

操作系统 第四次实验报告

周子锐 2100011032

2023 年 11 月 25 日

完成的 challenge 为:

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. Modify user/forktree.c to use sfork() instead of regular fork(). Also, once you have finished implementing IPC in part C, use your sfork() to run user/pingpongs. You will have to find a new way to provide the functionality of the global thisenv pointer.

其相关的信息在文章最后部分.

Exercise 1

使用 boot_map_region() 函数辅助实现, 注意地址不能超过 MMIOLIM.

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    static uintptr_t base = MMIIOBASE;
    size = ROUNDUP(size, PGSIZE);
    if (base + size > MMIOLIM) {
        panic("mmio_map_region: overflow MMIOLIM");
    }
    boot_map_region(kern_pgdir, base, size, pa, PTE_PCD | PTE_PWT | PTE_W);
    void *ret = (void *)base;
    base += size;
    return ret;
}
```

Exercise 2

单独将 MPENTRY_PADDR 对应的页面进行处理即可

```
void
page_init(void)
{
    ...
    for (i = 0; i < npages; i++) {
        if(i == MPENTRY_PADDR/PGSIZE){
            pages[i].pp_ref = 1;
            continue;
        }
        pages[i].pp_ref = 0;
        if ((i == 0) || ((i >= IOPHYSMEM / PGSIZE) && i < kernel_end)) {
            pages[i].pp_link = NULL;
        } else {
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        }
    }
}
```

Question 1

MPBOOTPHYS 宏的作用是计算出对应的物理地址。这个宏是必须的是因为这段代码的加载地址和链接地址并不相同，因此对于那些需要使用绝对地址进行寻址的地方需要使用这个宏进行转换，否则就会因为在 real mode 下访问高地址而出现问题。

Exercise 3

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

```
}  
  
}
```

Exercise 4

修改所有与 CPU 相关的信息.

```
void  
trap_init_percpu(void)  
{  
    uint32_t id = thiscpu->cpu_id;  
  
    // Setup a TSS so that we get the right stack  
    // when we trap to the kernel.  
    thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - id * (KSTKSIZE + KSTKGAP);  
    thiscpu->cpu_ts.ts_ss0 = GD_KD;  
    thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);  
  
    // Initialize the TSS slot of the gdt.  
    gdt[(GD_TSS0 >> 3) + id] = SEG16(STS_T32A, (uint32_t) (&thiscpu->cpu_ts),  
                                     sizeof(struct Taskstate) - 1, 0);  
    gdt[(GD_TSS0 >> 3) + id].sd_s = 0;  
  
    // Load the TSS selector (like other segment selectors, the  
    // bottom three bits are special; we leave them 0)  
    ltr(GD_TSS0 + (id << 3));  
  
    // Load the IDT  
    lidt(&idt_pd);  
}
```

Exercise 5

根据要求在四个地方添加操作即可, 代码略.

Question 2

注意到锁的获取发生在陷入内核之后, 而将上下文信息压栈发生在此之前. 故而如果使用公共的内核栈的话, 会由于压栈指令执行顺序的不确定性导致上下文无法被正确压入.

Exercise 6

根据要求循环便利所有的 `env` 即可, 如果找了一圈都没有找到, 那么就考虑再次运行可能存在的当前的 `env`

```
void sched_yield(void)
{
    struct Env *idle;
    uint32_t st = curenv ? ENVX(curenv->env_id) : 0;
    for (uint32_t i = 1; i <= NENV; i++) {
        uint32_t j = (st + i) % NENV;
        if (envs[j].env_status == ENV_RUNNABLE) {
            env_run(&envs[j]);
        }
    }
    if ((curenv) && (curenv->env_status == ENV_RUNNING)) {
        env_run(curenv);
    }
    sched_halt();
}
```

之后在 `kern/syscall.c/syscall()` 中添加 `SYS_YIELD`.

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

再在 `kern/init.c/i386_init()` 中创建若干运行指定程序的 `env`.

