Created: May 7 2018 12:31:00 Last Updated: May 21 2018 00:12:30

Lab 4 Report

Shuyang Cao

May 21 2018

Multiprocessor Support and Cooperative Multitasking

Multiprocessor Support

JOS supports “symmetric multiprocessing”(SMP), a multiprocessor model in which all CPUs have equivalent access to system resources such as memory and I/O buses. While all CPUs are functionally identical in SMP, during the boot proces they can be classified into two types: the bootstrap processor (BSP) is responsible for initializing the system and for booting the operating system; and the application processors (APs) are activated by the BSP only after the operating system is up and running. Which processor is the BSP is determined by the hardware and the BIOS.

In an SMP system, each CPU has an accompanying local APIC (LAPIC) uint. The LAPIC units are responsible for delivering interrupts through the system. The LAPIC also provides its connected CPU with a unique identifier. A processor accesses its LAPIC using memory-mapped I/O (MMIO). In MMIO, a portion of physical memory is hardwired to the registers of some I/O devices, so the same load/store instructions typically used to access memory can be used to access device registers. You also need to remember to set the mode of corresponding memory as `Write-through` in your page table.

Application Processor Bootstrap

Before booting up APs, the BSP should first collect information about the multiprocessor system, such as the total number of CPUs, their APIC IDs, and the MMIO address of the LAPIC unit. APs start in real mode, much like how the bootloader started in `boot\boot.S`. BSP copies the AP entry code to a memory location that is addressable in the real mode. Unlike with the bootloader, BSP has some control over where the AP will start executing code. In JOS we copy the entry code to `0x7000`, but any unused, page-aligned physical address below 640KB would work. BSP acitvates APs one after another, by sending `STARTUP IPIs` to the LAPIC unit of the corresponding AP, along with an initial CS:IP address at which the AP should start running its entry code. BSP waits for the AP to signal a `CPU_STARTED` flag in cpu_status field of its struct CpuInfo before going on to wake up the next one.

Exercise 1 & 2

> 1. Implement `mmio_map_region` in `kern/pmap.c`. To see how this is used, look at the beginning of `lapic_init` in `kern/lapic.c`. You'll have to do the next exercise, too, before the tests for mmio_map_region will run.
> 2. Read `boot_aps()` and `mp_main()` in `kern/init.c`, and the assembly code in `kern/mpentry.S`. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of `page_init()` in kern/pmap.c to avoid adding the page at `MPENTRY_PADDR` to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated `check_page_free_list()` test (but might fail the updated `check_kern_pgdir()` test, which we will fix soon).

`mmio_map_region` is only a wrapper of `boot_map_region`. It uses a static variable to store the last address it allocated, which should be updated each time it is invoked. What we need to add in `page_init()` is another condition check to avoid adding the page at `MPENTRY_PADDR` to the free list.

The codes for Exercise 1 & 2 is list below.

```diff
diff --git a/kern/pmap.c b/kern/pmap.c
index e4f7cce..db0fad9 100644
--- a/kern/pmap.c
+++ b/kern/pmap.c
@@ -350,7 +350,7 @@ page_init(void)
          *                            |    VGA Display     |
          *       0x000A0000(640KB) -> +--------------------+
          *                            |                    |
-        *                            |     Low Memory     |
+        *                            |     Low Memory     | <----- page at MPENTRY_PADDR is used
          *                            |                    |
          *       0x00001000(4KB) ---> |  ~~~~~~~~~~~~~~~~~ |
          *                            |   page 0 is used   |
@@ -399,33 +399,25 @@ page_init(void)
          *
          */

+
        page_free_list = NULL;

        pages[0].pp_ref = 1;            // mark physical page 0 as in use
-       size_t i;
-
-       for (i = 1; i < npages_basemem; ++i) {
-               // The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
-               // is free.
-               pages[i].pp_ref = 0;
-               pages[i].pp_link = page_free_list;
-               page_free_list = &pages[i];
-       }
-
-       size_t kernel_end_page = ((uint32_t)(boot_alloc(0)) - KERNBASE)/PGSIZE;
-       for (; i < kernel_end_page; ++i) {
-               // IO hole and kernel physical memory reside consecutively
-               // in physcial memory. So initialize their corresponding
-               // page entries in one loop
-               pages[i].pp_ref = 1;
-               pages[i].pp_link = NULL;
-       }
-
-       for (; i < npages; ++i) {
-               // The rest of extended memory are not used
-               pages[i].pp_ref = 0;
-               pages[i].pp_link = page_free_list;
-               page_free_list = &pages[i];
+       pages[0].pp_link = NULL;
+
+       // pege in [left_i, right_i) are used.
+       size_t left_i = PGNUM(IOPHYSMEM);
+       size_t right_i = ((uint32_t)(boot_alloc(0)) - KERNBASE)/PGSIZE;
+
+       for (size_t i = 1; i < npages; ++i) {
+               if ((i < left_i || i > right_i) && i != PGNUM(MPENTRY_PADDR)) {
+                       pages[i].pp_ref = 0;
+                       pages[i].pp_link = page_free_list;
+                       page_free_list = &pages[i];
+               } else {
+                       pages[i].pp_ref = 1;
+                       pages[i].pp_link = NULL;
+               }
        }
 }

@@ -723,7 +715,17 @@ mmio_map_region(physaddr_t pa, size_t size)
        // Hint: The staff solution uses boot_map_region.
        //
        // Your code here:
-       panic("mmio_map_region not implemented");
+       // panic("mmio_map_region not implemented");
+
+       size = ROUNDUP(size, PGSIZE);
+       uintptr_t ret = base;
+       base += size;
+
+       if (base > MMIOLIM)
+               panic("Reservation overflows MMIOLIM.\n");
+
```

```
+       boot_map_region(kern_pgdir, ret, size, pa, PTE_PCD | PTE_PWT | PTE_W);
+       return (void *)ret;
 }

 static uintptr_t user_mem_check_addr;
```

Answer to Q1

> 1. 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`?

`MPBOOTPHYS` is used to calculate absolute addresses of symbols in `kern/mpentry.S`. `boot/boot.S` is linked at `0x7C00` so it doesn't need it. But `kern/mpentry.S` is compiled as a part of kernel and doesn't begin where it is linked thus it would load `gdt` from a very high address before the page table is set up, resulting in APs boot failure.

Per-CPU State and Initialization

When writing a multiprocessor OS, it is important to distinguish between per-CPU state that is private to each processor, and global state that the whole system shares. In JOs, we should be aware of the following per-CPU states.

- Per-CPU kernel stack
  Because multiple CPUs can trap into the kernel simultaneously, we need a separate kernel stack for each processor to prevent them from interfering with each other's execution.
- Per-CPU TSS and TSS descriptor
  A per-CPU task state segment (TSS) is also needed in order to specify where each CPU's kernel stack lives. The TSS for CPU i is stored in cpus[i].cpu_ts, and the corresponding TSS descriptor is defined in the GDT entry gdt[(GD_TSS0 >> 3) + i].
- Per-CPU current environment pointer
  Since each CPU can run different user process simultaneously, we redefined the symbol `curenv` to refer to `cpus[cpunum()].cpu_env` (or `thiscpu->cpu_env`), which points to the environment currently executing on the current CPU (the CPU on which the code is running).
- Per-CPU system registers
  All registers, including system registers, are private to a CPU. Therefore, instructions that initialize these registers, such as `lcr3()`, `ltr()`, `lgdt()`, `lidt()`, etc., must be executed once on each CPU. Functions `env_init_percpu()` and `trap_init_percpu()` are defined for this purpose.

