```
/*
 * linux/mm/memory.c
 *
 * Copyright (C) 1991, 1992, 1993, 1994 Linus Torvalds
 */

/*
 * demand-loading started 01.12.91 - seems it is high on the list of
 * things wanted, and it should be easy to implement. - Linus
 */

/*
 * Ok, demand-loading was easy, shared pages a little bit tricker. Shared
 * pages started 02.12.91, seems to work. - Linus.
 *
 * Tested sharing by executing about 30 /bin/sh: under the old kernel it
 * would have taken more than the 6M I have free, but it worked well as
 * far as I could see.
 *
 * Also corrected some "invalidate()"s - I wasn't doing enough of them.
 */

/*
 * Real VM (paging to/from disk) started 18.12.91. Much more work and
 * thought has to go into this. Oh, well..
 * 19.12.91 - works, somewhat. Sometimes I get faults, don't know why.
 *              Found it. Everything seems to work now.
 * 20.12.91 -   Ok, making the swap-device changeable like the root.
 */

/*
 * 05.04.94 - Multi-page memory management added for v1.1.
 *              Idea by Alex Bligh (alex@cconcepts.co.uk)
 *
 * 16.07.99 - Support of BIGMEM added by Gerhard Wichert, Siemens AG
 *              (Gerhard.Wichert@pdb.siemens.de)
 *
 * Aug/Sep 2004 Changed to four level page tables (Andi Kleen)
 */

#include <linux/kernel_stat.h>
#include <linux/mm.h>
#include <linux/hugetlb.h>
#include <linux/mman.h>
#include <linux/swap.h>
#include <linux/highmem.h>
#include <linux/pagemap.h>
#include <linux/rmap.h>
#include <linux/acct.h>
#include <linux/module.h>
#include <linux/init.h>

#include <asm/pgalloc.h>
#include <asm/uaccess.h>
#include <asm/tlb.h>
#include <asm/tlbflush.h>
#include <asm/pgtable.h>

#include <linux/swapops.h>
#include <linux/elf.h>

#ifndef CONFIG_DISCONTIGMEM
/* use the per-pgdat data instead for discontigmem - mbligh */
```

```
unsigned long max_mapnr;
struct page *mem_map;

EXPORT_SYMBOL(max_mapnr);
EXPORT_SYMBOL(mem_map);
#endif

unsigned long num_physpages;
/*
 * A number of key systems in x86 including ioremap() rely on the assumption
 * that high_memory defines the upper bound on direct map memory, then end
 * of ZONE_NORMAL. Under CONFIG_DISCONTIG this means that max_low_pfn and
 * highstart_pfn must be the same; there must be no gap between ZONE_NORMAL
 * and ZONE_HIGHMEM.
 */
void * high_memory;
unsigned long vmalloc_earlyreserve;

EXPORT_SYMBOL(num_physpages);
EXPORT_SYMBOL(high_memory);
EXPORT_SYMBOL(vmalloc_earlyreserve);

/*
 * Note: this doesn't free the actual pages themselves. That
 * has been handled earlier when unmapping all the memory regions.
 */
static inline void clear_pmd_range(struct mmu_gather *tlb, pmd_t *pmd, unsigned long start, unsigned long end)
{
	struct page *page;

	if (pmd_none(*pmd))
		return;
	if (unlikely(pmd_bad(*pmd))) {
		pmd_ERROR(*pmd);
		pmd_clear(pmd);
		return;
	}
	if (!((start | end) & ~PMD_MASK)) {
		/* Only clear full, aligned ranges */
		page = pmd_page(*pmd);
		pmd_clear(pmd);
		dec_page_state(nr_page_table_pages);
		tlb->mm->nr_ptes--;
		pte_free_tlb(tlb, page);
	}
}

static inline void clear_pud_range(struct mmu_gather *tlb, pud_t *pud, unsigned long start, unsigned long end)
{
	unsigned long addr = start, next;
	pmd_t *pmd, *__pmd;

	if (pud_none(*pud))
		return;
	if (unlikely(pud_bad(*pud))) {
		pud_ERROR(*pud);
		pud_clear(pud);
		return;
	}

	pmd = __pmd = pmd_offset(pud, start);
	do {
		next = (addr + PMD_SIZE) & PMD_MASK;
```

```
		if (next > end || next <= addr)
			next = end;

		clear_pmd_range(tlb, pmd, addr, next);
		pmd++;
		addr = next;
	} while (addr && (addr < end));

	if (!((start | end) & ~PUD_MASK)) {
		/* Only clear full, aligned ranges */
		pud_clear(pud);
		pmd_free_tlb(tlb, __pmd);
	}
}


static inline void clear_pgd_range(struct mmu_gather *tlb, pgd_t *pgd, unsigned long start, unsigned long end)
{
	unsigned long addr = start, next;
	pud_t *pud, *__pud;

	if (pgd_none(*pgd))
		return;
	if (unlikely(pgd_bad(*pgd))) {
		pgd_ERROR(*pgd);
		pgd_clear(pgd);
		return;
	}

	pud = __pud = pud_offset(pgd, start);
	do {
		next = (addr + PUD_SIZE) & PUD_MASK;
		if (next > end || next <= addr)
			next = end;

		clear_pud_range(tlb, pud, addr, next);
		pud++;
		addr = next;
	} while (addr && (addr < end));

	if (!((start | end) & ~PGDIR_MASK)) {
		/* Only clear full, aligned ranges */
		pgd_clear(pgd);
		pud_free_tlb(tlb, __pud);
	}
}

/*
 * This function clears user-level page tables of a process.
 *
 * Must be called with pagetable lock held.
 */
void clear_page_range(struct mmu_gather *tlb, unsigned long start, unsigned long end)
{
	unsigned long addr = start, next;
	pgd_t * pgd = pgd_offset(tlb->mm, start);
	unsigned long i;

	for (i = pgd_index(start); i <= pgd_index(end-1); i++) {
		next = (addr + PGDIR_SIZE) & PGDIR_MASK;
		if (next > end || next <= addr)
			next = end;
```

```
			clear_pgd_range(tlb, pgd, addr, next);
			pgd++;
			addr = next;
		}
}

pte_t fastcall * pte_alloc_map(struct mm_struct *mm, pmd_t *pmd, unsigned long address)
{
		if (!pmd_present(*pmd)) {
			struct page *new;

			spin_unlock(&mm->page_table_lock);
			new = pte_alloc_one(mm, address);
			spin_lock(&mm->page_table_lock);
			if (!new)
				return NULL;
			/*
			 * Because we dropped the lock, we should re-check the
			 * entry, as somebody else could have populated it..
			 */
			if (pmd_present(*pmd)) {
				pte_free(new);
				goto out;
			}
			mm->nr_ptes++;
			inc_page_state(nr_page_table_pages);
			pmd_populate(mm, pmd, new);
		}
out:
		return pte_offset_map(pmd, address);
}

pte_t fastcall * pte_alloc_kernel(struct mm_struct *mm, pmd_t *pmd, unsigned long address)
{
		if (!pmd_present(*pmd)) {
			pte_t *new;

			spin_unlock(&mm->page_table_lock);
			new = pte_alloc_one_kernel(mm, address);
			spin_lock(&mm->page_table_lock);
			if (!new)
				return NULL;

			/*
			 * Because we dropped the lock, we should re-check the
			 * entry, as somebody else could have populated it..
			 */
			if (pmd_present(*pmd)) {
				pte_free_kernel(new);
				goto out;
			}
			pmd_populate_kernel(mm, pmd, new);
		}
out:
		return pte_offset_kernel(pmd, address);
}

/*
 * copy one vm_area from one task to the other. Assumes the page tables
 * already present in the new task to be cleared in the whole range
 * covered by this vma.
 *
 * dst->page_table_lock is held on entry and exit,
```

```
 * but may be dropped within p[mg]d_alloc() and pte_alloc_map().
 */

static inline void
copy_swap_pte(struct mm_struct *dst_mm, struct mm_struct *src_mm, pte_t pte)
{
	if (pte_file(pte))
		return;
	swap_duplicate(pte_to_swp_entry(pte));
	if (list_empty(&dst_mm->mmlist)) {
		spin_lock(&mmlist_lock);
		list_add(&dst_mm->mmlist, &src_mm->mmlist);
		spin_unlock(&mmlist_lock);
	}
}

static inline void
copy_one_pte(struct mm_struct *dst_mm, struct mm_struct *src_mm,
		pte_t *dst_pte, pte_t *src_pte, unsigned long vm_flags,
		unsigned long addr)
{
	pte_t pte = *src_pte;
	struct page *page;
	unsigned long pfn;

	/* pte contains position in swap, so copy. */
	if (!pte_present(pte)) {
		copy_swap_pte(dst_mm, src_mm, pte);
		set_pte(dst_pte, pte);
		return;
	}
	pfn = pte_pfn(pte);
	/* the pte points outside of valid memory, the
	 * mapping is assumed to be good, meaningful
	 * and not mapped via rmap - duplicate the
	 * mapping as is.
