torvalds / linux

<> Code Pull requests 321 Actions Projects Security Insights

v4.4-rc1 ▾ ···

linux / mm / madvise.c

Mike Kravetz mm: madvise allow remove operation for hugetlbfs … History

13 contributors +1

Raw Blame Copy File 30.46 KB

551 lines (491 sloc) | 13.9 KB

```c
/*
 *	linux/mm/madvise.c
 *
 * Copyright (C) 1999  Linus Torvalds
 * Copyright (C) 2002  Christoph Hellwig
 */

#include <linux/mman.h>
#include <linux/pagemap.h>
#include <linux/syscalls.h>
#include <linux/mempolicy.h>
#include <linux/page-isolation.h>
#include <linux/hugetlb.h>
#include <linux/falloc.h>
#include <linux/sched.h>
#include <linux/ksm.h>
#include <linux/fs.h>
#include <linux/file.h>
#include <linux/blkdev.h>
#include <linux/backing-dev.h>
#include <linux/swap.h>
#include <linux/swapops.h>

/*
 * Any behaviour which results in changes to the vma->vm_flags needs to
 * take mmap_sem for writing. Others, which simply traverse vmas, need
 * to only take it for reading.
 */
static int madvise_need_mmap_write(int behavior)
{
        switch (behavior) {
        case MADV_REMOVE:
        case MADV_WILLNEED:
        case MADV_DONTNEED:
                return 0;
        default:
                /* be safe, default to 1. list exceptions explicitly */
                return 1;
        }
}

/*
 * We can potentially split a vm area into separate
 * areas, each area with its own behavior.
 */
static long madvise_behavior(struct vm_area_struct *vma,
                     struct vm_area_struct **prev,
                     unsigned long start, unsigned long end, int behavior)
{
        struct mm_struct *mm = vma->vm_mm;
        int error = 0;
        pgoff_t pgoff;
        unsigned long new_flags = vma->vm_flags;

        switch (behavior) {
        case MADV_NORMAL:
                new_flags = new_flags & ~VM_RAND_READ & ~VM_SEQ_READ;
                break;
        case MADV_SEQUENTIAL:
                new_flags = (new_flags & ~VM_RAND_READ) | VM_SEQ_READ;
                break;
        case MADV_RANDOM:
                new_flags = (new_flags & ~VM_SEQ_READ) | VM_RAND_READ;
                break;
        case MADV_DONTFORK:
                new_flags |= VM_DONTCOPY;
                break;
        case MADV_DOFORK:
                if (vma->vm_flags & VM_IO) {
                        error = -EINVAL;
                        goto out;
                }
                new_flags &= ~VM_DONTCOPY;
                break;
        case MADV_DONTDUMP:
                new_flags |= VM_DONTDUMP;
                break;
        case MADV_DODUMP:
                if (new_flags & VM_SPECIAL) {
                        error = -EINVAL;
                        goto out;
                }
                new_flags &= ~VM_DONTDUMP;
                break;
        case MADV_MERGEABLE:
        case MADV_UNMERGEABLE:
                error = ksm_madvise(vma, start, end, behavior, &new_flags);
                if (error)
                        goto out;
                break;
        case MADV_HUGEPAGE:
        case MADV_NOHUGEPAGE:
                error = hugepage_madvise(vma, &new_flags, behavior);
                if (error)
                        goto out;
                break;
        }

        if (new_flags == vma->vm_flags) {
                *prev = vma;
                goto out;
        }

        pgoff = vma->vm_pgoff + ((start - vma->vm_start) >> PAGE_SHIFT);
        *prev = vma_merge(mm, *prev, start, end, new_flags, vma->anon_vma,
                          vma->vm_file, pgoff, vma_policy(vma),
                          vma->vm_userfaultfd_ctx);
        if (*prev) {
                vma = *prev;
                goto success;
        }

        *prev = vma;

        if (start != vma->vm_start) {
                error = split_vma(mm, vma, start, 1);
                if (error)
                        goto out;
        }

        if (end != vma->vm_end) {
                error = split_vma(mm, vma, end, 0);
                if (error)
```

```
                        goto out;
        }

success:
        /*
         * vm_flags is protected by the mmap_sem held in write mode.
         */
        vma->vm_flags = new_flags;

out:
        if (error == -ENOMEM)
                error = -EAGAIN;
        return error;
}

#ifdef CONFIG_SWAP
static int swapin_walk_pmd_entry(pmd_t *pmd, unsigned long start,
        unsigned long end, struct mm_walk *walk)
{
        pte_t *orig_pte;
        struct vm_area_struct *vma = walk->private;
        unsigned long index;

        if (pmd_none_or_trans_huge_or_clear_bad(pmd))
                return 0;

        for (index = start; index != end; index += PAGE_SIZE) {
                pte_t pte;
                swp_entry_t entry;
                struct page *page;
                spinlock_t *ptl;

                orig_pte = pte_offset_map_lock(vma->vm_mm, pmd, start, &ptl);
                pte = *(orig_pte + ((index - start) / PAGE_SIZE));
                pte_unmap_unlock(orig_pte, ptl);

                if (pte_present(pte) || pte_none(pte))
                        continue;
                entry = pte_to_swp_entry(pte);
                if (unlikely(non_swap_entry(entry)))
                        continue;

                page = read_swap_cache_async(entry, GFP_HIGHUSER_MOVABLE,
                                                                vma, index);
                if (page)
                        page_cache_release(page);
        }