Exercise 3 & 4

> 3. Modify `mem_init_mp()` (in `kern/pmap.c`) to map per-CPU stacks starting at `KSTACKTOP`, as shown in `inc/memlayout.h`. The size of each stack is `KSTKSIZE` bytes plus `KSTKGAP` bytes of unmapped guard pages. Your code should pass the new check in `check_kern_pgdir()`.
> 4. The code in `trap_init_percpu()` (`kern/trap.c`) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global ts variable any more.)

`mem_init_mp()` is another wrapper of `boot_map_region`. The memory layout of all kernel stacks manifested in `inc/memlayout.h`. We only need a loop to setup those page table entries. In `trap_init_percpu()`, our main work is to replace the former global `ts` variable with `thiscpu->cpu_ts`. The hint there also requires us to assign a correct value to `ts_iomb` to prevent unauthorized environments from doing IO. In JOS, we don't use IO map. By referring to ***Chapter 8.3.2 of INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986***, we assign `sizeof(struct Taskstate)` to `ts_iomb` to disable IO map.

The codes for these two exercises are shown below.

```
diff --git a/kern/pmap.c b/kern/pmap.c
index db0fad9..047f266 100644
--- a/kern/pmap.c
+++ b/kern/pmap.c
@@ -281,6 +281,10 @@ mem_init_mp(void)
        //
        // LAB 4: Your code here:

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

 // --------------------------------------------------------------
diff --git a/kern/trap.c b/kern/trap.c
index 732603d..e9ae70c 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -122,18 +122,24 @@ trap_init_percpu(void)

        // Setup a TSS so that we get the right stack
        // when we trap to the kernel.
-       ts.ts_esp0 = KSTACKTOP;
-       ts.ts_ss0 = GD_KD;
-       ts.ts_iomb = sizeof(struct Taskstate);
+       thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpunum() * (KSTKSIZE + KSTKGAP);
+       thiscpu->cpu_ts.ts_ss0 = GD_KD;
+       // We'll set the limit of TSS0 segment to be sizeof(struct Taskstate)-1.
+       // So we set ts_iomb to be sizeof(struct Taskstate). It's larger than the
+       // limit of TSS0 segment so that I/O permission bit map is disabled and only
+       // all I/O instructions in the 80386 program cause exceptions when CPL > IOPL.
+       // Refer to Chapter 8.3.2 of INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986
+       // for more details
+       thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);

        // Initialize the TSS slot of the gdt.
-       gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
-                                       sizeof(struct Taskstate) - 1, 0);
-       gdt[GD_TSS0 >> 3].sd_s = 0;
+       gdt[(GD_TSS0 >> 3) + cpunum()] = SEG16(STS_T32A, (uint32_t)&(thiscpu->cpu_ts),
+                                                               sizeof(struct Taskstate) - 1, 0);
+       gdt[(GD_TSS0 >> 3) + cpunum()].sd_s = 0;

        // Load the TSS selector (like other segment selectors, the
        // bottom three bits are special; we leave them 0)
-       ltr(GD_TSS0);
+       ltr(GD_TSS0 + (cpunum() << 3));

        // Load the IDT
        lidt(&idt_pd);
```

Locking

Some resources in kernel need exclusive access. While multiple CPUs may run kernel code simultaneously, we need lock to guarantee those resource are access to exclusively. In JOS, we use the simplest way for this goal. We add a big lock on the whole kernel. Only one AP is able to run kernel code at one time.

Exercise 5

> 5. Apply the big kernel lock as described above, by calling lock_kernel() and unlock_kernel() at the proper locations.

In JOS, a CPU enters the kernel during initialization or after trap and exits the kernel after scheduling. We only need to do locking as the first step after entering JOS kernel and do releasing the lock as the last step before exiting JOS kernel. Since there is only one CPU executing code before BSP boot APs, we don’t need to lock the kernel during initialization until we begin to boot APs. And booting APs is the last step of JOS boot-up initialization.

The code for this exercise is shown below.

```
diff --git a/kern/env.c b/kern/env.c
index d26e999..c543bf7 100644
--- a/kern/env.c
+++ b/kern/env.c
@@ -561,7 +561,7 @@ env_run(struct Env *e)
        curenv->env_status = ENV_RUNNING;
        ++(curenv->env_runs);
        lcr3(PADDR(e->env_pgdir));
-
+       unlock_kernel();
        env_pop_tf(&(curenv->env_tf));

        // panic("env_run not yet implemented");
diff --git a/kern/init.c b/kern/init.c
index 61da49e..4e8f727 100644
--- a/kern/init.c
+++ b/kern/init.c
@@ -50,6 +50,7 @@ i386_init(void)

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

        // Starting non-boot CPUs
        boot_aps();

@@ -116,9 +119,12 @@ mp_main(void)
        // only one CPU can enter the scheduler at a time!
        //
        // Your code here:
+       lock_kernel();
+
+       sched_yield();

        // Remove this after you finish Exercise 6
-       for (;;);
+       // for (;;);
 }

 /*

diff --git a/kern/trap.c b/kern/trap.c
index e9ae70c..7d96fb8 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -261,6 +261,7 @@ trap(struct Trapframe *tf)
                // Acquire the big kernel lock before doing any
                // serious kernel work.
                // LAB 4: Your code here.
+               lock_kernel();
                assert(curenv);

                // Garbage collect if current enviroment is a zombie
```

Answer to Q2

> 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.

1. CPUs will automatically push registers on kernel stack before entering kernel and invoking trap handler. So if different CPUs share a stack and there is a CPU running JOS kernel, another CPU will corrupt the kernel stack by pushing things on the stack before it is blocked by the kernel lock.
2. JOS kernel is an external kernel. In some cases the kernel stack is used even when CPU is not running in kernel mode. For example, a part of work of page fault handler is done by user programs. If CPUs share the kernel stack, the stack of the page fault handler of user program will be easily corrupted.

Round-Robin Scheduling & Exercise 6

JOS implements a simple scheduling algorithm, round-robin scheduling. It schedules processes in a FIFO style. To finish exercise 6, we must finish `sched_yield()` and the system call for it.

> 6. Implement round-robin scheduling in `sched_yield()` as described above. Don’t forget to modify `syscall()` to dispatch `sys_yield()`.
>   Make sure to invoke `sched_yield()` in `mp_main`.
>   Modify `kern/init.c` to create three (or more!) environments that all run the program `user/yield.c`.

There are two hints for `sched_yield()`. The first is that a CPU may be idle before so cur_env points to nothing. The other is that the loop search may failed to find another process to run. So remember to continue to run current process. In addition, because of hint 1, the current process may not exist.

The code for is exercise 6 is shown below.

```
diff --git a/kern/init.c b/kern/init.c
index 61da49e..4e8f727 100644
--- a/kern/init.c
+++ b/kern/init.c
@@ -50,6 +50,7 @@ i386_init(void)

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

        // Starting non-boot CPUs
        boot_aps();
@@ -59,7 +60,9 @@ i386_init(void)
        ENV_CREATE(TEST, ENV_TYPE_USER);
 #else
        // 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);
 #endif // TEST*

        // Schedule and run the first user environment!
@@ -116,9 +119,12 @@ mp_main(void)
        // only one CPU can enter the scheduler at a time!
        //
        // Your code here:
+       lock_kernel();
+
+       sched_yield();

        // Remove this after you finish Exercise 6
-       for (;;);
+       // for (;;);
 }

 /*
diff --git a/kern/sched.c b/kern/sched.c
index f595bb1..32cbb47 100644
--- a/kern/sched.c
+++ b/kern/sched.c
@@ -29,6 +29,18 @@ sched_yield(void)
        // below to halt the cpu.

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

        // sched_halt never returns
        sched_halt();
diff --git a/kern/syscall.c b/kern/syscall.c
index d1a6536..7e15c57 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -279,6 +279,7 @@ syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
                case SYS_cgetc: return (int32_t)sys_cgetc();
                case SYS_getenvid: return (int32_t)sys_getenvid();
                case SYS_env_destroy: return (int32_t)sys_env_destroy((envid_t)a1);
+               case SYS_yield: sys_yield(); return 0;
                default:
                        return -E_INVAL;
        }
```

Answer to Q3 & Q4

> 3. 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?

