操作系统Lab4

一、实验介绍

这个实验的核心是给单执行流内核加“并发能力”和“虚拟地址空间管理”，让内核能同时跑多个任务，还为后续功能打基础。

(1) 先补虚拟内存管理：给每个任务配“独立逻辑空间”

之前已经能实现虚拟地址到物理地址的映射，但没有统一管理每个任务的地址空间。这次要做的是：

1. 引入“虚拟内存描述结构”，专门记录每个执行实体（线程 / 进程）的虚拟地址布局。
2. 用“预映射”方式建页表，创建地址空间时一次性完成所有需要的映射，不用后续动态分配或置换页面。
3. 核心作用是给每个任务划分逻辑上的独立空间，为后续隔离用户进程做准备。

---

(2) 再引入内核线程：实现多任务并发

这是让系统从“单流”变“多流”的关键，核心是让多个任务轮流用 CPU：

1. 内核线程是特殊“进程”，只在内核态运行，还共享内核的地址空间和资源。
2. 要实现三个核心组件：
   - 线程控制块（记录线程状态、上下文等关键信息）；
   - 上下文切换（保存当前线程的 CPU 状态，加载下一个线程的状态，实现任务切换）；
   - 调度器（决定哪个线程先运行、运行多久，按规则分配 CPU 资源）。
3. 最终效果是多个内核线程“并发”执行，每个线程都像独占 CPU 一样。

（3）内核线程和用户进程的核心区别

比较项	内核线程	用户进程
运行模式	仅内核态	用户态、内核态切换
地址空间	共享内核地址空间	有独立的用户虚拟地址空间

（4）实验的核心价值

1. 系统层面：从只能跑一个任务的内核，变成能调度多个内核线程的并发内核。
2. 基础层面：搭好虚拟内存框架，后续能基于此实现用户进程、系统调用、缺页处理等关键功能。
3. ucore 特性：线程和进程统一管理，所有内核线程都属于 “ucore 内核本身” 这个唯一的内核进程。

二、文件变动

从文件变动来看，实验四的核心是新增**进程 / 线程管理**和**调度机制**，并完善虚拟内存管理以支撑多执行流，下面结合具体文件逐一解析：

（1）核心新增模块：进程 / 线程管理（`kern/process/`）

这是实验四最关键的新增内容，负责实现内核线程的创建、上下文管理和切换。

1. `proc.c` 和 `proc.h`
   - 定义**线程控制块（TCB）**结构（可能命名为 `struct proc_struct`），记录线程的状态（运行 / 就绪 / 阻塞）、上下文（寄存器值）、页表指针、栈地址等核心信息。
   - 实现线程创建（`proc_create` 或 `kernel_thread`）：分配 TCB、初始化栈和上下文、设置线程入口函数。
   - 处理线程退出（`proc_exit`）：释放资源、修改状态，由调度器回收。
   - 维护线程链表（如就绪队列、运行队列），供调度器选择下一个运行的线程。
2. `entry.S`
   - 实现内核线程入口函数 `kernel_thread_entry`：这是汇编级别的跳板，当线程首次被调度时，从这里跳转到线程的实际函数（如用户传入的 `func`），并在函数返回后处理线程退出。
3. `switch.S`
   - 核心中的核心：实现**上下文切换**的汇编逻辑（`switch_to` 函数）。
   - 功能：保存当前线程的 CPU 寄存器（如 `ra`、`sp`、`s0-s11` 等）到其栈或 TCB 中，再从下一个线程的栈或 TCB 中恢复寄存器，完成执行流的切换。
   - 注意：上下文切换需要关闭中断，避免切换过程中被打断导致状态混乱。

（2）调度机制（`kern/schedule/`）

实现多线程并发的 “调度器”，决定哪个线程获得 CPU 时间。

1. `sched.c` 和 `sched.h`
   - 采用**FIFO（先进先出）调度策略**：就绪队列中的线程按进入顺序依次执行，直到当前线程主动放弃 CPU（如阻塞）或时间片结束（实验四可能简化为主动切换）。

- 提供调度入口函数（如 `schedule`）：在时钟中断或线程主动让出 CPU 时被调用，从就绪队列中选一个线程，通过 `switch_to` 完成切换。
- 维护调度队列：可能用链表（`struct list_head`）管理就绪线程，`sched.c` 中实现队列的插入（`sched_enqueue`）、取出（`sched_dequeue`）等操作。

（3）虚拟内存管理完善（`kern/mm/`）

为多线程提供地址空间管理基础，虽然实验四不涉及用户进程隔离，但为后续做铺垫。

1. `vmm.c` 和 `vmm.h`
   - 新增**虚拟内存区域（VMA）**管理：定义 `struct mm_struct`（地址空间描述符）和 `struct vma_struct`（虚拟内存区域）。
   - `mm_struct` 代表一个线程 / 进程的完整虚拟地址空间，包含多个 `vma_struct`（如代码段、数据段、栈段等）。
   - 提供 VMA 操作：创建（`vma_create`）、查找（`find_vma`）、插入（`insert_vma`）等，用于管理虚拟地址的范围和属性（如可读 / 可写 / 可执行）。
2. `kmalloc.c` 和 `kmalloc.h`
   - 新增基于 slab 算法的内存分配器：相比实验三的简单页分配，`kmalloc` 可分配任意大小（而非整页）的内存，适合小对象（如 TCB、VMA 结构）的高效分配。
   - 实验中只需调用 `kmalloc` 和 `kfree` 即可，不用深究实现，但要注意其依赖物理内存管理（`pmm`）。
3. `pmm.c` 和 `pmm.h`
   - 完善页表操作：新增 `get_pte`（查找页表项）、`page_insert`（建立虚拟页到物理页的映射）、`page_remove`（删除映射）等函数，支持动态修改页表，为多线程的地址空间管理提供底层支持。

（4）初始化流程调整（`kern/init/init.c`）

实验四的内核初始化流程更复杂，最终切换到多线程运行：

1. 初始化步骤：
   - 先完成物理内存管理（`pmm_init`）、虚拟内存管理（`vmm_init`）、`kmalloc` 初始化。
   - 初始化进程系统（`proc_init`）：创建第一个内核线程（如 `idle` 线程，负责空闲时调度），再创建其他测试线程（如循环打印的线程）。
   - 最终通过调度器切换到 `idle` 线程，进入多线程并发状态。

（5）中断与陷阱支持（`kern/trap/trapentry.S`）

新增 `forkrets` 函数：配合线程创建（如 `do_fork`），在复制线程上下文后，从陷阱返回时正确恢复新线程的执行环境，确保新线程能正常启动。

（6）总结：实验四的核心目标

通过新增**进程管理**（TCB、线程创建）、**上下文切换**（`switch.S`）、**调度器**（FIFO 策略），让内核从“单执行流”升级为“多线程并发”；同时完善虚拟内存区域管理，为后续用户进程的独立地址空间打下基础。

简单说，实验四让 ucore 从 “一个人干活” 变成 “多个人轮流干活”，并给每个人划分了 “工作区域” 的管理规则。

三、实验执行流程

从内核初始化开始，先搞定内存和中断的基础准备，再通过进程管理和调度，让系统从单执行流变成多线程并发，最终验证功能。

简单拆解成一步步的逻辑：

1. **起点：内核总控函数 init**

   整个实验的所有操作，都是从 init 函数开始触发的，它是内核启动后的 “总入口”。

2. **第一步：物理内存初始化（pmm_init）**

   先建立空闲物理页的管理机制，比如记录哪些物理内存是可用的。这一步是基础，后面建页表、给线程分配栈，都需要从这里申请物理内存。

3. **第二步：中断 / 异常初始化（pic_init + idt_init）**

   和实验三的操作一样，初始化中断控制器（PIC）和中断描述符表（IDT）。目的是让 CPU 能识别和处理中断（比如后面调度需要的时钟中断）、异常，保证系统能正常响应外部事件。

