写在之前：

我把 DEBUG_LOCK 注释了 ,这样 primes 测试在 30s 的 timeout 情况下一定可以通过

不注释 DEBUG_LOCK 不会有问题，这一点已经证明过了，所有的测试不存在多放锁的问题，多放锁会引起 kernel page fault（DEBUG 了很久发现很多的 kernel page fault 是由于多放锁里的检测中的 printf 引起的）

其他方法就是修改 timeout，但是这样修改测试代码不太好

如果不注释 DEBUG_LOCK，我的代码对于 primes 测试 30s 的 timeout 是概率通过的，这点非常蛋疼（虽然比大多数人 30s 的 timeout 一定不过，需要改成 50s 要好很多）

PART A Mutiprocessor Support and Cooperate Mutitasking

这个Part主要是为了多CPU多进程做一些准备工作 ,把一个CPU一个 env 扩展到多个 CPU 多个 env

包括 kernel stack , ts (不同进程的 esp 和 ss) , lock , sched (进程转换) 和基本的 exofrok

Exercise 1. Read boot_aps() and mp_main() in kern/init.c, and the assembly code inkern/mpentry.S. 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.

```
void page_init(void) {
    ......
     if(i == MPENTRY_PADDR/PGSIZE) continue;
    ......
}
```

代码实现：

由于 MPENTRY_PADDR 处要加载 AP bootstrap code，所以这个位置所代表的 page 不能被标记为 free 的，修改代码当遇到该 page 时跳过，不放入 page_free_list

Exercise 2. Modify mem_init_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in our revised inc/memlayout.h. The size of each stack is KSTKSIZE bytes plus KSTKGAP bytes of unmapped guard pages.

```
// Modify mappings in kern_pgdir to support SMP
//     - Remap [IOMEMBASE, 2^32) to physical address [IOMEM_PADDR, 2^32)
//     - Map the per-CPU stacks in the region [KSTACKTOP-PTSIZE, KSTACKTOP)
// See the revised inc/memlayout.h
static void mem_init_mp(void)
{
    boot_map_region(kern_pgdir, IOMEMBASE, -IOMEMBASE, IOMEM_PADDR, PTE_W);

    // 由于 KSTKGAP 没有设置权限，所以是 0,只要访问到了，就是 PGFALT
    int i;
    for (i = 0; i < NCPU; i++)
        boot_map_region(kern_pgdir, KSTACKTOP - i * (KSTKSIZE + KSTKGAP) - KSTKSIZE, KSTKSIZE, PADDR(percpu_kstacks[i]), PTE_W);
}
```

代码实现：

把 cpu 的 stack 映射到 KSTACKTOP 处。

对于每个 cpu，需要把 PADDR(percpu_kstacks[i])位置大小为 KTSIZE 的内存映射到 KSTACKTOP - i * (KSTKSIZE + KSTKGAP) – KSTKSIZE 处

每个 STACK 需要留下 KSTKGAP 的空白以便于检查 overflow

由于是 kernel stack ,这一块对 user 是不开放的 ,而对 kerenl 是可写的（PTE_W）

Exercise 3. 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.)

```
// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    //  设置当前 cpu 的 taskstate 的 esp 指针的初始位置为其相应的在 pmap.c 中分配的位置
    int i = thiscpu->cpu_id;
    thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    thiscpu->cpu_ts.ts_ss0 = GD_KD;

    //wrmsr for system call
    extern void sysenter_handler();
    wrmsr(0x174, GD_KT, 0);                    /* SYSENTER_CS_MSR */
    wrmsr(0x175, thiscpu->cpu_ts.ts_esp0, 0);/* SYSENTER_ESP_MSR */
     wrmsr(0x176, sysenter_handler, 0);        /* SYSENTER_EIP_MSR */
```

```
    // TSS: 任务状态段(Task State Segment)
    // Initialize the TSS slot of the gdt.
    // Use gdt[(GD_TSS0 >> 3) + i] for CPU i's TSS descriptor;
    gdt[(GD_TSS0 >> 3) + i] = SEG16(STS_T32A, (uint32_t)
(&thiscpu->cpu_ts), sizeof(struct Taskstate), 0);
    gdt[(GD_TSS0 >> 3) + i].sd_s = 0;

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

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

代码实现：

先设置每个 cpu 的 taskstate 的 esp(普通寄存器)指针和 ss(段寄存器) 指针，普通寄存器指向之前 exercise2 分配的 KSTACKTOP - i * (KSTKSIZE + KSTKGAP)处，段寄存器默认为 GD_KT 处

设置 msr 寄存器 ,存储 CPU 信息。0x174 是 cs ,0x175 是 esp，0x176 是 sysenter_eip，0x175 处需要在 lab3 基础上修改 (因为每个 cpu 的 esp 位置不同)，为之前设置的 esp

之后设置 gdt（全局描述表）的 tss，再读取 tss

最后读取 idt

Exercise 4. Apply the big kernel lock as described above, by calling lock_kernel() and unlock_kernel() at the proper locations.

```
void env_run(struct Env *e)
{
    if(curenv != NULL)
        if(curenv->env_status == ENV_RUNNING)
            curenv->env_status = ENV_RUNNABLE;
    // if curenv == e, we still make curenv->status = ENV_RUNNING !!!!!
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs++;
    lcr3(PADDR(e->env_pgdir));
    unlock_kernel();
    env_pop_tf(&e->env_tf);
}
```

代码实现：

按照要求分别设置 lock_kernel()和 unlock_kernel()，需要注意的是在 env_run()中的 unlock_kernel()

DEBUG：

这里纠正之前 lab 的错误 ,就算 e 是 curenv ,e 的 env_status 还是得设置成 ENV_RUNNING 而不是 ENV_RUNABLE，否则当没有其他 env 能运行时，curenv 也不能运行（虽然这个是 Exercise5 的错误，但是这里一并说了）

Exercise 4.1. Implement ticket spinlock in kern/spinlock.c. You can define a macro USE_TICKET_SPIN_LOCK at the beginning of kern/spinlock.h to make it work. After you correctly implement ticket spinlock and define the macro, your code should pass spinlock_test(). Don`t use ticket spinlock before you implement all the rest exercises. Ticket spinlock may cause stressshed and primes fail because of its poor performance. (Optional) Consider why ticket spinlock is slow here.

