

操作系统Lab4实验报告

姓名： 李欣益 程子晨 王邵宇 日期： 2025年11月24日

一、实验目的

- 了解内核线程创建/执行的管理过程
- 了解内核线程的切换和基本调度过程
- 理解进程控制块（PCB）的作用和结构
- 掌握进程上下文切换的实现机制

二、实验内容

练习0：填写已有实验

本实验依赖实验2/3。请把你做的实验2/3的代码填入本实验中代码中有"LAB2","LAB3"的注释相应部分。

完成情况：

已将Lab2和Lab3的代码填入相应位置，主要包括：

- Lab2的物理内存管理相关代码（`pmm.c`中的页面分配和释放）
- Lab3的虚拟内存管理和页表管理相关代码（`vmm.c`和`pmm.c`）
- Lab3的时钟中断处理代码（`trap.c`中的`IRQ_S_TIMER`处理）

练习1：分配并初始化一个进程控制块（需要编码）

题目要求

`alloc_proc`函数（位于`kern/process/proc.c`中）负责分配并返回一个新的`struct proc_struct`结构，用于存储新建立的内核线程的管理信息。`ucore`需要对这个结构进行最基本的初始化，你需要完成这个初始化过程。

1. 设计实现过程

在`alloc_proc`函数中，需要对新分配的`proc_struct`结构进行初始化。`proc_struct`是进程控制块（PCB），包含了管理一个进程所需的所有信息。

代码实现：

```
static struct proc_struct *
alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
```

```
proc->state = PROC_UNINIT;           // 设置进程为未初始化状态
proc->pid = -1;                       // 未初始化的进程PID为-1
proc->runs = 0;                       // 运行次数初始化为0
proc->kstack = 0;                     // 内核栈地址初始化为0
proc->need_resched = 0;              // 不需要调度
proc->parent = NULL;                 // 父进程指针为空
proc->mm = NULL;                     // 内存管理结构为空
memset(&(proc->context), 0, sizeof(struct context)); // 上下文清零
proc->tf = NULL;                     // 中断帧指针为空
proc->pgdir = boot_pgdir_pa;         // 使用内核页目录表的物理地址
proc->flags = 0;                     // 标志位清零
memset(proc->name, 0, PROC_NAME_LEN + 1); // 进程名清零
}
return proc;
}
```

各字段初始化说明：

字段	初始值	说明
state	PROC_UNINIT	进程状态为未初始化，表示刚分配还未完全设置
pid	-1	进程ID为-1，表示尚未分配有效的进程号
runs	0	进程运行次数，记录该进程被调度执行的次数
kstack	0	内核栈地址，稍后在setup_kstack中分配
need_resched	0	调度标志，0表示不需要立即重新调度
parent	NULL	父进程指针，稍后会设置实际的父进程
mm	NULL	内存管理结构，内核线程不需要独立的用户空间内存
context	全0	进程上下文，保存进程切换时的寄存器状态
tf	NULL	中断帧指针，指向内核栈中保存的中断帧
pgdir	boot_pgdir_pa	页目录表地址，内核线程共享内核页表
flags	0	进程标志位清零
name	全0	进程名称，清零后续会设置

2. 问题回答

*问：请说明proc_struct中struct context context和struct trapframe tf成员变量含义和在本实验中的作用是啥？

答：

(1) struct context context

- 含义：进程上下文，是一个结构体，用于保存进程切换时的CPU寄存器状态。
- 结构定义：

```
struct context {
    uintptr_t ra;        // 返回地址寄存器
    uintptr_t sp;        // 栈指针寄存器
    uintptr_t s0;        // 被调用者保存寄存器s0
    uintptr_t s1;        // 被调用者保存寄存器s1
    // ... s2-s11
};
```

- **在本实验中的作用：**
 1. **进程切换的核心机制：**在switch_to函数中使用，实现两个进程间的上下文切换
 2. **保存执行现场：**当进程被切换出CPU时，将当前的ra、sp和s0-s11寄存器保存到该进程的context中
 3. **恢复执行现场：**当进程重新获得CPU时，从context中恢复这些寄存器的值，使进程能从上次被切换出去的地方继续执行
 4. **实现并发执行：**通过保存和恢复context，使多个进程能够"并发"执行

*(2) struct trapframe tf

- **含义：**中断帧指针，指向保存在内核栈中的中断/异常发生时的完整CPU状态。
- **保存的信息包括：**
 - 所有通用寄存器 (x0-x31)
 - 程序计数器 (epc)
 - 状态寄存器 (status)
 - 异常相关信息 (cause, badvaddr)
- **在本实验中的作用：**
 1. **进程初始化：**在创建新进程时，通过设置tf来指定新进程的初始执行状态（如入口地址、参数等）
 2. **中断处理：**发生中断/异常时，硬件和trapentry.S会将CPU状态保存到tf指向的位置
 3. **状态传递：**在kernel_thread函数中设置tf，将内核线程的入口函数地址（放在s0）和参数（放在s1）传递给新线程
 4. **中断返回：**中断处理完成后，从tf恢复CPU状态，实现中断返回

