Report [Lab 4: Preemptive Multitasking]

I complete the first challenge after exercise 15, completing an `ipc_send()` with no loops.

Part A: Multiprocessor Support and Cooperative Multitasking

Multiprocessor Support

Exercise 1

`mmio_map_region()` is used in `lapic_init()` to map LAPIC's 4K MMIO region into virtual address space for the convenience of accessing. Because back the pages between MMIOBASE and MMIOLIM are no normal page frames, their content can't be cached and must apply write-through policy. This is done by setting specify permission bits in the PTE by `boot_map_region()`

```
    size = ROUNDUP(size, PGSIZE);
    if (base + size > MMIOLIM)
    {
        panic("mmio_map_region: exceed MMIOLIM");
    }
    boot_map_region(kern_pgdir, base, size, pa, PTE_W | PTE_PCD | PTE_PWT);
    void * re = (void *)base;
    base += size;
    return re;
```

Application Processor Bootstrap

Exercise 2

`boot_aps()` is called from `i386_init()` to awake all application processors.

First it copies the assembly code in `kern/mpentry.S` to the physical address `MPENTRY_PADDR` (a low address to allow the code to run in real mode). Then each AP is given their own kernel stack to execute the assembly code starting at `mpentry_start` by calling `lapic_statap()`. Then it spins to wait for the AP startup before awaking the next.

In `mpentry.S`, things are quite similar to the BSP bootstrap. First protected mode is turned on then GDT is loaded and paging is enabled with `entry_pgdir`. Finally the control flow jumps to `mp_main()`

In `mp_main`, `kernel_pgdir` is loaded and then `lapic_init()`, `env_init_percpu()`, `trap_init_percpu()` is called to initialize local config. Then it set `thiscpu->cpu_status` as `CPU_STARTED` to announce this.

So we change `page_init()` in `pmap.c` to reserve space for AP start code.

```
        if (i == PGNUM(MPENTRY_PADDR)) // reserved for AP start code
            continue;
```

Question 1

`MPBOOTPHYS` is intended to interpret the address of the symbols `mpentry.S` uses e.g. `gdt` from kernel virtual address to the physical address at which the `mpentry.S` actually runs i.e. above `0x7000`

`kern/mpentry.S` and `boot/boot.S` are quite the same in their content but they are linked to different address by the linker. We can observe this using `objdump`, i.e they've got different VMA

```
$ objdump -x obj/kern/kernel
# ...
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00005d21  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
# ...
$ objdump -x obj/boot/boot.o
# ...
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000006a  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
# ...
```

So if we omit using `MPBOOTPHYS` to translate symbols in `kern/mpentry.S`, the code will try to address those symbols using high virtual addresses when we are still in real mode. That's definitely what we expected.

```
$ objdump -x obj/kern/kernel | grep gdt
f0105360 l       .text  00000000 gdtdesc
f0105348 l       .text  00000000 gdt
f0124340 g     O .data  00000068 gdt
f0124320 g     O .data  00000006 gdt_pd
```

Per-CPU State and Initialization

Exercise 3

In the past we only map `bootstack` to back-store BSP's kernel stack. Now we map `percpu_kstacks` to back-store each CPU's kernel stack because they share the same address space while needing to handle interrupts simultaneously.

```
    for (int i = 0; i < NCPU; i++)
    {
        intptr_t kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
        boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE, KSTKSIZE,
                        PADDR(percpu_kstacks[i]), PTE_W);
    }
```

Exercise 4

For each CPU, a TSS and a TSS descriptor is needed to indicate the stack position for stack switch happening at interrupt handling which changes CPL. We set `esp0` and `ss0` field as the kernel stack we preserved for `thiscpu` and modify the entry in GDT to point to the `cpu_ts` structure we initialized.

There is one field in TSS called `ts_iomb`, which points to the beginning of the I/O permission bit map and the end of the interrupt redirection bit map, that are stored in TSS at higher addresses if present. [1] We set it to end of TSS basic structure (which indicates the non-existence of interrupt redirection bit map) with the `limit` field in GDT entry set as `sizeof(struct Taskstate) - 1` (which indicates the non-existence of I/O permission bit map). If `ts_iomb` is set to 0, some range of address might be misinterpreted as I/O permission bit map, which is dangerous.

Then we load the TSS descriptor using `ltr` and the same IDT descriptor as BSP using `lidt`.

```
    int i = cpunum();
    intptr_t kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    thiscpu->cpu_ts.ts_esp0 = kstacktop_i;
    thiscpu->cpu_ts.ts_ss0 = GD_KD;
    thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);

    gdt[(GD_TSS0 >> 3) + i] = SEG16(STS_T32A, (uint32_t) (&thiscpu->cpu_ts),
                                    sizeof(struct Taskstate) - 1, 0);
    gdt[(GD_TSS0 >> 3) + i].sd_s = 0;

    ltr(GD_TSS0 + (i << 3));

    lidt(&idt_pd);

    return;
```

