Lab4：进程管理

练习 1：分配并初始化一个进程控制块

目标与思路

在内核创建线程之前，先通过 alloc_proc 分配并“最小初始化”一个进程控制块（PCB，struct proc_struct），不做任何资源分配（例如内核栈、地址空间复制、trapframe 构造）。这样可以保证 PCB 处于可预测的未就绪状态，便于后续步骤（setup_kstack/copy_mm/copy_thread/插入队列/唤醒）顺利进行。

本实验以 proc_init 中的自检为准，初始化后的 idleproc 必须满足检查条件，说明 alloc_proc 的默认值正确。

初始化内容

alloc_proc 在 kmalloc 成功后，仅设置以下“基本字段”值：

```
state = PROC_UNINIT;                                // 初始态，尚未就绪
pid = -1;                                           // 未分配真实 pid，后续由 get_pid 赋值
runs = 0;                                           // 运行计数清零
kstack = 0;                                         // 尚未分配内核栈，后续 setup_kstack
need_resched = 0;                                   // 默认不请求调度
parent = NULL;                                      // 父子关系在 fork 时建立
mm = NULL;                                          // 地址空间复制/共享在 copy_mm 处理
memset(&proc->context, 0, sizeof(struct context)); // context 清零
tf = NULL;                                          // 等待分配内核栈后由 copy_thread 建立
pgdir = boot_pgdir_pa;                              // 本工程使用物理地址，与 proc_init 的
检查一致
flags = 0;                                          // 初始无标志
memset(proc->name, 0, sizeof(proc->name));     // name 清零，后续 set_proc_name 设
置
```

> 链表指针（list_link/hash_link）由插入全局队列或哈希表时处理，alloc_proc 不需要改动。
>
> 资源分配与入口设置（内核栈、trapframe、context.ra/sp）由 setup_kstack 与 copy_thread 完成。

问题：

1. struct context context

   > 用于保存调度切换需要的少量寄存器（RISC-V 的 callee-saved 寄存器，含 ra/sp/s0~s11）。
   >
   > 作用：在 switch_to(&from->context, &to->context) 时保存/恢复这组寄存器，实现线程之间的上下文切换。新线程首次运行前，copy_thread 会设置 context.ra=forkret、context.sp=trapframe，使切换后先进入 forkret。

2. struct trapframe *tf

   > 它是一次 trap/中断/异常现场的完整寄存器快照，包含所有 14 个通用寄存器、status、epc 等。

> 作用：新线程创建时，copy_thread 将模板 trapframe 拷到"线程内核栈顶附近"，并令 proc->tf 指向它。首次调度通过 forkret(current->tf) 设置 sp 并跳到 __trapret，按统一的 trap 返回路径恢复 tf，最终 sret 到 epc=kernel_thread_entry，再进入线程入口函数（如 init_main）。

- context 用于"线程与线程之间"的调度切换（轻量寄存器集）。
- trapframe 用于"陷入与返回"的完整现场恢复（通用返回路径）。

  一个新线程要首次运行时，它的运行路径是：switch_to → forkret → forkrets(tf) → __trapret（恢复 tf）→ sret 到 kernel_thread_entry → 线程函数。

练习2：为新创建的内核线程分配资源

目标与思路

本练习目标是完善 `do_fork`，实现内核线程创建时的资源分配和状态复制。具体包括：分配进程控制块、内核栈、复制上下文和 trapframe、维护进程父子关系、分配唯一进程号、插入进程队列并唤醒新线程。这样可以保证新线程能被调度运行，并正确维护进程树结构。

本练习我们的目标是：实现 `do_fork` 函数，为新建的内核线程分配必要的资源，包括进程控制块、内核栈、内存管理信息等，并完成进程的初始化和调度准备。

设计思路和实现流程如下：

1. 调用 `alloc_proc` 分配进程控制块
2. 调用 `setup_kstack` 分配内核栈空间
3. 调用 `copy_mm` 复制或共享内存管理信息
4. 调用 `copy_thread` 设置陷阱帧和执行上下文
5. 将新进程加入进程管理数据结构
6. 唤醒新进程使其进入就绪状态
7. 返回新进程的PID

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;
    // 检查进程数是否已达上限
    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. 设置父进程指针
    proc->parent = current;

    // 3. 维护父子链表关系
    proc->cptr = current->cptr;
    if (proc->cptr != NULL) {
```

```
        proc->cptr->optr = proc;
    }
    current->cptr = proc;
    proc->optr = NULL;
    proc->yptr = NULL;

