JOS-Lab 4: Preemptive Multitasking

Yuning Zhang
5140379063

Part A: Multiprocessor Support and Cooperative Multitasking

Exercise 1
boot_aps() copies the mpentry code to the right address in the memory space and then boot each AP one by one. The booted AP will execute the assembly code in kern/mpentry.S first. It is very similar with the boot assembly code. First, it disables interupts. Second, it initializes some registers and sets the GDT. Third, it gets into the protected mode. Fourth, it sets the cr3 and turns on paging. Finally, it sets its own stack and calls mp_main(). mp_main() will do the initialization left and then sets it own status to tell boot_aps() that its booting is done.
Exercise 2
Modify mem_init_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in our revised inc/memlayout.h. The size of each stack is KSTKSIZE bytes plus KSTKGAP bytes of unmapped guard pages.

```
 for (i = 0; i < NCPU; ++i) {
        uintptr_t kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
        boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE, KSTKSIZE,
                PADDR(percpu_kstacks[i]), PTE_W);
        char *sa = (char *)(kstacktop_i - KSTKSIZE - KSTKGAP);
        while ((uintptr_t)sa < kstacktop_i - KSTKSIZE) {
            pte_t *pte = pgdir_walk(kern_pgdir, sa, 0);
            if (pte) {
                *pte = 0;
            }
            sa = sa + PGSIZE;
        }
 }
```

Exercise 3
Modify the code in trap_init_percpu() (kern/trap.c) to use separated TSS and TSS descriptor for each CPU. Also, set correct stack for each CPU's sysenter.

Exercise 4
In i386_init(), acquire the big kernel lock before waking up APs.
In mp_main(), acquire the lock after initializing the AP, and then call sched_yield() to start running environments on this AP.
In trap(), acquire the lock when trapped from user mode.
In env_run(), release the lock right before switching to user mode.

When enter syscall() acquire the lock and release it when return.

Exercise 4.1

Implement ticket spinlock in kern/spinlock.c.

```
lock:
 unsigned ticket = atomic_return_and_add(&lk->next, 1);
 while (atomic_return_and_add(&lk->own, 0) != ticket)
    asm volatile ("pause");

unlock:
 atomic_return_and_add(&lk->own, 1);
```

Ticket spinlock has poor performance because to achieve fairness, it must grant the lock according to the order of getting the ticket. Therefore, on average the CPU will spin for a longer time.

Exercise 5

in sched_yield(), implement simple round-robin scheduling. Search through 'envs' for an ENV_RUNNABLE environment in circular fashion starting just after the env this CPU was last running. Switch to the first such environment found. If no envs are runnable, but the environment previously running on this CPU is still ENV_RUNNING, it's okay to choose that environment.

When scheduling a new environment using env_run_tf(), the trapframe is needed. Therefore, we should modify the sysenter handler to construct a trapframe manually.

Exercise 6

sys_exofork(): Create the new environment with env_alloc(), set status to ENV_NOT_RUNNABLE, copy the register set from the current environment but set the eax to 0.

sys_env_set_status(): Use the envid2env() to translate an envid to a struct Env and set the status.

sys_page_alloc(): Use page_alloc() to allocate a page and then insert that page into the Env’s pgdir using page_insert(). Most code is just for checking.

sys_page_map(): Use page_lookup() to get the page from source env and then insert that page into the dstenv’s pgdir using page_insert(). Most code is just for checking.

sys_page_unmap(): Unmap a page using page_remove().
Because the sys_exofork() in lib.h uses int T_SYSCALL to call the system call and sys_page_map() has more than 4 arguments which the sysenter can not handle, I add the T_SYSCALL trap handler to allow user call the system call using this trap. Also, syscall() in lib/syscall.c is modified to use int T_SYSCALL when there are 5 arguments.

Part B: Copy-on-Write Fork

Exercise 7
sys_env_set_pgfault_upcall(): Use the envid2env() to translate an envid to a struct Env and set the env_pgfault_upcall.

Exercise 8
In page_fault_handler() in kern/trap.c, if there exists a page fault upcall in the environment, we should set up a page fault stack frame on the user exception stack. After that, we should modify the trapframe so that when returned the page fault upcall will be called. Finally, use env_run() to return to user.

