a short sleep, check if it was a premature sleep. If not, then
         * go fully to sleep until explicitly woken up.
         */
        if (!remaining &&
            prepare_kswapd_sleep(pgdat, reclaim_order, classzone_idx)) {
                trace_mm_vmscan_kswapd_sleep(pgdat->node_id);

                /* vmstat counters are not perfectly accurate and the estimated
                 * value for counters such as NR_FREE_PAGES can deviate from the
                [...snip]
                 */
                set_pgdat_percpu_threshold(pgdat, calculate_normal_threshold);

                if (!kthread_should_stop())
                        schedule();

                set_pgdat_percpu_threshold(pgdat, calculate_pressure_threshold);
        } else {
        [...snip]
        finish_wait(&pgdat->kswapd_wait, &wait);
}
```

Watermark

- balance_pgdat
- calculate_normal_threshold
- calculate_normal_threshold

mm_vmscan_wakeup_kswapd

```
mm/page_alloc.c static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order,
                                                                  struct alloc_context *ac)
    mm/page_alloc.c static void wake_all_kswapds(unsigned int order, const struct alloc_context *ac)
                        /* A zone is low on free memory, so wake its kswapd task to service it. */
        mm/vmscan.c     void wakeup_kswapd(struct zone *zone, int order, enum zone_type classzone_idx)
                        {
                                pg_data_t *pgdat;
                                [...snip]
                                pgdat = zone->zone_pgdat;
                                pgdat->kswapd_classzone_idx = kswapd_classzone_idx(pgdat,
                                                                                   classzone_idx);
                                pgdat->kswapd_order = max(pgdat->kswapd_order, order);
                                if (!waitqueue_active(&pgdat->kswapd_wait))
                                        return;

                                /* Hopeless node, leave it to direct reclaim */
                                if (pgdat->kswapd_failures >= MAX_RECLAIM_RETRIES)
                                        return;

                                if (pgdat_balanced(pgdat, order, classzone_idx))
                                        return;

                                trace_mm_vmscan_wakeup_kswapd(pgdat->node_id, classzone_idx, order);
                                wake_up_interruptible(&pgdat->kswapd_wait);
                        }
```

Systemtap mm_vmscan_kswapd_* exercise

- mm_vmscan_kswapd_wake
- mm_vmscan_kswapd_sleep
- mm_vmscan_wakeup_kswapd

stress-ng

- zypper -v in stress-ng
- stress-ng --vm 1 --vm-bytes 2048M --vm-keep --verbose

Systemtap mm_vmscan_kswapd_* example

```
global wakeup_kswapd
global kswapd_node

probe kernel.trace("mm_vmscan_wakeup_kswapd") {
        if (@defined($nid) && !wakeup_kswapd[$nid]) {
                wakeup_kswapd[$nid] = gettimeofday_us();
                printf("mm_vmscan_wakeup_kswapd node id: %d\n", $nid);
        }
}
probe kernel.trace("mm_vmscan_kswapd_wake") {
        if (@defined($nid) && !kswapd_node[$nid]) {
                kswapd_node[$nid] = gettimeofday_us();
                printf("mm_vmscan_kswapd_wake   node id: %d\n", $nid);
        }
}
probe kernel.trace("mm_vmscan_kswapd_sleep") {
        if (@defined($nid) && kswapd_node[$nid]) {
                waked = gettimeofday_us() - kswapd_node[$nid]
                printf("mm_vmscan_kswapd_sleep  node id: %d     wake up period: %d(us) %d(s)\n", $nid, waked, waked/1000000);
                kswapd_node[$nid] = NULL;
        }
}
```

Systemtap mm_vmscan_kswapd_* result

```
linux-g35h:/home/linux/tmp/kernel-tracing-tools-training/systemtap-test/mm # stap mm_vmscan_kswapd.stp
mm_vmscan_wakeup_kswapd node id: 0
mm_vmscan_kswapd_wake   node id: 0
mm_vmscan_kswapd_sleep  node id: 0      wake up period: 14350563(us) 14(s)
mm_vmscan_kswapd_wake   node id: 0
mm_vmscan_kswapd_sleep  node id: 0      wake up period: 4161585(us) 4(s)
mm_vmscan_kswapd_wake   node id: 0
mm_vmscan_kswapd_sleep  node id: 0      wake up period: 10310732(us) 10(s)
mm_vmscan_kswapd_wake   node id: 0
mm_vmscan_kswapd_sleep  node id: 0      wake up period: 17113421(us) 17(s)
mm_vmscan_kswapd_wake   node id: 0
```

Trace event in direct reclaim

Trace when allocation stalls occur

- Ftrace vmscan/mm_vmscan_direct_reclaim_begin will get a count of how many times stalls occur. It's equivalent to monitoring /proc/vmstat.
- However, if combined with vmscan/mm_vmscan_direct_reclaim_end then the total time spent on each stall can be calculated using either a post-processing script or systemtap. [Mel Gorman]

__alloc_pages_direct_reclaim context