    // 4. 分配内核栈
    if ((ret = setup_kstack(proc)) != 0)
    {
        goto bad_fork_cleanup_proc;
    }

    // 5. 复制或共享内存管理信息（本实验为内核线程，通常不处理 mm）
    if ((ret = copy_mm(clone_flags, proc)) != 0)
    {
        goto bad_fork_cleanup_kstack;
    }

    // 6. 复制trapframe和上下文
    copy_thread(proc, stack, tf);

    // 7. 分配唯一进程号
    proc->pid = get_pid();

    // 8. 插入哈希表和进程链表
    hash_proc(proc);
    list_add(&proc_list, &(proc->list_link));

    // 9. 增加进程计数
    nr_process++;

    // 10. 唤醒新进程
    wakeup_proc(proc);

    // 11. 返回新进程号
    ret = proc->pid;
fork_out:
    return ret;

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

关键点说明

- **进程关系维护**：通过设置`cptr`、`optr`、`yptr`指针维护进程的父子兄弟关系，这是进程树管理的基础。
- **进程状态转换**：通过`wakeup_proc`将进程状态从PROC_UNINIT变为PROC_RUNNABLE，使其可被调度。
- **内核栈设置**：`setup_kstack`分配KSTACKPAGE大小的内核栈空间，为进程提供内核态执行环境。

- **执行上下文初始化**：`copy_thread`设置陷阱帧和上下文，其中：
  1、设置`a0`寄存器为0，标识这是子进程
  2、设置返回地址为`forkret`，确保首次调度时正确进入
- **错误处理机制**：使用goto语句实现资源的层级清理，保证在任何步骤失败时都能正确释放已分配的资源。
- **资源分配顺序**：严格按照"进程控制块→内核栈→内存管理→线程上下文"的顺序分配资源，确保前序资源分配失败时能正确回滚。

问题回答

问：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：`uCore`能够为每个新fork的线程分配唯一的PID

uCore通过`get_pid`函数为每个新fork的线程分配唯一的进程号（pid）。`get_pid`会遍历所有已存在的进程，确保分配的 pid 不与当前系统中的任何进程重复。每次分配时，都会查重并跳过已被占用的 pid，最终保证每个新创建的进程（线程）都拥有唯一的 id。因此，uCore能够做到为每个新fork的线程分配唯一的 id，保证系统中所有进程（线程）的 pid 互不冲突。

1、`PID`分配机制：`uCore`通过`get_pid()`函数实现PID分配，该函数维护两个静态变量：

- `last_pid`：记录上次分配的PID
- `next_safe`：记录下一个安全的PID上限

2、唯一性保证算法：

- 首先尝试递增`last_pid`
- 如果`last_pid`达到或超过`next_safe`，则重新扫描进程列表
- 扫描过程中，如果发现PID冲突就递增`last_pid`，同时更新`next_safe`为大于`last_pid`的最小已用PID
- 通过这种"安全区间"机制确保分配的PID唯一

3、实现细节：

```
static int get_pid(void) {
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    // ... 扫描进程列表，确保PID唯一性
    return last_pid;
}
```

4、**边界处理**：当PID达到MAX_PID时回绕到1重新开始，通过静态断言确保MAX_PID > MAX_PROCESS，避免PID耗尽。

5、**并发安全**：目前我们的实现未显式处理并发（之后的实验LAB应该会实现），但在单核环境下通过进程调度的串行性隐含保证了PID分配的安全性。

总而言之言而总之，ucore的PID分配机制能够有效地为每个新创建的线程分配唯一的进程标识符。

练习 3：编写 proc_run 函数

目标与思路

`proc_run`用于将指定的进程切换到 CPU 上运行，实现进程的上下文切换。其核心流程包括：

1. 检查要切换的进程是否与当前正在运行的进程相同，如果相同则无需切换。
2. 禁用中断，保证切换过程的原子性（使用 `local_intr_save(x)` 和 `local_intr_restore(x)`）。
3. 切换当前进程指针 `current` 为目标进程。
4. 切换页表，使用 `lsatp(proc->pgdir)` 修改 SATP 寄存器，切换到新进程的地址空间。
5. 调用 `switch_to(&prev->context, &proc->context)` 实现上下文切换。
6. 允许中断，恢复系统响应能力。