Exercise 9
In this exercise, we need to complete the _pgfault_upcall routine in lib/pfentry.S. After calling the C page fault handler, we should “push” the trap-time %eip on to the trap-time stack. (Also, the saved trap-time %esp is modified accordingly) This step is the preparation for stack switching and %eip reloading. Then, restore the registers(except %esp) and the eflags. After that, switch the stack by “popl %esp” and reload the %eip by “ret”.

Exercise 10
Finish set_pgfault_handler() which sets the page fault handler function. The first time we register a handler, we need to allocate an exception stack using sys_page_alloc(), and tell the kernel to call the assembly-language _pgfault_upcall routine when a page fault occurs by sys_env_set_pgfault_upcall().

Exercise 11
In pgfault(), first, check that the faulting access was (1) a write, and (2) to a copy-on-write page. If not, panic. Second, allocate a new page, map it at a temporary location (PFTEMP), copy the data from the old page to the new page, then move the new page to the old page's address.

In duppage(), if the page is writable or copy-on-write, first create a copy-on-write mapping for the child. Second, mark the old mapping as copy-on-write. The ordering is quite important, because the sys_page_map()’s return value will be saved into the r variable on the stack. When duppage() the page for the stack, this write will cause a page fault immediately which cancels the sys_page_map()’s mark. For a read-only page, just create a new mapping for the child.

In fork(), first set up page fault handler appropriately. Second, allocate a new child environment. If we are the parent, copy address space with duppage (except the user exception stack). Then allocate the exception stack for child and set pgfault_upcall for child. Finally, set the child environment runnable. If we are the child, fix thisenv and return 0.

Part C: Preemptive Multitasking and Inter-Process communication (IPC)

Exercise 12

In kern/trapentry.S, provide handlers for IRQs 0 through 15. In trap_init(), initialize the appropriate entries in the IDT. In env_alloc(), set FL_IF in eflags to make user environments run with interrupts enabled:

e->env_tf.tf_eflags |= FL_IF;

Because the sysexit instruction does not restore the FL_IF flag, we should call sti instruction to turn on interruption before sysexit.

Exercise 13

Modify the kernel's trap_dispatch() function so that it calls sched_yield() to find and run a different environment whenever a clock interrupt takes place.

```
        // Handle clock interrupts. Don't forget to acknowledge the
        // interrupt using lapic_eoi() before calling the scheduler!
        if (tf->tf_trapno == IRQ_OFFSET + 0) {
    lapic_eoi();
    sched_yield();
                return;
        }
```

Inter-Process communication (IPC)

Exercise 14

sys_ipc_recv() first checks its argument. Then, it records that it wants to receive using env_ipc_recving and env_ipc_dstva. Finally, it marks itself not runnable and calls sched_yield()

sys_ipc_try_send() first checks if the destination Env is receiving, if not, it returns -E_IPC_NOT_RECV. If a page is needed to be sent, it performs some checking and then page_insert() that page into the destination Env's pgdir. Then it fills the destination Env with its information. Finally, it clears destination Env's env_ipc_recving flag, changes destination Env's eax to 0 to make it "return" 0 and marks it as RUNNABLE.

ipc_recv() just calls sys_ipc_recv() and extracts informations from the Env structure. If 'pg' is null, we should pass sys_ipc_recv UTOP which will understand as meaning "no page".

ipc_send() just keeps trying to call sys_ipc_try_send() and sys_yield() when sys_ipc_try_send() returns -E_IPC_NOT_RECV.

Challenge sfork()

(To turn on it, you should uncomment the define CHALLENGE in inc/lib.h)

In sfork(), first set up page fault handler appropriately. Second, allocate a new child environment. If we are the parent, share the address space with child using sys_page_map (except the stacks). Then allocate the exception stack for child and copy-on-write the user stack. Finally, set pgfault_upcall for child and set the child environment runnable.

Because of the sharing, the global thisenv is no longer usable and is set to NULL in sfork(). Every code using it needs to check for that and calculate it if necessary by &envs[ENVX(sys_getenvid())]; In the pingpongs.c, the program maintains a `myenv` variable on the separated stack.