4. **第三步：虚拟内存初始化（vmm_init）**

   建立内核的页表结构，把内核的虚拟地址和物理地址做 “静态映射”（就是一次性建好所有需要的映射）。这一步只保证虚拟内存能正常用，不处理缺页异常，也不搞页面换入换出。做完这步，系统就完成了 “内存虚拟化”，但还是单条执行流在跑。

5. **第四步：CPU 虚拟化（进程管理初始化 + 调度）**

   这是实现多线程并发的关键，核心是让多个线程分时用 CPU：

   - 先调用 proc_init 创建两个内核线程：idleproc（占位线程，没其他线程跑时就循环等待）和 initproc（第一个实际干活的线程，任务是输出 “Hello World”）；
   - 之后 idleproc 运行 cpu_idle ()，检测到需要调度时，调用 schedule () 选择可运行的线程（这里就是 initproc）；
   - 通过上下文切换，让 initproc 获得 CPU 执行权，最终输出 “Hello World”，验证线程创建和调度是否成功。

四、虚拟内存管理

（1）基本原理概述

1. **虚拟内存的本质**：是程序员 / CPU “看到” 的 “逻辑内存地址”，不是直接的物理内存地址。
   - 要么这个虚拟地址没有对应的实际物理内存；
   - 要么有对应，但两者地址不一样；
   - 靠操作系统的 “内存映射”（比如页表），自动把虚拟地址转换成物理地址，让 CPU 能找到实际数据。
2. **虚拟内存的核心作用（分两层）**：
   - 第一层：内存地址虚拟化（基础作用）。
     - 因为 CPU 访问的是虚拟地址，操作系统可以通过页表项限制访问范围，防止软件越界访问其他内存，实现 “内存访问保护”。
   - 第二层：优化内存使用（进阶作用）。

- 核心是“按需分页”：软件没实际访问某个虚拟地址时，不分配物理内存；只有真要访问了，才动态分配物理内存并建立映射。
- 还有“页换入换出”（当前 ucore 没实现）：把不常用的数据写到硬盘，腾出内存给常用数据，后续需要再从硬盘读回内存，相当于给程序员“扩容”了内存空间，能让更多程序同时运行。

简单说，虚拟内存就是给物理内存加了一层“中间层”—— 既保护了内存安全，又让内存用得更高效、“看起来更大”。

（2）页表项设计思路

这段代码是针对 RISC-V 架构下 Sv39 分页机制的**页表项和地址解析宏定义**，核心作用是把 64 位虚拟地址拆分成多级页表索引，以及定义页表项（PTE）的关键字段。理解这些宏是掌握多级页表地址转换的基础，下面逐部分解析：

（1）Sv39 虚拟地址与物理地址结构

RISC-V 的 Sv39 分页机制使用 39 位虚拟地址和 56 位物理地址，两者都按“多级索引 + 页内偏移”的结构划分，代码里的宏就是为了拆分这些结构。

1. 虚拟地址（39 位）拆分

虚拟地址被分成 4 个部分（总 39 位）：

```
+--------9位-------+--------9位-------+--------9位-------+---------12位--------
-+
|    VPN[2]         |    VPN[1]         |    VPN[0]         |       PGOFF
  |
|   （PDX1）        |   （PDX0）        |   （PTX）         |   （页内偏移）          |
+------------------+------------------+------------------+--------------------
--+
```

- **VPN（Virtual Page Number）**：虚拟页号，分 3 级（VPN [2]、VPN [1]、VPN [0]），每级 9 位，对应三级页表的索引。
- **PGOFF（Page Offset）**：页内偏移，12 位，对应 4KB（2^{12}=4096）页大小，用于定位页内具体字节。

代码中用宏来提取这几个部分：

- `PDX1(la)`：提取 VPN [2]（最高 9 位），对应一级页表（页目录）的索引。
- `PDX0(la)`：提取 VPN [1]（中间 9 位），对应二级页表（页目录）的索引。
- `PTX(la)`：提取 VPN [0]（较低 9 位），对应三级页表的索引。
- `PGOFF(la)`：提取页内偏移（最低 12 位）。

大大页

大页

页

2. 物理地址（56 位）与页表项（PTE）结构

物理地址被分成 4 个部分（总 56 位）：

```
+--------26位------+--------9位-------+--------9位-------+---------12位--------+
|    PPN[2]        |    PPN[1]        |    PPN[0]        |      PGOFF           |
|  （高阶物理页号） |  （中阶物理页号） |  （低阶物理页号） |  （页内偏移）         |
+------------------+------------------+------------------+----------------------+
```

- **PPN（Physical Page Number）**：物理页号，分 3 级（PPN [2]、PPN [1]、PPN [0]），合起来定位一个物理页。

页表项（PTE）用于存储虚拟页到物理页的映射，结构如下：

```
+--------26位------+--------9位-------+--------9位-------+----2位----+-----8位-----+
|    PPN[2]        |    PPN[1]        |    PPN[0]        |  保留位    | D|A|G|U|X|W|R|V |
+------------------+------------------+------------------+-----------+--------------+
```

- 高 26+9+9=44 位：存储物理页号（PPN [2:0]），对应物理地址的高 44 位（56-12=44）。
- 低 8 位：权限控制位（`V`、`R`、`W`、`X`、`U`等），后面详细说。

（2）核心宏解析

这些宏用于拆分地址、构建地址或提取页表项中的关键信息。

1. 地址拆分宏

- `PDX1(la)`：`((la) >> 30) & 0x1FF`
  - 右移 30 位（跳过低 30 位：PDX0 的 9 位 + PTX 的 9 位 + PGOFF 的 12 位），取低 9 位（0x1FF=511，刚好覆盖 9 位），得到一级页表索引。
- `PDX0(la)`：`((la) >> 21) & 0x1FF`
  - 右移 21 位（跳过 PTX 的 9 位 + PGOFF 的 12 位），取低 9 位，得到二级页表索引。
- `PTX(la)`：`((la) >> 12) & 0x1FF`
  - 右移 12 位（跳过 PGOFF 的 12 位），取低 9 位，得到三级页表索引。
- `PGOFF(la)`：`(la) & 0xFFF`
  - 取低 12 位（0xFFF=4095），得到页内偏移。
- `PPN(la)`：`(la) >> 12`
  - 右移 12 位，得到虚拟地址对应的“虚拟页号”（忽略页内偏移）。

2. 地址构建宏

- `PGADDR(d1, d0, t, o)`：`(d1 << 30) | (d0 << 21) | (t << 12) | o`
  - 用一级索引（d1）、二级索引（d0）、三级索引（t）和页内偏移（o）拼接出完整的虚拟地址。

3. 页表项地址提取宏

- `PTE_ADDR(pte)`：`((pte) & ~0x3FF) << (12 - 10)`

  作用是从页表项（pte）中提取物理页的起始地址。

  - `~0x3FF`：清除页表项的低 10 位（因为低 10 位包含权限位和保留位），保留高 54 位（其中 44 位是 PPN）。
  - 左移 `(12-10)=2` 位：将 PPN 转换为物理页起始地址（PPN 是页号，乘以页大小 4KB=2^{12}，等价于左移 12 位；而 PPN 在页表项中已经左移了 10 位，所以再补 2 位）。

（3）常量定义

- `PGSIZE = 4096`：页大小为 4KB（2^{12}）。
- `NPDEENTRY = 512`、`NPTEENTRY = 512`：每级页表有 512 个表项（因为每个索引是 9 位，2^9=512）。
- `PTSIZE = PGSIZE * NPTEENTRY = 4096*512 = 2MB`：一级页表项可映射的地址范围（因为一个页表项对应下一级页表，而下一级页表有 512 个项，每个映射 4KB，总 512*4KB=2MB）。
- 移位常量（`PGSHIFT=12`、`PDX0SHIFT=21`等）：对应各部分在地址中的偏移位（如 PGOFF 占低 12 位，所以 PTX 从第 12 位开始）。