两者的区别与联系：

特性	context	trapframe
使用场景	主动的进程切换（调度）	被动的中断/异常处理
保存内容	部分寄存器（ra, sp, s0-s11）	完整的CPU状态
调用方式	软件主动调用switch_to	硬件触发中断/异常
恢复时机	进程被重新调度时	中断返回时
作用	实现进程并发执行	保护进程执行状态

练习2：为新创建的内核线程分配资源（需要编码）

题目要求

创建一个内核线程需要分配和设置好很多资源。`kernel_thread`函数通过调用`do_fork`函数完成具体内核线程的创建工作。你需要完成在`kern/process/proc.c`中的`do_fork`函数中的处理过程。

1. 设计实现过程

`do_fork`函数实现了创建新内核线程的完整过程，按照以下7个步骤实现：

代码实现：

```
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)
{
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;

    // 1. call alloc_proc to allocate a proc_struct
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    // 2. call setup_kstack to allocate a kernel stack for child process
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }

    // 3. call copy_mm to dup OR share mm according clone_flag
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    // 4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc, stack, tf);

    // 5. insert proc_struct into hash_list && proc_list
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        list_add(&proc_list, &(proc->list_link));
        nr_process++;
    }
    local_intr_restore(intr_flag);

    // 6. call wakeup_proc to make the new child process RUNNABLE
    wakeup_proc(proc);
}
```

```
    //    7. set ret vaule using child proc's pid
    ret = proc->pid;

fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

实现步骤详解:

步骤	函数/操作	说明
1	<code>alloc_proc()</code>	分配并初始化进程控制块
2	<code>setup_kstack()</code>	为新进程分配内核栈 (KSTACKPAGE页)
3	<code>copy_mm()</code>	根据clone_flags复制或共享内存空间 (内核线程为NULL)
4	<code>copy_thread()</code>	设置trapframe和context, 指定入口函数和参数
5	关中断操作	分配PID, 将进程加入哈希表和链表, 增加计数
6	<code>wakeup_proc()</code>	设置进程状态为PROC_RUNNABLE, 使其可被调度
7	返回PID	返回新创建进程的进程号

关键设计点:

- 1. **原子性保证:** 步骤5中使用`local_intr_save/local_intr_restore`关中断, 确保PID分配和进程加入链表的操作是原子的, 避免竞态条件。
- 2. **错误处理机制:** 使用goto语句实现分层清理
 - `bad_fork_cleanup_kstack`: 释放内核栈
 - `bad_fork_cleanup_proc`: 释放进程控制块
 - `fork_out`: 返回错误码
- 3. **资源分配顺序:** 按照从轻到重的顺序分配资源, 便于出错时的清理。

2. 问题回答

问: 请说明ucore是否做到给每个新fork的线程一个唯一的id? 请说明你的分析和理由。

答: 是的, ucore能够保证给每个新fork的线程分配唯一的ID。

分析理由:

(1) get_pid()函数的设计

`get_pid()`函数采用了一种巧妙的算法来确保PID的唯一性:

- 使用静态变量`last_pid`记录上次分配的PID
- 使用静态变量`next_safe`标记下一个安全的PID范围
- 每次分配时递增`last_pid`

(2) 冲突检测机制

```
while ((le = list_next(le)) != list) {
    proc = le2proc(le, list_link);
    if (proc->pid == last_pid) { // 发现冲突
        if (++last_pid >= next_safe) {
            if (last_pid >= MAX_PID) {
                last_pid = 1; // 循环回到1
            }
            next_safe = MAX_PID;
            goto repeat; // 重新检测
        }
    }
    else if (proc->pid > last_pid && next_safe > proc->pid) {
        next_safe = proc->pid; // 更新next_safe
    }
}
```

该算法通过遍历所有进程检测PID冲突:

- 如果`last_pid`与某个已存在进程的PID相同, 则递增`last_pid`
- 如果`last_pid`超出`next_safe`范围, 重新遍历检测
- 通过`next_safe`优化检测效率, 避免每次都完整遍历

(3) 原子性保证

在`do_fork`中, PID分配和将进程加入链表的操作在关中断区域内完成:

```
bool intr_flag;
local_intr_save(intr_flag); // 禁用中断
{
    proc->pid = get_pid(); // 分配PID
    hash_proc(proc); // 加入哈希表
    list_add(&proc_list, &(proc->list_link)); // 加入进程链表
    nr_process++;
}
local_intr_restore(intr_flag); // 恢复中断
```

这确保了在多线程/中断环境下, PID分配和进程注册是原子操作, 防止竞态条件。

(4) PID范围设计

