OS Lab4 设计文档

516030910293 姚子航

一、多处理器与多任务协作

第一部分我们要将 JOS 系统扩展为可支持多个 CPU 的架构，我们采用的架构为 SMP 架构，即“对称多处理器”，每个 CPU 在功能上是等同的，平等的访问内存与外设，与之相对应的架构还有 NUMA 架构，由于多处理器技术是紧耦合，需要协调处理器之间的协同工作，解决资源争夺，它的可扩展性不如 NUMA 这种处理器各自有本地内存的架构。

虽然处理器地位是平等的，但是在启动时还是需要有一个处理器来唤醒其他所有的处理器，负责启动的处理器称为 Bootstrap processor，其他的处理器称为 Application processor，各个处理器都拥有一个本地的中断控制器(LAPIC)，各个 CPU 就通过发送处理器间中断(IPI), 处理器访问自己的 LAPIC 设备是通过 MMIO 而不是 PIO 的方式，在第一个 exercise 我们就要实现 MMIO 的映射操作，将一块物理地址的内容映射到 MMIOBASE 的区域。函数的实现较为简单，类似于 boot_alloc 中的实现，需要注意的有一个点，就是这块映射的 PTE 条目需要置上 PCD 与 PWT 位，由于 MMIO 操作的不是内存而是外设，我们不可以使用缓存来缓存这些修改，而是要将其直接发送到设备上。代码如下：

```
if((base + ROUNDUP(size, PGSIZE)) > MMIOLIM){
    panic("Reservation would overflow MMIOLIM!");
}
boot_map_region(kern_pgdir, base, ROUNDUP(size, PGSIZE), pa, PTE_PCD|PTE_PWT|
    PTE_W);
void* old_base = (void*)base;
base += ROUNDUP(size, PGSIZE);
return old_base;
```

之后就是在 SMP 架构下的系统启动，在 mp_init 函数中，从 BIOS 区域中读取了多处理器的配置信息，在 init 中的 Boot_aps 函数中，BSP 将 AP 的启动代码，即内核中的 mpentry.S 拷贝到 0x7000 地址处，之后，BSP 通过 STARTUP 进程间中断唤醒各个 CPU，每个 CPU 在被唤醒并执行完启动代码后会在自己的 CPUInfo 结构中置位来通知 BSP。Exercise2 就是要求我们不能将 0x7000 地址处的页放置到 freelist 中，类似于 IO hole 来实现这个要求即可。代码如下：

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

由于 CPU 可能同时进入内核态，所以为了防止冲突，我们需要各自拥有私有的内核栈，Exercise3 的实现很简单，就是要分配出各个 CPU 的内核栈，这个操作仿照之前映射一个内核栈的操作来做就可以，需要注意在内存布局中，为了防止一个 CPU 的内核栈由于某些错误访问溢出而影响到其他 CPU 的内核栈，在每个内核栈之间都留下了一些未映射的 Gap 防止越界。由于各个 CPU 都有自己的内核栈，所以在 trap 的时候各个 CPU 都有自己的 TSS 描述符,所以之前我们在 trap_init_percpu 中使用的 ts 全局变量现在已经不可以使用，我们要将

所有使用 ts 的地方，换成使用 thiscpu->cpu_ts 和 CPUID 全局变量来分别初始化各个 cpu 的 TSS，代码如下：

```
int cpuid = thiscpu->cpu_id;
thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpuid * (KSTKSIZE + KSTKGAP);
thiscpu->cpu_ts.ts_ss0 = GD_KD;
thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);

// Initialize the TSS slot of the gdt.
gdt[(GD_TSS0 >> 3) + cpuid] = SEG16(STS_T32A, (uint32_t) (&thiscpu->cpu_ts),
                    sizeof(struct Taskstate) - 1, 0);
gdt[(GD_TSS0 >> 3) + cpuid].sd_s = 0;
// Load the TSS selector (like other segment selectors, the
// bottom three bits are special; we leave them 0)
ltr(GD_TSS0 + (cpuid << 3));
// Load the IDT
lidt(&idt_pd);
```