（4）页表项权限位（低 8 位）

这些标志位控制对虚拟页的访问权限：

- `PTE_V = 0x001`：Valid（有效位），为 1 表示该页表项有效，CPU 可使用；为 0 则访问时触发缺页异常。
- `PTE_R = 0x002`：Read（可读），允许读取该页。
- `PTE_W = 0x004`：Write（可写），允许修改该页。
- `PTE_X = 0x008`：Execute（可执行），允许执行该页的代码。
- `PTE_U = 0x010`：User（用户态可访问），为 1 时用户态程序可访问；为 0 则只能内核态访问。

例如：一个页表项若为`PTE_V | PTE_R | PTE_X`，表示这是一个有效、可读、可执行的页（通常是代码段）。

（5）总结

这些宏和常量是操作 Sv39 多级页表的 “工具”：

1. 把虚拟地址拆分成三级索引和页内偏移，用于逐级查找页表。
2. 从页表项中提取物理页地址，完成虚拟地址到物理地址的转换。
3. 通过权限位控制内存访问，实现内存保护。

理解这些定义后，就能看懂后续代码中 “如何通过虚拟地址查页表”“如何设置页表项权限” 等核心操作了。

（3）使用多级页表实现虚拟存储

要想实现虚拟存储，我们需要把页表放在内存里，并且需要有办法修改页表，比如在页表里增加一个页面的映射或者删除某个页面的映射。

要想实现页面映射，我们最主要需要修改的是两个接口：

- `page_insert()`，在页表里建立一个映射
- `page_remove()`，在页表里删除一个映射

页表映射的增删操作、映射关系的验证，以及**精细化权限管理的必要性。**

（1）页表映射的“增”与"删"

`get_pte`：找到或创建页表项

作用：根据虚拟地址`la`，在多级页表中找到对应的页表项（PTE）地址；如果不存在且`create=1`，则自动分配各级页表页并创建页表项。

- **一级页表（PDX1）的处理**
  - 首先通过`PDX1(la)`获取虚拟地址在一级页表中的索引（9 位），找到对应的一级页表项`pdep1`（`pgdir[PDX1(la)]`）。
  - 检查`pdep1`是否有效（`*pdep1 & PTE_V`）：
    - 如果无效（页表不存在）：
      - 若`create=0`（不允许创建），直接返回`NULL`。
      - 若`create=1`，则调用`alloc_page()`分配一个物理页作为二级页表，并用`memset`清零（初始化页表项）。
      - 将这个物理页的页号（PPN）写入`pdep1`，并设置`PTE_V`（有效）和`PTE_U`（用户态可访问，根据场景调整），此时一级页表项生效。

    为什么要清零新页表?

    新分配的物理页可能残留旧数据，清零能确保未使用的页表项都是`0`（`PTE_V=0`），避免误判为有效映射。

  2. **二级页表（PDX0）的处理**
  - 一级页表项`pdep1`有效后，通过`PDE_ADDR(*pdep1)`提取二级页表的物理地址，再用`KADDR`转换为内核虚拟地址（因为 CPU 运行在虚拟地址空间，必须通过虚拟地址访问内存），得到二级页表的起始地址。
  - 用`PDX0(la)`获取二级页表索引，找到对应的二级页表项`pdep0`。
  - 检查`pdep0`是否有效，处理逻辑和一级页表完全一致：无效则分配新物理页作为三级页表，初始化后设置`pdep0`为有效。

  3. **三级页表（PTX）的处理**
  - 二级页表项`pdep0`有效后，同样通过`PDE_ADDR(*pdep0)`获取三级页表的物理地址，转换为虚拟地址后，用`PTX(la)`获取三级页表索引，最终返回该索引对应的页表项（PTE）地址。

举例：假设虚拟地址`la=0x12345678`，拆解后`PDX1=0x8`、`PDX0=0x12`、`PTX=0x34`，`get_pte`会先查一级页表的第`0x8`项，再查二级页表的第`0x12`项，最后返回三级页表第`0x34`项的地址。

关键：通过多级索引逐级查找，缺表时自动创建，为后续映射做好准备。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    pde_t *pdep1 = &pgdir[PDX1(la)];//找到对应的Giga Page
    if (!(*pdep1 & PTE_V)) {//如果下一级页表不存在，那就给它分配一页，创造新页表
        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);
        //我们现在在虚拟地址空间中，所以要转化为KADDR再memset.
        //不管页表怎么构造，我们确保物理地址和虚拟地址的偏移量始终相同，那么就可以用这种方式完成对物理内存的访问。
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);//注意这里R,W,X全零
    }
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];//再下一级页表
    //这里的逻辑和前面完全一致，页表不存在就现在分配一个
    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);
    }
    //找到输入的虚拟地址la对应的页表项的地址(可能是刚刚分配的)
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

`page_insert`：建立虚拟地址到物理页的映射

作用：将物理页 `page` 映射到虚拟地址 `la`，并设置权限 `perm`。

- **获取页表项地址**
  - 调用 `get_pte(pgdir, la, 1)`，确保能拿到 `la` 对应的 PTE 地址（如果不存在则创建各级页表）。若失败（如内存不足），返回错误。
- **处理旧映射**
  - 如果目标 PTE 已有效（`*ptep & PTE_V`），说明 `la` 之前已映射到某个物理页：
    - 若旧物理页就是 `page`（`p == page`）：说明是重复映射，只需减少 `page` 的引用计数（因为之前 `page_ref_inc` 多增了 1）。
    - 若旧物理页是其他页（`p != page`）：需要调用 `page_remove_pte` 删除旧映射（减少旧页的引用计数，必要时释放），为新映射腾出位置。
- **建立新映射**
  - 增加 `page` 的引用计数（`page_ref_inc(page)`）：因为现在 `la` 也指向它了。
  - 构造新的页表项：`*ptep = pte_create(page2ppn(page), PTE_V | perm)`，其中 `perm` 是权限位（如 `PTE_R | PTE_X`）。
  - 刷新 TLB（`tlb_invalidate`）：TLB 是 CPU 缓存的页表映射，修改页表后必须刷新，否则 CPU 会使用旧映射，导致错误。

关键：确保一个虚拟地址只映射到一个物理页，同时维护物理页的引用计数（被多少虚拟地址映射）。

```
int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    //pgdir是页表基址(satp)，page对应物理页面，la是虚拟地址
    pte_t *ptep = get_pte(pgdir, la, 1);
```

```
    //先找到对应页表项的位置，如果原先不存在，get_pte()会分配页表项的内存
    if (ptep == NULL) {
        return -E_NO_MEM;
    }
    page_ref_inc(page);//指向这个物理页面的虚拟地址增加了一个
    if (*ptep & PTE_V) { //原先存在映射
        struct Page *p = pte2page(*ptep);
        if (p == page) {//如果这个映射原先就有
            page_ref_dec(page);
        } else {//如果原先这个虚拟地址映射到其他物理页面，那么需要删除映射
            page_remove_pte(pgdir, la, ptep);
        }
    }
    *ptep = pte_create(page2ppn(page), PTE_V | perm);//构造页表项
    tlb_invalidate(pgdir, la);//页表改变之后要刷新TLB
    return 0;
}
```

`page_remove`与`page_remove_pte`：删除映射

作用：删除虚拟地址`la`对应的映射，释放不再被引用的物理页。

过程：

- `page_remove`先通过`get_pte`找到页表项`ptep`，再调用`page_remove_pte`处理。
- `page_remove_pte`：
  - 若页表项有效，获取对应的物理页`page`，减少其引用计数。
  - 若引用计数变为 0，释放该物理页（`free_page`）。
  - 清空页表项（`*ptep=0`），刷新 TLB。

