OS lab4

练习1：分配并初始化一个进程控制块（需要编码）

> 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）

进程相关数据结构

从**微观**上看，每个进程都对应一个 `proc_struct` 结构（进程控制块）用来保存各种管理信息，主要包括：

- **state**：进程当前状态（ `PROC_UNINIT` 、 `PROC_SLEEPING` 、 `PROC_RUNNABLE` 、 `PROC_ZOMBIE` ）
- **pid / name[]**：进程号 / 进程名
- **kstack**：进程内核栈地址
- **parent**：指向父进程块的指针，进程的父子关系组成了一棵进程树
- **mm**：内存管理结构，包括内存映射，虚存管理等内容
- **context**：进程上下文，保存关键寄存器的值，用于在进程切换中还原之前进程的运行状态
- **trapframe**：进程中断帧，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
- **pgdir**：页表基地址，保存进程页表根节点的物理地址
- **flags**：进程标志位
- **list_link / hash_link**：链表节点，被挂到进程链表和进程哈希表中

从**宏观**上看，为了管理系统中所有的 `proc_struct` ，维护如下全局变量：

- **current**：当前占用CPU且处于“运行”状态进程控制块指针
- **initproc**：根据指导书说明，本实验中指向一个内核线程，本实验以后指向第一个用户态进程
- **proc_list**：双向链表结构体 `list_entry_t` 的实例化对象，所有进程控制块的双向链表

- **hash_list**：双向链表结构体 `list_entry_t` 的实例化对象，所有进程控制块的哈希表

alloc_proc()函数设计实现

`alloc_proc()` 函数主要负责分配并返回一个新的struct proc_struct 进程控制块结构。它首先会通过 `kmalloc()` 函数创建一个 `struct proc_struct *proc` 变量，我们需要对该 proc 的 enum proc_state state、int pid、int runs等 12 种成员参数进行初始化。

由于在 `proc_init()` 里面存在对于 `alloc_proc()` 是否初始化正确的检测，相关代码如下：

代码块

```
void proc_init(void){
    ......
    if ((idleproc = alloc_proc()) == NULL)
    {
        panic("cannot alloc idleproc.\n");
    }
    ......
    if (idleproc->pgdir == boot_pgdir_pa && idleproc->tf == NULL &&
!context_init_flag && idleproc->state == PROC_UNINIT && idleproc->pid == -1 &&
idleproc->runs == 0 && idleproc->kstack == 0 && idleproc->need_resched == 0 &&
idleproc->parent == NULL && idleproc->mm == NULL && idleproc->flags == 0 &&
!proc_name_flag)
    {
        cprintf("alloc_proc() correct!\n");
    }
    ......
}
```

因此我们可以直接参考上述逻辑，反推出 `alloc_proc()` 函数的实现：

代码块

```
static struct proc_struct * alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
```

```
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->pgdir = boot_pgdir_pa;
        proc->flags = 0;
        memset(proc->name, 0, sizeof(proc->name));
    }
    return proc;
}
```

context 和 trapframe 成员变量的含义及作用

1. **`struct context context`**：

`context` 结构体保存了 `ra`，`sp` 寄存器，是进程在内核态切换时需要恢复的最少寄存器集合，对应**该进程执行的上下文**；在新建线程时 `context` 被初始化为能返回到内核入口并以陷入帧为栈顶继续执行；在进程调度切换时，`context` 作为 switch_to 的目标状态；

代码块

```
// pro.c
void switch_to(struct context *from, struct context *to); // ->switch.S
```

2. **`struct trapframe *tf`**：

`trapframe *tf` 为中断帧，当进程因为系统调用、中断、异常而被打断时，内核必须保存它原先的寄存器内容，trapframe即为**被打断时陷入内核前的寄存器快照**，具体内容包括通用寄存器、特权状态和下一条将执行的指令地址。