`e` points to a place in kernel memory space. Each user program has a complete copy of kernel page table, i.e., the content of kernel page table is a part of user program page table.

> 4. 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?

1. Process switching shoulbe be transparent to user program. So we need to save and restore registers while scheduling so that user program resumes at where it's paused and it won't feel it.
2. Scheduling is implemented using trap mechanism. It is a routine for trap mechanism to save registers when user program is trapped and restore registers when returning to user space.

System Calls for Environment Creation

In this part we add a new function to JOS kernel, the ability to create new process. We also need to add new system calls to provide the interfaces for user program. JOS doesn't offer a "create-new-process" system call for this function. Instead, JOS offer a bunch of system calls for creating a new process and let user programs control the flow of creating new processes. In other words, we implemet a Unix-like `fork()` entirely in user space.

System calls needed are listed below.

- *sys_exofork*
  This system call creates a new environment with an almost blank slate: nothing is mapped in the user portion of its address space, and it is not runnable. The new environment will have the same register state as the parent environment at the time of the `sys_exofork` call. In the parent, `sys_exofork` will return the `envid_t` of the newly created environment (or a negative error code if the environment allocation failed). In the child, however, it will return 0. (Since the child starts out marked as not runnable, `sys_exofork` will not actually return in the child until the parent has explicitly allowed this by marking the child runnable).
- *sys_env_set_status*
  Sets the status of a specified environment to ENV_RUNNABLE or ENV_NOT_RUNNABLE. This system call is typically used to mark a new environment ready to run, once its address space and register state has been fully initialized.
- *sys_page_alloc*:
  Allocates a page of physical memory and maps it at a given virtual address in a given environment's address space.
- *sys_page_map*
  Copy a page mappings from one environment's address space to another, leaving a memory sharing arrangement in place so that the new and the old mappings both refer to the same page of physical memory.
- *sys_page_unmap*
  Unmap a page mapped at a given virtual address in a given environment.

For all of the system calls above that accept environment IDs, the JOS kernel supports the convention that a value of 0 means "the current environment." This convention is implemented by `envid2env()` in `kern/env.c`.

Exercise 7

> 7. Implement the system calls described above in `kern/syscall.c` and make sure `syscall()` calls them. You will need to use various functions in `kern/pmap.c` and `kern/env.c`, particularly `envid2env()`. For now, whenever you call `envid2env()`, pass 1 in the checkperm parameter. Be sure you check for any invalid system call arguments, returning `-E_INVAL` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

The work for this exercise is a little bit tedious. As long as you fulfill all requirements in hints, it's easy to finish this exercise.

The code for exercise 7 is shown below.

```diff
diff --git a/kern/init.c b/kern/init.c
index 4e8f727..5b5b50a 100644
--- a/kern/init.c
+++ b/kern/init.c
@@ -60,9 +60,7 @@ i386_init(void)
        ENV_CREATE(TEST, ENV_TYPE_USER);
 #else
        // Touch all you want.
-       ENV_CREATE(user_yield, ENV_TYPE_USER);
-       ENV_CREATE(user_yield, ENV_TYPE_USER);
-       ENV_CREATE(user_yield, ENV_TYPE_USER);
+       ENV_CREATE(user_dumbfork, ENV_TYPE_USER);
 #endif // TEST*

        // Schedule and run the first user environment!
diff --git a/kern/syscall.c b/kern/syscall.c
index 7e15c57..222a58a 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -85,7 +85,18 @@ sys_exofork(void)
        // will appear to return 0.

        // LAB 4: Your code here.
-       panic("sys_exofork not implemented");
+       // panic("sys_exofork not implemented");
+       struct Env *child_env;
+       int err_code = env_alloc(&child_env, curenv->env_id);
+
+       if (err_code < 0)
+               return err_code;
+
+       child_env->env_status = ENV_NOT_RUNNABLE;
+       child_env->env_tf = curenv->env_tf;
+       child_env->env_tf.tf_regs.reg_eax = 0;
+
+       return child_env->env_id;
 }

 // Set envid's env_status to status, which must be ENV_RUNNABLE
@@ -105,7 +116,20 @@ sys_env_set_status(envid_t envid, int status)
        // envid's status.

        // LAB 4: Your code here.
-       panic("sys_env_set_status not implemented");
+       // panic("sys_env_set_status not implemented");
+       if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE) {
+               return -E_INVAL;
+       }
+
+       struct Env *e;
+       if (envid2env(envid, &e, true) < 0) {
+               return -E_BAD_ENV;
+       }
+
+
+       e->env_status = status;
+
+       return 0;
 }

 // Set the page fault upcall for 'envid' by modifying the corresponding struct
@@ -150,7 +174,26 @@ sys_page_alloc(envid_t envid, void *va, int perm)
        //   allocated!

        // LAB 4: Your code here.
-       panic("sys_page_alloc not implemented");
+       // panic("sys_page_alloc not implemented");
+       struct Env *e;
+
+       if (envid2env(envid, &e, true) < 0)
+               return -E_BAD_ENV;
+
+       if ((uintptr_t)va >= UTOP || (uintptr_t)va % PGSIZE ||
+               perm & ~PTE_SYSCALL || ~perm & (PTE_U | PTE_P))
+               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) {
+               page_free(pp);
+               return -E_NO_MEM;
+       }
+
+       return 0;
 }

 // Map the page of memory at 'srcva' in srcenvid's address space
@@ -181,7 +224,24 @@ sys_page_map(envid_t srcenvid, void *srcva,
        //   check the current permissions on the page.

        // LAB 4: Your code here.
-       panic("sys_page_map not implemented");
+       // panic("sys_page_map not implemented");
+       struct Env *srcenv, *dstenv;
+       if (envid2env(srcenvid, &srcenv, true) < 0 || envid2env(dstenvid, &dstenv, true) < 0)
+               return -E_BAD_ENV;
+
+       pte_t *pte_ptr;
+       struct PageInfo *pp;
+       if ((uintptr_t)srcva >= UTOP || (uintptr_t)srcva % PGSIZE ||
+               (uintptr_t)dstva >= UTOP || (uintptr_t)dstva % PGSIZE ||
+               (pp = page_lookup(srcenv->env_pgdir, srcva, &pte_ptr)) == NULL ||
+               perm & ~PTE_SYSCALL || ~perm & (PTE_U | PTE_P) ||
+               ((perm & PTE_W) && !((*pte_ptr) & PTE_W)))
+       return -E_INVAL;
+
+       if (page_insert(dstenv->env_pgdir, pp, dstva, perm) < 0)
+               return -E_NO_MEM;
+
+       return 0;
 }

 // Unmap the page of memory at 'va' in the address space of 'envid'.
@@ -197,7 +257,17 @@ sys_page_unmap(envid_t envid, void *va)
        // Hint: This function is a wrapper around page_remove().

        // LAB 4: Your code here.
-       panic("sys_page_unmap not implemented");
+       // panic("sys_page_unmap not implemented");
+       struct Env *e;
+
+       if (envid2env(envid, &e, true) < 0)
+               return -E_BAD_ENV;
+
+       if ((uintptr_t)va >= UTOP || (uintptr_t)va % PGSIZE)
+               return -E_INVAL;
+
+       page_remove(e->env_pgdir, va);
+       return 0;
 }

 // Try to send 'value' to the target env 'envid'.
@@ -275,11 +345,17 @@ syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
        // panic("syscall not implemented");

        switch (syscallno) {
-               case SYS_cputs: sys_cputs((const char *)a1, a2); return 0;
+               case SYS_cputs: sys_cputs((const char *)a1, a2); return 0;
                case SYS_cgetc: return (int32_t)sys_cgetc();
-               case SYS_getenvid: return (int32_t)sys_getenvid();
-               case SYS_env_destroy: return (int32_t)sys_env_destroy((envid_t)a1);
+               case SYS_getenvid:      return (int32_t)sys_getenvid();
+               case SYS_env_destroy:   return (int32_t)sys_env_destroy((envid_t)a1);
+               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);
+               case SYS_exofork:       return sys_exofork();
+               case SYS_env_set_status:        return sys_env_set_status((envid_t)a1, (int)a2);
                case SYS_yield: sys_yield(); return 0;
+
                default:
                    return -E_INVAL;
        }
```