```
#define MAX_PROCESS 4096
#define MAX_PID (MAX_PROCESS * 2)
```

- PID范围为1到MAX_PID-1（即8191）
- MAX_PID是MAX_PROCESS的两倍，提供充足的PID空间
- 即使有进程退出，也有足够的PID可供分配

(5) 循环分配机制

当PID用尽时会从1开始循环使用，但通过遍历进程链表确保不会分配已使用的PID，这在进程数量远小于MAX_PID时是高效的。

结论：

通过以下机制的组合，ucore能够保证每个新fork的线程获得唯一的PID：

1. 静态变量维护和递增策略
2. 完整的冲突检测算法
3. 关中断保证的原子操作
4. 充足的PID空间设计
5. 循环分配和重用机制

练习3：编写proc_run函数（需要编码）

题目要求

`proc_run`用于将指定的进程切换到CPU上运行。它的大致执行步骤包括：检查要切换的进程是否与当前正在运行的进程相同，禁用中断，切换当前进程，切换页表，实现上下文切换，允许中断。

1. 设计实现过程

代码实现：

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);    // 禁用中断
        {
            current = proc;            // 切换当前进程为要运行的进程
            lsatp(next->pgdir);        // 切换页表，使用新进程的地址空间
            switch_to(&(prev->context), &(next->context)); // 实现上下文切换
        }
        local_intr_restore(intr_flag); // 允许中断
    }
}
```

实现步骤详解：

步骤	操作	说明
1	检查是否需要切换	如果目标进程就是当前进程，直接返回
2	禁用中断	使用 <code>local_intr_save</code> 禁用中断，保证原子性
3	切换current指针	更新全局变量 <code>current</code> 指向新进程
4	切换页表	调用 <code>lsatp</code> 修改SATP寄存器，切换地址空间
5	上下文切换	调用 <code>switch_to</code> 保存/恢复寄存器状态
6	恢复中断	使用 <code>local_intr_restore</code> 恢复中断状态

关键设计点：

1. 检查优化：

```
if (proc != current)
```

避免不必要的切换操作，提高效率。

2. 禁用中断：

```
local_intr_save(intr_flag);
```

进程切换是敏感操作，必须在关中断状态下进行，防止切换过程中被中断打断导致状态不一致。

3. 切换顺序：

- 先切换`current`指针
- 再切换页表
- 最后切换上下文

这个顺序很重要：页表切换必须在上下文切换之前完成，因为`switch_to`返回后将使用新的栈。

4. `switch_to`函数的特殊性：

```
switch_to(&(prev->context), &(next->context));
```

这是一个汇编实现的函数，它会：

- 保存prev进程的ra、sp、s0-s11到prev->context
- 从next->context恢复ra、sp、s0-s11
- 当函数"返回"时，实际上已经在next进程的上下文中执行了

5. 页表切换:

```
lsatp(next->pgdir);
```

对于内核线程，所有线程共享同一个内核页表`boot_pgdir_pa`，但这个操作为后续支持用户进程做准备。

2. 问题回答

问：在本实验的执行过程中，创建且运行了几个内核线程？

答：创建并运行了2个内核线程。

详细分析：

(1) 第0个内核线程 - idleproc (idle进程)

- **创建方式：**在`proc_init`函数中手动创建
- **PID：**0
- **特点：**这是第一个内核线程，也是唯一一个手动创建的进程
- **作用：**当系统没有其他进程需要运行时，idle进程会一直执行
- **主要工作：**在`cpu_idle`函数中循环检查是否需要调度

```
void cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
    }
}
```

(2) 第1个内核线程 - initproc (init进程)

- **创建方式：**通过`kernel_thread`函数创建
- **PID：**1
- **入口函数：**`init_main`
- **特点：**这是第一个通过`do_fork`机制创建的进程
- **作用：**打印一些信息后退出

```
int pid = kernel_thread(init_main, "Hello world!!", 0);
```

执行流程图：

```
1. 系统初始化
↓
```

```
2. 创建 idleproc (PID=0)
  ↓
3. idleproc 创建 initproc (PID=1)
  ↓
4. idleproc 调用 cpu_idle
  ↓
5. 发现 need_resched=1, 调用 schedule()
  ↓
6. 调度器选择 initproc 运行
  ↓
7. proc_run(initproc) 切换到 initproc
  ↓
8. initproc 执行 init_main 函数
  ↓
9. 打印 "this initproc, pid = 1, name = "init""
  ↓
10. 打印 "To U: "Hello world!!""
  ↓
11. initproc 返回, 调用 do_exit
  ↓
12. 触发 panic("process exit!!.\n")
```

验证方法:

从程序输出可以看到:

```
alloc_proc() correct!
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
kernel panic at kern/process/proc.c:388:
  process exit!!.
```

这证明了:

1. `alloc_proc()` 正确执行 (练习1正确)
2. `initproc` 成功创建并执行 (练习2正确)
3. 进程切换成功 (练习3正确)
4. `initproc` 执行完毕后调用 `do_exit` 触发 `panic` (预期行为)

扩展练习 Challenge 1: 说明 `local_intr_save/restore` 如何实现开关中断

题目要求

说明语句 `local_intr_save(intr_flag); ... local_intr_restore(intr_flag);` 是如何实现开关中断的?