	 */
	page = NULL;
	if (pfn_valid(pfn))
		page = pfn_to_page(pfn);

	if (!page || PageReserved(page)) {
		set_pte(dst_pte, pte);
		return;
	}

	/*
	 * If it's a COW mapping, write protect it both
	 * in the parent and the child
	 */
	if ((vm_flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE) {
		ptep_set_wrprotect(src_pte);
		pte = *src_pte;
	}

	/*
	 * If it's a shared mapping, mark it clean in
	 * the child
	 */
	if (vm_flags & VM_SHARED)
		pte = pte_mkclean(pte);
	pte = pte_mkold(pte);
	get_page(page);
```

```c
	dst_mm->rss++;
	if (PageAnon(page))
		dst_mm->anon_rss++;
	set_pte(dst_pte, pte);
	page_dup_rmap(page);
}

static int copy_pte_range(struct mm_struct *dst_mm, struct mm_struct *src_mm,
		pmd_t *dst_pmd, pmd_t *src_pmd, struct vm_area_struct *vma,
		unsigned long addr, unsigned long end)
{
	pte_t *src_pte, *dst_pte;
	pte_t *s, *d;
	unsigned long vm_flags = vma->vm_flags;

	d = dst_pte = pte_alloc_map(dst_mm, dst_pmd, addr);
	if (!dst_pte)
		return -ENOMEM;

	spin_lock(&src_mm->page_table_lock);
	s = src_pte = pte_offset_map_nested(src_pmd, addr);
	for (; addr < end; addr += PAGE_SIZE, s++, d++) {
		if (pte_none(*s))
			continue;
		copy_one_pte(dst_mm, src_mm, d, s, vm_flags, addr);
	}
	pte_unmap_nested(src_pte);
	pte_unmap(dst_pte);
	spin_unlock(&src_mm->page_table_lock);
	cond_resched_lock(&dst_mm->page_table_lock);
	return 0;
}

static int copy_pmd_range(struct mm_struct *dst_mm, struct mm_struct *src_mm,
		pud_t *dst_pud, pud_t *src_pud, struct vm_area_struct *vma,
		unsigned long addr, unsigned long end)
{
	pmd_t *src_pmd, *dst_pmd;
	int err = 0;
	unsigned long next;

	src_pmd = pmd_offset(src_pud, addr);
	dst_pmd = pmd_alloc(dst_mm, dst_pud, addr);
	if (!dst_pmd)
		return -ENOMEM;

	for (; addr < end; addr = next, src_pmd++, dst_pmd++) {
		next = (addr + PMD_SIZE) & PMD_MASK;
		if (next > end || next <= addr)
			next = end;
		if (pmd_none(*src_pmd))
			continue;
		if (pmd_bad(*src_pmd)) {
			pmd_ERROR(*src_pmd);
			pmd_clear(src_pmd);
			continue;
		}
		err = copy_pte_range(dst_mm, src_mm, dst_pmd, src_pmd,
							vma, addr, next);
		if (err)
			break;
	}
	return err;
```

```
}

static int copy_pud_range(struct mm_struct *dst_mm, struct mm_struct *src_mm,
                pgd_t *dst_pgd, pgd_t *src_pgd, struct vm_area_struct *vma,
                unsigned long addr, unsigned long end)
{
        pud_t *src_pud, *dst_pud;
        int err = 0;
        unsigned long next;

        src_pud = pud_offset(src_pgd, addr);
        dst_pud = pud_alloc(dst_mm, dst_pgd, addr);
        if (!dst_pud)
                return -ENOMEM;

        for (; addr < end; addr = next, src_pud++, dst_pud++) {
                next = (addr + PUD_SIZE) & PUD_MASK;
                if (next > end || next <= addr)
                        next = end;
                if (pud_none(*src_pud))
                        continue;
                if (pud_bad(*src_pud)) {
                        pud_ERROR(*src_pud);
                        pud_clear(src_pud);
                        continue;
                }
                err = copy_pmd_range(dst_mm, src_mm, dst_pud, src_pud,
                                                        vma, addr, next);
                if (err)
                        break;
        }
        return err;
}

int copy_page_range(struct mm_struct *dst, struct mm_struct *src,
                struct vm_area_struct *vma)
{
        pgd_t *src_pgd, *dst_pgd;
        unsigned long addr, start, end, next;
        int err = 0;

        if (is_vm_hugetlb_page(vma))
                return copy_hugetlb_page_range(dst, src, vma);

        start = vma->vm_start;
        src_pgd = pgd_offset(src, start);
        dst_pgd = pgd_offset(dst, start);

        end = vma->vm_end;
        addr = start;
        while (addr && (addr < end-1)) {
                next = (addr + PGDIR_SIZE) & PGDIR_MASK;
                if (next > end || next <= addr)
                        next = end;
                if (pgd_none(*src_pgd))
                        goto next_pgd;
                if (pgd_bad(*src_pgd)) {
                        pgd_ERROR(*src_pgd);
                        pgd_clear(src_pgd);
                        goto next_pgd;
                }
                err = copy_pud_range(dst, src, dst_pgd, src_pgd,
                                                        vma, addr, next);
```

```c
			if (err)
				break;

next_pgd:
			src_pgd++;
			dst_pgd++;
			addr = next;
		}

		return err;
}

static void zap_pte_range(struct mmu_gather *tlb,
			pmd_t *pmd, unsigned long address,
			unsigned long size, struct zap_details *details)
{
		unsigned long offset;
		pte_t *ptep;

		if (pmd_none(*pmd))
			return;
		if (unlikely(pmd_bad(*pmd))) {
			pmd_ERROR(*pmd);
			pmd_clear(pmd);
			return;
		}
		ptep = pte_offset_map(pmd, address);
		offset = address & ~PMD_MASK;
		if (offset + size > PMD_SIZE)
			size = PMD_SIZE - offset;
		size &= PAGE_MASK;
		if (details && !details->check_mapping && !details->nonlinear_vma)
			details = NULL;
		for (offset=0; offset < size; ptep++, offset += PAGE_SIZE) {
			pte_t pte = *ptep;
			if (pte_none(pte))
				continue;
			if (pte_present(pte)) {
				struct page *page = NULL;
				unsigned long pfn = pte_pfn(pte);
				if (pfn_valid(pfn)) {
					page = pfn_to_page(pfn);
					if (PageReserved(page))
						page = NULL;
				}
				if (unlikely(details) && page) {
					/*
					 * unmap_shared_mapping_pages() wants to
					 * invalidate cache without truncating:
					 * unmap shared but keep private pages.
					 */
					if (details->check_mapping &&
					    details->check_mapping != page->mapping)
						continue;
					/*
					 * Each page->index must be checked when
					 * invalidating or truncating nonlinear.
					 */
					if (details->nonlinear_vma &&
					    (page->index < details->first_index ||
					     page->index > details->last_index))
						continue;
				}
```

```
			pte = ptep_get_and_clear(ptep);
			tlb_remove_tlb_entry(tlb, ptep, address+offset);
			if (unlikely(!page))
				continue;
			if (unlikely(details) && details->nonlinear_vma
				&& linear_page_index(details->nonlinear_vma,
						address+offset) != page->index)
				set_pte(ptep, pgoff_to_pte(page->index));
			if (pte_dirty(pte))
				set_page_dirty(page);
			if (PageAnon(page))
				tlb->mm->anon_rss--;
			else if	(pte_young(pte))
				mark_page_accessed(page);
			tlb->freed++;
			page_remove_rmap(page);
			tlb_remove_page(tlb, page);
			continue;
		}
		/*
		 * If details->check_mapping, we leave swap entries;
		 * if details->nonlinear_vma, we leave file entries.
		 */
		if (unlikely(details))
			continue;
		if (!pte_file(pte))
			free_swap_and_cache(pte_to_swp_entry(pte));
		pte_clear(ptep);
	}
	pte_unmap(ptep-1);
}

static void zap_pmd_range(struct mmu_gather *tlb,
		pud_t *pud, unsigned long address,
		unsigned long size, struct zap_details *details)
{
	pmd_t * pmd;
	unsigned long end;

	if (pud_none(*pud))
		return;
	if (unlikely(pud_bad(*pud))) {
		pud_ERROR(*pud);
		pud_clear(pud);
		return;
	}
	pmd = pmd_offset(pud, address);
	end = address + size;
	if (end > ((address + PUD_SIZE) & PUD_MASK))
		end = ((address + PUD_SIZE) & PUD_MASK);
	do {
		zap_pte_range(tlb, pmd, address, end - address, details);
		address = (address + PMD_SIZE) & PMD_MASK;
		pmd++;
	} while	(address && (address < end));
}

static void zap_pud_range(struct mmu_gather *tlb,
		pgd_t * pgd, unsigned long address,
		unsigned long end, struct zap_details *details)
{
	pud_t * pud;
```

```
	if (pgd_none(*pgd))
		return;
	if (unlikely(pgd_bad(*pgd))) {
		pgd_ERROR(*pgd);
		pgd_clear(pgd);
		return;
	}
	pud = pud_offset(pgd, address);
	do {
		zap_pmd_range(tlb, pud, address, end - address, details);
		address = (address + PUD_SIZE) & PUD_MASK;
		pud++;
	} while	(address && (address < end));
}

static void unmap_page_range(struct mmu_gather *tlb,
		struct vm_area_struct *vma, unsigned long address,
		unsigned long end, struct zap_details *details)
{
	unsigned long next;
	pgd_t *pgd;
	int i;

	BUG_ON(address >= end);
	pgd = pgd_offset(vma->vm_mm, address);
	tlb_start_vma(tlb, vma);
	for (i = pgd_index(address); i <= pgd_index(end-1); i++) {
		next = (address + PGDIR_SIZE) & PGDIR_MASK;
		if (next <= address || next > end)
			next = end;
		zap_pud_range(tlb, pgd, address, next, details);
		address = next;
		pgd++;
	}
	tlb_end_vma(tlb, vma);
}

#ifdef CONFIG_PREEMPT
# define ZAP_BLOCK_SIZE (8 * PAGE_SIZE)
#else
/* No preempt: go for improved straight-line efficiency */
# define ZAP_BLOCK_SIZE (1024 * PAGE_SIZE)
#endif

/**
 * unmap_vmas - unmap a range of memory covered by a list of vma's
 * @tlbp: address of the caller's struct mmu_gather
 * @mm: the controlling mm_struct
 * @vma: the starting vma
 * @start_addr: virtual address at which to start unmapping
 * @end_addr: virtual address at which to end unmapping
 * @nr_accounted: Place number of unmapped pages in vm-accountable vma's here
 * @details: details of nonlinear truncation or shared cache invalidation
 *
 * Returns the number of vma's which were covered by the unmapping.
