MIT6.828 Lab4: Preemptive Multitasking

Zhuofan Zhang

Augest 2020

Contents

Chapter 1

Multiprocessor Support and Cooperative Multitasking

1.1 Multiprocessor Support

本次 Lab 的内容是对 JOS 进行补充以提供多处理器支持，并实现任务调度功能。

JOS 实现的多核支持属于*对称多核支持 (Symmertric Multiprocessing, SMP)*，即所有处理器对资源（内存、IO 总线等）的访问是平等的。SMP 的概念是在系统初始化完成后建立的，在 Boot 阶段，仍然需要选择一个 CPU 作为*启动处理器 (Bootstrap Processor, BSP)*，用来初始化资源并启动 OS，最后启动其他的 CPU（称为*应用处理器 (Application Processor, AP)*）。对 BSP 的选择是由 BIOS 完成的，在当前实验阶段，我们相当于完成了 BSP 的启动内容。

在 SMP 体系中，每个 CPU 都拥有一个称为 LAPIC(local APIC) 的可编程中断单元，用于在系统中传递中断信息，并提供 CPU 的唯一识别信息。因此，与多个 CPU 的通讯依赖于 LAPIC 单元。

CPU 对 LAPIC 单元的访问使用的是一类特殊的 IO：memory-mapped I/O(MMIO)。MMIO 是一段硬连接到部分 IO 设备寄存器的物理地址区域，常用于访问设备的寄存器。JOS 的 VAS 在 MMIOBASE 处留有 4MB 的空间用于映射这段物理地址区域。

Exercise 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.

这个练习要求我们补全 mmio_map_region 函数。根据提示，这个函数用于将 MMIO 映射到上文提到的 VAS 指定区域，我们对其补全如下：

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    ...
    // Your code here:
```

```
    size = ROUNDUP(size, PGSIZE);
    if(base+size > MMIOLIM)
        panic("MMIO_MAP_REGION: out of range.");
    boot_map_region(kern_pgdir, base, size, pa, PTE_PCD|PTE_PWT|PTE_W);
    base += ROUNDUP(size, PGSIZE);
    return (void *)(base - size);
}
```

Application Processor Bootstrap

在完成对 MMIO 的映射后，可以开始对 AP 的启动。先启动的 BSP 必须能够获取 APs 的信息，如 CPU 数量、APIC IDs 等。这些信息被保留在 BIOS 中，kern/mpconfig.c/mp_init() 将其读出。

kern/init.c/boot_aps() 是整个流程的核心函数，它将 AP 的入口程序加载到内存，并发送启动信号。AP 的入口程序被加载到任意可用的低位 640k 地址（JOS 将其加载到 0x7000），它与 BSP 的入口程序有一定的区别。

AP 被启动时处于实模式，在经过与 BSP 相似的初始化后调用 mp_main() 进行进一步设置。

Exercise 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.

init.c/i386_init() 调用 boot_aps() 用于启动 APs，这个函数再将 AP 入口加载到内存中，调用 lapic_startap() 与 APs 通讯，AP 将执行入口代码（mpentry.S），并跳转控制至 init.c/mp_main()；由于 VAS 为 MMIO 提供一段区域映射，因此在页表初始化阶段需将这段区域的页表剔除出空闲页表，因此修改 page_init() 函数：

```
void
page_init(void)
{

    size_t i;
    // for situation(1)
    pages[0].pp_ref = 1;
    // for situation(2)
    size_t pages_in_ap_entry = MPENTRY_PADDR / PGSIZE;
    for (i = 1; i < npages_basemem; i++)
    {
        // Lab 4 code: to avoid mapping at the AP entry codes.
```

