```
/*
 * This routine handles page faults.   It determines the address,
 * and the problem, and then passes it off to one of the appropriate
 * routines.
 *
 * error_code:
 *   bit 0 == 0 means no page found, 1 means protection fault
 *   bit 1 == 0 means read, 1 means write
 *   bit 2 == 0 means kernel, 1 means user-mode
 */
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
	struct task_struct *tsk;
	struct mm_struct *mm;
	struct vm_area_struct * vma;
	unsigned long address;
	unsigned long page;
	unsigned long fixup;
	int write;

	/* get the address */
	__asm__("movl %%cr2,%0":"=r" (address));

	tsk = current;
	mm = tsk->mm;

	/*
	 * If we're in an interrupt or have no user
	 * context, we must not take the fault..
	 */
	if (in_interrupt() || mm == &init_mm)
		goto no_context;

	down(&mm->mmap_sem);

	vma = find_vma(mm, address);
	if (!vma)
		goto bad_area;
	if (vma->vm_start <= address)
		goto good_area;
	if (!(vma->vm_flags & VM_GROWSDOWN))
		goto bad_area;
	if (error_code & 4) {
```

```
		/*
		 * accessing the stack below %esp is always a bug.
		 * The "+ 32" is there due to some instructions (like
		 * pusha) doing post-decrement on the stack and that
		 * doesn't show up until later..
		 */
		if (address + 32 < regs->esp)
			goto bad_area;
	}
	if (expand_stack(vma, address))
		goto bad_area;
/*
 * Ok, we have a good vm_area for this memory access, so
 * we can handle it..
 */
good_area:
	write = 0;
	switch (error_code & 3) {
		default:	/* 3: write, present */
#ifdef TEST_VERIFY_AREA
			if (regs->cs == KERNEL_CS)
				printk("WP fault at %08lx\n", regs->eip);
#endif
			/* fall through */
		case 2:		/* write, not present */
			if (!(vma->vm_flags & VM_WRITE))
				goto bad_area;
			write++;
			break;
		case 1:		/* read, present */
			goto bad_area;
		case 0:		/* read, not present */
			if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
				goto bad_area;
	}

	/*
	 * If for any reason at all we couldn't handle the fault,
	 * make sure we exit gracefully rather than endlessly redo
	 * the fault.
	 */
	if (!handle_mm_fault(tsk, vma, address, write))
		goto do_sigbus;
```

```
	/*
	 * Did it hit the DOS screen memory VA from vm86 mode?
	 */
	if (regs->eflags & VM_MASK) {
		unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
		if (bit < 32)
			tsk->tss.screen_bitmap |= 1 << bit;
	}
	up(&mm->mmap_sem);
	return;

/*
 * Something tried to access memory that isn't in our memory map..
 * Fix it, but check if it's kernel or user first..
 */
bad_area:
	up(&mm->mmap_sem);

	/* User mode accesses just cause a SIGSEGV */
	if (error_code & 4) {
		tsk->tss.cr2 = address;
		tsk->tss.error_code = error_code;
		tsk->tss.trap_no = 14;
		force_sig(SIGSEGV, tsk);
		return;
	}
	/*
	 * Pentium F0 0F C7 C8 bug workaround.
	 */
	if (boot_cpu_data.f00f_bug) {
		unsigned long nr;
		nr = (address - idt) >> 3;
		if (nr == 6) {
			do_invalid_op(regs, 0);
			return;
		}
	}
no_context:
	/* Are we prepared to handle this kernel fault?	*/
	if ((fixup = search_exception_table(regs->eip)) != 0) {
		regs->eip = fixup;
		return;
	}
```

```
/*
 * Oops. The kernel tried to access some bad page. We'll have to
 * terminate things with extreme prejudice.
 *
 * First we check if it was the bootup rw-test, though..
 */
	if (boot_cpu_data.wp_works_ok < 0 &&
	    address == PAGE_OFFSET && (error_code & 1)) {
		boot_cpu_data.wp_works_ok = 1;
		pg0[0] = pte_val(mk_pte(PAGE_OFFSET, PAGE_KERNEL));
		local_flush_tlb();
		/*
		 * Beware: Black magic here. The printk is needed here to flush
		 * CPU state on certain buggy processors.
