lab4实验报告：进程管理

实验内容概述

本实验旨在理解和实现操作系统中的进程（内核线程）创建、初始化、资源分配和上下文切换机制。主要涉及 alloc_proc、do_fork 和 proc_run 等关键函数的实现。

实验内容

练习1：分配并初始化一个进程控制块（需要编码）

代码编写

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。

alloc_proc 函数的作用在于，当你创建一个新进程的时候，首先需要申请一块内存来存放 proc_struct 结构体。但因为 kmalloc 分配的内存可能包含残留的垃圾数据，所以我们需要一个量身定制的初始化函数，把它清洗成一个干净、合法但尚未填充具体内容的空壳，确保它不会被随机数据干扰。这一步做完后，才轮到 do_fork() 进场给它赋予具体的资源和身份。

所以我们要先看 proc_struct 结构体：

代码块

```
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
```

```
17  };
```

具体的初始化逻辑非常直观，主要是为了设置一个安全的默认状态：

1. **基本身份**：state 设置为 PROC_UNINIT，表示它还处于原始状态；pid 设为 -1，意味着还没分到合法的 pid ; runs 初始化0，因为它还一次没跑过。
2. **关键资源**：需要特别注意的是 pgdir（页表基址）的设置。因为我们创建的是内核线程，它们共享内核的地址空间，所以必须将它初始化为内核页表的物理基址 boot_pgdir_pa ，否则后续无法正确访问内核空间。
3. **清理工作**：其他的指针（如 mm , parent , tf ）统统设为 NULL，context 和 name 数组用 memset 清零。
4. **数据结构**：最后把 list_link和 hash_link 调用 list_init 进行初始化，防止链表操作时出现野指针异常。

代码块

```
1   if (proc != NULL) {
2           // 1. 设置基本状态
3           proc->state = PROC_UNINIT;
4           proc->pid = -1;
5           proc->runs = 0;
6
7           // 2. 设置栈与调度标志
8           proc->kstack = 0;
9           proc->need_resched = 0;
10
11          // 3. 设置亲缘关系与内存管理
12          proc->parent = NULL;
13          proc->mm = NULL; // 内核线程共享内核内存，初始为空
14
15          // 4. 初始化上下文 (Context)
16          memset(&(proc->context), 0, sizeof(struct context));
17
18          // 5. 设置中断帧与页表
19          proc->tf = NULL;
20          proc->pgdir = boot_pgdir_pa; // 关键：内核线程使用内核启动页表
21
22          // 6. 其他标志位与名称
23          proc->flags = 0;
24          memset(proc->name, 0, sizeof(proc->name));
25
26          // 7. 初始化链表节点
27          list_init(&(proc->list_link));
28          list_init(&(proc->hash_link));
```

```
29      }
```

问题回答

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）

我们先看源代码中对于 context 的定义：他存储了14个关键寄存器的值， ra 存储着进程的函数返回地址，当从别的进程切回来的时候，就会从ra（上次切走的地方）继续执行；sp 存储内核栈顶指针；剩下12个是被调用者保存的寄存器 。总的来说它就是起到一个切换进程时保存现场和恢复现场的作用。我们回忆上一次实验的中断处理的 trap_frame ，它相当于为整个进程拍了一个快照，存储的寄存器值要比这个多很多，初步能看出来二者功能上的差别。

代码块

```
1   struct context
2   {
3       uintptr_t ra;
4       uintptr_t sp;
5       uintptr_t s0;
6       uintptr_t s1;
7       uintptr_t s2;
8       uintptr_t s3;
9       uintptr_t s4;
10      uintptr_t s5;
11      uintptr_t s6;
12      uintptr_t s7;
13      uintptr_t s8;
14      uintptr_t s9;
15      uintptr_t s10;
16      uintptr_t s11;
17  };
```

接下来来看 tf 结构体的具体定义：

代码块

```
1   struct trapframe
2   {
3       struct pushregs gpr;
4       uintptr_t status;
5       uintptr_t epc;
6       uintptr_t badvaddr;
```

```
7       uintptr_t cause;
8   };
```

gpr 是所有通用寄存器， status 是特权级状态寄存器，epc 记录着 trap 发生时的指令的地址，badvaddr 在发生访存错误时记录出错的地址，cause 记录错误原因编码。

当 trap 发生时，CPU 会强制从用户态切换到内核态，并跳转到预设的 trapentry.S 中的 __alltraps 地址。

1. 保存现场 : __alltraps 的首要任务就是 在内核栈上构建一个 trapframe ，把 trap 发生瞬间的所有 CPU 寄存器（通用寄存器 + CSRs）都保存进去。
2. 传递信息 : 然后， __alltraps 会将这个栈上 trapframe 的地址作为参数，调用 C 语言实现的 trap() 函数。 proc_struct 中的 tf 成员变量就会指向这个地址。
3. 处理与返回 : C 函数 trap() 根据 tf->scause 判断 trap 类型，进行相应处理（如系统调用、缺页处理）。处理完毕后，返回到 __alltraps 。 __alltraps 再从 trapframe 中恢复所有寄存器，执行 sret 指令返回用户态，用户进程从 tf->sepc 处恢复执行，仿佛什么都没发生过。

成员变量	含义	在本实验中的作用
struct context context	进程上下文（Context）结构体，保存了进程在非中断/陷阱状态下被切换出去时，CPU 寄存器的值。	用于 进程切换（switch_to 函数）。它保存了进程在用户态或内核态执行时的关键寄存器（如栈指针 sp、返回地址 ra 等），使得进程在下次被调度运行时能够从上次停止的地方继续执行。
struct trapframe *tf	陷阱帧（Trapframe）指针，指向一块内存区域，用于保存进程从用户态/内核态陷入内核（如发生中断、异常或系统调用）时，CPU 寄存器的值。	用于 中断/异常/系统调用返回。它保存了进程在陷入内核前的状态，使得内核在处理完中断/异常后，能够通过 sret 指令恢复进程的执行现场，正确返回到被中断的代码处。在创建新进程时，tf 被初始化，用于设置新进程第一次执行时的起始状态。