Exercise5 是关于内核锁，由于各个 CPU 共享了相同的内核代码，所以为了防止他们执行中所出现的冲突，同一时间尽可能有一个 CPU 进入内核，所以我们要在进入内核与退出内核的几个节点获取锁或者释放锁，锁的实现已经给好，我们只需要在 i386_init, mp_main, trap 三个地方获取锁, 在 env_run 的时候释放锁即可。

Exercise6 的任务是实现内核中执行环境的调度，调度算法采用了比较简单的 Round-Robin 算法，实现的时候要注意几个问题：1.进入函数时有可能还没有运行任何环境，所以 curenv 有可能是 null，这种情况要考虑；2.如果转了一圈也没有找到可运行的环境，那么我们可以接着运行当前环境。实现代码如下：

```
int i;
if(!curenv){
    for(i = 0; i < NENV; i++){
        if(envs[i].env_status == ENV_RUNNABLE){
            env_run(&envs[i]);
        }
    }
}else{
    envid_t 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){
            env_run(&envs[i]);
        }
    }
    if(curenv->env_status == ENV_RUNNING){
        env_run(curenv);
    }
}
```

Exercise7 就是要实现几个 Env 相关的 system call，在实现 sys_exofork 函数中，有一些地方需要我们注意，我们直接将当前环境的 trapframe 拷贝给了新创建的环境，这符合 fork 的语义，还有就是我们需要将 trapframe 中的 rax 设置为 0，这实际上就是设置在新进程中 fork 的返回值为 0 这一语义。其余的 system call 实现都很简单，在此不予赘述。

二、Copy-on-Write Fork

在 xv6 的 fork 实现中，是将 parent 的地址空间拷贝一份新的到新创建的进程中，这个操作是很费时的，在实际的应用场景中，fork 操作紧接着一般是 exec 操作，这代表我们刚刚拷贝的大部分内存是没有用的，即将被新的内容所覆盖，所以由于这个原因，最近的 Unix 版本利用了虚存映射的特性，使得父进程与子进程在 fork 之后暂时公用一个地址空间，并将共享的页设置为只读，在两个进程中任意一个试图写这个页的时候，就会触发 pagefault，而此时就可以检测到这个 pagefault 是由于 copy-on-write 机制引起的，操作系统此时再去分配新的内存给访问的进程使用。应用这个优化，可以使得 fork 调用的成本大大减小。lab 使用的 JOS 是一个典型的 exo-kernel 架构，我们将 fork 与 copy-on-write 实现为了一个库，这样可以使得内核代码量减少，也使得用户可以根据自己的需要自定义 fork 策略，灵活性大大提高。

而实现用户态 fork 的一个重要前提就是我们应该如何控制 pagefault 检测 copy-on-write，对于 page fault handler 来说，如何根据地址信息判断执行操作是一件复杂的事情，为了灵活性，我们将自定义 page fault 的处理操作，Exercise8 就是实现定义用户 page fault 的系统调用，由于要自定义用户的 pagefault，安全性是一个首要的目标，所以在这个系统调用中，我们将获取 env 时的 checkperm 参数设置为 true，这样只有进程自身与父进程才可以设置用户 pagefault handler。

由于 page fault handler 在用户态执行，那么就不可以使用内核栈来进行操作，JOS 专门为 page fault handler 的执行分配了一个异常栈来执行处理操作，这个异常栈的位置就在用户态栈的上方的一个页。设置好异常栈之后，我们需要在 page fault 并进入 trap 的时候，像 CPU 自动 push trapframe 一样，在异常栈中设置好 Utrapframe，Exercise9 就是要在内核态的 page_fault_handler 设置 UTrapFrame 并回到内核态调用。这里需要注意两个点，一个就是我们需要检测 trapframe 中 esp 的值，如果 esp 就位于异常栈内，我们就需要将 Utrapframe 构造在 esp 的下方并留出 32bits 的空间；第二个就是我们需要检查一下 utrapframe 的构造基址是否是可写的，如果不是就说明我们由于某些错误溢出到了异常栈与用户栈之间的那块内存处。具体代码如下：

```
if(curenv->env_pgfault_upcall == NULL){
    // 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);
}else{
    struct UTrapframe* trapFramePtr;
    //Already in page fault handler
    if(tf->tf_esp <= UXSTACKTOP && tf->tf_esp >= UXSTACKTOP-PGSIZE){
        trapFramePtr = (struct UTrapframe*)(tf->tf_esp - sizeof(void*) - sizeof(struct
                        UTrapframe));
    }else{
        trapFramePtr = (struct UTrapframe*)(UXSTACKTOP - sizeof(struct UTrapframe));
    }
    user_mem_assert(curenv, (void*)trapFramePtr, sizeof(struct UTrapframe), PTE_W);
    trapFramePtr->utf_regs = tf->tf_regs;
    trapFramePtr->utf_eflags = tf->tf_eflags;
    trapFramePtr->utf_eip = tf->tf_eip;
```

```
        trapFramePtr->utf_err = tf->tf_err;
        trapFramePtr->utf_fault_va = fault_va;
        trapFramePtr->utf_esp = tf->tf_esp;
        tf->tf_esp = (uintptr_t) trapFramePtr;
        tf->tf_eip = (uintptr_t)curenv->env_pgfault_upcall;
        env_run(curenv);
    }
```