		 */
		printk("Ok");
		return;
	}

	if (address < PAGE_SIZE)
		printk(KERN_ALERT "Unable to handle kernel NULL pointer dereference");
	else
		printk(KERN_ALERT "Unable to handle kernel paging request");
	printk(" at virtual address %08lx\n",address);
	__asm__("movl %%cr3,%0" : "=r" (page));
	printk(KERN_ALERT "current->tss.cr3 = %08lx, %%cr3 = %08lx\n",
		tsk->tss.cr3, page);
	page = ((unsigned long *) __va(page))[address >> 22];
	printk(KERN_ALERT "*pde = %08lx\n", page);
	if (page & 1) {
		page &= PAGE_MASK;
		address &= 0x003ff000;
		page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];
		printk(KERN_ALERT "*pte = %08lx\n", page);
	}
	die("Oops", regs, error_code);
	do_exit(SIGKILL);

/*
 * We ran out of memory, or some other thing happened to us that made
 * us unable to handle the page fault gracefully.
 */
do_sigbus:
	up(&mm->mmap_sem);
```

```
        /*
         * Send a sigbus, regardless of whether we were in kernel
         * or user mode.
         */
        tsk->tss.cr2 = address;
        tsk->tss.error_code = error_code;
        tsk->tss.trap_no = 14;
        force_sig(SIGBUS, tsk);

        /* Kernel mode? Handle exceptions or die */
        if (!(error_code & 4))
                goto no_context;
}


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

        pgd = pgd_offset(vma->vm_mm, address);
        pmd = pmd_alloc(pgd, address);
        if (pmd) {
                pte_t * pte = pte_alloc(pmd, address);
                if (pte) {
                        if (handle_pte_fault(tsk, vma, address, write_access, pte)) {
                                update_mmu_cache(vma, address, *pte);
                                return 1;
                        }
                }
        }
        return 0;
}
```

```
static inline int handle_pte_fault(struct task_struct *tsk,
	struct vm_area_struct * vma, unsigned long address,
	int write_access, pte_t * pte)
{
	pte_t entry;

	lock_kernel();
	entry = *pte;

	if (!pte_present(entry)) {
		if (pte_none(entry))
			return do_no_page(tsk, vma, address, write_access, pte);

		return do_swap_page(tsk, vma, address, pte, entry, write_access);
	}

	entry = pte_mkyoung(entry);
	set_pte(pte, entry);
	flush_tlb_page(vma, address);
	if (write_access) {
		if (!pte_write(entry))
			return do_wp_page(tsk, vma, address, pte);

		entry = pte_mkdirty(entry);
		set_pte(pte, entry);
		flush_tlb_page(vma, address);
	}
	unlock_kernel();
	return 1;
}
```

```
/*
 * 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 and the kernel lock held.
 * We need to release the kernel lock as soon as possible..
 */
static int do_no_page(struct task_struct * tsk, struct vm_area_struct * vma,
	unsigned long address, int write_access, pte_t *page_table)
{
	unsigned long page;
	pte_t entry;

	if (!vma->vm_ops || !vma->vm_ops->nopage) {
		unlock_kernel();
		return do_anonymous_page(tsk, vma, page_table, write_access);
	}

	/*
	 * The third argument is "no_share", which tells the low-level code
	 * to copy, not share the page even if sharing is possible.    It's
	 * essentially an early COW detection.
	 */
	page = vma->vm_ops->nopage(vma, address & PAGE_MASK,
		(vma->vm_flags & VM_SHARED)?0:write_access);

	unlock_kernel();
	if (!page)
		return 0;

	++tsk->maj_flt;
	++vma->vm_mm->rss;
	/*
	 * 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.