 *
 * Unmap all pages in the vma list. Called under page_table_lock.
 *
 * We aim to not hold page_table_lock for too long (for scheduling latency
 * reasons). So zap pages in ZAP_BLOCK_SIZE bytecounts. This means we need to
 * return the ending mmu_gather to the caller.
 *
 * Only addresses between 'start' and 'end' will be unmapped.
```

```
 *
 * The VMA list must be sorted in ascending virtual address order.
 *
 * unmap_vmas() assumes that the caller will flush the whole unmapped address
 * range after unmap_vmas() returns. So the only responsibility here is to
 * ensure that any thus-far unmapped pages are flushed before unmap_vmas()
 * drops the lock and schedules.
 */
int unmap_vmas(struct mmu_gather **tlbp, struct mm_struct *mm,
		struct vm_area_struct *vma, unsigned long start_addr,
		unsigned long end_addr, unsigned long *nr_accounted,
		struct zap_details *details)
{
	unsigned long zap_bytes = ZAP_BLOCK_SIZE;
	unsigned long tlb_start = 0;	/* For tlb_finish_mmu */
	int tlb_start_valid = 0;
	int ret = 0;
	spinlock_t *i_mmap_lock = details? details->i_mmap_lock: NULL;
	int fullmm = tlb_is_full_mm(*tlbp);

	for ( ; vma && vma->vm_start < end_addr; vma = vma->vm_next) {
		unsigned long start;
		unsigned long end;

		start = max(vma->vm_start, start_addr);
		if (start >= vma->vm_end)
			continue;
		end = min(vma->vm_end, end_addr);
		if (end <= vma->vm_start)
			continue;

		if (vma->vm_flags & VM_ACCOUNT)
			*nr_accounted += (end - start) >> PAGE_SHIFT;

		ret++;
		while (start != end) {
			unsigned long block;

			if (!tlb_start_valid) {
				tlb_start = start;
				tlb_start_valid = 1;
			}

			if (is_vm_hugetlb_page(vma)) {
				block = end - start;
				unmap_hugepage_range(vma, start, end);
			} else {
				block = min(zap_bytes, end - start);
				unmap_page_range(*tlbp, vma, start,
						start + block, details);
			}

			start += block;
			zap_bytes -= block;
			if ((long)zap_bytes > 0)
				continue;

			tlb_finish_mmu(*tlbp, tlb_start, start);

			if (need_resched() ||
				need_lockbreak(&mm->page_table_lock) ||
				(i_mmap_lock && need_lockbreak(i_mmap_lock))) {
				if (i_mmap_lock) {
```

```
                                                /* must reset count of rss freed */
                                                *tlbp = tlb_gather_mmu(mm, fullmm);
                                                details->break_addr = start;
                                                goto out;
                                        }
                                        spin_unlock(&mm->page_table_lock);
                                        cond_resched();
                                        spin_lock(&mm->page_table_lock);
                                }

                                *tlbp = tlb_gather_mmu(mm, fullmm);
                                tlb_start_valid = 0;
                                zap_bytes = ZAP_BLOCK_SIZE;
                        }
                }
out:
        return ret;
}

/**
 * zap_page_range - remove user pages in a given range
 * @vma: vm_area_struct holding the applicable pages
 * @address: starting address of pages to zap
 * @size: number of bytes to zap
 * @details: details of nonlinear truncation or shared cache invalidation
 */
void zap_page_range(struct vm_area_struct *vma, unsigned long address,
                unsigned long size, struct zap_details *details)
{
        struct mm_struct *mm = vma->vm_mm;
        struct mmu_gather *tlb;
        unsigned long end = address + size;
        unsigned long nr_accounted = 0;

        if (is_vm_hugetlb_page(vma)) {
                zap_hugepage_range(vma, address, size);
                return;
        }

        lru_add_drain();
        spin_lock(&mm->page_table_lock);
        tlb = tlb_gather_mmu(mm, 0);
        unmap_vmas(&tlb, mm, vma, address, end, &nr_accounted, details);
        tlb_finish_mmu(tlb, address, end);
        acct_update_integrals();
        spin_unlock(&mm->page_table_lock);
}

/*
 * Do a quick page-table lookup for a single page.
 * mm->page_table_lock must be held.
 */
static struct page *
__follow_page(struct mm_struct *mm, unsigned long address, int read, int write)
{
        pgd_t *pgd;
        pud_t *pud;
        pmd_t *pmd;
        pte_t *ptep, pte;
        unsigned long pfn;
        struct page *page;

        page = follow_huge_addr(mm, address, write);
```

```
	if (! IS_ERR(page))
		return page;

	pgd = pgd_offset(mm, address);
	if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
		goto out;

	pud = pud_offset(pgd, address);
	if (pud_none(*pud) || unlikely(pud_bad(*pud)))
		goto out;

	pmd = pmd_offset(pud, address);
	if (pmd_none(*pmd) || unlikely(pmd_bad(*pmd)))
		goto out;
	if (pmd_huge(*pmd))
		return follow_huge_pmd(mm, address, pmd, write);

	ptep = pte_offset_map(pmd, address);
	if (!ptep)
		goto out;

	pte = *ptep;
	pte_unmap(ptep);
	if (pte_present(pte)) {
		if (write && !pte_write(pte))
			goto out;
		if (read && !pte_read(pte))
			goto out;
		pfn = pte_pfn(pte);
		if (pfn_valid(pfn)) {
			page = pfn_to_page(pfn);
			if (write && !pte_dirty(pte) && !PageDirty(page))
				set_page_dirty(page);
			mark_page_accessed(page);
			return page;
		}
	}

out:
	return NULL;
}

struct page *
follow_page(struct mm_struct *mm, unsigned long address, int write)
{
	return __follow_page(mm, address, /*read*/0, write);
}

int
check_user_page_readable(struct mm_struct *mm, unsigned long address)
{
	return __follow_page(mm, address, /*read*/1, /*write*/0) != NULL;
}

EXPORT_SYMBOL(check_user_page_readable);

/*
 * Given a physical address, is there a useful struct page pointing to
 * it? This may become more complex in the future if we start dealing
 * with IO-aperture pages for direct-IO.
 */

static inline struct page *get_page_map(struct page *page)
```

```
{
	if (!pfn_valid(page_to_pfn(page)))
		return NULL;
	return page;
}


static inline int
untouched_anonymous_page(struct mm_struct* mm, struct vm_area_struct *vma,
			 unsigned long address)
{
	pgd_t *pgd;
	pud_t *pud;
	pmd_t *pmd;

	/* Check if the vma is for an anonymous mapping. */
	if (vma->vm_ops && vma->vm_ops->nopage)
		return 0;

	/* Check if page directory entry exists. */
	pgd = pgd_offset(mm, address);
	if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
		return 1;

	pud = pud_offset(pgd, address);
	if (pud_none(*pud) || unlikely(pud_bad(*pud)))
		return 1;

	/* Check if page middle directory entry exists. */
	pmd = pmd_offset(pud, address);
	if (pmd_none(*pmd) || unlikely(pmd_bad(*pmd)))
		return 1;

	/* There is a pte slot for 'address' in 'mm'. */
	return 0;
}


int get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
		unsigned long start, int len, int write, int force,
		struct page **pages, struct vm_area_struct **vmas)
{
	int i;
	unsigned int flags;

	/*
	 * Require read or write permissions.
	 * If 'force' is set, we only require the "MAY" flags.