Locking

Exercise 5

Apply the big kernel lock at the indicated positions

```
// i386_init()
    // Acquire the big kernel lock before waking up APs
    // Your code here:
    lock_kernel();
```

```
// mp_main()
    // Now that we have finished some basic setup, call sched_yield()
    // to start running processes on this CPU.  But make sure that
    // only one CPU can enter the scheduler at a time!
    //
    // Your code here:
    lock_kernel();
    sched_yield();
```

```
// trap()
        // Trapped from user mode.
        // Acquire the big kernel lock before doing any
        // serious kernel work.
        // LAB 4: Your code here.
        assert(curenv);
        lock_kernel();
```

```
// env_run()
    if (curenv != NULL)
    {
        if (curenv->env_status == ENV_RUNNING)
        {
            curenv->env_status = ENV_RUNNABLE;
        }
        // we have saved context of curenv at _alltrap
    }
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs++;
    lcr3(PADDR(e->env_pgdir));

    unlock_kernel();

    env_pop_tf(&e->env_tf);
```

Question 2

Each CPU needs its own stack to store the information because it might be blocked during execution in kernel mode. For example if the kernel code in one CPU performs some IO then it's blocked, some other CPU will acquire the big kernel lock and run its kernel code, corrupting the content of the shared stack.

Round-Robin Scheduling

Exercise 6

First we acquire the index of current running environment by `ENVX(curenv->envid)`, or beginning from 0 if `curenv` is NULL. If there is no other environment `RUNNABLE` and `curenv` is `ENV_RUNNING`, we keep it as `curenv`.

```c
    // LAB 4: Your code here.
    for (int i = 0, j = curenv == NULL ? 0 : ENVX(curenv->env_id); i < NENV; i++, j++)
    {
        if (j >= NENV) j -= NENV;
        if (envs[j].env_status == ENV_RUNNABLE)
        {
            env_run(&envs[j]);
        }
    }
    if (curenv != NULL && curenv->env_status == ENV_RUNNING)
    {
        env_run(curenv);
    }
```

Dispatch `sys_yield()` in `syscall()`. Note that it never returns.

```c
    case SYS_yield:
        sys_yield();
```

create 3 environments running `user/yield.c` to test.

```c
    // Touch all you want.
    // ENV_CREATE(user_primes, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
```

```
hiesa@IL:~/workspace/6.828/lab$ make qemu-nox CPUS=2
# ...
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Back in environment 00001000, iteration 0.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001002, iteration 1.
# ...
```

Question 3

The pointer `e` points one element in the kernel data structure `envs`. Since all environments map the kernel part of virtual address space in the same manner, `e` points to the identical data before and after the addressing switch.

Question 4

Because environments are oblivious to context switch, if the old registers are not stored and later recovered, environment will run with corrupted registers, giving out unexpected result.

This is down by two parts. First part is down by x86 hardware, pushing program counter and stack register into the stack; second part is down at `_alltrap` and `trap()`, where all general registers are pushed onto stack forming a `Trapframe` structure and copied into `curenv->env_tf`.

System Calls for Environment Creation

Exercise 7

In this exercise we will implement a number of system calls for user environments.

```
    case SYS_exofork:
        return sys_exofork();
    case SYS_env_set_status:
        return sys_env_set_status(a1, a2);
    case SYS_page_alloc:
        return sys_page_alloc(a1, (void *)a2, a3);
    case SYS_page_map:
        return sys_page_map(a1, (void *)a2, a3, (void *)a4, a5);
    case SYS_page_unmap:
        return sys_page_unmap(a1, (void *)a2);
```

SYS_exofork

This function is to create a new environment which is basically what `env_alloc` creates except that its status is set as `ENV_NOT_RUNNABLE` for setups and its register values are copied from the parent environment.

We return the `env_id` of child environment while tweaking the the value of `%eax` of the child environment to 0 so the two environment involved see different return value.

```
    // LAB 4: Your code here.
    int retval;
    struct Env *child;

    if ((retval = env_alloc(&child, curenv->env_id)))
    {
        return retval;
    }

    child->env_status = ENV_NOT_RUNNABLE;
    child->env_tf = curenv->env_tf;
    child->env_tf.tf_regs.reg_eax = 0;

    return child->env_id;
```

SYS_env_set_status

The main effort in this and following functions are put into checking the validity of the arguments.

First we have to ensure the environment which calls this function i.e. `curenv` has the permission to manipulate the context of the environment indicated by `envid`. This is done in `envid2env()`, as whether the environment indicated by `envid` (if still existing) is `curenv` itself or its direct child.