这里我们需要重点分析一下 **`copy_thread()` 函数**：copy_thread 会同时设置 trapframe 和 context。对于trapframe，子进程恢复到用户态时，从父进程原来的寄存器开始执行；而对于 context，调度器切换到子进程时，会从内核态执行 `forkret()`，最后返回到用户态运行。

代码块

```
static void  //子进程要从 父进程被中断的位置继续执行，fork 的返回值让父子不同：父返回子 pid，子返回 0，所以子进程需要把 a0 = 0
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf)
{
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE - sizeof(struct trapframe)); //栈顶减去 trapframe 大小就是放 trapframe 的位置
    *(proc->tf) = *tf; //复制父进程的tf寄存器快照给子进程

    // Set a0 to 0 so a child process knows it's just forked
    proc->tf->gpr.a0 = 0;
```

```
9       proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp; //设置用户态的 sp
10
11      proc->context.ra = (uintptr_t)forkret; //子进程在第一次被调度时，会跳转到
    forkret()
12      proc->context.sp = (uintptr_t)(proc->tf); //切换到子进程时，它的内核栈起点是
    trapframe 的位置
13  }
```

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。

do_fork 设计实现过程

我们先调用 `alloc_proc` 获得全新 PCB，如若失败直接沿 `goto fork_out` 返回，对应代码注释的第 1 步：

代码块

```
1   // 1. call alloc_proc to allocate a proc_struct
2   if ((proc = alloc_proc()) == NULL)
3   {
4       // 分配失败则直接返回
```

```
5       goto fork_out;
6   }
7   proc->parent = current;  // 记录父进程指针
```

随后写入 `proc->parent = current`，继承父进程指针，确保等待/回收链正确。接着调用 `setup_kstack` 为子进程分配 KSTACKPAGE 大小的内核栈；一旦失败跳转到 `bad_fork_cleanup_proc` 释放 PCB，对应代码注释的第 2 步：

代码块

```
1   // 2. call setup_kstack to allocate a kernel stack for child process
2   if ((ret = setup_kstack(proc)) != 0)
3   {
4       // 内核栈分配失败，释放proc
5       goto bad_fork_cleanup_proc;
6   }
```

`copy_mm(clone_flags, proc)` 负责根据 `CLONE_VM` 决定复制还是共享地址空间（实际上本次实验中，`copy_mm` 函数里啥也没干）；失败时跳向 `bad_fork_cleanup_kstack` 释放栈资源，对应代码注释的第 3 步：

代码块

```
1   // 3. call copy_mm to dup OR share mm according clone_flag
2   if ((ret = copy_mm(clone_flags, proc)) != 0)
3   {
4       // 复制内存空间失败，释放内核栈
5       goto bad_fork_cleanup_kstack;
6   }
```

`copy_thread(proc, stack, tf)` 将 trapframe 与 context 拷贝到子进程栈顶，顺便把 `a0` 置零、栈顶指针指向新 tf，对应代码注释的第 4 步：

代码块

```
1   // 4. call copy_thread to setup tf & context in proc_struct
2   copy_thread(proc, stack, tf);  // 复制上下文到子进程
```

接着调用 `get_pid` 获得唯一 pid，`hash_proc` 把 PCB 插入 pid 哈希，`list_add(&proc_list, ...)` 把它链接进全局进程链表，对应代码注释的第 5 步：

代码块

```
// 5. insert proc_struct into hash_list && proc_list
proc->pid = get_pid();  // 获取唯一pid
hash_proc(proc);        // 加入哈希表方便查找
list_add(&proc_list, &(proc->list_link));  // 挂入进程链表
```

最后 `nr_process++` 与 `wakeup_proc(proc)` 把子进程变为 RUNNABLE，并用 `ret = proc->pid` 把 pid 返还父进程，对应代码注释的第 6,7 步：

代码块

```
// 6. call wakeup_proc to make the new child process RUNNABLE
// 7. set ret vaule using child proc's pid
nr_process++;  // 全局进程计数加一
wakeup_proc(proc);  // 置为可运行
ret = proc->pid;    // 返回子进程pid
```