```
        if(i == pages_in_ap_entry)
        {
            pages[i].pp_ref = 1;
            continue;
        }

        pages[i].pp_ref = 0;
        // At the first time, the page_free_list will be 'NULL'
        // When pp.link = NULL, it means that no more free_page after
        // this node.
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    // for situation(3)
    ...
}
```

Question

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?

AP 的入口程序与 BSP 的入口程序的区别主要有两点：(1)AP 入口程序不需要重复使能 A20 地址线；(2) 加载变量和跳转的地址计算模式不同，AP 入口程序使用了 MPBOOTPHYS。这个宏的目的是计算正确的物理地址（与链接地址不同，见 kernel.asm）。如果直接使用 boot.c，将加载错误的 GDT 或进行错误的跳转。

Per-CPU State and Initialization

支持多处理器的 OS 需要区分处理器私有信息与共享信息的管理。JOS 将大部分 per-CPU 信息放在 kern/cpu.h 中，其中比较重要的有：

- **CPU 内核栈**：支持多 CPU 同时陷入内核
- **TSS 及 TSS 描述符**：每个 CPU 拥有独立的 TSS 指明内核栈位置
- **当前运行 Env 指针**：用于记录每个 CPU 的当前运行进程
- **系统寄存器**：每个 CPU 需独立加载页表、GDT、LDT 等内容。

应当指出，每个 CPU 需要独立加载 GDT 到 GDTR(GDT Register)，但GDT 可以是共享的，JOS 采用了这种共享 GDT 的设计。在 BSP 和 APs 的启动阶段，处理器会加载各自的 Bootstrap

GDT；而在 kern/env.c 中定义了全局变量 gdt，并在 env_init_cpu() 装载到每个 CPU 的 GDTR 中。

Exercise 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.

这一步是完成所有内核栈的映射，并在每个内核栈区最后保留一个不映射的 Guide Page，最终完成代码段如下：

```
static void
mem_init_mp(void)
{
    uint32_t kstacktop_i = KSTACKTOP;
    for(int i = 0; i < NCPU; ++i)
    {
        kstacktop_i = KSTACKTOP - i * (KSTKSIZE+KSTKGAP);
        boot_map_region(kern_pgdir,
                        kstacktop_i-KSTKSIZE, KSTKSIZE,
                        PADDR(percpu_kstacks[i]),
                        PTE_W);
    }
}
```

Exercise 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.

在修改前，trap_init_percpu() 只能正确初始化 BSP，要求我们将其修改为可以适用于所有 CPU 的版本。根据提示，需要注意的是：(1) 全局变量 ts 不再可使用，应从 CpuInfo 数组中找到每个 CPU 对应的 TSS；(2)GDT 中各 CPU 信息所在条目偏移量不同。修改结果如下：

```
void
trap_init_percpu(void)
{
    // LAB 4: Your code here:
    thiscpu->cpu_ts.ts_esp0 = (uintptr_t)(percpu_kstacks[cpunum()] + KSTKSIZE);
    thiscpu->cpu_ts.ts_ss0 = GD_KD;
    thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);
```

```
    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;

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

    lidt(&idt_pd);
}
```

Locking

所有 APs 启动后将同时开始运行内核代码段，此时我们需要找到会发生竞态的代码段并解决竞态问题。JOS 使用的是名为 *big kernel lock* 的全局自旋锁，这个锁使得用户进程可以不受影响并行运行，但仅允许一个内核进程运行。

Exercise 5

In i386_init(), acquire the lock before the BSP wakes up the other CPUs.

In mp_main(), acquire the lock after initializing the AP and then call sched_yield().

In trap(), acquire the lock when trapped from user mode.

In env_run(), release the lock right before switching to user mode.

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

根据题目要求，我们对代码增加内核锁：

```
/**** i386_init() ****/
...
// Acquire the big kernel lock before waking up APs
// Your code here:
lock_kernel();
// Starting non-boot CPUs
boot_aps();
...

/**** mp_main() ****/
...
// Now that we have finished some basic setup, call sched_yield()
// to start running processes on this CPU.  But make sure that
// only one CPU can enter the scheduler at a time!
//
// Your code here:
   lock_kernel(); # Add in Lab4 Exercise 5
```

```
    sched_yield();
...

/**** trap() ****/
...
if ((tf->tf_cs & 3) == 3) {
    // Trapped from user mode.
    // Acquire the big kernel lock before doing any
    // serious kernel work.
    // LAB 4: Your code here.
    lock_kernel();
    assert(curenv);

      ...
}
...