```
// Check whether this CPU is holding the lock.
static int holding(struct spinlock *lock)
{
    // 检查持有 lock 的 cpu 是否是当前 cpu
    // 释放状态下 lock->own == lock_next，但是释放状态下 lock->cpu == 0 != thiscpu 该条件包含在之前的条件中
    //if(lock->own == lock->next) lock->own = lock->next = 0;
    return (lock->cpu == thiscpu);
}
```

```
void __spin_initlock(struct spinlock *lk, char *name)
{
    lk->own = lk->next = 0;
}

// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void spin_lock(struct spinlock *lk)
{
    int own = atomic_return_and_add(&lk->next, 1);
    while (atomic_return_and_add(&lk->own, 0) != own)
        asm volatile ("pause");
}

// Release the lock.
void spin_unlock(struct spinlock *lk)
{
    atomic_return_and_add(&lk->own, 1);
}
```

这里只贴 USER_TICKET_LOCK 部分

spin_lock（自旋锁）的实现方法简介：

lock 有两个值，own 和 next。own 表示锁当前的号码，next 表示等待的序号（和餐厅叫号一样）

当 own 和 next 相等，说明没有人在等待，也即这个锁是 free 的

拿锁时，把当前的 next 作为序号给拿锁者，next 自增长（这样下一个拿锁的人的序号就是当前 next+1），拿锁者只有当 own 和自己拿到的序号相等时才能拿锁（相当于被叫号）

放锁时，own 自增长

代码中值得注意的地方：

1.释放状态下 lock->own == lock_next，但是释放状态下 lock->cpu == 0 != thiscpu 该条件包含在之前的条件中

2.如果担心 own 或 next 超过 int 的最大值 ,可以在空闲时把 own 和 next 重新置为 0（见注释代码）

3.读取 lk->own 不是原子操作，可能违反 atomic 被篡改（事实上的确有问题），所以需要用 atomic_return_and_add 来包装

Exercise 5. Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield().

```
// Choose a user environment to run and run it.
void sched_yield(void)
{
    struct Env *idle;
    int i;
    // curenv->env_id 不是 数组中的下标，初始都为 0，应该为 ENVX(curenv->env_id )
    /*uint32_t now_id, env_id;
    if(curenv != NULL)
    {
        env_id = ENVX(curenv->env_id);
        for(now_id = env_id+1; now_id != env_id; now_id++)
        {
            if(now_id == NENV) now_id =0;
            if(envs[now_id].env_status == ENV_RUNNABLE && envs[now_id].env_type != ENV_TYPE_IDLE) env_run(envs+now_id);
        }
```

```
        // If no envs are runnable, but the environment previously running
on this CPU is still ENV_RUNNING, it's okay to choose that environment.
        if(curenv->env_status == ENV_RUNNING) env_run(curenv);
    }*/

    uint32_t env_id;
    if(curenv != NULL)
    {
        env_id = ENVX(curenv->env_id);
        for(i = (env_id+1)%NENV; i != env_id; i = (i+1)%NENV)
            if(envs[i].env_status == ENV_RUNNABLE &&
envs[i].env_type != ENV_TYPE_IDLE) env_run(&envs[i]);
        if(curenv->env_status == ENV_RUNNING) env_run(curenv);
    }
    ......
}

// Dispatches to the correct kernel function, passing the arguments.
int32_t syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3,
uint32_t a4, uint32_t a5)
{
    switch (syscallno){
```

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

这里贴出两段在我看来逻辑等价的代码，但是运行起来之前那个会卡在 free 最后一个 env 时，这个问题有点困惑

要求实现 Round-robin 式的选择程序，即从当前 env 下个到 envs 数组的末尾，再从头到当前 env 这个顺序寻找有没有可执行的 env

Round-robin 的好处是不会一个 env 一直运行而不给别的 env 运行机会，每个 env 的运行机会等价，总能顺序执行每一个 env

代码实现：

在 syscall 中注册一下 SYS_yield

envs 数组中从 now_id（当前 env 在 envs 中的 id）+1 开始到 NENV，再从 0 到 now_id-1 检查他们是不是 ENV_RUNNABLE（可执行）的，但是前提是他们不能是 IDLE

最后检查 curenv（即 envs[now_id]），对 curenv 没有 IDLE 这个限制，这个主要是因为在最初阶段运行 curenv 是 IDLE 的）

DEBUG：

如果之前 Exercise3 中 env_run 代码不修改(如果 e 就是 curenv，那把 curenv 的 env_status 设置成 ENV_RUNNABLE)，那么这里的 curenv 在第一次 env_run(curenv)之后下一次就不可能再选中自己。

假设我们只有一个 env 在运行，那么即使当前的 env 其实是可运行的，我们仍然无法运行这个 env 因为他是 ENV_RUNNABLE 的而不是 ENV_RUNNING 的(如果说我们有多个可运行 env 则不会发生错误，因为不会发生 env_run(curenv)的情况

当然如果我们把判断条件改为 curenv->env_status == ENV_RUNNABLE 也是错误的

假设我们当前只有一个可运行 env，在第一次 env_run(curenv)之前，其 env_status 是 ENV_RUNNING，那么就不符合判断条件而不可能 env_run(curenv)了

另外，一个在运行的 env 的 status 如果是 ENV_RUNNABLE 而不是 ENV_RUNNING 不是很奇怪么

Exercise 6. Implement the system calls described above in kern/syscall.c. 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.

kern/trapentry.S:

