JOS Lab4

Jen-tse Huang, 1500017836

Exercise 1

- `mmio_map_region()`

Map the memory, remember to check the overflow.

```
size = ROUNDUP(size, PGSIZE);
if (base + size > MMIOLIM) {
    panic("mmio_map_region overflows!");
}
boot_map_region(kern_pgdir, base, size, pa, PTE_W | PTE_PCD | PTE_PWT);

// Return the base of the reserved region
uintptr_t ret_base = base;
base += size;
return ret_base;
```

Exercise 2

`boot-aps()` first setup the memory and boot all the CPUs. And it wait for `mp_main()` to initiate the environment and traps.

Add this code in `page_init()` in `pamp.c`:

```
    } else if (i == MPENTRY_PADDR / PGSIZE) {
        // AP bootstrap code
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
```

After `make qemu`, we can see:

```
check_page_free_list() succeeded!
```

We pass the test.

- Q: Compare `kern/mpentry.S` side by side with `boot/boot.S`. Bearing in mind that `kern/mpentry.S` is compiled and linked to run above `KERNBASE` just like everything else in the kernel, what is the purpose of macro `MPBOOTPHYS`? Why is it necessary in `kern/mpentry.S` but not in `boot/boot.S`? In other words, what could go wrong if it were omitted in `kern/mpentry.S`?

  A: The macro is to convert the address to a low address. Because `kern/mpentry.S` is compiled and linked to run above `KERNBASE`, so all addresses are above `0xf0000000` which in real mode CPU could not access. If we remove it, there will be something wrong with the address.

Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

Exercise 3

- `mem_init_mp()`

We need to map the per-CPU stacks. This function is easy, just follow the instruction.

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

There is an interval of size `KSTKGAP`. This is to prevent one stack from overwriting the other.

After `make qemu`, we can see:

```
check_kern_pgdir() succeeded!
```

We pass the test.

Exercise 4

- `trap_init_percpu()`

First, use `thiscpu->cpu_ts` to replace `ts`.

Then, set the stack as what we set in `mem_init_mp()`, use `thiscpu->cpu_id` to replace the `i`

The TSS descriptor is `gdt[(GD_TSS0 >> 3) + cpu_idx]`, so the TSS selector is `GD_TSS0 + (cpu_idx << 3)`

```
struct Taskstate *this_ts = &thiscpu->cpu_ts;
uint8_t cpu_idx = thiscpu->cpu_id;

this_ts->ts_esp0 = KSTACKTOP - cpu_idx * (KSTKSIZE + KSTKGAP);
this_ts->ts_ss0 = GD_KD;
this_ts->ts_iomb = sizeof(struct Taskstate);

gdt[(GD_TSS0 >> 3) + cpu_idx] = SEG16(STS_T32A, (uint32_t) (this_ts),
                    sizeof(struct Taskstate) - 1, 0);
gdt[(GD_TSS0 >> 3) + cpu_idx].sd_s = 0;

ltr(GD_TSS0 + (cpu_idx << 3));
lidt(&idt_pd);
```

After `make qemu CPUS=4`, we can see:

```
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
```

We pass the test.

Exercise 5

There is always **one** environment running in kernel mode. So we need to add lock.

- `i386_init()`

Acquire the lock before the BSP wakes up the other CPUs.

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

- `mp_main()`

Acquire the lock after initializing the AP, and then call `sched_yield()` to start running environments on this AP.

```
    // 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()`

Acquire the lock when trapped from user mode.

```
        // Acquire the big kernel lock before doing any
        // serious kernel work.
        // LAB 4: Your code here.
        lock_kernel();
```

- `env_run()`

Release the lock *right before* switching to user mode.

```
// Release the lock before env_pop_tf()
unlock_kernel();

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

- Q: It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

Exercise 6

- `sched_yield()`

(Note that `curenv` equals to `thiscpu->cpu_env`)

We can get the index of environment by `curenv - envs`. The searching method uses a `cnt` to record total searching times and an `i`, which keep increasing itself by one and check not to overflow `NENV`.

```
// Search from the next environment after after the previously running environment,
// from beginning if there was no previously running environment
int i = (curenv == NULL) ? 0 : (curenv - envs + 1) % NENV;
for (int cnt = 0; cnt < NENV; ++ cnt) {
  if (envs[i].env_status == ENV_RUNNABLE) {
    env_run(&envs[i]);
  }
  i = (i + 1) % NENV;
}
// We can run the environment with the ENV_RUNNING status
// only when it is current environment and we cannot find other environment to run
if (curenv != NULL && curenv->env_status == ENV_RUNNING) {
  env_run(curenv);
}
```

- `syscall()`

```
case SYS_yield:
  sys_yield();
  return 0;
```

- `i386_init()`

Remember to comment the `ENV_CREATE(user_primes, ENV_TYPE_USER);`

```
// 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);
```

