进程管理

操作系统 lab4 实验报告

梁景铭　王一诺　强博
南开大学计算机学院

2025 年 11 月 23 日

摘要

本实验报告详细记录了 ucore Lab4 **进程管理**的实现过程。在已有物理内存管理的基础上，实验引入了**内核线程**机制，实现了线程的创建、调度与**上下文切换**。本文通过实现 `alloc_proc`（分配进程控制块）、`do_fork`（创建新线程）和 `proc_run`（执行线程）等核心函数，构建了**进程控制块**（`proc_struct`），并成功创建了 `idleproc`（空闲线程）和 `initproc`（初始线程）。实验最终将 ucore 从单一执行流的内核演进为支持多线程调度的内核。本文还深入分析了 `schedule` 调度器、`switch_to` 上下文切换、`get_pte` 页表遍历以及 `local_intr_save` 中断屏蔽等关键功能的实现原理。

关键词：进程管理；内核线程；上下文切换；do_fork；调度器

目录

1 实验内容与目标

1.1 实验目的

- 了解虚拟内存管理的基本结构，掌握虚拟内存的组织与管理方式
- 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

1.2 实验内容

在前面的实验中，我们已实现**物理内存管理**和**基础页表**，但当前系统仍为**单一执行流**，无法并发执行多任务，也未体现**虚拟内存隔离**的作用。

本实验将在此基础上进一步扩展，完成以下两方面内容：

首先，**完善虚拟内存管理**，实现基本的地址空间结构。通过引入**虚拟内存描述结构**，管理进程或线程的虚拟地址空间布局。与后续实验不同，本实验中的虚拟内存仍采用**预映射**方式，即在建立地址空间时一次性完成所有映射，不涉及按需分配或页面置换。

其次，**引入内核线程机制**，实现多执行流并发运行。内核线程是一种特殊的“进程”，让多个程序能**分时使用 CPU**。本实验将实现**线程控制块**、**上下文切换**和**调度器**，使内核能够调度多个执行实体轮流运行。

内核线程与用户进程的区别如下：

表 1: 内核线程与用户进程对比

比较项	内核线程	用户进程
运行模式	仅在内核态运行	在用户态和内核态之间切换
地址空间	共享内核地址空间	拥有独立的用户虚拟地址空间

通过本实验，系统将从“单一执行流的内核”发展为“**支持多线程调度的内核**”，为后续实现**用户进程**、**系统调用**、**缺页异常处理**等功能奠定基础。

需要注意的是，在 ucore 的调度和执行管理中，对**线程和进程做了统一的处理**。且由于 ucore 内核中的所有内核线程共享一个内核地址空间，所以它们从属于同一个**唯一的内核进程**，即 ucore 内核本身。

2 ucore 实验文档学习

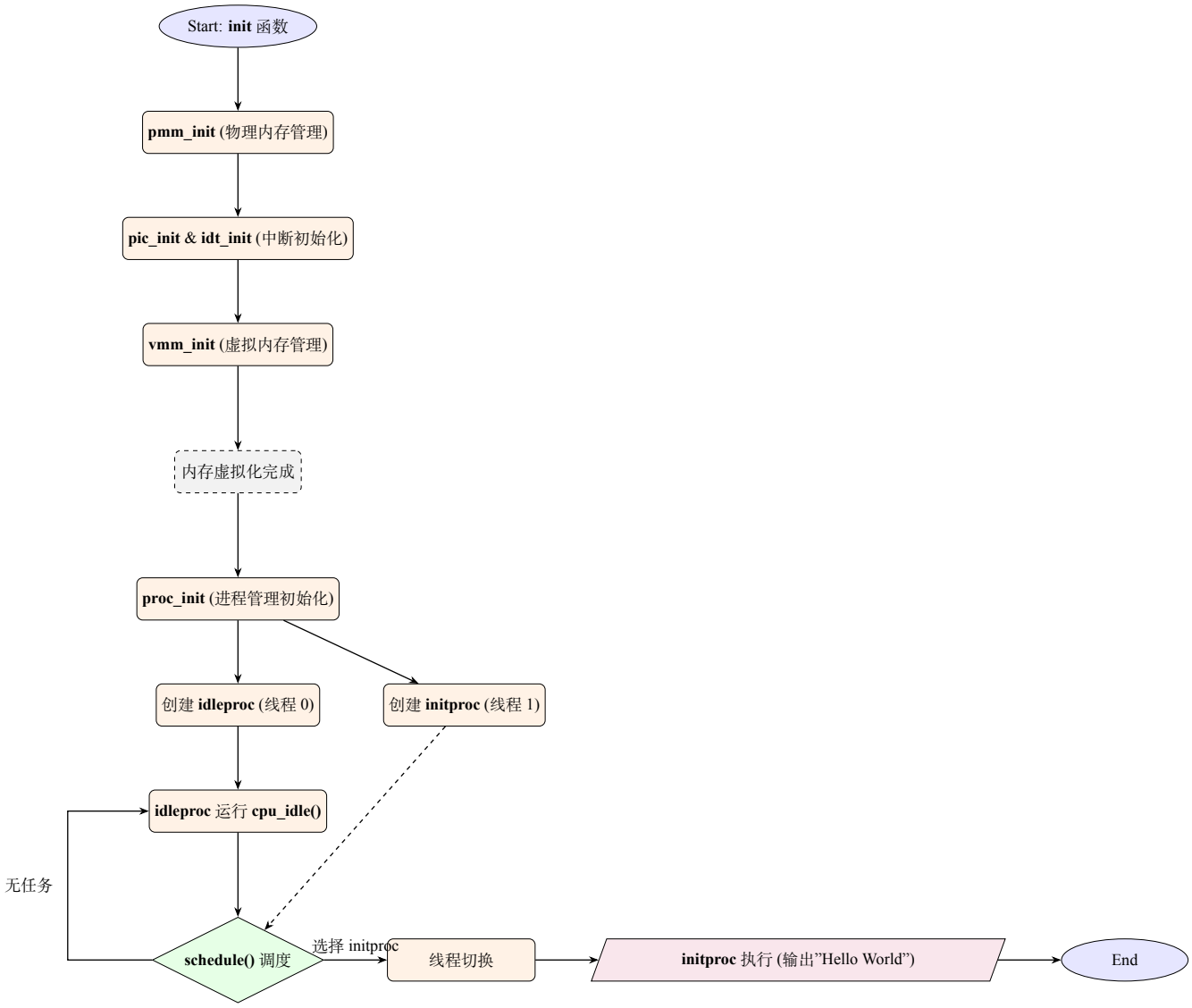
2.1 实验执行流程概述

实验流程始于 **init** 函数。首先进行**内存虚拟化**：调用 **pmm_init**（物理内存）、**pic_init/idt_init**（中断）和 **vmm_init**（虚拟内存）来初始化内核页表，完成静态映射。

随后进行 **CPU 虚拟化**：调用 **proc_init**（进程管理）创建两个内核线程：**idleproc**（空闲线程）和 **initproc**（任务线程）。