练习2：为新创建的内核线程分配资源（需要编码）

代码编写

do_fork 函数用于创建当前内核线程的一个副本。你需要完成在 kern/process/proc.c 中的 do_fork 函数中的处理过程。

首先我们分析kernel_thread函数的上下文：

1. 在init函数中调用了proc_init函数
2. 在proc_init函数中初始化了进程链表proc_list和hash表
3. 使用alloc_proc()初始化了一个初始化模板进程控制块PCB，在alloc_proc()中，主要是分配一个进程控制块的空间，然后初始化化该进程控制块的各个成员变量的值。大部分成员变量的值都初始化为0，特别的是proc->state = PROC_UNINIT;//未初始化、 proc->pid = -1;//未准备好状态、proc->pgdir = boot_pgdir_pa;//根页表的物理地址。
4. 设置idleproc的PCB属性，重要的是设置idleproc->state = PROC_RUNNABLE;表示其随时可被其他线程切换。
5. 使用kernel_thread(init_main, "Hello world!!", 0);初始化另一个可用线程，返回的变量是新初始化线程的pid。
6. 利用在lab2学习到的技巧，通过结构体成员变量的地址连续性，可以通过链表访问指定结构体的其他成员变量和找到结构体起始地址。我们可以获得新线程的PCB

我们再来分析kernel_thread函数在做什么？

总的来说kernel_thread函数是创建一个新的内核线程，该内核线程会执行我们指定的任务。

1. 在kernel_thread函数我们先构造一个trapframe用于新线程，用 trapframe 伪造 CPU 的现场，让新线程一开始就跳到指定函数执行。
2. 设计tf.gpr.s0为指定函数，tf.gpr.s1为指定函数的参数
3. 设计tf.status，读当前 CPU 状态，内核模式下运行，返回后开启中断，但当前中断关闭。因为我们在创建新的进程，所以关闭中断，但我们设计SPIE为1，当从中断恢复时，执行sret，硬件会自动设施SIE=SPIE恢复中断关闭前的中断使能状态,跳转到到trapframe的ecp执行。
4. **设置tf.epc = (uintptr_t)kernel_thread_entry**;这其实是练习4的内容，恢复中断后，跳转到ecp的位置开始执行，即跳转到kernel_thread_entry开始执行。kernel_thread_entry内容就是在我们伪造 CPU 的现场中，执行设计的任务函数。
5. kernel_thread调用do_fork函数，传入之前伪造的中断帧，即cpu现场。

现在我们看do_fork函数的实现

do_fork 创建一个新进程：先分配 PCB，再分配内核栈，再复制/创建地址空间，然后构造 trapframe 和 context，最后设置父子关系和 PID，并把新进程加入全局调度结构，置为可运行状态。

1. alloc_proc()：创建并初始化 PCB（proc_struct）这其实是练习2的内容，分配一个新的 PCB 结构、初始化其状态、pid = -1 等基本字段、设置根页表物理地址。
2. setup_kstack()：为子进程分配内核栈。除了0号线程在PCB初始化时直接使用uCore启动时设置的内核栈，但其他现成的内核栈必须要先分配。使用 alloc_pages() 分配 8KB。
3. copy_mm()：建立地址空间 mm.由于此处是内核线程创建，地址空间 mm为**null**。注意：`mm_struct` 描述 “用户进程的用户态虚拟地址空间”，内核线程永远运行在内核地址空间，不需要也不会进入用户态，所以不需要用户地址空间 mm，并且内核栈独立，但是内核地址空间共享。
4. copy_thread()：构造 trapframe（tf）与 context。这应该最关键的函数，设置当前线程的中断帧和上下文。中断帧用于恢复中断，返回CPU执行现场，就像CPU没有被打断过，而context的用于进程切换。我们重点阐述copy_thread()干了什么：
   a. 把 trapframe 放到内核栈的顶部。
   b. 约定fork() 的子进程 a0 返回 0
   c. kernel_thread 提供的 trapframe 复制到子进程。（kernel_thread设置）
   d. **设计trapframe的sp如果为内核真正执行时的cpu现场，针对内核线程就是tr本身，如果是用户态，执行用户栈**
   e. **设计context的sp为内核栈，该进程的CPU现场设置。**
   f. **设计context的ra返回地址为指定的forkret函数**
   g. 子进程第一次运行时，会返回context的ra寄存器指向的地址，即forkret，子进程第一次运行将从 forkret→trapframe→sret 开始。
5. 设置父子关系与新 PID，proc->pid = get_pid();
6. 将新进程插入内核进程表结构，即全局 proc_list 链表和按照按 PID 管理的哈希表。
7. 将进程置为就绪态（PROC_RUNNABLE）。
8. 管理进程计数，返回进程pid。