	 */
	flags = write ? (VM_WRITE | VM_MAYWRITE) : (VM_READ | VM_MAYREAD);
	flags &= force ? (VM_MAYREAD | VM_MAYWRITE) : (VM_READ | VM_WRITE);
	i = 0;

	do {
		struct vm_area_struct * vma;

		vma = find_extend_vma(mm, start);
		if (!vma && in_gate_area(tsk, start)) {
			unsigned long pg = start & PAGE_MASK;
			struct vm_area_struct *gate_vma = get_gate_vma(tsk);
			pgd_t *pgd;
			pud_t *pud;
			pmd_t *pmd;
```

```
				pte_t *pte;
				if (write) /* user gate pages are read-only */
					return i ? : -EFAULT;
				if (pg > TASK_SIZE)
					pgd = pgd_offset_k(pg);
				else
					pgd = pgd_offset_gate(mm, pg);
				BUG_ON(pgd_none(*pgd));
				pud = pud_offset(pgd, pg);
				BUG_ON(pud_none(*pud));
				pmd = pmd_offset(pud, pg);
				BUG_ON(pmd_none(*pmd));
				pte = pte_offset_map(pmd, pg);
				BUG_ON(pte_none(*pte));
				if (pages) {
					pages[i] = pte_page(*pte);
					get_page(pages[i]);
				}
				pte_unmap(pte);
				if (vmas)
					vmas[i] = gate_vma;
				i++;
				start += PAGE_SIZE;
				len--;
				continue;
			}

			if (!vma || (vma->vm_flags & VM_IO)
					|| !(flags & vma->vm_flags))
				return i ? : -EFAULT;

			if (is_vm_hugetlb_page(vma)) {
				i = follow_hugetlb_page(mm, vma, pages, vmas,
							&start, &len, i);
				continue;
			}
			spin_lock(&mm->page_table_lock);
			do {
				struct page *map;
				int lookup_write = write;

				cond_resched_lock(&mm->page_table_lock);
				while (!(map = follow_page(mm, start, lookup_write))) {
					/*
					 * Shortcut for anonymous pages. We don't want
					 * to force the creation of pages tables for
					 * insanly big anonymously mapped areas that
					 * nobody touched so far. This is important
					 * for doing a core dump for these mappings.
					 */
					if (!lookup_write &&
					    untouched_anonymous_page(mm,vma,start)) {
						map = ZERO_PAGE(start);
						break;
					}
					spin_unlock(&mm->page_table_lock);
					switch (handle_mm_fault(mm,vma,start,write)) {
					case VM_FAULT_MINOR:
						tsk->min_flt++;
						break;
					case VM_FAULT_MAJOR:
						tsk->maj_flt++;
						break;
```

```
					case VM_FAULT_SIGBUS:
						return i ? i : -EFAULT;
					case VM_FAULT_OOM:
						return i ? i : -ENOMEM;
					default:
						BUG();
					}
					/*
					 * Now that we have performed a write fault
					 * and surely no longer have a shared page we
					 * shouldn't write, we shouldn't ignore an
					 * unwritable page in the page table if
					 * we are forcing write access.
					 */
					lookup_write = write && !force;
					spin_lock(&mm->page_table_lock);
				}
				if (pages) {
					pages[i] = get_page_map(map);
					if (!pages[i]) {
						spin_unlock(&mm->page_table_lock);
						while (i--)
							page_cache_release(pages[i]);
						i = -EFAULT;
						goto out;
					}
					flush_dcache_page(pages[i]);
					if (!PageReserved(pages[i]))
						page_cache_get(pages[i]);
				}
				if (vmas)
					vmas[i] = vma;
				i++;
				start += PAGE_SIZE;
				len--;
			} while(len && start < vma->vm_end);
			spin_unlock(&mm->page_table_lock);
	} while(len);
out:
	return i;
}

EXPORT_SYMBOL(get_user_pages);

static void zeromap_pte_range(pte_t * pte, unsigned long address,
					unsigned long size, pgprot_t prot)
{
	unsigned long end;

	address &= ~PMD_MASK;
	end = address + size;
	if (end > PMD_SIZE)
		end = PMD_SIZE;
	do {
		pte_t zero_pte = pte_wrprotect(mk_pte(ZERO_PAGE(address), prot));
		BUG_ON(!pte_none(*pte));
		set_pte(pte, zero_pte);
		address += PAGE_SIZE;
		pte++;
	} while		(address && (address < end));
}

static inline int zeromap_pmd_range(struct mm_struct *mm, pmd_t * pmd,
```

```
                unsigned long address, unsigned long size, pgprot_t prot)
{
        unsigned long base, end;

        base = address & PUD_MASK;
        address &= ~PUD_MASK;
        end = address + size;
        if (end > PUD_SIZE)
                end = PUD_SIZE;
        do {
                pte_t * pte = pte_alloc_map(mm, pmd, base + address);
                if (!pte)
                        return -ENOMEM;
                zeromap_pte_range(pte, base + address, end - address, prot);
                pte_unmap(pte);
                address = (address + PMD_SIZE) & PMD_MASK;
                pmd++;
        } while (address && (address < end));
        return 0;
}

static inline int zeromap_pud_range(struct mm_struct *mm, pud_t * pud,
                                    unsigned long address,
                                    unsigned long size, pgprot_t prot)
{
        unsigned long base, end;
        int error = 0;

        base = address & PGDIR_MASK;
        address &= ~PGDIR_MASK;
        end = address + size;
        if (end > PGDIR_SIZE)
                end = PGDIR_SIZE;
        do {
                pmd_t * pmd = pmd_alloc(mm, pud, base + address);
                error = -ENOMEM;
                if (!pmd)
                        break;
                error = zeromap_pmd_range(mm, pmd, base + address,
                                          end - address, prot);
                if (error)
                        break;
                address = (address + PUD_SIZE) & PUD_MASK;
                pud++;
        } while (address && (address < end));
        return 0;
}

int zeromap_page_range(struct vm_area_struct *vma, unsigned long address,
                                        unsigned long size, pgprot_t prot)
{
        int i;
        int error = 0;
        pgd_t * pgd;
        unsigned long beg = address;
        unsigned long end = address + size;
        unsigned long next;
        struct mm_struct *mm = vma->vm_mm;

        pgd = pgd_offset(mm, address);
        flush_cache_range(vma, beg, end);
        BUG_ON(address >= end);
        BUG_ON(end > vma->vm_end);
```

```
	spin_lock(&mm->page_table_lock);
	for (i = pgd_index(address); i <= pgd_index(end-1); i++) {
		pud_t *pud = pud_alloc(mm, pgd, address);
		error = -ENOMEM;
		if (!pud)
			break;
		next = (address + PGDIR_SIZE) & PGDIR_MASK;
		if (next <= beg || next > end)
			next = end;
		error = zeromap_pud_range(mm, pud, address,
						next - address, prot);
		if (error)
			break;
		address = next;
		pgd++;
	}
	/*
	 * Why flush? zeromap_pte_range has a BUG_ON for !pte_none()
	 */
	flush_tlb_range(vma, beg, end);
	spin_unlock(&mm->page_table_lock);
	return error;
}

/*
 * maps a range of physical memory into the requested pages. the old
 * mappings are removed. any references to nonexistent pages results
 * in null mappings (currently treated as "copy-on-access")
 */
static inline void
remap_pte_range(pte_t * pte, unsigned long address, unsigned long size,
		unsigned long pfn, pgprot_t prot)
{
	unsigned long end;

	address &= ~PMD_MASK;
	end = address + size;
	if (end > PMD_SIZE)
		end = PMD_SIZE;
	do {
		BUG_ON(!pte_none(*pte));
		if (!pfn_valid(pfn) || PageReserved(pfn_to_page(pfn)))
			set_pte(pte, pfn_pte(pfn, prot));
		address += PAGE_SIZE;
		pfn++;
		pte++;
	} while (address && (address < end));
}

static inline int
remap_pmd_range(struct mm_struct *mm, pmd_t * pmd, unsigned long address,
		unsigned long size, unsigned long pfn, pgprot_t prot)
{
	unsigned long base, end;

	base = address & PUD_MASK;
	address &= ~PUD_MASK;
	end = address + size;
	if (end > PUD_SIZE)
		end = PUD_SIZE;
	pfn -= (address >> PAGE_SHIFT);
	do {
```

```
                pte_t * pte = pte_alloc_map(mm, pmd, base + address);
                if (!pte)
                        return -ENOMEM;
                remap_pte_range(pte, base + address, end - address,
                                (address >> PAGE_SHIFT) + pfn, prot);
                pte_unmap(pte);
                address = (address + PMD_SIZE) & PMD_MASK;
                pmd++;
        } while (address && (address < end));
        return 0;
}

static inline int remap_pud_range(struct mm_struct *mm, pud_t * pud,
                                  unsigned long address, unsigned long size,
                                  unsigned long pfn, pgprot_t prot)
{
        unsigned long base, end;
        int error;

        base = address & PGDIR_MASK;
        address &= ~PGDIR_MASK;
        end = address + size;
        if (end > PGDIR_SIZE)
                end = PGDIR_SIZE;
        pfn -= address >> PAGE_SHIFT;
        do {
                pmd_t *pmd = pmd_alloc(mm, pud, base+address);
                error = -ENOMEM;
                if (!pmd)
                        break;
                error = remap_pmd_range(mm, pmd, base + address, end - address,
                                (address >> PAGE_SHIFT) + pfn, prot);
                if (error)
                        break;
                address = (address + PUD_SIZE) & PUD_MASK;
                pud++;
        } while (address && (address < end));
        return error;
}

/* Note: this is only safe if the mm semaphore is held when called. */
int remap_pfn_range(struct vm_area_struct *vma, unsigned long from,
                    unsigned long pfn, unsigned long size, pgprot_t prot)
{
        int error = 0;
        pgd_t *pgd;
        unsigned long beg = from;
        unsigned long end = from + size;
        unsigned long next;
        struct mm_struct *mm = vma->vm_mm;
        int i;

        pfn -= from >> PAGE_SHIFT;
        pgd = pgd_offset(mm, from);
        flush_cache_range(vma, beg, end);
        BUG_ON(from >= end);

        /*
         * Physically remapped pages are special. Tell the
         * rest of the world about it:
         *   VM_IO tells people not to look at these pages
         *      (accesses can have side effects).
