# OSLab4报告

## 操作系统实验三

学号：2313501 姓名：杨楠欣 ;学号：2213230 姓名：向宇涵； 学号：2311366 姓名：邵莫涵

### 练习0：填写已有实验

本实验依赖实验2/3。把小组做的实验2/3的代码填入本实验中代码中有“LAB2”，“LAB3”的注释相应部分。

实际上，只有一处补全相关注释。需要填写lab3代码的文件是 `trap.c` ，补充 lab3 完整的时钟中断逻辑。

### 练习1：分配并初始化一个进程控制块

#### 1. struct context context 含义

`context` 就是进程在**内核态**运行时要恢复的寄存器快照，是进程调度、进程切换用的。

`struct context` 用来保存**内核态“上下文”寄存器**，也就是一组用于进程切换的寄存器状态（例如 `ra` 、 `sp` 、 `s0~s11` 。当内核在不同进程（或内核线程）之间切换时，会把当前进程的这些寄存器值保存到它的 `context` 里，然后从下一个要运行的进程的 `context` 里恢复这些寄存器。这样被切走的进程将来还能从原来的地方继续执行。

当创建一个新内核线程时， `alloc_proc` 会把 `context` 初始化：

- 让 `context.ra` 指向内核线程的入口函数
- 让 `context.sp` 指到该进程自己的内核栈顶部

如此一来，当调度器第一次切换到这个进程时，从 `context` 里恢复寄存器，相当于之前已经在这个线程里运行过，现在继续执行，就自然跳到想要的起始位置运行。

每次调度 `switch_to` 时，也是用 `context` 来保存当前进程寄存器、恢复下一个进程寄存器。

#### 2. struct trapframe *tf 含义

`struct trapframe` 用来保存**一次陷入内核（trap）时 CPU 的全部寄存器状态**，包括：通用寄存器、 `sepc` （异常返回地址）、 `sstatus` 等控制寄存器

当 CPU 因为系统调用/中断/异常跳进内核的 trap 入口时，第一件事就是按照固定格式把当前寄存器压栈，构成一个 `trapframe` ，然后内核通过指针 `tf` 访问它。

对于每个进程来说， `proc_struct` 里的 `tf` 指针，指向**该进程当前 trapframe 所在的位置**。当发生中断/异常/系统调用时：硬件/入口汇编会把寄存器保存到 trapframe；C 语言的 `trap` 函数会拿

到 `struct trapframe *tf`，根据里边的信息决定如何处理，然后修改 `tf->epc` 等字段表示返回到哪儿继续执行。

当从内核返回用户态时，恢复的正是这个 `tf` 里的寄存器：相当于“从 trap 前的现场继续执行”，或者“跳到一个新的入口（比如新进程的第一个指令）”。

此外，创建新进程时，可以伪造一个 `trapframe`，让它里面的 `sepc`、`sp` 等字段设置成想要的用户/内核入口地址和栈顶地址，然后指针 `tf` 指过去。
这样第一次从内核“返回”时，CPU 按照这个 trapframe 恢复寄存器，就像是这个进程是从用户态/内核态正常运行到这里产生了 trap 一样，顺利开始执行。

### 3. 初始化实现过程

`trap.c` 文件中，注释写明 `proc_struct` 中的12个字段需要初始化。

代码块

```
if (proc != NULL)
{
    proc->state = PROC_UNINIT;
    proc->pid = -1;
    proc->runs = 0;
    proc->kstack = 0;
    proc->need_resched = 0;
    proc->parent = NULL;
    proc->mm = NULL;
    memset(&(proc->context), 0, sizeof(struct context));
    proc->tf = NULL;
    proc->pgdir = boot_pgdir_pa;
    proc->flags = 0;
    memset(proc->name, 0, sizeof(proc->name));
    list_init(&(proc->list_link));
    list_init(&(proc->hash_link));
}
```