After `make qemu CPUS=2`, we can see:

```
SMP: CPU 0 found 2 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
[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.
Back in environment 00001001, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001002, iteration 2.
Back in environment 00001001, iteration 3.
Back in environment 00001000, iteration 4.
Back in environment 00001002, iteration 3.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
Back in environment 00001002, iteration 4.
All done in environment 00001001.
All done in environment 00001002.
[00001001] exiting gracefully
[00001001] free env 00001001
[00001002] exiting gracefully
[00001002] free env 00001002
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

We pass this exercise.

- Q: In your implementation of `env_run()` you should have called `lcr3()`. Before and after the call to `lcr3()`, your code makes references (at least it should) to the variable `e`, the argument to `env_run`. Upon loading the `%cr3` register, the addressing context used by the MMU is instantly changed. But a virtual address (namely `e`) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer `e` be dereferenced both before and after the addressing switch?
- Q: Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

Exercise 7

- `sys_exofork()`

Remember to use `curenv->envid` but not `ENVX(curenv->envid)`.

Then, **how to let the child return zero?** Note that when an environment is selected to run by `env_run()`, the values in `tf` will be popped in registers. Before this, the function was not returned yet. So we can modify the child's `tf.tf_regs.reg_eax` which the return value resides in to zero.

```
struct Env *new_env;
int ret = env_alloc(&new_env, curenv->env_id);
if (ret < 0) {
    return ret;
}
new_env->env_status = ENV_NOT_RUNNABLE;
new_env->env_tf = curenv->env_tf;
// Let the child return 0
new_env->env_tf.tf_regs.reg_eax = 0;
return new_env->env_id;
```

- `sys_env_set_status()`

```
if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE) {
    return -E_INVAL;
}
struct Env *e;
if (envid2env(envid, &e, 1) < 0) {
    return -E_BAD_ENV;
}
e->env_status = status;
return 0;
```

- `sys_page_alloc()`

Remember to free the page if `page_insert()` fails.

```
struct Env *e;
if (envid2env(envid, &e, 1) < 0) {
    return -E_BAD_ENV;
}
if ((uintptr_t)va >= UTOP ||
    (uintptr_t)va % PGSIZE != 0) {
    // Environment envid doesn't currently exist,
    // or the caller doesn't have permission to change envid
    return -E_INVAL;
}
if ((~perm & (PTE_U | PTE_P)) != 0 ||
    (perm & ~(PTE_U | PTE_P | PTE_AVAIL | PTE_W)) != 0) {
    // Perm is inappropriate

    return -E_INVAL;
    }
    struct PageInfo *pp = page_alloc(ALLOC_ZERO);
    if (pp == NULL) {
        return -E_NO_MEM;
    }
    if (page_insert(e->env_pgdir, pp, va, perm) < 0) {
        // Remember to free the page
        page_free(pp);
        return -E_NO_MEM;
    }
    return 0;
```

- `sys_page_map()`

Check all the restriction.

```
    struct Env *srcenv, *dstenv;
    if (envid2env(srcenvid, &srcenv, 1) < 0 ||
        envid2env(dstenvid, &dstenv, 1)) {
        return -E_BAD_ENV;
    }
    if ((uintptr_t)srcva >= UTOP ||
        (uintptr_t)srcva % PGSIZE != 0 ||
        (uintptr_t)dstva >= UTOP ||
        (uintptr_t)dstva % PGSIZE != 0) {
        // Environment envid doesn't currently exist,
        // or the caller doesn't have permission to change envid
        return -E_INVAL;
    }
    pte_t *pte;
    struct PageInfo *pp = page_lookup(srcenv->env_pgdir, srcva, &pte);
    if (pp == NULL) {
        // srcva is not mapped in srcenvid's address space
        return -E_INVAL;
    }
    if ((~perm & (PTE_U | PTE_P)) != 0 ||
        (perm & ~(PTE_U | PTE_P | PTE_AVAIL | PTE_W)) != 0) {
        // Perm is inappropriate
        return -E_INVAL;
    }
    if ((perm & PTE_W) == 1 && (*pte & PTE_W) == 0) {
        // Read-only
        return -E_INVAL;
    }
    if (page_insert(dstenv->env_pgdir, pp, dstva, perm) < 0) {
        return -E_NO_MEM;
    }
    return 0;
```

- `sys_page_unmap()`

If no page is mapped, the function silently succeeds.