Copy-on-Write Fork

`dumbfork` copy page contents from parents to children, which is not a good behaviour in most cases. In this part we will implement a “proper” Unix-like `fork()` with copy-on-write, as a user space library routine. Implementing `fork()` and copy-on-write support in user space has the benefit that the kernel remains much simpler and thus more likely to be correct. It also lets individual user-mode programs define their own semantics for `fork()` . A program that wants a slightly different implementation (for example, the expensive always-copy version like `dumbfork()` , or one in which the parent and child actually share memory afterward) can easily provide its own.

User Space Page Handler

A user-level copy-on-write `fork()` needs to know about page faults on write-protected pages. Copy-on-write is only one of many possible uses for user-level page fault handling. It’s common to set up an address space so that page faults indicate when some action needs to take place. A typical Unix kernel must keep track of what action to take when a page fault occurs in each region of a process’s space. There is a lot of information for the kernel to keep track of. Instead of taking the traditional Unix approach, JOS decide what to do about each page fault in user space, where bugs are less damaging. This design has the added benefit of allowing programs great flexibility in defining their memory regions.

In order to handle its own page faults, a user environment will need to register a page fault handler entrypoint with the JOS kernel. The user environment registers its page fault entrypoint via the new `sys_env_set_pgfault_upcall` system call. We have added a new member to the `Env structure` , `env_pgfault_upcall` , to record this information.

During normal execution, a user environment in JOS will run on the normal user stack: its `ESP` register starts out pointing at `USTACKTOP` , and the stack data it pushes resides on the page between `USTACKTOP-PGSIZE` and `USTACKTOP-1` inclusive. When a page fault occurs in user mode, however, the kernel will restart the user environment running a designated user-level page fault handler on a different stack, namely the user exception stack. In essence, we will make the JOS kernel implement automatic “stack switching” on behalf of the user environment, in much the same way that the x86 processor already implements stack switching on behalf of JOS when transferring from user mode to kernel mode!

The JOS user exception stack is also one page in size, and its top is defined to be at virtual address `UXSTACKTOP` , so the valid bytes of the user exception stack are from `UXSTACKTOP-PGSIZE` through `UXSTACKTOP-1` inclusive. While running on this exception stack, the user-level page fault handler can use JOS’s regular system calls to map new pages or adjust mappings so as to fix whatever problem originally caused the page fault. Then the user-level page fault handler returns, via an assembly language stub, to the faulting code on the original stack. Each user environment that wants to support user-level page fault handling will need to allocate memory for its own exception stack.

We now change the page fault handling code in `kern/trap.c` to handle page faults from user mode as follows. We will call the state of the user environment at the time of the fault the trap-time state. If there is no page fault handler registered, the JOS kernel destroys the user environment with a message as before. Otherwise, the kernel sets up a trap frame on the exception stack that looks like a `struct UTrapframe` from `inc/trap.h` .

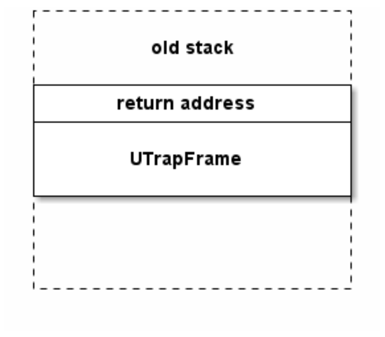
Multiple Page Fault & Return From User Space Kernel Handler

Since we handle page faults in user space, we lose the advantage of `iret` where CPU change stack for us automatically. Before we jump back to the normal routine of a user program, we must restore all registers before we jump back, i.e., we must implement the function of `iret` ourselves in user space.

To explain this procedure, we use *old stack* to refer to the stack used before current page fautl. So *old stack* may be a user stack or the exception stack when multiple page faults happen. At the end of our procedure, we use `ret` to return to the former routine. `ret` will pop out an integer to `eip` to jump back. Therefore, in fact, our task is to add a new integer on *old stack* and let `esp` point to the integer right before we call `ret` .

If there is only one page fault, *old stack* is the user stack. Since nothing exists below `utf_esp` , there is nothing to worry when we push an integer to *old stack*. The tricky part is how to handle them in the same manner when multiple page faults happen.

Because old values of registers are stored in UTrapFrame and we need registers to put return address on *old stack* and change the value of `utf_esp` , JOS kernel need to reserve 4 black bytes on exception stack when multiple page faults happen. After we finish setting up return address and `utf_esp` , we don’t use registers any more and pop from values from exception stack to registers. Finally we call `ret` .



Exercise 8 & 9 & 10 & 11

> 8. Implement the `sys_env_set_pgfault_upcall` system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a “dangerous” system call.
> 9. Implement the code in `page_fault_handler` in `kern/trap.c` required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack.
> 10. Implement the `_pgfault_upcall` routine in `lib/pfentry.S` . The interesting part is returning to the original point in the user code that caused the page fault. You’ll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the `EIP` .
> 11. Finish `set_pgfault_handler()` in `lib/pgfault.c` .

The process is destroyed if it runs out of exception stack.