```
...
#if defined(TEST)
    // Don't touch -- used by grading script!
    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);  
#endif // TEST*  
...
```

最后在 `kern/init.c/mp_main()` 中加上 `sched_yield()` 函数的调用. 运行命令 `make qemu CPUS=4` 即可得到:

```
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  
[00000000] new env 00001000  
[00000000] new env 00001001  
[00000000] new env 00001002  
Hello, I am environment 00001001.  
Hello, I am environment 00001002.  
Hello, I am environment 00001000.  
Back in environment 00001001, iteration 0.  
Back in environment 00001002, iteration 0.  
Back in environment 00001000, iteration 0.  
Back in environment 00001001, iteration 1.  
Back in environment 00001002, iteration 1.  
Back in environment 00001000, iteration 1.  
Back in environment 00001001, iteration 2.  
Back in environment 00001002, iteration 2.  
Back in environment 00001000, iteration 2.  
Back in environment 00001001, iteration 3.  
Back in environment 00001002, iteration 3.  
Back in environment 00001000, iteration 3.
```

图 1: Exercise 6 运行截图

Question 3

因为在 KERNBASE 以上的部分在所有的进程中的映射都是一样的.

Question 4

因为进程没有自己单独的内核栈, 所以它的上下文信息必须被保存在 PCB 之类的结构中来方便恢复运行状态. 在 `kern/trapentry.S` 中将 `trapframe` 的信息压入栈中, 然后调用 `kern/trap.c/trap()` 函数, 通过如下代码储存:

```
...  
// Copy trap frame (which is currently on the stack)  
// into 'curenv->env_tf', so that running the environment  
// will restart at the trap point.
```

```
curenv->env_tf = *tf;

// The trapframe on the stack should be ignored from here on.
tf = &curenv->env_tf;

...
```

Exercise 7

首先根据注释补充完整 kern/syscall.c 中的五个函数.

```
static env_id_t sys_exofork(void)
{
    struct Env *parent, *child;
    env_id_t ret;

    parent = curenv;
    ret = env_alloc(&child, parent->env_id);
    if (ret < 0) {
        return ret;
    }
    child->env_status = ENV_NOT_RUNNABLE;
    child->env_tf = parent->env_tf;
    child->env_tf.tf_regs.reg_eax = 0;
    return child->env_id;
}

static int sys_env_set_status(env_id_t env_id, int status)
{
    struct Env *env;
    int ret = env_id2env(env_id, &env, 1);
    if (ret < 0) {
        return ret;
    }
    if ((status != ENV_RUNNABLE) && (status != ENV_NOT_RUNNABLE)) {
        return -E_INVALID;
    }
    env->env_status = status;
    return 0;
}
```

```
static int sys_page_alloc(envid_t envid, void *va, int perm)
{
    struct Env *env;
    int ret = envid2env(envid, &env, 1);
    if (ret < 0) {
        return ret;
    }
    if (((uint32_t)va >= UTOP) || (va != ROUNDUP(va, PGSIZE))) {
        return -E_INVAL;
    }
    if ((perm & PTE_SYSCALL) != perm) {
        return -E_INVAL;
    }
    if ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) {
        return -E_INVAL;
    }
    struct PageInfo *page = page_alloc(ALLOC_ZERO);
    if (page == NULL) {
        return -E_NO_MEM;
    }
    ret = page_insert(env->env_pgdir, page, va, perm);
    if (ret < 0) {
        page_free(page);
        return ret;
    }
    return 0;
}

static int sys_page_map(envid_t srcenvid, void *srcva, envid_t dstenvid, void *dstva, int perm)
{
    struct Env *srcenv, *dstenv;
    int ret = envid2env(srcenvid, &srcenv, 1);
    if (ret < 0) {
        return ret;
    }
    ret = envid2env(dstenvid, &dstenv, 1);
    if (ret < 0) {
        return ret;
    }
}
```

```
if (((uint32_t)srcva >= UTOP) || (srcva != ROUNDUP(srcva, PGSIZE))) {
    return -E_INVALID;
}

if (((uint32_t)dstva >= UTOP) || (dstva != ROUNDUP(dstva, PGSIZE))) {
    return -E_INVALID;
}

pte_t *pte;
struct PageInfo *page = page_lookup(srcenv->env_pgdir, srcva, &pte);
if (page == NULL) {
    return -E_INVALID;
}

if ((perm & PTE_SYSCALL) != perm) {
    return -E_INVALID;
}

if ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) {
    return -E_INVALID;
}

if ((perm & PTE_W) && !(*pte & PTE_W)) {
    return -E_INVALID;
}

ret = page_insert(dstenv->env_pgdir, page, dstva, perm);
return ret;
}

static int
sys_page_unmap(envid_t envid, void *va)
{
    // Hint: This function is a wrapper around page_remove().