```
    struct Env *e;
    if (envid2env(envid, &e, 1) < 0) {
        return -E_BAD_ENV;
    }
    if ((uintptr_t)va >= UTOP ||
        (uintptr_t)va % PGSIZE != 0) {
        // Environment envid doesn't currently exist,
        // or the caller doesn't have permission to change envid
        return -E_INVAL;
    }
    page_remove(e->env_pgdir, va);
    return 0;
```

- `syscall()`

Remember to add cases in `syscall()`:

```
    case SYS_yield:
        sys_yield();
        return 0;
    case SYS_exofork:
        return sys_exofork();
    case SYS_env_set_status:
        return sys_env_set_status((envid_t)a1, (int)a2);
    case SYS_page_alloc:
        return sys_page_alloc((envid_t)a1, (void *)a2, (int)a3);
    case SYS_page_map:
        return sys_page_map((envid_t)a1, (void *)a2, (envid_t)a3, (void *)a4, (int)a5);
    case SYS_page_unmap:
        return sys_page_unmap((envid_t)a1, (void *)a2);
```

After `make grade`, we can see:

```
dumbfork: OK (0.9s)
    (Old jos.out.dumbfork failure log removed)
Part A score: 5/5
```

Part A ends.

Exercise 8

- `sys_env_set_pgfault_upcall`

The function sets `env_pgfault_cupcall` to `func`

```
struct Env *e;
if (envid2env(envid, &e, 1) < 0) {
    return -E_BAD_ENV;
}
e->env_pgfault_upcall = func;
return 0;
```

Remember to add in `syscall()`:

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

Exercise 9

- `page_fault_handler`

We firstly check if `curenv->env_pgfault_upcall` exists. Then if it is user stack now, we push the current state into exception stack. If it is exception stack now, we push the current state after an interval.

```
if (curenv->env_pgfault_upcall == NULL) {
    // The function does not exist
    cprintf("[%08x] user fault va %08x ip %08x\n", curenv->env_id, fault_va, tf->tf_eip);
    print_trapframe(tf);
    env_destroy(curenv);
}

struct UTrapframe *utf = NULL;
uint32_t cur_esp = tf->tf_esp;

if (cur_esp < USTACKTOP && cur_esp >= USTACKTOP - PGSIZE) {
    // User stack
    utf = (struct UTrapframe *)(UXSTACKTOP - sizeof(struct UTrapframe));
} else if (cur_esp < UXSTACKTOP && cur_esp >= UXSTACKTOP - PGSIZE) {
    // Exception stack
    utf = (struct UTrapframe *)(cur_esp - 4 - sizeof(struct UTrapframe));
}

// Check the tests above
user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe), PTE_U | PTE_W);

// Set UTrapframe
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;

// Branch to handler function
tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
tf->tf_esp = (uint32_t)utf;
```

```
    env_run(curenv);
```

Exercise 10

- `_pgfault_upcall`

`struct UTrapframe` and `struct PushRegs` are useful:

```
struct UTrapframe {
    /* information about the fault */
    uint32_t utf_fault_va;  /* va for T_PGFLT, 0 otherwise */
    uint32_t utf_err;
    /* trap-time return state */
    struct PushRegs utf_regs;
    uintptr_t utf_eip;
    uint32_t utf_eflags;
    /* the trap-time stack to return to */
    uintptr_t utf_esp;
} __attribute__((packed));

struct PushRegs {
    /* registers as pushed by pusha */
    uint32_t reg_edi;
    uint32_t reg_esi;
    uint32_t reg_ebp;
    uint32_t reg_oesp;      /* Useless */
    uint32_t reg_ebx;
    uint32_t reg_edx;
    uint32_t reg_ecx;
    uint32_t reg_eax;
} __attribute__((packed));
```

The pointer to trap-time stack is in `utf->esp`, which is the 13th word in `struct UTrapframe`, and `%eip` is in `utf->eip`, which is the 11th word. So the offset is `0x30` and `0x28` separately.

Remember to sub the `utf->esp` to push `%eip`.

```
    subl $0x4, 0x30(%esp)
    movl 0x30(%esp), %eax
    movl 0x28(%esp), %ebx
    movl %ebx, (%eax)
```

To pop all the registers, we need to modify `%esp` to the offset of registers, which is `0x8`.

```
    addl $0x8, %esp
    popal
```

To pop `eflags`, we need to modify `%esp` to the offset of `eflags`, which is `0x4` after `popal` add the `%esp` automatically.

```
addl $0x4, %esp
popfl
```

Then switch back to the trap-time stack, just in the `(%esp)`

```
movl (%esp), %esp
```

Finally, use `ret` to restore the `%eip`

```
ret
```

Exercise 11

- `set_pgfault_handler()`

If it is the first time (`_pgfault_handler == 0`), then we will need to set the page fault upcall for the current environment.

Each user environment that wants to support user-level page fault handling will need to allocate memory for its own exception stack, using the `sys_page_alloc()` system call.

Set the `envid` equals zero to refer to current environment.