The codes for these exericse are shown below.

```
diff --git a/kern/syscall.c b/kern/syscall.c
index 222a58a..fa0260b 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -144,7 +144,14 @@ static int
 sys_env_set_pgfault_upcall(envid_t envid, void *func)
 {
        // LAB 4: Your code here.
-       panic("sys_env_set_pgfault_upcall not implemented");
+       // panic("sys_env_set_pgfault_upcall not implemented");
+       struct Env *e;
+       if (envid2env(envid, &e, true) < 0)
+               return -E_BAD_ENV;
+
+       e->env_pgfault_upcall = func;
+
+       return 0;
 }

 // Allocate a page of memory and map it at 'va' with permission
@@ -354,6 +361,7 @@ syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
                case SYS_page_unmap:    return sys_page_unmap((envid_t)a1, (void *)a2);
                case SYS_exofork:       return sys_exofork();
                case SYS_env_set_status:        return sys_env_set_status((envid_t)a1, (int)a2);
+               case SYS_env_set_pgfault_upcall: return sys_env_set_pgfault_upcall((envid_t)a1, (void *)a2);
                case SYS_yield: sys_yield(); return 0;

                default:
diff --git a/kern/trap.c b/kern/trap.c
index 7d96fb8..40d04c5 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -346,6 +346,25 @@ page_fault_handler(struct Trapframe *tf)
        //   (the 'tf' variable points at 'curenv->env_tf').

        // LAB 4: Your code here.
+       if (curenv->env_pgfault_upcall != NULL) {
+               uintptr_t u_esp = (tf->tf_esp >= (UXSTACKTOP - PGSIZE) && tf->tf_esp <= UXSTACKTOP) ?
+                       tf->tf_esp - sizeof(uint32_t) - sizeof(struct UTrapframe) : UXSTACKTOP - sizeof(struct UTrapframe);
+               user_mem_assert(curenv, (void *)u_esp, sizeof(struct UTrapframe), PTE_U | PTE_W | PTE_P);
+
+               struct UTrapframe *utf = (struct UTrapframe *)u_esp;
+               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;
+
+               // tf is the same as &(curenv->env_tf)
+               tf->tf_esp = u_esp;
+               tf->tf_eip = (uintptr_t)(curenv->env_pgfault_upcall);
+
+               env_run(curenv);
+       }

        // Destroy the environment that caused the fault.
        cprintf("[%08x] user fault va %08x ip %08x\n",
diff --git a/lib/pfentry.S b/lib/pfentry.S
index f40aeeb..53d8df3 100644
--- a/lib/pfentry.S
+++ b/lib/pfentry.S
@@ -65,18 +65,31 @@ _pgfault_upcall:
        // ways as registers become unavailable as scratch space.
        //
        // LAB 4: Your code here.
+       // the size of UTrapframe is 13*4=52 bytes
+       movl 48(%esp), %eax             # move utf_esp to %eax
+       subl $4, %eax                   # %eax points to trap-time stack now, reserve 4 bytes for eip
+       movl %eax, 48(%esp)             # store trap-time stack pointer to utf_esp so that we can
+                                       # pop it to later when we want to set %esp but no register can be used.
+       movl 40(%esp), %ebx             # move utf_eip to %ebx
+       movl %ebx, (%eax)               # move utf_eip to trap-time stack

        // 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
+       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
+       popfl

        // Switch back to the adjusted trap-time stack.
        // LAB 4: Your code here.
+       popl %esp

        // Return to re-execute the instruction that faulted.
        // LAB 4: Your code here.
+       ret
diff --git a/lib/pgfault.c b/lib/pgfault.c
index a975518..fb68949 100644
--- a/lib/pgfault.c
+++ b/lib/pgfault.c
@@ -29,7 +29,12 @@ set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
        if (_pgfault_handler == 0) {
                // First time through!
                // LAB 4: Your code here.
-               panic("set_pgfault_handler not implemented");
+               // panic("set_pgfault_handler not implemented");
+               envid_t curenv_id = sys_getenvid();
+
+               if ((r = sys_page_alloc(curenv_id, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P)) < 0
+                       || (r = sys_env_set_pgfault_upcall(curenv_id, _pgfault_upcall)) < 0)
+                       panic("set_pgfault_handler: %e\n", r);
        }

        // Save handler pointer for assembly to call.
```

Implementing Copy-on-Write Fork

We now have the ability to implement copy-on-write fork. This task can be divided into two parts. Part one is to set copy-on-write bit in page table entries when forking a new process. Part two is to handle the page faults casued by writing to a copy-on-write page. Normally to handle a copy-on-write page fault includes allocating a new page, marking it as writable and copy the content to it from the copy-on-write page.

Exercise 12

> 12. Implement fork, duppage and pgfault in lib/fork.c.

The code for exericse 12 is shown below. Remember to clear the access bit and dirty bit in page table entries. Although it's antilogical, it's OK to do so since JOS never user these two bits to provide *swap* function.

```
diff --git a/kern/syscall.c b/kern/syscall.c
index fa0260b..d9aaab1 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -243,7 +243,7 @@ sys_page_map(envid_t srcenvid, void *srcva,
                (pp = page_lookup(srcenv->env_pgdir, srcva, &pte_ptr)) == NULL ||
                perm & ~PTE_SYSCALL || ~perm & (PTE_U | PTE_P) ||
                ((perm & PTE_W) && !((*pte_ptr) & PTE_W)))
-       return -E_INVAL;
+               return -E_INVAL;

        if (page_insert(dstenv->env_pgdir, pp, dstva, perm) < 0)
                return -E_NO_MEM;
diff --git a/lib/fork.c b/lib/fork.c
index 61264da..2e54f5f 100644
--- a/lib/fork.c
+++ b/lib/fork.c
@@ -25,6 +25,8 @@ pgfault(struct UTrapframe *utf)
        //   (see <inc/memlayout.h>).

        // LAB 4: Your code here.
+       if (!(err & FEC_WR && uvpt[PGNUM(addr)] & PTE_COW))
+               panic("pgfault failed.\n");

        // 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
@@ -33,8 +35,18 @@ pgfault(struct UTrapframe *utf)
        //   You should make three system calls.

        // LAB 4: Your code here.
+       if ((r = sys_page_alloc(0,(void *)PFTEMP, PTE_U | PTE_W | PTE_P)) < 0)
+               panic("Failed to allocate a new page: %e\n", r);

-       panic("pgfault not implemented");
+       memcpy((void *)PFTEMP, ROUNDDOWN(addr, PGSIZE), PGSIZE);
+
+       if ((r = sys_page_map(0, (void *)PFTEMP, 0, ROUNDDOWN(addr, PGSIZE), PTE_U | PTE_W | PTE_P)) < 0)
+               panic("Failed to map new page: %e\n", r);
+
+       if ((r = sys_page_unmap(0, (void *)PFTEMP)) < 0)
+               panic("Faile to unmap PFTEMP: %e\n", r);
+
+       // panic("pgfault not implemented");
 }

 //
@@ -54,7 +66,24 @@ duppage(envid_t envid, unsigned pn)
        int r;

        // LAB 4: Your code here.
-       panic("duppage not implemented");
+       // panic("duppage not implemented");
+       void *addr = (void *)(pn*PGSIZE);
+       int perm = PGOFF(uvpt[pn]) & PTE_SYSCALL;
+
+       if (perm & (PTE_W | PTE_COW)) {
+               perm |= PTE_COW;
+               perm &= ~PTE_W;
+
+               if ((r = sys_page_map(0, addr, envid, addr, perm)) < 0)
+                       panic("Failed to duppage on child enviroment: %e\n", r);
+
+               if ((r = sys_page_map(0, addr, 0, addr, perm)) < 0)
+                       panic("Failed to duppage on self enviroment: %e\n", r);
+       } else {
+               if ((r = sys_page_map(0, addr, envid, addr, perm)) < 0)
+                       panic("Failed to duppage on child enviroment: %e\n", r);
+       }
+
        return 0;
 }

@@ -77,8 +106,45 @@ duppage(envid_t envid, unsigned pn)
 envid_t
 fork(void)
 {
+       int r;
```

```
        // LAB 4: Your code here.
-       panic("fork not implemented");
+       // panic("fork not implemented");
+       set_pgfault_handler(pgfault);
+
+       // envid is different in parent and child process
+       envid_t envid = sys_exofork();
+       if (envid == 0) {       // if the process is the child process
+               thisenv = &envs[ENVX(sys_getenvid())];
+               return 0;
+       }
+
+       // copy address map
+       for (size_t i = 0; i < NPDENTRIES; ++i) {
+               if (uvpd[i] & PTE_P) {
+                       for (size_t j = 0; j < NPTENTRIES; ++j) {
+                               size_t pn = i * NPTENTRIES + j;
+
+                               if (pn == PGNUM(USTACKTOP))
+                                       goto COPY_END;
+
+                               if (uvpt[pn] & PTE_P)
+                                       duppage(envid, pn);
+                       }
+               }
+       }
+COPY_END:
+
+       if ((r = sys_page_alloc(envid, (void *)(UXSTACKTOP-PGSIZE), PTE_U | PTE_W | PTE_P)) < 0)
+               panic("Failed to allocate page for child's exception stack: %e", r);
+
+       extern void _pgfault_upcall(void);
+       if ((r = sys_env_set_pgfault_upcall(envid, _pgfault_upcall)) < 0)
+               panic("Failed to set page fault hanlder for child: %e\n", r);
+
+       if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
+               panic("Failed to set child's status as ENV_RUNNABLE", r);
+
+       return envid;
 }

 // Challenge!
diff --git a/lib/pgfault.c b/lib/pgfault.c
index fb68949..8d78846 100644
--- a/lib/pgfault.c
+++ b/lib/pgfault.c
@@ -30,10 +30,8 @@ set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
                // First time through!
                // LAB 4: Your code here.
                // panic("set_pgfault_handler not implemented");
-               envid_t curenv_id = sys_getenvid();
-
-               if ((r = sys_page_alloc(curenv_id, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P)) < 0
-                       || (r = sys_env_set_pgfault_upcall(curenv_id, _pgfault_upcall)) < 0)
+               if ((r = sys_page_alloc(0, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P)) < 0
+                       || (r = sys_env_set_pgfault_upcall(0, _pgfault_upcall)) < 0)
                        panic("set_pgfault_handler: %e\n", r);
        }
```

