操作系统 Lab4——进程管理

2313226 肖俊涛 2312282 张津硕 2311983 余辰民

密码与网络空间安全学院

实验目的

- 了解虚拟内存管理的基本结构，掌握虚拟内存的组织与管理方式
- 了解内核线程创建/执行的管理过程
- 了解内核线程的切换和基本调度过程

实验文件解析

1. **硬件层**：RISC-V CPU + 内存 + 中断控制器（PLIC）+ 时钟 + 串口/屏幕
2. **底层抽象：驱动 / SBI** → `kern/driver`
3. **核心内核机制**
   - 启动：`kern/init`
   - 内存管理：`kern/mm`
   - 中断和异常：`kern/trap`
   - 进程与调度：`kern/process` + `kern/schedule`
4. **辅助库**：`kern/libs` + 根目录 `libs`
5. **构建工具与链接脚本**：`tools/` + `Makefile`

Lab1/2/3 执行最小内核 + 内存管理 + trap，Lab4 在此基础上，**内核实现多个执行流（进程），并能来回切换**

---

kern/：内核主目录

`kern/debug/` —— 内核调试

- `assert.h`
  - 提供 `assert(cond)` 宏：

    ```
    assert(x != NULL);
    ```

  - 条件不满足时调用 `__panic()`，打印报错信息（文件、行号、函数）并停机。
  - 检查 `current != NULL`、`proc->state` 合法等。
- `panic.c`
  - 实现 `__panic(const char* file, int line, const char* fmt, ...)`
  - 功能：
    1. 打印“内核 panic 信息”
    2. 关闭中断
    3. 死循环或让 CPU 进入死状态（不会再返回）
- `kmonitor.c / kmonitor.h`
  - 内核监视器（类似一个简易 shell）。常见命令：
    - `help`
    - `backtrace`：调用 `kdebug.c` 解析函数名
  - 当内核 panic 或某处主动调 `kmonitor()`，你就能在 QEMU 里交互。
- `kdebug.c / kdebug.h / stab.h`
  - 负责从编译进内核的符号表里读出“某地址对应的函数/文件/行号”。
  - 在 `backtrace` 时：从当前栈帧向上走，逐层打印 `eip/ra` 对应的函数名。

当进程切换 / trap 返回实现错误时，用于记录调试信息

`kern/driver/` —— 硬件抽象 + 时钟 + 中断控制器

- `console.c / console.h`
  - 对底层串口/显示输出做了一层封装：
    - `cons_putc(int c)` / `cputchar()` 等。
  - `printf` 最终会走到这里。
  - **注意**：panic 信息、调试输出、kmonitor 的 echo 都依赖这个部分。
- `clock.c / clock.h`
  - 配置硬件时钟或通过 SBI 请求定时中断。
  - 提供函数：
    - `clock_init()`：设置时钟中断频率（比如 100Hz）
    - `clock_set_next_event()`：告诉硬件“下一次何时中断”
  - 中断发生时，在 `trap.c` 的 `interrupt_handler()` 里会识别“这是时钟中断”，然后：

```
ticks++;
if (ticks % TICK_NUM == 0) {
    sched_class->tick();
}
```

  - **Lab4 调度的关键触发点就在这里。**

- `intr.c / intr.h`
  - 提供 `intr_enable()` / `intr_disable()` 之类的接口。
  - 初始化中断向量、打开外部中断、软件中断等。
  - 内核在进行关键区操作（比如操作进程队列）时可能会关中断。
- `picirq.c / picirq.h`
  - 传统 x86 是 PIC/IOAPIC，这里是 RISC-V 下对应的中断控制层。
  - 目标：屏蔽硬件细节，统一成某种 `pic_enable()` / `pic_disable()` 风格。
- `dtb.c / dtb.h`
  - 解析 Device Tree Blob（dtb），从 QEMU 或真实板子里读出硬件配置信息：内存大小、外设等。
  - 有些初始化（比如物理内存边界）会参考 dtb 信息。、

时钟中断 → trap.c → schedule() → switch.S → 进程切换，其源头就是 `clock.c`。

`kern/init/` —— C 语言内核

- `entry.S`
  - 这是整个内核的**第一条指令。**
  - 做的主要事情：
    1. 设置临时栈指针 sp
    2. 关闭中断
    3. 初始化一些必要的寄存器（如 `gp`）
    4. 跳转到 C 函数 `kern_init()`
- `init.c`
  - 定义 `kern_init()`，相当于内核的“main（） 函数”

```
void kern_init(void) {
    cons_init();      // 初始化控制台
    print_kerninfo(); // 打印内核信息
    pmm_init();       // 物理内存管理
    vmm_init();       // 虚拟内存管理
    trap_init();      // 中断/异常机制
    clock_init();     // 时钟
    sched_init();     // 调度器
    proc_init();      // 进程系统初始化，建立第一个内核线程/idle
    cpu_idle();       // 进入调度循环
}
```

  - **Lab4 的 proc_init()/cpu_idle() 就是从这里进来的**

`kern/libs/` —— 内核内部的小型lib

- `stdio.c`
  - 实现内核态的 `cprintf()` 等格式化输出函数。
  - `printfmt.c`（在根 `libs/`）提供底层格式化逻辑，这里组合并输出到 console。
- `readline.c`
  - 在 kmonitor 中实现一行读取的功能，支持退格、编辑等。