PID 唯一性分析

分步解析get_pid函数：

先在 `get_pid` 中维护两个静态变量：`last_pid` 记录上一次成功的 pid，`next_safe` 标记下一段“安全区间”上界，且通过 `static_assert(MAX_PID > MAX_PROCESS)` 保证空间充足：

代码块

```
static_assert(MAX_PID > MAX_PROCESS);  // 断言可用 PID 空间大于最大进程数
struct proc_struct *proc;
list_entry_t *list = &proc_list, *le;
static int next_safe = MAX_PID, last_pid = MAX_PID;  // last_pid 记录上次分配，
next_safe 标记安全上界
// 注意：静态变量赋初值仅执行一次，起到记录分配历史的作用（唤醒了我C++沉睡的记忆）
```

函数首先执行 `++last_pid`，若到达 `MAX_PID` 就回绕到 1，并跳入 `inside` 标签重新扫描：

代码块

```
if (++last_pid >= MAX_PID)
{
    last_pid = 1;  // 超界后回绕到 1
    goto inside;
}
```

这个自增回绕机制保证 pid 不会溢出。接着，当 `last_pid >= next_safe` 时（`last_pid == next_safe` 即该 pid 已经分配出去），代码进入 `inside` / `repeat`，遍历整个

`proc_list`：

代码块

```
if (last_pid >= next_safe)
{
inside:
    next_safe = MAX_PID;  // 重新扫描前先将安全上界设为最大值
repeat:
    le = list;
    while ((le = list_next(le)) != list)
    {
        // 循环体
    }
```

当然，如果 `last_pid < next_safe`，直接分配。

如果发现某个进程 pid 与当前候选号相同，就再次递增 `last_pid` 并视情况回绕直到找到空位：

代码块

```
if (proc->pid == last_pid)
{
    if (++last_pid >= next_safe)
    {
        if (last_pid >= MAX_PID)
        {
            last_pid = 1;  // 再次超界则回绕
        }
        next_safe = MAX_PID;  // 重置安全上界并重新扫描
        goto repeat;
    }
}
```

遍历过程中还会记录所有大于 `last_pid` 的最小已分配 pid 作为新的 `next_safe`，于是下次调用时无需扫描完整链表即可快速跳到安全区域：

代码块

```
else if (proc->pid > last_pid && next_safe > proc->pid)
{
    next_safe = proc->pid;  // 更新下一个安全上界
}
```

综上，`get_pid` 每次返回前都确认 `last_pid` 尚未被任何 `proc->pid` 占用，ucore 能确保每个新 fork 的线程都拥有唯一 pid；旧 pid 只有在进程退出并从 `proc_list` 移除后才可能被重新分配。

练习3：编写proc_run 函数（需要编码）

> 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 设计实现过程

proc_run 就是把 alloc_proc 和 do_fork 里面准备好的所有状态接到 CPU 上。

1. 先判断是不是同一个进程：如果传进来的 proc 跟 current 已经一样了，那说明上下文已经在了，直接 return：

代码块

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

2. 因为要改全局 current，又要改页表、寄存器，所以要关闭中断，保证切换的原子性，用 local_intr_save / restore 把切换的内容包住：

代码块

```
1   bool intr_flag;
2   local_intr_save(intr_flag);
3   //【中间的代码】
4   local_intr_restore(intr_flag);
```

3. 然后保存老进程，切到新进程，把 current 先保存成 prev，再把 current 指向目标 proc，这样后面 switch_to 才知道从谁切到谁：

