Homework #4

周雨扬
2000013061

1 Challanges

本次作业完成了所有的代码补全任务，没有做 challange.

注：测试 `primes` 的时候因为电脑有点带不动，外加上虚拟机速度比本机更慢，他在 30 秒内大约只能处理大约 $100 \sim 200$ 个 `fork` 文件，因此会被评测程序直接杀死。

因此我在虚拟机上执行了 `make run-primes >run_primes.out`，将输出 `run_primes.out` 随源文件一并上传。

2 Exercise 1

mmio_map_region()

根据代码中给出的描述简单模拟即可。需要注意的是我们需要对映射的内存按整页取整，同时需要检查内存越界的问题。

3 Exercise 2

page_init()

注意到我们额外声明了一页内存 `MPENTRY_PADDR`, 用作每个处理器启动的引导区。因此直接将 `MPENTRY_PADDR` 对应的页从空页列表中一处即可。

Question 1

观察 `kern/mpentry.S` 上方注释，原因如下：

- 第一个被启动的 CPU 是冷启动，此时虚拟内存尚未完成设置，因此代码都是存储在低地址的引导块中。因而不需要对其进行映射操作。
- 之后所有启动的 CPU 实际上都是第一个 CPU 设置好环境后再启动。此时代码空间已经被与处理过，因此其初始运行位置高于 `KERNBASE`。但是其仍然需要在低地址空间下启动因为 `DS` 为 0。因此做了上述偏移。

4 Exercise 3

mem_init_mp()

根据函数描述简单操作即可。唯一需要注意的是在传入 `boot_map_region` 参数的时候，一定传入的是 `percpu_kstacks[i]` 对应的物理地址。

5 Exercise 4

trap_init_percpu()

首先我们需要用 `thiscpu->ts` 替换所有的 `ts`，因为我们只需要初始化当前 CPU 的中断机制即可。同理修改 `ts_esp0` 的时候也需要将栈顶修改为 Exercise 3 中，当前 CPU 对应的栈顶。

TSS 描述符的修改注释已经给出，之后 TSS 选择符照猫画虎，将描述符乘以 8 即可。其他不需要修改。

6 Exercise 5

i386_init(), mp_main(), trap(), env_run()

按照描述操作即可。注意我们需要在进入内核态时候上锁，推出内核态时候释放锁。

Question 2

考虑多个处理器同时出现了异常需要处理的情况。假定在处理器 1 处理异常的时候处理到一半，正好进行了上下文切换（例如最后我们实现的多处理器之间通信)，切换到了发生异常的处理器 2. 则在混用内核栈的情况下异常处理栈中信息会乱序排列，导致错误。

7 Exercise 6

sched_yield()

观察上下文发现 `curenv` 被定义成了 `thiscpu->cur_env`。这可以剩下一点代码长度。

我们可以通过 `curenv` 获取当前环境的编号。之后我们只需要对于所有的 `NENV` 个可能的环境从当前编号向后依次轮询，判定是否可以执行即可。如果可以执行则利用 `env_run` 执行当前环境。

最后切换回当前进程即可。注意只有当前进程状态为 `ENV_RUNNING` 才需要重启，否则扔弃继续等待即可。

syscall()

简单将 `sys_yield()` 加入可调用系统函数即可。

i386_main()

原先已经提供了一个能够启动 `user/prime` 的代码，简单修改，将其拷贝三遍即可。

Question 3

原因：因为所有的 `struct Env*` 都是指向内存初始化的时候声明的 `envs` 数组。实际上这个数组的寻址方式是固定的，且其为内核态数组其也不会被 COW 机制进行不需要的拷贝，因此不管是哪一个环境，访问 `env` 的数组总是固定的。

Question 4

原因：因为我们无法保证新的用户程序会不会修改相关寄存器的值。如果不保存的话相关状态无法还原，我们代码就无法正确的恢复原来的状态。

位置：`kern/env.c`, 函数 `env_pop_tf` 会备份当前环境的状态。

8 Exercise 7

sys_exofork()

按照函数给定的注释实现即可。注意由于操作系统中函数的返回值总是被存储在 `%eax` 中，因此我们需要在最后将 `env_alloc` 新建的环境的 `%eax` 赋值为 0, 满足子环境返回 0 的要求。

sys_env_set_status()

按照函数给定的注释实现即可, 没啥细节。

sys_page_alloc()

按照函数给定的注释实现即可。唯一需要注意的是，如果在 `page_insert` 插入的时候发生错误，我们需要将先前从空页表中提出来的空页再存回去。否则会导致内存泄漏。

sys_page_map()

按照函数给定的注释实现即可。注意需要检查所有的约束条件。

sys_page_unmap()

按照函数给定的注释实现即可。注意即使这一个页没有被分配，释放也可以被认为是成功的。

syscall()

将上述五个函数加入对应的函数调用列表即可。

9 Exercise 8

sys_env_set_pgfault_upcall()

按照函数给定的注释实现即可。记得再实现结束后加入 `syscall()` 中函数调用列表。

10 Exercise 9

page_fault_handler()

首先，如果没有设置用户态下页错误的处理函数，简单使用他们提供的代码即可。

否则我们需要将信息存储在当前处理器对应的异常栈上，首先确认压栈的位置。如果发生错误时侯的 %esp 指向的是用户栈，则我们可以简单的压倒异常栈栈顶；否则我们可以根据 %esp 得到异常栈栈顶位置。随后我们需要检查该片内存区域的权限以防无法访问的问题。

