lab4:进程管理

薄照轩 石爽 马思远

练习1：分配并初始化一个进程控制块（需要编码）

> 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

> 请在实验报告中简要说明你的设计实现过程。请回答如下问题： 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）

实现过程

1. 代码

kern/process/proc.c文件

```
alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        // 初始化进程状态为未初始化
        proc->state = PROC_UNINIT;
        // 初始化pid为-1（未分配）
        proc->pid = -1;
        // 运行次数初始化为0
        proc->runs = 0;
        // 内核栈地址初始化为0（后续分配）
        proc->kstack = 0;
        // 初始不需要调度
        proc->need_resched = 0;
        // 父进程指针为空
        proc->parent = NULL;
        // 内存管理结构为空（内核线程无需独立mm）
        proc->mm = NULL;
        // 上下文寄存器全部清零
        memset(&proc->context, 0, sizeof(struct context));
        // 中断帧指针为空
        proc->tf = NULL;
        // 页目录表基址复用内核页表（内核线程共享）
        proc->pgdir = boot_pgdir_pa;
        // 进程标志位初始化为0
        proc->flags = 0;
        // 进程名称清零
        memset(proc->name, 0, sizeof(proc->name));
```

```
        // 初始化链表节点（进程链表和哈希表）
        list_init(&proc->list_link);
        list_init(&proc->hash_link);
    }
    return proc;
}
```

2. 代码实现

- 进程状态设为 PROC_UNINIT，表示 PCB 未完全初始化；
- pid 设为 -1，标记未分配唯一进程 ID；
- 运行次数 runs 初始化为 0，统计进程后续执行次数；
- 内核栈地址 kstack 设为 0，后续通过 setup_kstack 分配实际栈空间；
- 调度标记 need_resched 设为 0，初始无需主动放弃 CPU；
- 父进程指针 parent、内存管理结构 mm、中断帧指针 tf 均设为 NULL，内核线程无需独立用户内存空间；
- 上下文 context 通过 memset 清零，确保寄存器状态无残留；
- 页目录表基址 pgdir 复用内核页表物理地址 boot_pgdir_pa，内核线程共享内核地址空间；
- 进程标志位 flags 设为 0，无特殊状态标记；
- 进程名称 name 清零，后续通过 set_proc_name 赋值；
- 链表节点 list_link 和 hash_link 初始化，用于将 PCB 接入全局进程链表和哈希表，方便管理。

3. 回答问题

struct context context

- 含义：存储进程的执行上下文，本质是一组关键寄存器的值集合（包括 ra、sp、s0~s11 等被调用者保存寄存器），用于描述进程在某一时刻的 CPU 执行状态。
- 作用：核心用于进程切换。当调度器选择新进程执行时，通过 switch_to 函数将当前进程的 context 保存到其 PCB 中，同时从新进程的 context 中恢复寄存器状态，使得新进程能够从上次被中断的位置继续执行。

struct trapframe *tf

- 含义：指向中断帧的指针，中断帧是进程发生中断（如系统调用、异常）时，由硬件或内核保存的完整执行现场，包含了进程在中断发生前的所有通用寄存器（如 a0~a7、t0~t6）、程序计数器（epc）、状态寄存器（status）等关键信息。
- 作用：用于内核线程的创建和执行初始化。在 do_fork 函数中，copy_thread 会将父进程的中断帧复制到新进程的内核栈顶部，并通过 tf 指针记录该中断帧的位置。新进程被调度执行时，forkret 函数会通过 tf 恢复所有寄存器状态，最终跳转到 epc 指向的入口函数（如 kernel_thread_entry），完成内核线程的启动。此外，tf 也为后续中断处理和系统调用返回提供了寄存器状态保存与恢复的载体

练习2：为新创建的内核线程分配资源（需要编码）

> 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

> 请在实验报告中简要说明你的设计实现过程。请回答如下问题： 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。

