OS Lab4

一、实验目的

总结下来，本次实验目的是让操作系统从“单线程单任务”的内核，升级为能管理多个执行流（线程）并支持虚拟空间结构的内核。

本次实验要实现：1.让系统具备基本的虚拟内存管理框架，为后续进程隔离、缺页处理等功能打基础；2.内核可以让多个任务“并发”进行，看起来像多个程序同时运行。（两个要求都是对内核而言，要对操作系统内核处理）

与前几次实验的联系：

- Lab2提供了“系统如何拥有并使用内存”（供线程分配内核栈使用）。
- Lab3提供了“系统如何响应外部事件、保存寄存器”（提供上下文保存/恢复机制和“中断框架”）。
- Lab4基于上述实验，构建“可并发执行”的多线程内核。

二、实验原理

1、实验执行流程：

为什么要记录这个部分，因为我本次Lab4与上几次Lab关系很大，要用到上几次Lab的知识和代码，所以我认为有必要彻底总结一下本次实验的流程。

因为本次实验目的是从“系统能启动并初始化内存”进化到“系统能创建与调度线程”，所以首先对整个流程进行总览：

阶段	模块	主要函数	所在文件	核心任务
1	启动与内核加载	`kern_init` / `init`	`kern/init/init.c`	初始化内核的总控函数，依次调用各子系统初始化函数
2	物理内存管理	`pmm_init`	`kern/mm/pmm.c`	探测物理内存，建立空闲页管理机制
3	中断与异常初始化	`pic_init`, `idt_init`	`kern/trap/trap.c`	设置中断控制器与中断描述符表，使 CPU 能响应中断
4	虚拟内存管理	`vmm_init`, `boot_map_segment`	`kern/mm/vmm.c`	建立页表结构，实现虚拟地址到物理地址的基本映射

阶段	模块	主要函数	所在文件	核心任务
5	进程/线程管理	`proc_init`, `alloc_proc`, `kernel_thread`, `schedule`	`kern/process/proc.c`	初始化 PCB、创建内核线程、实现调度机制
6	调度与上下文切换	`schedule`, `proc_run`, `switch_to`	`kern/process/schedule.c`	在多个进程间切换，分时共享 CPU
7	测试运行	`init_main`	`kern/init/init.c`（或线程入口）	第一个线程输出 “Hello World”，验证机制正确性

对整个流程进行详细分析：

1.1 内核入口：init（）--一切的起点

位于kern/init/init.c ， 是整个OS初始化的调度中心。按照顺序调用：

- pmm_init()：物理内存管理
- pic_init() / idt_init()： 中断控制初始化
- vmm_init()： 虚拟内存初始化
- proc_init()： 创建并初始化第一个进程

到此，整个“硬件-内核抽象层”建立完毕。

1.2 物理内存管理：pmm_init()

位于kern/mm/pmm.c ， 这部分主要为“虚拟内存+线程栈”提供底层支持。

这部分要做的是：

- 探测系统有多少内存；
- 划分哪些页留给内核；
- 建立页分配算法结构（如 `free_area_t`、空闲链表）；
- 提供分配接口（`alloc_page()`、`free_page()`）。

该部分可以理解为：让系统知道内存在哪里。

1.3 中断与异常：pic_init() ， idt_init()

位于kern/trap/trap.c 和kern / driver /picirq.c

1.4 虚拟内存初始化：vmm_init()

位于kern/mm/vmm.c ，该部分让内核在物理内存基础上建立虚拟地址映射关系，确保：

- 内核的逻辑地址有合法的物理页支撑
- 从此CPU通过分页机制访问内存

也就是说，这一步把内存从“物理世界”转向“虚拟世界”

1.5 进程管理初始化：proc_init()

位于kern/process/proc.c

1.6 总结：

整个过程用指导书中的话来总结就是：

如果要让内核线程运行，我们首先要**创建内核线程对应的进程控制块**，还需**把这些进程控制块通过链表连在一起**，便于随时进行插入，删除和查找操作等进程管理事务。这个链表就是进程控制块链表。然后再**通过调度器（scheduler）来让不同的内核线程在不同的时间段占用CPU执行**，调度器会按照一定策略选择哪个线程获得CPU执行权，实现对CPU的分时共享。