```
    if (sys_page_alloc(0, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P) < 0) {
        panic("set_pgfault_handler: sys_page_alloc failed!");
    }
    if (sys_env_set_pgfault_upcall(0, _pgfault_upcall) < 0) {
        panic("set_pgfault_handler: sys_env_set_pgfault_upcall failed!");
    }
```

After `make run-faultread`, we can see:

```
[00000000] new env 00001000
[00001000] user fault va 00000000 ip 00800039
TRAP frame at 0xf02af000 from CPU 0
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0x00000000
  err  0x00000004 [user, read, not-present]
  eip  0x00800039
```

```
    cs   0x----001b
    flag 0x00000086
    esp  0xeebfdfc0
    ss   0x----0023
[00001000] free env 00001000
```

After `make run-faultdie`, we can see:

```
[00000000] new env 00001000
i faulted at va deadbeef, err 6
[00001000] exiting gracefully
[00001000] free env 00001000
```

After `make run-faultalloc`, we can see:

```
[00000000] new env 00001000
fault deadbeef
this string was faulted in at deadbeef
fault cafebffe
fault cafec000
this string was faulted in at cafebffe
[00001000] exiting gracefully
[00001000] free env 00001000
```

After `make run-faultallocbad`, we can see:

```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va deadbeef
[00001000] free env 00001000
```

We pass the exercises above.

Exercise 12

- `fork()`

We need to set page fault upcall for the child because it is not copied from the parent. This is stored in `envs`.

`uvpd` is the beginning of page table directory and `uvpd` is the beginning of page tables. We can use them to check the permissions. They are defined in `lib/entry.S`:

```
    .globl uvpt
    .set uvpt, UVPT
    .globl uvpd
    .set uvpd, (UVPT+(UVPT>>12)*4)
```

So the code is:

```
    set_pgfault_handler(pgfault);

    envid_t envid = sys_exofork();
    if (envid < 0) {
        panic("fork: sys_exofork failed!");
    } else if (envid == 0) {
        thisenv = envs + ENVX(sys_getenvid());
        // The other work is done by parent, we can simply return in child
        return envid;
    }

    // In the parent
    // Copy the mapping of pages
    for (uint32_t addr = UTEXT; addr < UTOP; addr += PGSIZE) {
        // For parent's exception stack
        if (addr != UXSTACKTOP - PGSIZE) {
            if (((uvpd[PDX(addr)] & PTE_P) != 0) &&
                ((~uvpt[PGNUM(addr)] & (PTE_P | PTE_U)) == 0)) {
                duppage(envid, addr / PGSIZE);
            }
        }
    }
    // Allocate a page for child's exception stack
    if (sys_page_alloc(envid, (void *)(UXSTACKTOP - PGSIZE), PTE_P | PTE_U | PTE_W) < 0) {
        panic("fork: sys_page_alloc failed!");
    }
    // Setup page fault handler of child
    extern void _pgfault_upcall(void);
    if (sys_env_set_pgfault_upcall(envid, _pgfault_upcall) < 0) {
        panic("fork: sys_env_set_pgfault_upcall failed!");
    }
    // Mark the child as runnable and return
    if (sys_env_set_status(envid, ENV_RUNNABLE) < 0) {
        panic("fork: sys_env_set_status failed!");
    }

    return envid;
```

- `duppage()`

Remember to map the read-only pages separately:

```
    void *addr = (void *)(pn * PGSIZE);

    if ((uvpt[pn] & (PTE_W | PTE_COW)) != 0) {
        // Map child's address space first
        if (sys_page_map(0, addr, envid, addr, PTE_P | PTE_U | PTE_COW) < 0) {
            panic("duppage: sys_page_map envid failed!");
        }
        // Then map parent's
        if (sys_page_map(0, addr, 0, addr, PTE_P | PTE_U | PTE_COW) < 0) {
            panic("duppage: sys_page_map 0 failed!");
        }
    } else {
        // Sets both PTEs so that the page is not writeable
        if (sys_page_map(0, addr, envid, addr, PTE_P | PTE_U) < 0) {
            panic("duppage: sys_page_map envid (RO) failed!");
        }
    }

    return 0;
```

- Q: Why do we need to mark our copy-on-write again if it was already copy-on-write at the beginning of this function?

  A: Because there may be many `fork()` s, we should map each one separately.
- Q: Why mapping the address space of the child first?

  A: If we map the parent's address space first, parent's address space will change, so the child can see those changes, which is what we do not want to happen.
- `pgfault()`

Note: to check the error number, do not use `err == FEC_WR` , use `err & FEC_WR ==1` instead.

```
    if ((err & FEC_WR) == 0 ||
        (uvpd[PDX(addr)] & PTE_P) == 0 ||
        (~uvpt[PGNUM(addr)] & (PTE_COW | PTE_P)) != 0) {
        panic("pgdault failed!");
    }
```

Remember to `ROUNDOWN` the address, and `sys_page_unmap` the temporary address.

`sys_page_map` will automatically remove the previous page.