关键：通过引用计数追踪物理页的使用，避免提前释放仍被使用的页。

```
void page_remove(pde_t *pgdir, uintptr_t la) {
    pte_t *ptep = get_pte(pgdir, la, 0);//找到页表项所在位置
    if (ptep != NULL) {
        page_remove_pte(pgdir, la, ptep);//删除这个页表项的映射
    }
}
//删除一个页表项以及它的映射
static inline void page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep)
{
    if (*ptep & PTE_V) {  //(1) check if this page table entry is valid
        struct Page *page = pte2page(*ptep);  //(2) find corresponding page
to pte
        page_ref_dec(page);   //(3) decrease page reference
        if (page_ref(page) == 0) {
            //(4) and free this page when page reference reachs 0
            free_page(page);
        }
        *ptep = 0;                  //(5) clear page table entry
        tlb_invalidate(pgdir, la);  //(6) flush tlb
    }
}
```

`check_pgdir`：验证映射功能的正确性

这个函数通过一系列断言测试，验证页表操作的正确性，步骤如下：

这个测试函数通过“创建 - 修改 - 删除”映射的全流程，验证页表操作的正确性，每一步都有明确的校验目标：

1. 初始状态验证

- `assert(get_page(boot_pgdir, 0x0, NULL) == NULL)`：确保虚拟地址 `0x0` 初始无映射。

2. **第一次映射（`p1 -> 0x0`）**

- 分配 `p1`，调用 `page_insert` 映射到 `0x0`。
- 验证 PTE 存在且正确：`get_pte` 返回非空，`pte2page(*ptep) == p1`（PTE 指向 `p1`）。
- 验证引用计数：`page_ref(p1) == 1`（只有 `0x0` 映射它）。

3. **第二次映射（`p2 -> PGSIZE`）**

- 分配 `p2`，映射到 `PGSIZE`（4KB），权限 `PTE_U | PTE_W`。
- 验证权限位：`*ptep & PTE_U` 和 `*ptep & PTE_W` 为真，确保权限生效。
- 验证一级页表项权限：`boot_pgdir[0] & PTE_U` 为真（因为二级页表的权限继承自一级）。

4. **修改映射（`p1 -> PGSIZE`）**

- 调用 `page_insert` 将 `PGSIZE` 的映射从 `p2` 改为 `p1`。
- 验证引用计数：`p1` 的引用计数变为 2（被 `0x0` 和 `PGSIZE` 映射），`p2` 的引用计数变为 0（不再被映射）。
- 验证新映射：`pte2page(*ptep) == p1`，且权限位 `PTE_U` 被清除（因为新权限是 `0`）。

5. **删除映射**

- 删除 `0x0` 的映射：`p1` 的引用计数减为 1（只剩 `PGSIZE` 映射）。
- 删除 `PGSIZE` 的映射：`p1` 的引用计数减为 0（被释放）。
- 清理页表：释放一级页表项对应的物理页（二级页表），清空 `boot_pgdir[0]`，避免影响后续测试。

```
static void check_pgdir(void) {
    // assert(npage <= KMEMSIZE / PGSIZE);
    // The memory starts at 2GB in RISC-V
    // so npage is always larger than KMEMSIZE / PGSIZE
    assert(npage <= KERNTOP / PGSIZE);
    //boot_pgdir是页表的虚拟地址
    assert(boot_pgdir != NULL && (uint32_t)PGOFF(boot_pgdir) == 0);
    assert(get_page(boot_pgdir, 0x0, NULL) == NULL);
    //get_page()尝试找到虚拟内存0x0对应的页，现在当然是没有的，返回NULL

    struct Page *p1, *p2;
    p1 = alloc_page();//拿过来一个物理页面
    assert(page_insert(boot_pgdir, p1, 0x0, 0) == 0);//把这个物理页面通过多级页表映射到0x0
    pte_t *ptep;
    assert((ptep = get_pte(boot_pgdir, 0x0, 0)) != NULL);
    assert(pte2page(*ptep) == p1);
```

```
    assert(page_ref(p1) == 1);

    ptep = (pte_t *)KADDR(PDE_ADDR(boot_pgdir[0]));
    ptep = (pte_t *)KADDR(PDE_ADDR(ptep[0])) + 1;
    assert(get_pte(boot_pgdir, PGSIZE, 0) == ptep);
    //get_pte查找某个虚拟地址对应的页表项，如果不存在这个页表项，会为它分配各级的页表

    p2 = alloc_page();
    assert(page_insert(boot_pgdir, p2, PGSIZE, PTE_U | PTE_W) == 0);
    assert((ptep = get_pte(boot_pgdir, PGSIZE, 0)) != NULL);
    assert(*ptep & PTE_U);
    assert(*ptep & PTE_W);
    assert(boot_pgdir[0] & PTE_U);
    assert(page_ref(p2) == 1);

    assert(page_insert(boot_pgdir, p1, PGSIZE, 0) == 0);
    assert(page_ref(p1) == 2);
    assert(page_ref(p2) == 0);
    assert((ptep = get_pte(boot_pgdir, PGSIZE, 0)) != NULL);
    assert(pte2page(*ptep) == p1);
    assert((*ptep & PTE_U) == 0);

    page_remove(boot_pgdir, 0x0);
    assert(page_ref(p1) == 1);
    assert(page_ref(p2) == 0);

    page_remove(boot_pgdir, PGSIZE);
    assert(page_ref(p1) == 0);
    assert(page_ref(p2) == 0);

    assert(page_ref(pde2page(boot_pgdir[0])) == 1);
    free_page(pde2page(boot_pgdir[0]));
    boot_pgdir[0] = 0;//清除测试的痕迹

    cprintf("check_pgdir() succeeded!\n");
}
```

早期用“大页（Giga Page）”映射时，整个 1GB 虚拟地址范围共享相同的权限（如 `PTE_W=1`），这会导致严重的安全问题：例如内核代码段（.text）本应只读，但大页映射允许写入，可能被恶意修改。

1. 各段权限需求的必要性

- .text：必须 `PTE_R | PTE_X`（可读可执行）、`!PTE_W`（不可写），防止代码被篡改。
- .rodata：必须 `PTE_R`（只读）、`!PTE_W & !PTE_X`，防止数据被修改或当作代码执行。
- .data/.bss：必须 `PTE_R | PTE_W`（可读写）、`!PTE_X`，防止数据被当作代码执行（避免缓冲区溢出攻击）。

2. 如何实现精细化映射?

放弃原有的大页映射，新建页表，对每个段单独映射：

- 遍历内核的各段（.text、.rodata 等），获取它们的虚拟地址范围和所需权限。
- 对每个段的虚拟地址范围，调用 `page_insert` 逐个页映射，设置对应的权限。
- 映射完成后，通过修改 `satp` 寄存器（RISC-V 中存储页表基址的寄存器）切换到新页表，使新映射生效。

为什么要新建页表?