`kern/mm/` —— 物理/虚拟内存管理

- `memlayout.h`
  - 描述整个虚拟地址空间布局：
    - `KERNBASE`：内核基地址
    - 用户空间的上界/下界
    - 各种段的位置（内核栈、内核映射区等）
  - Lab4 创建进程时会用到一些“用户/内核地址边界”宏。
- `mmu.h`
  - RISC-V 页表项格式、页大小、宏定义：
    - 页表级数
    - PTE 的 bit 定义（V/R/W/X/U 等）
    - `PADDR`、`KADDR` 等转换。
- `pmm.c / pmm.h`
  - 物理内存分配器“通用接口”，封装具体算法
  - 常见函数：
    - `page_alloc()` / `page_free()`
    - `boot_alloc_page()`（启动阶段）
  - Lab4：为进程分配页表、用户栈、代码段物理页都要通过这里间接完成。
- `default_pmm.c / default_pmm.h`
  - 具体的物理页分配算法实现（基于链表的空闲页管理）
- `vmm.c / vmm.h`
  - 维护“虚拟内存结构”（比如 `struct mm_struct`），并提供：
    - `pgdir_alloc()`：为进程建立页表
    - `page_insert(pgdir, pa, va, perm)`：在页表中插入映射
    - `do_pgfault()`：缺页异常处理（后续实验会加）
  - Lab4 里，**加载用户程序时：需要创建页表、把 ELF 段映射到相应虚拟地址**，用的都是 vmm 的接口。
- `kmalloc.c / kmalloc.h`
  - 类似内核态 malloc/free，实现内核堆。
  - 为内核数据结构提供动态分配（如进程队列、文件表等）

proc.c 负责进程，真正的内存空间建立靠 `pmm + vmm`。

`kern/process/` —— 进程管理 （核心）

`proc.h`：进程控制块（PCB）

```
struct proc_struct {
    int pid;                            // 进程号
    volatile int state;                 // 进程状态（RUNNING、SLEEPING、ZOMBIE...）
    int runs;                           // 被调度运行的次数
    uintptr_t kstack;                   // 内核栈的基地址
    struct mm_struct *mm;               // 该进程的虚拟内存描述结构
    struct context context;             // switch.S 用的寄存器上下文
    struct trapframe *tf;               // 用户态 <-> 内核态切换时保存的寄存器现场
    list_entry_t list_link;             // 进程加入全局链表
    list_entry_t run_link;              // 进程加入就绪队列
    // ... 优先级、父子关系等字段
};
```

`proc.c`：进程生命周期管理

- **初始化**：
  - `proc_init()`：
    1. 初始化全局进程链表、就绪队列
    2. 分配一个 `idleproc`（idle 进程，用来在没有其他进程时占用 CPU）
    3. 创建第一个用户进程 `initproc`
    4. 设置 `current = idleproc`
- **创建进程**：
  - `alloc_proc()`：只分配和初始化一个空的 `proc_struct`，设置默认字段（状态、pid=-1 等）。
  - `setup_kstack(struct proc_struct *proc)`：分配内核栈（通常用 `alloc_pages(n)`，然后 `KADDR()`）。
  - `copy_mm()` 或 `setup_pgdir()`：
    - 为子进程建立地址空间，可以是：
      - 直接共享父进程（内核线程方式），或者
      - 拷贝父进程页表（fork 语义），或者
      - 从 ELF 建立全新地址空间（exec）。
  - `copy_thread()` / `setup_tf()`：
    - 初始化 trapframe，使得该进程将来第一次被调度运行时会从某个入口执行（如 `user_entry`）。
  - `wakeup_proc(struct proc_struct *proc)`：
    - 修改其状态为 `RUNNABLE`
    - 插入就绪队列，等待调度器 schedule() 把它拉出来运行。
- **销毁 / 退出**：
  - `do_exit(int code)`：
    - 释放 mm、物理内存、内核栈
    - 修改状态为 `ZOMBIE` 或直接回收
    - 通知父进程（后续 Lab 用）

    - `do_wait()`：
        - 等待子进程退出，收尸（回收资源）。
- **查找**：
    - `find_proc(pid)`：在全局进程链表中寻找指定 pid 的进程。

> `kern_init()` → `proc_init()` → 创建 idleproc/initproc → 调用 `cpu_idle()` → 触发 schedule() → 第一次使用 `switch_to()` 切出 idleproc，切入第一个真正进程。

`switch.S`：**上下文切换实现**

汇编级别的进程切换代码。

```
.globl switch_to
switch_to:
    # a0 = &from->context, a1 = &to->context
    # 保存 from 的 callee-saved 寄存器 s0-s11, ra, sp ...
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    ...
    # 恢复 to 的上下文
    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ...
    ret
```

- `context` 结构只负责 **内核态调度下的寄存器**（PC = ret 到调用点）。
- 当发生陷入（trap）时，用户态的寄存器现场保存在 `trapframe` 中（在内核栈上）

1. `struct context` 的字段顺序与 `switch.S` 保存/恢复一致；
2. `proc.c` 初始化 context 时， `ra` 指向进程刚被调度时指向的地址（`kernel_thread_entry` 或 `forkret`）。