/**** env_run() ****/
lcr3(PADDR(curenv->env_pgdir));

unlock_kernel(); # Add in Lab4 Exercise 5

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

注意到最后对 env_run() 的补全：因为 env_pop_tf() 不返回，因此锁的释放在调用该函数之前。

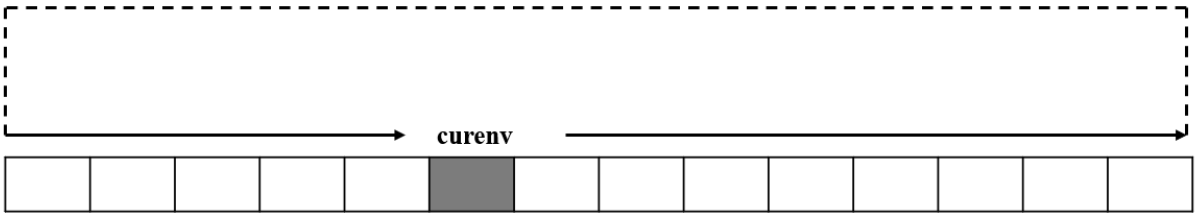
Question

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.

一个全局内核锁并不能取代多核多栈的功能。一个典型的例子是：每个 CPU 可以并行地陷入 trap，在这个过程中每个 CPU 都需要写内核栈。而 int 指令执行非原子且此时无上锁，仍会导致同时写入内核栈的错误。

1.2 Round-Robin Scheduling

JOS 在这里要求我们实现一个轮询 *(Round-Robin)* 的进程调度，这个功能的核心是 sched_yield() 函数。轮询调度检查当前运行的进程，从 envs 数组中排在其后面的进程中选取第一个状态为 RUNNABLE 的进程切换运行；若除当前进程外无可运行进程，则仍调度当前进程运行。轮询逻辑如下图所示：



Exercise 6

Implement round-robin scheduling in sched_yield() as described above.

根据本节内容，对 sched_yield() 代码进行补全：

```
void
sched_yield(void)
{
    struct Env *idle;

    // LAB 4: Your code here.
    bool findNext = false;
    if(!curenv)
    {
        // Situation 1: There is no running ENV
        idle = envs;
        for(int i = 0; i < NENV; ++i, ++idle)
            if(idle->env_status == ENV_RUNNABLE)
            {
                env_run(idle);
                findNext = true;
                break;
            }
    }
    else
    {
        // Situation 2: curenv is not NULL, do circular searching
        // Find the next ENV just behind the curenv
        int curenvIndex = ENVX(curenv->env_id);
        for(int i = 1; i < NENV; ++i)
        {
            idle = envs + (curenvIndex + i) % NENV;
            if(idle->env_status == ENV_RUNNABLE)
            {
                env_run(idle);
                findNext = true;
```

```
                break;
            }
        }
    }

    if(!findNext)
        // Situation 3: Only the curenv is runnable
        env_run(curenv);
    // sched_halt never returns
    sched_halt();
}
```

补全后我们可以对代码进行测试：根据提示，在 i386_init() 函数中增加下列语句段可建立若干进程，最后运行 make qemu CPUS=2 进行测试。

```
ENV_CREATE(user_yield, ENV_TYPE_USER);
```

Question

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?

因为参数 e 保存在内核栈中，而对于所用进程，内核段的内存映射是相同的。

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?

保存过程发生在 _alltrap 处，恢复过程发生在 env_pop_tf() 处。只有保证调度前后进程状态一直，才能确保进程运行不会出错。

1.3 System Calls for Environment Creation

完成上面部分的内容后，我们得到一个具备进程切换能力的内核，但其尚未提供用户进程接口，因此本节主要内容是构建用户态的进程创建接口。

*nix 系统提供系统调用 fork() 作为进程创建的用户接口，它复制调用它的进程的地址空间，创建新的进程。父进程与子进程的区别仅有进程 ID，且 fork() 在父进程中返回子进程 ID，在子进程中则返回 0。

JOS 实现了一组封装相对简单的系统调用，用于构建更高度封装的 fork()，主要包括：sys_exofork，sys_env_set_status，sys_page_alloc，sys_page_map 和 sys_page_unmap。

Exercise 7

Implement the system calls described above in kern/syscall.c and make sure syscall() calls them.whenever you call envid2env(), pass 1 in the checkperm parameter.