最后进入调度执行：**idleproc** 运行 **cpu_idle()**。当 **schedule()** 检测到 **initproc** 可运行时，触发**线程切换**，**initproc** 获得 CPU 执行权并输出”Hello World”。

图 1: 实验执行流程概述图



下面我们将首先分析如何使用多级页表进行虚拟内存管理，具体分析主要需要注意的关键问题和涉及的关键数据结构。

2.2 虚拟内存管理

2.2.1 基本原理概述

虚拟内存是程序员或 CPU “看到” 的逻辑内存。它与物理内存并非一一对应，二者地址通常不同。操作系统通过**内存映射**将虚拟地址自动转换为物理地址。

虚拟化的核心作用是**内存地址虚拟化**。在分页机制下，CPU 访问的是虚拟地址，而非物理地址。这使得操作系统能通过设置页表项来限定软件的访问空间，从而实现**内存访问保护**，防止程序越界。

在此基础上，操作系统可以实现更高级的内存管理技术。例如**按需分页**，即只在程序实际访问某个虚拟地址时才为其分配物理内存。此外，还有**页换入换出**技术，它将不常用的数据从内存移至硬盘，在需要时再读回内存。这些技术能为程序提供更大的逻辑内存空间，支持更多程序并发运行。

2.2.2 页表项设计思路

在 ucore 中，我们为 RISC-V 的 Sv39 三级页表定义了一系列操作宏。我们将高阶页表称为**页目录**，并将三级页表的索引分别命名为 **PDX1**（一级页目录索引）、**PDX0**（二级页目录索引）和 **PTX**（页表索引）。

相关的宏定义（位于 `kern/mm/mmu.h`）为虚拟内存管理提供了基础。下面列出了一些最关键的定义：

Listing 1: 关键宏定义 (mmu.h)

```
// 虚拟地址分解
// PDX1: 一级页目录索引 (VPN[2])
#define PDX1(la) ((((uintptr_t)(la)) >> PDX1SHIFT) & 0x1FF)
// PDX0: 二级页目录索引 (VPN[1])
#define PDX0(la) ((((uintptr_t)(la)) >> PDX0SHIFT) & 0x1FF)
// PTX: 页表索引 (VPN[0])
#define PTX(la) ((((uintptr_t)(la)) >> PTXSHIFT) & 0x1FF)

// 页内偏移
#define PGOFF(la) (((uintptr_t)(la)) & 0xFFF)

// 从页表项(PTE)中提取物理地址
// 将PTE中的PPN字段取出并左移，转换为物理地址
#define PTE_ADDR(pte)   (((uintptr_t)(pte) & ~0x3FF) << (PTXSHIFT -
    PTE_PPN_SHIFT))

// 页表项 (PTE) 标志位
#define PTE_V     0x001 // Valid (有效位)
#define PTE_R     0x002 // Read (可读)
#define PTE_W     0x004 // Write (可写)
#define PTE_X     0x008 // Execute (可执行)
#define PTE_U     0x010 // User (用户态可访问)
```

2.2.3 使用多级页表实现虚拟存储

为实现虚拟存储，关键在于动态修改内存中的页表。这主要通过在 `kern/mm/pmm.c` 中实现两个核心接口来完成：**page_insert** (建立映射) 和 **page_remove** (删除映射)。

测试与验证：我们通过 **check_pgdir** 函数来测试这些接口。该函数会尝试分配物理页面，并将其映射到虚拟地址（如 `0x0` 和 `PGSIZE`），然后检查 **page_ref**（物理页引用计数）是否正确，最后再移除这些映射，确保所有资源被正确释放。

核心实现：页表遍历：所有页表操作都依赖于 **get_pte** 函数，它负责遍历三级页表以找到虚拟地址对应的最终页表项（PTE）。如果 `create` 标志为真，它还会在遍历过程中**按需分配**缺失的页表（即页目录）。

Listing 2: get_pte 函数逻辑 (pmm.c)

```
// 寻找(有必要的时候分配)一个页表项
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    // 1. 找到一级页目录项 (L2 Page Table)
    pde_t *pdep1 = &pgdir[PDX1(la)];
    if (!(*pdep1 & PTE_V)) { // 如果下一级页表不存在
        // 如果 create 为 true, 则 alloc_page() 分配新页作为下一级页表
        // 并设置 PTE_V, PTE_U 标志位
        // ... (分配 L2 页表) ...
    }

    // 2. 找到二级页目录项 (L1 Page Table)
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
    if (!(*pdep0 & PTE_V)) { // 如果下一级页表不存在
        // 如果 create 为 true, 则 alloc_page() 分配新页作为下一级页表
        // ... (分配 L1 页表) ...
    }

    // 3. 返回最终的页表项 (PTE) 地址
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

插入与移除：**page_insert** 函数首先调用 **get_pte** (并设置 `create=1`) 来获取页表项。然后，它处理旧的映射（如果存在），增加新物理页的引用计数，最后构造新的 PTE 并**刷新 TLB**。而 **page_remove** 则调用 **page_remove_pte**，后者负责递减引用计数，并在计数为 0 时释放物理页面，最后清空 PTE。

内核重映射：启动时 `entry.S` 中建立的 **Giga Page** 映射虽然能让内核运行，但它过于粗糙，导致所有段（如 `.text`, `.rodata`, `.data`）都具有相同的读写执行权限，这存在安全隐患。

解决方案是进行**精细化重映射**：放弃现有的页表，**新建一个页表**，并为内核的 `.text`（可读、可执行）、`.rodata`（可读）、`.data` 和 `.bss`（可读、可写）等段设置正确的访问权限。最后，将 **satp** 寄存器指向这个新的页表基地址，完成切换。

2.3 内核线程管理

2.3.1 进程与线程的概念

程序是静态的可执行文件，而**进程**是程序被加载到内存中开始执行的实例。进程不仅包含静态的代码，还包括运行时的堆栈、寄存器等动态数据。

线程是进程中 “正在运行” 的部分，是**可以被调度的最小单元**。一个进程可以包含多个线程，它们共享相同的代码和内存空间，但拥有各自独立的 CPU 执行状态。在 ucore 中，**进程**更多地

作为**资源管理**的实体。

2.3.2 进程管理机制

引入进程是为了实现**任务调度**、充分利用**多核心处理器**以及实现**分时操作系统**，让多个用户或任务能“同时”使用计算机。

内核线程是一种特殊的进程，它只运行在**内核态**，并且所有内核线程**共享 ucore 内核内存空间**。相比之下，用户进程会在用户态和内核态切换，并拥有独立的内存空间。

为了管理线程，操作系统必须为每个线程设计一个**进程控制块** (在 ucore 中为 **proc_struct**)。这些控制块被组织在**进程控制块链表**中，便于管理。**调度器**负责按照特定策略选择线程，使其获得 CPU 执行权。本实验将创建两个内核线程：**idleproc**（空闲线程）和 **initproc**（第一个实际任务线程），初步实现进程的创建、调度和切换。

2.3.3 设计关键数据结构

进程控制块是在 ucore 中用于管理进程的核心数据结构，定义为 `struct proc_struct`（位于 `kern/process/proc.h`）。

Listing 3: 进程控制块 (proc_struct) 关键字段