2、虚拟内存管理：

其实在Lab2中已经提到了内存管理的概念，但是在Lab2中更多讲解的是物理内存的管理，本次实验是在物理内存管理的基础上，引入了虚拟内存管理，所以此处作为知识点归纳。

2.1 虚拟内存的概念：

虚拟内存是一种内存管理技术，为了提供计算机系统的有效内存利用，允许程序在无需考虑物理内存的情况下进行编程和执行。使程序能够使用比实际物理内存更大的内存空间。

- 虚拟地址与物理地址的映射：
  - 虚拟内存单元（虚拟地址）并不总是与物理内存单元一一对应，实际物理内存单元可能不存在，或者二者的地址不相等。
  - 操作系统通过页表（Page Table）建立虚拟地址与物理内存地址的对应关系，使得访问虚拟地址可以动态地转换为物理地址。

2.2 虚拟内存的作用和意义：

- 内存地址虚拟化：
  - 虚拟内存通过分页机制，将程序员或CPU“看到”的地址与实际物理地址完全隔离。在这个过程中，程序可以在一个受保护的空间内运行，避免了直接访问物理内存可能带来的安全隐患。
- 内存保护：
  - 通过配置页表项，操作系统可以限制程序对特定内存区域的访问，保护内存不被非法访问。
- 按需分页：
  - 操作系统仅在程序真正访问某个虚拟内存页时，才分配对应的物理内存。这种按需分配减少了内存的浪费，并允许多个程序在有限的物理内存中并发运行。
- 页面换入换出：

- 将不常访问的页面写入硬盘，以便为更活跃的页面腾出物理内存。当需要访问被换出的页面时，系统会将其从硬盘读回内存。

2.3 页表项设计思想：

- SV39页面表项结构：
  - 使用多级页表结构，通常为三级（SV39），每个页表项（PTE）包含有效位，读写权限等信息。

    比如在kern/mm/mmu.h 中，使用以下代码分别表示有效位、可读、可写、可执行位等：

```
// page table entry (PTE) fields
#define PTE_V     0x001 // Valid
#define PTE_R     0x002 // Read
#define PTE_W     0x004 // Write
#define PTE_X     0x008 // Execute
#define PTE_U     0x010 // User
```

  - 页表项的设计旨在有效管理虚拟地址到物理地址的映射，具体结构中涉及到虚拟地址的各个部分，如
    页目录索引、页表索引等。

2.4 实际操作中的实现：

现在我们要通过多级页表（如SV39）实现虚拟存储，要将虚拟地址映射到物理地址.在SV39结构中，每个虚拟地址被分为多个部分（多个级别的页目录和页表组成），每个部分负责管理不同的地址空间。

- 主要操作接口：
  - page_insert():
    - 在页表中建立虚拟地址到物理地址的映射。
    - 具体代码如下所示：

```
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;
}
/*
具体流程：
1、通过调用get_pte()获取指定虚拟地址的页表项。如果页表项不存在，get_pte()会根据需要分配新的页表项。
2、更新物理页面的引用计数，确保系统正确跟踪页面的使用情况。
3、如果原本已经有一个映射，检查其有效性并执行必要的更新或替换。
4、创建新的页表项并为其设置适当的权限（如可读、可写等）。
5、刷新TLB（Translation Lookaside Buffer），以反映新的映射。
*/
```

- page_remove():
  - 删除虚拟地址与物理页面之间的映射。
- get_pte()
  - 获取指定虚拟地址的页表项指针。其中参数包括：

    ①、pgdir：页目录的内核虚拟地址

    ②、la：需要映射的线性地址

    ③、create：一个标志，用于决定是否在找不到相关页时分配新的页
- check_pgdir()
  - 验证页表操作是否成功，包括检查映射是否正确、权限设置是否合理、引用计数是否正确等。是确保内存管理有效性的关键部分。