`entry.S`（**process 目录**）

不是内核 entry，而是**用户态程序的入口包装**

- 设置用户态的栈指针 `sp`
- 再跳到真正的 C 函数入口（如 `main`）
- 或者为 `exec` 后的程序提供一个统一起点。

`kern/schedule/` —— 调度器

- `sched.h`
    - 定义调度类接口 `struct sched_class`，包含：

```
struct sched_class {
    void (*enqueue)(struct proc_struct *proc);
    void (*dequeue)(struct proc_struct *proc);
    struct proc_struct* (*pick_next)(void);
    void (*proc_tick)(struct proc_struct *proc);
};
```

  - 以及 `extern struct sched_class* sched_class;` 全局变量。
  - 定义 `schedule()` 函数原型等。

- `sched.c`
  - 实现 `schedule()`：

```
void schedule(void) {
    struct proc_struct *next = sched_class->pick_next();
    if (next != current) {
        struct proc_struct *prev = current;
        current = next;
        switch_to(&(prev->context), &(next->context));
    }
}
```

  - 实现一个具体调度算法
    - run queue = 循环链表
    - `enqueue()` 把新就绪进程放到队尾
    - `pick_next()` 从队首挑 RUNNABLE 进程
    - `proc_tick()` 在每个时钟中断中被调用，减少时间片 count，到 0 时把进程重新排队
- 时钟中断（`clock.c`）→ `trap.c: interrupt_handler()` → `sched_class->proc_tick()` → (必要时) `schedule()`
- `cpu_idle()`：

```
while (1) {
    if (some_process_runnable)
        schedule();
    else
        halt();
}
```

`kern/sync/` —— 同步

- `sync.h`
  - 定义自旋锁等：

```
typedef struct {
    volatile int locked;
} spinlock_t;
```

  - 提供 `spin_lock` / `spin_unlock`：
    - 内部使用 `atomic.h`（根 libs）里的原子操作。

- 将来扩展 Lab（如文件系统）时，访问共享数据结构时会用锁保护。

`kern/trap/` —— 中断与异常（Trap）机制

- `trapentry.S`
  - 定义全局 trap 入口 `_trapentry`：
    1. 保存所有通用寄存器到当前内核栈（形成 `struct trapframe`）
    2. 切换到内核栈（如果当前在用户态）
    3. 调用 C 函数 `trap(struct trapframe *tf)`
- `trap.c`
  - `trap()` 根据 `tf->cause` 决定：
    - 是时钟中断→ `clock_interrupt()` + `schedule()`
    - 是外设中断→ 调对应 handler
    - 是系统调用→ `syscall()`，执行后返回
    - 是缺页异常→ `do_pgfault()`
  - `trap_init()`：
    - 配置 stvec（trap 向量基地址）
    - 使能相关中断。
- `trap.h`
  - 定义 `struct trapframe`：
    - 包含 **用户态所有寄存器值 + sepc + sstatus + stval 等。**
  - trapframe 用户上下文快照，一次系统调用/中断对应一个 trapframe。
- 用户态执行 → 产生 trap（系统调用/中断）→ trapentry.S 保存上下文到 `current->tf`
- 内核处理完 → 通过 `sret` 返回 → 恢复 tf 中的寄存器 → 回到用户态继续执行。

上下文切换两种方案：

1. **context**：进程在内核态被调度切走时保存的内核上下文（由 `switch.S` ）
2. **trapframe**：进程在用户态执行时，陷入内核保存的用户上下文（由 `trapentry.S` ）

`libs/`：通用基础库（内核+用户共享）

- `string.c / string.h`：`memcpy`, `memmove`, `strlen` 等。
- `stdio.h / printfmt.c`：格式化输出核心逻辑。
- `elf.h`：ELF 头与段表结构。**进程加载 ELF 可执行程序时用到**：

```
struct elfhdr { ... };
struct proghdr { ... };
```

- `riscv.h`：RISC-V 指令 & CSR 操作：
  - 宏和内联函数：
    - `read_csr(sstatus)`、`write_csr(stvec, addr)` 等。
- `sbi.h`：调用 RISC-V 的 SBI 接口（相当于固件服务）：
  - `sbi_console_putchar()`, `sbi_set_timer()` 等。

- `atomic.h`：原子操作（CAS、原子加减等）。
- `defs.h / error.h / list.h / hash.h`：
  - 通用宏定义、错误码、双向链表结构（`list_entry_t`）等。
  - 进程队列、就绪队列通常用 `list.h` 实现。

`tools/`：构建 & 链接 & 测试工具

- `kernel.ld` / `boot.ld`
  - 链接脚本，决定：
    - `.text` / `.data` / `.bss` 在物理/虚拟地址中的布局
    - 内核起始位置
  - 若布局不对，内核启动时可能找不到代码。
- `sign.c`
  - 对内核镜像进行签名或填充，方便 BootLoader 识别。
- `vector.c`
  - 若采用 C 写中断向量辅助逻辑，包含某些向量表相关内容。
- `gdbinit`
  - 预设 GDB 脚本：启动调试时自动加载符号、设置断点位置。
- `grade.sh`
  - 实验自动测试脚本，执行 QEMU + GDB 命令，观察输出