原大页映射的页表项覆盖范围太大（1GB），直接修改会影响整个范围的权限，而新建页表可以从零开始，按段精细设置，避免冲突。

五、内核线程管理

（1）介绍

这段话围绕 “进程与线程的概念”“引入进程的必要性” 以及 “ucore 中进程管理的实现思路” 展开，核心是为理解实验四中内核线程的管理打下基础。我们分三部分详细解析：

进程与线程：核心概念与区别

1. 程序 vs 进程

- **程序**：静态的可执行文件（如编译后的 `.exe` 或 `a.out`），包含代码、数据等静态信息，不占用 CPU 和内存资源（除非被加载）。
- **进程**：程序被加载到内存中运行后的动态实体，是 “正在执行的程序”。它不仅包含程序的代码和静态数据，还包括运行时的动态信息：
  - 寄存器状态（当前执行到哪条指令）；
  - 堆栈（函数调用、局部变量）；
  - 内存映射（虚拟地址空间布局）；
  - 打开的文件、信号等资源。

  简单说：**程序是 “死” 的文件，进程是 “活” 的运行实体。**

2. 进程 vs 线程

- **线程：从进程中剥离出的 “执行流”。一个进程可以有多个线程，它们共享进程的代码、内存空间、文件等资源，但各自拥有独立的 CPU 执行状态（寄存器、堆栈）。**
- **核心区别**：

维度	进程	线程
资源所有权	独立拥有内存、文件等资源	共享所属进程的资源
执行状态	有独立的执行状态	有独立的执行状态（寄存器、栈）
调度单位	可被调度，但粒度较粗	调度的最小单位，粒度更细
开销	创建 / 切换开销大（需分配资源）	创建 / 切换开销小（共享资源）

- **形象比喻**：进程像一个 “工厂”（拥有厂房、设备等资源），线程像工厂里的 “工人”（共享厂房设备，各自独立干活）。

（2）为什么需要进程?

进程的引入是操作系统发展的必然，核心解决了三个关键问题：

1. 支持多任务运行，提升用户体验

没有进程时，计算机一次只能运行一个程序（如早期的单用户单任务系统）。引入进程后，操作系统可以同时管理多个程序运行（如边听歌边写文档），用户无需等待一个任务结束再启动另一个。

2. 充分利用硬件资源

- **单核心 CPU**：通过 “时间片轮转” 调度（进程轮流占用 CPU，每次运行一小段时间），实现多任务 “并发”（宏观上同时运行，微观上交替执行）。
- **多核心 CPU**：多个进程可以同时在不同核心上运行，实现 “并行”，充分发挥多核性能。

3. 资源隔离与安全

每个进程拥有独立的虚拟地址空间，一个进程的错误（如内存越界）不会影响其他进程或操作系统，保证了系统的稳定性和安全性。

（3）ucore 中如何管理进程（内核线程）？

实验四聚焦于**内核线程**（特殊的进程）的管理，核心步骤包括：

1. 进程控制块（PCB）：描述线程的数据结构

内核需要一个数据结构来记录线程的所有信息，即 `proc_struct`（线程控制块）。它包含：

- 线程状态（运行、就绪、阻塞等）；
- 堆栈指针（记录函数调用栈）；
- 寄存器上下文（切换时需保存 / 恢复的 CPU 状态）；
- 页表指针（内核线程共享内核页表）；
- 链表节点（用于将所有线程链接成链表，便于管理）。

  简单说：`proc_struct` 是线程的 “身份证 + 状态报告”，操作系统通过它掌控线程的一切。

2. 进程链表：组织所有线程

所有创建的线程（如 `idleproc`、`initproc`）通过 `proc_struct` 中的链表节点链接成一个全局链表。这样操作系统可以方便地遍历、查找、添加或删除线程（如调度时从链表中选择下一个运行的线程）。

3. 调度器：决定哪个线程运行

调度器是进程管理的 “指挥官”，负责按一定策略（实验四采用 FIFO，即先进先出）从就绪线程中选择一个占用 CPU。当触发调度（如时钟中断、线程主动让出 CPU）时，调度器会：

- 从就绪链表中选出下一个线程；
- 通过 “上下文切换”（`switch_to` 函数）保存当前线程的状态，恢复新线程的状态，完成 CPU 使用权的交接。

4. 实验四中的两个关键内核线程

- **idleproc（0 号线程）**：系统启动后的 “占位线程”。当没有其他线程可运行时，它进入空闲循环（`cpu_idle`），等待新线程就绪。
- **initproc（1 号线程）**：第一个实际执行任务的线程（如输出 “Hello World”）。它通过 `kernel_thread` 函数创建，是后续用户进程创建的模板。

5. 内核线程的特殊性

与用户进程相比，内核线程有两个特点：

- **运行模式**：只运行在内核态（无需切换到用户态）；
- **地址空间**：所有内核线程共享内核的虚拟地址空间（无需为每个线程单独维护页表）。

（4）总结

这段话的核心是：

- 进程是 “动态运行的程序”，线程是进程内的 “独立执行流”，二者在资源共享和调度粒度上有本质区别。
- 进程的引入让操作系统能支持多任务、充分利用硬件、隔离资源，是现代操作系统的基础。
- 实验四中通过`proc_struct`（线程控制块）、进程链表、调度器和上下文切换，实现了内核线程的创建、管理和并发执行，为后续用户进程管理奠定了基础。

简单说，这部分内容解释了 “为什么需要进程” 以及 “ucore 如何用内核线程迈出多任务管理的第一步”。

（2）进程控制块

1. 进程控制结构体

在实验四中，进程管理信息用struct proc_struct表示，在*kern/process/proc.h*中定义如下：

```
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
};
```

`proc_struct`是描述进程 / 线程所有信息的核心结构，每个字段都对应进程管理的关键功能，重点字段解析如下：

1. 进程状态：`state`

- 定义了进程的四种状态：
  - `PROC_UNINIT`：未初始化（刚创建，还没准备好）；
  - `PROC_SLEEPING`：睡眠（阻塞，等待某个事件，不参与调度）；

    - `PROC_RUNNABLE`：就绪（可被调度执行，但当前没占用 CPU）；
    - `PROC_ZOMBIE`：僵尸（进程已结束，但资源未被父进程回收）。
- 状态转换是进程调度的核心（如就绪→运行、运行→睡眠）。

2. **进程标识与关系：`pid`、`parent`**

- `pid`：进程唯一 ID，用于标识不同进程（如 `idleproc` 的 `pid=0`，`initproc` 的 `pid=1`）。
- `parent`：父进程指针，形成“进程树”（除 `idleproc` 外，所有进程都有父进程）。父进程可管理子进程（如回收资源）。

3. 内存与地址空间：`mm`、`pgdir`

- `mm`：指向 `struct mm_struct`（内存管理结构），记录虚拟地址空间布局（如 VMA 区域）。内核线程共享内核地址空间，因此 `mm` 可能为 `NULL` 或指向同一结构。
- `pgdir`：页表根节点的物理地址。进程切换时，CPU 通过 `satp` 寄存器加载 `pgdir`，切换到新进程的地址空间（内核线程共享同一 `pgdir`）。

4. 执行状态：`context`、`tf`

- `context`：进程上下文，保存被调用者保存寄存器（`ra`、`sp`、`s0~s11`），用于进程切换时恢复执行现场（详见下文“进程上下文”）。
- `tf`：中断帧（`struct trapframe`），保存进程从用户态进入内核态时的 CPU 状态（如用户态寄存器、中断原因）。系统调用返回时，通过修改 `tf` 可设置返回值。

5. 内核栈：`kstack`

- 每个进程有独立的内核栈（2 个物理页，8KB），用于内核态执行（如处理中断、系统调用）。
- 作用：
    - 进程切换时，需根据 `kstack` 设置栈指针，确保中断处理使用正确的栈；
    - 内核栈不共享，进程退出时可通过 `kstack` 快速回收。
- 注意：内核栈很小，需避免栈溢出（如不分配大数组）。

6. 调度相关：`need_resched`、`runs`

- `need_resched`：标记进程是否需要被调度（如时间片用完），调度器会优先选择此类进程切换。
- `runs`：进程运行次数，用于调度策略（如实验四 FIFO 暂不依赖，但后续可用于优先级调整）。

7. 链表管理：`list_link`、`hash_link`

- `list_link`：将所有进程链接到全局 `proc_list` 链表，便于遍历所有进程（如调度时查找就绪进程）。
- `hash_link`：基于 `pid` 将进程放入 `hash_list` 哈希表，便于快速查找（如通过 `pid` 定位进程）。

2. 全局进程管理变量：系统级的“进程花名册”

ucore 通过以下全局变量管理所有进程，确保内核能快速访问和操作进程：

1. `current`：指向当前占用 CPU 的进程。仅在进程切换时修改，且需屏蔽中断保证原子性（避免切换过程被打断）。
2. `initproc`：指向第一个实际工作的内核线程（实验四中输出“Hello World”），后续将扩展为第一个用户进程。
3. `hash_list`：哈希表，按 `pid` 组织进程，支持快速查找（时间复杂度 O(1)）。

4. `proc_list`：双向链表，包含所有进程，便于遍历（如统计进程总数、查找就绪进程）。

3. 进程上下文：切换的快照

进程上下文是进程运行状态的“快照”，用于进程切换时保存和恢复关键寄存器，确保进程能无缝继续执行。

1. 保存哪些寄存器?

