OSLab4 文档

5140379064 陈俊

一、多 CPU 相关：

这部分内容主要是实现多 CPU 并行。

首先，修改 page_init，把 MPENTRY_PADDR 开始的一个 page 设为非空闲页，不放入 free_list 中。

之后，修改 mem_init_mp，对每个 CPU 在内核中的栈进行 boot_map_region 的映射。由于之前 lab 中设置的大页与 IOMEM 有点冲突，我关闭了大页，并且修改了在 mem_init 中的一些映射。

在 trap_init_percpu 中，我仿照了函数之前的写法，为每个 cpu 初始化了 tss 及其描述符。由于使用了 sysenter，因此对于每个 cpu，也需要进行不同的 wrmsr 的设置，以保证不同 cpu 使用不同的内核栈。

之后，我们为 kernel 态添加了内核锁，按照文档上的提示，在进入 kernel 态后拿锁；进入用户态前放锁，以保证不同 CPU 只有一个 CPU 能处于内核态。在 4.1 中，要求我们实现一个 ticket 锁，与老师上课介绍的相同，lock 有两个参数 own 和 next。只有当 own 和 next 不同时，说明有 CPU 持有这把锁。当有 CPU 要锁时，如果这个 CPU 不持有锁，使用 atomic_return_and_add，返回之前的 next，next++，只有当 lock 中的 own 与返回的 ticket 相同时，CPU 才能获得锁。放锁后，own 的值也会++（原子指令），以此来实现锁的分配。

多 CPU 还需实现不同 CPU 之间的调度。按照文档，我需要实现一个 Round-Robin 的调度。从当前运行的 env 开始，遍历所有的 env，遇到第一个可运行的 env 后，调度这个 env。如果没有找到，且之前运行的 env 依然为 RUNNING，则运行之前的 env。同时，修改 syscall 中的 sys_yield，让其调用 sched_yield，这样就实现了 CPU 不同 env 间的调度。

二、Fork 相关：

这部分先实现了一个简单的 exofork。实现了 sys_exofork，sys_env_set_status，sys_page_alloc，sys_page_map，sys_page_unmap 这一系列函数。这部分函数主要就是一些功能的实现和封装，大部分代码均为错误情况的检测。Fork 的实现只需调用 env_alloc 函数 alloc 一个 env，将其 status 置为 not_runnable，复制父进程的 tf，并将其返回值置为 0。这里我遇到的一个问题是我使用的是 sysenter，但是我发现 syscall 会进入 trap 中，这让我之前有点疑惑。后来我发现，在 lib/syscall.c 中，关于 exofork 的实现是 inline 的，是一段使用了 int 语句的汇编代码。因此，我按照 trap 部分实现了 syscall，同时支持 int 和 sysenter，，这样就通过了测试。

三、COW fork 相关：

这部分实现了使用了 COW 技术的 fork。为了实现这一功能，最重要的就是处理 page fault。当我们访问内存触发 page fault，并发现这是一个 COW 页时，我们就可以处理这个 page fault，重新分配一个新的页。

最开始，实现 sys_env_set_pgfault_upcall，用来设置 env 的 pgfault_upcall，只需把函数指针存入 env 中的变量即可。

之后，按照文档，我们知道，当发生 pgfault 后，我们要将 Utrapframe 压入 UXSTACKTOP，如果在这期间发生递归，则空 32bit 位置，继续往下压。我们只需根据 tf 中 tf_esp 的位置，来判断是否位于 UXSTACK。每次发生 pgfault，填充好 utf 放到 UXSTACK 上，修改 tf 的 eip 为 upcall，esp 为 utf 的指针，并调用 env_run，就可以正确

