Lab4 Solution

516030910006 方俊杰

Exercise 1.

实现 mmio_map_region，查看其使用在 kern/lapic.c 的 lapic_init 的开始部分，作用类似 boot_map_region，将输入的 [pa, pa+size] 映射到 虚拟地址的 [base，base+size]
需要检查其输入不能超过 MMIO hole，用 MMIOLIM 限制。根据提示使用 boot_map_region 映射。

Exercise 2.

阅读 boot_aps 和 mp_main 在 kern/init.c 以及 kern/mpentry.S 的汇编代码，理解 BSP 启动 APs 的控制流。修改 在kern/pmap.c 的 page_init 函数，使得 MPENTRY_PADDR 不被认为是 free。查看 memlayout 得知 MPENTRY_PADD 是 0x7000，可见在 page 1，将其从第一个循环移除。

Question 1.

比较 kern/mpentry.S 和 boot/boot.S，可以知道 MPBOOTPHYS 的作用是 计算symbol 的绝对地址，而不是依赖 linker 来填充。在 mpentry.S 执行时 BSP 处于保护模式，但是 AP 还没有开启页表仍处于实模式，只能使用物理地址，需要该宏转换

Exercise 3.

修改 mem_init_mp() (在 kern/pmap.c) 映射每个 cpu 的 stack 起始于 KSTACKTOP，就是要使得 虚拟内存地址 kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP) 映射到对应数组即 percpu_kstacks[i] 上，注意有一个用于 guard 的 GAP。

Exercise 4.

修改全局 TSS 以适应多处理器，不能再使用全局的 ts 变量，每个 CPU 都需要有 TSS，即 thiscpu->cpu_ts。使用 gdt[(GD_TSS0 >> 3) + i] 作为 TSS descriptor。根据给出的代码依次初始化每个 CPU 的 TSS。运行 make qemu CPUS=4 可以看到 SMP: CPU 1/2/3 starting

Exercise 5.

应用 big kernel lock 使得每个处理器同时只有一个 env 可以进入 kernel mode。有几处地方需要修改，在 init.c 的 i386_init 中 BSP 唤醒其他 AP 前拿锁；在 mp_main 初始化 AP 后拿锁并调用 sched_yield；在 trap.c 的 trap 函数中判断为从 user mode 陷入时拿锁；在 env.c 的 env_run 函数的 env_pop_tf 之前释放锁。

Question 2.

看上去使用了 kernel lock 保证了同时只有一个 CPU 会进入 kernel code，那为什么还要给每个 CPU 独立的 kernl stack 呢？
在不同的内核栈上保存了不同的信息，如果某CPU从内核返回时留下了将来要用的信息,就需要单独

的栈。中断到来会使用kernel stack，如果一个 CPU 在内核态时，另一个CPU 有一个中断到来可能会改变原来 CPU 在内核栈的数据。

Exercise 6.

遍历ENVS 中的所有 ENV，遇到第一个 RUNNABLE 就运行。根据 hint 完成。

Question 3.4.

在实现 env_run 时调用了 lcr3，但在这前后都解引用了 e ，但是页表换了，为什么都能正确解引用？
e 是处于内核地址的，不同的环境的页表映射中对于内核地址的映射都是相同的，虽然页表换了，但是 e 的映射仍旧是对的。

无论何时 kernel 转换 env 都要保存 前一个 env 的寄存器，为什么？何时发生？
这些信息是前一个 env 下次再被调度到时能够正确运行的保证，因此需要保存。
在 trap 进入内核前保存信息，即 kern/trapentry.S 中；在 env.c 的 env_run 调用 env_pop_tf 时恢复。

Exercise 7.

实现声明在 kern/syscall.c 中的系统调用，需要使用很多在 kern/pmap.c 和 kern/env.c 中的函数特别是 envid2env。
注意根据 hint 检查参数，非法的返回 相应 error code。
这几个大部分都是 pmap.c 或者 env.c 中函数的封装，只是要注意 user mode 的传入参数检查。
测试时发现创建的 yield env 一定要有 3 个或 以上否则会卡住。

Exercise 8.

实现 sys_env_set_pgfault_upcall 系统调用，用 envid 得到 env，然后把 env_pgfault_upcall 域设为 func。

Exercise 9.