```

```c
        return 0;
}

static void force_swapin_readahead(struct vm_area_struct *vma,
                unsigned long start, unsigned long end)
{
        struct mm_walk walk = {
                .mm = vma->vm_mm,
                .pmd_entry = swapin_walk_pmd_entry,
                .private = vma,
        };

        walk_page_range(start, end, &walk);

        lru_add_drain();        /* Push any new pages onto the LRU now */
}

static void force_shm_swapin_readahead(struct vm_area_struct *vma,
                unsigned long start, unsigned long end,
                struct address_space *mapping)
{
        pgoff_t index;
        struct page *page;
        swp_entry_t swap;

        for (; start < end; start += PAGE_SIZE) {
                index = ((start - vma->vm_start) >> PAGE_SHIFT) + vma->vm_pgoff;

                page = find_get_entry(mapping, index);
                if (!radix_tree_exceptional_entry(page)) {
                        if (page)
                                page_cache_release(page);
                        continue;
                }
                swap = radix_to_swp_entry(page);
                page = read_swap_cache_async(swap, GFP_HIGHUSER_MOVABLE,
                                                                NULL, 0);
                if (page)
                        page_cache_release(page);
        }

        lru_add_drain();        /* Push any new pages onto the LRU now */
}
#endif          /* CONFIG_SWAP */

/*
 * Schedule all required I/O operations.  Do not wait for completion.
 */
```

```c
static long madvise_willneed(struct vm_area_struct *vma,
                             struct vm_area_struct **prev,
                             unsigned long start, unsigned long end)
{
        struct file *file = vma->vm_file;

#ifdef CONFIG_SWAP
        if (!file) {
                *prev = vma;
                force_swapin_readahead(vma, start, end);
                return 0;
        }

        if (shmem_mapping(file->f_mapping)) {
                *prev = vma;
                force_shm_swapin_readahead(vma, start, end,
                                        file->f_mapping);
                return 0;
        }
#else
        if (!file)
                return -EBADF;
#endif

        if (IS_DAX(file_inode(file))) {
                /* no bad return value, but ignore advice */
                return 0;
        }

        *prev = vma;
        start = ((start - vma->vm_start) >> PAGE_SHIFT) + vma->vm_pgoff;
        if (end > vma->vm_end)
                end = vma->vm_end;
        end = ((end - vma->vm_start) >> PAGE_SHIFT) + vma->vm_pgoff;

        force_page_cache_readahead(file->f_mapping, file, start, end - start);
        return 0;
}

/*
 * Application no longer needs these pages.  If the pages are dirty,
 * it's OK to just throw them away.  The app will be more careful about
 * data it wants to keep.  Be sure to free swap resources too.  The
 * zap_page_range call sets things up for shrink_active_list to actually free
 * these pages later if no one else has touched them in the meantime,
 * although we could add these pages to a global reuse list for
 * shrink_active_list to pick up before reclaiming other pages.
 *
 * NB: This interface discards data rather than pushes it out to swap,
 * as some implementations do.  This has performance implications for
 * applications like large transactional databases which want to discard
 * pages in anonymous maps after committing to backing store the data
 * that was kept in them.  There is no reason to write this data out to
 * the swap area if the application is discarding it.
 *
 * An interface that causes the system to free clean pages and flush
 * dirty pages is already available as msync(MS_INVALIDATE).
 */
static long madvise_dontneed(struct vm_area_struct *vma,
                             struct vm_area_struct **prev,
                             unsigned long start, unsigned long end)
{
        *prev = vma;
        if (vma->vm_flags & (VM_LOCKED|VM_HUGETLB|VM_PFNMAP))
                return -EINVAL;

        zap_page_range(vma, start, end - start, NULL);
        return 0;
}

/*
 * Application wants to free up the pages and associated backing store.
 * This is effectively punching a hole into the middle of a file.
 */
static long madvise_remove(struct vm_area_struct *vma,
                                struct vm_area_struct **prev,