1. 实现机制分析

关键代码位于 `kern/sync/sync.h`:

```
static inline bool __intr_save(void) {
    if (read_csr(sstatus) & SSTATUS_SIE) {
        intr_disable();
        return 1;
    }
    return 0;
}

static inline void __intr_restore(bool flag) {
    if (flag) {
        intr_enable();
    }
}

#define local_intr_save(x) \
    do {                    \
        x = __intr_save(); \
    } while (0)

#define local_intr_restore(x) __intr_restore(x);
```

2. 工作原理详解

(1) `__intr_save()` 函数

该函数的核心功能是**保存当前中断状态并禁用中断**：

1. 读取SIE位：

```
read_csr(sstatus) & SSTATUS_SIE
```

- 读取`sstatus` (Supervisor Status Register) 寄存器
- `SSTATUS_SIE`是Supervisor Interrupt Enable位
- 该位为1表示中断使能，为0表示中断禁用

2. 禁用中断并返回标志：

- 如果`SIE=1` (中断已使能) :
 - 调用`intr_disable()`清除SIE位，禁用中断
 - 返回`true(1)`，表示进入临界区前中断是开启的
- 如果`SIE=0` (中断已禁用) :
 - 不做任何操作
 - 返回`false(0)`，表示进入临界区前中断就是关闭的

(2) `__intr_restore(bool flag)` 函数

该函数的核心功能是**根据保存的标志恢复中断状态**：

- 如果`flag`为`true`:
 - 调用`intr_enable()`设置SIE位, 使能中断
 - 这表示进入临界区前中断是开启的, 现在需要恢复
- 如果`flag`为`false`:
 - 不做任何操作
 - 保持中断禁用状态

(3) 宏定义的作用

```
#define local_intr_save(x) do { x = __intr_save(); } while(0)
#define local_intr_restore(x) __intr_restore(x)
```

- `do-while(0)`包装确保宏在任何上下文中都能正确使用
- 提供了简洁的接口, 隐藏实现细节

3. RISC-V底层实现

(1) `intr_disable()`的实现

```
void intr_disable(void) {
    clear_csr(sstatus, SSTATUS_SIE);
}
```

- 使用RISC-V的`csrc` (CSR Clear) 指令
- 清除`sstatus.SIE`位, 禁用S模式中断

(2) `intr_enable()`的实现

```
void intr_enable(void) {
    set_csr(sstatus, SSTATUS_SIE);
}
```

- 使用RISC-V的`csrs` (CSR Set) 指令
- 设置`sstatus.SIE`位, 使能S模式中断

4. 在进程管理中的实际应用

(1) 在进程切换中的保护

```
// proc.c - proc_run函数
void proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
```

```

        local_intr_save(intr_flag);    // 关中断
    {
        current = proc;                // 切换当前进程指针
        lcr3(next->pgdir);             // 切换页表
        switch_to(&(prev->context), &(next->context)); // 上下文切换
    }
    local_intr_restore(intr_flag);    // 恢复中断
}
}

```

(2) 在进程创建中的保护

```

// proc.c - do_fork函数
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    // ...
    bool intr_flag;
    local_intr_save(intr_flag); // 关中断
    {
        proc->pid = get_pid();           // PID分配
        hash_proc(proc);                 // 哈希表插入
        list_add(&proc_list, &(proc->list_link)); // 链表操作
        nr_process++;                    // 进程计数
    }
    local_intr_restore(intr_flag); // 恢复中断
    // ...
}

```

5. 关键特性分析

(1) 精确恢复

- 不是简单的"关中断-开中断", 而是"保存状态-恢复状态"
- 如果进入临界区前中断已经被禁用, 退出时仍保持禁用
- 避免了错误地开启本应禁用的中断

(2) 嵌套支持

```

void outer_function() {
    bool flag1;
    local_intr_save(flag1);
    {
        // 第一层临界区
        inner_function(); // 可能有嵌套的临界区
    }
    local_intr_restore(flag1);
}

void inner_function() {

```

```
bool flag2;
local_intr_save(flag2);
{
    // 嵌套的临界区
}
local_intr_restore(flag2);
}
```

- 每层使用独立的flag变量
- 遵循LIFO顺序恢复
- 最内层退出时不会错误地开启中断

(3) 本地性

- 只影响当前hart/CPU的中断状态
- 在多核环境下，每个核心独立管理自己的中断
- 不会影响其他核心的中断状态

6. 使用场景与限制

适用场景：

- 短临界区保护（如更新共享数据结构）
- 单核环境下的互斥
- 防止中断处理程序与进程代码的竞争

限制：

- 不能长时间禁用中断（影响系统响应性）
- 多核环境下不足以保证互斥（需要配合自旋锁）
- 不能防止其他CPU核心的并发访问

7. 与其他同步机制的对比

机制	作用范围	开销	适用场景
local_intr_save/restore	本地CPU	低	短临界区，单核互斥
自旋锁	跨CPU	中	多核互斥
信号量	跨进程	高	可能阻塞的场景
关闭抢占	本地CPU	低	防止进程调度