根据源码空缺处的提示，完善上述接口。

```
/* sys_exofork */
static envid_t
sys_exofork(void)
{
   // LAB 4: Your code here.
   struct Env *childEnv;
   int val = env_alloc(&childEnv, curenv->env_id);
   if(val < 0)
      return val;

   childEnv->env_status == ENV_NOT_RUNNABLE;
   childEnv->env_tf = curenv->env_tf;
   childEnv->env_tf.tf_regs.reg_eax = 0;
   return childEnv->env_id;
}

/* sys_env_set_status */
static int
sys_env_set_status(envid_t envid, int status)
{
   // LAB 4: Your code here.
   if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
      return -E_INVAL;

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

```
/* sys_page_alloc */
static int
sys_env_set_status(envid_t envid, int status)
{
   // LAB 4: Your code here.
   if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
      return -E_INVAL;

   struct Env *e;
   if(envid2env(envid, &e, 1) < 0)
      return -E_BAD_ENV;
   e->env_status = status;
   return 0;
}

/* sys_page_map */
static int
sys_env_set_status(envid_t envid, int status)
{
   // LAB 4: Your code here.
   if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
      return -E_INVAL;

   struct Env *e;
   if(envid2env(envid, &e, 1) < 0)
      return -E_BAD_ENV;
   e->env_status = status;
   return 0;
}

/* sys_page_unmap */
static int
sys_page_unmap(envid_t envid, void *va)
{
   // LAB 4: Your code here.
   struct Env *e;
   if(envid2env(envid, &e, 1) < 0)
      return -E_BAD_ENV;
   if(va >= (void *)UTOP || va != ROUNDDOWN(va, PGSIZE))
      return -E_INVAL;

   page_remove(e->env_pgdir, va);
   return 0;
}
```

Chapter 2

Copy-on-Write Fork

2.1 User-level page fault handling

Part B 部分的内容是实现 fork() 以提供用户的进程创建接口。在实现基础功能之外，JOS 也考虑这样一个问题：当创建新进程时，需要将父进程地址空间的内容复制到子进程中，产生较大的开销。然而实际上，在用户态下创建进程后，大部分情况下会直接调用 exec()，用新内容填充子进程的地址空间。

为避免产生这一不必要开销，JOS 采用了一种被称为写时复制 *(Copy-on-Write)* 的策略：创建进程后，父进程与子进程共享内存映射，直到其中一方修改内存内容再进行内存复制。具体实现时，子进程创建时仅从父进程拷贝内存映射，并将共享映射的页标记为只读。当其中一方对共享页进行写操作时，触发 Page Fault 进行页复制。

Setting the Page Fault Handler

Part B 第一部分内容是构建用户态的 Page Fault Handler。

JOS 为 Env 结构添加了 env_pgfault_upcall 成员，用于记录 Page Fault 的处理函数，并构建系统调用 set_env_pgfault_upcall 用于设置这一成员。

Exercise 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.

这一个练习要求我们实现上述设置处理函数的系统调用。根据实验提示，完善代码如下：

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env *e;
    if(envid2env(envid, &e, 1) < 0)
        return -E_BAD_ENV;
```

```
    e->env_pgfault_upcall = func;
    return 0;
}
```

Normal and Exception Stacks in User Environments

当在用户态发生 Page Fault 时，JOS 会从用户栈切换到用户异常栈 (User Exception Stack)，它位于 UXSTACKTOP 处，初始为一个页大小（见 memlayout.h）。

Invoking the User Page Fault Handler

本 Lab 将 Page Fault 发生时的进程状态称为 *trap-time state*，当 Page Fault Handler 存在时，JOS 内核将这些状态压入用户异常栈中，形成 *UTrapframe* 的形式；若 Hanlder 不存在，则直接销毁该用户进程。

如果用户进程已经处于异常状态后再次进入异常（例如 Page Fault Handler 自身发生异常），则在当前异常栈顶继续延伸，首先将一个空的 32 字节入栈（作为不同异常状态帧之间的分界），再将本次异常的状态以 UTrapframe 形式填充。