注意：根页表的物理地址的得来，在进入init内核初始化函数时，使用汇编语言开启sv39虚拟内存模式，并且初始化页表，汇编代码如下

代码块

```
.section .data
    # 由于我们要把这个页表放到一个页里面，因此必须 12 位对齐
    .align PGSHIFT
    .global boot_page_table_sv39
# 分配 4KiB 内存给预设的三级页表
# 内核提前准备好的一个 4KB 页表（boot_page_table_sv39是最高级页表）
boot_page_table_sv39:
```

```
8       # 0xffffffff_c0000000 map to 0x80000000 (1G)
9       # 前 511 个页表项均设置为 0 ，因此 V=0 ，意味着是空的(unmapped)
10      .zero 8 * 511
11      # 设置最后一个页表项，PPN=0x80000，标志位 VRWXAD 均为 1
12      # U = 0用户态不能访问
13      # 该页被映射为可读可写可执行的有效内核页
14      # .quad表示往当前内存位置写入一个 8 字节（64-bit）数
15      # PTE是64位，物理地址 = 0x80000000+10位信号位
16      # 所示这条操作将虚拟地址范围的最后一个大大页（512项）PDE 指向 物理地址 0x80000000，
    分配1GB
17      .quad (0x80000 << 10) | 0xcf # VRWXAD
18
19      .global boot_hartid
```

定义全局变量boot_page_table_sv39，其是三级页表的根页表，大小是4K，该4K空间的值被初始化为前511项页表项为全0，最后一项页表项映射物理地址0x8000000000000000，即DBT探测到的可分配物理内存的起始地址。

在物理内存管理初始化中，有如下代码

代码块

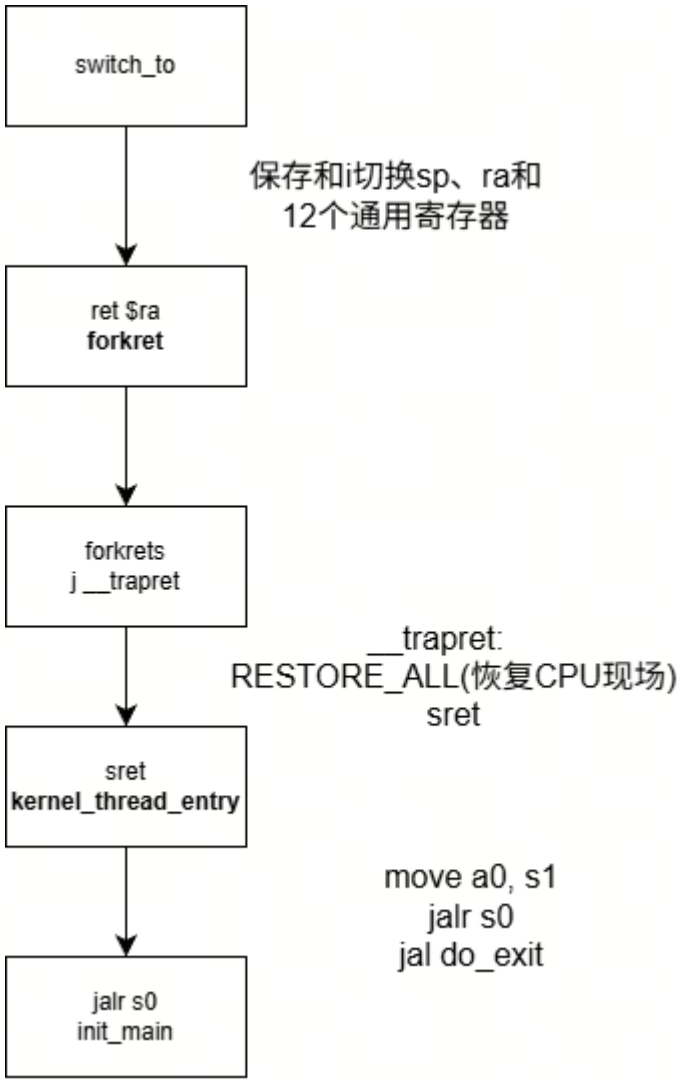
```
1       extern char boot_page_table_sv39[];
2       // boot_pgdir_va 是指向内核启动时使用的根页表（boot_page_table_sv39）的内核虚拟地
    址指针
3       boot_pgdir_va = (pte_t *)boot_page_table_sv39;
4       // __m_kva - va_pa_offset;  计算出对应的物理地址
5       boot_pgdir_pa = PADDR(boot_pgdir_va);
```

计算得到该页表的虚拟地址，但是CPU在开启虚拟地址模式后，只能运行在虚拟空间，在kernel.ld文件中，可以发现内核是被加载到虚拟空间的高地址的，所以此处的boot_page_table_sv39本来就在虚拟空间上，通过PADDR可以得到对应的物理地址。将根页表的物理起始地址写入PCB，用于进程切换。

综上，**do_fork() 分配了内核栈，构造了trapframe ，构造了context；第一次调度新进程时，先恢复 context 进入 forkret，然后通过中断恢复机制恢复 trapframe，从而进入新任务的执行入口。**

创建一个内核线程需要分配和初始化多种资源，而 alloc_proc 仅分配了记录进程基本信息的 PCB，并未真正建立运行所需的环境。uCore 通过 do_fork 执行真正的内核线程创建：它以当前线程为模板，复制其执行状态，并为新线程分配独立的资源（如内核栈、trapframe、context、PID 等），从而生成一个在逻辑上与原线程行为相同但独立执行的新内核线程。

实验4大致流程图



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。

1.Core 的 `pid` 分配器在系统中最多同时存在 4096 个进程，但可以使用 8192 个不同的 PID 空间，因此 **PID 分配器在回绕（wrap around）之前总能找到未使用的 PID，而不会遇到“新 PID 与当前某个活跃进程重复”的情况**。换句话说，虽然 PID 最终会循环，但因为“可用 PID 数量（8192）远大于系统中可同时存在的最大活跃进程数量（4096）”，所以当 PID 计数器回到开头时，之前使用过但已经退出的进程对应的 PID 都已经被回收，不会和仍然存活的进程冲突。

