```
/*
 * Select the task with maximal swap_cnt and try to swap out a page.
 * N.B. This function returns only 0 or 1.  Return values != 1 from
 * the lower level routines result in continued processing.
 */
```

static int ***swap_out***(unsigned int priority, int gfp_mask)

```
{
    struct task_struct * p, * pbest;
    int counter, assign, max_cnt;

    /*
     * We make one or two passes through the task list, indexed by
     * assign = {0, 1}:
     *    Pass 1: select the swappable task with maximal RSS that has
     *          not yet been swapped out.
     *    Pass 2: re-assign rss swap_cnt values, then select as above.
     *
     * With this approach, there's no need to remember the last task
     * swapped out.  If the swap-out fails, we clear swap_cnt so the
     * task won't be selected again until all others have been tried.
     *
     * Think of swap_cnt as a "shadow rss" - it tells us which process
     * we want to page out (always try largest first).
     */
    counter = nr_tasks / (priority+1);
    if (counter < 1)
        counter = 1;
    if (counter > nr_tasks)
        counter = nr_tasks;

    for (; counter >= 0; counter--) {
        assign = 0;
        max_cnt = 0;
        pbest = NULL;
    select:
        read_lock(&tasklist_lock);
        p = init_task.next_task;
        for (; p != &init_task; p = p->next_task) {
            if (!p->swappable)
                continue;
            if (p->mm->rss <= 0)
                continue;
            /* Refresh swap_cnt? */
```

```
                if (assign)
                        p->mm->swap_cnt = p->mm->rss;
                if (p->mm->swap_cnt > max_cnt) {
                        max_cnt = p->mm->swap_cnt;
                        pbest = p;
                }
        }
        read_unlock(&tasklist_lock);
        if (!pbest) {
                if (!assign) {
                        assign = 1;
                        goto select;
                }
                goto out;
        }

        if (swap_out_process(pbest, gfp_mask))
                return 1;
    }
out:
    return 0;
}


static int swap_out_process(struct task_struct * p, int gfp_mask)
{
    unsigned long address;
    struct vm_area_struct* vma;

    /*
     * Go through process' page directory.
     */
    address = p->mm->swap_address;

    /*
     * Find the proper vm-area
     */
    vma = find_vma(p->mm, address);
    if (vma) {
        if (address < vma->vm_start)
                address = vma->vm_start;

        for (;;) {
                int result = swap_out_vma(p, vma, address, gfp_mask);
```

```
                if (result)
                        return result;
                vma = vma->vm_next;
                if (!vma)
                        break;
                address = vma->vm_start;
        }
    }

    /* We didn't find anything for the process */
    p->mm->swap_cnt = 0;
    p->mm->swap_address = 0;
    return 0;
}
```

```
static int swap_out_vma(struct task_struct * tsk, struct vm_area_struct * vma,
    unsigned long address, int gfp_mask)
{
    pgd_t *pgdir;
    unsigned long end;

    /* Don't swap out areas which are locked down */
    if (vma->vm_flags & VM_LOCKED)
        return 0;

    pgdir = pgd_offset(tsk->mm, address);

    end = vma->vm_end;
    while (address < end) {
        int result = swap_out_pgd(tsk, vma, pgdir, address, end, gfp_mask);
        if (result)
            return result;
        address = (address + PGDIR_SIZE) & PGDIR_MASK;
        pgdir++;
    }
    return 0;
}
```

static inline int ***swap_out_pgd***(struct task_struct * tsk, struct vm_area_struct * vma, pgd_t *dir, unsigned long address, unsigned long end, int gfp_mask)
{
	pmd_t * pmd;
	unsigned long pgd_end;

	if (pgd_none(*dir))
		return 0;
	if (pgd_bad(*dir)) {
		printk("swap_out_pgd: bad pgd (%08lx)\n", pgd_val(*dir));
		pgd_clear(dir);
		return 0;
	}

	pmd = pmd_offset(dir, address);

	pgd_end = (address + PGDIR_SIZE) & PGDIR_MASK;
	if (end > pgd_end)
		end = pgd_end;

	do {
		int result = ***swap_out_pmd***(tsk, vma, pmd, address, end, gfp_mask);
		if (result)
			return result;
		address = (address + PMD_SIZE) & PMD_MASK;
		pmd++;
	} while (address < end);
	return 0;
}

static inline int ***swap_out_pmd***(struct task_struct * tsk, struct vm_area_struct * vma, pmd_t *dir, unsigned long address, unsigned long end, int gfp_mask)
{
	pte_t * pte;
	unsigned long pmd_end;