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

该练习要求我们实现 Page Fault Handler 对用户态处理的部分。根据前几节的内容可知，我们需要将栈切换到用户异常栈、将 trap-time state 入栈，并运行 upcall 处理程序。根据这一系列要求与代码提示，完成代码如下：

```
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.

    // LAB 3: Your code here.
    if(tf->tf_cs == GD_KT)
        panic("page_fault_handler: kernel page fault");
```

```
    // LAB 4: Your code here.
    struct UTrapframe* uTrapFrame;
    if(curenv->env_pgfault_upcall)
    {
        if((tf->tf_esp < UXSTACKTOP) && (tf->tf_esp >= UXSTACKTOP - PGSIZE))
            uTrapFrame = (struct UTrapframe*)(tf->tf_esp - sizeof(struct UTrapframe) - 4);
        else
            uTrapFrame = (struct UTrapframe*)(UXSTACKTOP - sizeof(struct UTrapframe));

        // Checks that environment 'env' is allowed to access the memory.
        // If not, this function will not return.
        user_mem_assert(curenv, uTrapFrame, sizeof(struct UTrapframe), PTE_W);

        uTrapFrame->utf_fault_va = fault_va;
        uTrapFrame->utf_err = tf->tf_err;
        uTrapFrame->utf_regs = tf->tf_regs;
        uTrapFrame->utf_eip = tf->tf_eip;
        uTrapFrame->utf_eflags = tf->tf_eflags;
        uTrapFrame->utf_esp = tf->tf_esp;

        tf->tf_eip = (uintptr_t)(curenv->env_pgfault_upcall);
        tf->tf_esp = (uintptr_t)uTrapFrame;

        // will never return
        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);
}
```

User-mode Page Fault Entrypoint

这一节内容要求我们实现用于调用 Page Fault Handler 的汇编代码，JOS 称之为 *upcall* 函数。upcall 函数在进程陷入 trap 并分发到 page_fault_handler() 调用，它负责调用真正的 Page Fault Handler （例如，fork 默认使用 lib/fork.c/pgfault()），并负责将控制交回给发生 Page Fault 的代码段并恢复环境，重新执行。

Exercise 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.

要求我们补全的代码段 _pgfault_upcall 中，已经完成了对 Handler 的调用并返回，因此需完善的内容是对运行环境的恢复。其中的难点在于：如何将控制交回。根据 Lab 的提示，我们仅需在用户态操作（不经过内核操作），且我们不能使用 jmp 指令：jmp 指令需要使用一个寄存器存储跳转地址，但返回时所有的寄存器都被恢复，不能再修改。因此，只能使用 ret 命令，但由于当前进程处于用户异常栈上，直接返回会导致出错，因此需要做一些准备操作。