实现过程

1. 代码

kern/process/proc.c文件

```
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. 分配并初始化进程控制块
    if ((proc = alloc_proc()) == NULL)
    {
        goto fork_out;
    }

    // 2. 为子进程分配内核栈（KSTACKPAGE为栈页数，默认2页）
    if (setup_kstack(proc) != 0)
    {
        goto bad_fork_cleanup_proc;
    }

    // 3. 复制/共享内存管理结构（内核线程共享内核地址空间，该函数为空实现）
    if (copy_mm(clone_flags, proc) != 0)
    {
        goto bad_fork_cleanup_kstack;
    }

    // 4. 设置子进程的中断帧和上下文（关键：继承父进程执行状态）
    copy_thread(proc, stack, tf);

    // 5. 分配唯一pid
    proc->pid = get_pid();
    // 设置父进程为当前进程
    proc->parent = current;
```

```
    // 6. 将新进程加入全局进程链表和哈希表
    list_add(&proc_list, &proc->list_link);
    hash_proc(proc);
    // 进程计数加1
    nr_process++;

    // 7. 唤醒新进程，设置为就绪态
    wakeup_proc(proc);

    // 8. 返回子进程pid作为成功结果
    ret = proc->pid;

fork_out:
    return ret;

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

2. 代码实现

前置检查：进程数量限制

首先判断当前系统进程总数 `nr_process` 是否达到上限 `MAX_PROCESS`：

- 若已达上限，返回错误码 `-E_NO_FREE_PROC`，终止创建流程；
- 若未达上限，初始化返回值 `ret` 为 `-E_NO_MEM`（默认内存分配失败状态），进入后续资源分配流程。

资源分配：从 PCB 到内核栈

（1）分配并初始化进程控制块（PCB）

调用 `alloc_proc` 函数申请一块内存作为新进程的 PCB：

- 若 `alloc_proc` 返回 NULL（内存分配失败），直接跳转到 `fork_out` 返回错误；
- 若申请成功，获得新进程的 `struct proc_struct` 指针 `proc`，此时 PCB 已完成基础初始化（如状态为 `PROC_UNINIT`、pid 为 -1 等）。

（2）分配内核栈

调用 `setup_kstack(proc)` 为新内核线程分配独立内核栈：

- `setup_kstack` 通过 `alloc_pages(KSTACKPAGE)` 申请 `KSTACKPAGE` 个连续物理页（实验中默认 2 页，共 8KB），并将物理地址转换为内核虚拟地址，赋值给 `proc->kstack`；
- 若栈分配失败（返回非 0），跳转到 `bad_fork_cleanup_proc`，通过 `kfree(proc)` 释放已分配的 PCB，避免内存泄漏。

（3）内存管理信息处理（内核线程特殊处理）

调用 `copy_mm(clone_flags, proc)` 处理内存管理结构：

- 由于本实验创建的是内核线程，所有内核线程共享内核地址空间，无需独立的用户内存管理结构（`current->mm` 为 NULL），因此 `copy_mm` 内部仅通过 `assert` 验证状态，无实际复制操作，直接返回 0；
- 若后续扩展为用户进程，此函数会根据 `clone_flags` 决定是复制父进程页表（`CLONE_VM` 不置位）还是共享页表（`CLONE_VM` 置位）。

状态复制：上下文与中断帧

调用 `copy_thread(proc, stack, tf)` 复制父进程的执行状态，为新线程启动做准备：

- **中断帧（trapframe）复制**：在新进程内核栈顶部（`proc->kstack + KSTACKSIZE - sizeof(struct trapframe)`）分配中断帧空间，将父进程的中断帧 `tf` 完整复制到该位置，并更新 `proc->tf` 指向此中断帧；
- **关键寄存器调整**：将新进程中断帧的 `a0` 寄存器设为 0（标记为子进程，区别于父进程的返回值），`sp` 寄存器设为中断帧地址（确保中断返回时栈指针正确）；