```
struct proc_struct {
    enum proc_state state;      // 进程状态 (如: UNINIT, RUNNABLE)
    int pid;                    // 进程ID
    uintptr_t kstack;           // 内核栈的地址
    volatile bool need_resched; // 是否需要调度
    struct proc_struct *parent; // 父进程指针
    struct mm_struct *mm;       // 内存管理结构
    struct context context;     // 进程上下文 (用于切换)
    struct trapframe *tf;       // 中断帧 (用于中断/异常)
    uintptr_t pgdir;            // 页目录基址 (satp寄存器)
    list_entry_t list_link;     // 进程链表
    list_entry_t hash_link;     // 哈希表链表
};
```

- **mm**: 保存内存管理信息，包括虚拟内存映射。
- **state**: 进程所处的状态，如 `PROC_UNINIT`（未初始化）、`PROC_RUNNABLE`（可运行）等。
- **parent**: 指向父进程。所有进程构成一棵进程树。
- **context**: 保存进程切换时需要恢复的**关键寄存器**（如 `ra`, `sp`, `s0-s11`），用于恢复执行现场。
- **tf** (中断帧): 当进程从用户态进入内核态（如系统调用）时，保存其完整的执行状态。
- **pgdir**: 存储页表根节点的物理地址。进程切换时，内核需将此地址加载到 **satp** 寄存器以切换地址空间。

- **kstack**: 指向分配给该线程的**内核栈**（ucore 中为 2 个物理页）。内核栈用于内核态运行，并在发生特权级改变时保存信息。

为了管理所有进程，ucore 还维护了几个全局变量：

- **current**: 指向当前正在占用 CPU 的进程。
- **initproc**: 指向第一个内核线程（后续实验中将指向第一个用户进程）。
- **hash_list** 和 **proc_list**: 分别用于哈希表和双向链表，组织所有的进程控制块。

进程上下文指的是进程在某一时刻的运行现场。在 ucore 中，它通过 `struct context` 保存。我们不需要保存所有寄存器，而是巧妙地利用了 C 语言的函数调用约定：因为进程切换发生在函数调用中，编译器会自动保存**调用者保存**的寄存器；我们只需要在 `context` 中手动保存**被调用者保存**的寄存器（如 `s0-s11`）即可。

2.3.4 创建并执行内核线程

当 **alloc_proc** 函数分配了一个新的进程控制块后，就可以创建线程了。

内核线程的创建相对简单，它通常只是内核中的一个函数。最关键的是，内核线程**不需要建立各自的页表**。

这是因为 ucore 在启动时已经通过 `boot_pgdir` 建立了一个完整的**内核虚拟空间**。所有的内核线程都**共享**这个内核虚拟空间，因此它们都可以访问整个物理内存。从这个角度看，所有内核线程都被 ucore 内核这个“大进程”所统一管理。

创建第 0 个内核线程 idleproc **idleproc** 是一个特殊的**空闲进程**，它的主要任务是在系统没有其他可运行任务时占用 CPU，从而**统一进程调度**。

proc_init 函数（在 `kern_init` 中被调用）负责创建这个第 0 号线程。它首先将**当前的执行上下文**（即 ucore 的启动流）“包装”成第一个内核线程。

具体步骤如下：

- **初始化进程链表**：调用 **list_init** 初始化 `proc_list`。
- **分配进程控制块**：调用 **alloc_proc** 为当前上下文分配一个 `proc_struct` 结构。
- **初步初始化**：将 `proc_struct` 清零，并设置关键字段：
  - `state = PROC_UNINIT`: 进程处于“初始”态。
  - `pid = -1`: 进程 ID 尚未分配。
  - `pgdir = boot_pgdir`: **共享内核页表**。这再次说明所有内核线程共享同一个内核地址空间。
- **最终初始化**：将这个新进程设置为 **idleproc**，并赋予其合法身份：
  - `pid = 0`: 确立其为 0 号进程。
  - `state = PROC_RUNNABLE`: 进程准备就绪，可以被调度。

- `kstack = (uintptr_t)bootstack`: **使用 ucore 的启动栈**作为其内核栈。其他线程后续需要单独分配内核栈。
- `need_resched = 1`: **设置立即调度标志**。这非常重要，它使得 **idleproc** 在执行 **cpu_idle** 函数时，会马上调用 **schedule** 函数，让出 CPU 给其他更有意义的任务。
- `set_proc_name(idleproc, "idle")`: 命名为"idle"。

`idleproc` 只是“继承”了 ucore 的启动流，接下来我们将创建第一个真正的、执行具体任务的内核线程。

创建第 1 个内核线程 initproc `idleproc` 线程在完成初始化后，就通过 **cpu_idle** 进入空闲循环。因此，ucore 需要创建第一个真正的**工作线程 initproc**，它在本实验中的任务是输出字符串。

进程创建在操作系统中通常通过“复制”父进程（**do_fork**）来完成。但对于第一个进程，没有父进程可以复制，因此我们必须**手动构造**一个“模板”**中断帧**（trapframe）。

这个创建过程由 **kernel_thread** 函数发起，它负责构造一个临时的中断帧，然后调用 **do_fork** 来创建新线程。

Listing 4: kernel_thread 函数 (proc.c)

```
int kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));

    // 1. 设置要执行的函数 (s0) 和参数 (s1)
    tf.gpr.s0 = (uintptr_t)fn;
    tf.gpr.s1 = (uintptr_t)arg;

    // 2. 设置sstatus寄存器:
    // SSTATUS_SPP: 设置为 Supervisor (内核) 模式
    // SSTATUS_SPIE: 允许中断
    // ~SSTATUS_SIE: 暂时禁用中断
    tf.status = (read_csr(sstatus) | SSTATUS_SPP | SSTATUS_SPIE) & ~
        SSTATUS_SIE;

    // 3. 设置入口点 (epc) 为 kernel_thread_entry
    // 当 `trapentry.S` 恢复现场时, pc将指向这里
    tf.epc = (uintptr_t)kernel_thread_entry;

    // 4. 调用 do_fork 创建新进程
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

do_fork 函数是创建线程的核心，它主要完成以下工作：

1. **分配进程控制块**: 调用 **alloc_proc**。
2. **分配内核栈**: 调用 **setup_stack**。

3. **复制内存管理**: 调用 **copy_mm**。对于内核线程，此项为 NULL，因为它们共享内核地址空间。

4. **设置中断帧和上下文**: 调用 **copy_thread**。

5. **加入进程链表**：将新进程加入 proc_list。

6. **设为就绪态**：设置 state = PROC_RUNNABLE。

7. **返回 PID**。

其中，**copy_thread** 函数负责将 kernel_thread 中构造的**模板中断帧**复制到新线程的**内核栈顶**，并设置好新线程的**上下文**。