3、进程控制块PCB：

3.1 PCB的主要作用：

远程控制块PCB是操作系统管理控制进程运行的信息集合，也就是说，操作系统是通过管理PCB来管理进程，PCB也是进程存在的唯一标识。（用于描述线程的数据结构）。

3.2 PCB的主要信息：

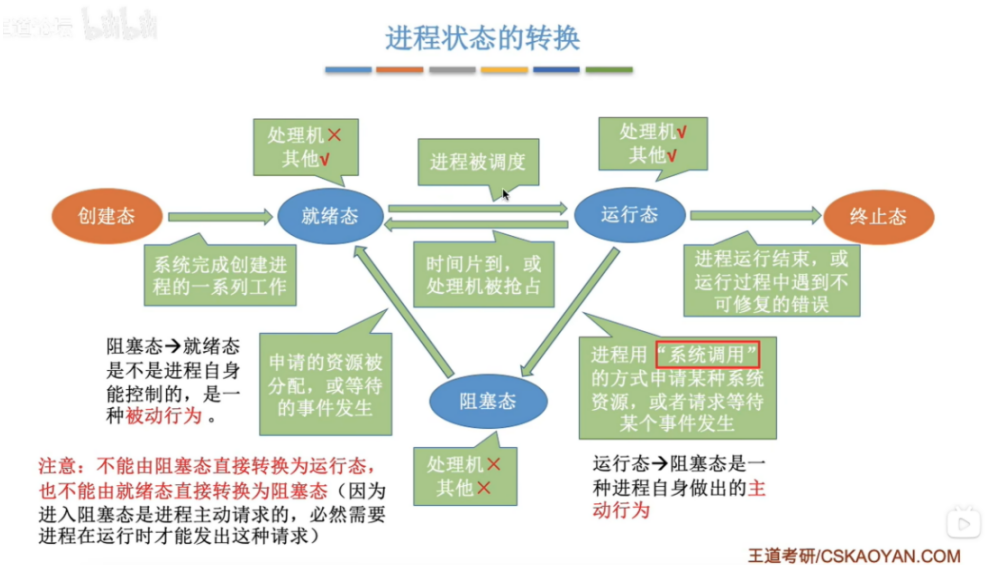
在实验中用如下代码设计PCB（struct proc_struct）：

```
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
```

```
};
```

对其中成员变量做说明：

- mm：保存了内存管理的信息，包括内存映射、虚存管理等。
- state：保存了进程所处的状态。一般uCore中进程状态有四种：（也可以参考下图）
  - PROC_UNINIT（未初始化）：进程控制块刚被创建，尚未完成初始化
  - PROC_SLEEPING（睡眠/阻塞）：进程因等待某个事件或资源而暂时无法执行
  - PROC_RUNNABLE（就绪/运行）：进程已准备好运行，可能正在等待 CPU 或正在 CPU 上执行
  - PROC_ZOMBIE（僵尸）：进程已执行完毕，等待父进程回收其资源



- parent：保存了父进程的指针。在内核中，只有内核创建的idle进程没有父进程，其他进程都有父进程。通俗点来讲就是保存指向父进程 PCB 的指针，提供一个“联系”的接口。
- context：保存了进程执行的上下文，也就是几个关键的寄存器的值。在提问阶段详细回答了。
- tf：保存了进程的中断帧。具体的可以查看Lab3中断的内容。
- pgdir：页目录的基址。之前几次实验知道了CPU通过satp寄存器找到当前页表的根节点，从而进行地址翻译。padir字段保存的就是**每个进程的页表页表根目录的物理地址**。当进行进程切换时，内核需要将下一个要运行进程的 `pgdir` 值加载到 `satp` 寄存器中，这样才能正确地切换到新的地址空间。（页目录：存放指向页表的指针的结构）
- kstack：在创建内核线程时，操作系统会为每个内核线程分配一个独立的内核栈（kstack）。该栈位于内核地址空间中，专门用于内核线程在执行期间保存局部变量、返回地址以及其他状态信息。也就是在创建PCB时，系统会为其分配相应的kstack空间。

同时，为了管理PCB，uCore还涉及以下全部变量：

- static struct proc*current：当前占用CPU并且处于“运行态”的PCB的指针。
- static struct proc*initproc：指向第一个内核线程的指针。
- static list_entry_t hash_list[HASH_LIST_SIZE]：所有PCB的哈希表，proc_struct中的成员变量hash_link将基于pid链接入这个哈希表中。为了查找。
- list_entry_t proc_list：所有PCB的双向线性列表，proc_struct中的成员变量list_link将链接入这个链表中。为了遍历。