| 字段 | 初始化值 | 解释 |
| --- | --- | --- |
| state | PROC_UNINIT | 表示进程刚被创建，尚未放入就绪队列，还没准备运行。 |
| pid | -1 | 还没有正式分配 PID。PID 一般在 do_fork() 或 proc_run() 时由系统统一分配。 |
| runs | 0 | 运行次数为 0，说明进程从未被调度运行。 |
| kstack | 0 | 内核栈基地址为0，代表尚未分配。通常在 fork 时分配。 |
| need_resched | 0（false） | 默认不需重新调度，只有运行一段时间或被抢占时才设置为 1。 |
| parent | NULL | 父进程关系未确定，在 do_fork() 里设置 parent = current。 |
| mm | NULL | 默认没有内存管理结构（内核线程的 mm 为 NULL），用户进程 fork 时再分配。 |
| context | 清零 | context 保存进程切换时 CPU 寄存器状态，清零确保切换时不会跳到未知地址。 |
| tf | NULL | trapframe 保存中断/异常时 CPU 状态，只有发生中断或系统调用时才建立。 |
| pgdir | boot_pgdir_pa | 对于内核线程可直接设为 boot_pgdir；用户进程 fork 时分配。 |
| flags | 0 | 清空标志位。可能用于标记 PF_EXITING 等特殊状态，初始化时保持干净。 |
| name[] | 全清零 | 避免脏数据，便于后续用 snprintf(proc->name, …) 填充。 |

此外，进程通过链表管理，因此还需要初始化进程链表节点。`list_link`、`hash_link` 分别用 `list_init` 初始化，避免野指针，否则加入队列时可能破坏链表结构。

## 练习2：为新创建的内核线程分配资源

### 1. 补全 do_fork 函数

目标：完成在kern/process/proc.c中的do_fork函数中的处理过程。大致执行步骤包括：

- 调用alloc_proc，首先获得一块用户信息块。
- 为进程分配一个内核栈。
- 复制原进程的内存管理信息到新进程（但内核线程不必做此事）
- 复制原进程上下文到新进程
- 将新进程添加到进程列表
- 唤醒新进程
- 返回新进程号

代码块

```
1  int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)
2  {
3      ...
4      if ((proc = alloc_proc()) == NULL) {
5          goto fork_out;                    // 分配 PCB 失败
6      }
7
8      if (setup_kstack(proc) != 0) {        // 分配内核栈失败
```

```
 9          goto bad_fork_cleanup_proc;
10      }
11
12      if (copy_mm(clone_flags, proc) != 0) {  // 复制/共享内存失败
13          goto bad_fork_cleanup_kstack;
14      }
15
16      proc->parent = current;                 // 设置父进程
17      proc->pid = get_pid();                  // 分配唯一 PID
18      proc->pgdir = current->pgdir;           // 共享当前页表（内核线程场景）
19      copy_thread(proc, stack, tf);           // 设置 trapframe 与初始上下文
20
21      hash_proc(proc);                        // 加入哈希表
22      list_add(&proc_list, &(proc->list_link)); // 加入进程链表
23      nr_process++;                           // 进程计数 +1
24
25      wakeup_proc(proc);                      // 置为 RUNNABLE
26      ret = proc->pid;                        // 返回子进程 PID
27
28  fork_out:
29      ...
30  ...
31  }
```

关键步骤：

- 分配 PCB：`alloc_proc()`
- 分配内核栈：setup_kstack(proc)
- 复制/共享内存：`copy_mm(clone_flags, proc)`（内核线程这里为空实现）
- 设父进程 / PID / 页表：`proc->parent = current; proc->pid = get_pid(); proc->pgdir = current->pgdir;`
- 设置初始执行现场：`copy_thread(proc, stack, tf)`
- 加入哈希与全局进程链表：`hash_proc` + `list_add`
- 更新全局计数：`nr_process++`
- 唤醒：wakeup_proc(proc)
- 返回子进程 PID

## 2. 说明ucore是否做到给每个新fork的线程一个唯一的id?

ucore 会为每一个通过 `do_fork()` 创建的新进程（线程）分配一个唯一的进程 ID（pid）。在代码中，每次进行 `do_fork()` 的时候就会触发

`proc->pid = get_pid()` 逻辑，这个逻辑就是 为现在复制的进程ID分配一个唯一的ID。

在 `do_fork()` 的执行步骤中，代码首先通过 `if ((proc =alloc_proc()) == NULL) {goto fork_out}` 创建一个新的 `struct proc_struct` 结构，但此时**并没有设置 pid**。随后，在复制父进程信息前，执行 `proc->pid = get_pid()`，`get_pid()` 函数按系统全局 pid 分配策略为每个新创建的进程/线程设定一个新的进程号，不会复用当前正在运行的进程号，因此不会出现进程 ID 重复的情况，也就做到了给每个新fork的线程一个唯一的id。