根据提示用 tf 初始化建立 UTrapFrame, 其对应地址在 UXSTACKTOP 下，但是如果发生 PF 递归，会是依次增长的地址，要根据 tf 的 esp 计算是否为第一个得到，如果不是第一个需要有一个 额外的 empty word。然后改变 curenv 的 eip 和 esp 换栈换运行程序，调用 env_run 运行。

Exercise 10.

实现 lib/pfentry.S 中的 _pgfault_upcall 直接返回到引起 PF 的代码，而不用经过 kernel。不能调用'jmp'，也不能从异常堆栈中调用'ret'。将 trap time %eip push 到 trap-time stack，切换到该堆栈并调用'ret'，将%eip恢复到故障前的值。

Exercise 11.

实现 lib/pgfault.c 的 set_pgfault_handler，使得用户可以注册自己的 PF handler。使用 sys_page_alloc 分配 exception stack 在 UXSTACK - PGSIZE 往上一个 page，使用 sys_env_set_pgfault_upcall 注册 _pgfault_upcall。运行 make run-faultXXX 可以看到处理了 user PF。

Exercise 12.

实现 fork 、dumbfork 和 pgfault 在 lib/fork.c。

1. pgfault：
   真正的 PF 用户处理函数
   首先检查是否为写 ，使用 err & FEC_WR
   再检查是否 COW ，使用 memlayout.h 的只读映射 uvpt，用 PGNUM 作索引得到 PTE ，检查 PTE_COW
   hint 提示需要使用 3 次系统调用
   第一次，sys_page_alloc （注意 0 表示当前env），分配页到临时位置 PFTEMP
   然后 memmove()，把 addr 即 引起 PF 的虚拟地址内容拷贝到 PFTEMP
   第二次，sys_page_map，把临时位置映射到 发生 PF 的位置，并修改权限为 用户可读
   第三次，sys_page_unmap，解除 temp 的映射
2. duppage：
   映射 parent 的页到 child
   如果 用 uvpt[pn] 获取到的 pte 是 可写（W） 或者 COW 的，都需要把映射的页也置为 COW
   如果是 COW 表出现嵌套 fork，是吗
   然后把 parent 的 addr （= pn *PGSIZE）映射到 child 的相同 addr
   如果是 COW，还要反映射自己
3. fork：
   为自己设置 pgfault handler
   用 sys_exofork 创建子 env
   对于 child process，要修改 thisenv
   对于 parent process，要用 duppage 映射 UTOP 以下的地址空间给 child
   还要创建 exception stack 给 child
   要注册 pgfault upcall，注意 extern 得到该函数
   完成后设置状态 RUNNABLE

Exercise 13.

修改 kern/trapentry.S 和 kern/trap.c 初始化 IRQs 0-15 的 IDT表，并提供 handler。然后修改 env_alloc （在 kern/env.c）使得进入 user mode 时开启中断。同样要 在 sched_halt 中取消对 sti 的注释，使得不屏蔽中断。
查看 inc/trap.h 发现只给出了 6 个中断。类似上前面的lab处理 exception 处理的时候，对 16 个 interrupt 处理。在 trapentry.S 以汇编的形式用 TRAPHANDLER_NOEC 声明 handler，在 trap.c 用 extern 导入，并用 SETGATE 写入 idt。在 env.c 的 env_alloc 修改 e->env_tf.tf_eflags |= FL_IF 开启中断。在 sched_halt 也取消对 sti 的注释。

Exercise 14.

现在 make run-spin 不会像之前那样一直 spin 了，但是会出现 Hardware Interrupt
原因是没有真正的 handler 来处理中断。
修改 trap_dispatch 使得其在时钟中断发生时寻找一个不同的环境来运行。根据 注释，发现是 IRQ_OFFSET + IRQ_TIMER 时, 先调用 lapic_eoi 确认中断，再调用 sched_yield 调度。现在也不会出现 Hardware Interrupt了，调度到父进程 kill 了 spin child。

Exercise 15.