Listing 5: copy_thread 函数 (proc.c)

```
static void copy_thread(struct proc_struct *proc, uintptr_t esp, struct
    trapframe *tf) {
    // 1. 在新内核栈的顶部为中断帧分配空间
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE - sizeof(
        struct trapframe));
    // 2. 复制 kernel_thread 传来的模板中断帧
    *(proc->tf) = *tf;

    // 3. 设置子进程的返回值为 0（存储在 a0 寄存器）
    proc->tf->gpr.a0 = 0;
    proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;

    // 4. 设置上下文：
    // ra（返回地址）指向 forkret，这是所有新线程的统一出口
    proc->context.ra = (uintptr_t)forkret;
    // sp（栈顶）指向刚设置好的中断帧
    proc->context.sp = (uintptr_t)(proc->tf);
}
```

通过 **copy_thread** 的设置，当新线程被调度时，它会从 **forkret** 函数开始执行，其栈顶则指向已配置好的中断帧，从而为后续的 **kernel_thread_entry** 做好准备。

调度并执行内核线程 initproc　**proc_init** 执行完毕后，系统中存在 idleproc 和 initproc 两个线程。当前的执行现场就是 idleproc。当内核初始化完成并调用 **cpu_idle** 函数时，idleproc 将主动让出 CPU。

在 **cpu_idle** 的循环中，它会检查 current->need_resched 标志。由于 idleproc 在创建时此标志被设为 1，它会立即调用 **schedule** 函数触发调度。

调度器 (schedule 函数**)** 是并发执行的基础。在实验四中，这是一个简单的**先进先出**调度器，其逻辑如下：

1. 将当前进程（current）的 need_resched 置为 0。

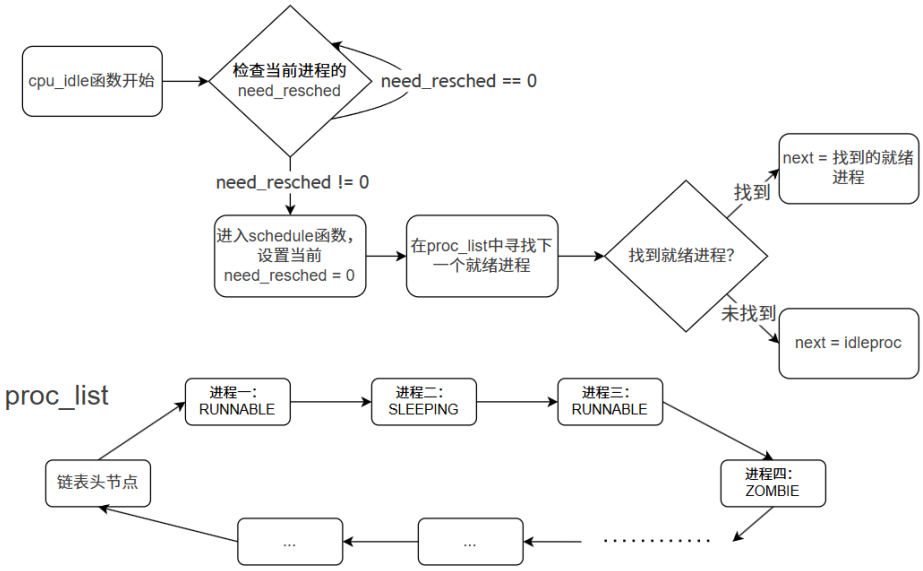


图 2: schedule 函数调度逻辑示意图

2. 从 `proc_list` 链表中查找**下一个**处于 **PROC_RUNNABLE**（就绪态）的进程 `next`。

3. 如果找不到，则 `next` 默认为 `idleproc`。

4. 找到后，调用 **proc_run** 函数执行进程切换。

由于 `proc_list` 中只有 `idleproc`（当前）和 `initproc`（就绪态），调度器会选中 `initproc`。

proc_run 函数会调用 **switch_to** 函数（位于 `kern/process/switch.S`）来完成真正的上下文切换。此函数接收两个参数：‘from‘（`a0` 寄存器）和 ‘to‘（`a1` 寄存器）。

Listing 6: switch_to 汇编 (switch.S)

```
# void switch_to(struct context* from, struct context* to)
.globl switch_to
switch_to:
    # 1. 保存 'from' 进程的被调用者保存寄存器
    sd ra, 0*8(a0)
    sd sp, 1*8(a0)
    sd s0, 2*8(a0)
    sd s1, 3*8(a0)
    sd s2, 4*8(a0)
    sd s3, 5*8(a0)
    sd s4, 6*8(a0)
    sd s5, 7*8(a0)
    sd s6, 8*8(a0)
    sd s7, 9*8(a0)
    sd s8, 10*8(a0)
    sd s9, 11*8(a0)
    sd s10, 12*8(a0)
```

```
    sd s11, 13*8(a0)

    # 2. 恢复 'to' 进程的被调用者保存寄存器
    ld ra, 0*8(a1)
    ld sp, 1*8(a1)
    ld s0, 2*8(a1)
    # ... (省略 s1 到 s10 的加载) ...
    ld s11, 13*8(a1)

    # 3. 返回 (ra 寄存器中现在是 'to' 进程的 ra)
    ret
```

切换后的执行流：**switch_to** 函数切换完成后，CPU 并不会返回到 schedule 函数，而是返回到新进程（initproc）的 ra 寄存器所指向的地址。

回顾 **copy_thread** 函数,我们设置了新进程的 context.ra 指向 **forkret** 函数,并且 context.sp 指向其**中断帧**（proc->tf）。

Listing 7: forkrets 和 kernel_thread_entry (trapentry.S)

```
    .globl forkrets
forkrets:
    # a0 此时是 switch_to 的第二个参数 a1 (即新进程的 context)
    # 但 copy_thread 中设置了 context.sp = proc->tf
    # switch_to 中 ld sp, 1*8(a1) 已经将 a0 (即 context.sp) 恢复到 sp
    # 因此 sp 现在指向中断帧 (proc->tf)

    # 在 RISC-V 中, forkret 的功能被简化
    # 它在 C 代码中被调用 (见 proc.c/forkret),
    # 并且 sp 已经在 switch_to 中被正确设置
    # forkret 会调用 trap_ret
    j __trapret  (*@\textit{... C 代码调用 \texttt{trap\_ret}}@*)

.globl kernel_thread_entry
kernel_thread_entry:
    # __trapret 恢复现场后, epc 指向这里
    # 此时 s0 和 s1 是从中断帧恢复的

    # 1. 将参数 (s1) 移动到 a0
    mv a0, s1
    # 2. 跳转到函数 (s0)
    jalr s0

    # 3. 函数返回后, 调用 do_exit 退出
    jal do_exit
```

switch_to 恢复 sp 和 ra 后，‘ret‘ 指令将跳转到 **forkret**。在 ‘forkret‘（或其后的 ‘__trapret‘）

中，系统会从栈上（即 proc->tf 的位置）**恢复所有寄存器**。

因为 **kernel_thread** 设置了 tf.epc 指向 **kernel_thread_entry**，并设置了 s0（函数指针）和 s1（参数），所以 ‘__trapret‘ 最终会跳转到 **kernel_thread_entry**。此函数再跳转到 s0 中存储的函数（即 init_main），从而成功启动第一个内核线程。