Preemptive Multitasking and Inter-Process communication (IPC)

Clock Interrupts and Preemption

In this part we implement preemptive round-robin scheduling. To do this, we must extend the JOS to allow the kernel to support external hardware interrupts from the clock hardware. External interrupts are referred to as IRQs. There are 16 possible IRQs in JOS, numbered 0 throug 15, mapped in IDT starting from `IRQ_OFFSET`. To simplify JOS, external device interrupts are always disabled when in the kernel and enabled when in user space. External interrupts are controlled by the `FL_IF` flag bit of the `%eflags` register (see `inc/mmu.h`). While the bit can be modified in several ways, because of our simplification, we will handle it solely through the process of saving and restoring %eflags register as we enter and leave user mode. Interrupts are masked with the very first instruction of the bootloader until we enter user space first time.

To implement preemptive round-robin scheduling, we need to handle clock interrupts. We program the hardware to generate clock interrupts periodically, which will for control back to the kernel where we can re-schedule processes. We should utilize `lapic_init` and `pic_init` (from `i386_init` in `init.c`) to help us program the hardware.

Exercise 13 & 14

> 13. Exercise 13. Modify `kern/trapentry.S` and `kern/trap.c` to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in `env_alloc()` in `kern/env.c` to ensure that user environments are always run with interrupts enabled.
>   Also uncomment the `sti` instruction in `sched_halt()` so that idle CPUs unmask interrupts.
> 14. 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.

Remember to invoke `lapic_eoi()` before your call `sched_yield` when you disptach timer interrupt. Otherwire, the hardware won't generate another timer interrupt again.

The codes for exercise 13 & 14 are shown below.

```
diff --git a/kern/env.c b/kern/env.c
index c543bf7..af7a67f 100644
--- a/kern/env.c
+++ b/kern/env.c
@@ -271,6 +271,7 @@ env_alloc(struct Env **newenv_store, envid_t parent_id)

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

        // Clear the page fault handler until user installs one.
        e->env_pgfault_upcall = 0;
diff --git a/kern/init.c b/kern/init.c
index 5b5b50a..be3c446 100644
--- a/kern/init.c
+++ b/kern/init.c
@@ -60,7 +60,9 @@ i386_init(void)
        ENV_CREATE(TEST, ENV_TYPE_USER);
 #else
        // Touch all you want.
-       ENV_CREATE(user_dumbfork, ENV_TYPE_USER);
+       ENV_CREATE(user_spin, ENV_TYPE_USER);
+       ENV_CREATE(user_spin, ENV_TYPE_USER);
+       ENV_CREATE(user_spin, ENV_TYPE_USER);
 #endif // TEST*

        // Schedule and run the first user environment!
diff --git a/kern/sched.c b/kern/sched.c
index 32cbb47..505bbac 100644
--- a/kern/sched.c
+++ b/kern/sched.c
@@ -87,7 +87,7 @@ sched_halt(void)
                "pushl $0\n"
                "pushl $0\n"
                // Uncomment the following line after completing exercise 13
-               //"sti\n"
+               "sti\n"
                "1:\n"
                "hlt\n"
                "jmp 1b\n"
diff --git a/kern/trap.c b/kern/trap.c
index 40d04c5..e14fee5 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -215,6 +215,9 @@ trap_dispatch(struct Trapframe *tf)
                                          return;
                case T_PGFLT: page_fault_handler(tf);
                                          return;
+               case IRQ_OFFSET + IRQ_TIMER: lapic_eoi();
+                                                                  sched_yield();
+                                                                  return;
                case T_SYSCALL: tf->tf_regs.reg_eax = syscall(
                                                                 tf->tf_regs.reg_eax,
                                                                 tf->tf_regs.reg_edx,
diff --git a/kern/trapentry.S b/kern/trapentry.S
index aa4ed86..5576132 100644
--- a/kern/trapentry.S
+++ b/kern/trapentry.S
@@ -67,7 +67,13 @@
        TRAPHANDLER(trap_alignment_check, T_ALIGN)                                              # 17    alignment check
        TRAPHANDLER_NOEC(trap_machine_check, T_MCHK)                                    # 18    machine check
        TRAPHANDLER_NOEC(trap_SIMD_floating_point_error, T_SIMDERR)             # 19    SIMD floating point error
+       TRAPHANDLER_NOEC(trap_timer, IRQ_OFFSET+IRQ_TIMER)                              # 32    timer interrupt
+       TRAPHANDLER_NOEC(trap_keyboard, IRQ_OFFSET+IRQ_KBD)                             # 33    keyboard interrupt
+       TRAPHANDLER_NOEC(trap_serial_port, IRQ_OFFSET+IRQ_SERIAL)               # 36    serial port interrupt
+       TRAPHANDLER_NOEC(trap_spurious, IRQ_OFFSET+IRQ_SPURIOUS)                # 39    spurious interrupt
+       TRAPHANDLER_NOEC(trap_ide, IRQ_OFFSET+IRQ_IDE)                                  # 46    ide interrupt
        TRAPHANDLER_NOEC(trap_system_call, T_SYSCALL)                                   # 48    system call
+       TRAPHANDLER_NOEC(trap_error, IRQ_OFFSET+IRQ_ERROR)                              # 51    error interrupt

 .data
 .globl idt_entries, idt_entries_end
@@ -92,8 +98,38 @@ idt_entries:
        .long trap_alignment_check                                      # 17    alignment check
        .long trap_machine_check                                        # 18    machine check
        .long trap_SIMD_float_point_error                       # 19    SIMD floating point error
-       .fill 28, 4, 0                                                          #               28 null entries
```

```
+       .long 0                                                                             # 20    null entry
+       .long 0                                                                             # 21    null entry
+       .long 0                                                                             # 22    null entry
+       .long 0                                                                             # 23    null entry
+       .long 0                                                                             # 24    null entry
+       .long 0                                                                             # 25    null entry
+       .long 0                                                                             # 26    null entry
+       .long 0                                                                             # 27    null entry
+       .long 0                                                                             # 28    null entry
+       .long 0                                                                             # 29    null entry
+       .long 0                                                                             # 30    null entry
+       .long 0                                                                             # 31    null entry
+       .long trap_timer                                                            # 32    timer interrupt
+       .long trap_keyboard                                                         # 33    keyboard interrupt
+       .long 0                                                                             # 34    null entry
+       .long 0                                                                             # 35    null entry
+       .long trap_serial_port                                              # 36    serial port interrupt
+       .long 0                                                                             # 37    null entry
+       .long 0                                                                             # 38    null entry
+       .long trap_spurious                                                         # 39    squrious interrupt
+       .long 0                                                                             # 40    null entry
+       .long 0                                                                             # 41    null entry
+       .long 0                                                                             # 42    null entry
+       .long 0                                                                             # 43    null entry
+       .long 0                                                                             # 44    null entry
+       .long 0                                                                             # 45    null entry
+       .long trap_ide                                                              # 46    ide interrupt
+       .long 0                                                                             # 47    null entry
        .long trap_system_call                                              # 48    system call
+       .long 0                                                                             # 49    null entry
+       .long 0                                                                             # 50
+       .long trap_error                                                            # 51    error interrupt
 idt_entries_end:

 /*
```

Inter-Process communication (IPC)

We will add a few more system calls for IPC. The “message” that user enviroments can send to each other using JOS’s IPC mechanism consists of two components: a single 32-bit value and optionally a single page mapping. Allowing environments to pass page mappings in messages provides an efficient way to transfer more data that won’t fit into a single 32-bit integer, and also allows environments to set up shared memory arrangements easily.