代码块

```
#define MAX_PROCESS 4096
#define MAX_PID (MAX_PROCESS * 2)
```

2.**每次 PID 计数器递增到达上限需要回绕时，代码会强制重新扫描整个进程链表（通过 `goto inside → goto repeat`），并从 1 开始重新寻找一个 未被占用的 PID；在扫描过程中，如果某个 PID 已被占用，则继续向上递增，且通过维护 `next_safe`（记录所有存活进程中 大于 last_pid 的最小 PID），确保搜索范围始终落在“空闲区间”；若即将超出该区间，也会强制回绕并重新全表扫描。**由于 MAX_PID（8192）远大于 MAX_PROCESS（4096），即使回绕发生，也总能在 1～

MAX_PID 范围中找到某个“当前无任何进程使用的 PID”。通过维护last_pid和next_safe之间的可用安全区间和从头扫描进程控链表排除冲突pid，可以解决当出现回环时，pid回到1，而其他在线线程pid在1-4096随机分布分布的问题。

确保回绕不会导致 PID 与当前存活进程冲突，从而不会出现重复 PID 的问题。

代码块

```
get_pid(void)
{
    static_assert(MAX_PID > MAX_PROCESS);
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    //next_safe下一次搜索 PID 时允许的搜索上限，用于减少遍历，加速查找，找 PID 时有效搜索范围的上限
    //last_pid = 上一次分配的 PID（继续递增从它开始找）
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    //如果last_pid达到上限，则回绕到1，并且重新开始扫描，goto inside
    if (++last_pid >= MAX_PID)
    {
        last_pid = 1;
        goto inside;
    }
    // 如果last_pid大于了next_pid，说明目前的安全区已经不可靠了，需要重新扫描进程链表，更新next和last
    if (last_pid >= next_safe)
    {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        // 从头寻找
        while ((le = list_next(le)) != list)
        {
            // 从结构体找到具体线程
            proc = le2proc(le, list_link);
            // 如果存在进程和当前的可用pid冲突
            if (proc->pid == last_pid)
            {
                // 尝试用下一个 pid
                // 如果下一个pid不在安全区内，则需要跟新next或者last回环然后重新扫描
                if (++last_pid >= next_safe)
                {
                    if (last_pid >= MAX_PID)
                    {
                        last_pid = 1;
                    }
```

```
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            // 当前进程的 PID 大于 last_pid，且比当前的 next_safe 还小
            // 在所有 pid > last_pid 的进程里，找出 最小的那个 pid，记为 next_safe
            else if (proc->pid > last_pid && next_safe > proc->pid)
            {
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
    // 综上，一旦 goto repeat，
    //说明 last_pid 换成了一个新候选值，必须重新检查整个进程链表，确保它没被占用。
}
```

练习3：编写 proc_run 函数（需要编码）

代码编写：

proc_run 用于将指定的进程切换到 CPU 上运行。

该函数负责将 CPU 的控制权从当前的进程/线程 (`current`) **安全、原子地**转移到目标进程/线程 (`proc`)。

代码块

```
void proc_run(struct proc_struct *proc)
{
    //首先检查目标进程是否就是当前已经在运行的进程
    if (proc != current)
    {
        // LAB4:EXERCISE3 YOUR CODE
        /*
         * Some Useful MACROs, Functions and DEFINEs, you can use them in below implementation.
         * MACROs or Functions:
         *   local_intr_save():        Disable interrupts
         *   local_intr_restore():     Enable Interrupts
         *   lsatp():                   Modify the value of satp register
         *   switch_to():              Context switching between two processes
         */
        // 1. 禁用中断，确保上下文切换的原子性
        //切换完成之前不会被时钟中断或其他中断打断
        bool intr_flag;
```

```
18          local_intr_save(intr_flag);
19
20          // 2. 更新 current 指针
21          //将当前进程保存到局部变量 prev
22          //将全局指针 current 更新为目标进程 proc
23          struct proc_struct *prev = current;
24          current = proc;
25
26          // 3. 如果新进程有独立的地址空间（mm != NULL），则切换页表
27          // 在 RISC-V 中，这通过修改 satp 寄存器来实现。
28          if (current->mm) {
29              //将新进程的页目录基地址（current->pgdir）写入 satp 寄存器。
30              //修改完后使用新进程的页表来进行虚拟地址到物理地址的转换
31              //实现隔离和切换
32              lsatp(current->pgdir);
33          }
34
35          // 4. 执行上下文切换
36          // switch_to 会将 prev 的上下文保存到 prev->context，
37          // 并从 current->context 恢复上下文，跳转到 current 之前暂停的位置。
38          switch_to(&(prev->context), &(current->context));
39
40          // 5. 恢复中断状态
41          local_intr_restore(intr_flag);
42      }
43  }
```

***内核线程** (`idleproc`, `initproc` 等) 通常 **`mm` 为 NULL**，表示它们共享内核的地址空间，不需要切换页表。

***用户进程** 的 `mm` 字段不为 NULL，表示它们有独立的虚拟地址空间

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?