```
TRAPHANDLER_NOEC(entry48, T_SYSCALL);
```

kern/trap.c:

```
void trap_init(void)
{
    ……
    extern void entry48();
    SETGATE(idt[T_SYSCALL],      0, GD_KT, entry48,      3);
    ……
}
```

```
static void trap_dispatch(struct Trapframe *tf)
{
    ......
    switch(tf->tf_trapno)
    {
        ......
        case T_SYSCALL:
            tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax,
                                          tf->tf_regs.reg_edx,
                                          tf->tf_regs.reg_ecx,
                                          tf->tf_regs.reg_ebx,
                                          tf->tf_regs.reg_edi,
                                          0);
            return;
        ......
    }
    ......
}
```

kern/syscall.c:

```
int32_t syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3,
uint32_t a4, uint32_t a5){
    switch (syscallno){
        ……
        case SYS_page_alloc:
            return sys_page_alloc(a1, (void *)a2, a3);
        case SYS_page_map:
            return sys_page_map((envid_t)*((uint32_t*)a1),
(void*)*((uint32_t*)a1+1), (envid_t)*((uint32_t*)a1+2),
(void*)*((uint32_t*)a1+3), (int)*((uint32_t*)a1+4));
        case SYS_page_unmap:
            return sys_page_unmap(a1, (void *)a2);
        case SYS_env_set_status:
            return sys_env_set_status(a1, a2);
        case SYS_exofork:
            return sys_exofork();
        case SYS_env_set_pgfault_upcall:
            return sys_env_set_pgfault_upcall(a1, (void *)a2);
        ……
    }
}
```

```
void my_syscall(struct Trapframe *tf){
    lock_kernel();
    ......
    unlock_kernel();
    return;
}

static envid_t sys_exofork(void)
{
    struct Env *e;

    int ret = env_alloc(&e, curenv->env_id);
    if (ret < 0) return ret;

    e->env_status = ENV_NOT_RUNNABLE;
    e->env_tf = curenv->env_tf;
    // eax contains the return val
    e->env_tf.tf_regs.reg_eax = 0;
    return e->env_id;
}
```

```
static int sys_env_set_status(envid_t envid, int status)
{
	struct Env *e;

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

static int sys_page_alloc(envid_t envid, void *va, int perm)
{
	struct Env *e;

	//   -E_INVAL if va >= UTOP, or va is not page-aligned.
	//   -E_INVAL if perm is inappropriate (see above).
	if (va >= (void*)UTOP || (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P) ||
PGOFF(va) != 0 || (perm & (~PTE_SYSCALL)) != 0)
		return -E_INVAL;
	//   -E_BAD_ENV if environment envid doesn't currently exist, or the
caller doesn't have permission to change envid.
```

```
    // You should set envid2env's third argument to 1, which will
    // check whether the current environment has permission to set
    // envid's status.
    if (envid2env(envid, &e, 1) < 0) return -E_BAD_ENV;
    struct Page *p = page_alloc(ALLOC_ZERO);

    //   -E_NO_MEM if there's no memory to allocate the new page
    if (!p) return -E_NO_MEM;
    //   -E_NO_MEM if there's no memoryto allocate any necessary page
tables
    if (page_insert(e->env_pgdir, p, va, perm) < 0)
    {
        page_free(p);
        return -E_NO_MEM;
    }

    memset(page2kva(p), 0, PGSIZE);

    return 0;
}
```

```
static int sys_page_map(envid_t srcenvid, void *srcva,
        envid_t dstenvid, void *dstva, int perm){
    pte_t* pte;
    struct Env* s_env;
    struct Env* d_env;
    // 判断是否在 UTOP 之上或者没对齐
    if (srcva >= (void*)UTOP || ROUNDUP(srcva,PGSIZE) != srcva ||
dstva >= (void*)UTOP || ROUNDUP(dstva,PGSIZE) != dstva ) return -
E_INVAL;
    // 没有权限
    if ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) return -E_INVAL;
    if ((perm & (~PTE_SYSCALL)) != 0 ) return -E_INVAL;
    if(envid2env(srcenvid, &s_env, 1) < 0 || envid2env(dstenvid, &d_env,
1) < 0) return -E_BAD_ENV;

    struct Page* p = page_lookup(s_env->env_pgdir, srcva, &pte);
    if (p == NULL) return -E_INVAL;
    if ((perm & PTE_W) != 0 && ((*pte) & PTE_W) == 0 ) return -E_INVAL;
    if(page_insert(d_env->env_pgdir, p, dstva, perm) < 0) return -
E_NO_MEM;
    return 0;
}
```

```
static int sys_page_unmap(envid_t envid, void *va)
{
	struct Env *e;

	// You should set envid2env's third argument to 1, which will
	// check whether the current environment has permission to set
	// envid's status.
	if (envid2env(envid, &e, 1) < 0) return -E_BAD_ENV;

	if (va >= (void*)UTOP || ROUNDUP(va, PGSIZE) != va) return -E_INVAL;

	page_remove(e->env_pgdir, va);