We also checks whether the status is not `ENV_RUNNABLE` or `ENV_NOT_RUNNABLE`.

```
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
    {
        return -E_INVAL;
    }

    struct Env* e;
    int retval;

    if ((retval = envid2env(envid, &e, true)))
    {
        return retval;
    }

    e->env_status = status;

    return 0;
```

SYS_page_alloc

In this function we allocate a physical page with `page_alloc()` and insert it at the address indicated by `va` using `page_insert()` with `perm` as the permission, after we have checked that the `va` is under `UTOP` and page_aligned and `perm` is valid (present bit and user bit are set while bits other than those set in `PTE_SYSCALL` aren't set).

```
    // LAB 4: Your code here.
    if ((intptr_t)(va) >= UTOP || (intptr_t)(va) % PGSIZE)
    {
        return -E_INVAL;
    }
    if ((~perm & PTE_P) || (~perm & PTE_U) || (perm & ~PTE_SYSCALL))
    {
        return -E_INVAL;
    }

    struct Env *e;
    struct PageInfo *p;
    int retval;

    if ((retval = envid2env(envid, &e, true)))
    {
        return retval;
    }

```

```c
    p = page_alloc(ALLOC_ZERO);
    if (p == NULL)
    {
        return -E_NO_MEM;
    }
    if ((retval = page_insert(e->env_pgdir, p, va, perm)))
    {
        page_free(p);
        return -E_NO_MEM;
    }

    return 0;
```

SYS_page_map

In this function we map the physical page indicated by `srcva` in source environment's address space (which can be queried by `page_lookup()`) then map it at `dstva` in destination environment using `page_insert` with new permission bits `perm`.

Same checks like those above are performed upon both source and destination `va` and `envid`. In addition, we have to ensure that a read-only page in the source can't be mapped at the destination with the write bit set. This is done by checking the PTE in source virtual address space which the `page_lookup()` stored

```c
    // LAB 4: Your code here.
    struct Env *srce, *dste;
    struct PageInfo *p;
    pte_t *pte;
    int retval;

    if ((retval = envid2env(srcenvid, &srce, true)))
    {
        return retval;
    }
    if ((retval = envid2env(dstenvid, &dste, true)))
    {
        return retval;
    }

    if ((intptr_t)(srcva) >= UTOP || (intptr_t)(srcva) % PGSIZE)
    {
        return -E_INVAL;
    }
    if ((intptr_t)(dstva) >= UTOP || (intptr_t)(dstva) % PGSIZE)
    {
        return -E_INVAL;
    }

    p = page_lookup(srce->env_pgdir, srcva, &pte);
    if (p == NULL)
    {
        return -E_INVAL;
    }
    if ((~perm & PTE_P) || (~perm & PTE_U) || (perm & ~PTE_SYSCALL))
    {
```

```
            return -E_INVAL;
        }
        if ((perm & PTE_W) && (~(*pte) & PTE_W))
        {
            return -E_INVAL;
        }

        if ((retval = page_insert(dste->env_pgdir, p, dstva, perm)))
        {
            return retval;
        }

        return 0;
```

SYS_page_unmap

In this function, we check the validity of the arguments and then use `page_remove()` to remove the physical page at `va` in the address space of the environment with `envid`.

If there wasn't any page at `va`, the function silently succeed, like the behavior of `page_remove()` it calls.

```
        // LAB 4: Your code here.
        struct Env *e;
        int retval;

        if ((retval = envid2env(envid, &e, true)))
        {
            return retval;
        }
        if ((intptr_t)(va) >= UTOP || (intptr_t)(va) % PGSIZE)
        {
            return -E_INVAL;
        }

        page_remove(e->env_pgdir, va);

        return 0;
```

Part B: Copy-on-Write Fork

User-level page fault handling

Setting the Page Fault Handler

Exercise 8

We check the validity of `envid` before we set up `env_pgfault_upcall` like what we do in the previous exercise.

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
```

```
    struct Env *e;
    int retval;

    if ((retval = envid2env(envid, &e, true)))
    {
        return retval;
    }

    e->env_pgfault_upcall = func;

    return 0;

    // panic("sys_env_set_pgfault_upcall not implemented");
}
```

```
    case SYS_env_set_pgfault_upcall:
        return sys_env_set_pgfault_upcall(a1, (void *)a2);
```

Invoking the User Page Fault Handler

Exercise 9

If current environment has set up its `env_pgfault_upcall`, we should transfer control to user's registered handler when page fault occurs. To provide information for user defined page fault handler to recover the fault and return to previous instruction causing the fault, we need to push a `UTrapframe` structure onto user exception stack.

There are two cases which can be determined by the value of `%esp` at trap time:

1. The page fault occurs when user environment is running with normal stack. Then we should switch the stack to start just below `UXSTACKTOP`
2. The page fault occurs during the handling of another fault, where we should switch to a new stack below current exception stack position i.e. trap-time `%esp`, leaving a 4-word gap for scratch space. (used by the assembly language stub to help page fault handle return)

Either case, we should make sure there is enough available space for us to store the `UTrapframe structure`

```
    if (curenv->env_pgfault_upcall)
    {
        struct UTrapframe *utf;

        if (tf->tf_esp >= UXSTACKTOP - PGSIZE && tf->tf_esp < UXSTACKTOP)
        {
            // already running on the user exception stack
            utf = (struct UTrapframe*)(tf->tf_esp - 4 - sizeof(struct UTrapframe));
        }
        else
        {
            // running on normal user stack
            utf = (struct UTrapframe*)(UXSTACKTOP - sizeof(struct UTrapframe));
```