一共创建了两个内核线程，其中**线程1(`idleproc`)**负责启动和调度，并将 CPU 让给了线程 2。**线程 2 (`initproc`)**成功获得了 CPU，并执行了它的初始化代码，通过日志可以证明它的存在和运行。

1、内核启动入口 (kern_init)

首先，我们从内核的 C 语言入口 `kern_init` 开始，我们知道这个是为了按顺序初始化所有的核心子系统。可以看到 `kern_init` 通过调用 `proc_init` **准备了线程环境**，然后将控制权交给 `cpu_idle` ，**启动了调度器**。

代码块

```
    pmm_init(); // init physical memory management

    pic_init(); // init interrupt controller
    idt_init(); // init interrupt descriptor table

    vmm_init();  // init virtual memory management

    proc_init(); // init process table

    clock_init();  // init clock interrupt
    intr_enable(); // enable irq interrupt

    cpu_idle(); // run idle process
```

2、`proc_init` 函数创建了第一个线程 `idleproc` (PID 0)

`dleproc` 是系统的**空闲线程**，也是**第一个**内核线程。它不会通过正常的 `fork` 流程创建，而是直接在内存中配置，以便接管启动后的 CPU 控制权。

代码块

```
// alloc_proc分配 PCB 结构体
if ((idleproc = alloc_proc()) == NULL)
{
    panic("cannot alloc idleproc.\n");
}
// ... (中间是 alloc_proc 的自检代码，确保 alloc_proc 工作正常) ...

// 设置 PID 为 0，状态为 可运行
idleproc->pid = 0;
idleproc->state = PROC_RUNNABLE;

//将其内核栈 (kstack) 指向 bootstack。
//这使得 idleproc 能够接管内核启动时使用的栈空间，从而成为 CPU 上第一个实际的线程实体。
idleproc->kstack = (uintptr_t)bootstack; // 使用启动栈
//显式设置调度标志标志，确保 kern_init 结束时调用 cpu_idle 时，
//会立即触发第一次调度，将控制权让给 initproc。
idleproc->need_resched = 1;
set_proc_name(idleproc, "idle");
nr_process++;
```

在 `proc_init` 函数开始执行时，CPU 仍然在执行内核的**启动代码**，使用的是**启动栈**（`bootstack`）。此时，系统还没有正式的线程身份。

```
// ... 初始化 idleproc 的 PCB ...
idleproc->kstack = (uintptr_t)bootstack; // ① 将 PCB 与启动栈关联
// ...
current = idleproc; // ② 宣布：我（CPU）现在的身份是 idleproc!
```

当第 ② 步执行后，运行在 CPU 上的 C 代码（仍然是 `proc_init`）通过修改 `current` 指针，将自己的身份正式设定为 **`idleproc`**。

3、在 `idleproc` 成为 `current` 线程后，它立即执行 `proc_init` 的下一条语句来创建 `initproc` (PID 1)。

`initproc` 是第一个通过标准的**进程/线程创建机制**（封装的 `do_fork`）产生的线程。它负责执行高层初始化任务（例如启动用户 shell）。

代码块

```
//调用kernel_thread（）来创建新的线程，入口函数是init_main，
//底层执行do_fork，为 initproc 分配新的 PCB、新的内核栈，并将其状态设置为 PROC_RUNNABLE
int pid = kernel_thread(init_main, "Hello world!!", 0);
if (pid <= 0)
{
    panic("create init_main failed.\n");
}
//通过返回的 pid 找到新创建的 proc_struct
//并将其保存到全局变量 initproc 中，然后命名为 "init"。
initproc = find_proc(pid);
set_proc_name(initproc, "init");
```

4、两个线程的交替运行 (`cpu_idle` & 调度)

`cpu_idle` 函数是系统中第一个内核线程 **`idleproc` (空闲进程/线程)** 在完成初始化后，实际执行的主循环。当 `idleproc` 运行时，这表明**当前没有其他高优先级的任务可供运行**。它不执行任何实际工作，而是持续循环等待，一旦有其他线程准备就绪，它会立即调用 `schedule()` 将 CPU 让给新线程。

代码块

```
// cpu_idle - at the end of kern_init, the first kernel thread idleproc will do below works
void cpu_idle(void)
{
    //idleproc 永远不会退出，它会持续运行，直到系统关闭。
    while (1)
    {
```

```
        //检查当前正在运行的线程（即 idleproc 自己）的 need_resched 标志。
        if (current->need_resched)
        {
            //如果 need_resched 为真，则调用调度器。
            schedule();
        }
    }
}
```

(1) `idleproc` 运行时发现 `need_resched = 1`。

(2) `schedule()` 调用 `proc_run(initproc)`，执行上下文切换。

控制权从 `idleproc` 转移到 `initproc`。

(3) `initproc` 开始执行它的入口函数 `init_main`。

* **`need_resched` 为什么会设置为 1？** 在 `proc_init` 中，`idleproc` 的这个标志被显式设置为 1；或者当其他线程（如 `initproc`）准备就绪时，内核也会设置该标志。