`struct context` 仅保存 14 个寄存器：`ra`（返回地址）、`sp`（栈指针）、`s0~s11`（被调用者保存寄存器）。

- 原因：利用编译器对函数调用的处理规则 —— 寄存器分为“调用者保存”和“被调用者保存”：
  - 调用者保存寄存器（如 `t0~t6`）：由函数调用者负责保存和恢复，编译器会自动生成代码处理；
  - 被调用者保存寄存器（如 `s0~s11`）：由被调用函数负责保存和恢复，进程切换函数（`switch_to`）需手动保存。

  因此，只需保存被调用者保存寄存器，即可完整恢复进程执行状态，减少切换开销。

2. 上下文的作用

当调度器选择新进程运行时，通过 `switch_to` 函数：

- 保存当前进程的 `context`（将寄存器值写入内存）；
- 加载新进程的 `context`（从内存恢复寄存器值）。

  这样，新进程就能从上次被中断的位置继续执行，仿佛从未被打断。

六、创建并执行内核线程

（1）总体流程

一、内核线程的特殊性：共享内核空间

内核线程是运行在内核态的特殊线程，它的最大特点是**没有独立的地址空间**，所有内核线程共享 ucore 启动时就已经建立好的内核虚拟空间（由 `boot_pgdir` 页表描述）。

- 原因：ucore 启动后，`vmm_init` 已经完成了内核地址空间的静态映射（如内核代码段、数据段、物理内存等的虚拟地址→物理地址映射），`boot_pgdir` 是这个页表的根节点。
- 影响：内核线程无需自己创建页表，直接使用 `boot_pgdir`，因此所有内核线程访问的是同一块物理内存（内核内存），共享内核的代码和数据。

二、创建内核线程的核心步骤：从 `alloc_proc` 到运行

创建内核线程的过程，本质是初始化 `proc_struct`（进程控制块），并为其准备运行环境（栈、上下文等），关键步骤如下：

1. 分配并初始化进程控制块（`alloc_proc`）

`alloc_proc` 函数负责创建一个空白的 `proc_struct`，主要工作：

- 调用 `kmalloc` 分配 `struct proc_struct` 内存（基于 slab 分配器，高效分配小对象）。
- 初始化

```
proc_struct
```

的字段：

- 状态设为 `PROC_UNINIT`（未初始化）；
- `pid` 暂时设为 -1（后续由 `set_proc_name` 等函数设置）；
- 初始化链表节点（`list_link`、`hash_link`），便于加入全局进程链表；
- 其他字段（如 `parent`、`mm`、`pgdir` 等）设为默认值（如 `mm=NULL`，因为内核线程共享地址空间）。

2. 为内核线程准备内核栈（`setup_kstack`）

每个内核线程需要独立的内核栈（`kstack`），用于内核态执行时的函数调用和局部变量存储：

- 调用 `alloc_pages` 分配 2 个连续物理页（8KB，`KSTACKSIZE` 定义）作为内核栈。
- 栈的虚拟地址通过 `KADDR` 转换（内核虚拟地址 = 物理地址 + `KERNBASE` 偏移），并将栈顶地址（`kstack + KSTACKSIZE`）记录到 `proc_struct->kstack`。

3. 初始化进程上下文（`copy_context`）

上下文（`struct context`）保存线程切换时需要恢复的寄存器，确保线程能正确启动和恢复：

- 初始化 `context` 中的 `ra`（返回地址）为内核线程入口函数（如 `kernel_thread_entry`，在 `entry.S` 中定义）。
- 初始化 `sp`（栈指针）为内核栈顶（`kstack + KSTACKSIZE`），确保线程启动时使用正确的栈。

4. 设置页表（简化：共享内核页表）

内核线程无需创建独立页表，直接使用内核的 `boot_pgdir`：

- 将 `proc_struct->pgdir` 设为 `boot_pgdir` 的物理地址（因为 `satp` 寄存器需要物理地址）。
- 这样，线程运行时，CPU 通过 `satp` 加载 `boot_pgdir`，访问的是内核共享的虚拟地址空间。

5. 设置线程状态和名称，加入进程链表

- 将 `proc_struct->state` 设为 `PROC_RUNNABLE`（就绪状态），使其能被调度器选中。
- 通过 `set_proc_name` 设置线程名称（如 “idle”“init”）。
- 将 `proc_struct` 通过 `list_link` 加入全局 `proc_list` 链表，通过 `hash_link` 加入 `hash_list` 哈希表，便于管理和查找。

6. 调度执行：从创建到运行

创建完成后，内核线程处于 “就绪” 状态，等待调度器调度：

- 当调度器（`schedule` 函数）触发时，从就绪队列中选择该线程。
- 通过 `switch_to` 函数切换上下文：保存当前线程的 `context`，加载新线程的 `context`（`ra` 和 `sp` 等寄存器）。
- 新线程从 `kernel_thread_entry` 开始执行，最终跳转到用户指定的函数（如 `initproc` 的 “Hello World” 函数）。

实验四中的两个关键内核线程

1. **idleproc（0 号线程）**：
   - 系统启动后创建的第一个线程，父进程为 `NULL`。

- 功能：当没有其他就绪线程时，执行 `cpu_idle` 进入空闲循环，等待新线程就绪。

2. **initproc（1 号线程）**：
   - 由 `idleproc` 创建，是第一个实际执行任务的线程。
   - 功能：执行测试函数（如输出 “Hello World”），验证线程创建和调度机制。

（2）创建第0个内核线程idleproc

`idleproc` 是系统启动后的第一个 “占位” 线程，看似 “无所事事”，实则是多线程调度的关键。我们一步步拆解：

`idleproc` 的核心作用：系统空闲时的 “占位者”

`idleproc`（空闲进程）是一个特殊的内核线程，主要功能有两个：

1. **填充 CPU 空闲时间**：当系统没有其他就绪线程时，`idleproc` 会占用 CPU，执行一个空循环（`cpu_idle`），避免 CPU “无任务可执行” 的状态。
2. **统一调度逻辑**：让调度器始终有一个 “可调度” 的线程，简化调度器的实现（无需处理 “无就绪线程” 的特殊情况）。

`idleproc` 的创建过程：从 “临时执行流” 到 “正式线程”

`idleproc` 的创建是在 `proc_init` 函数中完成的，本质是将系统启动后的 “临时执行流”（从 `kern_init` 开始的代码执行）包装成一个正式的内核线程。步骤如下：

- **1. 初始化进程管理基础设施**
  - 在创建 `idleproc` 前，先初始化全局进程链表（`proc_list`）和哈希表（`hash_list`），这些结构用于管理所有线程的 `proc_struct`（进程控制块）。
- **2. 分配并初始化 `proc_struct`（进程控制块）**
  - 调用 `alloc_proc` 函数为 `idleproc` 分配 `proc_struct`，并进行初步初始化：
    - **状态设为 `PROC_UNINIT`**：表示线程 “未初始化”，正在准备资源。
    - **`pid` 设为 -1**：临时值，表示尚未分配正式 ID。
    - **`pgdir` 设为 `boot_pgdir`**：复用内核已建立的页表（所有内核线程共享内核地址空间）。