```
        }

        // validity check of exception stack
        user_mem_assert(curenv, utf, sizeof(struct UTrapframe), PTE_W);
```

Then we fill in the fields of `UTrapFrame` and change `tf_eip` and `tf_esp` so that the environment will run the registered page fault handler with a new stack by `env_run()`

```
        // fill in utf
        utf->utf_fault_va = fault_va;
        utf->utf_err = tf->tf_err;
        utf->utf_regs = tf->tf_regs;
        utf->utf_eip = tf->tf_eip;
        utf->utf_eflags = tf->tf_eflags;
        utf->utf_esp = tf->tf_esp;

        // run the user page fault handler with new stack
        tf->tf_eip = (intptr_t)curenv->env_pgfault_upcall;
        tf->tf_esp = (uintptr_t)utf;
        env_run(curenv);
```

User-mode Page Fault Entrypoint

Exercise 10

In this exercise we will complete the assembly to return to the instruction which invokes the page fault from page fault handler, which means we have to restore the execution state with much precaution.

First we need to restore the general registers, after which we can no longer use them to store temporary values. Then we need to restore `eflags` using `popfl`, after which we can no longer use arithmetic operations. Finally we have to restore `%esp` and `%eip`, which is the most sophisticate phase.

Since all general registers are restored, we can't use `jmp` since it needs a register to store the address. Neither can we simply use `ret` on exception stack otherwise the process will run with a wrong `%esp`.

So we will do some tricks, pushing the trap time `%eip` we want to return to onto the *trap time* stack and let `%esp` point to it before we use `ret`. That will set `%esp` and `%eip` both in correct place.

```
    // LAB 4: Your code here.
    movl 40(%esp), %eax // trap-time eip
    subl $4, 48(%esp)   // tweak the trap-time stack for later ret
    movl 48(%esp), %edx
    movl %eax, (%edx)  // use 4 bytes below trap-time stack as scratch space for later ret

    // Restore the trap-time registers.  After you do this, you
    // can no longer modify any general-purpose registers.
    // LAB 4: Your code here.
    addl $8, %esp // now point to utf_regs
    popal
```

```

    // Restore eflags from the stack.  After you do this, you can
    // no longer use arithmetic operations or anything else that
    // modifies eflags.
    // LAB 4: Your code here.
    addl $4, %esp // now point to utf_eflags
    popfl

    // Switch back to the adjusted trap-time stack.
    // LAB 4: Your code here.
    movl (%esp), %esp

    // Return to re-execute the instruction that faulted.
    // LAB 4: Your code here.
    ret
```

Exercise 11

In the user library function `set_pgfault_handler`, we use a global variable to store the user provided page fault handler. If it's called the first time, it will allocate a page for user exception stack and register the assembly page fault handler entry point with a system call.

```
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
    int r;

    if (_pgfault_handler == 0) {
        // First time through!
        // LAB 4: Your code here.
        sys_page_alloc(0, (void *)(UXSTACKTOP - PGSIZE), PTE_W | PTE_U | PTE_P);
        sys_env_set_pgfault_upcall(0, _pgfault_upcall);
        // panic("set_pgfault_handler not implemented");
    }

    // Save handler pointer for assembly to call.
    _pgfault_handler = handler;
}
```

Implementing Copy-on-Write Fork

Exercise 12

In this exercise we will exploit the user-level page fault handler we implemented above to complete the function of copy-on-write fork.

The 11th bit of PTE is left available for software so we use it as the COW bit. We implement a user level page fault handler to deal with page fault which is a write access to a COW page.

```c
    // LAB 4: Your code here.
    pte_t *pte = PGADDR(PDX(UVPT), PDX(addr), PTX(addr) << 2);

    if (~err & FEC_WR)
    {
        panic("pgfault: not write access");
    }
    if (~(*pte) & PTE_COW)
    {
        panic("pgfault: not access to copy-on-write page");
    }

```

We use the page at `PFTEMP` which is below `UTEXT` as the scratch space. We allocate a temporary page there and copy the content of the fault page to it. Then we unmap the COW page and map the new private page frame at the fault address.

```c
    // LAB 4: Your code here.
    if ((r = sys_page_alloc(0, (void *)PFTEMP, PTE_P | PTE_U | PTE_W)))
    {
        panic("pgfault: %e\n", r);
    }
    memcpy((void *)PFTEMP, ROUNDDOWN(addr, PGSIZE), PGSIZE);
    if ((r = sys_page_unmap(0, ROUNDDOWN(addr, PGSIZE))))
    {
        panic("pgfault: %e\n", r);
    }
    if ((r = sys_page_map(0, (void *)PFTEMP, 0, ROUNDDOWN(addr, PGSIZE), PTE_P | PTE_U | PTE_W)))
    {
        panic("pgfault: %e\n", r);
    }
```

In `fork()`, we register the COW page fault handler first and use `sys_exofork()` to create the child environment with a virtual address which is current empty below `UTOP`. Then we use `duppage()` to copy all pages existing in current environment under `UTOP` to the child environment expect of the page of user exception stack because if it's marked as copy-on-write, no one would do the work for it.

After that we register the same page fault handler for the child (as now the code of the page fault handler also resides at the same address in child's virtual address space) and allocate a page for its user exception stack much like what we do when we first enter `set_pgfault_handler()`. Then things are done and we mark the child as `ENV_RUNNABLE`