代码块

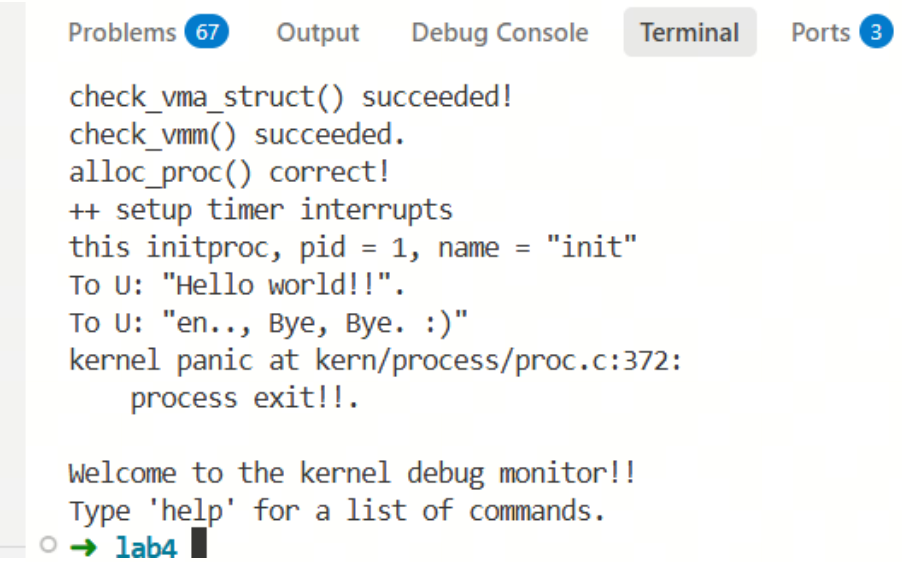
```
1   struct proc_struct *prev = current;
2   current = proc;
3   lsatp(proc->pgdir);
4   switch_to(&(prev->context), &(proc->context));
```

prev在 alloc_proc 里清零，context 是do_fork 里填好的

代码块

```
1    //【alloc_proc中】
2    memset(&(proc->context), 0, sizeof(struct context));
3    //【do_fork中】
4    copy_thread(proc, stack, tf);
```

4. 切页表：satp 寄存器决定地址空间，新进程的页表基址在 do_fork 结束前就放到 proc->pgdir 里了，proc_run 只要调用 lsatp(proc->pgdir) 就能切换地址空间
5. 切上下文：最后用 switch_to(&(prev->context), &(proc->context))。这两个 context 一个在 alloc_proc 里清零，另一个在 copy_thread 里设置好返回点、栈顶，所以下一次 CPU 真正执行的是新的 trapframe / 上下文
6. **验证：make qemu执行后终端输出如下**（do_exit还没实现所以会报panic）



可以看到alloc_proc函数correct，成功创建了进程 init，pid为1，具体的创建流程见后文。

创建且运行了几个内核线程

两个进程，分别如下，详细分析见后文：

1. idleproc（pid 0）：proc_init 先用 alloc_proc 创建 idleproc，自身设置为 `PROC_RUNNABLE` 并立刻作为 `current` 运行
2. initproc（pid 1）：随后 kernel_thread(init_main, ...) 通过 do_fork 再建一个线程，find_proc 把它取出来命名为 “init”，之后由调度器切换运行

为了验证这个结论，添加日志输出如下：

代码块

```
//在程序开头定义
static void
log_proc_created(const char *tag, struct proc_struct *proc){
    cprintf("[proc] %s pid=%d name=\"%s\" parent=%d\n", tag, proc->pid,
            proc->name, (proc->parent != NULL) ? proc->parent->pid : -1);
}
//在do_fork中添加
    log_proc_created("do_fork", proc);
//在proc_init函数中添加
    log_proc_created("proc_init", idleproc);
```

再次make，后端输出如下，可以看到通过alloc_proc和do_fork分别生成了两个process

```
check_vmm() succeeded.
alloc_proc() correct!
[proc] proc_init pid=0 name="idle" parent=-1
[proc] do_fork pid=1 name="" parent=0
++ setup timer interrupts
this initproc, pid = 1, name = "init"
```