3. **完善 `idleproc` 的初始化：赋予 “正式身份”**

初步初始化后，进一步设置关键字段，让 `idleproc` 成为一个可调度的线程：

- `pid = 0`：分配第一个正式 ID（0），确立 “第 0 个内核线程” 的身份。
- `state = PROC_RUNNABLE`：状态改为 “就绪”，表示可以被调度执行。
- `kstack = (uintptr_t)bootstack`：指定内核栈为系统启动时的初始栈（`bootstack`）。这是因为 `idleproc` 是从启动时的执行流 “改造” 而来的，直接复用已有的栈，无需重新分配。
- `need_resched = 1`：标记 “需要调度”。`idleproc` 的主要任务是在无其他线程时占位，一旦有其他就绪线程（如 `initproc`），就立即触发调度，让出 CPU。
- **设置名称为 “idle”**：通过 `set_proc_name` 标识线程身份，便于调试。

4. `idleproc` 的执行逻辑：`cpu_idle` 函数

`idleproc` 的核心执行函数是 `cpu_idle`，逻辑很简单：

```c
void cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule(); // 如果需要调度，就调用调度器切换线程
        }
    }
}
```

- 当 `need_resched` 为 1 时（初始设置为 1），`idleproc` 会立即调用 `schedule`，让调度器选择其他线程（如 `initproc`）执行。
- 当没有其他线程就绪时，`need_resched` 为 0，`idleproc` 会在循环中 “空转”，占用 CPU。

`idleproc` 的特殊性：继承启动时的执行流

`idleproc` 与后续创建的线程（如 `initproc`）最大的不同是：它不是 “全新创建” 的，而是对系统启动后已有执行流的 “包装”。

- 系统启动时，从 `entry.S` 开始执行，经过 `kern_init`，此时的执行流还没有对应的 `proc_struct`。
- `proc_init` 通过为这个执行流分配 `proc_struct`，并复用其栈（`bootstack`），将其转化为 `idleproc`，使其成为一个可被调度器管理的正式线程。

总结

`idleproc` 的创建是 ucore 从 “单执行流” 转向 “多线程管理” 的第一步：

1. 它通过复用启动时的执行流和栈，成为系统中第一个可调度的线程。
2. 其 `need_resched = 1` 的设置，确保系统启动后能快速切换到真正工作的线程（如 `initproc`）。
3. 作为 “空闲占位者”，它让调度器的实现更简单，同时在无任务时填充 CPU 时间。

简单说，`idleproc` 是系统的 “备胎线程”：平时让贤，忙时占位，是多线程调度不可或缺的基础。

（3）创建第1个内核线程initproc

核心是通过 `kernel_thread` 和 `do_fork` 函数，从无到有构造一个可执行的内核线程，并让它继承系统资源、进入就绪状态。我们分步骤解析：

1. `initproc` 的作用：系统第一个 “干活” 的线程

`idleproc`（0 号线程）主要负责占位和触发调度，而 `initproc`（1 号线程）是第一个实际执行任务的线程：

- 实验四中，它的任务是执行 `init_main` 函数（输出 “Hello World”），验证线程创建和调度机制。
- 后续实验中，它会扩展为创建其他内核线程或用户进程，是系统任务的 “发起者”。

2. 创建内核线程的核心函数：`kernel_thread`

> 第0个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行cpu_idle函数开始过退休生活了。所以uCore接下来还需创建其他进程来完成各种工作，但idleproc内核子线程自己不想做，于是就通过调用kernel_thread函数创建了一个内核线程init_main。在实验四中，这个子内核线程的工作就是输出一些字符串，然后就返回了（参看init_main函数）。但在后续的实验中，init_main的工作就是创建特定的其他内核线程或用户进程（实验五涉及）。

`kernel_thread` 是创建内核线程的入口，它的核心工作是**手动构造一个中断帧（`trapframe`）**，作为新线程的初始执行状态，然后调用 `do_fork` 完成创建。

1. 构造中断帧（`struct trapframe tf`）

中断帧记录了线程的初始 CPU 状态（寄存器、程序计数器等），对于 `initproc` 这类 “无父进程可复制” 的线程，需要手动初始化：

- **清零初始化**：`memset(&tf, 0, sizeof(tf))`，确保初始状态干净。
- 设置函数指针和参数：
  - `tf.gpr.s0 = (uintptr_t)fn`：`s0` 寄存器保存线程要执行的函数（如 `init_main`）。
  - `tf.gpr.s1 = (uintptr_t)arg`：`s1` 寄存器保存函数参数（`init_main` 的参数）。

    这两步确保线程启动后能正确调用目标函数。
- 设置特权级和中断状态（`tf.status`）：
  - `SSTATUS_SPP`：设置为 “超级用户模式”（内核线程运行在内核态）。
  - `SSTATUS_SPIE`：保存 “之前的中断使能状态”（便于后续恢复）。
  - `~SSTATUS_SIE`：暂时禁用中断（避免线程创建过程被打断）。
- 设置程序计数器（`tf.epc`）：

  `tf.epc = (uintptr_t)kernel_thread_entry`，指定线程启动后第一条指令的地址（`kernel_thread_entry` 是汇编实现的入口函数，负责最终跳转到 `fn`）。

2. 调用 `do_fork` 完成创建

`kernel_thread` 最后调用 `do_fork(clone_flags | CLONE_VM, 0, &tf)`，将构造好的中断帧传给 `do_fork`，由后者完成线程的实际创建。

3. 线程创建的核心逻辑：`do_fork` 函数

`kernel_thread` 最后调用 `do_fork(clone_flags | CLONE_VM, 0, &tf)`，将构造好的中断帧传给 `do_fork`，由后者完成线程的实际创建。

1. **分配并初始化进程控制块（`alloc_proc`）**
   - 调用 `alloc_proc` 为 `initproc` 分配 `struct proc_struct`，并初始化基础字段（如状态为 `PROC_UNINIT`、`pid` 为 - 1、共享内核页表 `boot_pgdir` 等）。
2. **分配并初始化内核栈（`setup_stack`）**
   - 为 `initproc` 分配独立的内核栈（2 个物理页，8KB），并将栈顶地址记录到 `proc->kstack`。与 `idleproc` 复用启动栈不同，`initproc` 的栈是全新分配的，确保独立性。
3. **处理内存管理（`copy_mm`）**
   - 由于 `initproc` 是内核线程，无需独立的用户地址空间，`copy_mm` 简单地将 `proc->mm` 设为 `NULL`（表示共享内核地址空间）。
4. **设置中断帧和上下文（`copy_thread`）**
   - 在 kernel 栈上分配中断帧空间：

     ```
     proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE - sizeof(struct trapframe))
     ```

     内核栈从高地址向低地址生长，因此中断帧被放在栈顶（栈的最高地址处）。

- **复制中断帧**：`*(proc->tf) = *tf`，将 `kernel_thread` 构造的中断帧复制到新线程的内核栈。
- **标记为子进程**：`proc->tf->gpr.a0 = 0`，`a0` 是函数返回值寄存器，设置为 0 表示这是新创建的子线程（区别于父线程）。
- 初始化上下文（`proc->context`）：
  - `proc->context.ra = (uintptr_t)forkret`：`ra`（返回地址）设为 `forkret` 函数，线程切换后从这里开始执行。
  - `proc->context.sp = (uintptr_t)(proc->tf)`：`sp`（栈指针）设为中断帧的地址，确保切换后使用正确的栈。