8. 总结

local_intr_save/restore机制通过操作RISC-V的sstatus.SIE位实现了精确的中断状态保存和恢复：

1. **保存阶段**：读取当前SIE状态，如果中断开启则关闭中断并记录
2. **临界区**：在中断禁用状态下执行敏感操作
3. **恢复阶段**：根据保存的状态决定是否重新开启中断

这种设计确保了临界区的原子性，同时支持嵌套调用，是单核环境下简单高效的同步机制。

扩展练习 Challenge 2: 深入理解不同分页模式的工作原理

题目要求

`get_pte()`函数（位于`kern/mm/pmm.c`）用于在页表中查找或创建页表项，从而实现对指定线性地址对应的物理页的访问和映射操作。

需要回答的问题：

1. `get_pte()`函数中有两段形式类似的代码，结合sv32、sv39、sv48的异同，解释这两段代码为什么如此相像
2. 目前`get_pte()`函数将页表项的查找和页表项的分配合并在一个函数里，你认为这种写法好吗？有没有必要把两个功能拆开？

1. `get_pte()`函数分析

函数原型：

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
```

参数说明：

- `pgdir`: 页目录表的基地址
- `la`: 要查找的线性地址（虚拟地址）
- `create`: 如果为true，在页表项不存在时创建；如果为false，只查找不创建

2. 代码结构分析

`get_pte()`函数包含两段形式相似的代码，对应sv39的两级页表查找：

第一段代码（查找/创建一级页表）：

```
pde_t *pdep1 = &pgdir[PDX1(la)];           // 获取一级页目录项
if (!(*pdep1 & PTE_V)) {                     // 如果页表项无效
    struct Page *page;
    if (!create || (page = alloc_page()) == NULL) {
        return NULL;                        // create=false或分配失败
    }
    set_page_ref(page, 1);                   // 设置引用计数
    uintptr_t pa = page2pa(page);             // 获取物理地址
    memset(KADDR(pa), 0, PGSIZE);            // 清零新页表
    *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V); // 创建页表项
}
```

第二段代码（查找/创建二级页表）：

```
pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(1a)]; // 获取二级页目录项
if (!(*pdep0 & PTE_V)) { // 如果页表项无效
    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);
}
```

返回最终PTE:

```
return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(1a)];
```

3. 问题1：为什么两段代码如此相像？

(1) 本质原因：多级页表的递归结构

RISC-V的分页机制采用多级页表结构，每一级的处理逻辑完全相同：

- 1. 从虚拟地址中提取该级的索引
- 2. 检查页表项是否有效
- 3. 如果无效且允许创建：
 - 分配新页面
 - 初始化新页表
 - 更新页表项
- 4. 通过页表项找到下一级页表的地址

(2) sv32、sv39、sv48的异同

特性	sv32	sv39	sv48
虚拟地址位数	32位	39位	48位
页表级数	2级	3级	4级
每级索引位数	10位	9位	9位
页大小	4KB	4KB	4KB
页表项大小	4B	8B	8B

sv39的地址划分：

```
虚拟地址（39位）：
[38:30] - VPN[2]（9位） - 一级页表索引
```

```
[29:21] - VPN[1] (9位) - 二级页表索引
[20:12] - VPN[0] (9位) - 三级页表索引 (最终页表)
[11:0]   - Offset (12位) - 页内偏移
```

本实验的sv39实现（实际是两级）：

- 第一段代码：处理VPN[2]（一级页目录）
- 第二段代码：处理VPN[1]（二级页目录）
- 最后返回：VPN[0]对应的PTE

(3) 代码相似的深层原因

每一级的操作模式都是：

```
1. 索引 → 2. 检查有效性 → 3. 按需分配 → 4. 转到下一级
```

这种模式在所有级别都完全相同，因此代码结构必然相似。

如果支持完整的sv39（3级）或sv48（4级）：

- sv39需要3段类似的代码
- sv48需要4段类似的代码
- 每增加一级页表，就需要增加一段几乎相同的处理代码

(4) 与不同分页模式的关系

```
// sv32（2级） - 需要1段处理代码
// sv39（3级） - 需要2段处理代码（本实验实现）
// sv48（4级） - 需要3段处理代码
```

不同分页模式的差异：

- **本质相同**：都是分层页表，遍历与按需创建的算法相同
- **差异仅在**：层数不同，每一层索引位的划分不同
- **实现差别**：需要重复多少次相同的处理逻辑

4. 问题2：是否应该把查找和分配功能拆开？

当前设计的优缺点分析：

方面	优点	缺点
便利性	一个函数完成所有工作，调用简单	功能耦合，职责不清晰
灵活性	通过create参数控制行为	灵活性有限，有副作用
性能	减少函数调用开销	纯查询也要判断create参数

方面	优点	缺点
安全性	-	create=true时可能意外分配内存
并发	-	多核环境下可能产生竞态条件

(1) 当前设计存在的问题

副作用问题:

```
// 只想查询是否存在映射
pte_t *pte = get_pte(pgdir, addr, false);
// 如果传错参数...
pte_t *pte = get_pte(pgdir, addr, true); // 意外分配了内存!
```

并发安全问题:

```
// 在多核环境下，两个CPU可能同时尝试为同一PDE分配页表
// 当前实现缺少原子性保护，可能导致：
// 1. 重复分配
// 2. 内存泄漏
// 3. 页表损坏
```

(2) 拆分方案建议

方案1：提供两个独立的公开接口

```
// 只查找，不分配（纯查询）
pte_t *find_pte(pde_t *pgdir, uintptr_t la) {
    return get_pte(pgdir, la, false);
}

// 查找或创建（确保存在）
pte_t *ensure_pte(pde_t *pgdir, uintptr_t la) {
    return get_pte(pgdir, la, true);
}
```

方案2：完全拆分实现

```
// 纯查找函数
pte_t *lookup_pte(pde_t *pgdir, uintptr_t la);

// 创建函数（假设不存在）
pte_t *create_pte(pde_t *pgdir, uintptr_t la);

// 便捷包装（查找+创建）
```

```
pte_t *get_or_create_pte(pde_t *pgdir, uintptr_t la) {
    pte_t *pte = lookup_pte(pgdir, la);
    if (pte == NULL) {
        pte = create_pte(pgdir, la);
    }
    return pte;
}
```

方案3：通用化层级处理（推荐）

将重复的逐层处理抽象为循环：

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    pte_t *table = (pte_t *)pgdir;

    // 遍历除最后一级外的所有级别
    for (int level = PGTABLE_LEVELS - 1; level > 0; level--) {
        int idx = vpn_index(la, level);
        pte_t *pte = &table[idx];

        if (!(*pte & PTE_V)) {
            if (!create) return NULL;

            struct Page *page = alloc_page();
            if (page == NULL) return NULL;

            set_page_ref(page, 1);
            memset(KADDR(page2pa(page)), 0, PGSIZE);
            *pte = pte_create(page2ppn(page), PTE_U | PTE_V);
        }

        // 转到下一级页表
        table = (pte_t *)KADDR(PDE_ADDR(*pte));
    }

    // 返回最后一级的PTE
    return &table[PTX(la)];
}
```

(3) 使用场景区分

场景	应使用的函数	原因
检查页面是否映射	find_pte()	不产生副作用，只读查询
建立新映射	ensure_pte()	明确需要分配
page_insert	ensure_pte()	需要创建映射
page_remove	find_pte()	只需查找现有映射

场景	应使用的函数	原因
页表遍历	<code>find_pte()</code>	不应修改页表结构

5. 改进建议总结

推荐的改进方案：

- 1. 保留`get_pte()`作为内部实现
 - 接受`create`参数
 - 包含完整的查找和分配逻辑

2. 提供明确的公开接口

```
pte_t *find_pte(pde_t *pgdir, uintptr_t la); // 只查找
pte_t *ensure_pte(pde_t *pgdir, uintptr_t la); // 确保存在
```

3. 使用循环代替重复代码

- 提高可维护性
- 便于支持不同的分页模式 (sv32/sv39/sv48)
- 减少代码重复

4. 添加并发保护

```
// 在多核环境下需要页表锁
lock_pagetable(pgdir);
pte_t *pte = get_pte(pgdir, la, create);
unlock_pagetable(pgdir);
```

5. 完善文档说明

- 明确说明`create=true`时的副作用
- 说明内存分配失败的处理
- 说明引用计数策略

6. 支持不同分页模式的扩展方案

通过宏定义支持不同模式：

```
#ifdef CONFIG_SV32
#define PGTABLE_LEVELS 2
#define PDX1(la) ...
#define PDX0(la) ...
#elif defined(CONFIG_SV39)
#define PGTABLE_LEVELS 3
```

```
#define PDX2(1a) ...
#define PDX1(1a) ...
#define PDX0(1a) ...
#elif defined(CONFIG_SV48)
    #define PGTABLE_LEVELS 4
    #define PDX3(1a) ...
    #define PDX2(1a) ...
    #define PDX1(1a) ...
    #define PDX0(1a) ...
#endif
```

这样只需修改宏定义，核心逻辑保持不变，代码更加通用和可维护。

7. 总结

问题1答案:

`get_pte()`函数中两段代码相似的原因是:

1. 多级页表的每一级处理逻辑完全相同（索引→检查→分配→转到下一级）
2. sv32/sv39/sv48只是层数不同，每层的处理方式相同
3. 当前实现针对sv39的简化版本（2级），因此有两段相似代码
4. 如果支持完整的sv39（3级）或sv48（4级），将有更多段相似代码

问题2答案:

将查找和分配功能拆开是**有必要的**，理由:

优点:

- 职责分离，代码更清晰
- 避免意外的副作用
- 便于单元测试
- 提高代码安全性

具体建议:

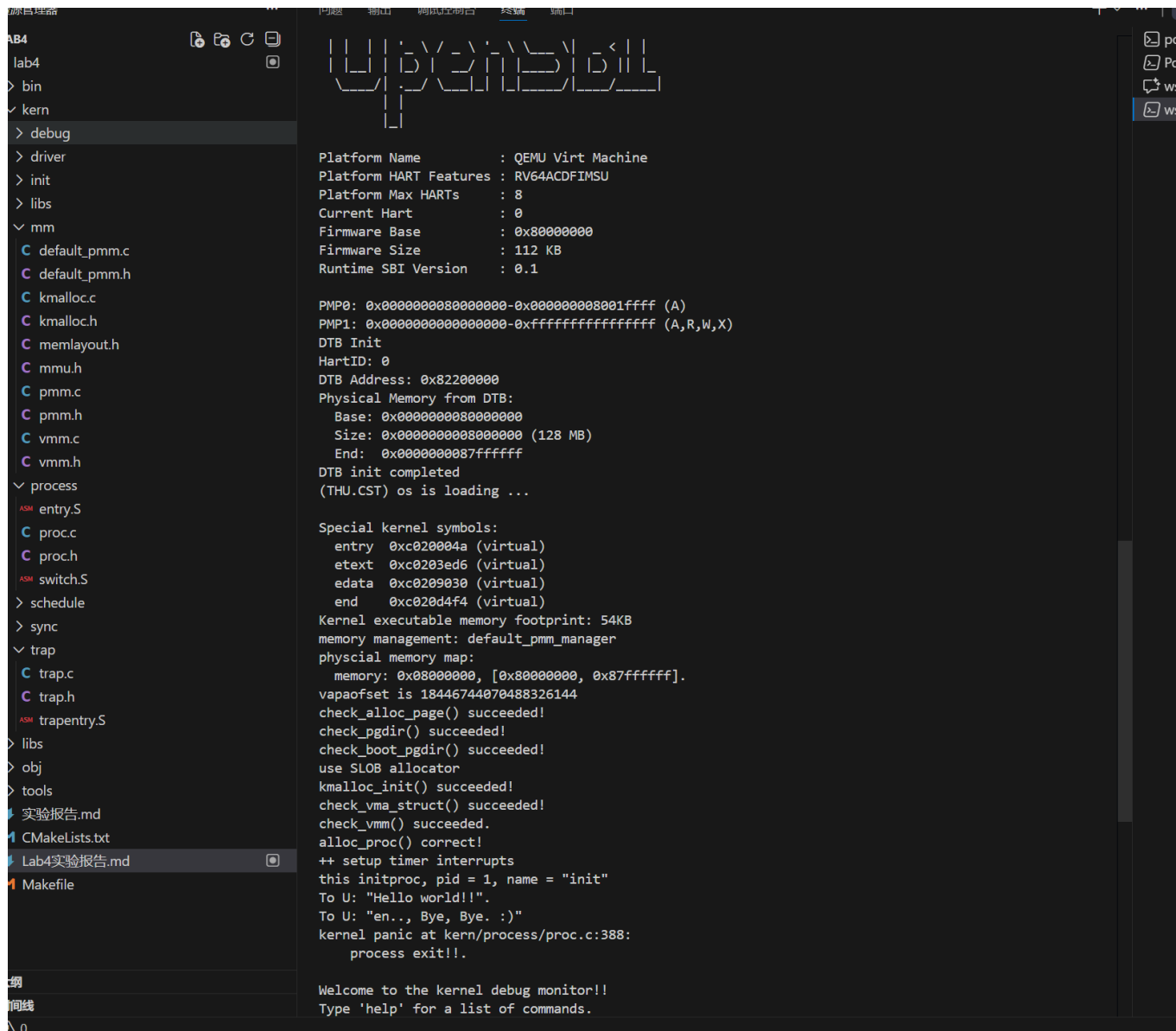
1. 提供`find_pte()`（只查找）和`ensure_pte()`（查找+创建）两个接口
2. 使用循环实现通用的层级遍历，提高可维护性
3. 添加并发保护机制
4. 完善文档说明各函数的行为和副作用