                                unsigned long start, unsigned long end)
{
        loff_t offset;
        int error;
        struct file *f;

        *prev = NULL;   /* tell sys_madvise we drop mmap_sem */

        if (vma->vm_flags & VM_LOCKED)
                return -EINVAL;

        f = vma->vm_file;

        if (!f || !f->f_mapping || !f->f_mapping->host) {
                        return -EINVAL;
        }

        if ((vma->vm_flags & (VM_SHARED|VM_WRITE)) != (VM_SHARED|VM_WRITE))
                return -EACCES;

        offset = (loff_t)(start - vma->vm_start)
                        + ((loff_t)vma->vm_pgoff << PAGE_SHIFT);

        /*
         * Filesystem's fallocate may need to take i_mutex.  We need to
         * explicitly grab a reference because the vma (and hence the
         * vma's reference to the file) can go away as soon as we drop
         * mmap_sem.
         */
        get_file(f);
        up_read(&current->mm->mmap_sem);
        error = vfs_fallocate(f,
                                FALLOC_FL_PUNCH_HOLE | FALLOC_FL_KEEP_SIZE,
                                offset, end - start);
        fput(f);
        down_read(&current->mm->mmap_sem);
        return error;
}

#ifdef CONFIG_MEMORY_FAILURE
/*
 * Error injection support for memory error handling.
 */
static int madvise_hwpoison(int bhv, unsigned long start, unsigned long end)
{
        struct page *p;
        if (!capable(CAP_SYS_ADMIN))
                return -EPERM;
        for (; start < end; start += PAGE_SIZE <<
                                compound_order(compound_head(p))) {
                int ret;

                ret = get_user_pages_fast(start, 1, 0, &p);
                if (ret != 1)
                        return ret;

                if (PageHWPoison(p)) {
                        put_page(p);
                        continue;
                }
                if (bhv == MADV_SOFT_OFFLINE) {
                        pr_info("Soft offlining page %#lx at %#lx\n",
                                page_to_pfn(p), start);
                        ret = soft_offline_page(p, MF_COUNT_INCREASED);
                        if (ret)
                                return ret;
                        continue;
                }
                pr_info("Injecting memory failure for page %#lx at %#lx\n",
                       page_to_pfn(p), start);
                /* Ignore return value for now */
                memory_failure(page_to_pfn(p), 0, MF_COUNT_INCREASED);
        }
        return 0;
}
#endif

static long
madvise_vma(struct vm_area_struct *vma, struct vm_area_struct **prev,
                unsigned long start, unsigned long end, int behavior)
{
        switch (behavior) {
        case MADV_REMOVE:
                return madvise_remove(vma, prev, start, end);
        case MADV_WILLNEED:
                return madvise_willneed(vma, prev, start, end);
        case MADV_DONTNEED:
                return madvise_dontneed(vma, prev, start, end);
        default:
                return madvise_behavior(vma, prev, start, end, behavior);
        }
}

static bool
madvise_behavior_valid(int behavior)
{
        switch (behavior) {
        case MADV_DOFORK:
        case MADV_DONTFORK:
        case MADV_NORMAL:
        case MADV_SEQUENTIAL:
        case MADV_RANDOM:
        case MADV_REMOVE:
        case MADV_WILLNEED:
        case MADV_DONTNEED:
#ifdef CONFIG_KSM
        case MADV_MERGEABLE:
        case MADV_UNMERGEABLE:
#endif
#ifdef CONFIG_TRANSPARENT_HUGEPAGE
        case MADV_HUGEPAGE:
        case MADV_NOHUGEPAGE:
#endif
        case MADV_DONTDUMP:
        case MADV_DODUMP:
                return true;
```

```
	default:
		return false;
	}
}

/*
 * The madvise(2) system call.
 *
 * Applications can use madvise() to advise the kernel how it should
 * handle paging I/O in this VM area.  The idea is to help the kernel
 * use appropriate read-ahead and caching techniques.  The information
 * provided is advisory only, and can be safely disregarded by the
 * kernel without affecting the correct operation of the application.
 *
 * behavior values:
 *  MADV_NORMAL - the default behavior is to read clusters.  This
 *		results in some read-ahead and read-behind.
 *  MADV_RANDOM - the system should read the minimum amount of data
 *		on any access, since it is unlikely that the appli-
 *		cation will need more than what it asks for.
 *  MADV_SEQUENTIAL - pages in the given range will probably be accessed