```c
envid_t
fork(void)
{
    // LAB 4: Your code here.
    envid_t child;
    int r;

    set_pgfault_handler(&pgfault);

```

```c
    if ((child = sys_exofork()) < 0)
    {
        panic("fork: %e", child);
    }

    // I'm the child
    if (child == 0)
    {
        thisenv = &envs[ENVX(sys_getenvid())];
        return child;
    }

    for (int i = 0; i <= PDX(USTACKTOP); i++)
    {
        pde_t *pde = PGADDR(PDX(UVPT), PDX(UVPT), (i << 2));
        if (~(*pde) & PTE_P) continue;
        for (int j = 0; j < (PGSIZE >> 2); j++)
        {
            if (i == PDX(USTACKTOP) && j >= PTX(USTACKTOP))
            {
                break;
            }
            pte_t *pte = PGADDR(PDX(UVPT), i, j << 2);
            if (~(*pte) & PTE_P) continue;

            duppage(child, i * (PGSIZE >> 2) + j);
        }
    }

    extern void _pgfault_upcall(void);
    if ((r = sys_page_alloc(child, (void *)(UXSTACKTOP - PGSIZE), PTE_W | PTE_U | PTE_P)))
    {
        panic("fork: %e\n", r);
    }
    if ((r = sys_env_set_pgfault_upcall(child, _pgfault_upcall)))
    {
        panic("fork: %e\n", r);
    }

    if ((r = sys_env_set_status(child, ENV_RUNNABLE)))
    {
        panic("fork: %e\n", r);
    }

    return child;

    // panic("fork not implemented");
}

```

In `duppage()`, we simply maps the read-only pages with the same `perm` to the other environment while for writable or copy-on-write pages, we map the page in the new environment as a COW page and then mark it as COW in current address space.

It's important to mark a page as COW in the child before marking it in the parent because if we mark the page in parent first, there would be a race between writing to the page in the parent and marking it COW in the child. For example if the page stands for the current stack we might invoke the page fault handler when pushing the arguments onto the stack to call the `sys_page_map` so that the page is turned back to a normal writable page before it's marked COW in the child. Then the same page frame will be writable in the parent while COW in the child so the write in the parent would affect what the child will read. What a disaster!

For the same reason we should always mark the page COW in the parent even it's COW at the beginning, as it might turn to a writable page in the procedure.

```
static int
duppage(envid_t envid, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    pte_t *pte = PGADDR(PDX(UVPT), pn / (PGSIZE >> 2), (pn % (PGSIZE >> 2))
<< 2);
    void *addr = (void *)(pn * PGSIZE);

    if ((*pte) & (PTE_W | PTE_COW))
    {
        // writable or copy-on-write page
        int perm = (((*pte) & PTE_SYSCALL)  & (~PTE_W)) | PTE_COW;
        if ((r = sys_page_map(0, addr, envid, addr, perm)))
        {
            panic("duppage: %e\n",  r);
        }
        if ((r = sys_page_map(0, addr, 0, addr, perm)))
        {
            panic("duppage: %e\n", r);
        }
    }
    else
    {
        // read-only page
        int perm = (*pte) & PTE_SYSCALL;
        if ((r = sys_page_map(0, addr, envid, addr, perm)))
        {
            panic("duppage: %e\n", r);
        }
    }
    // panic("duppage not implemented");
    return 0;
}

```

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

Clock Interrupts and Preemption

Interrupt discipline

Exercise 13

> We do the same thing as in previous lab.

```
// trapentry.S
TRAPHANDLER_NOEC(ENTRY_IRQ0, IRQ_OFFSET + 0)
TRAPHANDLER_NOEC(ENTRY_IRQ1, IRQ_OFFSET + 1)
TRAPHANDLER_NOEC(ENTRY_IRQ2, IRQ_OFFSET + 2)
TRAPHANDLER_NOEC(ENTRY_IRQ3, IRQ_OFFSET + 3)
TRAPHANDLER_NOEC(ENTRY_IRQ4, IRQ_OFFSET + 4)
TRAPHANDLER_NOEC(ENTRY_IRQ5, IRQ_OFFSET + 5)
TRAPHANDLER_NOEC(ENTRY_IRQ6, IRQ_OFFSET + 6)
TRAPHANDLER_NOEC(ENTRY_IRQ7, IRQ_OFFSET + 7)
TRAPHANDLER_NOEC(ENTRY_IRQ8, IRQ_OFFSET + 8)
TRAPHANDLER_NOEC(ENTRY_IRQ9, IRQ_OFFSET + 9)
TRAPHANDLER_NOEC(ENTRY_IRQ10, IRQ_OFFSET + 10)
TRAPHANDLER_NOEC(ENTRY_IRQ11, IRQ_OFFSET + 11)
TRAPHANDLER_NOEC(ENTRY_IRQ12, IRQ_OFFSET + 12)
TRAPHANDLER_NOEC(ENTRY_IRQ13, IRQ_OFFSET + 13)
TRAPHANDLER_NOEC(ENTRY_IRQ14, IRQ_OFFSET + 14)
TRAPHANDLER_NOEC(ENTRY_IRQ15, IRQ_OFFSET + 15)
```