异常栈上的信息以 UTrapframe 的形式存在，我们逐步进行处理。首先，栈顶两个寄存器与返回无关，因此直接跳过；取出 UTrapframe 中存放的 trap-time EIP、trap time ESP 信息，分别是陷入 trap 的代码句偏移地址（也即等待返回的地址）与用户栈的偏移地址，将返回地址放入用户栈中；恢复所有通用寄存器（此后不再修改），此时栈顶指针到了 trap-time EIP 的位置，我们跳过它并恢复 EFLAGS；最后，将用户栈地址装载到 ESP，再调用 ret 即可，此时用户栈栈顶存放的正是前一步放入的 trap-time EIP，因此可以直接返回陷入 trap 的位置。实际代码如下：

```
.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

    // LAB 4: Your code here.
    addl $8, %esp

    movl 0x20(%esp), %eax
    movl 0x28(%esp), %ebx
    subl $4, %ebx
    movl %ebx, 0x28(%esp)
    movl %eax, (%ebx)
    popal

    addl $4, %esp
    popf

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

Exercise 11

Finish set_pgfault_handler() in lib/pgfault.c.

补充用于设置 upcall 函数及 handler 函数的接口，同时初始化用户异常栈，为其分配内存。代码如下：

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

    if (_pgfault_handler == 0) {
        // First time through!
        // LAB 4: Your code here.
        envid_t envId = sys_getenvid();
        if(sys_page_alloc(envId, (void *)UXSTACKTOP, PTE_W | PTE_U | PTE_P) < 0)
            panic("sys_page_alloc fail: happened during set_pgfault_handler.");
        if(sys_env_set_pgfault_upcall(envId, _pgfault_upcall) < 0)
            panic("sys_env_set_pgfault_upcall fail: err in set_pgfault_handler.");

    }

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

2.2 Implementing Copy-on-Write Fork

这一节将完整实现用户态 fork() 接口，且使用 Copy-on-Write 策略（直接复指策略的用户进程创建接口为 dumbfork() ）。