3 实验练习解答

3.1 练习 1：分配并初始化一个进程控制块

3.1.1 问题重现

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。

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？（提示通过看代码和编程调试可以判断出来）

3.1.2 实现

观察 kern/process/proc.h 文件，文件中定义了进程管理信息的结构体 proc_struct，定义如下：

Listing 8: proc_struct 结构体定义

```
struct proc_struct {
    enum proc_state state;                      // Process state
    int pid;                                    // Process ID
    int runs;                                   // the running times of Proces
    uintptr_t kstack;                           // Process kernel stack
    volatile bool need_resched;                 // bool value: need to be
       rescheduled to release CPU?
    struct proc_struct *parent;                 // the parent process
    struct mm_struct *mm;                       // Process's memory management
       field
    struct context context;                     // Switch here to run process
    struct trapframe *tf;                       // Trap frame for current
       interrupt
    uintptr_t pgdir;                            // the base addr of Page
       Directroy Table(PDT)
    uint32_t flags;                             // Process flag
    char name[PROC_NAME_LEN + 1];               // Process name
    list_entry_t list_link;                     // Process link list
    list_entry_t hash_link;                     // Process hash list
};
```

根据进程结构体的定义，alloc_proc 函数具体实现就是要初始化一个进程块，对定义的各成员变量赋初值，增加的代码如下：

Listing 9: proc_struct 结构体定义

```
static struct proc_struct *
alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        // LAB4:EXERCISE1 2313825
        proc->state = PROC_UNINIT;  //初始化进程状态为未初始化
        proc->pid = -1;             //初始化进程id为-1表示尚未分配有效id
        proc->runs = 0;             //运行次数为0
        proc->kstack = 0;           //内核栈指针初始化为0
        proc->need_resched = 0;     //不需要重新调度
        proc->parent = NULL;        //父进程指针为空
        proc->mm = NULL;            //内存管理结果为空
        memset(&(proc->context), 0, sizeof(struct context));    //上下文结
            构清零
        proc->tf = NULL;            //陷阱帧指针为空
        proc->pgdir = boot_pgdir_pa;          //页目录基址
        proc->flags = 0;            //进程标志为0
        memset(proc->name, 0, PROC_NAME_LEN+1);     //进程名初始化为空串
    }
    return proc;
}
```

proc_struct 结构体的 struct context context 成员变量保存的是进程的上下文信息，也就是在进程切换时需要保存和恢复的寄存器状态，具体在 kern/process/proc.h 文件中的定义如下：

Listing 10: proc_struct 结构体定义

```
struct context
{
    uintptr_t ra;
    uintptr_t sp;
    uintptr_t s0;
    uintptr_t s1;
    uintptr_t s2;
    uintptr_t s3;
    uintptr_t s4;
    uintptr_t s5;
    uintptr_t s6;
    uintptr_t s7;
    uintptr_t s8;
    uintptr_t s9;
    uintptr_t s10;
    uintptr_t s11;
```

```
};
```

而 struct trapframe *tf 成员变量是中断/异常发生时需要保存到的信息，在 lab3 中 trap.h 文件中的定义如下：

Listing 11: proc_struct 结构体定义

```
struct pushregs
{
    uintptr_t zero; // Hard-wired zero
    uintptr_t ra;   // Return address
    uintptr_t sp;   // Stack pointer
    uintptr_t gp;   // Global pointer
    uintptr_t tp;   // Thread pointer
    uintptr_t t0;   // Temporary
    uintptr_t t1;   // Temporary
    uintptr_t t2;   // Temporary
    uintptr_t s0;   // Saved register/frame pointer
    uintptr_t s1;   // Saved register
    uintptr_t a0;   // Function argument/return value
    uintptr_t a1;   // Function argument/return value
    uintptr_t a2;   // Function argument
    uintptr_t a3;   // Function argument
    uintptr_t a4;   // Function argument
    uintptr_t a5;   // Function argument
    uintptr_t a6;   // Function argument
    uintptr_t a7;   // Function argument
    uintptr_t s2;   // Saved register
    uintptr_t s3;   // Saved register
    uintptr_t s4;   // Saved register
    uintptr_t s5;   // Saved register
    uintptr_t s6;   // Saved register
    uintptr_t s7;   // Saved register
    uintptr_t s8;   // Saved register
    uintptr_t s9;   // Saved register
    uintptr_t s10;  // Saved register
    uintptr_t s11;  // Saved register
    uintptr_t t3;   // Temporary
    uintptr_t t4;   // Temporary
    uintptr_t t5;   // Temporary
    uintptr_t t6;   // Temporary
};

struct trapframe
{
    struct pushregs gpr;
```

```
40      uintptr_t status;
41      uintptr_t epc;
42      uintptr_t badvaddr;
43      uintptr_t cause;
44  };
```

3.2 练习 2：为新创建的内核线程分配资源

3.2.1 问题重现

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。do_kernel 函数会调用 alloc_proc 函数来分配并初始化一个进程控制块，但 alloc_proc 只是找到了一小块内存用以记录进程的必要信息，并没有实际分配这些资源。

ucore 一般通过 do_fork 实际创建新的内核线程。do_fork 的作用是，创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同。因此，我们实际需要"fork"的东西就是 stack 和 trapframe。在这个过程中，需要给新内核线程分配资源，并且复制原进程的状态。你需要完成在 kern/process/proc.c 中的 do_fork 函数中的处理过程。它的大致执行步骤包括：

- 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

3.2.2 实现

do_fork 函数是创建新进程（内核线程）的核心。在 `kern/process/proc.c` 中，我们根据练习要求，补全了 **do_fork** 函数的资源分配和初始化逻辑。

该函数严格按照注释提示的 7 个步骤执行，并加入了必要的**错误处理**（goto 语句），确保在任何一步分配失败时，都能正确回滚已分配的资源（如释放已分配的 `proc_struct` 或内核栈）。

Listing 12: do_fork 函数实现 (proc.c)

```
1  int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
2      int ret = -E_NO_MEM;
3      struct proc_struct *proc;
4
5      // 1. 调用 alloc_proc 分配进程控制块
```

```
6       if ((proc = alloc_proc()) == NULL) {
7           goto fork_out;
8       }
9
10      proc->parent = current;
11
12      // 2. 调用 setup_kstack 分配内核栈
13      if ((ret = setup_kstack(proc)) != 0) {
14          goto bad_fork_cleanup_proc;
15      }
16
17      // 3. 调用 copy_mm 复制或共享内存管理
18      if ((ret = copy_mm(clone_flags, proc)) != 0) {
19          goto bad_fork_cleanup_kstack;
20      }
21
22      // 4. 调用 copy_thread 设置中断帧和上下文
23      copy_thread(proc, stack, tf);
24
25      // 5. 将进程插入哈希表和进程列表 (需要关闭中断)
26      bool intr_flag;
27      local_intr_save(intr_flag);
28      {
29          proc->pid = get_pid(); // 获取唯一PID
30          hash_proc(proc);       // 加入哈希表
31          list_add(&proc_list, &(proc->list_link)); // 加入进程链表
32          nr_process++;
33      }
34      local_intr_restore(intr_flag);
35
36      // 6. 调用 wakeup_proc 唤醒新进程
37      wakeup_proc(proc);
38
39      // 7. 返回新进程的 PID
40      ret = proc->pid;
41
42  fork_out:
43      return ret;
44
45  // 错误处理回滚
46  bad_fork_cleanup_kstack:
47      put_kstack(proc);
48  bad_fork_cleanup_proc:
49      free_proc(proc);
50      goto fork_out;
```

```
51 }
```