```
mm/page_alloc.c static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order,
                                                                  struct alloc_context *ac)
                    /* The really slow allocator path where we enter direct reclaim */
    mm/page_alloc.c     static inline struct page *__alloc_pages_direct_reclaim(gfp_t gfp_mask, unsigned int order,
                                        unsigned int alloc_flags, const struct alloc_context *ac,
                                        unsigned long *did_some_progress)
                    /* Perform direct synchronous page reclaim */
       mm/page_alloc.c static int __perform_reclaim(gfp_t gfp_mask, unsigned int order, const struct alloc_context *ac)
           mm/vmscan.c     unsigned long try_to_free_pages(struct zonelist *zonelist, int order, gfp_t gfp_mask, nodemask_t *nodemask)
                               trace_mm_vmscan_direct_reclaim_begin(order, sc.may_writepage, sc.gfp_mask, sc.reclaim_idx);
                               trace_mm_vmscan_direct_reclaim_end(nr_reclaimed);
               mm/vmscan.c     /*
                                * This is the main entry point to direct page reclaim.
                                *
                                * If a full scan of the inactive list fails to free enough memory then we
                                * are "out of memory" and something needs to be killed.
                                [...snip]
                                */
                               static unsigned long do_try_to_free_pages(struct zonelist *zonelist, struct scan_control *sc)
```

try_to_free_pages()

```
unsigned long try_to_free_pages(struct zonelist *zonelist, int order,
                                gfp_t gfp_mask, nodemask_t *nodemask)
{
        unsigned long nr_reclaimed;
        struct scan_control sc = {
        [...snip]
        };

        /*
         * Do not enter reclaim if fatal signal was delivered while throttled.
         * 1 is returned so that the page allocator does not OOM kill at this
         * point.
         */
        if (throttle_direct_reclaim(sc.gfp_mask, zonelist, nodemask))
                return 1;

        trace_mm_vmscan_direct_reclaim_begin(order,
                                sc.may_writepage,
                                sc.gfp_mask,
                                sc.reclaim_idx);

        nr_reclaimed = do_try_to_free_pages(zonelist, &sc);

        trace_mm_vmscan_direct_reclaim_end(nr_reclaimed);

        return nr_reclaimed;
}
```

Systemtap mm_vmscan_direct_reclaim exercise

- mm_vmscan_direct_reclaim_begin
- mm_vmscan_direct_reclaim_end

Systemtap mm_vmscan_direct_reclaim example

```
global stall_entry

probe kernel.trace("mm_vmscan_direct_reclaim_begin") {
        stall_entry[tid()] = gettimeofday_us()
}
probe kernel.trace("mm_vmscan_direct_reclaim_end") {
        if (stall_entry[tid()]) {
                stalled = gettimeofday_us() - stall_entry[tid()]
                printf("thread: %-8d stall: %16d(us)\n", tid(), stalled)
                stall_entry[tid()] = 0;
        }
}
```

Systemtap mm_vmscan_direct_reclaim result

```
linux-g35h:/home/linux/tmp/kernel-tracing-tools-training/systemtap-test/mm # stap mm_vmscan_direct_reclaim_stall.stp

thread: 1130     stall:           34756(us)
thread: 1130     stall:           20313(us)
thread: 32161    stall:           73080(us)
thread: 31920    stall:          109688(us)
```

Trace event in compaction context

CONFIG_COMPACTION

```
#
# support for memory compaction
config COMPACTION
        bool "Allow for memory compaction"
        def_bool y
        select MIGRATION
        depends on MMU
        help
          Compaction is the only memory management component to form
          high order (larger physically contiguous) memory blocks
          reliably. The page allocator relies on compaction heavily and
          the lack of the feature can lead to unexpected OOM killer
          invocations for high order memory requests. You shouldn't
          disable this option unless there really is a strong reason for
          it and then we would be really interested to hear about that at
          linux-mm@kvack.org.
```

Memory compaction context