---

lab4 核心路径

1. **内核启动**：`entry.S → kern_init()`
2. **初始化内核机制**：内存管理、trap、时钟、调度器
3. **初始化进程系统**：`proc_init()`
   - 创建 `idleproc`（idle）
   - 创建 `initproc`（第一用户进程）
4. **进入调度循环**：`cpu_idle()` → 不断 `schedule()`
5. **时钟中断驱动调度**：
   - 时钟中断 → `trapentry.S` → `trap()`
     → `clock_interrupt()` → `sched_class->proc_tick()` →（必要时）`schedule()`
6. **schedule() 决定下一个进程**：
   - `pick_next()` 从就绪队列挑一个 RUNNABLE 的进程 `next`
   - 调用 `switch_to(&(current->context), &(next->context))`
7. **switch.S 完成"寄存器层面"的切换**，ret 到新进程的内核态 context。

练习1：分配并初始化一个进程控制块

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）

---

alloc_proc函数是一个实现进程控制块初始化的函数，他负责初始化一个进程控制块，首先我们需要找到代码的位置，然后根据这个结构的中的结构进行初始化。

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        // LAB4:EXERCISE1 2312282 CODE
        proc->state = PROC_UNINIT;//进程状态设置为未启动的状态
        proc->pid = -1;//进程ID设置为-1，表示未分配
        proc->runs = 0;//运行次数初始化为0
        proc->kstack = 0;//内核栈指针初始化为0
        proc->need_resched = 0;//不需要重新调度
        proc->parent = NULL;//父进程指针初始化为NULL
        proc->mm = NULL;//内存管理结构初始化为NULL
        memset(&(proc->context), 0, sizeof(struct context));//上下文结构体清零
        proc->tf = NULL;//陷阱帧指针初始化为NULL
        proc->pgdir = boot_pgdir_pa;//页目录基址设置为引导页目录物理地址
        proc->flags = 0;//进程标志初始化为0
        memset(proc->name, 0, sizeof(proc->name));//进程名称清零
        /*
         * below fields in proc_struct need to be initialized
         *       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
         */

    }
    return proc;
}
```

如代码所示，每一个结构体中各个成员变量的初始化设置如代码所示，其中注释中说明了每一个成员变量的初始化含义。

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

struct context context: 内核调度时保存的“内核态上下文”

含义：

- 进程的上下文切换信息，保存了进程被切换出去时的关键寄存器状态
- 包括：一些关键的寄存器，这些寄存器的值用于在进程切换中还原之前进程的运行状态

作用：

- **进程切换**：当发生进程切换时，保存当前进程的执行上下文，以便下次恢复执行时能继续从正确的位置运行

通过`memset(&(proc->context), 0, sizeof(struct context))`进行初始化。

struct trapframe *tf：用户态的寄存器上下文，保存在进程的内核栈顶部（trapentry.S）

含义：

- 保存了进程的中断帧。当进程从用户空间跳进内核空间的时候，进程的执行状态被保存在了中断帧中
- 包含所有通用寄存器、段寄存器、错误码、中断号等信息

作用：

- **中断处理**：在中断/异常发生时保存处理器的完整状态
- **系统调用**：用户态到内核态切换时保存用户态上下文
- **进程恢复**：从内核态返回用户态时恢复进程执行环境
- **特权级切换**：协助完成用户态和内核态之间的切换

在初始化代码中设置为`NULL`，因为在进程创建时还没有发生中断。

练习2：为新创建的内核线程分配资源

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。

接下来我们要进城创建内核进程，如何创建，在课堂上讲过，我们通过fork来将正在运行的进程信息fork到新的进程中来，以此实现了新进程的创建。

```
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;
    // LAB4:EXERCISE2  2312282 YOUR CODE
    // 1. 调用alloc_proc函数，获取一块用户信息块
    proc = alloc_proc();
    if (proc == NULL)
        goto fork_out;

    // 2.使用setup_kstack函数 直接为进程分配一个内核栈
    if (setup_kstack(proc) < 0)
        goto bad_fork_cleanup_proc;

    // 3.利用copy_mm（）函数 复制原来进程的内存管理信息到新进程中来
    if (copy_mm(clone_flags, proc) < 0)
        goto bad_fork_cleanup_kstack;

    // 4. 利用copy_thread（）函数复制原来进程上下文到新进程中来
    copy_thread(proc, stack, tf);

    // 5. 完善新进程的信息
    proc->parent = current;
    proc->pid = get_pid();//注意，该函数时设置进程编号的函数，这个函数规定了每个线程只能有一个唯一的id。
    proc->state = PROC_UNINIT; // 在wakeup_proc中设置为RUNNABLE
    proc->runs = 0;

    // 6. 添加到进程列表中（libs/list.h 提供 list_add_before/list_add_after）
    /* 使用 list_add_before 在队尾插入 */
    list_add_before(&proc_list, &proc->list_link);
    hash_proc(proc);

    nr_process++;//目前进程数加1

    // 7. 唤醒新进程
    wakeup_proc(proc);

    // 8. 返回新进程号
    ret = proc->pid;

    /*
     * Some Useful MACROs, Functions and DEFINEs, you can use them in below implementation.
```

```
     * MACROs or Functions:
     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack
     *   copy_mm:      process "proc" duplicate OR share process "current"'s mm
according clone_flags
     *                 if clone_flags & CLONE_VM, then "share" ; else
"duplicate"
     *   copy_thread:  setup the trapframe on the  process's kernel stack top
and
     *                 setup the kernel entry point and stack of process
     *   hash_proc:    add proc into proc hash_list
     *   get_pid:      alloc a unique pid for process
     *   wakeup_proc:  set proc->state = PROC_RUNNABLE
     * VARIABLES:
     *   proc_list:    the process set's list
     *   nr_process:   the number of process set
     */

    //    1. call alloc_proc to allocate a proc_struct
    //    2. call setup_kstack to allocate a kernel stack for child process
    //    3. call copy_mm to dup OR share mm according clone_flag
    //    4. call copy_thread to setup tf & context in proc_struct
    //    5. insert proc_struct into hash_list && proc_list
    //    6. call wakeup_proc to make the new child process RUNNABLE
    //    7. set ret vaule using child proc's pid

fork_out:
    return ret;

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

如代码所示，具体实现过程如注释所示。

请说明ucore是否做到给每个新fork的线程一个唯一的id?