5. **将新线程加入进程链表**
   - 通过 `list_add` 将 `proc->list_link` 加入全局 `proc_list`，`proc->hash_link` 加入 `hash_list`，便于调度器管理和查找。
6. **设置线程状态为就绪**
   - `proc->state = PROC_RUNNABLE`，使 `initproc` 进入就绪队列，等待调度器选中执行。
7. **返回新线程的 `pid`**
   - `do_fork` 最终返回新线程的 `pid`（`initproc` 的 `pid=1`），完成创建。

4. `initproc` 的启动逻辑：从创建到执行

`initproc` 创建后处于就绪状态，当 `idleproc` 执行 `cpu_idle` 时，由于 `need_resched=1`，会调用 `schedule` 调度器：

1. 调度器从就绪队列中选中 `initproc`。
2. 通过 `switch_to` 函数切换上下文：保存 `idleproc` 的 `context`，加载 `initproc` 的 `context`（`ra=forkret`，`sp=中断帧地址`）。
3. `initproc` 从 `forkret` 开始执行，最终通过 `kernel_thread_entry` 跳转到 `init_main` 函数，输出 “Hello World”。

总结

`initproc` 的创建是 ucore 多线程能力的关键验证：

- 通过 `kernel_thread` 手动构造中断帧，解决了 “无父进程可复制” 的问题。
- `do_fork` 完成了从资源分配到状态设置的全流程，确保新线程具备执行条件。
- 最终通过调度器切换，`initproc` 获得 CPU 执行权，证明了线程创建和调度机制的正确性。

简单说，`initproc` 是系统的 “第一个劳动者”，它的成功运行标志着 ucore 多线程框架的搭建完成。

（4）调度并执行内核线程iniitproc

ucore 中内核线程的调度与切换过程，核心是 `idleproc` 如何让出 CPU，`schedule` 调度器如何选择 `initproc`，以及 `switch_to` 如何完成上下文切换，最终让 `initproc` 执行。我们分步骤解析：

1. 调度的触发：`idleproc` 主动让出 CPU

`idleproc`（0 号线程）的核心逻辑在 `cpu_idle` 函数中，它是调度的起点：

```c
void cpu_idle(void) {
    while (1) {
        if (current->need_resched) { // current指向idleproc
            schedule(); // 触发调度
        }
    }
}
```

- `idleproc` 初始化时被设置为 `need_resched=1`（需要调度），因此进入 `cpu_idle` 后会立即调用 `schedule`，主动让出 CPU。
- 这一设计确保系统初始化完成后，不会停留在 `idleproc`，而是切换到真正工作的线程（如 `initproc`）。

2. 进程状态：调度的准则

ucore 的进程有四种状态，调度器只关注 `PROC_RUNNABLE`（就绪 / 运行）状态：

- `PROC_UNINIT`：未初始化（如刚创建的进程），不参与调度。
- `PROC_RUNNABLE`：就绪（等待 CPU）或正在运行，是调度器的候选对象。
- `PROC_SLEEPING`：阻塞（等待资源 / 事件），暂时不参与调度。
- `PROC_ZOMBIE`：已结束（等待父进程回收），不参与调度。

3. 调度器 `schedule`：选择下一个运行的线程

`chedule` 是调度的核心函数，采用简单的 FIFO（先进先出）策略，步骤如下：

1. 重置当前进程的调度标志

将 `current->need_resched` 设为 0（表示当前进程暂时不需要被调度）。

2. 查找下一个就绪进程（`next`）

遍历全局进程链表 `proc_list`，寻找状态为 `PROC_RUNNABLE` 的进程，查找规则：

- 若当前进程是 `idleproc`：从链表头开始查找，确保所有就绪进程有平等机会。
- 若当前进程是其他线程：从当前进程的下一个位置开始查找，实现简单的轮转调度（Round-Robin）。
- 若找不到就绪进程：默认选择 `idleproc`（保底，避免 CPU 无任务可执行）。

在实验四中，`proc_list` 中只有 `idleproc`（`PROC_RUNNABLE`）和 `initproc`（`PROC_RUNNABLE`），因此 `schedule` 会选中 `initproc`。

3. 执行进程切换：`proc_run(next)`

找到 `next`（`initproc`）后，调用 `proc_run` 完成切换，核心做三件事：

- 更新 `current` 指针：`current = next`，标记当前运行的进程为 `initproc`。
- 切换页表：将 `next->pgdir`（内核线程共享 `boot_pgdir`）加载到 `satp` 寄存器，确保地址空间正确（此处内核线程共享页表，实际无变化）。
- 切换上下文：调用 `switch_to` 保存当前进程（`idleproc`）的上下文，恢复 `next`（`initproc`）的上下文。

4. 上下文切换：`switch_to`的底层实现

`switch_to`是用汇编实现的关键函数，负责保存和恢复进程的执行现场（寄存器状态），完成 CPU 使用权的真正交接。

1. 保存当前进程（`from`）的上下文

将被调用者保存寄存器（`ra`、`sp`、`s0~s11`）写入`from->context`：

```
STORE ra, 0*REGBYTES(a0)    // 保存返回地址ra到from->context.ra
STORE sp, 1*REGBYTES(a0)    // 保存栈指针sp到from->context.sp
STORE s0, 2*REGBYTES(a0)    // 保存s0~s11到from->context.s0~s11
...（依次保存s1~s11）
```

这些寄存器是进程恢复执行所必需的（调用者保存寄存器由编译器自动处理）。

2. 恢复新进程（`to`）的上下文

从`to->context`中读取寄存器值，恢复到 CPU：

```
LOAD ra, 0*REGBYTES(a1)    // 从to->context.ra恢复ra
LOAD sp, 1*REGBYTES(a1)    // 从to->context.sp恢复sp
LOAD s0, 2*REGBYTES(a1)    // 从to->context.s0~s11恢复s0~s11
...（依次恢复s1~s11）
```

恢复后，CPU 的寄存器状态与`initproc`被中断前一致。

3. 跳转执行：从`switch_to`返回

`switch_to`的`ret`指令会跳转到`to->context.ra`（`initproc`的`context.ra`被设置为`forkret`），因此接下来执行`forkret`函数。

5. `initproc`的启动：从上下文切换到执行任务

`initproc`的执行流程从`forkret`开始，逐步跳转到目标函数（`init_main`）：

1. `forkret`：设置栈并跳转到中断恢复逻辑

```
forkrets:
    move sp, a0           // a0是initproc->tf（中断帧地址），将栈指针指向中断帧
    j __trapret           // 跳转到中断恢复函数
```

`__trapret`是中断处理的通用出口，负责从`tf`（中断帧）中恢复所有寄存器（包括`epc`、`status`等）。

2. `__trapret`：恢复中断帧，跳转到`kernel_thread_entry`

`__trapret`从`initproc->tf`中恢复寄存器，其中`tf.epc`被设置为`kernel_thread_entry`（`kernel_thread`中初始化），因此 CPU 会跳转到`kernel_thread_entry`执行。

3. `kernel_thread_entry`：调用目标函数（`init_main`）

```
kernel_thread_entry:
    move a0, s1          // s1保存函数参数（init_main的参数），传给a0（函数调用约定）
    jalr s0              // s0保存函数指针（init_main），跳转到init_main执行
    jal do_exit          // 函数执行完毕后，调用do_exit退出
```

至此，`initproc`成功执行`init_main`函数（输出 “Hello World”），完成其任务。

6. 总结

整个调度与执行过程的核心逻辑是：

1. `idleproc`通过`cpu_idle`触发调度（`schedule`）。
2. `schedule`选择`initproc`作为下一个运行的线程。
3. `switch_to`完成上下文切换，保存`idleproc`的状态，恢复`initproc`的状态。
4. `initproc`通过`forkret`、`__trapret`、`kernel_thread_entry`最终执行目标函数。

这一过程实现了多线程的并发执行，是 ucore 从单任务到多任务的关键一步。