 *		once, so they can be aggressively read ahead, and
 *		can be freed soon after they are accessed.
 *  MADV_WILLNEED - the application is notifying the system to read
 *		some pages ahead.
 *  MADV_DONTNEED - the application is finished with the given range,
 *		so the kernel can free resources associated with it.
 *  MADV_REMOVE - the application wants to free up the given range of
 *		pages and associated backing store.
 *  MADV_DONTFORK - omit this area from child's address space when forking:
 *		typically, to avoid COWing pages pinned by get_user_pages().
 *  MADV_DOFORK - cancel MADV_DONTFORK: no longer omit this area when forking.
 *  MADV_MERGEABLE - the application recommends that KSM try to merge pages in
 *		this area with pages of identical content from other such areas.
 *  MADV_UNMERGEABLE- cancel MADV_MERGEABLE: no longer merge pages with others.
 *
 * return values:
 *  zero    - success
 *  -EINVAL - start + len < 0, start is not page-aligned,
 *		"behavior" is not a valid value, or application
 *		is attempting to release locked or shared pages.
 *  -ENOMEM - addresses in the specified range are not currently
 *		mapped, or are outside the AS of the process.
 *  -EIO    - an I/O error occurred while paging in data.
 *  -EBADF  - map exists, but area maps something that isn't a file.
 *  -EAGAIN - a kernel resource was temporarily unavailable.
 */
```

```c
SYSCALL_DEFINE3(madvise, unsigned long, start, size_t, len_in, int, behavior)
{
        unsigned long end, tmp;
        struct vm_area_struct *vma, *prev;
        int unmapped_error = 0;
        int error = -EINVAL;
        int write;
        size_t len;
        struct blk_plug plug;

#ifdef CONFIG_MEMORY_FAILURE
        if (behavior == MADV_HWPOISON || behavior == MADV_SOFT_OFFLINE)
                return madvise_hwpoison(behavior, start, start+len_in);
#endif
        if (!madvise_behavior_valid(behavior))
                return error;

        if (start & ~PAGE_MASK)
                return error;
        len = (len_in + ~PAGE_MASK) & PAGE_MASK;

        /* Check to see whether len was rounded up from small -ve to zero */
        if (len_in && !len)
                return error;

        end = start + len;
        if (end < start)
                return error;

        error = 0;
        if (end == start)
                return error;

        write = madvise_need_mmap_write(behavior);
        if (write)
                down_write(&current->mm->mmap_sem);
        else
                down_read(&current->mm->mmap_sem);

        /*
         * If the interval [start,end) covers some unmapped address
         * ranges, just ignore them, but return -ENOMEM at the end.
         * - different from the way of handling in mlock etc.
         */
        vma = find_vma_prev(current->mm, start, &prev);
        if (vma && start > vma->vm_start)
                prev = vma;

        blk_start_plug(&plug);
        for (;;) {
                /* Still start < end. */
                error = -ENOMEM;
                if (!vma)
                        goto out;

                /* Here start < (end|vma->vm_end). */
                if (start < vma->vm_start) {
                        unmapped_error = -ENOMEM;
                        start = vma->vm_start;
                        if (start >= end)
                                goto out;
                }

                /* Here vma->vm_start <= start < (end|vma->vm_end) */
                tmp = vma->vm_end;
                if (end < tmp)
                        tmp = end;

                /* Here vma->vm_start <= start < tmp <= (end|vma->vm_end). */
                error = madvise_vma(vma, &prev, start, tmp, behavior);
                if (error)
                        goto out;
                start = tmp;
                if (prev && start < prev->vm_end)
                        start = prev->vm_end;
                error = unmapped_error;
                if (start >= end)
                        goto out;
                if (prev)
                        vma = prev->vm_next;
                else    /* madvise_remove dropped mmap_sem */
                        vma = find_vma(current->mm, start);
        }
out:
        blk_finish_plug(&plug);
        if (write)
                up_write(&current->mm->mmap_sem);
        else
                up_read(&current->mm->mmap_sem);

        return error;
}
```