## 练习3：编写proc_run 函数

### 1. 编写proc_run 函数

#### 函数作用概述

把 CPU 从当前进程切换到目标进程。

#### 函数实现

代码块

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
         * Steps:
         * 1. 关中断
         * 2. 切换 current 指针
         * 3. 切换页表（SATP）
         * 4. 上下文切换 switch_to(from, to)
         * 5. 开中断
         */
        bool intr_flag;
        local_intr_save(intr_flag);
        {
            struct proc_struct *prev = current;
            current = proc;                          // 更新当前进程
            if (prev->pgdir != proc->pgdir) {        // 若页表不同则切换
                lsatp((unsigned int)proc->pgdir);
                barrier();                           // 防止乱序
            }
            cprintf("proc_run: switch %d -> %d\n", prev->pid, proc->pid);
            switch_to(&(prev->context), &(proc->context));
        }
        local_intr_restore(intr_flag);

    }
}
```

- 当要切换到的目标进程不是当前正在运行的进程时，才执行切换操作。
- 首先进行关中断，这是因为更新 current、切换页表并调用 switch_to 需要作为不可被打断的原子步骤执行，关中断可防止在中间状态下被中断处理程序或异常打断并看到不一致的 current或页表，从而避免竞争与错误。
- 然后，将 current（旧进程）保存到prev变量，用于后续上下文切换时的状态保存。更新全局变量 current 为proc（新进程），使系统认知的 “当前运行进程” 变为目标进程。
- 比较旧进程 prev 和新进程 proc 的页表基地址 pgdir 。若两者不同，说明新进程使用独立的地址空间，需通过 lsatp 函数修改 satp 寄存器，将其设置为 proc->pgdir，使 MMU 使用新进程的页表进行地址转换。执行barrier() 内存屏障，防止编译器或 CPU 对指令重排序，确保页表切换完成后再执行后续指令，避免指令在旧页表映射下执行导致地址错误。
- 调用switch_to(&(prev->context), &(proc->context))完成实际的 CPU 状态切换。

  我们可以先看到文件 switch.S

代码块

```
#include <riscv.h>

.text
# void switch_to(struct proc_struct* from, struct proc_struct* to)
.globl switch_to
switch_to:
    # save from's registers
    STORE ra, 0*REGBYTES(a0)
    STORE sp, 1*REGBYTES(a0)
    STORE s0, 2*REGBYTES(a0)
    STORE s1, 3*REGBYTES(a0)
    STORE s2, 4*REGBYTES(a0)
    STORE s3, 5*REGBYTES(a0)
    STORE s4, 6*REGBYTES(a0)
    STORE s5, 7*REGBYTES(a0)
    STORE s6, 8*REGBYTES(a0)
    STORE s7, 9*REGBYTES(a0)
    STORE s8, 10*REGBYTES(a0)
    STORE s9, 11*REGBYTES(a0)
    STORE s10, 12*REGBYTES(a0)
    STORE s11, 13*REGBYTES(a0)

    # restore to's registers
    LOAD ra, 0*REGBYTES(a1)
    LOAD sp, 1*REGBYTES(a1)
    LOAD s0, 2*REGBYTES(a1)
    LOAD s1, 3*REGBYTES(a1)
    LOAD s2, 4*REGBYTES(a1)
    LOAD s3, 5*REGBYTES(a1)
```

```
30      LOAD s4, 6*REGBYTES(a1)
31      LOAD s5, 7*REGBYTES(a1)
32      LOAD s6, 8*REGBYTES(a1)
33      LOAD s7, 9*REGBYTES(a1)
34      LOAD s8, 10*REGBYTES(a1)
35      LOAD s9, 11*REGBYTES(a1)
36      LOAD s10, 12*REGBYTES(a1)
37      LOAD s11, 13*REGBYTES(a1)
38
39      ret
```

该文件实现了进程上下文切换的核心逻辑，其功能是保存当前进程的执行状态并恢复目标进程的执行状态。

函数switch_to的参数为两个指针：from（a0 寄存器传递）和to（a1 寄存器传递），分别指向待切换出的进程（prev）和待切换入的进程（proc）的 context 结构体。

**保存当前进程（from）的寄存器状态**

通过STORE指令将当前 CPU 中的关键寄存器值保存到from->context（即prev->context）中，包括：

- ra：返回地址寄存器，记录当前进程下一步要执行的指令地址
- sp：栈指针寄存器，记录当前进程的内核栈位置
- s0-s11：12 个被调用者保存寄存器。

这些寄存器的状态是进程执行的关键上下文，保存后可保证后续切换回该进程时能继续正常执行。

**恢复目标进程（to）的寄存器状态**

通过LOAD指令从to->context（即proc->context）中加载之前保存的寄存器值到 CPU 中，加载的寄存器与保存的一一对应（ra、sp、s0-s11）。

这一步将目标进程的执行状态恢复到 CPU 中，使得 CPU 接下来的执行基于目标进程的上下文。

**跳转至目标进程的执行点**

最后执行ret指令，该指令会使用刚恢复的ra寄存器的值作为跳转地址，从而让 CPU 开始执行目标进程的指令流，即从proc->context.ra记录的地址继续运行。

- 通过local_intr_restore(intr_flag)恢复之前保存的中断状态。

## 2. 执行过程中，创建且运行了几个内核线程

两个，第 0 个内核线程 `idleproc` 和 第 1 个真正的内核线程 `initproc`。

## Challenge 1：语句如何实现开关中断?

在 proc_run() 中：

local_intr_save(intr_flag); // 关本地中断并把之前状态保存到 intr_flag

执行 current 指针切换、必要时写入 SATP、并调用 switch_to 的操作

local_intr_restore(intr_flag); // 按先前状态还原中断，如果之前是开的就重新开

可以看到 /kern/sync/sync.h 中的相关代码

代码块