         *   VM_RESERVED tells swapout not to try to touch
```

```
	 *		this region.
	 */
	vma->vm_flags |= VM_IO | VM_RESERVED;

	spin_lock(&mm->page_table_lock);
	for (i = pgd_index(beg); i <= pgd_index(end-1); i++) {
		pud_t *pud = pud_alloc(mm, pgd, from);
		error = -ENOMEM;
		if (!pud)
			break;
		next = (from + PGDIR_SIZE) & PGDIR_MASK;
		if (next > end || next <= from)
			next = end;
		error =	remap_pud_range(mm, pud, from, end - from,
					pfn + (from >> PAGE_SHIFT), prot);
		if (error)
			break;
		from = next;
		pgd++;
	}
	/*
	 * Why flush? remap_pte_range has a BUG_ON for !pte_none()
	 */
	flush_tlb_range(vma, beg, end);
	spin_unlock(&mm->page_table_lock);

	return error;
}

EXPORT_SYMBOL(remap_pfn_range);

/*
 * Do pte_mkwrite, but only if the vma says VM_WRITE. We do this when
 * servicing faults for write access. In the normal case, do always want
 * pte_mkwrite. But get_user_pages can cause write faults for mappings
 * that do not have writing enabled, when used by access_process_vm.
 */
static inline pte_t maybe_mkwrite(pte_t pte, struct vm_area_struct *vma)
{
	if (likely(vma->vm_flags & VM_WRITE))
		pte = pte_mkwrite(pte);
	return pte;
}

/*
 * We hold the mm semaphore for reading and vma->vm_mm->page_table_lock
 */
static inline void break_cow(struct vm_area_struct * vma, struct page * new_page, unsigned long address,
		pte_t *page_table)
{
	pte_t entry;

	flush_cache_page(vma, address);
	entry = maybe_mkwrite(pte_mkdirty(mk_pte(new_page, vma->vm_page_prot)),
			      vma);
	ptep_establish(vma, address, page_table, entry);
	update_mmu_cache(vma, address, entry);
}

/*
 * This routine handles present pages, when users try to write
 * to a shared page. It is done by copying the page to a new address
 * and decrementing the shared-page counter for the old page.
```

```
 *
 * Goto-purists beware: the only reason for goto's here is that it results
 * in better assembly code.. The "default" path will see no jumps at all.
 *
 * Note that this routine assumes that the protection checks have been
 * done by the caller (the low-level page fault routine in most cases).
 * Thus we can safely just mark it writable once we've done any necessary
 * COW.
 *
 * We also mark the page dirty at this point even though the page will
 * change only once the write actually happens. This avoids a few races,
 * and potentially makes it more efficient.
 *
 * We hold the mm semaphore and the page_table_lock on entry and exit
 * with the page_table_lock released.
 */
static int do_wp_page(struct mm_struct *mm, struct vm_area_struct * vma,
		unsigned long address, pte_t *page_table, pmd_t *pmd, pte_t pte)
{
	struct page *old_page, *new_page;
	unsigned long pfn = pte_pfn(pte);
	pte_t entry;

	if (unlikely(!pfn_valid(pfn))) {
		/*
		 * This should really halt the system so it can be debugged or
		 * at least the kernel stops what it's doing before it corrupts
		 * data, but for the moment just pretend this is OOM.
		 */
		pte_unmap(page_table);
		printk(KERN_ERR "do_wp_page: bogus page at address %08lx\n",
				address);
		spin_unlock(&mm->page_table_lock);
		return VM_FAULT_OOM;
	}
	old_page = pfn_to_page(pfn);

	if (!TestSetPageLocked(old_page)) {
		int reuse = can_share_swap_page(old_page);
		unlock_page(old_page);
		if (reuse) {
			flush_cache_page(vma, address);
			entry = maybe_mkwrite(pte_mkyoung(pte_mkdirty(pte)),
					      vma);
			ptep_set_access_flags(vma, address, page_table, entry, 1);
			update_mmu_cache(vma, address, entry);
			pte_unmap(page_table);
			spin_unlock(&mm->page_table_lock);
			return VM_FAULT_MINOR;
		}
	}
	pte_unmap(page_table);

	/*
	 * Ok, we need to copy. Oh, well..
	 */
	if (!PageReserved(old_page))
		page_cache_get(old_page);
	spin_unlock(&mm->page_table_lock);

	if (unlikely(anon_vma_prepare(vma)))
		goto no_new_page;
	if (old_page == ZERO_PAGE(address)) {
```

```
			new_page = alloc_zeroed_user_highpage(vma, address);
			if (!new_page)
				goto no_new_page;
		} else {
			new_page = alloc_page_vma(GFP_HIGHUSER, vma, address);
			if (!new_page)
				goto no_new_page;
			copy_user_highpage(new_page, old_page, address);
		}
		/*
		 * Re-check the pte - we dropped the lock
		 */
		spin_lock(&mm->page_table_lock);
		page_table = pte_offset_map(pmd, address);
		if (likely(pte_same(*page_table, pte))) {
			if (PageAnon(old_page))
				mm->anon_rss--;
			if (PageReserved(old_page)) {
				++mm->rss;
				acct_update_integrals();
				update_mem_hiwater();
			} else
				page_remove_rmap(old_page);
			break_cow(vma, new_page, address, page_table);
			lru_cache_add_active(new_page);
			page_add_anon_rmap(new_page, vma, address);

			/* Free the old page.. */
			new_page = old_page;
		}
		pte_unmap(page_table);
		page_cache_release(new_page);
		page_cache_release(old_page);
		spin_unlock(&mm->page_table_lock);
		return VM_FAULT_MINOR;

no_new_page:
		page_cache_release(old_page);
		return VM_FAULT_OOM;
}

/*
 * Helper functions for unmap_mapping_range().
 *
 * __ Notes on dropping i_mmap_lock to reduce latency while unmapping __
 *
 * We have to restart searching the prio_tree whenever we drop the lock,
 * since the iterator is only valid while the lock is held, and anyway
 * a later vma might be split and reinserted earlier while lock dropped.
 *
 * The list of nonlinear vmas could be handled more efficiently, using
 * a placeholder, but handle it in the same way until a need is shown.
 * It is important to search the prio_tree before nonlinear list: a vma
 * may become nonlinear and be shifted from prio_tree to nonlinear list
 * while the lock is dropped; but never shifted from list to prio_tree.
 *
 * In order to make forward progress despite restarting the search,
 * vm_truncate_count is used to mark a vma as now dealt with, so we can
 * quickly skip it next time around. Since the prio_tree search only
 * shows us those vmas affected by unmapping the range in question, we
 * can't efficiently keep all vmas in step with mapping->truncate_count:
 * so instead reset them all whenever it wraps back to 0 (then go to 1).
 * mapping->truncate_count and vma->vm_truncate_count are protected by
```

```c
 * i_mmap_lock.
 *
 * In order to make forward progress despite repeatedly restarting some
 * large vma, note the break_addr set by unmap_vmas when it breaks out:
 * and restart from that address when we reach that vma again. It might
 * have been split or merged, shrunk or extended, but never shifted: so
 * restart_addr remains valid so long as it remains in the vma's range.
 * unmap_mapping_range forces truncate_count to leap over page-aligned
 * values so we can save vma's restart_addr in its truncate_count field.
 */
#define is_restart_addr(truncate_count) (!((truncate_count) & ~PAGE_MASK))

static void reset_vma_truncate_counts(struct address_space *mapping)
{
	struct vm_area_struct *vma;
	struct prio_tree_iter iter;

	vma_prio_tree_foreach(vma, &iter, &mapping->i_mmap, 0, ULONG_MAX)
		vma->vm_truncate_count = 0;
	list_for_each_entry(vma, &mapping->i_mmap_nonlinear, shared.vm_set.list)
		vma->vm_truncate_count = 0;
}

static int unmap_mapping_range_vma(struct vm_area_struct *vma,
		unsigned long start_addr, unsigned long end_addr,
		struct zap_details *details)
{
	unsigned long restart_addr;
	int need_break;

again:
	restart_addr = vma->vm_truncate_count;
	if (is_restart_addr(restart_addr) && start_addr < restart_addr) {
		start_addr = restart_addr;
		if (start_addr >= end_addr) {
			/* Top of vma has been split off since last time */
			vma->vm_truncate_count = details->truncate_count;
			return 0;
		}
	}

	details->break_addr = end_addr;
	zap_page_range(vma, start_addr, end_addr - start_addr, details);

	/*
	 * We cannot rely on the break test in unmap_vmas:
	 * on the one hand, we don't want to restart our loop
	 * just because that broke out for the page_table_lock;
	 * on the other hand, it does no test when vma is small.