处理 pgfault。

之后，就是修改 lib/pfentry.S 中的汇编代码。在_pgfault_upcall 函数中，我们可以看到，它先调用了_pgfault_handler，处理了 pgfault，然后就需要回到原来的 eip，并且拥有原来的通用寄存器的值，以及原先的 eflag。首先，先把 tf 中的 eip 拿出来，放在 eax 寄存器中。之后，把 tf 的 esp 拿出来，放在 ecx 中。我们将 ecx 的值减 4，把 eax 存入这个地址，再把 ecx 写回到 tf 的 esp 中。之后，popal，恢复所有的通用寄存器，跳过 eip，popfl，恢复 eflag，再 pop esp，恢复初始地址。由于之前的改动，eip 就处于 esp 的位置。Ret 指令后，程序就恢复了原先的一切。

按照文档，我们还需实现 lib/pgfault.c 中的 set_pgfault_handler 函数。如果 _pgfault_handler 为 null，则说明这是第一次初始化。因此需要在这里，进行 UXSTACKTOP 的 page_alloc，以及对 env 的 pgfault_upcall 的赋予。之后，把 pgfault_handler 改为提供的 handler。

实现了上面这些之后，我们就可以实现基于 COW 的 fork。在 fork.c 中，我们需要首先 set pgfault 函数。当 page fault 发生后，我们需要判断该 fault 是不是一个 write fault（通过 FEC_WR），以及该页是不是一个 COW 页。若是，先 alloc 一个页，与 TFTEMP 这一虚拟地址绑定，将发生 fault 的 va 的内容 memmove 到 TFTEMP 这一地址上，之后，通过 map，将发生 fault 的 va 这一虚拟地址映射到 TFTEMP 对应的页上，这样，va 对应就变成了新页，同时拥有 PTE_W 权限。之后，unmap TFTEMP，这样就实现了 pgfault 的处理。当 fork 时，需要 duppage。在 COW 中，将所有的 UTOP 以下的页（不包括 UXSTACKTOP）都进行 duppage。使用 sys_page_map，映射同一个物理地址，将页表中的 perm 改为 PTE_COW。同时，父进程也需要将那个对应的页表改为 PTE_COW。在对新的 env 进行完页的分配以及 upcall 的设置后，将其状态设为 RUNNABLE，这样整个 COW fork 就实现了。

四、Preemptive Multitasking 和 IPC 相关

类似于之前 lab 的 trap，对 0-15 IRQ 进行相应的处理，增加 entry handler 以及 SETGATE，使其能正确进入 trap。在对于时钟中断的处理上，我本来直接在 trap_dispatch 中关于时钟的中断处理中调用了 sched_yield，但并没有达到预期效果。后来，知道要调用 lapic_eoi 函数，表示已经接收到中断，才能正确运行。

在 IPC 中，实现了 sys_ipc_recv 和 sys_ipc_try_send 函数。在 sys_ipc_try_send 函数中，如果 envid 指向的 env 没有 block 在 recv，或者 env_ipc_from 不为 0（表示有其他 env 在发送 ipc），则返回-E_IPC_NOT_RECV。如果 srcva 小于 UTOP，就以权限为 perm 映射到目标 env 的 dstva 上。之后，也要修改目标 env 的一系列值，包括 recving、value、env_ipc_from 以及 env 的状态等，同时修改 env 的 tf，将 eax（返回值）设为 0。

在 sys_ipc_recv 中，就是简单的设置自己的 recving 为 1，以及自己的 dstva，将 env_ipc_from 设为 0，将自己状态设为 NOT_RUNNABLE，之后调用 sched_yield，等待其他 env 发送信息将其唤醒。

lib/ipc.c 是对 syscall 的一层包装。sys_ipc_recv 从 env 中获得返回的值，发送的 env 以及相应的 perm。sys_ipc_send 则是利用无限循环不断地尝试发送，如果返回值为-E_IPC_NOT_RECV，则继续尝试，直到出现其他错误或者发送成功。

五、Challenge

实现了第二个 challenge，基于优先级的调度。具体内容详见 answe-lab4.txt，测试程序为 user/prior.c。（我将 challenge 部分代码注释了，使用 round-robin。如果想测试，可以将 sched.c 中 round-robin 部分注释，删去 priority 部分的注释，即可测试）