代码实现

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
        bool intr_flag;
        local_intr_save(intr_flag);                 // 1. 关中断
        struct proc_struct *prev = current;
        current = proc;                             // 2. 切换当前进程
        lsatp(proc->pgdir);                         // 3. 切换页表
        switch_to(&(prev->context), &(proc->context)); // 4. 上下文切换
        local_intr_restore(intr_flag);              // 5. 开中断
    }
}
```

关键点说明

- `local_intr_save(x)` 和 `local_intr_restore(x)` 用于关/开中断，防止切换过程中被打断。
- `lsatp(pgdir)` 切换 SATP 寄存器，实现地址空间切换。
- `switch_to` 汇编实现，保存/恢复 RISC-V 的 callee-saved 寄存器，实现进程上下文切换。

问题回答

在本实验的执行过程中，创建且运行了几个内核线程?

答：**共创建并运行了 2 个内核线程**，分别是：

1. **idleproc（pid=0，内核空闲线程）**
    - 在 `proc_init()` 中创建
    - 通过 `alloc_proc()` 分配进程控制块
    - 设置 `pid=0`，状态为 `PROC_RUNNABLE`
    - 使用内核启动栈 `bootstack`
    - 在 `cpu_idle()` 中循环调用 `schedule()` 进行进程调度
2. **initproc（pid=1，初始化线程）**
    - 在 `proc_init()` 中通过 `kernel_thread(init_main, "Hello world!!", 0)` 创建
    - 运行 `init_main` 函数，打印初始化信息
    - 执行完毕后调用 `do_exit()` 退出

这两个线程都通过 `proc_run` 完成了切换到 CPU 上的运行。从运行输出可以验证：

```
alloc_proc() correct!                          // 练习1检查通过
this initproc, pid = 1, name = "init"         // initproc 运行了 init_main
To U: "Hello world!!".                         // 传递的参数被正确接收
To U: "en.., Bye, Bye. :)"                    // init_main 执行完毕
kernel panic at kern/process/proc.c:400:      // do_exit 还未实现
    process exit!!.
```

Challenge 1：中断开关机制的实现

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?

答： `local_intr_save` 和 `local_intr_restore` 通过操作 RISC-V 处理器的 sstatus 寄存器中的 SIE（Supervisor Interrupt Enable）位来实现中断的开关。

在 kern/sync/sync.h 中定义了这两个宏：

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

`__intr_save()` 函数首先读取当前的 sstatus 寄存器状态，检查 SSTATUS_SIE 位是否被设置（表示中断当前是否被启用）。如果 SIE 位为 1（中断已启用），则调用 `intr_disable()` 关闭中断，并返回 1 来记录之前的状态；如果 SIE 位为 0（中断已禁用），则直接返回 0。这样做的目的是保存当前的中断状态：

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

而在 kern/driver/intr.c 中，`intr_disable()` 通过清除 sstatus 寄存器中的 SSTATUS_SIE 位来禁用中断：

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

对应地，`intr_enable()` 则设置 SSTATUS_SIE 位来启用中断：

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

而 `__intr_restore()` 函数接收之前保存的中断状态标志。如果标志为 1，说明之前中断是启用的，则调用 `intr_enable()` 恢复中断；如果标志为 0，说明之前中断已被禁用，则什么都不做，保持禁用状态：

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

这样的设计使得 `local_intr_save` 和 `local_intr_restore` 构成了一对嵌套安全的中断保护机制。在临界区代码执行前调用 `local_intr_save(intr_flag)` 保存当前中断状态并关闭中断，保证临界区内的操作不会被中断打断；临界区执行完毕后调用 `local_intr_restore(intr_flag)` 恢复之前的中断状态。这样即使在嵌套调用的情况下也能正确处理：如果外层调用前中断是禁用的，即使内层打开了中断，外层的 `local_intr_restore` 也会再次关闭它。这种机制广泛应用在 pmm.c 的 `alloc_pages`、`free_pages`、`nr_free_pages` 等函数中，以及 proc.c 的 `proc_run` 函数中，保证内存管理和进程调度的原子性。

Challenge 2：深入理解分页机制与 get_pte 函数设计

Challenge：深入理解不同分页模式的工作原理和 get_pte 函数设计

第一部分：不同分页模式的工作原理分析

答：RISC-V 体系结构支持多种分页模式，包括 sv32、sv39 和 sv48。这些模式的共同目标都是将虚拟地址映射到物理地址，但采用的页表级数和寻址范围不同。sv32 适用于 32 位系统，使用两级页表；sv39 和 sv48 适用于 64 位系统，分别使用三级和四级页表。本实验中的 uCore 采用 sv39 模式，这是当前 RISC-V 系统的主流选择。

在 sv39 中，64 位虚拟地址被分为若干部分：高 25 位用于符号扩展，接下来的 39 位被分为三个 9 位的索引字段（分别对应第 1、第 0 级和页内偏移），以及最后 12 位的页内偏移。虚拟地址到物理地址的转换过程是一个多级页表的查询过程：首先使用虚拟地址的高位索引在第 1 级页表（也称为页目录表）中查找条目，该条目指向下一级页表的物理地址；然后使用虚拟地址的中间位索引在第 0 级页表中继续查找，最后得到物理页面的地址，与虚拟地址的页内偏移部分组合得到最终的物理地址。

在 kern/mm/pmm.c 的 `get_pte()` 函数中，这个多级查询过程被清晰地实现了。第一段代码使用 PDX1(la) 在第 1 级页表（页目录表）中查找对应条目，检查该条目的有效位 PTE_V；如果无效且 create 为真，则分配一个新的第 0 级页表页面，设置其有效位并将其物理地址存储在第 1 级页表条目中：