	 */
	need_break = need_resched() ||
			need_lockbreak(details->i_mmap_lock);

	if (details->break_addr >= end_addr) {
		/* We have now completed this vma: mark it so */
		vma->vm_truncate_count = details->truncate_count;
		if (!need_break)
			return 0;
	} else {
		/* Note restart_addr in vma's truncate_count field */
		vma->vm_truncate_count = details->break_addr;
		if (!need_break)
			goto again;
```

```
	}

	spin_unlock(details->i_mmap_lock);
	cond_resched();
	spin_lock(details->i_mmap_lock);
	return -EINTR;
}

static inline void unmap_mapping_range_tree(struct prio_tree_root *root,
					    struct zap_details *details)
{
	struct vm_area_struct *vma;
	struct prio_tree_iter iter;
	pgoff_t vba, vea, zba, zea;

restart:
	vma_prio_tree_foreach(vma, &iter, root,
			details->first_index, details->last_index) {
		/* Skip	quickly over those we have already dealt with */
		if (vma->vm_truncate_count == details->truncate_count)
			continue;

		vba = vma->vm_pgoff;
		vea = vba + ((vma->vm_end - vma->vm_start) >> PAGE_SHIFT) - 1;
		/* Assume for now that PAGE_CACHE_SHIFT == PAGE_SHIFT */
		zba = details->first_index;
		if (zba < vba)
			zba = vba;
		zea = details->last_index;
		if (zea > vea)
			zea = vea;

		if (unmap_mapping_range_vma(vma,
			((zba - vba) << PAGE_SHIFT) + vma->vm_start,
			((zea - vba + 1) << PAGE_SHIFT) + vma->vm_start,
				details) < 0)
			goto restart;
	}
}

static inline void unmap_mapping_range_list(struct list_head *head,
					    struct zap_details *details)
{
	struct vm_area_struct *vma;

	/*
	 * In nonlinear VMAs there is no correspondence between virtual address
	 * offset and file offset. So we must perform an exhaustive search
	 * across *all* the pages in each nonlinear VMA, not just the pages
	 * whose virtual address lies outside the file truncation point.
	 */
restart:
	list_for_each_entry(vma, head, shared.vm_set.list) {
		/* Skip quickly over those we have already dealt with */
		if (vma->vm_truncate_count == details->truncate_count)
			continue;
		details->nonlinear_vma = vma;
		if (unmap_mapping_range_vma(vma, vma->vm_start,
					vma->vm_end, details) < 0)
			goto restart;
	}
}
```

```
/**
 * unmap_mapping_range - unmap the portion of all mmaps
 * in the specified address_space corresponding to the specified
 * page range in the underlying file.
 * @address_space: the address space containing mmaps to be unmapped.
 * @holebegin: byte in first page to unmap, relative to the start of
 * the underlying file. This will be rounded down to a PAGE_SIZE
 * boundary. Note that this is different from vmtruncate(), which
 * must keep the partial page. In contrast, we must get rid of
 * partial pages.
 * @holelen: size of prospective hole in bytes. This will be rounded
 * up to a PAGE_SIZE boundary. A holelen of zero truncates to the
 * end of the file.
 * @even_cows: 1 when truncating a file, unmap even private COWed pages;
 * but 0 when invalidating pagecache, don't throw away private data.
 */
void unmap_mapping_range(struct address_space *mapping,
		loff_t const holebegin, loff_t const holelen, int even_cows)
{
	struct zap_details details;
	pgoff_t hba = holebegin >> PAGE_SHIFT;
	pgoff_t hlen = (holelen + PAGE_SIZE - 1) >> PAGE_SHIFT;

	/* Check for overflow. */
	if (sizeof(holelen) > sizeof(hlen)) {
		long long holeend =
			(holebegin + holelen + PAGE_SIZE - 1) >> PAGE_SHIFT;
		if (holeend & ~(long long)ULONG_MAX)
			hlen = ULONG_MAX - hba + 1;
	}

	details.check_mapping = even_cows? NULL: mapping;
	details.nonlinear_vma = NULL;
	details.first_index = hba;
	details.last_index = hba + hlen - 1;
	if (details.last_index < details.first_index)
		details.last_index = ULONG_MAX;
	details.i_mmap_lock = &mapping->i_mmap_lock;

	spin_lock(&mapping->i_mmap_lock);

	/* serialize i_size write against truncate_count write */
	smp_wmb();
	/* Protect against page faults, and endless unmapping loops */
	mapping->truncate_count++;
	/*
	 * For archs where spin_lock has inclusive semantics like ia64
	 * this smp_mb() will prevent to read pagetable contents
	 * before the truncate_count increment is visible to
	 * other cpus.
	 */
	smp_mb();
	if (unlikely(is_restart_addr(mapping->truncate_count))) {
		if (mapping->truncate_count == 0)
			reset_vma_truncate_counts(mapping);
		mapping->truncate_count++;
	}
	details.truncate_count = mapping->truncate_count;

	if (unlikely(!prio_tree_empty(&mapping->i_mmap)))
		unmap_mapping_range_tree(&mapping->i_mmap, &details);
	if (unlikely(!list_empty(&mapping->i_mmap_nonlinear)))
		unmap_mapping_range_list(&mapping->i_mmap_nonlinear, &details);
```

```
		spin_unlock(&mapping->i_mmap_lock);
}
EXPORT_SYMBOL(unmap_mapping_range);

/*
 * Handle all mappings that got truncated by a "truncate()"
 * system call.
 *
 * NOTE! We have to be ready to update the memory sharing
 * between the file and the memory map for a potential last
 * incomplete page. Ugly, but necessary.
 */
int vmtruncate(struct inode * inode, loff_t offset)
{
	struct address_space *mapping = inode->i_mapping;
	unsigned long limit;

	if (inode->i_size < offset)
		goto do_expand;
	/*
	 * truncation of in-use swapfiles is disallowed - it would cause
	 * subsequent swapout to scribble on the now-freed blocks.
	 */
	if (IS_SWAPFILE(inode))
		goto out_busy;
	i_size_write(inode, offset);
	unmap_mapping_range(mapping, offset + PAGE_SIZE - 1, 0, 1);
	truncate_inode_pages(mapping, offset);
	goto out_truncate;

do_expand:
	limit = current->signal->rlim[RLIMIT_FSIZE].rlim_cur;
	if (limit != RLIM_INFINITY && offset > limit)
		goto out_sig;
	if (offset > inode->i_sb->s_maxbytes)
		goto out_big;
	i_size_write(inode, offset);

out_truncate:
	if (inode->i_op && inode->i_op->truncate)
		inode->i_op->truncate(inode);
	return 0;
out_sig:
	send_sig(SIGXFSZ, current, 0);
out_big:
	return -EFBIG;
out_busy:
	return -ETXTBSY;
}

EXPORT_SYMBOL(vmtruncate);

/*
 * Primitive swap readahead code. We simply read an aligned block of
 * (1 << page_cluster) entries in the swap area. This method is chosen
 * because it doesn't cost us any seek time. We also make sure to queue
 * the 'original' request together with the readahead ones...
 *
 * This has been extended to use the NUMA policies from the mm triggering
 * the readahead.
 *
 * Caller must hold down_read on the vma->vm_mm if vma is not NULL.
 */
```

```
void swapin_readahead(swp_entry_t entry, unsigned long addr,struct vm_area_struct *vma)
{
#ifdef CONFIG_NUMA
        struct vm_area_struct *next_vma = vma ? vma->vm_next : NULL;
#endif
        int i, num;
        struct page *new_page;
        unsigned long offset;

        /*
         * Get the number of handles we should do readahead io to.
         */
        num = valid_swaphandles(entry, &offset);
        for (i = 0; i < num; offset++, i++) {
                /* Ok, do the async read-ahead now */
                new_page = read_swap_cache_async(swp_entry(swp_type(entry),
                                                           offset), vma, addr);
                if (!new_page)
                        break;
                page_cache_release(new_page);
#ifdef CONFIG_NUMA
                /*
                 * Find the next applicable VMA for the NUMA policy.
                 */
                addr += PAGE_SIZE;
                if (addr == 0)
                        vma = NULL;
                if (vma) {
                        if (addr >= vma->vm_end) {
                                vma = next_vma;
                                next_vma = vma ? vma->vm_next : NULL;
                        }
                        if (vma && addr < vma->vm_start)
                                vma = NULL;
                } else {
                        if (next_vma && addr >= next_vma->vm_start) {
                                vma = next_vma;
                                next_vma = vma->vm_next;
                        }
                }
#endif
        }
        lru_add_drain();                /* Push any new pages onto the LRU now */
}

/*
 * We hold the mm semaphore and the page_table_lock on entry and
 * should release the pagetable lock on exit..
 */
static int do_swap_page(struct mm_struct * mm,
        struct vm_area_struct * vma, unsigned long address,
        pte_t *page_table, pmd_t *pmd, pte_t orig_pte, int write_access)
{
        struct page *page;
        swp_entry_t entry = pte_to_swp_entry(orig_pte);
        pte_t pte;
        int ret = VM_FAULT_MINOR;

        pte_unmap(page_table);
        spin_unlock(&mm->page_table_lock);
        page = lookup_swap_cache(entry);
        if (!page) {
                swapin_readahead(entry, address, vma);
```

```
		page = read_swap_cache_async(entry, vma, address);
		if (!page) {
			/*
			 * Back out if somebody else faulted in this pte while
			 * we released the page table lock.