可以实现，具体依据是`get_pid()`函数的内容

1. 基本分配策略

```
static int next_safe = MAX_PID, last_pid = MAX_PID;
if (++last_pid >= MAX_PID)
{
    last_pid = 1;  // 超过最大值时回绕到1
    goto inside;
}
```

- 使用递增分配策略，从1开始分配
- 当达到`MAX_PID`时回绕到1重新开始

2. 唯一性保证机制

```
while ((le = list_next(le)) != list)
{
    proc = le2proc(le, list_link);
    if (proc->pid == last_pid)  // 发现PID冲突
    {
        if (++last_pid >= next_safe)
        {
            // 重新扫描所有进程
            goto repeat;
        }
    }
    else if (proc->pid > last_pid && next_safe > proc->pid)
    {
        next_safe = proc->pid;  // 更新安全边界
    }
}
```

关键保证机制：

- **遍历检查**：每次分配前遍历整个进程链表检查PID是否已被使用
- **冲突处理**：如果发现冲突，递增PID并重新检查
- **安全边界**：通过`next_safe`记录下一个已被占用的PID，优化搜索范围

3. 完备性证明

```
static_assert(MAX_PID > MAX_PROCESS);
```

这是一个静态断言，这个静态断言确保了：

- **PID数量 > 最大进程数**：只要有空闲PID就一定能分配成功
- **不会死锁**：最多检查MAX_PID次就能找到可用PID

当 PID 增加到最大编号时，uCore 会回绕（wrap around），从 1 开始重新扫描整个 PID 空间，找到第一个未被占用的 PID。

具体逻辑如下：

1. `last_pid++`，若超出 `MAX_PID`，则重新设为 1。
2. 遍历所有进程（全局进程链表），判断当前 pid 是否被占用。
3. 若被占用，则继续 `last_pid++`，直到找到一个未使用的 pid。
4. 若扫描一圈仍未找到，则说明系统进程过多，fork 失败。

该机制保证 **PID 唯一性** 与 **PID 可重用性**，避免 PID 用尽。

练习3：编写proc_run 函数

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？

---

proc_run函数时将我们指定的进程换到CPU上面来执行，具体的切换主要注意的是，切换过程中我们需要禁用中断，并且将相关的页表和上下文进行切换，切换完成之后，再次允许中断。

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
        // LAB4:EXERCISE3  2312282 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
         */
        // LAB4:EXERCISE3
        // 1. 禁用中断,使用我们的local_intr_save（）函数，注意的是我们需要一个指针指向原来的进程，也即是现在的进程，用来保证页表和上下文的切换
        bool intr_flag;
        struct proc_struct *prev = current;
        local_intr_save(intr_flag);

        // 2. 将当前进程指针切换到新进程
        current = proc;

        // 3.切换页表 (satp),修改SATP寄存器
        lsatp((unsigned int)proc->pgdir);

        // 4. 实现上下文切换
        switch_to(&(prev->context), &(proc->context));

        // 5. 允许中断
        local_intr_restore(intr_flag);
    }
}
```

在本实验的执行过程中，创建且运行了几个内核线程?

共运行了 2 个内核线程：

1. **idleproc**
    - PID=0
    - 在系统没有别的 RUNNABLE 进程时运行
    - 一直 while(1) 中执行 schedule()
    - 表示空闲进程。在操作系统中，空闲进程是一个特殊的进程，它的主要目的是在系统没有其他任务需要执行时，占用 CPU 时间，同时便于进程调度的统一化。通过执行cpu_idle函数，也就是一直循环，找到需要执行的进程。
2. **initproc** **
    - 真正运行的“测试线程”
    - fork的idleproc进程，其内容是打印**Hello world!!**

proc_init() 的流程如下：

```
1. 创建 idleproc（第一个内核线程）
2. 创建 initproc（第二个内核线程）
```

在运行过程中：

- CPU 最先跑 idleproc
- 然后 schedule() 切换到 initproc
- initproc 打印信息后再主动让出 CPU
- idleproc 继续工作

所以本实验实际上验证了**两个内核线程之间的调度与切换**

```
zjs0914@ubuntu:~/opt/riscv/labcode/lab4$ make qemu
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc020004a (virtual)
  etext  0xc0203e9a (virtual)
  edata  0xc0209030 (virtual)
  end    0xc020d4ec (virtual)
Kernel executable memory footprint: 54KB
memory management: default_pmm_manager
physcial memory map:
  memory: 0x08000000, [0x80000000, 0x87ffffff].
vapaofset is 18446744070488326144
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_vmm() succeeded.
alloc_proc() correct!
++ setup timer interrupts
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
kernel panic at kern/process/proc.c:405:
    process exit!!.
```

challenge 1：local_intr_save 和 local_intr_restore 如何实现开关中断?

在 uCore 中，经常看到如下代码模式：