```
    addr = (void *)ROUNDDOWN(addr, PGSIZE);
    if (sys_page_alloc(0, (void *)PFTEMP, PTE_P | PTE_W | PTE_U) < 0) {
        panic("pgfault: sys_page_alloc failed!");
    }
    memcpy((void *)PFTEMP, addr, PGSIZE);
    if (sys_page_map(0, (void *)PFTEMP, 0, addr, PTE_P | PTE_W | PTE_U) < 0) {
        panic("pgfault: sys_page_map failed!");
    }
    if (sys_page_unmap(0, (void *)PFTEMP) < 0) {
        panic("pgfault: sys_page_unmap failed!");
    }
```

After `make run-forktree` , we can see that:

```
[00000000] new env 00001000
1000: I am ''
[00001000] new env 00001001
[00001000] new env 00001002
[00001000] exiting gracefully
```

```
[00001000] free env 00001000
1001: I am '0'
[00001001] new env 00002000
[00001001] new env 00001003
[00001001] exiting gracefully
[00001001] free env 00001001
2000: I am '00'
[00002000] new env 00002001
[00002000] new env 00001004
[00002000] exiting gracefully
[00002000] free env 00002000
2001: I am '000'
[00002001] exiting gracefully
[00002001] free env 00002001
1002: I am '1'
[00001002] new env 00003001
[00001002] new env 00003000
[00001002] exiting gracefully
[00001002] free env 00001002
3000: I am '11'
[00003000] new env 00002002
[00003000] new env 00001005
[00003000] exiting gracefully
[00003000] free env 00003000
3001: I am '10'
[00003001] new env 00004000
[00003001] new env 00001006
[00003001] exiting gracefully
[00003001] free env 00003001
4000: I am '100'
[00004000] exiting gracefully
[00004000] free env 00004000
2002: I am '110'
[00002002] exiting gracefully
[00002002] free env 00002002
1003: I am '01'
[00001003] new env 00003002
[00001003] new env 00005000
[00001003] exiting gracefully
[00001003] free env 00001003
5000: I am '011'
[00005000] exiting gracefully
[00005000] free env 00005000
3002: I am '010'
[00003002] exiting gracefully
[00003002] free env 00003002
1004: I am '001'
[00001004] exiting gracefully
[00001004] free env 00001004
1005: I am '111'
[00001005] exiting gracefully
[00001005] free env 00001005

1006: I am '101'
```

```
[00001006] exiting gracefully
[00001006] free env 00001006
```

And `make grade`:

```
faultread: OK (0.9s)
faultwrite: OK (1.0s)
faultdie: OK (1.0s)
   (Old jos.out.faultdie failure log removed)
faultregs: OK (0.9s)
   (Old jos.out.faultregs failure log removed)
faultalloc: OK (1.1s)
   (Old jos.out.faultalloc failure log removed)
faultallocbad: OK (0.9s)
   (Old jos.out.faultallocbad failure log removed)
faultnostack: OK (1.9s)
   (Old jos.out.faultnostack failure log removed)
faultbadhandler: OK (2.2s)
   (Old jos.out.faultbadhandler failure log removed)
faultevilhandler: OK (2.1s)
   (Old jos.out.faultevilhandler failure log removed)
forktree: OK (2.0s)
   (Old jos.out.forktree failure log removed)
Part B score: 50/50
```

Part B ends.

Exercise 13

- `trapentry.S`

```
TRAPHANDLER_NOEC(IRQ_0, IRQ_OFFSET + 0)
TRAPHANDLER_NOEC(IRQ_1, IRQ_OFFSET + 1)
TRAPHANDLER_NOEC(IRQ_2, IRQ_OFFSET + 2)
TRAPHANDLER_NOEC(IRQ_3, IRQ_OFFSET + 3)
TRAPHANDLER_NOEC(IRQ_4, IRQ_OFFSET + 4)
TRAPHANDLER_NOEC(IRQ_5, IRQ_OFFSET + 5)
TRAPHANDLER_NOEC(IRQ_6, IRQ_OFFSET + 6)
TRAPHANDLER_NOEC(IRQ_7, IRQ_OFFSET + 7)
TRAPHANDLER_NOEC(IRQ_8, IRQ_OFFSET + 8)
TRAPHANDLER_NOEC(IRQ_9, IRQ_OFFSET + 9)
TRAPHANDLER_NOEC(IRQ_10, IRQ_OFFSET + 10)
TRAPHANDLER_NOEC(IRQ_11, IRQ_OFFSET + 11)
TRAPHANDLER_NOEC(IRQ_12, IRQ_OFFSET + 12)
TRAPHANDLER_NOEC(IRQ_13, IRQ_OFFSET + 13)
TRAPHANDLER_NOEC(IRQ_14, IRQ_OFFSET + 14)
TRAPHANDLER_NOEC(IRQ_15, IRQ_OFFSET + 15)
```

- `trap_init()`