	 */
	flush_page_to_ram(page);
	entry = mk_pte(page, vma->vm_page_prot);
	if (write_access) {
		entry = pte_mkwrite(pte_mkdirty(entry));
	} else if (atomic_read(&mem_map[MAP_NR(page)].count) > 1 &&
		   !(vma->vm_flags & VM_SHARED))
		entry = pte_wrprotect(entry);
	put_page(page_table, entry);
	/* no need to invalidate: a not-present page shouldn't be cached */
	return 1;
}


/*
 * This only needs the MM semaphore
 */
static int do_anonymous_page(struct task_struct *tsk, struct vm_area_struct *vma, pte_t *page_table, int write_access)
{
	pte_t entry = pte_wrprotect(mk_pte(ZERO_PAGE, vma->vm_page_prot));
	if (write_access) {
		unsigned long page = __get_free_page(GFP_USER);
		if (!page)
			return 0;
		clear_page(page);
		entry = pte_mkwrite(pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
		vma->vm_mm->rss++;
		tsk->min_flt++;
		flush_page_to_ram(page);
	}
	put_page(page_table, entry);
	return 1;
}
```

```
static int do_wp_page(struct task_struct * tsk, struct vm_area_struct * vma,
	unsigned long address, pte_t *page_table)
{
	pte_t pte;
	unsigned long old_page, new_page;
	struct page * page_map;

	pte = *page_table;
	new_page = __get_free_page(GFP_USER);
	/* Did someone else copy this page for us while we slept? */
	if (pte_val(*page_table) != pte_val(pte))
		goto end_wp_page;
	if (!pte_present(pte))
		goto end_wp_page;
	if (pte_write(pte))
		goto end_wp_page;
	old_page = pte_page(pte);
	if (MAP_NR(old_page) >= max_mapnr)
		goto bad_wp_page;
	tsk->min_flt++;
	page_map = mem_map + MAP_NR(old_page);

	/*
	 * We can avoid the copy if:
	 * - we're the only user (count == 1)
	 * - the only other user is the swap cache,
	 *   and the only swap cache user is itself,
	 *   in which case we can remove the page
	 *   from the swap cache.
	 */

	switch (atomic_read(&page_map->count)) {

	case 2:
		if (!PageSwapCache(page_map))
			break;
		if (swap_count(page_map->offset) != 1)
			break;
		delete_from_swap_cache(page_map);
		/* FallThrough */
```

```
	case 1:
		/* We can release the kernel lock now.. */
		unlock_kernel();

		flush_cache_page(vma, address);
		set_pte(page_table, pte_mkdirty(pte_mkwrite(pte)));
		flush_tlb_page(vma, address);
end_wp_page:
		if (new_page)
			free_page(new_page);
		return 1;
	}

	unlock_kernel();
	if (!new_page)
		return 0;

	if (PageReserved(mem_map + MAP_NR(old_page)))
		++vma->vm_mm->rss;
	copy_cow_page(old_page,new_page);
	flush_page_to_ram(old_page);
	flush_page_to_ram(new_page);
	flush_cache_page(vma, address);
	set_pte(page_table,pte_mkwrite(pte_mkdirty(mk_pte(new_page,vma-
	>vm_page_prot))));
	free_page(old_page);
	flush_tlb_page(vma, address);
	return 1;

bad_wp_page:
	printk("do_wp_page: bogus page at address %08lx (%08lx)\n",address,old_page);
	send_sig(SIGKILL, tsk, 1);
	if (new_page)
		free_page(new_page);
	return 0;
}
```

```
/*
 * This is called with the kernel lock held, we need
 * to return without it.
 */
static int do_swap_page(struct task_struct * tsk,
	struct vm_area_struct * vma, unsigned long address,
	pte_t * page_table, pte_t entry, int write_access)
{
	if (!vma->vm_ops || !vma->vm_ops->swapin) {
		swap_in(tsk, vma, page_table, pte_val(entry), write_access);
		flush_page_to_ram(pte_page(*page_table));
	} else {
		pte_t page = vma->vm_ops->swapin(vma, address - vma->vm_start +
vma->vm_offset, pte_val(entry));
		if (pte_val(*page_table) != pte_val(entry)) {
			free_page(pte_page(page));
		} else {
			if (atomic_read(&mem_map[MAP_NR(pte_page(page))].count) > 1 &&
			    !(vma->vm_flags & VM_SHARED))
				page = pte_wrprotect(page);
			++vma->vm_mm->rss;
			++tsk->maj_flt;
			flush_page_to_ram(pte_page(page));
			set_pte(page_table, page);
		}
	}
	unlock_kernel();
	return 1;
}
```

```
/*
 * The tests may look silly, but it essentially makes sure that
 * no other process did a swap-in on us just as we were waiting.
 *
 * Also, don't bother to add to the swap cache if this page-in
 * was due to a write access.
 */
```

```
void swap_in(struct task_struct * tsk, struct vm_area_struct * vma,
        pte_t * page_table, unsigned long entry, int write_access)
{
        unsigned long page;
        struct page *page_map = lookup_swap_cache(entry);

        if (!page_map) {
                swapin_readahead(entry);
                page_map = read_swap_cache(entry);
        }
        if (pte_val(*page_table) != entry) {
                if (page_map)
                        free_page_and_swap_cache(page_address(page_map));
                return;
        }
        if (!page_map) {
                set_pte(page_table, BAD_PAGE);
                swap_free(entry);
                oom(tsk);
                return;
        }

        page = page_address(page_map);
        vma->vm_mm->rss++;
        tsk->min_flt++;
        swap_free(entry);

        if (!write_access || is_page_shared(page_map)) {
                set_pte(page_table, mk_pte(page, vma->vm_page_prot));
                return;
        }
        /*
         * The page is unshared and we're going to dirty it - so tear
         * down the swap cache and give exclusive access to the page to
         * this process.