```
bool intr_flag;
local_intr_save(intr_flag);
{
    // 临界区代码
    // 执行需要原子性的操作
}
local_intr_restore(intr_flag);
```

这段代码的作用是在临界区**临时关闭中断**，执行完临界区代码后**恢复中断状态。**下面详细分析其实现原理。

宏定义位置

这两个宏定义在 `kern/sync/sync.h` 文件中：

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

核心函数实现

在同一个文件中，定义了核心函数：

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

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

底层中断控制函数

实际的开关中断操作在 `kern/driver/intr.c` 中实现：

```
/* intr_enable - enable irq interrupt */
void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }

/* intr_disable - disable irq interrupt */
void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }
```

实现原理深入分析

RISC-V 的 sstatus 寄存器

`sstatus`（Supervisor Status Register）是 RISC-V 架构中的**监管者状态寄存器**，用于控制 CPU 在 Supervisor 模式下的行为。

关键位定义（来自 `libs/riscv.h`）：

```
#define SSTATUS_SIE  0x00000002  // Supervisor Interrupt Enable (第1位)
```

- **SSTATUS_SIE**：Supervisor 模式中断使能位
  - `1`：允许 Supervisor 模式下的中断
  - `0`：禁止 Supervisor 模式下的中断

CSR 操作指令

RISC-V 提供了专门的 CSR（Control and Status Register）操作指令：

```
// 读取 CSR 寄存器
#define read_csr(reg) ({ unsigned long __tmp; \
  asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
  __tmp; })

// 设置 CSR 寄存器中的位（csrrs 指令：CSR Read and Set bits）
#define set_csr(reg, bit) ({ unsigned long __tmp; \
  asm volatile ("csrrs %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \
  __tmp; })

// 清除 CSR 寄存器中的位（csrrc 指令：CSR Read and Clear bits）
#define clear_csr(reg, bit) ({ unsigned long __tmp; \
  asm volatile ("csrrc %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \
  __tmp; })
```

指令说明：

- `csrr rd, csr`：读取 CSR 到通用寄存器
- `csrrs rd, csr, rs1`：读取 CSR 到 rd，同时将 CSR 中对应 rs1 中为 1 的位设置为 1
- `csrrc rd, csr, rs1`：读取 CSR 到 rd，同时将 CSR 中对应 rs1 中为 1 的位清除为 0

完整执行流程

示例代码：

```
void schedule(void) {
    bool intr_flag;
    local_intr_save(intr_flag);  // 第1步：保存并关闭中断
    {
        // 临界区代码
        current->need_resched = 0;
        // ... 其他操作
    }
    local_intr_restore(intr_flag);  // 第2步：恢复中断状态
}
```

第1步：`local_intr_save(intr_flag)` 的执行

1. **宏展开**：

```
do {
    intr_flag = __intr_save();
} while (0);
```

2. `__intr_save()` **执行**：

```
static inline bool __intr_save(void) {
    // 读取 sstatus 寄存器，检查 SIE 位
    if (read_csr(sstatus) & SSTATUS_SIE) {
        // 如果中断是开启的（SIE=1）
        intr_disable();  // 关闭中断
        return 1;        // 返回 1，表示之前中断是开启的
    }
    return 0;  // 返回 0，表示之前中断已经是关闭的
}
```

3. `intr_disable()` **执行**：

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

展开为汇编：

```
csrrc t0, sstatus, 0x00000002  ; 读取 sstatus 到 t0，同时清除 SIE 位
```

4. **结果**：
   - `intr_flag` 被设置为 `1`（如果之前中断是开启的）或 `0`（如果之前中断是关闭的）
   - CPU 的中断被关闭（`sstatus.SIE = 0`）

第2步：`local_intr_restore(intr_flag)` 的执行

1. **宏展开**：

```
__intr_restore(intr_flag);
```

2. `__intr_restore()` **执行**：

```
static inline void __intr_restore(bool flag) {
    if (flag) {  // 如果之前中断是开启的
        intr_enable();  // 重新开启中断
    }
    // 如果 flag=0，说明之前中断就是关闭的，不需要恢复
}
```

3. `intr_enable()` **执行**：

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

展开为汇编：

```
csrrs t0, sstatus, 0x00000002  ; 读取 sstatus 到 t0，同时设置 SIE 位
```

4. **结果**：
   - 如果 `intr_flag == 1`，中断被重新开启（`sstatus.SIE = 1`）
   - 如果 `intr_flag == 0`，中断保持关闭状态

设计优势分析

1. 状态保存与恢复：这种设计的核心优势是**保存并恢复中断状态**，而不是简单地关闭和开启：

- **如果进入临界区前中断是开启的**：关闭中断 → 执行临界区 → 重新开启中断
- **如果进入临界区前中断已经是关闭的**：保持关闭 → 执行临界区 → 保持关闭

这避免了**嵌套调用**时的问题：