Exercise10 要实现的是 pagefault handler 的 upcall 帮助函数，这段函数是一段汇编代码，其中有许多 tricky 的部分，为了正常的返回原程序执行位置，在某些操作之后，我们就不可以使用某些特殊指令防止我们修改程序状态，这就需要我们小心的安排我们的汇编代码，我的做法是将 trap 发生的地址放置在 esp 下方的四字节，这样我们就可以等到回到用户态栈再调用 ret 指令。汇编代码如下：

```
_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
    addl $8, %esp            // pop fault-va and errno
    movl 32(%esp) , %eax    // store trap-time eip in %eax
    movl 40(%esp) , %ebx    // store trap-time esp in %ebx
    subl $4 , %ebx
    movl %ebx , 40(%esp)    // store back esp
    movl %eax , (%ebx)
    popal
    addl $4, %esp            // pop trap-time eip
    popf
    popl %esp
    ret
```

Exercise11 要做的就是将_pgfault_handler 这个全局变量设置一下，要注意的是如果这是我们第一次设置，我们还要通过调用几个 system call 实现 upcall 的设置。

Exercise12 就是要实现 Copy-on-write 操作，我们先梳理一下 fork 的流程：1.父进程将 pgfault 设置为用户态 page fault handler；2.父进程调用 sys_exofork 系统调用创建一个子进程；3.对于 UTOP 地址之下的每一个可读页或 COW 页，在两个进程中都重新映射为 COW 页，注意，在这一步中我们重新映射的顺序必须为先映射子进程再映射父进程，这是有原因的，让我们看看先映射父进程会发生什么，如果我们要映射的正好是 esp 所在的页，我们在映射父进程结束后，页面变为 COW，在映射子进程时，由于函数调用导致触发 page fault，重新分配了一个新的可写页，这样在映射之后，子进程是 COW，而父进程却是一个可写页，这不符合 COW 的语义。在代码中还提到了一个点就是原本就是 COW 的页也需要重新映射为 COW，这样做的原因依然和栈有关系，如果当前栈是 COW，那么在我们执行操作的时候它就会触发 page fault 换成一个可读写页，这样也会导致上述情况发生。还有一个需要注意的地方就是，我们不应该把异常栈的那页设置为 COW，否则的话会引起 page fault 的递归调用导致最终崩溃。4.父进程将子进程的 page fault 入口点设置为和自己一样；5.现在子进程已经可以运行，标记为 Runnable。而在 page fault 的 handler 中，我们需要检查 error code 查看是

否是因为写了一个 COW 页导致的 page fault，如果是就重新分配一个页面代替旧的映射。
fork 代码如下：

```
set_pgfault_handler(pgfault);
envid_t envid = sys_exofork();
if (envid < 0)
    panic("sys_exofork: %e", envid);
if (envid == 0) {
    // We're the child.
    // The copied value of the global variable 'thisenv'
    // is no longer valid (it refers to the parent!).
    // Fix it and return 0.
    thisenv = &envs[ENVX(sys_getenvid())];
    return 0;
}
for(int i = 0; i<UTOP/PGSIZE; i++){
    pde_t pde = uvpd[i/NPTENTRIES];
    if(pde & PTE_P){
        duppage(envid, i);
    }
}
sys_env_set_pgfault_upcall(envid, _pgfault_upcall);
sys_env_set_status(envid, ENV_RUNNABLE);
return envid;
```

三、抢占式多任务及进程间通讯

为了实现抢占式调度，我们需要先能够处理诸如时钟这种硬件中断，我按照 lab3 中设置异常的做法，在 trapentry.S 和 trap.c 中设置好了描述符与中断处理程序，在收到 clock 中断时，调用 lapic_eoi 函数回应收到中断，再调用 sced_yield 函数调度下一个可运行进程。这样就完成了 Exercise13 和 Exercise14.