	return 0;
}
```

lib/syscall.c

```
int sys_page_map(envid_t srcenv, void *srcva, envid_t dstenv, void *dstva,
int perm){
    //return syscall(SYS_page_map, 1, srcenv, (uint32_t) srcva, dstenv,
(uint32_t) dstva, perm); should be attentioned in the lab which the fifth
param will alyways be 0 because we don't have enough registers so we
decide to make the first param an array
    uint32_t arglist[5];
    arglist[0] = (uint32_t) srcenv;
    arglist[1] = (uint32_t) srcva;
    arglist[2] = (uint32_t) dstenv;
    arglist[3] = (uint32_t) dstva;
    arglist[4] = (uint32_t) perm;
    return syscall(SYS_page_map, 1, (uint32_t)arglist, 0, 0, 0, 0);
}
```

值得注意的地方：

1.lib/syscall.c 中我们发现，只有 sys_exofork 是 inline 在 inc/lib.h 里的，exofork 和其他 syscall 不同不走 lib/syscall.c，而是和 trap 类似的过程做的处理，所以需要在 trapentry.S 中声明，并在 trap_init 里注册 idt，trapdispatch 中截获处理（虽然非常不美观），当然不需要在结束后 unlock_kernel，trap 函数会做好 lock_kernel 和 unlock_kernel

2.数值寄存器数量只有 5 个，不能装下 6 个参数，a5 的值为 0，而 sys_page_map 有 5 个参数，需要修改 lib/syscall.c，把 5 个参数作为 1 个数组传入再在 kern/syscall.c 中解压（虽然非常不美观）

3.envid2env 这个函数如果第三个参数设为 1，那么会自动检测是否是 bad environment

4.为了保证 syscall 的顺序性以及安全性，我们需要在 my_syscall 的开头和结尾分别调用 lock_kernel 和 unlock_kernel，否则会出现：1.在 DEBUG_LOCK 开启时，tf（cs 寄存器）被篡改，无法从 kerenl 态回到 user 态，在 trap 函数中因为不是 user 态没有拿锁，env_run 时多放锁导致触发 DEBUG_LOCK 的检测从而发生 kerenl page fault；2.之后 primes 会顺序紊乱，出现问题

5.之前 lab 的问题，基本会集中在这里爆发（有部分问题不会）

代码实现：

针对 sys_exofrok 和 sys_page_map 进行修改 ,修改 my_syscall

在 kern/syscall.c 里的 syscall 函数注册相应的调用

sys_exofork 包装 env_alloc，由于新创建的 env 是空壳无法跑，得把其 env_status 设置成 ENV_NOT_RUNNABLE，等待之后 user 调整完以后通过 sys_env_set_status 变成 ENV_RUNNABLE，因为是 fork ,所以 tf(状态)设置成和 curenv 一样 ,但把 tf 中的 reg_eax 设为 0 作为子进程的返回值（env_id），而父进程返回 e->env_id

sys_env_set_status 通过 envid2env 利用 envid 找出 e 并把其 env_status 置为 status

env_page_alloc 包装了 page_alloc 和 page_insert ,并把 page 内容置为 0，insert 不成功时需要 page_free

sys_page_map 包装了 page_lookup 和 page_insert，通过 srcenvid 和 dstenvid 取出 srcenv 和 dstenv，从 srcenv 中取出位于 srcva 的 page，并 insert 到 dstenv 的 dstva 中

sys_page_unmap 包装了 page_remove，用 envid 取出 e，且把 e 中 va 位置的 page 删除

PART B Copy-on-Write Fork

这个 Part 的核心就是完成一个 copy-on-write 的 fork，即写时拷贝的 fork

起因是因为 fork 出的子进程绝大多数都会直接 excv 别的进程，和父进程执行相同代码的子进程毕竟是少数

如果是完整的 fork 把父进程的资源全部复制一遍的话，那么会造成大量的资源浪费（资源复制时间的浪费，复制出来的资源没有被使用），造成 kerenl 的低效率

鉴于这个原因，需要实现一个 copy-on-write 的 fork（写时拷贝）

copy-on-write 的 fork 的核心思想是 child 除了 UXSTACK 之外，child 只把 parent 的内存映射到自己的内存上，并设为 COW（即可读但是写时拷贝），但是不设置自己的内存。只有当试图写这些内存中的某个 page 时，才会触发 page_fault 并调用注册的 page_fault_handler 去重新拷贝一份 page

整个执行过程：

1.用户调用 fork

2. 利 用 set_pagefault_handler 设 置 pfentry.S 中 的 _pagefault_handler 入 口 为 pgfault ， 并 用 sys_env_set_pgfault_upcall 设 置 env_pagefault_upcall 为 _pagefault_upcall（ _pagefault_upcall 中调用_pagefault_handler）

3. fork 中用之前的 sys_exofork 创建子进程

4.用 duppage 映射内存，设置子进程的 env_pagefualt_upcall（同样为_pagefault_upcall）和 UXSTACK

5.把子进程的状态设为 ENV_RUNNABLE 并返回

6.在发生 SYS_PGFLT 时，被 trap 截住进入 trapdispatch 再进入 page_fault_handler，设置处理的 UXSTACK，把 eip 指向之前设置的 env_pgfault_upcall（即_pgfault_upcall），_pgfault_upcall 再调用 pgfalut 处理之后返回

Exercise 7. 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.

```
static int sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
	struct Env *e;
	if(envid2env(envid, &e, 1) < 0) return -E_BAD_ENV;
	// register environment's page fault entrypoint
	e->env_pgfault_upcall = func;
	return 0;
}
int32_t syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
	switch (syscallno){
	......
	    case SYS_env_set_pgfault_upcall:
	        return sys_env_set_pgfault_upcall(a1, (void *)a2);
	.....
	}
}
```

代码实现：

在 syscall 中注册 SYS_env_set_pgfault_upcall

用 envid2env 从 envid 中取出 e，同时检查

把 env_pgfault_upcall 的位置指向 func（即 pgfault 的处理函数入口）

Exercise 8. 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. (What happens if the user environment runs out of space on the exception stack?)