```
extern void IRQ_0();
extern void IRQ_1();
extern void IRQ_2();
extern void IRQ_3();
extern void IRQ_4();
extern void IRQ_5();
extern void IRQ_6();
extern void IRQ_7();
extern void IRQ_8();
extern void IRQ_9();
extern void IRQ_10();
extern void IRQ_11();
extern void IRQ_12();
extern void IRQ_13();
extern void IRQ_14();
extern void IRQ_15();
SETGATE(idt[IRQ_OFFSET + 0], 0, GD_KT, IRQ_0, 0);
SETGATE(idt[IRQ_OFFSET + 1], 0, GD_KT, IRQ_1, 0);
SETGATE(idt[IRQ_OFFSET + 2], 0, GD_KT, IRQ_2, 0);
SETGATE(idt[IRQ_OFFSET + 3], 0, GD_KT, IRQ_3, 0);
SETGATE(idt[IRQ_OFFSET + 4], 0, GD_KT, IRQ_4, 0);
SETGATE(idt[IRQ_OFFSET + 5], 0, GD_KT, IRQ_5, 0);
SETGATE(idt[IRQ_OFFSET + 6], 0, GD_KT, IRQ_6, 0);
SETGATE(idt[IRQ_OFFSET + 7], 0, GD_KT, IRQ_7, 0);
SETGATE(idt[IRQ_OFFSET + 8], 0, GD_KT, IRQ_8, 0);
SETGATE(idt[IRQ_OFFSET + 9], 0, GD_KT, IRQ_9, 0);
SETGATE(idt[IRQ_OFFSET + 10], 0, GD_KT, IRQ_10, 0);
SETGATE(idt[IRQ_OFFSET + 11], 0, GD_KT, IRQ_11, 0);
SETGATE(idt[IRQ_OFFSET + 12], 0, GD_KT, IRQ_12, 0);
SETGATE(idt[IRQ_OFFSET + 13], 0, GD_KT, IRQ_13, 0);
SETGATE(idt[IRQ_OFFSET + 14], 0, GD_KT, IRQ_14, 0);
SETGATE(idt[IRQ_OFFSET + 15], 0, GD_KT, IRQ_15, 0);
```

- `env_alloc()`

```
e->env_tf.tf_eflags |= FL_IF;
```

- `sched_halt()`

```
  "sti\n"
```

Exercise 14

- `trap_dispatch()`

The clock interrupt is IRQ 0.

```
    if (tf->tf_trapno == IRQ_OFFSET + 0) {
        lapic_eoi();
        sched_yield();
    }
```

After `make run-spin`, we can see:

```
[00000000] new env 00001000
I am the parent.  Forking the child...
[00001000] new env 00001001
I am the parent.  Running the child...
I am the child.  Spinning...
I am the parent.  Killing the child...
[00001000] destroying 00001001
[00001000] free env 00001001
[00001000] exiting gracefully
[00001000] free env 00001000
```

Multiple CPUS also work (`make CPUS=2 run-spin`):

```
SMP: CPU 0 found 2 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
[00000000] new env 00001000
I am the parent.  Forking the child...
[00001000] new env 00001001
I am the parent.  Running the child...
I am the child.  Spinning...
I am the parent.  Killing the child...
[00001000] destroying 00001001
[00001000] exiting gracefully
[00001000] free env 00001000
[00001001] free env 00001001
```

And after `make grade`:

```
spin: OK (1.9s)
    (Old jos.out.spin failure log removed)
stresssched: OK (2.4s)
    (Old jos.out.stresssched failure log removed)
```

Exercise 15

- `syscall()`

First, add cases in `syscall()`:

```
    case SYS_ipc_recv:
        return sys_ipc_recv((void *)a1);
    case SYS_ipc_try_send:
        return sys_ipc_try_send((envid_t)a1, (uint32_t) a2, (void *)a3, (unsigned) a4);
```

- `sys_ipc_recv()`

```
    if ((uint32_t)dstva < UTOP && PGOFF(dstva) != 0) {
        return -E_INVAL;
    }
    curenv->env_ipc_recving = 1;
    curenv->env_ipc_dstva = dstva;
    curenv->env_status = ENV_NOT_RUNNABLE;

    sys_yield();
```

- `sys_ipc_try_send()`

Remember to set the return value of receiving environment to zero, because `sys_ipc_recv` has gave up CPU and cannot return directly.

```
    struct Env *e;

    if (envid2env(envid, &e, 0) < 0) {
        // Environment envid doesn't currently exist
        return -E_BAD_ENV;
    }
    if (e->env_ipc_recving != 1) {
        // envid is not currently blocked in sys_ipc_recv
        return -E_IPC_NOT_RECV;
    }
    if ((uint32_t)srcva < UTOP) {
        // srcva < UTOP, want to send a page
        if (PGOFF(srcva) != 0) {
            // srcva is not page-aligned
            return -E_INVAL;
        }
        if ((~perm & (PTE_U | PTE_P)) != 0 ||
            (perm & ~(PTE_U | PTE_P | PTE_AVAIL | PTE_W)) != 0) {
            // Perm is inappropriate
            return -E_INVAL;
        }
        pte_t *pte;
        struct PageInfo *pp = page_lookup(curenv->env_pgdir, srcva, &pte);
        if (pp == NULL) {
            // srcva is not mapped in the caller's address space
            return -E_INVAL;
        }
        if ((perm & PTE_W) == 1 && (*pte & PTE_W) == 0) {
            // Read-only
            return -E_INVAL;
        }
        if ((uint32_t)e->env_ipc_dstva < UTOP) {
            // Want to receive a page
            if (page_insert(e->env_pgdir, pp, e->env_ipc_dstva, perm) < 0) {
                // There's not enough memory to map srcva in envid's address space
                return -E_NO_MEM;
            } else {
                // Succeeded!
                e->env_ipc_perm = perm;
            }
        }
    }