```
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);
}
```

第二段代码的逻辑与第一段完全相同，只是操作的是下一级页表。它使用 PDX0(la) 在第 0 级页表中查找最终的页表项，检查该条目的有效位；如果无效且 create 为真，则分配一个新的物理页面，并将其物理地址存储在第 0 级页表条目中：

```
pde_t *pdep0 = &((pde_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);
}
```

这两段代码之所以相似，正是因为 sv32、sv39、sv48 等模式都采用了相同的设计思想：多级页表的每一级都遵循相同的查询逻辑——检查当前级别的页表项的有效位（PTE_V），如果有效则继续向下查询，否则按需分配新的下一级页表。无论是两级、三级还是四级页表，这个逻辑都是一致的，只是级数不同而已。这体现了分页机制设计的优雅之处：通过统一的多级索引方案，支持不同大小的虚拟地址空间，而核心的页表遍历算法保持不变。

第二部分：get_pte() 函数设计评价

关于将查找和分配合并在一个函数中的设计，这种写法各有利弊，需要根据具体的系统设计哲学来权衡。

从优点来看，将查找和分配合并在 `get_pte()` 中确实提高了代码的便利性和简洁性。以 `page_insert()` 为例，当我们需要为某个虚拟地址建立映射关系时，只需调用一次 `get_pte(pgdir, la, 1)`，就能自动完成整个多级页表的导航过程，包括按需创建中间级的页表。这使得调用者的代码更加简洁：

```
pte_t *ptep = get_pte(pgdir, la, 1);
if (ptep == NULL) {
    return -E_NO_MEM;
}
```

如果要实现"纯查询"功能，`get_pte()` 也通过引入 `create` 参数优雅地支持了这种需求。在 `page_remove()` 中，我们调用 `get_pte(pgdir, la, 0)` 来查询页表项，如果页表不存在就返回 NULL，而不会触发任何分配逻辑，这正是我们只想查询而不想创建的需求。

从代码可读性和维护性角度看，`get_pte()` 的单一职责原则被某种程度上违反了。函数同时承担了"查找"和"分配"两项职责，增加了代码的复杂度。当出现页表相关的问题时，需要理解整个多级分配流程才能诊断。此外，如果后续需要修改分配策略或实现更复杂的页表管理（如支持大页面、NUMA感知的分配等），就需要修改 `get_pte()` 函数本身，这会影响所有依赖它的代码。

综合来看，**在当前的 uCore 实验环境中，这种合并的设计是合理的**，理由如下：首先，分配策略相对固定且明确，不太可能频繁变更；其次，通过 `create` 参数已经实现了有效的功能分化，满足了不同场景的需求；再次，代码的便利性带来的益处在相对简单的系统中更为显著。

然而，**如果系统演化到更复杂的阶段，就可能有必要拆分**。例如：如果需要支持多种内存分配策略（如优先级分配、NUMA感知分配等），就应该抽取分配逻辑独立出来；如果为了提高代码的可测试性，需要独立测试查找逻辑和分配逻辑，也应该拆分；如果需要更细粒度的控制，比如查询页表而不触发任何副作用，就需要分离出一个纯查询函数。具体的拆分方案可以考虑设计一个 `get_pte_lookup()` 函数用于纯查询（不分配任何页表），一个 `get_pte_or_alloc()` 函数用于查询或分配，这样可以让代码的意图更明确，模块化更好。但对于当前的实验需求，现有的统一设计已经足够高效和优雅了。