唤醒函数 (wakeup_proc) 调整：为了配合 **do_fork** 的逻辑，我们对 wakeup_proc 函数进行了修改。原 sched.c 中的版本包含一个断言 assert(proc->state != PROC_UNINIT)，这会阻止刚 **alloc_proc**（状态为 PROC_UNINIT）的进程被唤醒。

因此，我们**删除了 sched.c 中的旧版本**，并在 proc.c 中添加了新实现，允许将处于 PROC_UNINIT 状态的进程直接唤醒为 PROC_RUNNABLE 状态。

Listing 13: wakeup_proc 新实现 (proc.c)

```
1 // wakeup_proc - 唤醒进程，将其状态置为 RUNNABLE
2 void wakeup_proc(struct proc_struct *proc)
3 {
4     if (proc->state == PROC_UNINIT || proc->state == PROC_SLEEPING)
5     {
6         proc->state = PROC_RUNNABLE;
7     }
8 }
```

3.2.3 问题回答

问题：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。

回答：是，ucore 通过 **get_pid** 函数和**关闭中断**的机制，确保了每个新创建的线程都拥有唯一的 ID。

分析与理由：

1. **PID 分配函数**: 在 do_fork 函数中，系统通过调用 **get_pid()** 函数来获取一个新的进程 ID。虽然我们没有分析 get_pid 的内部实现，但从函数命名和上下文来看，它的核心职责就是（通过递增全局计数器或循环查找空闲 ID 的方式）返回一个当前未被使用的、唯一的 PID。

2. **原子操作**：PID 的分配和进程的注册（加入 hash_list 和 proc_list）是在一个**关闭中断的临界区**内完成的（通过 local_intr_save 和 local_intr_restore 实现）。

3. **防止竞争**：这种**原子操作**至关重要。它能有效防止**竞争条件**（Race Condition）的发生。如果允许多核 CPU 或中断同时执行 do_fork，两个线程可能会在同一时刻调用 get_pid 并获取到相同的 ID。通过关闭中断，ucore（在单核环境下）确保了“获取 ID、加入哈希表、加入进程链表”这一系列操作是**不可分割**的，从而保证了 PID 的唯一性。

4. **哈希表支持**：hash_proc(proc) 函数将进程加入哈希表，这通常用于根据 PID 快速查找进程。这也间接支持了 PID 的管理和唯一性检查。

3.3 练习 3：编写 proc_run 函数

3.3.1 问题重现

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，如果相同则不需要切换。
- 禁用中断。你可以使用/kern/sync/sync.h 中定义好的宏 local_intr_save(x) 和 local_intr_restore(x) 来实现关、开中断。
- 切换当前进程为要运行的进程。
- 切换页表，以便使用新进程的地址空间。/libs/riscv.h 中提供了 lsatp(unsigned int pgdir) 函数，可实现修改 SATP 寄存器值的功能。
- 实现上下文切换。/kern/process 中已经预先编写好了 switch.S，其中定义了 switch_to() 函数。可实现两个进程的 context 切换。
- 允许中断。

请回答如下问题：
在本实验的执行过程中，创建且运行了几个内核线程?

3.3.2 实现

实现功能 .

首先明确我们需要实现的目标是什么，总结来看，需要通过补齐 proc_run 函数的设计，把指定的进程 proc 切换为当前运行的进程，即让 proc 在 CPU 上运行。它实现了进程的上下文切换，确保 CPU 从当前进程切换到目标进程。而具体的设计步骤问题中已经给出，按照这些目标逐步进行设计即可。

设计思路 .

- 检查是否需要切换：

  如果目标进程 proc 已经是当前进程 current，则无需切换，直接返回。如果不是当前进程，则需要进行上下文切换。这用一个条件判断就可以简单实现；

- 保存当前进程状态：

  使用 struct proc_struct *prev = current; 保存当前进程的指针，以便在切换后能够恢复。

- 关闭中断：

  为了确保上下文切换过程不会被中断打断，需要禁用中断。kern/sync/sync.h 中提供了 local_intr_save(x) 和 local_intr_restore(x) 来实现关、开中断：

Listing 14: local_intr_save/restore 开关中断函数

```
1    static inline bool __intr_save(void) {
2        if (read_csr(sstatus) & SSTATUS_SIE) {
3            intr_disable();
4            return 1;
5        }
```

```
6           return 0;
7       }//中断保存与关闭
8
9       static inline void __intr_restore(bool flag) {
10          if (flag) {
11              intr_enable();
12          }
13      }//中断恢复
14
15      #define local_intr_save(x) \
16          do {                   \
17              x = __intr_save(); \
18          } while (0)
19      #define local_intr_restore(x) __intr_restore(x);
```

这段代码定义了

__intr_save：中断保存与关闭。通过读取 sstatus 寄存器的 SSTATUS_SIE 位，检查当前中断是否启用。如果中断启用，则调用 intr_disable() 关闭中断，并返回 1 表示中断之前是启用的。如果中断未启用，则直接返回 0。

__intr_restore：中断恢复。根据传入的标志 flag 决定是否恢复中断。如果 flag 为 1，则调用 intr_enable() 恢复中断。如果 flag 为 0，则保持中断关闭状态。

宏定义 local_intr_save: 调用 __intr_save() 函数，将当前中断状态保存到变量 x 中，并关闭中断。

宏定义 local_intr_restore：调用 __intr_restore(x) 函数，根据变量 x 的值恢复中断状态。

[这段设计中用到的 intr_disable() 和 intr_enable() 来自 kern/driver/intr.c，前者使用 set_csr 指令将 sstatus 寄存器的 SSTATUS_SIE 位设置为 1 从而开启中断，后者使用 clear_csr 指令将将 sstatus 寄存器的 SSTATUS_SIE 位清除为 0，从而关闭中断]

所以，就可以通过调用 local_intr_save(intr_flag) 关闭中断，并保存中断状态到 intr_flag，以确保上下文切换过程不会被中断打断。

- 切换进程：

为了切换当前进程为目标进程，可以首先更新 current 指针为目标进程 proc，然后调用 lsatp(proc->pgdir) 设置目标进程的页表基地址 PDT，随后调用 switch_to(&(prev->context), &(proc->context)) 完成上下文切换，将 CPU 的执行状态从当前进程切换到目标进程。