    // Send the value
    e->env_ipc_recving = 0;
    e->env_ipc_value = value;
    e->env_ipc_from = curenv->env_id;
    e->env_status = ENV_RUNNABLE;
    e->env_tf.tf_regs.reg_eax = 0;

    return 0;
```

- `ipc_recv()`

```
    if (from_env_store != NULL) {
        *from_env_store = 0;
    }
    if (perm_store != NULL) {
        *perm_store = 0;
    }
    if (pg == NULL) {
        // Set a value > UTOP
        pg = (void *)-1;
    }
    int r = sys_ipc_recv(pg);
    if (r < 0) {
        // Error
        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;
```

- `ipc_send()`

```
    if (pg == NULL) {

        // Set a value > UTOP
        pg = (void *)-1;
    }
    int r;
    while ((r = sys_ipc_try_send(to_env, val, pg, perm)) != 0) {
        if (r == 0) {
            // Succeed
            break;
        } else if (r == -E_IPC_NOT_RECV) {
            // Retry
            sys_yield();
        } else {
            // Error
            panic("ipc_send failed!");
        }
    }
```

After `make run-pingpong`, we can see:

```
[00000000] new env 00001000
[00001000] new env 00001001
send 0 from 1000 to 1001
1001 got 0 from 1000
1000 got 1 from 1001
1001 got 2 from 1000
1000 got 3 from 1001
1001 got 4 from 1000
1000 got 5 from 1001
1001 got 6 from 1000
1000 got 7 from 1001
1001 got 8 from 1000
1000 got 9 from 1001
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000
[00001001] exiting gracefully
[00001001] free env 00001001
```

After `make run-primes`, we can see:

```
[00000000] new env 00001000
[00001000] new env 00001001
CPU 0: 2 [00001001] new env 00001002
CPU 0: 3 [00001002] new env 00001003
CPU 0: 5 [00001003] new env 00001004
CPU 0: 7 [00001004] new env 00001005
CPU 0: 11 [00001005] new env 00001006
... ...
CPU 0: 8093 [000013fa] new env 000013fb
CPU 0: 8101 [000013fb] new env 000013fc
CPU 0: 8111 [000013fc] new env 000013fd

CPU 0: 8117 [000013fd] new env 000013fe
CPU 0: 8123 [000013fe] new env 000013ff
CPU 0: 8147 [000013ff] user panic in <unknown> at lib/fork.c:116: fork: sys_exofork failed!
```

After `make grade`, we can see:

```
spin: OK (1.7s)
stresssched: OK (2.5s)
sendpage: OK (1.7s)
    (Old jos.out.sendpage failure log removed)
pingpong: OK (1.9s)
    (Old jos.out.pingpong failure log removed)
primes: OK (4.0s)
    (Old jos.out.primes failure log removed)
Part C score: 25/25

Score: 80/80
```

Part C ends, and this lab ends.

Challenge 8

I implement a not-looping `ipc_send()`.

- `ipc_send()`

The goal is not to let `ipc_send()` loops, so its code is:

```
    if (pg == NULL) {
        // Set a value > UTOP
        pg = (void *)-1;
    }
    if (sys_ipc_try_send(to_env, val, pg, perm) < 0) {
        panic("ipc_send failed!");
    }
```

Before this challenge, the receiver always blocks itself and the sender always loops for a blocked receiver. I want to design a mechanism that we block the environment who calls `ipc_send()` or `ipc_recv()` first. Later another environment will call `ipc_recv()` or `ipc_send()`, and this environment need not block itself or loop because he knows that his partner is ready.

The sender can judge whether the receiver is ready by receiver's `env_ipc_recving` field. The receiver needs another field in `struct Env` to judge this, and it also needs to handle multiple senders. Thus I design a field `envid_t env_ipc_sending_list[10]` in `struct Env` to be the list of senders. **The number 10 is the max number of senders that can be added to this list.**

- `struct Env`

Notice that when sending a message, the fields `uint32_t env_ipc_value`, `void *env_ipc_dstva` and `int env_ipc_perm` is useless. So we can store the messages to be sent in these fields. The modified parts in `struct Env` is :

```
    // Lab 4 IPC (Add Challenge 8)
    bool env_ipc_recving;             // Env is blocked receiving
    envid_t env_ipc_sending_list[10];   // List of envid of senders
    void *env_ipc_va;                 // Virtual address
    uint32_t env_ipc_value;           // Data value
    envid_t env_ipc_from;             // envid of the sender
    int env_ipc_perm;                 // Perm of page mapping
```

- `sys_ipc_try_send()`

This function firstly check restrictions. Then if the receiver is blocked, send the value (and page). If not, store those messages in its own fields, add its `envid` to receiver's `env_ipc_sending_list`, mark itself as `ENV_NOT_RUNNABLE` and give up the CPU.