```
mm/page_alloc.c    static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order, struct alloc_context *ac)
                       /* Try memory compaction for high-order allocations before reclaim */
    mm/page_alloc.c    static struct page * __alloc_pages_direct_compact(gfp_t gfp_mask, unsigned int order, unsigned int alloc_flags,
                                                                         const struct alloc_context *ac, enum compact_priority prio,
                                                                         enum compact_result *compact_result)
                           /* Direct compact to satisfy a high-order allocation */
        mm/compaction.c    enum compact_result try_to_compact_pages(gfp_t gfp_mask, unsigned int order,
                                                unsigned int alloc_flags, const struct alloc_context *ac,
                                                enum compact_priority prio)
            mm/compaction.c    static enum compact_result compact_zone(struct zone *zone, struct compact_control *cc)
                                       trace_mm_compaction_begin(start_pfn, cc->migrate_pfn, cc->free_pfn, end_pfn, sync);
                                       trace_mm_compaction_migratepages(cc->nr_migratepages, err, &cc->migratepages);
                                       trace_mm_compaction_end(start_pfn, cc->migrate_pfn, cc->free_pfn, end_pfn, sync, ret);
                mm/compaction.c enum compact_result compaction_suitable(struct zone *zone, int order,
                                                                        unsigned int alloc_flags,
                                                                        int classzone_idx)
                                       trace_mm_compaction_suitable(zone, order, ret);
                               /* migrate_pages - migrate the pages specified in a list, to the free pages supplied as the target for the page migration */
                mm/migrate.c    int migrate_pages(struct list_head *from, new_page_t get_new_page,
                                                  free_page_t put_new_page, unsigned long private,
                                                  enum migrate_mode mode, int reason)
                                       trace_mm_migrate_pages(nr_succeeded, nr_failed, mode, reason);
```

compaction_suitable()

```
static enum compact_result compact_zone(struct zone *zone, struct compact_control *cc)
{
        enum compact_result ret;
        unsigned long start_pfn = zone->zone_start_pfn;
        unsigned long end_pfn = zone_end_pfn(zone);
        const bool sync = cc->mode != MIGRATE_ASYNC;

        cc->migratetype = gfpflags_to_migratetype(cc->gfp_mask);
        ret = compaction_suitable(zone, cc->order, cc->alloc_flags,
                                                        cc->classzone_idx);
        /* Compaction is likely to fail */
        if (ret == COMPACT_SUCCESS || ret == COMPACT_SKIPPED)
                return ret;

        /* huh, compaction_suitable is returning something unexpected */
        VM_BUG_ON(ret != COMPACT_CONTINUE);
```

mm_compaction_suitable

```
enum compact_result compaction_suitable(struct zone *zone, int order,
                                        unsigned int alloc_flags,
                                        int classzone_idx)
{
        enum compact_result ret;
        int fragindex;

        ret = __compaction_suitable(zone, order, alloc_flags, classzone_idx,
                                    zone_page_state(zone, NR_FREE_PAGES));
        [...snip]
        trace_mm_compaction_suitable(zone, order, ret);
        if (ret == COMPACT_NOT_SUITABLE_ZONE)
                ret = COMPACT_SKIPPED;

        return ret;
}
```

__compaction_suitable()

```
/*
 * compaction_suitable: Is this suitable to run compaction on this zone now?
 * Returns
 *   COMPACT_SKIPPED  - If there are too few free pages for compaction
 *   COMPACT_SUCCESS  - If the allocation would succeed without compaction
 *   COMPACT_CONTINUE - If compaction should run now
 */
static enum compact_result __compaction_suitable(struct zone *zone, int order,
                                        unsigned int alloc_flags,
                                        int classzone_idx,
                                        unsigned long wmark_target)
{
        unsigned long watermark;

        if (is_via_compact_memory(order))
                return COMPACT_CONTINUE;

        watermark = zone->watermark[alloc_flags & ALLOC_WMARK_MASK];
        /*
         * If watermarks for high-order allocation are already met, there
         * should be no need for compaction at all.
         */
        if (zone_watermark_ok(zone, order, watermark, classzone_idx,
                                                              alloc_flags))
                return COMPACT_SUCCESS;
```

mm_compaction_begin/end and mm_compaction_migratepages