代码块

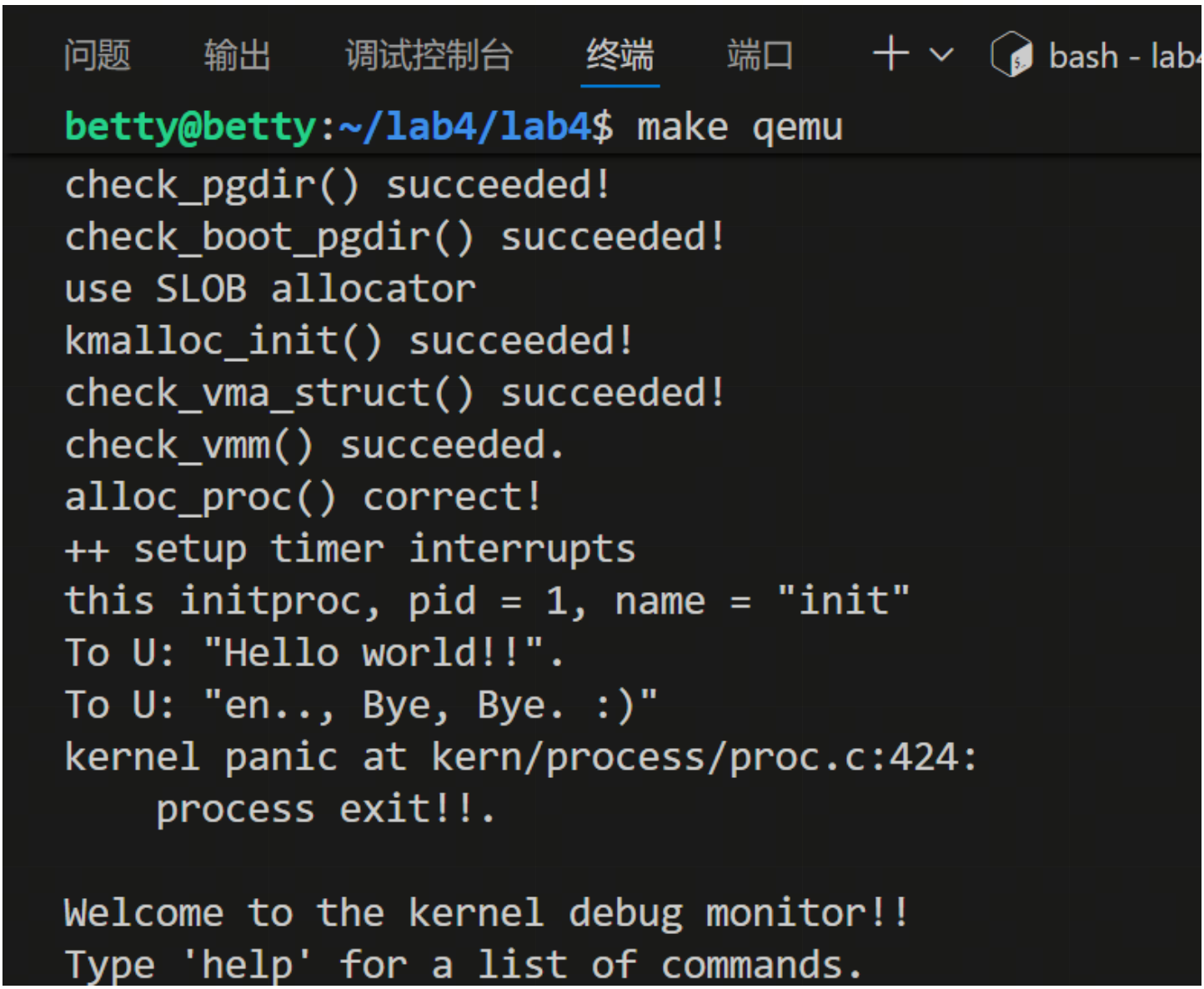
```
void
schedule(void) {
    //1、准备和初始化
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    //禁用中断，防止在调度过程中被打断
    local_intr_save(intr_flag);
    {
        //清除当前进程的重新调度标志，因为它现在正在执行调度。
        current->need_resched = 0;
        //2、选择下一个可运行进程
        //它遍历全局进程列表 proc_list，寻找第一个处于 PROC_RUNNABLE 状态的进程。
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
        } while (le != last);

        //3、如果没有找到可运行进程，则选择 idleproc 作为下一个运行的进程。
        if (next == NULL || next->state != PROC_RUNNABLE) {
```

```
27              next = idleproc;
28          }
29          next->runs ++;
30          if (next != current) {
31              //4、切换到下一个进程
32              proc_run(next);
33          }
34      }
35      local_intr_restore(intr_flag);
36  }
```

根据调度可知，会在全局进程列表中选择`initproc`，然后执行`proc_run(initproc)`函数。

5、日志显示



`this initproc, pid = 1, name = "init"`表明内核成功创建了**`initproc`**（初始化线程），其 PID 为 1。这是系统中的第二个线程（第一个是空闲线程`idleproc`，PID 0），它负责执行系统的更高层初始化任务。`To U: "Hello world!!"`和`To U: "en.., Bye, Bye.`

`:)"` 这几行是 `initproc` 线程中入口函数 (`init_main`) 内部通过 `cprintf` 打印出来的**预期输出**。这说明 `initproc` 成功被调度并执行了它的逻辑代码，表明进程切换机制工作正常。

日志**证明了线程 2 (`initproc`) 成功地被创建、被调度，并在线程 1 (`idleproc`) 之后获得了 CPU 的控制权并运行。** 这证实了系统已经成功从单线程内核启动环境，过渡到了有两个独立内核线程（ `idleproc` 和 `initproc` ）的多任务环境。

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?

说明： `local_intr_save(intr_flag);` 和 `local_intr_restore(intr_flag);` 是一对用于原子地控制中断状态的宏。

SIE是控制 CPU **是否响应**来自外部设备或时钟的**所有中断**的全局开关（1时CPU允许中断发送、0时CPU忽略所有中断请求）。flag则是内存中保存的“历史记录”。

`intr_flag` 和 SIE 的关系是通过 `local_intr_save` 宏建立的：

- **保存 (`local_intr_save`):**
  - 读取 SIE 位的值，并将其赋值给 `intr_flag` 。
  - 清除 SIE 位（即关闭中断）。
- **恢复 (`local_intr_restore`):**
  - 检查 `intr_flag` 的值。
  - 如果 `intr_flag` 显示原始状态是开启的，则设置 SIE 位（即重新开启中断）。

(1)其中 `local_intr_save(intr_flag)` 的作用是保存当前的中断状态，然后禁用中断。

代码块

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

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

```
13
14  void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }
```