```
// trap.c
// entry point for IRQ 0 ~ 15
extern void ENTRY_IRQ0();
extern void ENTRY_IRQ1();
extern void ENTRY_IRQ2();
extern void ENTRY_IRQ3();
extern void ENTRY_IRQ4();
extern void ENTRY_IRQ5();
extern void ENTRY_IRQ6();
extern void ENTRY_IRQ7();
extern void ENTRY_IRQ8();
extern void ENTRY_IRQ9();
extern void ENTRY_IRQ10();
extern void ENTRY_IRQ11();
extern void ENTRY_IRQ12();
extern void ENTRY_IRQ13();
extern void ENTRY_IRQ14();
extern void ENTRY_IRQ15();
```

```
// trap_init() in trap.c
    // set up IDT entries for IRQ 0 ~ 15
    SETGATE(idt[IRQ_OFFSET + 0], 0, GD_KT, &ENTRY_IRQ0, 0);
    SETGATE(idt[IRQ_OFFSET + 1], 0, GD_KT, &ENTRY_IRQ1, 0);
    SETGATE(idt[IRQ_OFFSET + 2], 0, GD_KT, &ENTRY_IRQ2, 0);
```

```
    SETGATE(idt[IRQ_OFFSET + 3], 0, GD_KT, &ENTRY_IRQ3, 0);
    SETGATE(idt[IRQ_OFFSET + 4], 0, GD_KT, &ENTRY_IRQ4, 0);
    SETGATE(idt[IRQ_OFFSET + 5], 0, GD_KT, &ENTRY_IRQ5, 0);
    SETGATE(idt[IRQ_OFFSET + 6], 0, GD_KT, &ENTRY_IRQ6, 0);
    SETGATE(idt[IRQ_OFFSET + 7], 0, GD_KT, &ENTRY_IRQ7, 0);
    SETGATE(idt[IRQ_OFFSET + 8], 0, GD_KT, &ENTRY_IRQ8, 0);
    SETGATE(idt[IRQ_OFFSET + 9], 0, GD_KT, &ENTRY_IRQ9, 0);
    SETGATE(idt[IRQ_OFFSET + 10], 0, GD_KT, &ENTRY_IRQ10, 0);
    SETGATE(idt[IRQ_OFFSET + 11], 0, GD_KT, &ENTRY_IRQ11, 0);
    SETGATE(idt[IRQ_OFFSET + 12], 0, GD_KT, &ENTRY_IRQ12, 0);
    SETGATE(idt[IRQ_OFFSET + 13], 0, GD_KT, &ENTRY_IRQ13, 0);
    SETGATE(idt[IRQ_OFFSET + 14], 0, GD_KT, &ENTRY_IRQ14, 0);
    SETGATE(idt[IRQ_OFFSET + 15], 0, GD_KT, &ENTRY_IRQ15, 0);
```

In `env_alloc()`, we initialize the value of `%eflags` for user environments to be `FL_IF` so that user environments always run with interrupt enabled.

```
    // Enable interrupts while in user mode.
    // LAB 4: Your code here.
    e->env_tf.tf_eflags = FL_IF;
```

By the way, since we design JOS so that interrupt is always disabled in kernel, all entries in IDT are interrupt gates which will clear IF bit when being passed through.

And now we set the IF bit when CPU is idle in `sched_halt()` to allow it to be awoken by interrupts.

```
    // Reset stack pointer, enable interrupts and then halt.
    asm volatile (
        "movl $0, %%ebp\n"
        "movl %0, %%esp\n"
        "pushl $0\n"
        "pushl $0\n"
        // Uncomment the following line after completing exercise 13
        "sti\n"
        "1:\n"
        "hlt\n"
        "jmp 1b\n"
    : : "a" (thiscpu->cpu_ts.ts_esp0));
```

Handling Clock Interrupts

Exercise 14

When interrupt with vector `IRQ_OFFSET + TIMER` is delivered, we transfer the control to `sched_yield()`. But remember to acknowledge the interrupt or it won't be generated again.