4、上下文切换：

其实在Lab3中就已经出现了上下文切换这个概念，只不过在Lab3中的上下文切换涉及的结构是trapframe，在内核态和用户态之间切换，每次陷入（trap）后仍返回同一个进程；而在本次Lab4实验中，上下文切换指的是多个线程轮流执行，涉及struct context结构，是内核态之间的切换。

4.1 struct context结构

```
struct context
{
    uintptr_t ra;  //Return Address（返回地址寄存器）
    uintptr_t sp;  //Stack Pointer（栈顶指针）
    uintptr_t s0;  //s0--s11都是通用寄存器。
    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;
};
```

struct context结构是实现上下文切换的基本单元，负责保存一个线程（或进程）的CPU寄存器上下文，一般嵌在进程控制块PCB或线程控制块TCB中。

其中的ra、sp、s0到s11指的都是CPU寄存器，负责上下文切换时保存from部分、加载to部分。

那为什么只保存一部分寄存器就可以实现上下文切换呢?

回答：寄存器可以分为调用者保存（caller-saved）寄存器和被调用者保存（callee-saved）寄存器。因为线程切换在一个函数当中，所以编译器会自动帮助我们生成保存和恢复调用者保存寄存器的代码，在实际的进程切换过程中我们只需要保存被调用者保存寄存器就行。

4.2 switch_to()函数

switch_to()函数源码部分：

```
# void switch_to(struct proc_struct* from, struct proc_struct* to)
.globl switch_to    //声明一个全局函数，供C语言调用
switch_to:
    # save from's registers
    STORE ra, 0*REGBYTES(a0) //把寄存器ra的内容写入到内存地址‘a0+0’开始的位置。
    STORE sp, 1*REGBYTES(a0)
    STORE s0, 2*REGBYTES(a0)
    STORE s1, 3*REGBYTES(a0)
    STORE s2, 4*REGBYTES(a0)
    STORE s3, 5*REGBYTES(a0)
    STORE s4, 6*REGBYTES(a0)
    STORE s5, 7*REGBYTES(a0)
    STORE s6, 8*REGBYTES(a0)
```

```
    STORE s7, 9*REGBYTES(a0)
    STORE s8, 10*REGBYTES(a0)
    STORE s9, 11*REGBYTES(a0)
    STORE s10, 12*REGBYTES(a0)
    STORE s11, 13*REGBYTES(a0)
/*上述为“保存当前进程的上下文（from）”部分。a0指的是第一个传入参数，即from的指针（当前进程的struct）；保存当前进程中所有被调用者保存寄存器（callee-saved registers），包括：ra、sp、s0-s11*/

    # restore to's registers
    LOAD ra, 0*REGBYTES(a1)
    LOAD sp, 1*REGBYTES(a1)
    LOAD s0, 2*REGBYTES(a1)
    LOAD s1, 3*REGBYTES(a1)
    LOAD s2, 4*REGBYTES(a1)
    LOAD s3, 5*REGBYTES(a1)
    LOAD s4, 6*REGBYTES(a1)
    LOAD s5, 7*REGBYTES(a1)
    LOAD s6, 8*REGBYTES(a1)
    LOAD s7, 9*REGBYTES(a1)
    LOAD s8, 10*REGBYTES(a1)
    LOAD s9, 11*REGBYTES(a1)
    LOAD s10, 12*REGBYTES(a1)
    LOAD s11, 13*REGBYTES(a1)
/*上述为“恢复目标进程的上下文（to）”部分。其中，a1是第二个参数，即to的指针（目标进程的struct），使用“LOAD”将a1地址中的寄存器值重新加载如CPU寄存器中。*/

    ret
/*ret是一个伪指令，相当于“jalr x0，0(ra)”,意思是：跳转到寄存器ra所存放的地址去执行，并且不保存返回地址（因为目标寄存器是 x0，恒为 0）。所以ra 寄存器的值决定了跳转目的地。
```