- 保存状态：首先通过读取当前 CPU 的全局中断使能状态。`intr_flag = read_csr(sstatus) & SSTATUS_SIE;` 将 `sstatus` 寄存器中的 **SIE** 位保存到局部变量 `intr_flag` 中。这样就知道进入临界区前中断是开还是关。
- 禁用中断：接着清除 `sstatus` 中的 SIE 位。**立即禁用** CPU 对外部中断的响应。CPU 进入临界区。

(2) `local_intr_restore(intr_flag)` 的作用是根据保存的 `intr_flag` 恢复原始的中断状态。

代码块

```
1   #define local_intr_restore(x) __intr_restore(x);
2
3   static inline void __intr_restore(bool flag) {
4       if (flag) {
5           intr_enable();
6       }
7   }
8
9   /* intr_enable - enable irq interrupt */
10  void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }
```

- 检查标志位：检查 `intr_flag` 的值。如果 `intr_flag` 为真（原始状态是开启的），则开启中断。
- 恢复状态：`set_csr(sstatus, SSTATUS_SIE);` 恢复 `sstatus` 中的 SIE 位。如果原始状态是开启的，则重新开启中断。如果原始状态是关闭的，则保持关闭状态。

*不直接使用 `intr_disable()` 和 `intr_enable()` 的原因是为了确保**嵌套调用的安全性**和**原子性**。

`local_intr_save` 确保了从调用点到 `local_intr_restore` 之间的代码是一个**不可分割的临界区**。

2. 深入理解不同分页模式的工作原理（思考题）

get_pte() 函数（位于 kern/mm/pmm.c）用于在页表中查找或创建页表项。

问题1

· 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。

解释：

代码块

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
{
    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);
        #计物理页引用计数为1
        uintptr_t pa = page2pa(page);
        #获取该页物理地址
        memset(KADDR(pa), 0, PGSIZE);
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
        #构造页表项并填入上一级页表

    }
    pde_t *pdep0 = &((pte_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);
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

这两段代码之所以长得像，是因为处理中间级页表的逻辑时互通的，无论是 SV32（二级页表）、SV39（三级页表）还是 SV48（四级页表），它们的工作原理都是层级递进的。每一级页表项（PTE）通常指向下一级页表的物理页基址。在查找虚拟地址对应的最终物理页时，如果中间某一级页表不存在

（即 PTE_V 位为 0），我们就无法继续往下走。因此，为了建立映射，我们必须为缺失的中间层级分配一个新的物理页，清零，并将其物理地址填入当前级的页表项中。

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？

我觉得这种写法在目前的实验里没有问题。

get_pte()本身是一个查询工作，获得当前根页表对应虚拟地址的页表项，如果中间的页目录没有，它负责创建中间页表，但是并没有填充最后具体的物理页。get_pte本身是只是为完成最终物理页映射打下的铺垫，无论是读物理写还是写物理页，我们都需要一条完整的指向最终物理页的页表访问路径。但是当我们在访问一个物理页的时候，发现路径并不完整，就应该进行修复。

这种写法保证我们在访问一个物理页时无论是写还是读，它的页表查询路径是完整的，将页表路径填充和物理页映射解耦，保证映射的物理页一定是有可达路径的，而不是一个不能通过页表访问到的物理页。这样的操作更加安全，就像原子操作，绑定在一起，要么给我一个可用的 PTE 指针，要么报错，不会出现“中间断层”的状态，之后再调用比如page_create或者page_insert映射最终物理页。

同时这样的写法也可以减少重复代码、接口更简洁，不需要在编写程序时先判断是否有效，如果无效再调用函数创建映射。

同时这样写的好处是提高性能，一边查找页表，如果发现无效，则立刻创建页表，不需要反复调用函数从根页表开始重新层层查询。一次查找，按需创建，不需要查两次。

（总结一下就是如下观点）

1. **职责清晰与解耦**：`get_pte` 的核心职责是**“维护页表结构的完整性”。它将“中间级页表的创建（路径构建）”与“最终物理页的映射（内容填充）”解耦。它确保了当内核需要访问或修改一个页表项时，通往该项的各级页目录不仅存在，而且是连续有效的。
2. **逻辑原子性与安全性**：这种写法将“查询”与“按需分配”绑定为一个逻辑原子操作。这消除了“页表路径断裂”的风险，保证了任何返回的有效 PTE 指针，其背后的索引路径都是完整的，避免了访问不可达物理页的隐患。
3. **性能优化**：在多级页表架构中，页表游走（Page Table Walk）是昂贵的内存操作。该写法通过一次遍历完成查询与必要的分配，避免了“先查询失败、再重新从根遍历分配”的冗余开销，显著提高了内存管理子系统的效率，同时也让代码接口更加简洁。

知识点总结与对比

实验中重要的知识点及其与 OS 原理的对应

1.父进程与子进程

在进程控制块 `proc_struct` 中，内核会为每个进程保存一个指向其父进程的指针 `parent`。整个操作系统中，只有内核在启动时创建的第一个进程（通常为 `idle` 或第 0 号进程）没有父进程；其后

的所有进程（包括用户进程和内核线程）都是由某个已有进程通过 `do_fork()` 创建，因此形成了天然的父子关系。这种父子关系以树的形式存在，即所有用户进程构成一棵**进程树**，而此结构能够帮助操作系统对进程生命周期进行管理与协调。

父子关系不仅是一种“来源关系”，更是管理机制的基础。父进程对子进程拥有若干特殊职责和权限，例如：

1. **资源回收（reap）**：当子进程退出后会变成僵尸进程，必须由父进程调用 `wait()` / `waitpid()` 来收集退出码并释放 PCB 等资源，否则资源会泄漏。
2. **信号传递与控制**：在类 Unix 系统中，父进程可以向子进程发送信号（如终止、暂停、继续运行等），用于过程控制与作业管理。
3. **继承与环境传递**：子进程会继承父进程的文件描述符、地址空间结构、当前工作目录等初始环境，并在此基础上运行自己的代码。
4. **异常情况下的收容**：如果父进程提前退出，内核会把孤儿进程重新挂到 `init` 或 `idle` 进程下，由这些系统级进程继续承担资源回收的责任，以保证系统的稳定性。