When sending or receiving a message, the process is de-scheduled until the message is transferred successfully. When an environment is waiting to receive a message, any other environment can send it a message - not just a particular environment, and not just environments that have a parent/child arrangement with the receiving environment.

When an environment calls `sys_ipc_recv` with a valid `dstva` parameter (below `UTOP`), the environment is stating that it is willing to receive a page mapping. If the sender sends a page, then that page should be mapped at `dstva` in the receiver’s address space. If the receiver already had a page mapped at `dstva` , then that previous page is unmapped. When an environment calls `sys_ipc_try_send` with a valid `srcva` (below `UTOP`), it means the sender wants to send the page currently mapped at `srcva` to the receiver, with permissions perm. After a successful IPC, the sender keeps its original mapping for the page at `srcva` in its address space, but the receiver also obtains a mapping for this same physical page at the dstva originally specified by the receiver, in the receiver’s address space. As a result this page becomes shared between the sender and receiver.

Exercise 15

> 15. Implement `sys_ipc_recv` and `sys_ipc_try_send` in `kern/syscall.c` . Read the comments on both before implementing them, since they have to work together. When you call `envid2env` in these routines, you should set the checkperm flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.
>
> Then implement the `ipc_recv` and `ipc_send` functions in `lib/ipc.c` .

Both sending and receiving library routines are synchronized, which means they don’t return until there is a result, either success or failure. Since kernel sets the process as `NOT_RUNNABLE` and won’t reset its state until there is a result, we can rely on kernel for synchronization. The sending system call will return immediately. So we need to loop in the library routine until there is a result. Furthermore, we give up CPU in each iteration to ease CPU.

In `ipc_send` we assign a value above `UTOP` to state that we don’t want to send a page.

The code for this exercise is shown below.

```
diff --git a/kern/syscall.c b/kern/syscall.c
index d9aaab1..eb59692 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -240,8 +240,8 @@ sys_page_map(envid_t srcenvid, void *srcva,
        struct PageInfo *pp;
        if ((uintptr_t)srcva >= UTOP || (uintptr_t)srcva % PGSIZE ||
                (uintptr_t)dstva >= UTOP || (uintptr_t)dstva % PGSIZE ||
-               (pp = page_lookup(srcenv->env_pgdir, srcva, &pte_ptr)) == NULL ||
                perm & ~PTE_SYSCALL || ~perm & (PTE_U | PTE_P) ||
+               (pp = page_lookup(srcenv->env_pgdir, srcva, &pte_ptr)) == NULL ||
                ((perm & PTE_W) && !((*pte_ptr) & PTE_W)))
                return -E_INVAL;

@@ -319,7 +319,46 @@ static int
 sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
 {
        // LAB 4: Your code here.
-       panic("sys_ipc_try_send not implemented");
+       // panic("sys_ipc_try_send not implemented");
+       struct Env *dst_env;
+       if (envid2env(envid, &dst_env, false) < 0)
+               return -E_BAD_ENV;
+
+       if (dst_env->env_ipc_recving == false)
+               return -E_IPC_NOT_RECV;
+
+       if ((uintptr_t)srcva < UTOP) {
+               // many requirements are asserted in sys_page_map.
+               // we can rely on sys_page_map to do it.
+
+               pte_t *pte_ptr;
+               struct PageInfo *pp;
+               // (uintptr_t)srcva < UTOP is asserted in outer if.
+               // (uintptr_t)dst_env->env_ipc_dstva % PGSIZE == 0 is asserted in sys_ipc_recv
+               // no need to check them
+               // (uintptr_t)dst_env->env_ipt_dstva < UTOP must be asserted to check whether
+               // the reciever really wants to recieve a page
+               if ((uintptr_t)dst_env->env_ipc_dstva >= UTOP || (uintptr_t)srcva % PGSIZE ||
+                       perm & ~PTE_SYSCALL || ~perm & (PTE_U | PTE_P) ||
+                       (pp = page_lookup(curenv->env_pgdir, srcva, &pte_ptr)) == NULL ||
+                       ((perm & PTE_W) && !((*pte_ptr) & PTE_W)))
+                       return -E_INVAL;
+
+               if (page_insert(dst_env->env_pgdir, pp, dst_env->env_ipc_dstva, perm) < 0)
+                       return -E_NO_MEM;
+
+               dst_env->env_ipc_perm = perm;
+       } else {
+               dst_env->env_ipc_perm = 0;
+       }
+
+       dst_env->env_ipc_recving = false;
+       dst_env->env_ipc_from = curenv->env_id;
+       dst_env->env_ipc_value = value;
+       dst_env->env_status = ENV_RUNNABLE;
+       dst_env->env_tf.tf_regs.reg_eax = 0;
+
+    return 0;
 }

 // Block until a value is ready.  Record that you want to receive
@@ -337,7 +376,17 @@ static int
 sys_ipc_recv(void *dstva)
 {
        // LAB 4: Your code here.
-       panic("sys_ipc_recv not implemented");
+       // panic("sys_ipc_recv not implemented");
+
+       if ((uintptr_t)dstva < UTOP && (uintptr_t)dstva % PGSIZE)
+               return -E_INVAL;
+
+       curenv->env_ipc_recving = true;
+       curenv->env_ipc_dstva = dstva;
+       curenv->env_status = ENV_NOT_RUNNABLE;
+
+       sched_yield();
+
        return 0;
 }

@@ -363,7 +412,8 @@ syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
                case SYS_env_set_status:        return sys_env_set_status((envid_t)a1, (int)a2);
                case SYS_env_set_pgfault_upcall: return sys_env_set_pgfault_upcall((envid_t)a1, (void *)a2);
                case SYS_yield: sys_yield(); return 0;
-
+               case SYS_ipc_try_send: return sys_ipc_try_send((envid_t)a1, (uint32_t)a2, (void *)a3, (unsigned int)a4);
+               case SYS_ipc_recv: return sys_ipc_recv((void *)a1);
                default:
                        return -E_INVAL;
        }
diff --git a/lib/ipc.c b/lib/ipc.c
index 2e222b9..827e68c 100644
--- a/lib/ipc.c
+++ b/lib/ipc.c
@@ -23,8 +23,22 @@ int32_t
 ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
 {
        // LAB 4: Your code here.
-       panic("ipc_recv not implemented");
-       return 0;
+       // panic("ipc_recv not implemented");
+       if (pg == NULL)
+               pg = (void *)(UTOP + PGSIZE);
+
+       int r = sys_ipc_recv(pg);
+
+       if (from_env_store)
+               *from_env_store = r < 0 ? 0 : thisenv->env_ipc_from;
+
+       if (perm_store)
+               *perm_store = r < 0 ? 0 : thisenv->env_ipc_perm;
+
+       if (r < 0)
+               return r;
+
+       return thisenv->env_ipc_value;
 }

 // Send 'val' (and 'pg' with 'perm', if 'pg' is nonnull) to 'toenv'.
@@ -39,7 +53,21 @@ void
 ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
 {
        // LAB 4: Your code here.
-       panic("ipc_send not implemented");
+       // panic("ipc_send not implemented");
+       if (pg == NULL)
+               pg = (void *)(UTOP + PGSIZE);
+
+       while (true) {
+               int r = sys_ipc_try_send(to_env, val, pg, perm);
+
+               if (r == 0)
+                       return;
+
+               if (r < 0 && r != -E_IPC_NOT_RECV)
+                       panic("Failed to try_send: %e\n", r);
+
+               sys_yield();
+       }
 }

 // Find the first environment of the given type.  We'll use this to
```

Challenge

> Implement a shared-memory `fork()` called `sfork()`. This version should have the parent and child share all their memory pages (so writes in one environment appear in the other) except for pages in the stack area, which should be treated in the usual copy-on-write manner.