```
void page_fault_handler(struct Trapframe *tf)
{
	uint32_t fault_va;
	// Read processor's CR2 register to find the faulting address
	fault_va = rcr2();
	// Handle kernel-mode page faults.
	// Trapped from kernel mode, panic
	// CPL 是 CS 的最低两位
	if((tf->tf_cs & 3) != 3) panic("Kernel page fault!");

	// if there exists an environment's page fault upcall
	if (curenv->env_pgfault_upcall)
	{
		struct UTrapframe *user_tf = NULL;
		// page fault in another page fault
		if (UXSTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp < UXSTACKTOP) user_tf = (struct UTrapframe *)(tf->tf_esp - 4 - sizeof(struct
```

```
UTrapframe));
        // a page fault from user environment
        else if (USTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp <
USTACKTOP) user_tf = (struct UTrapframe *)(UXSTACKTOP - sizeof(struct
UTrapframe));
        // stack overflow
        else ;
        if(user_tf != NULL)
        {
            user_mem_assert(curenv, user_tf, sizeof(user_tf), PTE_U |
PTE_W);
            // set the user trap frame to record the tf
            user_tf->utf_fault_va = fault_va;
            user_tf->utf_err = tf->tf_err;
            user_tf->utf_regs = tf->tf_regs;
            user_tf->utf_eip = tf->tf_eip;
            user_tf->utf_eflags = tf->tf_eflags;
            user_tf->utf_esp = tf->tf_esp;
            // modify tf to change what the user environment runs
            // modify the tf_esp pointing to the user stack
            tf->tf_esp = (uintptr_t) user_tf;
            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",
        curenv->env_id, fault_va, tf->tf_eip);
    print_trapframe(tf);
    env_destroy(curenv);
}
```

代码实现：

首先需要检测有没有 env_pgfault_upcall（即有没有办法处理 pgfault）

如果是 tf_esp 是在 USTACK 中，说明是从 user 态发生的 page_fault，这时需要在 UXSTACKTOP 开出一块空间放 user_tf

如果 tf_esp 已经在 UXSTACK 中，说明是在处理一个 page_fault 时发生了另一个 page_fault，这时需要在 tf_esp 下开出另一块空间放 user_tf

如果 tf_esp 不在 UXSTACK 或 USTACK 中，说明 page_fault 过多导致 overflow，那就保持 user_tf 为 NULL，说明 overflow，不处理 pgfault

如果 tf_esp 不是 NULL，那么就把 tf 的信息存储在 user_tf 中，再让 tf 指向 user_tf，修改 tf_eip 指向 env_pgfault_upcall，再运行 curenv 使其执行 env_pgfault_upcall（至于最后的弹栈工作，交给_pgfault_upcall 处理）

Exercise 9. 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.

```
.text
.globl _pgfault_upcall
_pgfault_upcall:
    // Call the C page fault handler.
    pushl %esp              // function argument: pointer to UTF
    movl _pgfault_handler, %eax
    call *%eax
    addl $4, %esp           // pop function argument

    movl 0x30(%esp), %eax // trap-time esp
    movl 0x28(%esp), %ebx // trap-time eip
    subl $0x4, %eax         // add and sub cannot be done after popfl, so deal the sub to stack pointer here
    movl %ebx, (%eax)       // build return address
    movl %eax, 0x30(%esp)

    // Restore the trap-time registers.  After you do this, you
```

```
// can no longer modify any general-purpose registers.
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.
addl $4, %esp
popfl

// Switch back to the adjusted trap-time stack.
movl (%esp), %esp

// Return to re-execute the instruction that faulted.
ret
```

代码实现：

按照要求一步一步来，把 return address 设为 traptime eip，之后把 register 和 eflag 给 pop 出来，最后 ret

Exercise 10. Finish set_pgfault_handler() in lib/pgfault.c

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

    if (_pgfault_handler == 0) {
        // First time through!
        // get the current envid
        envid_t envid = sys_getenvid();

        // alloc the exception to current env with permmision
        r = sys_page_alloc(envid, (void*)(UXSTACKTOP-PGSIZE), PTE_U | PTE_P | PTE_W);
        if(r < 0) panic("set_pgfault_handler %e\n",r);

        // tell the kernel to call the assembly-language _pgfault_upcall routine when a page fault occurs.
        sys_env_set_pgfault_upcall(envid, _pgfault_upcall);
    }
    // Save handler pointer for assembly to call.
    _pgfault_handler = handler;
}
```

代码实现：

如果是第一次设置_pgfault_handler 那么先开一块 UXSTACK，再设置一下 envid 的 env_pgfault_upcall 为 _pgfault_upcall（_pgfault_handler 是在_pgfault_upcall 中被调用的）

最后把_pgfault_handler 设为 handler

Exercise 11. Implement fork, duppage and pgfault in lib/fork.c.

```
static void pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    int r;
    // examin the err a write fault
    if ((err & FEC_WR) == 0) panic("[lib/fork.c pgfault]: not a write fault!");
    // extern volatile pde_t vpd[];        // VA of current page directory
    // examin the page directory exists (although in my situation, we can check all bits but not just PTE_P)
    if ((vpd[PDX(addr)] & PTE_P) == 0) panic("[lib/fork.c pgfault]: page directory not exists!");
    // extern volatile pte_t vpt[];        // VA of "virtual page table"
    // examin the fault a copy on write fault
    if ((vpt[PGNUM(addr)] & PTE_COW) == 0) panic("[lib/fork.c pgfault]: not a copy-on-write fault!");

    envid_t envid = sys_getenvid();
    // allocate a new page and map it at PFTEMP
    r = sys_page_alloc(envid, (void*)PFTEMP, PTE_U | PTE_W | PTE_P);
```

```
    if(r < 0) panic("[lib/fork.c pgfault]: alloc temporary location failed %e",
r);

    // void *memmove(void *dest, const void *src, size_t n);
    // copy the data form old page va to the new page PFTEMP
    void* va = (void*)ROUNDDOWN(addr, PGSIZE);
    memmove((void*)PFTEMP, va, PGSIZE);

    // move the new page to the old page's address
    r = sys_page_map(envid, (void*)PFTEMP, envid, va, PTE_U | PTE_W |
PTE_P);
    if(r < 0) panic("[lib/fork.c pgfault]: %e", r);

    r = sys_page_unmap(envid, (void *) PFTEMP);
    if(r < 0) panic("[lib/fork.c pgfault]: %e", r);
}
```