         */
    delete_from_swap_cache(page_map);
    set_pte(page_table,pte_mkwrite(pte_mkdirty(mk_pte(page,vma->vm_page_prot))));
        return;
}
```

```
struct page * lookup_swap_cache(unsigned long entry)
{
        struct page *found;

        while (1) {
                found = find_page(&swapper_inode, entry);
                if (!found)
                        return 0;
                if (found->inode != &swapper_inode || !PageSwapCache(found))
                        goto out_bad;
                if (!PageLocked(found)) {
                        return found;
                }
                __free_page(found);
                __wait_on_page(found);
        }

out_bad:
        printk (KERN_ERR "VM: Found a non-swapper swap page!\n");
        __free_page(found);
        return 0;
}

static inline struct page *find_page(struct inode * inode, unsigned long offset)
{
                return __find_page(inode, offset, *page_hash(inode, offset));
}


static inline struct page * __find_page(struct inode * inode, unsigned long offset,
struct page *page)
{
                goto inside;
                for (;;) {
                                page = page->next_hash;
inside:
                                if (!page)
                                                goto not_found;
```

```
			if (page->inode != inode)
				continue;
			if (page->offset == offset)
				break;
		}
		/* Found the page. */
		atomic_inc(&page->count);
		set_bit(PG_referenced, &page->flags);
not_found:
		return page;
}
```

void ***swapin_readahead***(unsigned long entry)

```
{

	int i;

	struct page *new_page;

	unsigned long offset = SWP_OFFSET(entry);
	struct swap_info_struct *swapdev = SWP_TYPE(entry) + swap_info;

	offset = (offset >> page_cluster) << page_cluster;

	i = 1 << page_cluster;
	do {
		/* Don't read-ahead past the end of the swap area */
		if (offset >= swapdev->max)
			break;
		/* Don't block on I/O for read-ahead */
		if (atomic_read(&nr_async_pages) >= pager_daemon.swap_cluster)
			break;
		/* Don't read in bad or busy pages */
		if (!swapdev->swap_map[offset])
			break;
		if (swapdev->swap_map[offset] == SWAP_MAP_BAD)
			break;
		if (test_bit(offset, swapdev->swap_lockmap))
			break;

		/* Ok, do the async read-ahead now */
```

```
		new_page = read_swap_cache_async
						(SWP_ENTRY(SWP_TYPE(entry), offset), 0);

		if (new_page != NULL)
			__free_page(new_page);
		offset++;
	} while (--i);
	return;
}


struct page * read_swap_cache_async(unsigned long entry, int wait)
{
	struct page *found_page = 0, *new_page;
	unsigned long new_page_addr;

#ifdef DEBUG_SWAP
	printk("DebugVM: read_swap_cache_async entry %08lx%s\n",
		       entry, wait ? ", wait" : "");
#endif
	/*
	 * Make sure the swap entry is still in use.
	 */
	if (!swap_duplicate(entry))	/* Account for the swap cache */
		goto out;
	/*
	 * Look for the page in the swap cache.
	 */
	found_page = lookup_swap_cache(entry);
	if (found_page)
		goto out_free_swap;


	new_page_addr = __get_free_page(GFP_USER);
	if (!new_page_addr)
		goto out_free_swap;	/* Out of memory */
	new_page = mem_map + MAP_NR(new_page_addr);

	/*
	 * Check the swap cache again, in case we stalled above.
	 */
```

```
	found_page = lookup_swap_cache(entry);
	if (found_page)
		goto out_free_page;
	/*
	 * Add it to the swap cache and read its contents.
	 */
	if (!add_to_swap_cache(new_page, entry))
		goto out_free_page;

	set_bit(PG_locked, &new_page->flags);
	rw_swap_page(READ, entry, (char *) new_page_addr, wait);

	return new_page;

out_free_page:
	__free_page(new_page);
out_free_swap:
	swap_free(entry);
out:
	return found_page;
}
```