    // LAB 4: Your code here.

    struct Env *env;
    int ret = envid2env(envid, &env, 1);
    if (ret < 0) {
        return ret;
    }

    if (((uint32_t)va >= UTOP) || (va != ROUNDUP(va, PGSIZE))) {
        return -E_INVALID;
    }

    page_remove(env->env_pgdir, va);
}
```



```

return 0;
}

```

再如下补充 kern/syscall.c/syscall() 函数.

```

...
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);
...

```

最后修改 kern/init.c/i386_init() 函数, 使其能够运行 dumbfork 程序, 运行截图如下:

```

check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
0: I am the parent!
0: I am the child!
1: I am the parent!
1: I am the child!
2: I am the parent!
2: I am the child!
3: I am the parent!
3: I am the child!
4: I am the parent!
4: I am the child!
5: I am the parent!
5: I am the child!
6: I am the parent!
6: I am the child!
7: I am the parent!
7: I am the child!
8: I am the parent!
8: I am the child!
9: I am the parent!
9: I am the child!
[00001000] exiting gracefully
[00001000] free env 00001000
6: I am the child!
7: I am the parent!
7: I am the child!
8: I am the parent!
8: I am the child!
9: I am the parent!
9: I am the child!
[00001000] exiting gracefully
[00001000] free env 00001000
10: I am the child!
11: I am the child!
12: I am the child!
13: I am the child!
14: I am the child!
15: I am the child!
16: I am the child!
17: I am the child!
18: I am the child!
19: I am the child!
[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.

```

图 2: dumbfork 程序运行截图

This completes Part A.

Exercise 8

按照注释完成即可

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

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

Exercise 9

需要指出的是, 如果是正常执行时出现的 page fault, 那么内核会在一个新的名为'exception stack' 的栈上重启该用户进程. 然后会在这个栈上压一个 UTrapframe. 如果在 page fault handler 中又出现了 page fault, 则会嵌套处理, 不过此时压栈的位置不是 UXSTACKTOP 而为 tf->tf_esp, 同时这个压栈的操作会在栈的顶部预先空出 4B 大小的位置, 方便指出返回地址.

```
void
page_fault_handler(struct Trapframe *tf)
{
    ...
    if (curenv->env_pgfault_upcall) {
        struct UTrapframe *utf;
        if (ROUNDDOWN(tf->tf_esp, PGSIZE) == UXSTACKTOP - PGSIZE) {
            utf = (struct UTrapframe *) (tf->tf_esp - sizeof(struct UTrapframe) - 4);
        } else {
            utf = (struct UTrapframe *) (UXSTACKTOP - sizeof(struct UTrapframe));
        }
        user_mem_assert(curenv, (void *) utf, 1, PTE_W);
        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;
    }
}
```

```
    curenv->env_tf.tf_eip = (uintptr_t) curenv->env_pgfault_upcall;
    curenv->env_tf.tf_esp = (uintptr_t) utf;
    env_run(curenv);
}

// Destroy the environment that caused the fault.
cprintf("[%08x] user fault va %08x ip %08x\n",
        curenv->env_id, fault_va, tf->tf_eip);
print_trapframe(tf);
env_destroy(curenv);
}
```

Exercise 10

首先需要修正储存的 esp 寄存器 (因为有 4B 的预留空间), 之后用到的 trick 是: 利用之前压栈预留的 4B 空间计算并储存返回地址, 之后正常进行弹栈和恢复操作, 最后通过 `ret` 指令跳转到压入的 `tf` 所指出的返回地址即可.

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

addl $0x8, %esp
popal

addl $0x4, %esp
popfl

popl %esp

ret
```