```
static inline bool __intr_save(void) {
    if (read_csr(sstatus) & SSTATUS_SIE) {
        intr_disable();
        return 1;
    }
    return 0;
}
#define local_intr_save(x) \
    do {                   \
        x = __intr_save(); \
    } while (0)
```

执行 local_intr_save(intr_flag) 后，传入的 intr_flag 会根据原始中断是否开启被赋值为 1 (开启）或 0（关闭），起一个记录的作用。且无论原始状态如何，当前中断都会被关闭（若原本开启则关闭，原本关闭则保持关闭）。

代码块

```
static inline void __intr_restore(bool flag) {
    if (flag) {
        intr_enable();
    }
}
#define local_intr_restore(x) __intr_restore(x);
```

local_intr_restore(intr_flag) 则会根据之前通过 intr_flag 保存的中断状态，恢复中断的开启或关闭状态，避免因临时关闭中断而影响系统原本的中断配置。

具体地说，若 intr_flag 为 1，即原始中断是开启的，则重新开启中断。若 intr_flag 为 0，即原始中断是关闭的，则保持中断关闭。最终恢复到执行 local_intr_save 之前的中断状态。

## Challenge 2：理解不同分页模式的工作原理

### 1. 两段形式类似的代码为什么相像？

先看到现在的`get_pte()`函数代码，首先我们知道，Sv32是2级页表格式、Sv39是3级页表格式、Sv48是4级页表格式，那么虚拟地址的解析过程都会遵循同一个逻辑，那就是首先按照虚拟地址进行分段、接着每级页表都会使用部分索引页表、接着如果该级不存在这个页表就会进行分配、接着继续下一层的分配、最终找到PTE。

那么不管是在哪一级的页表找特定想要的项，都会执行这一套逻辑，观察我们这里实现的Sv39的3级页表找页表项的代码，可以看到首先在一级页表中进行定位，如果第一级页表项不存在，表示下一级页表还没有进行分配，如果`create==false`，表示只查找、不创建或者分配物理页失败，那么就返回NULL；接着进入第二级页表的处理，从第一级页表项取出物理地址，转成虚拟地址，接着取二级页表项的地址，最终得到PTE的位置。这一整套逻辑适用于多级页表的形式，因此代码会相像。

代码块

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
{
    pde_t *pdep1 = &pgdir[PDX1(la)];
    if (!(*pdep1 & PTE_V))
    {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL)
        {
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }
    pde_t *pdep0 = &((pte_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
    if (!(*pdep0 & PTE_V))
    {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL)
        {
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
```

```
30   }
```

### 2. 页表项的查找和分配合并在一个函数中好吗？有必要拆开吗？

把两个逻辑写在一个函数中有优点也有缺点，对于**好的地方**，写在一起使调用方使用起来更方便，调用者只需要决定是想查一查某个地址是否有PTE还是查不到直接建立中间页表，而这个决定只需要设置create为0或者1就可以，非常方便。

但是对于**不好的地方**，一个函数干两件事情，职责不惮以，在进行调试的时候不容易分清楚失败的原因是什么，究竟是原本就没有这一项还是分配页表失败了；同时这还会对权限和安全不友好，新建的中间页表自带 `PTE_U` 标记，即用户态可访问，如果要实现用户进程和权限的隔离就不友好了。

那么有必要把二者拆开吗，对于ucore现在的代码量是可以不用拆开的，因为现在的使用场景非常有限，只有建立内核线程的栈映射和为简单的内核环境做映射，大部分地方调用这个函数的时候都是希望”查不到就建一个“，因此在这里没必要拆开。