这段代码负责：在操作系统内核中，从当前正在运行的进程（`from`）切换到另一个待运行的进程（`to`）。也就是说：

- 把当前进程的一些关键寄存器内容保存（STORE）到它自己的 `proc_struct` 结构（a0）中；
- 然后从目标进程 `to` 的结构（a1）中恢复（LOAD）它以前保存的寄存器内容；
- 接着用 `ret` 返回，此时实际上程序流“跳转”到了被切换进的进程。

5、内核线程管理：

5.1 内核线程和用户进程：

这部分的实验指导中提到了内核线程和用户进程两个概念，我认为有必要对这两个概念进行一个透彻的分析。

- 内核线程：
  - 内核线程是在内核态下执行的线程，通常用于处理系统任务，直接使用内核的资源。
  - 在这次实验中，会创建一个或多个内核线程（给内核线程分配进程控制块“proc_struct”），用于执行特定的系统功能，比如资源管理、设备驱动、系统调度等。例如例子中的idleproc、initproc内核线程。

- 用户进程：
  - 用户进程会在用户态和内核态交替执行，主要用于运行用户应用程序。有各自独立的用户空间内存，并需要通过系统调用与内核通信。
  - 本次实验中涉及的initproc可能会在后续实验中演变为用户进程，负责用户层级的任务。
- 内核线程和用户进程的区别：
  - 内核线程只运行在内核态，而用户进程会在在用户态和内核态交替运行。
  - 所有内核线程直接使用共同的ucore内核内存空间，不需为每个内核线程维护单独的内存空间，而用户进程需要维护各自的用户内存空间。

5.2 创建并执行内核线程（实验主体部分）：

1、创建第0个内核线程idleproc：

- idleproc概念：是一个特殊的内核线程，代表系统空闲时运行的线程。主要作用是在没有其他任务执行时占用CPU时间，确保CPU处于活跃状态。

2、创建第1个内核线程initproc：

- 在uCore操作系统中，第一个内核线程的创建是通过kernel_thread函数实现的。该函数负责为新的内核线程创建一个初始化好的中断帧，并通过do_fork函数把它转化为一个新的进程。

3、调度并执行内核线程initproc

三、练习

练习1：

一、代码分析

练习1主要使用代码（kern/process/proc.c），该代码实现了ucore内核中的进程机制（process/thread mechanism），负责描述、创建、调度和管理所有执行实体（线程或进程）。proc.c代码的作用总结下来就是：“它是操作系统内核在建立和管理进程表、分配PID、为每个进程分配栈和上下文的核心逻辑”

在ucore中，存在以下信息：

- 线程和进程本质相同（统一用proc_struct）表示。
- 内核线程共享一个地址空间

proc.c代码中的各个模块如下所示：

模块	功能
进程控制块（`proc_struct`）	保存线程状态

模块	功能
进程创建（`do_fork()`）	创建新线程
上下文切换（`proc_run()`）	切换 CPU 执行流
线程初始化（`proc_init()`）	建立两个线程：idleproc 与 initproc
调度（`schedule()`）	（在别的文件实现）运行下一个线程
退出机制（`do_exit()`）	线程终止（这里只是占位）

总之就是操作系统要管理各个进程，能让多个任务并发运行，并随时切换它们。大致操作步骤可以总结为：

```
alloc_proc() ----->创建进程的“空白表”
setup_kstack() --->给这个表分配内核栈空间
copy_mm()  --->填入内存空间信息
copy_thread()  --->设计寄存器/上下文
do_fork()   --->把它连成流程，生成新进程
proc_run()  --->实际切换执行
```

二、代码补充

要补充proc.c中的alloc_proc()、do_fork()、proc_run()三个部分。

1、alloc_proc()--创建新的进程初始块

该部分目的是创建一个新的进程初始块(proc_struct)。整个流程大体意思是：当你创建一个新的进程（或内核线程）时，系统要先在内核里分配一个proc_struct结构体，然后把它的所有字段都初始化为合适、干净的状态。