Exercise 11

第一次设置 page handler 时首先为 exception stack 分配内存, 再通过系统调用设置 handler.

```
void
```

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

    if (_pgfault_handler == 0) {
        int ret = sys_page_alloc(0, (void *) (UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P);
        if (ret < 0) {
            panic("set_pgfault_handler: sys_page_alloc failed: %e", ret);
        }
        ret = sys_env_set_pgfault_upcall(0, _pgfault_upcall);
        if (ret < 0) {
            panic("set_pgfault_handler: sys_env_set_pgfault_upcall failed: %e", ret);
        }
    }

    _pgfault_handler = handler;
}
```

Exercise 12

补全 lib/fork.c 中的三个函数.

static void pgfault(struct UTrapframe *utf)

首先检查这个 fault 是否是由于写一个 COW 页面. 之后请求一个新的页面, 复制原内面内容并修改映射关系.

```
static void
pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    int r;

    if (!((err & FEC_WR) && (uvpt[PGNUM(addr)] & PTE_COW))) {
        panic("pgfault: faulting access was not a write or to a copy-on-write page");
    }

    addr = ROUNDDOWN(addr, PGSIZE);
```

```

int ret = sys_page_alloc(0, PFTEMP, PTE_U | PTE_W | PTE_P);
if (ret < 0) {
    panic("pgfault: sys_page_alloc failed: %e", ret);
}
memmove(PFTEMP, addr, PGSIZE);
ret = sys_page_map(0, PFTEMP, 0, addr, PTE_U | PTE_W | PTE_P);
if (ret < 0) {
    panic("pgfault: sys_page_map failed: %e", ret);
}
ret = sys_page_unmap(0, PFTEMP);
if (ret < 0) {
    panic("pgfault: sys_page_unmap failed: %e", ret);
}
}

```

static int duppage(envid_t environ, unsigned pn)

根据页面 pn 是否可写分类讨论. 如果可写则设置 COW 位后先映射到子进程再映射到父进程, 否则直接进行映射,

```

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

    // LAB 4: Your code here.
    void *addr = (void *) (pn * PGSIZE);
    int perm = uvpt[pn] & PTE_SYSCALL;
    if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW)) {
        perm &= ~PTE_W;
        perm |= PTE_COW;
        int ret = sys_page_map(0, addr, environ, addr, perm);
        if (ret < 0) {
            panic("duppage: sys_page_map failed: %e", ret);
        }
        ret = sys_page_map(0, addr, 0, addr, perm);
        if (ret < 0) {
            panic("duppage: sys_page_map failed: %e", ret);
        }
    } else {

```

```
int ret = sys_page_map(0, addr, envid, addr, perm);
if (ret < 0) {
    panic("duppage: sys_page_map failed: %e", ret);
}
}
return 0;
}
```

envid_t fork(void)

大致做了这几件事

- 父进程设置 page handler.
- 父进程调用 `sys_exofork()` 创建子进程.
- 子进程设置 env.
- 父进程将 UTOP 以下的页面通过 `duppage()` 函数复制到子进程.
- 父进程为子进程分配 exception stack.
- 父进程设置子进程的 page handler.
- 父进程设置子进程的状态为 `ENV_RUNNABLE`.

```
envid_t
fork(void)
{
    // LAB 4: Your code here.
    extern void _pgfault_upcall(void);

    set_pgfault_handler(pgfault);
    envid_t envid = sys_exofork();
    if (envid < 0) {
        panic("fork: sys_exofork failed: %e", envid);
    }
    if (envid == 0) {
        // child process
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }
}
```

```
}  
// parent process  
  
for (uintptr_t addr = 0; addr < USTACKTOP; addr += PGSIZE) {  
    if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P)) {  
        duppage(envid, PGNUM(addr));  
    }  
}  
  
int ret = sys_page_alloc(envid, (void *) (USTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P);  
if (ret < 0) {  
    panic("fork: sys_page_alloc failed: %e", ret);  
}  
  
ret = sys_env_set_pgfault_upcall(envid, _pgfault_upcall);  
if (ret < 0) {  
    panic("fork: sys_env_set_pgfault_upcall failed: %e", ret);  
}  
  
ret = sys_env_set_status(envid, ENV_RUNNABLE);  
if (ret < 0) {  
    panic("fork: sys_env_set_status failed: %e", ret);  
}  
  
return env;  
}
```

This ends part B.

Exercise 13

在 kern/trapentry.S 中添加硬件中断相关信息:

```
...  
TRAPHANDLER_NOEC(handler_timer, IRQ_OFFSET + IRQ_TIMER)  
TRAPHANDLER_NOEC(handler_kbd, IRQ_OFFSET + IRQ_KBD)  
TRAPHANDLER_NOEC(handler_serial, IRQ_OFFSET + IRQ_SERIAL)  
TRAPHANDLER_NOEC(handler_spurious, IRQ_OFFSET + IRQ_SPURIOUS)  
TRAPHANDLER_NOEC(handler_ide, IRQ_OFFSET + IRQ_IDE)  
TRAPHANDLER_NOEC(handler_error, IRQ_OFFSET + IRQ_ERROR)  
...
```

在 kern/trap.c/trap_init() 中添加 descriptor:

```
void handler_timer();
void handler_kbd();
void handler_serial();
void handler_spurious();
void handler_ide();
void handler_error();

void
trap_init(void)
{
    ...
    SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, handler_timer, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_KBD], 0, GD_KT, handler_kbd, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_SERIAL], 0, GD_KT, handler_serial, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_SPURIOUS], 0, GD_KT, handler_spurious, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_IDE], 0, GD_KT, handler_ide, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_ERROR], 0, GD_KT, handler_error, 0);

    // Per-CPU setup
    trap_init_percpu();
}
```

最后修改 kern/env.c/env_alloc() 函数, 使得新创建的进程在用户态中能够接收中断.

```
int
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;

    ...
}
```


这之后修改 `kern/sched.c/sched_halt()` 函数使得其可以执行 `sti` 指令, 运行 `usr/spin.c` 得到如下的结果.

```
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[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...
TRAP frame at 0xf029b07c from CPU 0
  edi 0x00000000
  esi 0x00000000
  ebp 0xeebdfd0
  oesp 0xeffffdc
  ebx 0x00000000
  edx 0xeebfde88
  ecx 0x0000001d
  eax 0x0000001d
  es 0x----0023
  ds 0x----0023
  trap 0x00000020 Hardware Interrupt
  err 0x00000000
  eip 0x00000060
  cs 0x----001b
  flag 0x00000282
  esp 0xeebdfc8
  ss 0x----0023
[00001001] free env 00001001
I am the parent. Killing the child...
[00001000] exiting gracefully
[00001000] free env 00001000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
```

图 3: 运行 `spin.c` 的结果

Exercise 14

在 `kern/trap.c/trap_dispatch()` 中添加如下代码, 其中函数 `lapic_eoi()` 用来接受中断.

```
static void
trap_dispatch(struct Trapframe *tf)
{
    ...

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

Exercise 15

首先是 kern/syscall.c 中的两个函数.

`static int sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)`

仿照 `sys_page_alloc()` 进行权限判断. 如果存在页面映射的需求则将对页面利用 `page_insert()` 函数插入到目标进程中.

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    struct Env *dstenv;
    bool page_transfer = false;
    int ret = envid2env(envid, &dstenv, 0);
    if (ret < 0) {
        return ret;
    }
    if (!dstenv->env_ipc_recving) {
        return -E_IPC_NOT_RECV;
    }
    if ((uint32_t)srcva < UTOP) {
        if ((uint32_t)srcva != ROUNDUP((uint32_t)srcva, PGSIZE)) {
            return -E_INVALID;
        }
        if ((perm & PTE_SYSCALL) != perm) {
            return -E_INVALID;
        }
        if ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) {
            return -E_INVALID;
        }
        pte_t *pte;
        struct PageInfo *page = page_lookup(curenv->env_pgdir, srcva, &pte);
        if (page == NULL) {
            return -E_INVALID;
        }
        if ((perm & PTE_W) && !(*pte & PTE_W)) {
            return -E_INVALID;
        }
    }
}
```

```

    if (((*pte) & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) {
        return -E_INVALID;
    }
    if ((uint32_t)dstenv->env_ipc_dstva < UTOP) {
        ret = page_insert(dstenv->env_pgdir, page, dstenv->env_ipc_dstva, perm);
        if (ret < 0) {
            return ret;
        }
        page_transfer = true;
    }
}
dstenv->env_ipc_recving = false;
dstenv->env_ipc_from = curenv->env_id;
dstenv->env_ipc_value = value;
if (page_transfer) {
    dstenv->env_ipc_perm = perm;
} else {
    dstenv->env_ipc_perm = 0;
}
dstenv->env_status = ENV_RUNNABLE;
dstenv->env_tf.tf_regs.reg_eax = 0;
return 0;
}

```

static int sys__ipc__recv(void *dstva)

按照要求设置状态即可.