```
static enum compact_result compact_zone(struct zone *zone, struct compact_control *cc)
{
        enum compact_result ret;
        unsigned long start_pfn = zone->zone_start_pfn;
        unsigned long end_pfn = zone_end_pfn(zone);
        const bool sync = cc->mode != MIGRATE_ASYNC;
        [...snip]
        trace_mm_compaction_begin(start_pfn, cc->migrate_pfn,
                                  cc->free_pfn, end_pfn, sync);
        migrate_prep_local();
        while ((ret = compact_finished(zone, cc)) == COMPACT_CONTINUE) {
                int err;
                switch (isolate_migratepages(zone, cc)) {
                [...snip]
                }
                err = migrate_pages(&cc->migratepages, compaction_alloc,
                                compaction_free, (unsigned long)cc, cc->mode,
                                MR_COMPACTION);
                trace_mm_compaction_migratepages(cc->nr_migratepages, err,
                                                 &cc->migratepages);
                /* All pages were either migrated or will be released */
                [...snip]
check_drain:
                [...snip]
        }
out:
        /*
         * Release free pages and update where the free scanner should restart,*/
        if (cc->nr_freepages > 0) {
                unsigned long free_pfn = release_freepages(&cc->freepages);
        [...snip]
        }
        trace_mm_compaction_end(start_pfn, cc->migrate_pfn,
                                cc->free_pfn, end_pfn, sync, ret);
        return ret;
}
```

mm_migrate_pages

```
/*
 * migrate_pages - migrate the pages specified in a list, to the free pages
 *                 supplied as the target for the page migration
[...snip]
 */
int migrate_pages(struct list_head *from, new_page_t get_new_page,
                free_page_t put_new_page, unsigned long private,
                enum migrate_mode mode, int reason)
{
        [...snip]
        for(pass = 0; pass < 10 && retry; pass++) {
                retry = 0;
                list_for_each_entry_safe(page, page2, from, lru) {
                [...snip]
                                rc = unmap_and_move(get_new_page, put_new_page,
                                                private, page, pass > 2, mode,
                                                reason);
                        switch(rc) {
                        [...snip]
                        case -EAGAIN:
                                retry++;
                                break;
                        case MIGRATEPAGE_SUCCESS:
                                nr_succeeded++;
                                break;
                        default:
                        [...snip]
                        }
                }
        }
        [...snip]
        trace_mm_migrate_pages(nr_succeeded, nr_failed, mode, reason);
        [...snip]
        return rc;
}
```

Systemtap mm_compaction exercise

- mm_compaction_suitable
- mm_compaction_begin/end
- mm_compaction_migratepages
  - mm_migrate_pages

Systemtap mm_compaction example

```
global compact_result
probe begin {
        compact_result[0]="COMPACT_NOT_SUITABLE_ZONE"
        compact_result[1]="COMPACT_SKIPPED"
        compact_result[2]="COMPACT_DEFERRED"
        compact_result[3]="COMPACT_NO_SUITABLE_PAGE"
        compact_result[4]="COMPACT_CONTINUE"
        compact_result[5]="COMPACT_COMPLETE"
        compact_result[6]="COMPACT_PARTIAL_SKIPPED"
        compact_result[7]="COMPACT_CONTENDED"
        compact_result[8]="COMPACT_SUCCESS"
}
probe kernel.trace("mm_compaction_suitable") {
        printf ("mm_compaction_suitable node: %d        order: %d       ret: %d (%s)\n",
                $zone->node, $order, $ret, compact_result[$ret])
}
/* TP_ARGS(zone_start, migrate_pfn, free_pfn, zone_end, sync) */
probe kernel.trace("mm_compaction_begin") {
        printf ("mm_compaction_begin zone_start: %x, migrate_pfn: %x\n",
                $zone_start, $migrate_pfn)
}
/* TP_ARGS(zone_start, migrate_pfn, free_pfn, zone_end, sync, status) */
probe kernel.trace("mm_compaction_end") {
        printf ("mm_compaction_begin zone_start: %x, migrate_pfn: %x, status: %d\n",
                $zone_start, $migrate_pfn, $status)
}
```

Systemtap mm_compaction example result

```
linux-g35h:/home/linux/tmp/kernel-tracing-tools-training/systemtap-mm # stap mm_compaction.stp
mm_compaction_suitable node: 0  order: 1        ret: 1 (COMPACT_SKIPPED)
mm_compaction_suitable node: 0  order: 1        ret: 1 (COMPACT_SKIPPED)
mm_compaction_suitable node: 0  order: 1        ret: 3 (COMPACT_NO_SUITABLE_PAGE)
mm_compaction_suitable node: 0  order: 1        ret: 3 (COMPACT_NO_SUITABLE_PAGE)
mm_compaction_suitable node: 0  order: 2        ret: 1 (COMPACT_SKIPPED)
mm_compaction_suitable node: 0  order: 2        ret: 3 (COMPACT_NO_SUITABLE_PAGE)
```

Other trace_mm_compaction_* events