实现 上述两个 syscall （sys_ipc_recv 和 sys_ipc_try_send） 在 kern/syscall.c，以及两个 wrapper（ipc_recv 和 ipc_send）。
先查看 struct Env 增加了需要使用到的域。根据注释实现两个 syscall，有大量的参数检查，使得两个 env 可以传递信息。然后实现两个封装函数，注意到 sender 的封装要循环直到发出信息成功，receiver 需要检查域以更新可能的参数。

Challenge

实现 fixed-priority scheduler 固定优先级调度。
首先需要在 struct Env 增加一个域 env_pr 表示环境的执行优先级，越小优先级越高。
然后需要一个新的 syscall （set_env_pr）使得 user mode 可以改变当前环境的优先级，这需要修改较多文件,包括：

- inc/syscall.h，在 enum 增加新的 syscall 标识符
- inc/lib.h，声明新的 user 可见系统调用
- kern/syscall.c， 实现新的系统调用 set_env_pr 改变当前环境的优先级。并在 syscall 函数中增加新的 dispatch 分支
- lib/syscall.c，声明该 syscall 的 stub

然后需要修改调度函数，使得每次调度遍历所有环境，找到 RUNNABLE 中优先级最小的一个并与当前环境比较后运行优先级高者。
另外修改了 fork() 使得所有环境在创建时的优先级都是最高的 0。
还另写一个测试程序验证调度，在该测试程序中，父进程会依次 fork 出 3 个子进程。每个子进程被 fork 出来后即调用 set_env_pr 改变优先级为对应的 fork 次数，就是父进程循环次数 i。然后子进程进入一个循环会每次打印一段信息后调用 sys_yield，共循环 9 次，在循环到第 3 次时，子进程会修改自己的优先级为 （4-i）。
以下是运行截图，效果良好。
程序分为几个阶段：

1. 父进程创建了 3 个子进程，子进程各以优先级 0 运行一次，然后改变优先级到 i。
2. child 1 第二次改变优先级之前，进入子进程循环，此时 child 1 优先级最高，即使重新调度也会是它被运行。3 次循环后 child 1 优先级变为 3。此时 child 1 2 3 的优先级分别为 3 2 3，所以接下来 child 2 被调度，child 2 改变优先级没有变化，child 2 执行 9 次循环结束
3. 此时 child 1 3 的优先级都是 3，则会以 RR 的方式轮询，child 1 3 各自运行。然后 child 3 运行 3 次后改变优先级为 1，则 child 3 优先级最高，一直被调度直到退出

4. 只剩下优先级为 3 的 child 1 执行直到退出，则所有环境退出

```
i am environment 00001000
[00001000] new env 00001001
[00001000] new env 00001002
child 1 is running with priority 0
child 1 change priority from 0 to 1
child 1 yield with priority 1, iteration 1.
[00001000] new env 00001003
[00001000] exiting gracefully
[00001000] free env 00001000
child 3 is running with priority 0
child 3 change priority from 0 to 3
child 3 yield with priority 3, iteration 1.
child 2 is running with priority 0
child 2 change priority from 0 to 2
child 2 yield with priority 2, iteration 1.
child 1 yield with priority 1, iteration 2.
child 1 yield with priority 1, iteration 3.
child 1 change priority from 1 to 3
child 2 yield with priority 2, iteration 2.
child 2 yield with priority 2, iteration 3.
child 2 change priority from 2 to 2
child 2 yield with priority 2, iteration 4.
child 2 yield with priority 2, iteration 5.
child 2 yield with priority 2, iteration 6.
child 2 yield with priority 2, iteration 7.
child 2 yield with priority 2, iteration 8.
child 2 yield with priority 2, iteration 9.
[00001002] exiting gracefully
[00001002] free env 00001002
child 1 yield with priority 3, iteration 4.
child 3 yield with priority 3, iteration 2.
child 1 yield with priority 3, iteration 5.
child 3 yield with priority 3, iteration 3.
child 3 change priority from 3 to 1
child 3 yield with priority 1, iteration 4.
child 3 yield with priority 1, iteration 5.
child 3 yield with priority 1, iteration 6.
child 3 yield with priority 1, iteration 7.
child 3 yield with priority 1, iteration 8.
child 3 yield with priority 1, iteration 9.
[00001003] exiting gracefully
[00001003] free env 00001003
child 1 yield with priority 3, iteration 6.
child 1 yield with priority 3, iteration 7.
child 1 yield with priority 3, iteration 8.
child 1 yield with priority 3, iteration 9.
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
```