```

static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if (((uint32_t)dstva < UTOP) && ((uint32_t)dstva != ROUNDUP((uint32_t)dstva, PGSIZE))) {
        return -E_INVALID;
    }
    curenv->env_ipc_recving = true;
    curenv->env_ipc_dstva = dstva;
    curenv->env_status = ENV_NOT_RUNNABLE;
    sched_yield();
    return 0;
}

```

```
}
```

接下来是 lib/ipc.c 中的两个函数.

int32_t ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)

按照注释实现即可, 注意当系统调用失败时需要将两个内存地址的值设为 0(如果做得到的话).

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    if (pg == NULL) {
        pg = (void *) UTOP;
    } else {
        pg = ROUNDDOWN(pg, PGSIZE);
    }
    int ret = sys_ipc_recv(pg);
    if (ret < 0) {
        if (from_env_store != NULL) {
            *from_env_store = 0;
        }
        if (perm_store != NULL) {
            *perm_store = 0;
        }
        return ret;
    }
    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;
}
```

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

循环发送直到成功为止.

```
void
```

```
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    if (pg == NULL) {
        pg = (void *) UTOP;
    } else {
        pg = ROUNDDOWN(pg, PGSIZE);
    }
    int ret;
    while ((ret = sys_ipc_try_send(to_env, val, pg, perm)) < 0) {
        if (ret != -E_IPC_NOT_RECV) {
            panic("ipc_send: %e", ret);
        }
        sys_yield();
    }
}
```

最终测试结果如下:



```
dumbfork: OK (1.1s)
Part A score: 5/5
) faultread: OK (1.0s)
  faultwrite: OK (1.0s)
  faultdie: OK (1.0s)
  faultregs: OK (1.0s)
  faultalloc: OK (1.0s)
  faultallocbad: OK (1.0s)
  faultnostack: OK (1.0s)
  faultbadhandler: OK (1.0s)
  faultevilhandler: OK (1.0s)
  forktree: OK (1.0s)
) Part B score: 50/50

spin: OK (1.1s)
stresssched: OK (1.1s)
sendpage: OK (0.7s)
  (Old jos.out.sendpage failure log removed)
pingpong: OK (1.0s)
  (Old jos.out.pingpong failure log removed)
primes: OK (1.9s)
  (Old jos.out.primes failure log removed)
Part C score: 25/25
Score: 80/80
```

图 4: 最终测试结果

This ends part C.

Challenge

该 challenge 处于 Part B 的末尾, 需要实现 `sfork` 函数, 即共享除了栈空间以外内存的 `fork` 函数. 为此需要实现一个新的 `duppage` 方法, 这里记作 `sduppage()`. 页面的共享则通过设置不同的标志位实现.

```
static int
sduppage(envid_t envid, unsigned pn)
{
    void *addr = (void *)(pn * PGSIZE);
    int perm = uvpt[pn] & PTE_SYSCALL;
    int ret = sys_page_map(0, addr, envid, addr, perm);
    if (ret < 0) {
        panic("sduppage: sys_page_map failed: %e", ret);
    }
    return 0;
}

// Challenge!
int
sfork(void)
{
    extern void _pgfault_upcall(void);

    set_pgfault_handler(pgfault);
    envid_t envid = sys_exofork();
    if (envid < 0) {
        panic("fork: sys_exofork failed: %e", envid);
    }
    if (envid == 0) {
        // child process
        return 0;
    }
    // parent process

    for (uintptr_t addr = 0; addr < (USTACKTOP - PGSIZE); addr += PGSIZE) {
        if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_U))
        {
            sduppage(envid, PGNUM(addr));
        }
    }
}
```

```

}

duppage(envid, PGNUM(USTACKTOP - PGSIZE));

int ret = sys_page_alloc(envid, (void *) (UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P);
if (ret < 0) {
    panic("fork: sys_page_alloc failed: %e", ret);
}

ret = sys_env_set_pgfault_upcall(envid, _pgfault_upcall);
if (ret < 0) {
    panic("fork: sys_env_set_pgfault_upcall failed: %e", ret);
}

ret = sys_env_set_status(envid, ENV_RUNNABLE);
if (ret < 0) {
    panic("fork: sys_env_set_status failed: %e", ret);
}

return env;
}