其中，lsatp 定义于 libs/riscv.h，用于修改 SATP 寄存器的值以切换页表基地址；proc->pgdir 为内核线程指向 boot_pgdir，在 proc_init 中设定；switch_to 函数实现于 kern/process/switch.S，是一个用汇编语言实现的上下文切换函数，用于在两个进程或线程之间切换 CPU 的执行上下文。它的设计基于保存当前进程的寄存器状态并恢复目标进程的寄存器状态：

1. 保存当前进程的寄存器状态：将当前进程的寄存器值保存到 from 指针指向的内存区域。from 是一个指向当前进程控制块（struct proc_struct）的指针，寄存器状态会存储在该结构

的上下文字段中。

2. 恢复目标进程的寄存器状态：从 to 指针指向的内存区域加载目标进程的寄存器值。to 是一个指向目标进程控制块的指针，寄存器状态会从该结构的上下文字段中恢复。

- 恢复中断：

切换完成之后，调用 local_intr_restore(intr_flag) 恢复之前保存的中断状态。

宏定义 local_intr_restore 在前面已经说明，他定义在 kern/sync/sync.h 中。

实现代码 .

完整实现代码如下：

Listing 15: proc_run 完整实现

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current) //如果目标进程 proc 不是当前进程 current
    {
        bool intr_flag;
        struct proc_struct *prev = current; //保存当前进程的指针
        local_intr_save(intr_flag); //关闭中断，并保存中断状态到 intr_flag
            ，以确保上下文切换过程不会被中断打断
        {
            current = proc; //更新 current 指针为目标进程 proc
            lsatp(proc->pgdir); //调用 lsatp 修改 SATP 寄存器的值以切换页表
                基地址
            switch_to(&(prev->context), &(proc->context)); //上下文切换，保
                存prev寄存器状态，加载proc寄存器状态
        }
        local_intr_restore(intr_flag); //恢复之前保存的中断状态
    }
}
```

3.3.3 问题回答

本实验执行过程中创建且运行了几个内核线程?

本实验流程中只创建并运行了两个内核线程——idleproc（pid 0，占位调度）和 initproc（pid 1，实际工作线程）。

具体来看，proc_init 将当前启动流包装为 idleproc（pid=0，状态 PROC_RUNNABLE，栈为 bootstack，need_resched=1），随后 kernel_thread(init_main, "Hello world!!", 0) 通过 do_fork/copy_thread 复制模板 trapframe，find_proc 确认 pid=1，命名为 "init"，最终创建 initproc。

而整个运行路径就是，cpu_idle 检测 need_resched 后调用 schedule()；调度器在 proc_list 中按 FIFO/RR 查找唯一就绪的非 idle 线程 initproc，调用 proc_run 完成一次切换；init_main 打印 hello world 后返回。期间未再创建新线程。

3.4 扩展练习 Challenge

3.4.1 问题重现

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

3.4.2 开关中断的实现机制与 do...while(0) 的作用

在 proc_run 等函数中,我们使用 local_intr_save(intr_flag); 和 local_intr_restore(intr_flag); 来创建**临界区**，以保护 current 指针切换等关键操作不被中断打断。

开关中断的实现：这些宏定义在 kern/sync/sync.h 和 kern/driver/intr.c 中，其核心是围绕 **sstatus** 寄存器的 **SIE** 位进行操作。

Listing 16: 开关中断相关宏定义 (sync.h)

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

#define local_intr_restore(x) __intr_restore(x);

static inline bool __intr_save(void) {
    if (read_csr(sstatus) & SSTATUS_SIE) {
        // 1. 检查SIE位，如果中断是开启的
        intr_disable(); // 2. 关闭中断
        return 1;       // 3. 返回 1 (true)，表示之前是开启的
    }
    return 0; // 中断本就是关闭的，返回 0 (false)
}

static inline void __intr_restore(bool flag) {
    if (flag) {
        // 1. 仅当 flag = 1 (即之前是开启) 时
        intr_enable(); // 2. 才重新开启中断
    }
    // 否则 (flag = 0)，保持中断关闭状态
}
```

Listing 17: 中断使能函数 (intr.c)

```
/* intr_enable - 开启中断 */
void intr_enable(void) {
    // 将SSTATUS寄存器的SIE位设置为1
    set_csr(sstatus, SSTATUS_SIE);
}

```

```
/* intr_disable - 关闭中断 */
void intr_disable(void) {
    // 将SSTATUS寄存器的SIE位清除为0
    clear_csr(sstatus, SSTATUS_SIE);
}
```

执行流程分析：

1. `local_intr_save(intr_flag)`：
   - 调用 `__intr_save` 函数。
   - `__intr_save` 读取 `sstatus` 寄存器，**检查 SIE 位**。
   - 如果 SIE 位为 1（中断开启），则调用 `intr_disable`（清除 SIE 位）来**关闭中断**，并返回 **true**。
   - 如果 SIE 位为 0（中断本就关闭），则**直接返回 false**。
   - 无论哪种情况，CPU 在此之后都处于**中断关闭**状态。返回值（true 或 false）被保存在 `intr_flag` 变量中。
2. `local_intr_restore(intr_flag)`：
   - 调用 `__intr_restore` 函数，并传入之前保存的 `intr_flag`。
   - `__intr_restore` 检查 `intr_flag` 变量。
   - **如果 `intr_flag` 为 true**（意味着在进入临界区之前中断是开启的），则调用 `intr_enable`（设置 SIE 位）来**恢复中断**。
   - **如果 `intr_flag` 为 false**（意味着进入时中断就是关闭的），则**不做任何操作**，保持中断关闭状态。

这种“保存-关闭-恢复”的机制确保了临界区代码（如进程切换）的**原子性**，同时又不会破坏临界区**嵌套**时更外层的中断状态。

为什么使用 `do { ... } while (0)`？

这是 C 语言中编写**多语句宏**的一种标准技巧，主要目的是为了**保证宏在任何语法结构中都能表现得像一条单独的 C 语句**。

如果不使用 `do...while(0)`，我们可能会这样定义：`#define local_intr_save(x) { x = __intr_save(); }`

这在简单的使用中没有问题，但如果用在 `if...else` 结构中，就会导致语法错误：

Listing 18: do...while(0) 解决的语法问题

```
if (condition)
    local_intr_save(flag); // 展开为: if (condition) { flag = __intr_save()
        ; };
else
    // ...