	if (pmd_none(*dir))
		return 0;
	if (pmd_bad(*dir)) {

```
            printk("swap_out_pmd: bad pmd (%08lx)\n", pmd_val(*dir));
            pmd_clear(dir);
            return 0;
        }

        pte = pte_offset(dir, address);

        pmd_end = (address + PMD_SIZE) & PMD_MASK;
        if (end > pmd_end)
            end = pmd_end;

        do {
            int result;
            tsk->mm->swap_address = address + PAGE_SIZE;
            result = try_to_swap_out(tsk, vma, address, pte, gfp_mask);
            if (result)
                return result;
            address += PAGE_SIZE;
            pte++;
        } while (address < end);
        return 0;
}


/*
 * The swap-out functions return 1 if they successfully
 * threw something out, and we got a free page. It returns
 * zero if it couldn't do anything, and any other value
 * indicates it decreased rss, but the page was shared.
 *
 * NOTE! If it sleeps, it *must* return 1 to make sure we
 * don't continue with the swap-out. Otherwise we may be
 * using a process that no longer actually exists (it might
 * have died while we slept).
 */
static int try_to_swap_out(struct task_struct * tsk, struct vm_area_struct*
vma, unsigned long address, pte_t * page_table, int gfp_mask)
{
        pte_t pte;
        unsigned long entry;
        unsigned long page;
        struct page * page_map;
```

```
	pte = *page_table;
	if (!pte_present(pte))
		return 0;
	page = pte_page(pte);
	if (MAP_NR(page) >= max_mapnr)
		return 0;
	page_map = mem_map + MAP_NR(page);

	if (pte_young(pte)) {
		/*
		 * Transfer the "accessed" bit from the page
		 * tables to the global page map.
		 */
		set_pte(page_table, pte_mkold(pte));
		set_bit(PG_referenced, &page_map->flags);
		return 0;
	}

	if (PageReserved(page_map)
	    || PageLocked(page_map)
	    || ((gfp_mask & __GFP_DMA) && !PageDMA(page_map)))
		return 0;

	/*
	 * Is the page already in the swap cache? If so, then
	 * we can just drop our reference to it without doing
	 * any IO - it's already up-to-date on disk.
	 *
	 * Return 0, as we didn't actually free any real
	 * memory, and we should just continue our scan.
	 */
	if (PageSwapCache(page_map)) {
		entry = page_map->offset;
		swap_duplicate(entry);
		set_pte(page_table, __pte(entry));
drop_pte:
		vma->vm_mm->rss--;
		flush_tlb_page(vma, address);
		__free_page(page_map);
		return 0;
	}

	/*
```

```
 * Is it a clean page? Then it must be recoverable
 * by just paging it in again, and we can just drop
 * it..
 *
 * However, this won't actually free any real
 * memory, as the page will just be in the page cache
 * somewhere, and as such we should just continue
 * our scan.
 *
 * Basically, this just makes it possible for us to do
 * some real work in the future in "shrink_mmap()".
 */
if (!pte_dirty(pte)) {
	pte_clear(page_table);
	goto drop_pte;
}

/*
 * Don't go down into the swap-out stuff if
 * we cannot do I/O! Avoid recursing on FS
 * locks etc.
 */
if (!(gfp_mask & __GFP_IO))
	return 0;

/*
 * Ok, it's really dirty. That means that
 * we should either create a new swap cache
 * entry for it, or we should write it back
 * to its own backing store.
 *
 * Note that in neither case do we actually
 * know that we make a page available, but
 * as we potentially sleep we can no longer
 * continue scanning, so we migth as well
 * assume we free'd something.
 *
 * NOTE NOTE NOTE! This should just set a
 * dirty bit in page_map, and just drop the
 * pte. All the hard work would be done by
 * shrink_mmap().
 *
 * That would get rid of a lot of problems.
 */
```

```
        flush_cache_page(vma, address);
        if (vma->vm_ops && vma->vm_ops->swapout) {
                pid_t pid = tsk->pid;
                pte_clear(page_table);
                flush_tlb_page(vma, address);
                vma->vm_mm->rss--;

                if (vma->vm_ops->swapout(vma, page_map))
                        kill_proc(pid, SIGBUS, 1);
                __free_page(page_map);
                return 1;
        }

        /*
         * This is a dirty, swappable page.  First of all,
         * get a suitable swap entry for it, and make sure
         * we have the swap cache set up to associate the
         * page with that swap entry.
         */
        entry = get_swap_page();
        if (!entry)
                return 0; /* No swap space left */

        vma->vm_mm->rss--;
        tsk->nswap++;
        set_pte(page_table, __pte(entry));
        flush_tlb_page(vma, address);
        swap_duplicate(entry);     /* One for the process, one for the swap cache */
        add_to_swap_cache(page_map, entry);
        /* We checked we were unlocked way up above, and we
            have been careful not to stall until here */
        set_bit(PG_locked, &page_map->flags);

        /* OK, do a physical asynchronous write to swap.  */
        rw_swap_page(WRITE, entry, (char *) page, 0);

        __free_page(page_map);
        return 1;
}
```