之后我们需要将错误信息逐次压入栈中。观察错误信息结构 struct UTrapframe 可以获知所有需要压入的信息，逐个存储即可。

最后我们需要设置调用处理函数的方式，以及处理函数结束后的返回地址。分别将处理函数指针压入捕获信息的 %eip 寄存器，struct UTrapframe的下标压入捕获信息的 %esp 寄存器即可。此时只需要调用 env_run 便可进入处理函数了。

11 Exercise 10

pfentry.S

我们需要实现的是恢复原来的相关寄存器状态，并且恢复执行代码。

第一步我们需要将 %eip 压入发生错误之前的栈。这样子最后我们就可以利用 ret 语句，返回到错误发生的位置。观察栈结构，发现发生错误时 %esp 相对位移为 0x30, %eip 相对位移为 0x28. 据此简单实现即可。

第二步我们需要还原寄存器。寄存器相对栈顶的位移是 8, 因此将栈顶减去 8 后 popal 还原寄存器状态即可。

第三步我们需要还原标志位。标志位相对此时栈顶寄存器的位移是 4, 因此将栈顶减去 4 后 popfl 还原状态位即可。

最后只剩下我们修改后的发生错误时的栈顶坐标，将其移动到 %esp 后返回即可。

12 Exercise 11

set_pgfault_handler

第一次声明页错误的时候，我们需要先声明异常栈。之后在设置页异常调用函数位我们在上一问中写的 _pgfault_upcall

其余细节可以看注释实现。

13 Exercise 12

pgfault()

这里我们需要处理 COW 的问题。

在求出页地址后，如果该问题不是写时候的问题 (检查 err & FEC_WR)，也不是 COW 的问题 (一级页表权限没有 PTE_P, 或者是二级页表权限没有 PTE_P 或 PTE_COW)，则确实发生页错误。

否则我们可以通过如下操作复制当前页：

- 首先声明一个页，从临时地址映射过来，权限为 PTE_P,U,W(该错误由用户态写引起)。将旧的页数据拷贝到新的页数据。

- 插入新建页到旧地址，权限位同上。
- 删除临时地址到新建位映射，防止用户误操作。

duppage()

这里我们需要处理 `fork` 时页复制的问题。

如果当前页是可写的 (页中 `PTE_W,PTE_COW`) 至少有一个为真，则此时为写拷贝。我们需要同时将当前环境，和子环境下的内存页权限位同时设置为 `P,U,COW`，利用两次 `sys_page_map` 完成。

否则此时为读拷贝，不会产生 COW，因此只需要将子环境的页权限位设置为 `P,U` 只读即可。

fork()

首先我们得把页错误设置为上面的 `pgfault`, 否则遇到 COW 就会直接报错，而不是继续执行。

之后我们调用 `sys_exofork` 完成子环境建立。

如果子环境返回，根据上面的设置其必须等到父环境将设置为可执行后在执行，因此只需要设置 `thisenv` 即可。

如果父环境返回，则我们需要将当前环境的相关页映射拷贝到子环境。但是异常栈除外! 其余的我们只需要判定本机手否有访问权限即可。如果有就 `duppage()`. 判定是否有权限类似于 `pgfault()`, 略。

之后我们需要给子环境设置页错误返回，可以抄上一个问题的实现。最后只需要将子环境设置为可以执行即可。

14 Exercise 13

trapentry.S, trap_init()

实际上你可以把它当成编号为 `IRQ_OFFSET` 到 `IRQ_OFFSET + 15` 的特殊的不需要记录错误码的异常。直接按照上次 Lab 的写法抄 16 次即可。

15 Exercise 14

trap_dispatch()

按照注释完成即可，没有细节。

最大的坑点在于，你在实现系统调用 `sys_destroy` 的时候，传入参数必须是 a1 而非当前环境的编号，因为此时可能会出现父线程杀死子线程的情况。

16 Exercise 15

sys_ipc_recv()

首先我们检查如果目标地址需要且没有对齐，则直接退出。否则我们可以直接设置当前的 `ipc` 状态为接收中，设置目标页地址，将当前环境设置为不可执行后，就可以挂起等发送端了。

sys_ipc_try_send()

所有的异常情况注释已经完全讨论了所以我们直接按照注释抄就行了。

如果我们需要发送整页的信息，我们需要在发送成功后，额外将目标环境的 `ipc` 结构下权限信息进行相关设置。最后如果全部成功，我们就可以将目标环境接收状态撤出，置接受数据的来源为本机，接受的整数为参数给定值，寄存器 `%eax` 作为返回值设置为 0。之后目标环境就可以重新执行了。

唯一需要注意的是检查目标环境时候 `perm` 要为 0，因为完全可以子线程向父线程发送信息。毋须满足上述约束。

上述两个函数需要加入系统调用中。

ipc_recv()

接收端等待操作会在 `sys_ipc_recv()` 中完成，因此直接根据函数的注释进行相关参数的读取即可。

ipc_send()

发送端发送的时候有可能接收端还没有进入接收状态。因此需要在每一次不成功的 `sys_ipc_try_send()` 判断失败原因。如果失败是因为接收端没有进入接收态，则需要进行轮转等待后再次尝试；否则才可以返回错误。