```

```
// 宏展开后，分号 '};' 结束了 if 语句
// 导致 'else' 分支无所适从，编译失败!
// 错误: 'else' without a previous 'if'
```

通过使用 `do { ... } while (0)` 包装，宏展开后变为：`if (condition) do { flag = __intr_save(); } while (0);`

这个 `do...while(0)` 结构被 C 编译器视为一**条单独的语句**，并且末尾的分号是这条语句所必需的，这完美地解决了 `if...else` 语句中的语法问题。

3.5 扩展练习 Challenge

3.5.1 深入理解不同分页模式的工作原理 (思考题)

问题 1： `get_pte()` 函数（位于 `kern/mm/pmm.c`）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。

回答：这两段代码之所以高度相似，是因为它们在执行**完全相同的逻辑操作**：**“在多级页表树中向下遍历一级，并在必要时创建下一级页表”**。

代码分析： `get_pte` 函数在 Sv39 模式下工作，该模式使用三级页表（L2, L1, L0）。

- **第一段相似代码**：

```
pde_t *pdep1 = &pgdir[PDX1(la)]; // 1. 获取L2页表项
if (!(*pdep1 & PTE_V)) {        // 2. 检查是否有效
    struct Page *page;
    if (!create || (page = alloc_page()) == NULL) { // 3. 如果无效且
        create=true
        return NULL;
    }
    // 4. 分配、清零、并链接新的L1页表
    set_page_ref(page, 1);
    uintptr_t pa = page2pa(page);
    memset(KADDR(pa), 0, PGSIZE);
    *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
}
```

这段代码的作用是：根据 L2 索引（`PDX1`）查找 L2 页表项。如果该项无效（即指向 L1 的页表不存在），则**分配一个新页充当 L1 页表**，并将其物理地址填入 L2 页表项中。

- **第二段相似代码**：

```
pde_t *pdep0 = &((pte_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)]; // 1. 获
    取L1页表项
if (!(*pdep0 & PTE_V)) {        // 2. 检查是否有效
    struct Page *page;
    if (!create || (page = alloc_page()) == NULL) { // 3. 如果无效且
        create=true
```

```
5           return NULL;
6       }
7       // 4. 分配、清零、并链接新的L0页表
8       set_page_ref(page, 1);
9       uintptr_t pa = page2pa(page);
10      memset(KADDR(pa), 0, PGSIZE);
11      *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
12  }
```

这段代码的作用是：根据 L1 索引（PDX0）查找 L1 页表项。如果该项无效（即指向 L0 的页表不存在），则**分配一个新页充当 L0 页表**（即最终的 PTE 页），并将其物理地址填入 L1 页表项中。

与 Sv32, Sv39, Sv48 的联系： RISC-V 的 Sv32, Sv39, Sv48 分页模式均采用**多级页表**结构，它们的核心工作原理都是**递归地遍历一个页表树**。

- **Sv32**：使用 2 级页表 (L1, L0)。get_pte 函数将**只需要 1 段**这样的代码（即上述的 “第二段相似代码”），用于从 L1 遍历到 L0。
- **Sv39**：使用 3 级页表 (L2, L1, L0)。get_pte 函数需要**重复 2 次**这个逻辑（如上所示），一次从 L2 到 L1，一次从 L1 到 L0。
- **Sv48**：使用 4 级页表 (L3, L2, L1, L0)。get_pte 函数将需要**重复 3 次**这个逻辑，分别遍历 L3、L2 和 L1。

因此，这两段代码如此相像，是因为它们是**多级页表遍历算法的标准步骤**。这种代码结构（或其递归形式）是实现所有 Sv-X 分页模式的基础，区别仅仅在于这个 “遍历-创建” 的步骤需要**重复的次数**不同。

问题 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？

回答： 这种将 “查找” 和 “创建” 功能合一的写法（通过 create 标志来控制）是一种常见的实现方式，但它有利有弊。

优点（合并写法）：

1. **高效性（针对创建）**：在需要创建映射时（如 page_insert），这种方式效率最高。它**只需要遍历一次页表树**。如果在遍历过程中发现某一级的页表不存在，它可以**立即分配**并继续向下遍历，而不需要返回错误、分配、再重新从头遍历。
2. **便捷性**：对于上层调用者（如 page_insert）来说，接口非常便捷。它不需要关心页表是否已存在，只需调用 get_pte(..., 1) 就能确保获得一个可用的 PTE 地址。

缺点（合并写法）：

1. **违反单一职责原则**：函数命名为 get_pte（获取 PTE），但其内部却可能执行**分配内存**和**修改页表**的写操作。这种 “get” 函数却有副作用的设计，会降低代码的可读性和可维护性，容易误导开发者。

2. **职责混淆**：当用于**纯查找**时（如 get_page 调用 get_pte(..., 0)），函数内部依然保留着 create 为 true 的分支逻辑，使得代码更复杂。

3. **错误处理复杂**：如果 create 为 true 时内存分配失败，函数返回 NULL。调用者需要区分“查找不到返回 NULL”和“创建失败返回 NULL”这两种情况。

是否有必要拆分？有必要，拆分后架构更清晰。
一种更清晰的设计是将其拆分为两个函数：

1. **find_pte(pgdir, la)**：**纯查找**函数。
   - **职责**：只读。严格遍历页表，如果中间任何一级页表或 PTE 不存在，立即返回 NULL。
   - **优点**：没有副作用，逻辑简单，适用于所有只读操作（如 get_page）。
2. **alloc_pte(pgdir, la)**：**查找或创建**函数。
   - **职责**：可写。遍历页表，如果发现中间页表不存在，则**立即分配**并填充。最终保证返回一个有效的 PTE 地址。
   - **优点**：封装了所有创建逻辑，供 page_insert 等写操作调用。

结论：虽然当前的合并写法在性能上（针对创建）有优势，但从**软件工程、代码可读性和可维护性**的角度来看，将其拆分为 **find_pte**（只读查找）和 **alloc_pte**（查找并创建）两个职责单一的函数，是更好、更健壮。

4 实验所遇问题

4.1 问题：alloc_proc 中 pgdir 初始化错误

问题描述： 在实现练习一的 alloc_proc 函数时，我们负责初始化新分配的 proc_struct 结构体。在最初的实现中，我们将其中的页目录基址成员变量初始化为：

```
proc->pgdir = 0;
```

错误表现： 这个初始化导致 make grade 测试失败。失败发生在内核尝试进行进程切换时。具体来说，当 proc_run 函数被调用以切换到新创建的进程(如 initproc)时，它会执行 lsatp(proc->pgdir) 命令。由于 proc->pgdir 为 0，这会导致 satp 寄存器被设置为一个无效的（空）页表基地址，使得 CPU 无法进行地址转换，从而立即引发异常（Page Fault）或系统崩溃。

分析与解决过程： 我们通过分析 proc_init 和 proc_run 的代码，认识到了 pgdir 的真正作用。它必须指向一个**有效的页表根节点的物理地址**。

在本实验中，我们创建的 idleproc 和 initproc 都是**内核线程**。根据实验文档，所有内核线程都运行在内核态，并且**共享同一个内核地址空间**。这个共享的内核地址空间是由内核启动时创建的 boot_pgdir（其物理地址存储在 boot_pgdir_pa）来管理的。

因此，任何新创建的内核线程都应该使用这个已经存在的内核页表。

解决方案： 我们将 alloc_proc 函数中的初始化代码修正为：

```
proc->pgdir = boot_pgdir_pa;  // 页目录基址指向内核页表的物理地址
```

结果： 修正后，新创建的 idleproc 和 initproc 的 pgdir 成员都正确地指向了 boot_pgdir_pa。当 proc_run 执行上下文切换时，lsatp 会加载一个有效的页表基址，使得进程切换和后续执行得以成功。