因此，父子进程不仅体现了进程创建来源，还构成了进程生命周期管理、资源回收、安全控制以及系统稳定性的核心机制。通过维护这棵“进程树”，内核能够高效、可靠地追踪所有进程并在适当时进行清理、调度和控制。

state：当前进程状态，在 uCore 中定义了四种：

状态	含义
PROC_UNINIT	未初始化
PROC_SLEEPING	睡眠（等待某事件）
PROC_RUNNABLE	可运行（等待CPU）
PROC_ZOMBIE	僵尸态（等待父进程回收）

2.trapframe 与 context 的区别

trapframe 是“用户态 → 内核态”时保存的 CPU 寄存器现场。它记录的是“当前用户态程序正在执行的位置与所有寄存器”，用于在系统调用或中断完成后恢复用户态程序继续执行。

context 是进程在内核态调度时保存的“内核态寄存器现场”。 它用于记录调度器需要恢复的最少量寄存器，使 CPU 能继续执行上一次被切换出去的内核调度点。context 用于进程之间切换。context **只保存内核态的部分寄存器，ra、sp和被调用者寄存器。**context不需要恢复CPU执行现场，那些信息存储在中断帧中，只需要保存进程切换时要用的寄存器。

内核栈是进程在内核态执行的基础，也是保存 trapframe的物理空间。

3.PCB中存储的是根页表的物理地址

内核在虚拟空间上，我们在entry.s得到的oot_page_table_sv39根页表的地址起始地址其实是在虚拟空间中，但是为什么我们专门在内存管理系统初始化后，将虚拟地址转为物理地址，并且存在PCB中的根页表也是物理地址呢?

因为 CPU 的 satp 寄存器要求保存的是页表根的“物理地址（PPN）”，不能使用虚拟地址。RISC-V 的页表根放在寄存器：satp = MODE | ASID | PPN，这里的 PPN 是物理页号，我们通过物理页加上偏移量可以访问到具体的物理空间中的地址。

其次，页表中存储的都是物理地址而不是虚拟地址。这个就像先有鸡还是先有蛋的问题， 虚拟地址没办法用于页表切换，虚拟地址必须通过页表才能转换成物理地址，如果页表又存虚拟地址，那虚拟地址访问虚拟地址不能得到物理地址，构成死循环。

内核自己访问页表时才需要虚拟地址，CPU 不能用虚拟地址，只能用物理地址，CPU 切换页表用物理地址。

4.0号线程在PCB初始化时直接使用uCore启动时设置的内核栈，但其他现成的内核栈必须要先分配。

因为 0 号线程在内核启动前就必须运行，而此时内存管理器和页分配器尚未初始化，无法动态分配内核栈；其他线程创建于内核已初始化之后，所以必须通过分配器动态分配自己的内核栈。

在entry.s中可以发现第一个内核栈是用汇编代码预留的8K空间。

代码块

```
1   .section .data
2       # .align 2^12
3       .align PGSHIFT
4       # 栈低
5       .global bootstack
6   bootstack:
7       .space KSTACKSIZE
8       # 栈顶 sp
9       .global bootstacktop
```

OS 原理中很重要但在实验中没有对应上的知识点

1.在现代处理器设计中，在进程切换时，代码会先关闭中断，操作结束后再打开中断，这点是和课堂和设计实际不符合的，请找到实验中具体的地方。

使用如下函数设置SIE关闭中断或者打开中断

代码块

```
1   void proc_run(struct proc_struct *proc)
2   {
3       if (proc != current)
```

```
4       {
5           // LAB4:EXERCISE3 YOUR CODE
6           /*
7            * Some Useful MACROs, Functions and DEFINEs, you can use them in
    below implementation.
8            * MACROs or Functions:
9            *   local_intr_save():        Disable interrupts
10           *   local_intr_restore():     Enable Interrupts
11           *   lsatp():                   Modify the value of satp register
12           *   switch_to():              Context switching between two processes
13           */
14          bool intr_flag;
15          local_intr_save(intr_flag);           // 关中断，保证切换原子性
16
17          struct proc_struct *prev = current;
18          current = proc;
19
20          // 切换页表（riscv：加载 satp 为新进程页表根；pgdir 已是物理地址）
21          lsatp(proc->pgdir);                   // 如果你的平台需要 MAKE_SATP，可改
    成 lsatp(MAKE_SATP(proc->pgdir))
22          // riscv 切换 satp 会隐式刷新 TLB；若你有 sfence.vma 宏，也可在此调用。
23
24          // 上下文切换（只需切被调用者保存寄存器）
25          switch_to(&(prev->context), &(proc->context));
26
27          local_intr_restore(intr_flag);        // 开中断
28      }
29  }
```

uCore 主要假设单核，所有内核运行都在同一个 CPU 上执行，关中断就意味着 不会被其它线程或中断打断，从而可以保护临界区，避免竞态。但是在真正的现代系统中，即使关掉当前 CPU 的中断，其他 CPU 仍然可以访问同一个共享数据结构，关中断不能用来作为多核同步机制。现代系统使用更高级的**锁机制**来控制并发操作。

2.把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为**页换入换出**（page　swap in/out），但是这段我们目前的ucore中还没有实现。这种内存管理技术给了程序员更大的内存空间，从而可以让更多的程序在内存中并发运行。