			 */
			spin_lock(&mm->page_table_lock);
			page_table = pte_offset_map(pmd, address);
			if (likely(pte_same(*page_table, orig_pte)))
				ret = VM_FAULT_OOM;
			else
				ret = VM_FAULT_MINOR;
			pte_unmap(page_table);
			spin_unlock(&mm->page_table_lock);
			goto out;
		}

		/* Had to read the page from swap area: Major fault */
		ret = VM_FAULT_MAJOR;
		inc_page_state(pgmajfault);
		grab_swap_token();
	}

	mark_page_accessed(page);
	lock_page(page);

	/*
	 * Back out if somebody else faulted in this pte while we
	 * released the page table lock.
	 */
	spin_lock(&mm->page_table_lock);
	page_table = pte_offset_map(pmd, address);
	if (unlikely(!pte_same(*page_table, orig_pte))) {
		pte_unmap(page_table);
		spin_unlock(&mm->page_table_lock);
		unlock_page(page);
		page_cache_release(page);
		ret = VM_FAULT_MINOR;
		goto out;
	}

	/* The page isn't present yet, go ahead with the fault. */

	swap_free(entry);
	if (vm_swap_full())
		remove_exclusive_swap_page(page);

	mm->rss++;
	acct_update_integrals();
	update_mem_hiwater();

	pte = mk_pte(page, vma->vm_page_prot);
	if (write_access && can_share_swap_page(page)) {
		pte = maybe_mkwrite(pte_mkdirty(pte), vma);
		write_access = 0;
	}
	unlock_page(page);

	flush_icache_page(vma, page);
	set_pte(page_table, pte);
	page_add_anon_rmap(page, vma, address);

	if (write_access) {
```

```
                        if (do_wp_page(mm, vma, address,
                                        page_table, pmd, pte) == VM_FAULT_OOM)
                                ret = VM_FAULT_OOM;
                        goto out;
                }

                /* No need to invalidate - it was non-present before */
                update_mmu_cache(vma, address, pte);
                pte_unmap(page_table);
                spin_unlock(&mm->page_table_lock);
out:
                return ret;
}

/*
 * We are called with the MM semaphore and page_table_lock
 * spinlock held to protect against concurrent faults in
 * multithreaded programs.
 */
static int
do_anonymous_page(struct mm_struct *mm, struct vm_area_struct *vma,
                        pte_t *page_table, pmd_t *pmd, int write_access,
                        unsigned long addr)
{
                pte_t entry;
                struct page * page = ZERO_PAGE(addr);

                /* Read-only mapping of ZERO_PAGE. */
                entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));

                /* ..except if it's a write access */
                if (write_access) {
                        /* Allocate our own private page. */
                        pte_unmap(page_table);
                        spin_unlock(&mm->page_table_lock);

                        if (unlikely(anon_vma_prepare(vma)))
                                goto no_mem;
                        page = alloc_zeroed_user_highpage(vma, addr);
                        if (!page)
                                goto no_mem;

                        spin_lock(&mm->page_table_lock);
                        page_table = pte_offset_map(pmd, addr);

                        if (!pte_none(*page_table)) {
                                pte_unmap(page_table);
                                page_cache_release(page);
                                spin_unlock(&mm->page_table_lock);
                                goto out;
                        }
                        mm->rss++;
                        acct_update_integrals();
                        update_mem_hiwater();
                        entry = maybe_mkwrite(pte_mkdirty(mk_pte(page,
                                                        vma->vm_page_prot)),
                                        vma);
                        lru_cache_add_active(page);
                        SetPageReferenced(page);
                        page_add_anon_rmap(page, vma, addr);
                }

                set_pte(page_table, entry);
```

```
		pte_unmap(page_table);

		/* No need to invalidate - it was non-present before */
		update_mmu_cache(vma, addr, entry);
		spin_unlock(&mm->page_table_lock);
out:
		return VM_FAULT_MINOR;
no_mem:
		return VM_FAULT_OOM;
}

/*
 * do_no_page() tries to create a new page mapping. It aggressively
 * tries to share with existing pages, but makes a separate copy if
 * the "write_access" parameter is true in order to avoid the next
 * page fault.
 *
 * As this is called only for pages that do not currently exist, we
 * do not need to flush old virtual caches or the TLB.
 *
 * This is called with the MM semaphore held and the page table
 * spinlock held. Exit with the spinlock released.
 */
static int
do_no_page(struct mm_struct *mm, struct vm_area_struct *vma,
		unsigned long address, int write_access, pte_t *page_table, pmd_t *pmd)
{
		struct page * new_page;
		struct address_space *mapping = NULL;
		pte_t entry;
		unsigned int sequence = 0;
		int ret = VM_FAULT_MINOR;
		int anon = 0;

		if (!vma->vm_ops || !vma->vm_ops->nopage)
				return do_anonymous_page(mm, vma, page_table,
									pmd, write_access, address);
		pte_unmap(page_table);
		spin_unlock(&mm->page_table_lock);

		if (vma->vm_file) {
				mapping = vma->vm_file->f_mapping;
				sequence = mapping->truncate_count;
				smp_rmb(); /* serializes i_size against truncate_count */
		}
retry:
		cond_resched();
		new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, &ret);
		/*
		 * No smp_rmb is needed here as long as there's a full
		 * spin_lock/unlock sequence inside the ->nopage callback
		 * (for the pagecache lookup) that acts as an implicit
		 * smp_mb() and prevents the i_size read to happen
		 * after the next truncate_count read.
		 */

		/* no page was available -- either SIGBUS or OOM */
		if (new_page == NOPAGE_SIGBUS)
				return VM_FAULT_SIGBUS;
		if (new_page == NOPAGE_OOM)
				return VM_FAULT_OOM;

		/*
```

```c
	 * Should we do an early C-O-W break?
	 */
	if (write_access && !(vma->vm_flags & VM_SHARED)) {
		struct page *page;

		if (unlikely(anon_vma_prepare(vma)))
			goto oom;
		page = alloc_page_vma(GFP_HIGHUSER, vma, address);
		if (!page)
			goto oom;
		copy_user_highpage(page, new_page, address);
		page_cache_release(new_page);
		new_page = page;
		anon = 1;
	}

	spin_lock(&mm->page_table_lock);
	/*
	 * For a file-backed vma, someone could have truncated or otherwise
	 * invalidated this page. If unmap_mapping_range got called,
	 * retry getting the page.
	 */
	if (mapping && unlikely(sequence != mapping->truncate_count)) {
		sequence = mapping->truncate_count;
		spin_unlock(&mm->page_table_lock);
		page_cache_release(new_page);
		goto retry;
	}
	page_table = pte_offset_map(pmd, address);

	/*
	 * This silly early PAGE_DIRTY setting removes a race
	 * due to the bad i386 page protection. But it's valid
	 * for other architectures too.
	 *
	 * Note that if write_access is true, we either now have
	 * an exclusive copy of the page, or this is a shared mapping,
	 * so we can make it writable and dirty to avoid having to
	 * handle that later.
	 */
	/* Only go through if we didn't race with anybody else... */
	if (pte_none(*page_table)) {
		if (!PageReserved(new_page))
			++mm->rss;
		acct_update_integrals();
		update_mem_hiwater();

		flush_icache_page(vma, new_page);
		entry = mk_pte(new_page, vma->vm_page_prot);
		if (write_access)
			entry = maybe_mkwrite(pte_mkdirty(entry), vma);
		set_pte(page_table, entry);
		if (anon) {
			lru_cache_add_active(new_page);
			page_add_anon_rmap(new_page, vma, address);
		} else
			page_add_file_rmap(new_page);
		pte_unmap(page_table);
	} else {
		/* One of our sibling threads was faster, back out. */
		pte_unmap(page_table);
		page_cache_release(new_page);
		spin_unlock(&mm->page_table_lock);
```

```
			goto out;
		}

		/* no need to invalidate: a not-present page shouldn't be cached */
		update_mmu_cache(vma, address, entry);
		spin_unlock(&mm->page_table_lock);
out:
		return ret;
oom:
		page_cache_release(new_page);
		ret = VM_FAULT_OOM;
		goto out;
}

/*
 * Fault of a previously existing named mapping. Repopulate the pte
 * from the encoded file_pte if possible. This enables swappable
 * nonlinear vmas.
 */
static int do_file_page(struct mm_struct * mm, struct vm_area_struct * vma,
		unsigned long address, int write_access, pte_t *pte, pmd_t *pmd)
{
		unsigned long pgoff;
		int err;

		BUG_ON(!vma->vm_ops || !vma->vm_ops->nopage);
		/*
		 * Fall back to the linear mapping if the fs does not support
		 * ->populate:
		 */
		if (!vma->vm_ops || !vma->vm_ops->populate ||
					(write_access && !(vma->vm_flags & VM_SHARED))) {
			pte_clear(pte);
			return do_no_page(mm, vma, address, write_access, pte, pmd);
		}

		pgoff = pte_to_pgoff(*pte);

		pte_unmap(pte);
		spin_unlock(&mm->page_table_lock);

		err = vma->vm_ops->populate(vma, address & PAGE_MASK, PAGE_SIZE, vma->vm_page_prot, pgoff, ➟
➥ 0);
		if (err == -ENOMEM)
			return VM_FAULT_OOM;
		if (err)
			return VM_FAULT_SIGBUS;
		return VM_FAULT_MAJOR;
}

/*
 * These routines also need to handle stuff like marking pages dirty
 * and/or accessed for architectures that don't do it in hardware (most
 * RISC architectures). The early dirtying is also good on the i386.
