```
/*
 * The background pageout daemon, started as a kernel thread
 * from the init process.
 *
 * This basically executes once a second, trickling out pages
 * so that we have _some_ free memory available even if there
 * is no other activity that frees anything up. This is needed
 * for things like routing etc, where we otherwise might have
 * all activity going on in asynchronous contexts that cannot
 * page things out.
 *
 * If there are applications that are active memory-allocators
 * (most normal use), this basically shouldn't matter.
 */
int kswapd(void *unused)
{
        struct task_struct *tsk = current;

        kswapd_process = tsk;
        tsk->session = 1;
        tsk->pgrp = 1;
        strcpy(tsk->comm, "kswapd");
        sigfillset(&tsk->blocked);

        /*
         * Tell the memory management that we're a "memory allocator",
         * and that if we need more memory we should get access to it
         * regardless (see "__get_free_pages()"). "kswapd" should
         * never get caught in the normal page freeing logic.
         *
         * (Kswapd normally doesn't need memory anyway, but sometimes
         * you need a small amount of memory in order to be able to
         * page out something else, and this flag essentially protects
         * us from recursively trying to free more memory as we're
         * trying to free the first piece of memory in the first place).
         */
        tsk->flags |= PF_MEMALLOC;

        while (1) {
                /*
                 * Wake up once a second to see if we need to make
                 * more memory available.
                 *
```

```
		 * If we actually get into a low-memory situation,
		 * the processes needing more memory will wake us
		 * up on a more timely basis.
		 */
		do {
			if (nr_free_pages >= freepages.high)
				break;

			if (!do_try_to_free_pages(GFP_KSWAPD))
				break;
		} while (!tsk->need_resched);
		run_task_queue(&tq_disk);
		tsk->state = TASK_INTERRUPTIBLE;
		schedule_timeout(HZ);
	}
}



/*
 * We need to make the locks finer granularity, but right
 * now we need this so that we can do page allocations
 * without holding the kernel lock etc.
 *
 * We want to try to free "count" pages, and we need to
 * cluster them so that we get good swap-out behaviour. See
 * the "free_memory()" macro for details.
 */
static int do_try_to_free_pages(unsigned int gfp_mask)
{
	int priority;
	int count = SWAP_CLUSTER_MAX;

	lock_kernel();

	/* Always trim SLAB caches when memory gets low. */
	kmem_cache_reap(gfp_mask);

	priority = 6;
	do {
		while (shrink_mmap(priority, gfp_mask)) {
```

```
                        if (!--count)
                                goto done;
                }

                /* Try to get rid of some shared memory pages.. */
                if (gfp_mask & __GFP_IO) {
                        while (shm_swap(priority, gfp_mask)) {
                                if (!--count)
                                        goto done;
                        }
                }

                /* Then, try to page stuff out.. */
                while (swap_out(priority, gfp_mask)) {
                        if (!--count)
                                goto done;
                }

                shrink_dcache_memory(priority, gfp_mask);
        } while (--priority >= 0);
done:
        unlock_kernel();

        return priority >= 0;
}




int shrink_mmap(int priority, int gfp_mask)
{
        static unsigned long clock = 0;
        unsigned long limit = num_physpages;
        struct page * page;
        int count;

        count = limit >> priority;

        page = mem_map + clock;
```

```
do {
        int referenced;

        /* This works even in the presence of PageSkip because
         * the first two entries at the beginning of a hole will
         * be marked, not just the first.
         */
        page++;
        clock++;
        if (clock >= max_mapnr) {
                clock = 0;
                page = mem_map;
        }
        if (PageSkip(page)) {
                /* next_hash is overloaded for PageSkip */
                page = page->next_hash;
                clock = page - mem_map;
        }

        referenced = test_and_clear_bit(PG_referenced, &page->flags);

        if (PageLocked(page))
                continue;

        if ((gfp_mask & __GFP_DMA) && !PageDMA(page))
                continue;

        /* We can't free pages unless there's just one user */
        if (atomic_read(&page->count) != 1)
                continue;

        count--;

        /*
         * Is it a page swap page? If so, we want to
         * drop it if it is no longer used, even if it
         * were to be marked referenced..
         */
        if (PageSwapCache(page)) {
                if (referenced && swap_count(page->offset) != 1)
                        continue;
                delete_from_swap_cache(page);
                return 1;
        }

        if (referenced)
            continue;

        /* Is it a buffer page? */
        if (page->buffers) {
            if (buffer_under_min())
                continue;
            if (!try_to_free_buffers(page))
                continue;
            return 1;
        }

        /* is it a page-cache page? */
        if (page->inode) {
            if (pgcache_under_min())
                continue;
            remove_inode_page(page);
            return 1;
        }

    } while (count > 0);
    return 0;
}

/*
 * This must be called only on pages that have
 * been verified to be in the swap cache.
 */
void delete_from_swap_cache(struct page *page)
{
    long entry = page->offset;
    remove_from_swap_cache (page);
    swap_free (entry);
}


static inline void remove_from_swap_cache(struct page *page)
{
    if (!page->inode) {
```

```
                printk ("VM: Removing swap cache page with zero inode hash "
                        "on page %08lx\n", page_address(page));
                return;
        }
        if (page->inode != &swapper_inode) {
                printk ("VM: Removing swap cache page with wrong inode hash "
                        "on page %08lx\n", page_address(page));
        }
        PageClearSwapCache (page);
        remove_inode_page(page);
}

void remove_inode_page(struct page *page)
{
                  remove_page_from_hash_queue(page);
                  remove_page_from_inode_queue(page);
                  page_cache_release(page);
}
```

```
int shm_swap (int prio, int gfp_mask)
{
        pte_t page;
        struct shmid_kernel *shp;
        unsigned long swap_nr;
        unsigned long id, idx;
        int loop = 0;
        int counter;

        counter = shm_rss >> prio;
        if (!counter || !(swap_nr = get_swap_page()))
                return 0;

  check_id:
        shp = shm_segs[swap_id];
        if (shp == IPC_UNUSED || shp == IPC_NOID || shp->u.shm_perm.mode &
SHM_LOCKED ) {
                next_id:
```

```
        swap_idx = 0;
        if (++swap_id > max_shmid) {
            swap_id = 0;
            if (loop)
                goto failed;
            loop = 1;
        }
        goto check_id;
    }
    id = swap_id;

 check_table:
    idx = swap_idx++;
    if (idx >= shp->shm_npages)
        goto next_id;

    page = __pte(shp->shm_pages[idx]);
    if (!pte_present(page))
        goto check_table;
    if((gfp_mask&__GFP_DMA)&&
       !PageDMA(&mem_map[MAP_NR(pte_page(page))]))
        goto check_table;
    swap_attempts++;

    if (--counter < 0) { /* failed */
        failed:
        swap_free (swap_nr);
        return 0;
    }
    if (atomic_read(&mem_map[MAP_NR(pte_page(page))].count) != 1)
        goto check_table;
    shp->shm_pages[idx] = swap_nr;
    rw_swap_page_nocache (WRITE, swap_nr, (char *) pte_page(page));
    free_page(pte_page(page));
    swap_successes++;
    shm_swp++;
    shm_rss--;
    return 1;
}
```