```
    // LAB 4: Your code here.
    struct Env *e;
    pte_t *pte;
    struct PageInfo *pp = NULL;

    if (envid2env(envid, &e, 0) < 0) {
        // Environment envid doesn't currently exist
        return -E_BAD_ENV;
    }
    if ((uint32_t)srcva < UTOP) {
        // srcva < UTOP, want to send a page
        if (PGOFF(srcva) != 0) {
            // srcva is not page-aligned
            return -E_INVAL;
        }
        if ((~perm & (PTE_U | PTE_P)) != 0 ||
            (perm & ~(PTE_U | PTE_P | PTE_AVAIL | PTE_W)) != 0) {
            // Perm is inappropriate
            return -E_INVAL;
        }
        if ((pp = page_lookup(curenv->env_pgdir, srcva, &pte)) == NULL) {
            // srcva is not mapped in the caller's address space
            return -E_INVAL;
        }
        if ((perm & PTE_W) == 1 && (*pte & PTE_W) == 0) {
            // Read-only
            return -E_INVAL;
        }
    }
    if (e->env_ipc_recving != 1) {
        // envid is not currently blocked in sys_ipc_recv
        if (e->env_ipc_sending_list[10 - 1] != 0) {
            panic("Full sending list!");
        }
        for (int i = 0; i < 10; ++ i) {
            if (e->env_ipc_sending_list[i] == 0) {
                e->env_ipc_sending_list[i] = curenv->env_id;
                break;
            }
        }
        curenv->env_ipc_value = value;
        if ((uint32_t)srcva < UTOP) {
            curenv->env_ipc_va = srcva;
            curenv->env_ipc_perm = perm;
        }
        curenv->env_status = ENV_NOT_RUNNABLE;

        sys_yield();
        return 0;
    }

    // Send the page
    if ((uint32_t)e->env_ipc_va < UTOP) {
        // Want to receive a page
        if (page_insert(e->env_pgdir, pp, e->env_ipc_va, perm) < 0) {
            // There's not enough memory to map srcva in envid's address space
            return -E_NO_MEM;
        } else {
            // Succeeded!
            e->env_ipc_perm = perm;
        }
    }

    // Send the value
    e->env_ipc_recving = 0;
    e->env_ipc_value = value;
    e->env_ipc_from = curenv->env_id;
    e->env_status = ENV_RUNNABLE;
    e->env_tf.tf_regs.reg_eax = 0;

    return 0;
```

- `sys_ipc_recv()`

The receiver firstly check the restrictions. Then if the sender is not ready, block itself and give up the CPU. If the sender is ready, it store the messages from sender's fields, pop the `envid` of sender out of `env_ipc_sending_list[0]` and move the other `envid`s backward. Finally it mark the sender as `ENV_RUNNABLE` and set the return value of the sender.

```
    if ((uint32_t)dstva < UTOP && PGOFF(dstva) != 0) {
        return -E_INVAL;
    }
    if (curenv->env_ipc_sending_list[0] != 0) {
        struct Env *e;
        if (envid2env(curenv->env_ipc_sending_list[0], &e, 0) < 0) {
            // Environment envid doesn't currently exist
            return -E_BAD_ENV;
        }
        curenv->env_ipc_value = e->env_ipc_value;

        curenv->env_ipc_from = curenv->env_ipc_sending_list[0];
        if ((uint32_t)dstva < UTOP && (uint32_t)e->env_ipc_va < UTOP) {
            if (page_insert(curenv->env_pgdir, page_lookup(e->env_pgdir, e->env_ipc_va, NULL), dstva, e-
>env_ipc_perm) < 0) {
                // There's not enough memory to map srcva in envid's address space
                return -E_NO_MEM;
            } else {
                // Succeeded!
                curenv->env_ipc_perm = e->env_ipc_perm;
            }
        }
        for (int i = 0; i < 10 - 1; ++ i) {
            curenv->env_ipc_sending_list[i] = curenv->env_ipc_sending_list[i + 1];
        }
        e->env_status = ENV_RUNNABLE;
        e->env_tf.tf_regs.reg_eax = 0;

        return 0;
    }
    curenv->env_ipc_recving = 1;
    curenv->env_ipc_va = dstva;
    curenv->env_status = ENV_NOT_RUNNABLE;

    sys_yield();
    return 0;
```

Then `make grade`:

```
dumbfork: OK (0.9s)
Part A score: 5/5

faultread: OK (0.9s)
faultwrite: OK (0.9s)
faultdie: OK (1.2s)
faultregs: OK (0.8s)
faultalloc: OK (1.0s)
faultallocbad: OK (2.1s)
faultnostack: OK (0.8s)
faultbadhandler: OK (2.1s)
faultevilhandler: OK (1.0s)
forktree: OK (2.0s)
Part B score: 50/50

spin: OK (2.0s)
stresssched: OK (2.4s)
sendpage: OK (0.6s)
pingpong: OK (1.9s)
primes: OK (3.8s)
Part C score: 25/25

Score: 80/80
```