```
void function_a() {
    bool flag1;
    local_intr_save(flag1);  // 关闭中断，flag1=1
    {
        function_b();  // 嵌套调用
    }
    local_intr_restore(flag1);  // 恢复中断（开启）
}

void function_b() {
    bool flag2;
    local_intr_save(flag2);  // 中断已关闭，flag2=0
    {
        // 临界区代码
    }
    local_intr_restore(flag2);  // 恢复中断（保持关闭，因为之前就是关闭的）
}
```

2. 原子性保证：使用 `csrrs` 和 `csrrc` 指令可以**原子性地**修改 CSR 寄存器，避免了竞态条件。

3. 最小化中断关闭时间：只在必要的临界区关闭中断，减少对系统响应性的影响。

使用场景

在 uCore 中，`local_intr_save/restore` 主要用于保护以下临界区：

1. **内存管理**（`kern/mm/pmm.c`）：

```
struct Page *alloc_pages(size_t n) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        page = pmm_manager->alloc_pages(n);
    }
    local_intr_restore(intr_flag);
    return page;
}
```

2. **进程调度**（`kern/schedule/sched.c`）：

```
void schedule(void) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        // 选择下一个运行的进程
        // 切换进程上下文
    }
    local_intr_restore(intr_flag);
}
```

3. **控制台输出**（`kern/driver/console.c`）：

```
void cons_putc(int c) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        sbi_console_putchar((unsigned char)c);
    }
    local_intr_restore(intr_flag);
}
```

总结

`local_intr_save` 和 `local_intr_restore` 的实现机制：

1. **保存状态**：通过读取 `sstatus.SIE` 位，保存中断的原始状态
2. **关闭中断**：使用 `csrrc` 指令原子性地清除 `SIE` 位
3. **执行临界区**：在中断关闭的情况下执行需要原子性的代码
4. **恢复状态**：根据保存的状态，使用 `csrrs` 指令恢复中断状态

这种设计保证了：

- **嵌套安全性**：支持嵌套调用而不破坏中断状态
- **原子性**：使用硬件指令保证操作的原子性
- **最小化影响**：只在必要时关闭中断，减少对系统响应性的影响
- **代码简洁**：通过宏定义提供简洁的接口

challenge 2：深入理解不同分页模式的工作原理

本思考题要求分析 `get_pte()` 函数（位于 `kern/mm/pmm.c`）中两段相似代码的设计原理，并结合 sv32、sv39、sv48 分页模式的异同进行解释。

get_pte() 函数分析

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
{
    // 第一段代码：处理第一级页目录项 (pdep1)
    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：处理第二级页目录项 (pdep0)
    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);
    }

    // 返回最终的页表项地址
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

两段代码的相似性

两段代码的结构完全一致，都执行以下步骤：

1. **获取页表项指针**：通过地址索引获取对应的页目录项
2. **检查有效性**：判断页表项的 `PTE_V` 位是否为 0（无效）
3. **条件分配**：如果无效且 `create` 为真，则分配新页
4. **初始化页表**：
    - 设置页引用计数为 1

- 将页内容清零
- 创建页表项并设置权限位（`PTE_U | PTE_V`）

RISC-V 分页模式：sv32、sv39、sv48 的异同

RISC-V 定义了多种分页模式，用于不同位宽的地址空间：

模式	虚拟地址位数	物理地址位数	页表级数	地址空间大小
sv32	32位	34位	2级	4GB
sv39	39位	56位	3级	512GB
sv48	48位	56位	4级	256TB

sv32（2级页表）

```
虚拟地址：31-22位 | 21-12位 | 11-0位
          PDX     |  PTX    | PGOFF
```

sv39（3级页表）

```
虚拟地址：38-30位 | 29-21位 | 20-12位 | 11-0位
          PDX2    |  PDX1   |  PTX    | PGOFF
```

sv48（4级页表）

```
虚拟地址：47-39位 | 38-30位 | 29-21位 | 20-12位 | 11-0位
          PDX3    |  PDX2   |  PDX1   |  PTX    | PGOFF
```

共同特点

1. **统一的页表项格式**：所有模式都使用相同的 PTE 结构
   - 物理页号（PPN）
   - 标志位（V, R, W, X, U, G, A, D 等）
2. **相同的页大小**：所有模式都使用 4KB 页大小
3. **相同的索引宽度**：每级页表使用 9 位索引（512 个条目）
4. **递归的页表结构**：每一级页表本身也是一个 4KB 页

从 `kern/mm/mmu.h` 中的定义可以看出，本 uCore 实现使用的是**三级页表结构**（类似 sv39）：

```
#define PDX1SHIFT  30  // 第一级页目录索引偏移
#define PDX0SHIFT  21  // 第二级页目录索引偏移
#define PTXSHIFT   12  // 页表索引偏移
```

地址划分：

- **PDX1**：30-21 位（9位，512个条目）
- **PDX0**：21-12 位（9位，512个条目）
- **PTX**：12-0 位中的高9位（页表索引）
- **PGOFF**：12-0 位（页内偏移）

多级页表的核心思想是**递归和自相似**：

- **第一级页目录**（pdep1）：指向第二级页目录的物理页
- **第二级页目录**（pdep0）：指向页表的物理页
- **页表**：指向实际物理页

每一级在逻辑上都是相同的结构：

- 都是一个 4KB 的页
- 都包含 512 个 8 字节的条目
- 每个条目要么指向下一级页表，要么指向物理页

在遍历多级页表时，每一级都需要执行相同的操作：

1. **检查存在性**：当前级页表项是否存在（`PTE_V` 位）
2. **按需分配**：如果不存在且需要创建，分配新页
3. **初始化**：清零新页并设置基本权限
4. **继续遍历**：使用当前页表项指向的地址访问下一级

这正是 `get_pte()` 函数中两段代码相似的根本原因：**它们都在执行多级页表遍历中的同一级操作，只是处理的层级不同。**

代码相似性的优势

1. **代码复用**：相同的逻辑可以处理不同层级
2. **易于扩展**：如果需要支持 sv48（4级），只需再添加一段相同结构的代码
3. **易于理解**：统一的模式使代码更清晰

函数设计评价：查找与分配是否应该分离？

当前 `get_pte()` 函数将两个功能合并：

- **查找功能**：遍历页表，找到对应的 PTE
- **分配功能**：如果页表不存在，按需分配新页

合并设计的优点

1. 简化调用接口