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

    void* addr = (void*)(pn * PGSIZE);
    pde_t pde = vpd[PDX(addr)];;
    pte_t pte = vpt[PGNUM(addr)];

    if ((uint32_t)addr >= UTOP) panic("duppage: duplicate page above UTOP!");
    if ((pte & PTE_P) == 0) panic("[lib/fork.c duppage]: page table not present!");
    if ((pde & PTE_P) == 0) panic("[lib/fork.c duppage]: page directory not present!");

    //If the page is writable or copy-on-write
    if (pte & (PTE_W | PTE_COW))
    {
        // the new mapping must be created copy-on-write
        r = sys_page_map(sys_getenvid(), addr, envid, addr, PTE_U | PTE_P | PTE_COW);
        if (r < 0) panic("[lib/fork.c duppage]: map page copy on write %e",
```

```
r);


        // remarked our map with permission copy on write
        r = sys_page_map(sys_getenvid(), addr, sys_getenvid(), addr,
PTE_U | PTE_P | PTE_COW);
        if (r < 0) panic("[lib/fork.c duppage]: map page copye on write %e",
r);
    }
    else
    {
        r = sys_page_map(sys_getenvid(), addr, envid, addr, PTE_U | PTE_P);
        if (r < 0) panic("[lib/fork.c duppage]:map page in read only %e",
r);
    }

    return 0;
}
```

```
envid_t fork(void)
{
	extern void _pgfault_upcall (void);
	int r;
	int pn;

	// Set up our page fault handler appropriately.
	set_pgfault_handler(pgfault);

	envid_t forkid;
	// Create a child.
	forkid = sys_exofork();

	if (forkid < 0) panic("fork error:%e",forkid);

	// whether the fork id is a child
	else if (forkid == 0)
	{
		// fix "thisenv" in the child process.
		thisenv = &envs[ENVX(sys_getenvid())];
		return 0;
	}
```

```
    // Copy our address space to the child.
    // the structure can be seen in inc/memlayout.h
    for (pn = UTEXT/PGSIZE; pn < UTOP/PGSIZE; pn++)
    {
        // Neither user exception stack should ever be marked copy-on-write
        if (pn == (UXSTACKTOP-PGSIZE) / PGSIZE) continue;
        // We should make sure the page table directory and the page table exist and the page table can enter
        if (((vpd[pn/NPTENTRIES] & PTE_P) != 0) && ((vpt[pn] & PTE_P) != 0) && ((vpt[pn] & PTE_U) != 0)) duppage(forkid, pn);
    }

    // set page fault stack of the child
    r = sys_page_alloc(forkid, (void *)(UXSTACKTOP-PGSIZE), PTE_U | PTE_W | PTE_P);
    if(r < 0) panic("[lib/fork.c fork]: exception stack error %e\n",r);

    // set page fault handler of the child.
    r = sys_env_set_pgfault_upcall(forkid, (void *)_pgfault_upcall);
    if(r < 0) panic("[lib/fork.c fork]: pgfault_upcall error %e\n",r);
```

```
	// mark the child status as runnable
	r = sys_env_set_status(forkid, ENV_RUNNABLE);
	if(r < 0) panic("[lib/fork.c fork]: status error %e\n",r);

	return forkid;
}

int page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
	// Fill this function in
	pte_t* pte;
	// find whether va is in pgdir
	 struct Page* pg=page_lookup(pgdir,va,&pte);
	 // if va is in pgdir
	 if(pg==pp)
	{
		// find the virtual address of page table entry of va in pgdir
		pte = pgdir_walk(pgdir,va,1);

		// Reexamin in lab4 that we should invalidate va in tlb !!!!!
		tlb_invalidate(pgdir, va);
```

```
        // reset the permission
        *pte = page2pa(pp) | perm | PTE_P;
        return 0;
    }
    // remove va from pgdir
    else if(pg) page_remove(pgdir,va);

    // now we can be sure that va is not in pgdir
    pte = pgdir_walk(pgdir,va,1);

    if(!pte) return -E_NO_MEM;

     // set the permission
    *pte = page2pa(pp) | perm | PTE_P;
    pp->pp_ref++;

    return 0;
}
```

代码实现：

pgfault：新开一个 page，把原内容拷贝一份，最后放回原处。具体过程为：检查权限，在 PFTEMP 新开一块 page，权限为 PTE_U | PTE_W（用户可读写的，原来的是 PTE_COW），把 va 处的原内容拷到 PFTEMP，把 PFTEMP 这个 page 重新映射回 va 处，最后解除在 PFTEMP 处的映射

duppage：检查 page 权限，如果是 PTE_W | PTE_COW（可写或写时拷贝的）那就把当前 env 的 page 映射到目标 env(envid)处，并设为 COW 的，之后再把自己的 page 重设为 COW 的（如果是 PTE_W）。如果不是 PTE_W | PTE_COW，那么就是只读的，映射时设置权限为 PTE_U

fork：先利用 sys_exofork 创建一个 child env，把返回值放在 forkid 中。如果是 child env，那 forkid 是 0，修改 thisenv 之后即可返回，如果不是 0，说明是父进程，要用 duppage 把内存映射到子进程(除了 UXSTSCK)，创建子进程的 UXSTACK，_pgfault_upcall，最后把子进程的 env_status 设为 ENV_RUNNABLE

DEBUG：

之前的 lab 忘记在 page_insert 中加入 tlb_invalidate，导致 tlb 缓存中的 page 的 perm 和实际 page 的 perm 不相符出错（即 PTE_W 和 PTE_COW 的不同），另一种解决办法是无论之前存不存在 page，一律先 page_remove

PART C Preemptive Multitasking and Inter-Process communication (IPC)

这个 Part 主要完成 IRQ（Interrupt Request）和进程间通信（IPC）。

到目前为止，我们发现，如果一个 env 一直在执行而不 sched 的话，我们是没有办法从该进程那里拿回主动权的（如 user/spin.c）。所以我们需要实现 IRQ（Interrupt Request）

如果我们没有 IRQ，那么我们的 IPC 将无法实现，因为 IPC 的 send 方当 receive 方没有收到时会无限 resend，如果没有 IRQ，那么 receive 方将因为 os 一直在运行 send 而无法收到消息，os 将会卡死。

另外需要注意的是，接收中断有两个层面，一个是硬件层面接收中断，另一个是 os 层面接收中断

Exercise 12. 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 inenv_alloc() in kern/env.c to ensure that user environments are always run with interrupts enabled.

kern/trapentry.S