创建init的整体流程

1. proc_init：proc_init 是进程子系统的入口，它先把全局的 `proc_list` 和哈希表清空，为之后登记进程做铺垫

代码块

```
void proc_init(void){
    int i;
    list_init(&proc_list);
    for (i = 0; i < HASH_LIST_SIZE; i++)
    {
        list_init(hash_list + i);
    }
```

2. **先造出 idleproc**，proc_init 调 alloc_proc() 分出一个崭新的 PCB，这一步把状态字段全清零，`pgdir` 指到 boot_pgdir，`context` 和名字也都清零

代码块

```
    if ((idleproc = alloc_proc()) == NULL)
      {
          panic("cannot alloc idleproc.\n");
      }
      // check the proc structure
      int *context_mem = (int *)kmalloc(sizeof(struct context));  //创建一个context结构体，进程上下文，定义见proc.h
      memset(context_mem, 0, sizeof(struct context));
      int context_init_flag = memcmp(&(idleproc->context), context_mem, sizeof(struct context));

      int *proc_name_mem = (int *)kmalloc(PROC_NAME_LEN);
      memset(proc_name_mem, 0, PROC_NAME_LEN);
      int proc_name_flag = memcmp(&(idleproc->name), proc_name_mem, PROC_NAME_LEN);

      if (idleproc->pgdir == boot_pgdir_pa && idleproc->tf == NULL && !context_init_flag && idleproc->state == PROC_UNINIT && idleproc->pid == -1 && idleproc->runs == 0 && idleproc->kstack == 0 && idleproc->need_resched == 0 && idleproc->parent == NULL && idleproc->mm == NULL && idleproc->flags == 0 && !proc_name_flag)
      {
          cprintf("alloc_proc() correct!\n");
      }
```

接着给它赋值 `pid=0`、`state=PROC_RUNNABLE`、`kstack=bootstack`，并命名为 `"idle"`，还把 `current` 指过去，所以现在 CPU 实际上在跑 idleproc

代码块

```
      idleproc->pid = 0;
      idleproc->state = PROC_RUNNABLE;
      idleproc->kstack = (uintptr_t)bootstack;
      idleproc->need_resched = 1;
      set_proc_name(idleproc, "idle");
      nr_process++;
      current = idleproc;
```

3. 为了得到 init 进程，idleproc 通过 kernel_thread 触发一次 do_fork。kernel_thread 会准备一个内核态 trapframe，然后直接走到 do_fork 去复制出“孩子”

代码块

```
    int pid = kernel_thread(init_main, "Hello world!!", 0);  // 以context为参数，创建init_main线程（或者说进程，似乎这概念没分的那么清晰）
    if (pid <= 0){
        panic("create init_main failed.\n");
    }
```

4. **do_fork 里完成 initproc 的 PCB 构建：**

- 先再调一次 alloc_proc 拿到子 PCB，并设置 `proc->parent = current`（也就是 idleproc）

代码块

```
if ((proc = alloc_proc()) == NULL)
    {
        // 分配失败则直接返回
        goto fork_out;
    }
    proc->parent = current;  // 记录父进程指针
```

- setup_kstack 给它分配新的内核栈；copy_mm 这里是空操作（内核线程不需要用户态地址空间）

代码块

```
  if ((ret = setup_kstack(proc)) != 0)
    {
        // 内核栈分配失败，释放proc
        goto bad_fork_cleanup_proc;
    }
```

- copy_thread 把父进程传进来的 trapframe 复制一份到子进程的栈顶，同时设好 `context.ra=forkret`、`context.sp=trapframe`，确保之后能切进去执行 init_main

代码块

```
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf)
{
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE - sizeof(struct trapframe));
    *(proc->tf) = *tf;

    // Set a0 to 0 so a child process knows it's just forked
    proc->tf->gpr.a0 = 0;
    proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;
```

```
9
10      proc->context.ra = (uintptr_t)forkret;
11      proc->context.sp = (uintptr_t)(proc->tf);
12  }
```