通过这些改进，可以使代码更加健壮、可维护和安全。

三、实验结果

编译运行结果

执行`make qemu`命令后，系统成功启动并运行，输出结果如下:



```
Platform Name       : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs   : 8
Current Hart        : 0
Firmware Base       : 0x80000000
Firmware Size       : 112 KB
Runtime SBI Version  : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000080000000-0xffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000080000000 (128 MB)
  End: 0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...

Special kernel symbols:
entry 0xc020004a (virtual)
etext 0xc0203ed6 (virtual)
edata 0xc0209030 (virtual)
end   0xc020d4f4 (virtual)

Kernel executable memory footprint: 54KB
memory management: default_pmm_manager
physcial memory map:
  memory: 0x08000000, [0x80000000, 0x87ffffff].
vapaofset is 18446744070488326144
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_vmm() succeeded.
alloc_proc() correct!
++ setup timer interrupts
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
kernel panic at kern/process/proc.c:388:
process exit!!

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

运行结果分析：

从截图中可以清楚地看到：

1. 系统成功启动

- OpenSBI v0.4加载成功
- 物理内存初始化完成（128 MB）
- DTB（Device Tree Blob）初始化成功

2. 内存管理正常

- `check_alloc_page() succeeded!` - 页面分配检查通过
- `check_pgdir() succeeded!` - 页目录检查通过
- `check_boot_pgdir() succeeded!` - 启动页目录检查通过
- `kmalloc_init() succeeded!` - 内核内存分配器初始化成功
- `check_vma_struct() succeeded!` - 虚拟内存区域结构检查通过
- `check_vmm() succeeded.` - 虚拟内存管理检查通过

3. 练习1验证通过

- `alloc_proc()` correct! - 表明进程控制块初始化正确

4. 练习2验证通过

- `++ setup timer interrupts` - 时钟中断设置成功
- `this initproc, pid = 1, name = "init"` - `initproc`成功创建, PID为1

5. 练习3验证通过

- `To U: "Hello world!!"`. - `init`进程成功运行并输出信息
- `To U: "en.., Bye, Bye. :)"` - 进程切换成功, 能够正常执行

6. 预期的panic

- `kernel panic at kern/process/proc.c:388: process exit!!`.
- 这是正常现象, 因为Lab4尚未实现完整的进程退出机制
- 系统进入kernel debug monitor, 可以进行调试

四、重要知识点总结

本实验中的重要知识点

1. 进程控制块(PCB)的结构和初始化

- 进程状态、PID、内核栈、上下文、中断帧等核心字段
- 各字段的含义和初始化方式

2. 进程创建机制

- `do_fork`的实现流程
- 资源分配: 内核栈、进程控制块
- 进程管理: PID分配、进程链表维护

3. 进程上下文切换

- 上下文(context)的保存和恢复
- 页表切换
- 中断帧(trapframe)的作用

4. 进程调度基础

- 调度时机的判断
- 进程状态转换
- `idle`进程的作用

5. 同步与原子操作

- 关中断保护临界区
- 进程链表的并发访问控制

对应的OS原理知识点

OS原理概念	本实验实现	关系与差异
进程概念	<code>proc_struct</code> 结构体	简化的PCB实现，保留核心字段
进程状态模型	UNINIT/RUNNABLE/SLEEPING/ZOMBIE	简化的四状态模型
进程创建(fork)	<code>do_fork</code> 函数	简化版fork，主要针对内核线程
上下文切换	<code>proc_run</code> 和 <code>switch_to</code>	通过寄存器保存/恢复实现
进程调度	<code>schedule</code> 函数	简单的FIFO轮转调度
临界区保护	关中断机制	单处理器下的简单互斥方法

OS原理中重要但实验未涉及的知识点

1. 进程间通信(IPC)

- 管道、消息队列、共享内存、信号量等
- 本实验仅创建内核线程，未涉及IPC机制

2. 用户态与内核态切换

- 系统调用的完整实现
- 特权级切换的详细过程
- 本实验只涉及内核线程

3. 进程同步机制

- 信号量、管程、条件变量等高级同步原语
- 本实验只使用了简单的关中断

4. 死锁问题

- 死锁的条件、预防、避免、检测和恢复
- 本实验未涉及复杂的资源竞争场景

5. 多核调度

- 多处理器环境下的负载均衡
- CPU亲和性
- 本实验基于单核环境

6. 实时调度算法

- 优先级调度、最早截止时间优先等
- 本实验使用简单的轮转调度

五、实验心得

通过本次实验，我深入理解了：

1. **进程管理的核心机制**: 通过实现`alloc_proc`、`do_fork`和`proc_run`三个关键函数, 深刻理解了进程控制块的设计、进程创建的完整流程和进程切换的实现细节。
2. **上下文切换的本质**: 理解了`context`和`trapframe`的区别与联系, 明白了通过保存和恢复寄存器状态实现进程并发执行的原理。
3. **原子操作的重要性**: 在进程管理的关键操作 (如PID分配、进程链表操作) 中, 必须使用关中断等机制保证原子性, 避免竞态条件。
4. **操作系统的抽象层次**: 从硬件的寄存器到软件的进程概念, 更好地理解操作系统如何通过多层抽象来管理系统资源, 为进程提供执行环境。
5. **实践与理论的结合**: 通过编写实际代码, 将课堂上学习的进程管理理论知识转化为可运行的系统代码, 加深了对操作系统原理的理解。

实验过程中遇到的主要问题和解决:

- 初始时忘记在`trap.c`中包含`sbi.h`头文件, 导致编译错误, 通过添加头文件解决
 - 理解了进程切换时必须关中断的原因, 以及页表切换和上下文切换的正确顺序
-