根据代码中的注释部分，发现proc_struct结构体中的字段如下所示：

字段	类型	含义	在系统里的用途
`state`	枚举	当前进程状态	如 RUNNABLE、SLEEPING 等
`pid`	整型	进程标识号	唯一编号
`runs`	整型	被调度次数	调试、统计用
`kstack`	地址	栈基地址	该线程的内核栈
`need_resched`	布尔	是否要重新调度	控制调度器行为
`parent`	指针	父进程	确立“谁创建了我”
`mm`	指针	内存映射结构	管理虚拟内存页
`context`	结构	CPU寄存器保存区	负责 `switch_to` 切换寄存器
`tf`	指针	中断帧（trap frame）	保存异常/中断状态

字段	类型	含义	在系统里的用途
`pgdir`	地址	页表地址	告诉硬件用哪张页表寻址
`flags`	整型	运行标志位	控制退出等状态
`name`	字符数组	名字	打印调试用

补充内容（对PCB中的字段进行初始化）：

```
        proc->state = PROC_UNINIT;              // 初始状态
        proc->pid = -1;                         // 未分配PID
        proc->runs = 0;                         // 运行次数
        proc->kstack = 0;                       // 未分配内核栈
        proc->need_resched = 0;                 // 不急于调度
        proc->parent = NULL;                    // 无父进程
        proc->mm = NULL;                        // 无内存空间结构
        memset(&(proc->context), 0, sizeof(struct context)); // 清空上下文
        proc->tf = NULL;                        // 无trapframe
        proc->pgdir = boot_pgdir_pa;            // 内核页表基地址共享
        proc->flags = 0;                        // 标志清零
        memset(proc->name, 0, PROC_NAME_LEN + 1); // 名称清空
```

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。

代码部分

```
//    1. 调用alloc_proc，首先获得一块用户信息块。
    proc = alloc_proc();
    if (proc == NULL) {
```

```
        goto fork_out;
    }

    //    2. 调用 setup_kstack 为子进程分配内核栈
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }

    //    3. 根据clone_flag调用copy_mm进行复制或共享mm
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    //    4. 调用 copy_thread 来在 proc_struct 中设置 tf 和上下文
    copy_thread(proc, stack, tf);

    //    5. 将proc_struct插入到hash_list和proc_list中
    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. 调用wakeup_proc使新的子进程变为可运行状态
    wakeup_proc(proc);

    //    7. 使用子进程的 PID 设置返回值
    ret = proc->pid;
```

问题回答

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。
是的，ucore能够给每个新fork的线程一个唯一的id。理由和分析如下：

- 理由：系统通过多层次的保护机制确保PID的唯一性：首先通过static_assert(MAX_PID > MAX_PROCESS)确保PID数量足够；在具体的分配过程中，get_pid()函数使用last_pid和next_safe两个变量构成的循环查找算法，从last_pid+1开始遍历，遇到已被占用的PID就继续递增，同时动态更新next_safe作为安全边界；更重要的是，在do_fork中PID分配和进程添加操作都在关中断保护下原子执行，避免了竞态条件。这种设计结合进程数限制和原子操作，确保了在系统资源允许范围内每个新创建进程都能获得唯一的PID标识符。

练习3：编写proc_run函数（需要编程）

```
bool intr_flag;
struct proc_struct *prev = current;

local_intr_save(intr_flag);
    {
        current = proc;
        lsatp((unsigned int)proc->pgdir);
        switch_to(&prev->context, &proc->context);
    }
local_intr_restore(intr_flag);
```

这段是函数proc_run的补充代码，该函数的作用是将当前执行的进程切换到目标进程`proc`运行，补充代码是确定当前运行进程不是目标进程`proc`时，实现进程的切换操作。
首先声明布尔变量`intr_flag`用来保存内核的中断状态，接着用prev来保存当前运行的进程。
接下来进行进程的切换，先调用函数`local_intr_save`来禁用当前处理器的中断并将中断的原始状态保存到`intr_flag`中，这是因为进程切换涉及到进程上下文的切换，避免中断打断切换过程，导致上下文混乱。接着将当前进程指针指向目标进程proc,调用`lsatp`函数将目标进程的页表基地址写入satp寄存器，最后调用`switch_to`函数切换进程的上下文。
最后调用`local_intr_restore`函数根据`intr_flag`的值恢复先前的处理器中断状态。