```
    if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER)
    {
        lapic_eoi();
        sched_yield();
    }
```

Inter-Process communication (IPC)

Implementing IPC

Exercise 15

In `sys_ipc_try_send()`, we tries to query the receiver's `env_ipc_recving` to see if it's ready for an IPC

```
    struct Env* e;
    int r;

    if ((r = envid2env(envid, &e, false)))
    {
        return r;
    }

    if(!e->env_ipc_recving)
    {
        return -E_IPC_NOT_RECV;
    }
```

Then if both sender and receiver are willing to transfer a page, we check if the arguments are valid and then perform the same operations as in `sys_page_map()` (we don't call it directly since in `sys_page_map()` the caller must pass the permission check in `envid2env()` for both the source and destination while IPC can happen between any environments)

```
    // only tries to transfer a page when they're both willing
    if ((intptr_t)(srcva) < UTOP && (intptr_t)(e->env_ipc_dstva) < UTOP)
    {
        if ((intptr_t)(srcva) % PGSIZE)
        {
            return -E_INVAL;
        }
        if ((~perm & PTE_P) || (~perm & PTE_U) || (perm & ~PTE_SYSCALL))
        {
            return -E_INVAL;
        }

        // we don't call sys_page_map() to do this since it
        // have more strict permisson request
        struct PageInfo* p;
        pte_t *pte;
        p = page_lookup(curenv->env_pgdir, srcva, &pte);
        if (p == NULL)
        {
            return -E_INVAL;
        }
        if ((perm & PTE_W) && (~(*pte) & PTE_W))
        {
            return -E_INVAL;
        }
        if ((r = page_insert(e->env_pgdir, p, e->env_ipc_dstva, perm)))
        {
```

```
                return r;
            }
        }
        else // now all checks have been passed at this position
        {
            e->env_ipc_perm = 0;
        }
```

Then we fill in the IPC relevant field in the environment receiving and mark it as runnable again. The receiver's `%eax` stored in its `env_tf` is modified to 0 to act as the return value of `sys_ipc_recv()` called by the receiver.

```
    e->env_ipc_recving = 0;
    e->env_ipc_from = curenv->env_id;
    e->env_ipc_value = value;
    e->env_ipc_perm = perm;

    e->env_status = ENV_RUNNABLE; // again we don't use sys_env_set_status for permission reason
    e->env_tf.tf_regs.reg_eax = 0; // return 0 for sys_ipc_recv()

    return 0;
```

In `sys_ipc_rev()`, we first check the validity of `dstva` and set `env_ipc_recving` true. Then we mark current environment as not runnable and give up CPU.

```
static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if ((intptr_t)(dstva) < UTOP && (intptr_t)(dstva) % PGSIZE)
    {
        return -E_INVAL;
    }
    curenv->env_ipc_recving = true;
    curenv->env_ipc_dstva = dstva;

    curenv->env_status = ENV_NOT_RUNNABLE;
    sched_yield();

    // panic("sys_ipc_recv not implemented");

    return 0; // the function actually doesn't return here
}
```

For the wrapper function, if the argument `pg` is set as `NULL` to indicate the unwilling of transferring a page in the IPC, we translate it as `UTOP` as an address `>= UTOP` is with the same semantic in the system calls. In `ipc_send()`, we keep trying to send the message and give up CPU after each failure because of `E_IPC_NOT_RECV` as a nice environment.

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
```

```c
    // LAB 4: Your code here.
    int r;

    // an address >= UTOP means no receiving page in sys_ipc_recv()
    pg = pg == NULL ? (void *)UTOP : pg;
    do
    {
        r = sys_ipc_try_send(to_env, val, pg, perm);
        if (r)
        {
            if (r != -E_IPC_NOT_RECV)
            {
                panic("ipc_send: %e", r);
            }
            else
            {
                sys_yield();
            }
        }
    }
    while(r);

    // panic("ipc_send not implemented");
}

int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    // an address >= UTOP means no receiving page in sys_ipc_recv()
    int r;
    pg = pg == NULL ? (void *)UTOP : pg;
    if ((r = sys_ipc_recv(pg)))
    {
        if(from_env_store != NULL) *from_env_store = 0;
        if(perm_store != NULL) *perm_store = 0;
        return r;
    }
    if (from_env_store != NULL) *from_env_store = thisenv->env_ipc_from;
    if (perm_store != NULL) *perm_store = thisenv->env_ipc_perm;
    return thisenv->env_ipc_value;

    // panic("ipc_recv not implemented");
    // return 0;
}
```

Challenge!

`ipc_send()` have to loop because `sys_ipc_try_send()` would fail when the receiver is not receiving or some other environment get to send to it before. This is an error recoverable in term of time. For example if the scheduler let a sender runs `sys_ipc_try_send()` before the receiver calls `sys_ipc_recv()`, the first `sys_ipc_try_send()` will fail while the second one will success. That's we have to loop to wait for the receiver.

We can modify the IPC mechanism of the kernel to help environments to handle this. The idea is that we pause the sender if the receiver is not ready and record it in the receiver's waiting queue, and each time the receiver calls `sys_ipc_recv()` we try to do IPC with the first environment in the queue (actually a last-in-first-out here because of my implementation) and wake the sender up on whether a success or failure, returning from `sys_ipc_try_send()` without the need of further loops.

We add two pointers in the environment structure since an environment can wait for a receiver to call `sys_ipc_recv()` as some other environment are waiting for it to call `sys_ipc_recv()`, which is the case in `user/primes`

```
    struct Env *env_ipc_queue; // the head of IPC waiting queue (this is receiver)
    struct Env *env_ipc_next; // next waiting environment in the same waiting queue (this is sender)
```

They are initialized in `env_alloc()`

```
    // clean the IPC waiting queue pointers
    e->env_ipc_queue = NULL;
    e->env_ipc_next = NULL;
```

The changes in `syscall.c` are explained in the comments.