Lab 为我们梳理了 fork() 调用的控制流：

1. 首次调用时，父进程将 *pgfault()* 装载为 *Page Fault Handler*，并设置自身 *upcall* 函数
2. 父进程调用 *sys_exofork()* 创建子进程
3. 调用 *duppage()* 将所有页映射复制到子进程中，再重写可写页/*COW* 页映射为 *COW*
4. 父进程用户异常栈映射不复制，直接为子进程的用户异常栈分配一页空间
5. 为子进程设置 *Page Fault* 的 *upcall* 函数入口
6. 设置子进程状态为 *RUNNABLE*

当进程不对 COW 页进行写操作时，不发生内存复制；当发生写操作时，将导致 Page Fault 发生。陷入 trap 后，控制流如下：

1. 内核将控制交给 upcall 函数，它调用 Page Fault Handler（默认为 pgfault()）

2. pgfault() 确认 Page Fault 是对页的写操作导致、且页面为 COW 页，否则 panic

3. pgfault() 分配新页并复制原先页的内容。

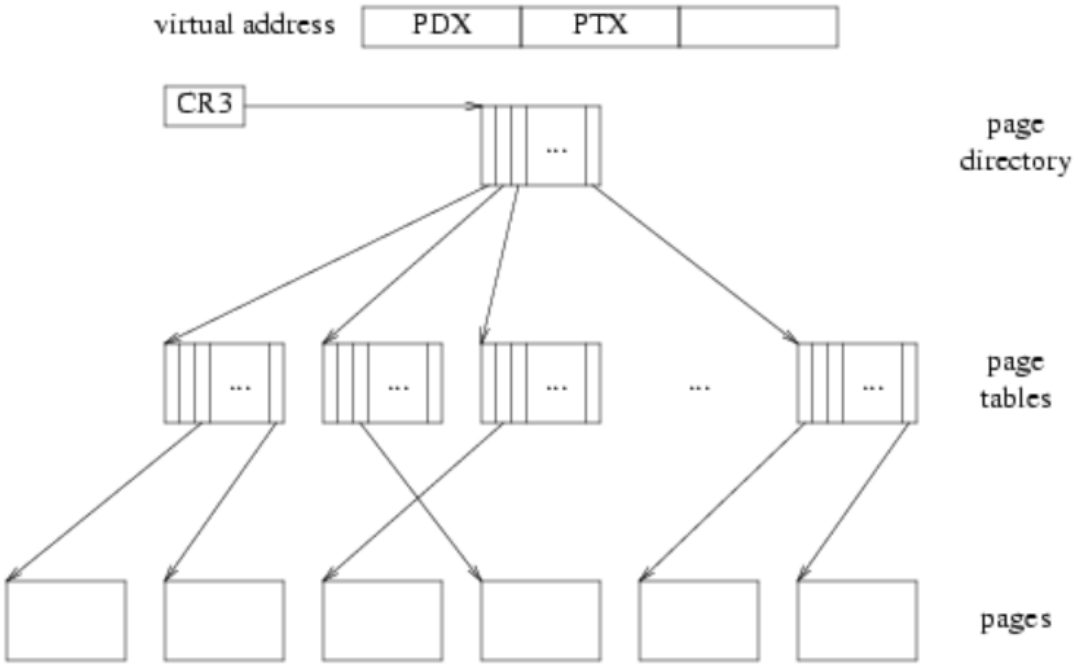
Exercise 12

Implement fork, duppage and pgfault in lib/fork.c. Test your code with the forktree program.

这个 Exercise 要求我们补全 fork 的控制流，使其支持 COW 功能，并完成 COW 的后备支持。因此，我们首先开始搭建 fork 的框架。

fork 的控制流 Lab 已经为我们梳理了，其中有个细节要展开解释：fork 如何操作页表？

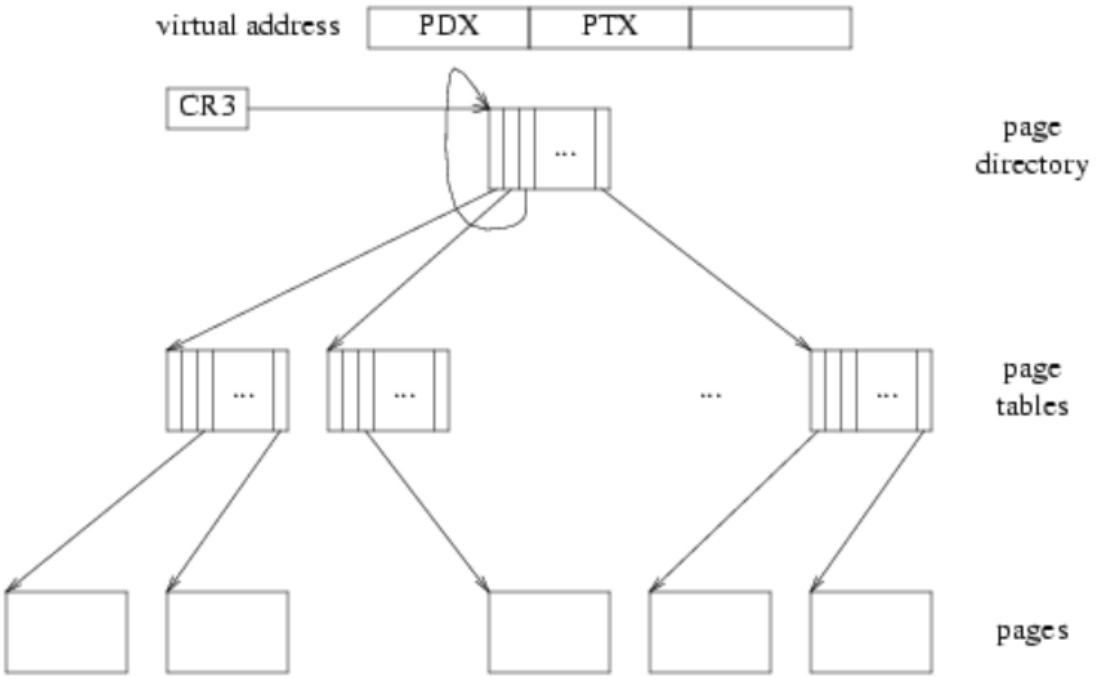
JOS 在自己的 VAS 中映射了一个 *UVPT* 段，来解决对页表的访问问题。我们知道，x86 使用二级页表的分页机制，对地址进行翻译时遵循这样的流程：*根据 PDX 在页目录中找到页表物理地址、根据 PTX 在页表中找到对应的物理页地址、加上偏移得到实际物理地址*，如下图所示：



x86 硬件实际按照上述流程执行，而并不区分页表和页目录。因此，JOS 使用了一个相当 trick 的方法：将页目录作为页表进行查询，从而得到页目录/页表自身。将 UVPT 段映射到页目录自身的物理地址后，x86 对 PDX 与 UVPT 相同的段进行地址翻译时，根据对应的 PDE 找到自身，在第二步翻译时实际上是根据 PTX 对页目录进行搜索，根据 PTX 偏移量得到的正是对应页表的 PTE。JOS 定义了一个虚拟地址为 UVPT 的数组 uvpt，作为页表访问入口。

如果需要访问页目录呢？在访问页表时首先需要确定页表是否存在，这是记录在 PDE 中的。

实际访问逻辑与上述访问页表逻辑是相同的，只需将地址的 PDX 与 PTX 都设置为与 UVPT 的 PDX 段相同即可。JOS 定义了数组 uvpd，虚拟地址为 (UVPT+(UVPT » PGSHIFT))。



在 Lab3 Exercise 段，我们已经设置了页目录 UVPT 段中存放的物理地址，因此可以直接使用。根据上述控制流梳理，补全 fork 函数：