```
mm/compaction.c:        trace_mm_compaction_defer_compaction(zone, order);
mm/compaction.c:        trace_mm_compaction_deferred(zone, order);
mm/compaction.c:        trace_mm_compaction_defer_reset(zone, order);
mm/compaction.c:        trace_mm_compaction_isolate_freepages(*start_pfn, blockpfn,
mm/compaction.c:        trace_mm_compaction_isolate_migratepages(start_pfn, low_pfn,
mm/compaction.c:        trace_mm_compaction_finished(zone, cc->order, ret);
mm/compaction.c:        trace_mm_compaction_suitable(zone, order, ret);
mm/compaction.c:        trace_mm_compaction_begin(start_pfn, cc->migrate_pfn,
mm/compaction.c:                trace_mm_compaction_migratepages(cc->nr_migratepages, err,
mm/compaction.c:        trace_mm_compaction_end(start_pfn, cc->migrate_pfn,
mm/compaction.c:        trace_mm_compaction_try_to_compact_pages(order, gfp_mask, prio);
mm/compaction.c:        trace_mm_compaction_kcompactd_wake(pgdat->node_id, cc.order,
mm/compaction.c:        trace_mm_compaction_wakeup_kcompactd(pgdat->node_id, order,
mm/compaction.c:                trace_mm_compaction_kcompactd_sleep(pgdat->node_id);
```

IO congestion

Gary Lin
SUSE Labs Taipei

Writeback Trace Events

- writeback_congestion_wait
- writeback_wait_iff_congested

congestion_wait()

```
long congestion_wait(int sync, long timeout)
{
    long ret;
    unsigned long start = jiffies;

    DEFINE_WAIT(wait); /* Put the current task into a wait entry */
    wait_queue_head_t wqh = &congestion_wqh[sync]; /* Choose the queue */

    prepare_to_wait(wqh, &wait, TASK_UNINTERRUPTIBLE);
    ret = io_schedule_timeout(timeout);
    finish_wait(wqh, &wait);

    trace_writeback_congestion_wait(jiffies_to_usecs(timeout),
                                    jiffies_to_usecs(jiffies - start));

    return ret;
}
```

Wait at most *timeout* jiffies

ext4_bio_write_page()

```
retry_encrypt:
    data_page = fscrypt_encrypt_page(inode, page, PAGE_SIZE, 0,
                                     page->index, gfp_flags);
    if (IS_ERR(data_page)) {
        ret = PTR_ERR(data_page);
        if (ret == -ENOMEM && wbc->sync_mode == WB_SYNC_ALL) {
            if (io->io_bio) {
                ext4_io_submit(io);
                congestion_wait(BLK_RW_ASYNC, HZ/50);
            }
            gfp_flags |= __GFP_NOFAIL;
            goto retry_encrypt;
        }
        data_page = NULL;
        goto out;
    }
```

do_writepages()

```
int do_writepages(struct address_space *mapping,
                  struct writeback_control *wbc)
{
    int ret;

    if (wbc->nr_to_write <= 0)
        return 0;
    while (1) {
        if (mapping->a_ops->writepages)
            ret = mapping->a_ops->writepages(mapping, wbc);
        else
            ret = generic_writepages(mapping, wbc);
        if ((ret != -ENOMEM) || (wbc->sync_mode != WB_SYNC_ALL))
            break;
        cond_resched();
        congestion_wait(BLK_RW_ASYNC, HZ/50);
    }
    return ret;
}
```

Functions invoking do_writepages()

- writeback_signle_inode() -> do_writepages()
  - iput() fs/inode.c
    - iput_final() fs/inode.c
      - write_inode_now() fs/fs-writeback.c
  - sync_inode_metadata() fs/fs-writeback.c
    - sync_inode() fs/fs-writeback.c
- __filemap_fdata_write_range() -> do_writepages()
  - sync_file_range() fs/sync.c
  - filemap_fdatawrite_range() mm/filemap.c
  - file_write_and_wait_range() mm/filemap.c

shrink_inactive_list()

```
static noinline_for_stack unsigned long
shrink_inactive_list(unsigned long nr_to_scan, struct lruvec *lruvec,
                     struct scan_control *sc, enum lru_list lru)
{
    ...