```
// The IDT entries 32-47 correspond to the IRQs 0-15.
/*
// Hardware IRQ numbers. We receive these as (IRQ_OFFSET+IRQ_WHATEVER)
#define IRQ_TIMER        0
#define IRQ_KBD          1
#define IRQ_SERIAL       4
#define IRQ_SPURIOUS     7
#define IRQ_IDE         14
#define IRQ_ERROR       19
*/
TRAPHANDLER_NOEC(irq0, IRQ_OFFSET+IRQ_TIMER);
TRAPHANDLER_NOEC(irq1, IRQ_OFFSET+IRQ_KBD);
TRAPHANDLER_NOEC(irq2, IRQ_OFFSET+2);
TRAPHANDLER_NOEC(irq3, IRQ_OFFSET+3);
TRAPHANDLER_NOEC(irq4, IRQ_OFFSET+IRQ_SERIAL);
```

```
TRAPHANDLER_NOEC(irq5, IRQ_OFFSET+5);
TRAPHANDLER_NOEC(irq6, IRQ_OFFSET+6);
TRAPHANDLER_NOEC(irq7, IRQ_OFFSET+IRQ_SPURIOUS);
TRAPHANDLER_NOEC(irq8, IRQ_OFFSET+8);
TRAPHANDLER_NOEC(irq9, IRQ_OFFSET+9);
TRAPHANDLER_NOEC(irq10, IRQ_OFFSET+10);
TRAPHANDLER_NOEC(irq11, IRQ_OFFSET+11);
TRAPHANDLER_NOEC(irq12, IRQ_OFFSET+12);
TRAPHANDLER_NOEC(irq13, IRQ_OFFSET+13);
TRAPHANDLER_NOEC(irq14, IRQ_OFFSET+IRQ_IDE);
TRAPHANDLER_NOEC(irq15, IRQ_OFFSET+15);
TRAPHANDLER_NOEC(irq19, IRQ_OFFSET+IRQ_ERROR);
```

kern/trap.c

```
void trap_init(void)
{
    ......
    extern void irq0();
    extern void irq1();
    extern void irq2();
    extern void irq3();
    extern void irq4();
```

```
extern void irq5();

extern void irq6();

extern void irq7();

extern void irq8();

extern void irq9();

extern void irq10();

extern void irq11();

extern void irq12();

extern void irq13();

extern void irq14();

extern void irq15();

extern void irq19();

……

SETGATE(idt[IRQ_OFFSET+IRQ_TIMER],      0, GD_KT, irq0,      0);

SETGATE(idt[IRQ_OFFSET+IRQ_KBD],        0, GD_KT, irq1,      0);

SETGATE(idt[IRQ_OFFSET+2],              0, GD_KT, irq2,      0);

SETGATE(idt[IRQ_OFFSET+3],              0, GD_KT, irq3,      0);

SETGATE(idt[IRQ_OFFSET+IRQ_SERIAL],         0, GD_KT, irq4,  0);

SETGATE(idt[IRQ_OFFSET+5],              0, GD_KT, irq5,      0);

SETGATE(idt[IRQ_OFFSET+6],              0, GD_KT, irq6,      0);

SETGATE(idt[IRQ_OFFSET+IRQ_SPURIOUS], 0, GD_KT, irq7,        0);

SETGATE(idt[IRQ_OFFSET+8],              0, GD_KT, irq8,   0);
```

```
    SETGATE(idt[IRQ_OFFSET+9],              0, GD_KT, irq9,      0);
    SETGATE(idt[IRQ_OFFSET+10],             0, GD_KT, irq10,     0);
    SETGATE(idt[IRQ_OFFSET+11],             0, GD_KT, irq11,     0);
    SETGATE(idt[IRQ_OFFSET+12],             0, GD_KT, irq12,     0);
    SETGATE(idt[IRQ_OFFSET+13],             0, GD_KT, irq13,     0);
    SETGATE(idt[IRQ_OFFSET+IRQ_IDE],        0, GD_KT, irq14,     0);
    SETGATE(idt[IRQ_OFFSET+15],             0, GD_KT, irq15,     0);
    SETGATE(idt[IRQ_OFFSET+IRQ_ERROR],          0, GD_KT, irq19, 0);
    ......
}

kern/pmap.c
int env_alloc(struct Env **newenv_store, envid_t parent_id)
{
    ......
    // FL_IF: interrupt flag,which means the env can be interrupted
    e->env_tf.tf_eip = e->env_tf.tf_eip | FL_IF;
    ......
}
```

代码实现：

和 trap 一样，在 trapentry.S 中声明，在 trap.c 中注册,并在 env_alloc 中开启 FL_IF，接收中断

Exercise 13. Modify the kernel's trap_dispatch() function so that it calls sched_yield() to find and run a different environment whenever a clock interrupt takes place.

kern/trap.c

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

    ......

    switch(tf->tf_trapno) {

        ......

        case IRQ_OFFSET + IRQ_TIMER:

            lapic_eoi();      // acknowledge the interrupt using lapic_eoi()

            sched_yield();  // calling the scheduler

            return;

        ......

    }

    ......

}
```

kern/trapentry.S

```
.globl sysenter_handler;

.type sysenter_handler, @function;

.align 2;

sysenter_handler:

    ……

    //sysenter will automatically disable interrupt, but sysexit won't recover it

    sti

    sysexit
```