 *
 * There is also a hook called "update_mmu_cache()" that architectures
 * with external mmu caches can use to update those (ie the Sparc or
 * PowerPC hashed page tables that act as extended TLBs).
 *
 * Note the "page_table_lock". It is to protect against kswapd removing
 * pages from under us. Note that kswapd only ever _removes_ pages, never
 * adds them. As such, once we have noticed that the page is not present,
 * we can drop the lock early.
```

```
 *
 * The adding of pages is protected by the MM semaphore (which we hold),
 * so we don't need to worry about a page being suddenly been added into
 * our VM.
 *
 * We enter with the pagetable spinlock held, we are supposed to
 * release it when done.
 */
static inline int handle_pte_fault(struct mm_struct *mm,
	struct vm_area_struct * vma, unsigned long address,
	int write_access, pte_t *pte, pmd_t *pmd)
{
	pte_t entry;

	entry = *pte;
	if (!pte_present(entry)) {
		/*
		 * If it truly wasn't present, we know that kswapd
		 * and the PTE updates will not touch it later. So
		 * drop the lock.
		 */
		if (pte_none(entry))
			return do_no_page(mm, vma, address, write_access, pte, pmd);
		if (pte_file(entry))
			return do_file_page(mm, vma, address, write_access, pte, pmd);
		return do_swap_page(mm, vma, address, pte, pmd, entry, write_access);
	}

	if (write_access) {
		if (!pte_write(entry))
			return do_wp_page(mm, vma, address, pte, pmd, entry);

		entry = pte_mkdirty(entry);
	}
	entry = pte_mkyoung(entry);
	ptep_set_access_flags(vma, address, pte, entry, write_access);
	update_mmu_cache(vma, address, entry);
	pte_unmap(pte);
	spin_unlock(&mm->page_table_lock);
	return VM_FAULT_MINOR;
}

/*
 * By the time we get here, we already hold the mm semaphore
 */
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct * vma,
		unsigned long address, int write_access)
{
	pgd_t *pgd;
	pud_t *pud;
	pmd_t *pmd;
	pte_t *pte;

	__set_current_state(TASK_RUNNING);

	inc_page_state(pgfault);

	if (is_vm_hugetlb_page(vma))
		return VM_FAULT_SIGBUS; /* mapping truncation does this. */

	/*
	 * We need the page table lock to synchronize with kswapd
	 * and the SMP-safe atomic PTE updates.
```

```c
	 */
	pgd = pgd_offset(mm, address);
	spin_lock(&mm->page_table_lock);

	pud = pud_alloc(mm, pgd, address);
	if (!pud)
		goto oom;

	pmd = pmd_alloc(mm, pud, address);
	if (!pmd)
		goto oom;

	pte = pte_alloc_map(mm, pmd, address);
	if (!pte)
		goto oom;

	return handle_pte_fault(mm, vma, address, write_access, pte, pmd);

 oom:
	spin_unlock(&mm->page_table_lock);
	return VM_FAULT_OOM;
}

#ifndef __ARCH_HAS_4LEVEL_HACK
/*
 * Allocate page upper directory.
 *
 * We've already handled the fast-path in-line, and we own the
 * page table lock.
 *
 * On a two-level or three-level page table, this ends up actually being
 * entirely optimized away.
 */
pud_t fastcall *__pud_alloc(struct mm_struct *mm, pgd_t *pgd, unsigned long address)
{
	pud_t *new;

	spin_unlock(&mm->page_table_lock);
	new = pud_alloc_one(mm, address);
	spin_lock(&mm->page_table_lock);
	if (!new)
		return NULL;

	/*
	 * Because we dropped the lock, we should re-check the
	 * entry, as somebody else could have populated it..
	 */
	if (pgd_present(*pgd)) {
		pud_free(new);
		goto out;
	}
	pgd_populate(mm, pgd, new);
 out:
	return pud_offset(pgd, address);
}

/*
 * Allocate page middle directory.
 *
 * We've already handled the fast-path in-line, and we own the
 * page table lock.
 *
 * On a two-level page table, this ends up actually being entirely
```

```
 * optimized away.
 */
pmd_t fastcall *__pmd_alloc(struct mm_struct *mm, pud_t *pud, unsigned long address)
{
	pmd_t *new;

	spin_unlock(&mm->page_table_lock);
	new = pmd_alloc_one(mm, address);
	spin_lock(&mm->page_table_lock);
	if (!new)
		return NULL;

	/*
	 * Because we dropped the lock, we should re-check the
	 * entry, as somebody else could have populated it..
	 */
	if (pud_present(*pud)) {
		pmd_free(new);
		goto out;
	}
	pud_populate(mm, pud, new);
 out:
	return pmd_offset(pud, address);
}
#else
pmd_t fastcall *__pmd_alloc(struct mm_struct *mm, pud_t *pud, unsigned long address)
{
	pmd_t *new;

	spin_unlock(&mm->page_table_lock);
	new = pmd_alloc_one(mm, address);
	spin_lock(&mm->page_table_lock);
	if (!new)
		return NULL;

	/*
	 * Because we dropped the lock, we should re-check the
	 * entry, as somebody else could have populated it..
	 */
	if (pgd_present(*pud)) {
		pmd_free(new);
		goto out;
	}
	pgd_populate(mm, pud, new);
out:
	return pmd_offset(pud, address);
}
#endif

int make_pages_present(unsigned long addr, unsigned long end)
{
	int ret, len, write;
	struct vm_area_struct * vma;

	vma = find_vma(current->mm, addr);
	if (!vma)
		return -1;
	write =	(vma->vm_flags & VM_WRITE) != 0;
	if (addr >= end)
		BUG();
	if (end	> vma->vm_end)
		BUG();
	len = (end+PAGE_SIZE-1)/PAGE_SIZE-addr/PAGE_SIZE;
```

```c
	ret = get_user_pages(current, current->mm, addr,
			len, write, 0, NULL, NULL);
	if (ret < 0)
		return ret;
	return ret == len ? 0 : -1;
}

/*
 * Map a vmalloc()-space virtual address to the physical page.
 */
struct page * vmalloc_to_page(void * vmalloc_addr)
{
	unsigned long addr = (unsigned long) vmalloc_addr;
	struct page *page = NULL;
	pgd_t *pgd = pgd_offset_k(addr);
	pud_t *pud;
	pmd_t *pmd;
	pte_t *ptep, pte;

	if (!pgd_none(*pgd)) {
		pud = pud_offset(pgd, addr);
		if (!pud_none(*pud)) {
			pmd = pmd_offset(pud, addr);
			if (!pmd_none(*pmd)) {
				ptep = pte_offset_map(pmd, addr);
				pte = *ptep;
				if (pte_present(pte))
					page = pte_page(pte);
				pte_unmap(ptep);
			}
		}
	}
	return page;
}

EXPORT_SYMBOL(vmalloc_to_page);

/*
 * Map a vmalloc()-space virtual address to the physical page frame number.
 */
unsigned long vmalloc_to_pfn(void * vmalloc_addr)
{
	return page_to_pfn(vmalloc_to_page(vmalloc_addr));
}

EXPORT_SYMBOL(vmalloc_to_pfn);

/*
 * update_mem_hiwater
 *	- update per process rss and vm high water data
 */
void update_mem_hiwater(void)
{
	struct task_struct *tsk = current;

	if (tsk->mm) {
		if (tsk->mm->hiwater_rss < tsk->mm->rss)
			tsk->mm->hiwater_rss = tsk->mm->rss;
		if (tsk->mm->hiwater_vm < tsk->mm->total_vm)
			tsk->mm->hiwater_vm = tsk->mm->total_vm;
	}
}
```

```
#if !defined(__HAVE_ARCH_GATE_AREA)

#if defined(AT_SYSINFO_EHDR)
struct vm_area_struct gate_vma;

static int __init gate_vma_init(void)
{
	gate_vma.vm_mm = NULL;
	gate_vma.vm_start = FIXADDR_USER_START;
	gate_vma.vm_end = FIXADDR_USER_END;
	gate_vma.vm_page_prot = PAGE_READONLY;
	gate_vma.vm_flags = 0;
	return 0;
}
__initcall(gate_vma_init);
#endif

struct vm_area_struct *get_gate_vma(struct task_struct *tsk)
{
#ifdef AT_SYSINFO_EHDR
	return &gate_vma;
#else
	return NULL;
#endif
}

int in_gate_area_no_task(unsigned long addr)
{
#ifdef AT_SYSINFO_EHDR
	if ((addr >= FIXADDR_USER_START) && (addr < FIXADDR_USER_END))
		return 1;
#endif
	return 0;
}

#endif /* __HAVE_ARCH_GATE_AREA */
```