Note that it's ok to share the variables to store sending and receiving variables since one environment can not be a sender and a receiver simultaneous, although it can be the sender while some others are trying to send message to it as explained above.

```
// handle the IPC to dst from src (the head of dst's waiting queue)
// contains much of the original version of sys_ipc_try_send()
// can be called both from the sender or the receiver when
// (1) receiver calls sys_ipc_recv() when some environments are waiting to send
// (2) sender calls sys_ipc_try_send() when the receiver is ready to receiver
// see sys_ipc_try_send() for possible errors and more information
static int
handle_ipc(struct Env* dst)
{
    int r;

    // pop the front of the waiting queue
    struct Env* src = dst->env_ipc_queue;
    assert(src != NULL);
    dst->env_ipc_queue = src->env_ipc_next;

    // restore the arguments from IPC relevant field
    void *srcva = src->env_ipc_dstva;
    uint32_t value = src->env_ipc_value;
    int perm = src->env_ipc_perm;

    // only tries to transfer a page when they're both willing
    if ((intptr_t)(srcva) < UTOP && (intptr_t)(dst->env_ipc_dstva) < UTOP)
```

```
    {
        if ((intptr_t)(srcva) % PGSIZE)
        {
            r = -E_INVAL;
            goto ret;
        }
        if ((~perm & PTE_P) || (~perm & PTE_U) || (perm & ~PTE_SYSCALL))
        {
            r = -E_INVAL;
            goto ret;
        }

        // we don't call sys_page_map() to do this since it
        // have more strict permisson request
        struct PageInfo* p;
        pte_t *pte;
        p = page_lookup(src->env_pgdir, srcva, &pte);
        if (p == NULL)
        {
            r = -E_INVAL;
            goto ret;
        }
        if ((perm & PTE_W) && (~(*pte) & PTE_W))
        {
            r = -E_INVAL;
            goto ret;
        }
        if ((r = page_insert(dst->env_pgdir, p, dst->env_ipc_dstva, perm)))
        {
            goto ret;
        }
    }
    else // now all checks have been passed at this position
    {
        dst->env_ipc_perm = 0;
    }

    dst->env_ipc_recving = 0;
    dst->env_ipc_from = src->env_id;
    dst->env_ipc_value = value;
    dst->env_ipc_perm = perm;

    r = 0;
ret:
    // store return value in sender's or receiver's %eax
    // in case they're sleeping
    if (src->env_status == ENV_NOT_RUNNABLE)
    {
        src->env_status = ENV_RUNNABLE;
        src->env_tf.tf_regs.reg_eax = r;
    }
    if (dst->env_status == ENV_NOT_RUNNABLE && !r) // receiver only wake up on success
    {
        dst->env_status = ENV_RUNNABLE;
        dst->env_tf.tf_regs.reg_eax = r;
```

```c
    }
    // cprintf("handl_ipc(): from %x to %x, value = %d, retval = %d\n",
    //      src->env_id, dst->env_id, value, r);
    return r;
}

static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    struct Env* e;
    int r;

    if ((r = envid2env(envid, &e, false)))
    {
        return r;
    }

    // add current environment to the head of waiting queue of receiving environemnt
    curenv->env_ipc_next = e->env_ipc_queue;
    e->env_ipc_queue = curenv;
    // store the arguments in IPC relevant field
    curenv->env_ipc_value = value;
    curenv->env_ipc_dstva = srcva;
    curenv->env_ipc_perm = perm;

    if(!e->env_ipc_recving)
    {
        // return -E_IPC_NOT_RECV;

        // give up CPU if receiver isn't ready
        // instead of return -E_IPC_NOT_RECV
        curenv->env_status = ENV_NOT_RUNNABLE;
        sched_yield();
    }
    // otherwise do the IPC
    return handle_ipc(e);

    // panic("sys_ipc_try_send not implemented");
}

static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    int r;

    if ((intptr_t)(dstva) < UTOP && (intptr_t)(dstva) % PGSIZE)
    {
        return -E_INVAL;
    }
    curenv->env_ipc_recving = true;
    curenv->env_ipc_dstva = dstva;

    // travel IPC waiting queue (loop because some might fail)
```

```c
    while (curenv->env_ipc_queue != NULL)
    {
        r = handle_ipc(curenv);
        if (!r) return r;
    }

    // no valid waiting environment, give up the CPU
    curenv->env_status = ENV_NOT_RUNNABLE;
    sched_yield();

    // panic("sys_ipc_recv not implemented");

    return 0; // the function actually doesn't return here
}
```

Now we can implement an `ipc_send()` without loops.

```c
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    int r;

    // an address >= UTOP means no receiving page in sys_ipc_recv()
    pg = pg == NULL ? (void *)UTOP : pg;
    r = sys_ipc_try_send(to_env, val, pg, perm);
    if (r)
    {
        panic("ipc_send: %e", r);
    }

    // panic("ipc_send not implemented");
}
```

Reference

[1] IA32 volume 32, section 6.2.1