- 分配唯一 pid (get_pid())，把它挂到进程哈希表和 `proc_list` 里，更新 `nr_process`，最后 `wakeup_proc` 让它处于 RUNNABLE 状态

代码块

```
1       proc->pid = get_pid();  // 获取唯一pid
2       hash_proc(proc);        // 加入哈希表方便查找
3       list_add(&proc_list, &(proc->list_link));  // 挂入进程链表
4       nr_process++;  // 全局进程计数加一
5       wakeup_proc(proc);  // 置为可运行
6       log_proc_created("do_fork", proc);  //日志日志
7       ret = proc->pid;    // 返回子进程pid
```

5. 调度器之后会跑到 initproc：一旦 `schedule()` 发现有这个 RUNNABLE 的子进程，就会通过 proc_run 切换到它；因为 copy_thread 已把 context/stack 指到 `forkret -> init_main`，所以子进程第一次运行就跳进 init_main，打印“this initproc…”那几行，说明 init 线程成功启动

代码块

```
1   void cpu_idle(void) {
2       while (1) {
3           if (current->need_resched) {
4               schedule();
5           }
6       }
7   }
8
9   init_main(void *arg)
10  {
11      cprintf("this initproc, pid = %d, name = \"%s\"\n", current->pid,
    get_proc_name(current));
12      cprintf("To U: \"%s\".\n", (const char *)arg);
13      cprintf("To U: \"en.., Bye, Bye. :)\"\n");
14      return 0;
15  }
```

综上，idleproc 创建成功后，靠 kernel_thread/do_fork 这一套流程把 initproc（pid=1）创建出来；拷贝 trapframe + 设置 context 的那几步确保它一上线就能执行 init_main，所以 qemu 日志里能看到

initproc 的输出，证明整个创建链路跑通了。

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？
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，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。
   - 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？

扩展一：中断开关实现

- `local_intr_save(x)`：通过读取 sstatus 检查 SSTATUS_SIE 位是否为 1，若当前中断已开启则先调用 `intr_disable()` 关闭中断并返回 1，否则返回 0；该返回值赋给 x 用于后续恢复判断；该函数**用于确保当前段内无中断打断，同时记录是否需要在退出时恢复中断**，体现中断的**原子性**。
- `local_intr_restore(x)`：若原先开启过中断 x 为 1，则调用 `intr_enable()` 重新开启；若 x 为 0，则保持关闭状态不变。该函数主要**用于按原始状态恢复中断**，以维护系统**中断全局状态**。
- 引入 `do { ... } while (0)`：强制调用方在末尾写分号，主要用于 if (...) ... else ... 场景，避免悬挂 else问题；同时形成局部作用域，便于在宏体内多条语句的安全调用。
- 具体关闭中断时 `intr_disable()` 会清除 sstatus 的 SSTATUS_SIE 位，禁用内核态外部中断；开启中断时 `intr_enable()` 置位 sstatus 的 SSTATUS_SIE 位，允许内核态外部中断。

代码块

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

```
9
10  static inline void __intr_restore(bool flag) {
11      if (flag) {
12          intr_enable();
13      }
14  }
15
16  #define local_intr_save(x) \
17      do {                   \
18          x = __intr_save(); \
19      } while (0)
20  #define local_intr_restore(x) __intr_restore(x);
21
22  // driver/intr.c
23  void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }
24  void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }
```

扩展二：深入理解不同分页模式的工作原理

问题1：get_pte()函数相似代码分析：

这两段代码复用的同一模式：检测有效位、必要时分配一页作为下一级页表、清零该页、设置页目录项指向新页并置位标志。这是确保页表层级存在的通用流程，第一段针对高一级目录项 PDX1(la)（L2），第二段针对次一级目录项 PDX0(la) （L1）。它们相似是由于层级递归的必然性导致：SV39 是多级页表结构，非叶子层级的“缺页则分配页表页并初始化”步骤在每一层都相同，只是索引与被写入的目录项不同；最终返回叶子层的 PTE 指针。