To finish `sfork()` , we also need to implement a share-memory `duppage()` called `sduppage()` . `sduppage()` takes `envid` , `pn` and an extra arguments `cow_enabled` that tells `sduppage()` whether we want to do copy-on-write. The content in `sfork()` is almost the same as `fork()` , except that we invoke `sduppage()` instead of `duppage()` . To distinguish between stack and heap, we copy page table from `USTACKTOP` to `UTEXT` . Since user stack always grows from stack top continuously, all pages belongs to user stack until we meet the first page that's not mapped.

Apart from user stack, `thisenv` is another part of memory that should only belongs to one process. Because `thisenv` is declared as a global variable, which resides in heap, it will be considered a shared variable. One method to avoid this is to allocate a fixed page in every user space that will not be shared among process and store `thisenv` at this page. Because the address of this fixed page is determined when we design our OS, user process won't use it for other purposes. And the library routine is also able to treat this page the same as user stack. This need to re-design the memory layout so I didn't finish this.

To test my `sfork()` I wrote a test program called `sfork.c` , in which the parent process changes a global variable and the child process get the new value by accessing the same global variable.

```
#include <inc/lib.h>

int share = 1;

void umain(int argc, char **argv)
{
    int ch = sfork ();

    if (ch != 0) {
        cprintf ("I'm parent with share num = %d\n", share);
        share = 2;
    } else {
        sys_yield();
        cprintf ("I'm child with share num = %d\n", share);
    }
}
```

The child process gives up the CPU the first time it is scheduled so that we can guarantee the parent process changes the global variable before the child process access it.

The output of this test program is show below.

```
6828 decimal is 15254 octal!
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 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
I'm parent with share num = 1
[00001000] exiting gracefully
[00001000] free env 00001000
I'm child with share num = 2
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> QEMU: Terminated
```

The code for this challenge is show below.

```
diff --git a/kern/Makefrag b/kern/Makefrag
index 00c7efd..466faa3 100644
--- a/kern/Makefrag
+++ b/kern/Makefrag
@@ -75,7 +75,8 @@ KERN_BINFILES +=      user/idle \
                        user/fairness \
                        user/pingpong \
                        user/pingpongs \
-                       user/primes
+                       user/primes \
+                       user/sfork
 KERN_OBJFILES := $(patsubst %.c, $(OBJDIR)/%.o, $(KERN_SRCFILES))
 KERN_OBJFILES := $(patsubst %.S, $(OBJDIR)/%.o, $(KERN_OBJFILES))
 KERN_OBJFILES := $(patsubst $(OBJDIR)/lib/%, $(OBJDIR)/kern/%, $(KERN_OBJFILES))
diff --git a/lib/fork.c b/lib/fork.c
index 2e54f5f..bd50755 100644
--- a/lib/fork.c
+++ b/lib/fork.c
@@ -87,6 +87,38 @@ duppage(envid_t envid, unsigned pn)
        return 0;
 }
 
+//
+// duppage for share-fork
+//
+// Returns: 0 on success, < 0 on error.
+// It is also OK to panic on error.
+//
+static int
+sduppage(envid_t envid, unsigned pn, int cow_enabled)
+{
+
+       int r;
+
+       void *addr = (void *)(pn * PGSIZE);
+       int perm = PGOFF(uvpt[pn]) & PTE_SYSCALL;
+
+       if (cow_enabled && (perm & PTE_W)) {
+               perm |= PTE_COW;
+               perm &= ~PTE_W;
+
+               if ((r = sys_page_map(0, addr, envid, addr, perm)) < 0)
+                       panic("sduppage: Failed to duppage on child enviroment: %e\n", r);
+
+               if ((r = sys_page_map(0, addr, 0, addr, perm)) < 0)
+                       panic("sduppage: Failed to duppage on self enviroment: %e\n", r);
+       } else {
+               if ((r = sys_page_map(0, addr, envid, addr, perm)) < 0)
+                       panic("sduppage: Failed to duppage on child enviroment: %e\n", r);
+       }
+
+       return 0;
+}
+
 //
 // User-level fork with copy-on-write.
 // Set up our page fault handler appropriately.
@@ -151,6 +183,39 @@ COPY_END:
 int
 sfork(void)
 {
-       panic("sfork not implemented");
-       return -E_INVAL;
-}
+       //panic("sfork not implemented");
+       int r;
+
+       //LAB 4: Your code here.
+       set_pgfault_handler(pgfault);
+
+       envid_t envid = sys_exofork();
+       if (envid < 0)
+               panic("sys_exofork: %e", envid);
+
+       if (envid == 0) {
+               thisenv = &envs[ENVX(sys_getenvid())];
+               return 0;
+       }
+
+       bool stackarea = true;
+       for (uint32_t addr = USTACKTOP - PGSIZE; addr >= UTEXT; addr -= PGSIZE) {
+               if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P))
+                       sduppage(envid, PGNUM(addr), stackarea);
+               else
+                       stackarea = false;
+       }
+
+       if ((r = sys_page_alloc(envid, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P)) < 0)
+               panic("sfork: Failed to allocate page for child's exception stack: %e", r);
+
+       extern void _pgfault_upcall(void);
+       if ((r = sys_env_set_pgfault_upcall(envid, _pgfault_upcall)) < 0)
+               panic("sfork: Failed to set page fault hanlder for child: %e\n", r);
+
+       if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
+               panic("sfork: Failed to set child's status as ENV_RUNNABLE", r);
+
+       return envid;
+       //return -E_INVAL;
+}
\ No newline at end of file
diff --git a/user/sfork.c b/user/sfork.c
new file mode 100644
index 0000000..eaa4242
--- /dev/null
+++ b/user/sfork.c
@@ -0,0 +1,16 @@
+#include <inc/lib.h>
+
+int share = 1;
+
+void umain(int argc, char **argv)
+{
+    int ch = sfork ();
+
+    if (ch != 0) {
+        cprintf ("I'm parent with share num = %d\n", share);
+        share = 2;
+    } else {
+        sys_yield();
+        cprintf ("I'm child with share num = %d\n", share);
+    }
+}
\ No newline at end of file
```

Grade

Finally, we got our grade.

```
dumbfork: OK (4.4s)
Part A score: 5/5

faultread: OK (0.8s)
faultwrite: OK (1.3s)
faultdie: OK (1.9s)
faultregs: OK (2.0s)
faultalloc: OK (1.0s)
faultallocbad: OK (2.0s)
faultnostack: OK (1.9s)
faultbadhandler: OK (2.0s)
faultevilhandler: OK (1.8s)
forktree: OK (2.2s)
Part B score: 50/50

spin: OK (2.2s)
stresssched: OK (1.8s)
sendpage: OK (2.0s)
pingpong: OK (1.7s)
primes: OK (2.7s)
Part C score: 25/25

Score: 80/80
```

End of Lab 4 Report

Email: caoshuyang@pku.edu.cn GitHub: JOS-Lab