- **上下文（context）初始化**：将 `proc->context.ra` 设为 `forkret` 函数地址（新进程首次调度时的入口），`proc->context.sp` 设为中断帧地址（上下文切换时栈指针起点），确保新进程能正确恢复执行状态。

进程管理：接入全局列表与激活

（1）分配唯一 PID

调用 `get_pid()` 函数为新进程分配唯一进程 ID：

- `get_pid` 通过遍历全局进程链表 `proc_list` 和哈希表 `hash_list`，确保返回的 pid 未被其他进程占用（范围 1~`MAX_PID-1`，0 预留为 idle 进程 pid）；
- 将分配到的 pid 赋值给 `proc->pid`，标记新进程的唯一身份。

（2）设置父进程与接入全局列表

- **父进程关联**：将 `proc->parent` 设为当前进程 `current`，建立进程间的父子关系，便于后续资源回收（如父进程等待子进程退出）；
- **链表接入**：通过 `list_add(&proc_list, &proc->list_link)` 将新进程的 PCB 加入全局进程链表 `proc_list`，便于调度器遍历所有进程；通过 `hash_proc(proc)` 将 PCB 加入哈希表 `hash_list`，便于通过 pid 快速查找进程。

（3）更新进程计数与激活进程

- 进程总数 `nr_process` 加 1，反映系统当前进程数量；
- 调用 `wakeup_proc(proc)` 将新进程状态从 `PROC_UNINIT` 改为 `PROC_RUNNABLE`，标记为 “就绪态”，使其能被调度器选中执行。

结果返回：返回子进程 PID

将 `ret` 赋值为新进程的 pid（`proc->pid`），跳转到 `fork_out` 返回，此时父进程会获得子进程的 pid，新进程则进入就绪队列等待调度。

错误处理：资源回滚

若某一步资源分配失败（如内核栈分配失败），通过标签跳转实现资源回滚：

- `bad_fork_cleanup_kstack`：释放已分配的内核栈（`put_kstack(proc)`），再跳转到 `bad_fork_cleanup_proc` 释放 PCB；
- `bad_fork_cleanup_proc`：释放 PCB（`kfree(proc)`），确保未完成创建的进程不占用内存资源。

3. 回答问题

ucore 能为每个新 fork 线程分配唯一 ID get_pid 是实现 pid 唯一性的核心函数，其设计思路是 “遍历检查 + 安全区间优化”

- 静态变量 last_pid 记录上一次分配的 pid，初始值为 MAX_PID（pid 最大上限，实验中通常设为 32768）；
- 每次调用 get_pid 时，last_pid 先自增 1：若超过 MAX_PID，则重置为 1（pid 从 1 开始，0 预留为 idle 进程），避免 pid 超出有效范围。

  若 last_pid 进入 “安全区间”（last_pid >= next_safe，next_safe 记录下一个可能无冲突的 pid），则遍历全局进程链表 proc_list get_pid 函数的遍历检查逻辑确保不会分配重复 pid，proc_init 中通过 assert(initproc->pid == 1) 验证新进程 pid 正确性，进一步证明唯一性。

练习3：编写proc_run 函数（需要编码）

> 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?

实现过程

1. 代码

kern/process/proc.c文件

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
       // 保存当前中断状态并禁用中断（保证切换原子性）
        bool intr_flag;
        local_intr_save(intr_flag);
```

```
        // 更新当前运行进程指针
        struct proc_struct *prev = current;
        current = proc;

        // 切换页表：加载新进程的页目录表到satp寄存器
        lsatp(proc->pgdir);

        // 上下文切换：保存prev的上下文，恢复proc的上下文
        switch_to(&prev->context, &proc->context);

        // 恢复中断状态
        local_intr_restore(intr_flag);
    }
}
```