扩展练习Challenge：

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

首先分析这两个宏（或内联函数）的功能：

宏名	功能
`local_intr_save(x)`	关闭当前 CPU 的中断，但**保存原来的中断使能状态**到变量 `x`
`local_intr_restore(x)`	根据 `x` 的值，**恢复之前的中断状态**（如果原来开，就重新开；原来关，就保持关）

这两个宏主要是通过操作CPU的状态控制寄存器（sstatus）,特别是其中断使能位SIE：

- SIE（中断使能位）：位于sstatus第1位：
  - 置1：开启中断
  - 置0：关闭中断

两个宏的伪代码大致如下：

```
#define local_intr_save(x)    \
    do {                      \
        x = read_csr(sstatus);         /* 读取当前状态寄存器 */ \
        clear_csr(sstatus, SSTATUS_SIE); /* 清除SIE位，关闭中断 */ \
    } while (0)
```

local_intr_save中将当前sstatus内容保存在x中，并且清除SIE位，关闭中断（即不能进行中断处理，防止切换时被打断）；最后将保存的状态保存到x，方便之后还原。

```
==========================================================================================
    #define local_intr_restore(x) \
    do {                      \
        if (x & SSTATUS_SIE)  \
            set_csr(sstatus, SSTATUS_SIE);  /* 如果原来开中断，就重新开启 */ \
    } while (0)
```

local_intr_restore根据x中记录的状态判断是否要恢复中断。

对于这个部分，我在初次回答的时候认为完全没有必要保存更改前的状态到x，最后恢复时直接在local_intr_restore宏里取反不就行了，所以下面我就来反驳一下我当初的想法：

首先举个例子，假设此时"local_intr_restore"宏采用的是“**取反**”逻辑。举的例子中：“关中断操作可以嵌套”：

```
void foo() {
    bool flag;
    local_intr_save(flag);   // 保存并关闭中断（假设原来是开的）
    bar();                   // bar里也可能关中断
    local_intr_restore(flag);
}

void bar() {
    bool flag2;
    local_intr_save(flag2);  // 如果当前已经是关的，这里再关一次
    ...
    local_intr_restore(flag2);
}
//以foo()函数为主体，用flag和flag2变量来记录当前中断状态。
```

假设最开始中断是开的（ON=1）：

步骤	状态	说明
初始	ON	系统中断允许
foo 调用 `save(flag)`	OFF	flag=1, 手动关闭中断
bar 调用 `save(flag2)`	OFF	flag2=0，因为当前中断已经是关闭的
bar 调用 `restore(flag2)`	❗取反策略→ON	啪，中断提前被打开
foo 调用 `restore(flag)`	❗取反策略→OFF	啪，中断又被错误关掉

原本逻辑是执行完foo（）之后把中断打开，但现在若使用“取反”逻辑，最后中断又被关闭，逻辑错误。

所以，不能简单取反，必须要在save宏中记录初始的中断状态，并在restore宏中根据保存的状态进行恢复。

2、深入理解不同分页模式的工作原理（思考题）：

题干：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。这在操作系统中的分页机制下，是实现虚拟内存与物理内存之间映射关系非常重要的内容。

- 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。

回答：

get_pet()函数中两段类型相似的代码分别是处理页目录项（PDE）和0页表项（PTE）（页目录：存放指向页表的指针的结构），两段代码都是先进行有效位检查（分别对pgdir中的PDE和PDE中的PTE检查），若无效且允许创建，就分配物理页。因为：使用页目录(PDE)和页表（PTE）不同的分页模式时，基本结构、功能一样，只是操作数的数据结构和层级不同，代码相似使得易于理解和维护。在sv32、sv39、sv48中，它们的页目录项（PDE）和页表项（PTE）都有相似的结构，都基于有效位和物理页编号。

- 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：

get_pte()负责同时查找和可能创建对应于特定线性地址（LA）的页表项（PTE）。我认为这种写法是好的：

- 简化了使用流程：首先对于用户，只需要调用get_pte()一个函数就可以，一个函数中就实现获取页表项、创建页表项两个功能，简化了使用流程；
- 减少函数调用开销：减少函数调用的次数，在性能上会有所上升。