```
envid_t
fork(void)
{
    // LAB 4: Your code here.
    // Step1: install the pgfault() as the C-level page fault handler.
    set_pgfault_handler(pgfault);
    // Step2: create a child environment.
    envid_t envId = sys_exofork();
    if(envId == 0)
    {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }
    // Step3: copy the address map of the parent to the child.
    uint32_t addr;
    for(addr = 0; addr < USTACKTOP; addr += PGSIZE)
        if((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P))
            duppage(envId, PGNUM(addr));
    // Step4: allocate user-exception stack
    if(sys_page_alloc(envId, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_P | PTE_W) < 0)
        panic("fork failed: error happened in sys_page_alloc.");
    // Step5: set up the pgfault handler upcall
```

```
    sys_env_set_pgfault_upcall(envId, _pgfault_upcall);
    // Step6: set the status of the child to be 'RUNNABLE'
    if(sys_env_set_status(envId, ENV_RUNNABLE) < 0)
        panic("fork failed: error happened in sys_env_set_status.");

    return envId;

}
```

接着我们补全 duppage() 函数，这个函数用于将父进程的地址映射复制到子进程中，并修改可写页为 COW。这里需要注意的是修改描述位信息为 COW 的顺序：**需先复制并设置子进程的页，再修改父进程页**。如果先设置了父进程页为 COW，若在设置子进程页前发生修改该页的事件（例如发生栈修改），由于父进程已设置为 COW，将触发 Page Fault，父进程指向新页，同时摘除了 COW 位。子进程复制后将指向这个页，而父进程修改该页时却不再引发 Page Fault，从而可能导致子进程出错。同理，即使父进程页已经是 COW，也应当在设置完子进程页后再设置一次。完善代码如下：

```
static int
duppage(envid_t envid, unsigned pn)
{
    // LAB 4: Your code here.
    // pn: page number
    void* addr = (void *)(pn * PGSIZE);
    pte_t pte = get_pte(addr);
    if(!(pte & PTE_P))
        return -1;
    if((pte & PTE_W) || (pte & PTE_COW))
    {
        if(sys_page_map(0, addr, envid, addr, PTE_P | PTE_COW | PTE_U) < 0)
            panic("duppage failed: sys_page_map error.");
        if(sys_page_map(0, addr, 0, addr, PTE_U | PTE_P | PTE_COW) < 0)
            panic("duppage failed: sys_page_map error.");
    }
    else
    {
        // The page is read-only
        if(sys_page_map(0, addr, envid, addr, PTE_P | PTE_U) < 0)
            panic("duppage failed: sys_page_map error.");
    }
    return 0;
}
```

剩下的便是补全作为 Page Fault Handler 的 pgfault() 函数，它负责复制内存内容并分配给需写的

进程。补全代码如下：

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

    // Check that the faulting access was (1) a write, and (2) to a
    // copy-on-write page.  If not, panic.
    // Hint:
    //   Use the read-only page table mappings at uvpt
    //   (see <inc/memlayout.h>).

    // LAB 4: Your code here.
    // if(!(err & FEC_WR) || !(uvpt[PGNUM(addr)] & PTE_COW))
    //   panic("pgfault error: the page has no copy-on-write or write flag.");
    if(!(err & FEC_WR) || !(get_pte(addr) & PTE_COW))
        panic("pgfault error: the page has no copy-on-write or write flag.");



    // 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
    // page to the old page's address.
    // Hint:
    //   You should make three system calls.
    // LAB 4: Your code here.
    envid_t envId = sys_getenvid();
    addr = ROUNDDOWN(addr, PGSIZE);
    if(sys_page_alloc(envId, PFTEMP, PTE_U | PTE_P | PTE_W) < 0)
        panic("pgfault error: sys_page_alloc failed.");
    memmove(PFTEMP, addr, PGSIZE);
    if(sys_page_map(envId, PFTEMP, envId, addr, PTE_U | PTE_P | PTE_W) < 0)
        panic("pgfault error: sys_page_map failed.");
    if(sys_page_unmap(envId, PFTEMP) < 0)
        panic("pgfault error: sys_page_unmap failed.");

}
```

Chapter 3

Preemptive Multitasking and IPC

3.1 Clock Interrupts and Preemption

3.2 Inter-Process communication (IPC)