```
// 当前设计：一个函数完成所有操作
pte_t *ptep = get_pte(pgdir, la, 1);  // create=1 表示需要时分配
```

如果分离，调用者需要：

```
// 分离设计：需要多次调用
pte_t *ptep = find_pte(pgdir, la);
if (ptep == NULL) {
    alloc_page_table(pgdir, la);
    ptep = find_pte(pgdir, la);
}
```

2. 原子性保证：合并设计可以保证查找和分配操作的原子性，避免并发问题。
3. 减少代码重复：许多调用场景都需要"查找，不存在则分配"的逻辑，合并避免了重复代码。
4. 性能优化：可以在一次遍历中完成查找和分配，减少页表遍历次数。

合并设计的缺点

1. 职责不清晰：违反了单一职责原则（SRP），一个函数承担了多个职责。
2. 灵活性不足：某些场景可能只需要查找而不需要分配，但当前设计通过 `create` 参数控制，不够直观。
3. 错误处理复杂：函数可能因为分配失败返回 NULL，调用者难以区分是"不存在"还是"分配失败"。

分离设计的优点

如果拆分为两个函数：

```
// 纯查找函数
pte_t *find_pte(pde_t *pgdir, uintptr_t la);

// 分配函数
int alloc_page_table(pde_t *pgdir, uintptr_t la);
```

1. 职责清晰：每个函数只做一件事，符合单一职责原则。
2. 更好的错误处理：可以区分"页表不存在"和"分配失败"两种错误。
3. 更灵活的组合：调用者可以根据需要组合使用：

```
// 只查找
pte_t *ptep = find_pte(pgdir, la);

// 查找并分配
if (find_pte(pgdir, la) == NULL) {
    if (alloc_page_table(pgdir, la) != 0) {
        // 处理分配失败
    }
}
```

分离设计的缺点

1. 接口复杂化：调用者需要处理更多细节，代码更冗长。
2. 性能开销：可能需要多次遍历页表。
3. 并发问题：查找和分配之间的时间窗口可能导致竞态条件。

结论

对于 uCore 这样的教学操作系统，当前的设计是合理的，原因如下：

1. **教学目的**：合并设计更直观，学生更容易理解页表查找和分配的完整流程
2. **代码简洁**：减少了函数数量和调用复杂度
3. **实用性强**：大多数场景都需要"查找或创建"的操作

但在生产环境中，建议采用分离设计，因为：

1. **可维护性**：职责清晰，便于维护和测试
2. **可扩展性**：更容易添加新功能（如页表预分配、延迟分配等）
3. **错误处理**：更精确的错误信息有助于调试

知识点整理

实验中的重要知识点与OS原理对应

1. **进程与线程模型**
   - PCB 结构（proc_struct）
   - 用户态/内核态上下文（trapframe/context）
2. **进程状态与管理**
   - 新建、就绪、运行、退出
   - 进程链表与就绪队列
3. **处理机调度机制**
   - 时钟中断驱动调度
   - 时间片轮转（RR）算法
   - schedule() 与 switch_to()
4. **上下文切换原理**
   - 保存内核态 context
   - 保存用户态 trapframe
   - sret 中断返回
5. **虚拟内存与地址空间**
   - 每个进程独立页表
   - get_pte() 查找多级页表
   - SATP 切换地址空间
6. **中断与同步机制**
   - local_intr_save/restore
   - 临界区保护
   - 禁止中断与保证调度原子性
7. **fork 创建子进程机制**
   - PCB 初始化
   - 复制地址空间
   - 分配内核栈
   - 唤醒新进程

OS原理中重要但实验未涉及的知识点

1. **系统调用机制及用户态程序执行模型**（系统调用表、参数传递、内核返回路径）
2. **进程生命周期的完整实现**（exec、wait、僵尸进程、进程树管理）
3. **内核态/用户态地址空间的动态扩展与页面置换算法**（LRU、OPT、工作集模型）
4. **高级内存管理技术**（Copy-On-Write、Shared Memory）
5. **进程同步与互斥**（信号量、管程、条件变量、自旋锁、死锁处理）
6. **文件系统与外存管理**（inode、目录结构、文件描述符、磁盘调度）
7. **设备驱动与 I/O 系统**（中断处理层次、DMA、缓冲区缓存）
8. **系统启动与引导**（Bootloader、内核加载、硬件初始化）
9. **操作系统保护机制与虚拟化扩展**（特权级隔离、虚拟机支持）