2. 回答问题

- 本实验中，系统创建并运行了 2 个内核线程：idleproc（PID=0）和 initproc（PID=1），并通过 proc_run 完成了它们之间的切换。 1.idleproc （pid=0）： 通过 alloc_proc 创建，作为调度的“空转线程”，初始 need_resched=1 以驱动后续调度。 2.initproc （pid=1）： 由 kernel_thread(init_main, ...) 创建 => 内核线程。其入口 epc=kernel_thread_entry，最终执行 init_main。

扩展练习Challenge

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

这对宏定义在 kern/sync/sync.h 中，依赖 RISC-V 汇编指令实现

1.local_intr_save(intr_flag)

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

// 被调用的函数
static inline bool __intr_save(void) {
    // 1. 读取 sstatus 寄存器，并检查 SIE (Supervisor Interrupt Enable) 位
    if (read_csr(sstatus) & SSTATUS_SIE) {
        // 2. 如果 SIE 位为 1，说明当前中断是开启的
        intr_disable(); // 3. 调用 intr_disable() 函数来禁用中断
        return 1;       // 4. 返回 1，表示“我已经关闭了一个原本开启的中断”
    }
    // 5. 如果 SIE 位为 0，说明中断本来就是关闭的
    return 0;           // 6. 返回 0，表示“中断本来就是关闭的，我什么也没做”
}
```

工作流程：

1. 当你调用 `local_intr_save(intr_flag)` 时，它会执行 `intr_flag = __intr_save();`。
2. `__intr_save` 函数首先通过 `read_csr(sstatus)` 读取 `sstatus` 寄存器的值。
3. 它检查 `SSTATUS_SIE` 位（这是中断使能位）。
4. **如果该位为 1**：意味着当前系统是允许中断的。于是它调用 `intr_disable()` 来关闭中断，并返回 `1`。这个 `1` 被赋值给了 `intr_flag`。
5. **如果该位为 0**：意味着系统已经处于禁止中断的状态。此时它什么也不用做，直接返回 `0`。这个 `0` 被赋值给了 `intr_flag`。

最终，`intr_flag` 变量保存了一个“令牌”：

- 如果 `intr_flag` 是 `1`，说明是这个宏把中断关掉的。
- 如果 `intr_flag` 是 `0`，说明中断在这个宏调用之前就已经是关着的。

2.local_intr_restore(intr_flag)

```
// 宏定义
#define local_intr_restore(x) __intr_restore(x);

// 被调用的函数
static inline void __intr_restore(bool flag) {
    // 1. 检查传入的 flag
    if (flag) {
        // 2. 如果 flag 为 true (非0)，则调用 intr_enable() 开启中断
        intr_enable();
    }
    // 3. 如果 flag 为 false (0)，则什么也不做
}
```

工作流程：

1. 当你调用 `local_intr_restore(intr_flag)` 时，它会执行 `__intr_restore(intr_flag)`。
2. `__intr_restore` 函数检查传入的 `flag`（也就是之前保存的 `intr_flag`）。
3. **如果 `flag` 是 `1`**：这表示“之前是我把中断关掉的”。所以现在需要调用 `intr_enable()` 把中断重新打开，恢复到进入临界区之前的状态。
4. **如果 `flag` 是 `0`**：这表示“中断本来就是关着的”。所以现在什么也不用做，保持中断关闭状态即可。

> 深入理解不同分页模式的工作原理（思考题）

- get_pte() 是一个“一站式”服务，它不仅能查找PTE，还能在需要时创建所需的页表结构。
- RISC-V 的 SV32、SV39、SV48 是三种不同的分页模式，主要区别在于虚拟地址空间大小和页表的级数 核心共同点： 页大小：都支持 4KB 标准页大小（也支持更大的巨页，但核心逻辑一致）。 页表项格式：页表项（PTE）的核心格式是统一的，都包含物理页框号（PPN）、有效位（V）以及一系列权限位（R/W/X/U）。 查找流程：尽管级数不同，但查找过程的逻辑是递归和相似的。都是将虚拟地址按固定位数分割成若干个索引，从最高级页表开始，依次向下级页表查找，直到找到最终的 PTE。 get_pte() 函数的设计巧妙地利用了上述共同点，使其具有很强的通用性。你所观察到的“两段形式类似的代码”，正是对应了页表查找过程中的两个层级。

- 这两段代码之所以相像，是因为它们执行的是**完全相同的逻辑操作**，只是操作的对象是页表层级中的**不同级别**。`get_pte()` 函数通过重复应用这一 “检查 - 分配 - 映射” 的逻辑，就能灵活地处理 2 级（SV32）、3 级（SV39）甚至更多级（SV48）的页表结构。这种设计避免了为每一种分页模式都编写一套独立的、冗余的查找代码，体现了优秀的代码复用思想。

- get_pte() 将 “查找页表项” 和 “创建缺失的页表” 这两个功能合并在一个函数里,这种设计有利有弊： 利：简洁高效并且封装性好 弊：缺乏灵活性

- 我认为还是拆开比较好，将 “查找” 和 “分配” 这两个职责分离开，提供更纯粹、更灵活的接口。 我们可以创建一个内部函数，比如 _get_pte，它只负责查找，不做任何分配。

基于 `_get_pte`，我们可以提供两个对外的、功能明确的函数。一个负责纯粹的查找功能。一个负责查找并在必要时创建。