```

这里的问题是无法再使用 `thisenv` 因为它也被两个进程共享了, 解决方法是在使用它的时候重新通过进程调用获取正确的值. 修改使用了这个变量的 `lib/ipc.c/ipc_recv()` 函数如下

```

int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    if (pg == NULL) {
        pg = (void *) UTOP;
    } else {
        pg = ROUNDDOWN(pg, PGSIZE);
    }

    int ret = sys_ipc_recv(pg);
    if (ret < 0) {
        if (from_env_store != NULL) {
            *from_env_store = 0;
        }
        if (perm_store != NULL) {
            *perm_store = 0;
        }
    }
}

```

```
    return ret;
}

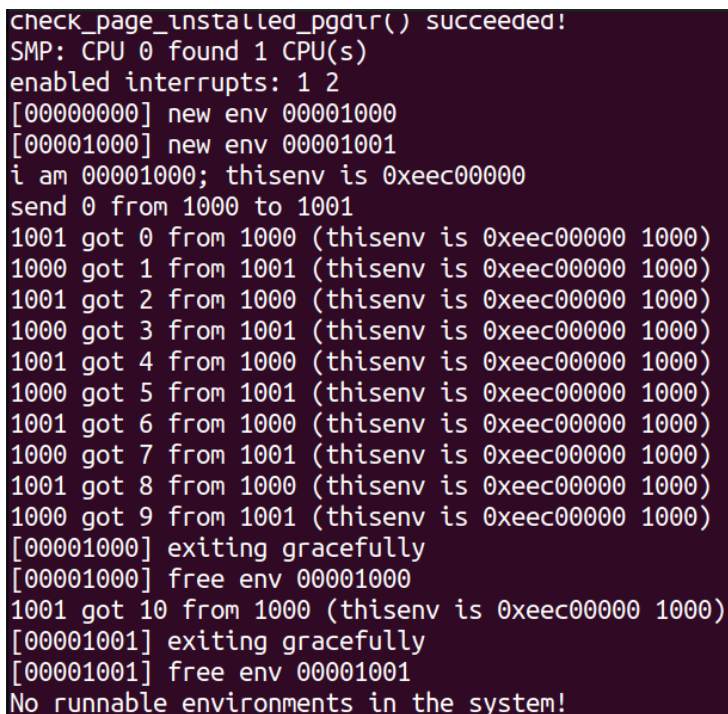
// uncomment this line if you want to do the `sfork` challenge
const volatile struct Env *thisenv = envs + ENVX(sys_getenvid());

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;
}
```

修改后可以成功运行 `usr/pingpongs.c`, 输出结果如下:



```
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 am 00001000; thisenv is 0xeec00000
send 0 from 1000 to 1001
1001 got 0 from 1000 (thisenv is 0xeec00000 1000)
1000 got 1 from 1001 (thisenv is 0xeec00000 1000)
1001 got 2 from 1000 (thisenv is 0xeec00000 1000)
1000 got 3 from 1001 (thisenv is 0xeec00000 1000)
1001 got 4 from 1000 (thisenv is 0xeec00000 1000)
1000 got 5 from 1001 (thisenv is 0xeec00000 1000)
1001 got 6 from 1000 (thisenv is 0xeec00000 1000)
1000 got 7 from 1001 (thisenv is 0xeec00000 1000)
1001 got 8 from 1000 (thisenv is 0xeec00000 1000)
1000 got 9 from 1001 (thisenv is 0xeec00000 1000)
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000 (thisenv is 0xeec00000 1000)
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
```

图 5: 运行 `usr/pingpongs.c` 的结果

This completes the lab.