问题2：get_pte()函数功能合并分析：

`get_pte()` 函数查找虚拟地址 la 对应的 PTE，但如果没有就返回 NULL， 如果页目录/页表不存在且create=true，则自动分配新页表，并返回对应 PTE 地址；这是按需分配页表的典型写法。

- 优点：简单，使用方便。我们无需关心页表是否存在，也无需手动创建页表。
- 缺点：功能耦合在一起，比较复杂。

针对本实验来说，我们认为该设计是可以保留的，通过 create 标志就已经实现了简单清晰的语义分离；但如果后续实验框架变得复杂，则可以把查找与分配拆为两个函数，一个只查找不创建，一个专门用于创建。

代码块

```
1  pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) //在页目录 pgdir 中查找虚
   拟地址 la 对应的 PTE 最终页表项，如果 create=true 并且缺页，则自动分配表页
2  {
3      pde_t *pdep1 = &pgdir[PDX1(la)]; //从虚拟地址 la 中取出一级页目录索引
4      if (!(*pdep1 & PTE_V)) //PTE_V有效标志位
```

```
    {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL)
        {
            return NULL;
        }
        set_page_ref(page, 1); //页表页属于内核管理，引用计数设为 1。
        uintptr_t pa = page2pa(page); //把 page 转换成物理地址
        memset(KADDR(pa), 0, PGSIZE); //把物理地址 pa 转为“内核虚拟地址”，并把新页表清空
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V); //把新页表写入一级目录项
    }
    pde_t *pdep0 = &((pte_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)]; //从一级目录项里取出“二级页表的物理地址”，将物理地址 pa 转换成内核虚拟地址，将其转换为 pte_t 数组后再取出二级页目录项索引
    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)]; //根据虚拟地址 la 的页内偏移（PTX(la)）找到该地址的最终 PTE
}
```

重要知识点总结

一、实验中的核心知识点及其与OS原理的对应关系

> - 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

本实验中重要的知识点主要在于**进程控制块、进程状态、进程创建、进程上下文切换与进程调度**等方面。**进程控制块**是操作系统管理进程的核心数据结构，包含进程的所有状态信息，实验对应`struct proc_struct`结构体，包含state、pid、kstack、parent、context、tf、pgdir等成员，还增加了context和trapframe结构；**进程状态模型**包括`enum proc_state`定义的PROC_UNINIT、

PROC_SLEEPING、PROC_RUNNABLE、PROC_ZOMBIE四种状态；**进程创建机制**通过复制父进程创建子进程，实现进程的繁衍，对应 `do_fork()` 函数完成PCB分配、内核栈设置、上下文复制等完整创建流程；而**进程上下文切换**可以保存当前进程状态，恢复目标进程状态，实现CPU时间的分时复用，实验中的context只保存了关键寄存器的值（ra、sp），体现了最小化保存原则；实验还通过 `need_resched` 标志触发 `schedule()` **调度机制**，该函数选择RUNNABLE进程作为下一个运行的进程，实现了简单的协作式调度，但没有涉及时间片轮转等复杂调度算法；除上述内容之外，实验还涉及到重要的**虚拟内存管理和页表**机制，重点在于如何进行内核线程的管理。

二、操作系统原理中重要但实验未涉及的知识点

> - 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

首先，实验中所有进程都是内核线程，没有实现真正的**用户态进程**，也即是说缺少用户地址空间管理、系统调用接口等用户进程相关机制；其次，实验没有实现信号、管道、消息队列、共享内存等**进程间通信方式**，缺少进程同步信号量、互斥锁的具体实现；此外，实验未涉及完整的**进程生命周期管理，**缺少进程终止和资源回收，没有实现僵尸进程的处理和父进程等待机制；最后，在**进程调度方面**，实验采用的是简单的FIFO调度，没有实现优先级调度、时间片轮转、多级队列等复杂调度策略，而这些都是老师在课上提到的重点，需要我们更加注意。