最后一个 Exercise 就是关于 IPC，在 IPC 的实现中，我们使用了 Unix pipe 模型，使用共享内存作为一个消息通道传输信息，具体的流程是这样的：1.需要接受消息的进程调用 ipc_recv 系统调用，并传送一个地址作为接收消息物理页的虚拟映射地址。yield 并等待消息；2.发送方进程调用 ipc_send 系统调用，并传送一个虚拟地址映射给内核，内核会将这个映射后面的物理页映射到接收端那里。按照代码注释上的提示，成功完成了 IPC 各个函数的编写。由于代码篇幅较长不在此列出。

本次 lab 我选择的 challenge 是实现 user-defined processor exceptions handler,实现的过程比较简单，参考 user-defined page fault handler 实现所需代码即可实现，需要注意的地方就是处理器异常不同于 page fault，page fault 的 handler 在执行结束后应返回出错的指令重新开始执行，但是处理器异常一般是不可补救的错误，比如除零错等等，我们在 handler 中也不应该跳过出错的指令，应该是将 environment 销毁，所以我在 user-defined exception 的 upcall 中，添加了一条指令：call exit, 这两条指令的效果等同于调用 exit()函数即 sys_env_destroy(0)函数，销毁当前执行环境。为了能够使得 handler 可以查看当前异常号，我在 UTrapFrame 结构体中添加了 utf_trapno 属性，注意添加该属性时，要注意 utrapFrame 结构发生变化需检查 pagefault upcall 是否还可以正常运行。为了测试，我写了一个简单的测试程序 testexcepthandler 在 user 文件夹，测试程序如下：

```
#include <inc/lib.h>
```

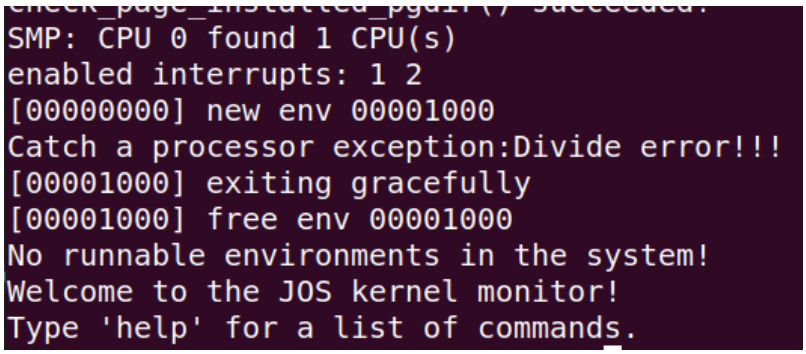
```
int zero;

static const char * const excnames[] = {
        "Divide error",
        "Debug",
        "Non-Maskable Interrupt",
        "Breakpoint",
        "Overflow",
        "BOUND Range Exceeded",
        "Invalid Opcode",
        "Device Not Available",
        "Double Fault",
        "Coprocessor Segment Overrun",
        "Invalid TSS",
        "Segment Not Present",
        "Stack Fault",
        "General Protection",
        "Page Fault",
        "(unknown trap)",
        "x87 FPU Floating-Point Error",
        "Alignment Check",
        "Machine-Check",
        "SIMD Floating-Point Exception"
    };

static void
exception(struct UTrapframe *utf)
{
    int r;
   cprintf("Catch a processor exception:%s!!!\n", excnames[utf->utf_trapno]);
   exit();
}

void
umain(int argc, char **argv)
{
    set_exception_handler(exception);
   zero = 0;
   cprintf("1/0 is %08x!\n", 1/zero);
    return;
}
```

测试结果如下图：



可以看到我们成功触发了 exception handler 并且异常类型正确。