代码实现：

在 trap_dispatch 中截获 IRQ_TIMER，用 lapic_eoi 承认该 interrupt，再用 sys_yield()转让运行权

DEBUG:

首先需要知道一件事，接收中断有两个层面，一个是硬件层面接收中断，另一个是 os 层面接收中断，打开 FL_IF 只是 os 层面接收中断，但是如果硬件层面没有接收中断，那么即使 os 的 FL_IF 开着也是没有用的，因为 os 本身无法探知中断

sysenter 会自动包括 cli（即关闭硬件层面的中断），但是 sysexit 并不会开启硬件层面的中断，需要我们自己调用 sti 开启。这点需要特别注意！！！！！

如果不用 sti 那么在单核的情况下，一旦父进程使用了 sys_yield，他将关闭硬件层面的中断，那么子进程就永远收不到中断了（当然多核的情况下，其他核的中断仍然是可用的）

Exercise 14. 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.

kern/syscall.c

```
int32_t syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5){
    switch (syscallno){
        ......
        case SYS_ipc_recv:
            return sys_ipc_recv((void*)a1);
        case SYS_ipc_try_send:
            return sys_ipc_try_send((envid_t)a1, a2, (void*)a3, (int)a4);
        ......
    }
}
```

```
static int sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva,
unsigned perm)
{
    int r;
    pte_t* pte;
    struct Env* dstenv;
    struct Page* p;
    if ((r = envid2env(envid, &dstenv, 0)) < 0) return -E_BAD_ENV;
    if (!dstenv->env_ipc_recving || dstenv->env_ipc_from != 0)
        return -E_IPC_NOT_RECV;
    if (srcva < (void*)UTOP)
    {
        if(ROUNDUP(srcva, PGSIZE) != srcva) return -E_INVAL;
        if ((perm & ~PTE_SYSCALL) != 0) return -E_INVAL;
        if ((perm & 5) != 5) return -E_INVAL;
        dstenv->env_ipc_perm = 0;
        p = page_lookup(curenv->env_pgdir, srcva, &pte);
        if (p == NULL || ((perm & PTE_W) > 0 && !(*pte & PTE_W) > 0))
            return -E_INVAL;
        if(page_insert(dstenv->env_pgdir,  p,  dstenv->env_ipc_dstva,
perm)<0) return -E_NO_MEM;
    }
```

```
    dstenv->env_ipc_recving = 0;
    dstenv->env_ipc_from = curenv->env_id;
    dstenv->env_ipc_value = value;
    dstenv->env_ipc_perm = perm;
    dstenv->env_tf.tf_regs.reg_eax = 0;
    dstenv->env_status = ENV_RUNNABLE;
    return 0;
}
static int sys_ipc_recv(void *dstva)
{
    if (ROUNDDOWN (dstva, PGSIZE) != dstva && dstva < (void*)UTOP)
        return -E_INVAL;
    curenv->env_status = ENV_NOT_RUNNABLE;
    curenv->env_ipc_dstva = dstva;
    curenv->env_ipc_from = 0;
    curenv->env_ipc_recving = 1;
    sched_yield();
    return 0;
}
```

lib/ipc.c

```
int32_t ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // for sfork challenge, we should repalce thisenv by envs+ENVX(sys_getenvid()) because thisenv is a global variable defined in lib/libmain.c which means that if we use thisenv in sfork, thisenv is write shared the parent's thisenv will be changed by child and point to the child's env
    if (!pg)
        pg = (void*)UTOP;
    int r = sys_ipc_recv(pg);
    if (r >= 0) {
        if(perm_store != NULL)
            *perm_store = envs[ENVX(sys_getenvid())].env_ipc_perm;
        if(from_env_store != NULL)
            *from_env_store = envs[ENVX(sys_getenvid())].env_ipc_from;
        return envs[ENVX(sys_getenvid())].env_ipc_value;
    }
    if(perm_store != NULL) *perm_store = 0;
    if(from_env_store != NULL) *from_env_store = 0;
    return r;
}
```

```
void ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    if(!pg) pg = (void*)UTOP;
    int r;
    while((r = sys_ipc_try_send(to_env,val,pg,perm)) != 0)
    {
        if(r != -E_IPC_NOT_RECV )
            //panic ("[lib/ipc.c ipc_send]: sys try send failed : %e", r);
            break;
    }
    sys_yield();
}
```

代码实现：

sys_ipc_try_send 获取 envid 所指代的 dstenv ,检查其中内容，如果还没收到，返回-E_IPC_NOT_RECV，如果收到了，那么把 page 插入 dstenv 中，并修改 dstenv 中的信息

sys_ipc_recv：修改自己的信息，把 env_ipc_from 置为 0，env_ipc_receiving 置为 1,（表示已经收到），同时把自己置为 ENV_NOT_RUNNABLE，并 yield 给其他进程

ipc_send ：如果 sys_ipc_try_send 一直不成功且是-E_IPC_NOT_RECV（即 dst 未收到），那么再次尝试，直到成功，再 yield 给其他进程

ipc_recv ：执行 sys_ipc_recv ，修改 perm_store 和 from_env_store 并返回 env_ipc_value