    while (unlikely(too_many_isolated(pgdat, file, sc))) {
        congestion_wait(BLK_RW_ASYNC, HZ/10);

        /* We are about to die and free our memory. Return now. */
        if (fatal_signal_pending(current))
            return SWAP_CLUSTER_MAX;
    }

    ...
```

wait_iff_congestion()

```
long wait_iff_congestion(struct pglist_data *pgdat, int sync,
                         long timeout)
{
    ...

    if (atomic_read(&nr_wb_congested[sync]) == 0 ||
        !test_bit(PGDAT_CONGESTED, &pgdat->flags)) {
            cond_resched();

            /* In case we scheduled, work out time remaining */
            ret = timeout - (jiffies - start)
            if (ret < 0)
                ret = 0;
            goto out;
    }

    /* Sleep until uncongested or a write happens */
    prepare_to_wait(wqh, &wait, TASK_UNINTERRUPTIBLE);
    ret = io_schedule_timeout(timeout);
    finish_wait(wqh, &wait);
out:
    trace_writeback_wait_iff_congested(jiffies_to_usecs(timeout),
                                       jiffies_to_usecs(jiffies - start));
    ...
}
```

Skip if there is no congestion

shrink_inactive_list()

```
static noinline_for_stack unsigned long
shrink_inactive_list(unsigned long nr_to_scan, struct lruvec *lruvec,
                     struct scan_control *sc, enum lru_list lru)
{
    ...

    /*
     * Stall direct reclaim for IO completions if underlying BDIs or zone
     * is congested. Allow kswapd to continue until it starts encountering
     * unqueued dirty pages or cycling through the LRU too quickly.
     */
    if (!sc->hibernation_mode && !current_is_kswapd() &&
        current_may_throttle())
        wait_iff_congested(pgdat, BLK_RW_ASYNC, HZ/10);

    ...
```

Two cases

Reduce writeback from page reclaim context

- Fixing writeback from direct reclaim

  https://lwn.net/Articles/396561/

- Reduce writeback from page reclaim context V4, Mel Gorman

Avoid swapout/swapin

- Issue
  - Huge latency is caused when an application accesses the swapped out page
  - We can’t avoid swapout even if swappiness == 0
- Solution
  - Change the behavior with swappiness == 0
    - With this value the kernel doesn’t swapout any anon pages while it has enough filebacked pages
    - If we set cgroup swappiness to 0, we can avoid swap out completely for the processes in the cgroup [1]
  - mm: avoid swapping out with swappiness==0
    - Fe35004fbf9eaf67482b074a2e032abb9c89b1dd
    - Satoru Moriya <satoru.moriya@hds.com>

Others mm trace events

Trace when a process stalls due to dirty page limiting

- When there is a lot of dirty data in the system, a writer may stall waiting on data to be cleaned. It's not always obvious that it occurred but it can be detected by tracing writeback/balance_dirty_pages [Mel Gorman]

Other MM events

- Getting value from the other ones requires some knowledge of the MM implementation. However, broadly speaking the following is possible
  - all kswapd wakeup/sleep activity
  - all internal reclaim decisions both direct and kswapd
  - slab allocations/frees
  - page migration activity
  - khugepaged activity
  - kcompactd activity
  - tlb flush activity

Mel Gorman [8]

Q&A

Reference

- [1] Reducing Memory Access Latency, Satoru Moriya, Linux Technology Center, Yokohama Research Lab., Hitachi, Ltd
- [2] Fixing writeback from direct reclaim, Jonathan Corbet, LWN.net
- [3] Reduce writeback from page reclaim context V4, Mel Gorman, Mon, 19 Jul 2010, kernel patch
- [4] Understanding The Linux Virtual Memory Manager, Mel Gorman, July 9, 2007
- [5] Memory – Part 1: Memory Types, Florent Bruneau, July 3, 2013

Reference (cont.)

- [6] mmap, Wikipedia
- [7] Cache replacement policies#LRU, Wikipedia
- [8] https://wiki.microfocus.net/index.php/SUSE_Labs_Taipei/Training/Kernel_Tracing_Training#Memory_Management

Feedback to
jlee@suse.com

Thank you.


SUSE®
We adapt. You succeed.

Corporate Headquarters
Maxfeldstrasse 5
90409 Nuremberg
Germany

+49 911 740 53 0 (Worldwide)
www.suse.com

Join us on:
www.opensuse.org