操作系统lab4实验报告

小组成员：

陈秋彤（2311815）徐盈蕊（2311344）杨欣瑞（2312246）

练习1：分配并初始化一个进程控制块（需要编码）

alloc_proc函数负责分配并返回一个新的`proc_struct`，我们先来看看`proc_struct`结构体包含哪些内容：

```
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
};
```

可以看到，`proc_sruct`结构体中包含了进程的状态、进程ID、进程的内核栈、进程的上下文、进程的页表等信息。根据这个结构体，我们可以回答指导书的问题：**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?**

`struct context context`是进程的执行上下文，主要是被调用者保存的寄存器，`context`结构体包含以下寄存器，`ra`表示返回地址，`sp`表示栈指针，`s0`-`s11`是保存寄存器。`context`是在当进程被切换出去时，`context`会保存当前CPU状态；当进程被重新调度时，可以用来恢复之前保存的执行状态。

```
struct context
{
    uintptr_t ra;
    uintptr_t sp;
    uintptr_t s0;
    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 trapframe *tf`是当前进程的中断帧，保存中断/异常/系统调用时的完整CPU状态，包含所有通用寄存器、程序计数器、状态寄存器等完整现场信息。有关`trapframe`结构体的内容，我们在`lab3`中已经做过分析了，`gpr`是32个通用寄存器，`status`记录了核心状态，`epc`是程序计数器，`badvaddr`是异常地址，`cause`是异常原因。

```
struct trapframe {
    struct pushregs gpr;
    uintptr_t status;
    uintptr_t epc;
    uintptr_t badvaddr;
    uintptr_t cause;
};
```

`context`用于进程之间主动协作切换的上下文保存。这种切换发生在内核态，是操作系统调度器主动发起的行为，其核心机制类似于一个高级的、跨进程的函数调用。因此，被调用者需要保存并恢复那些需要跨调用保持不变的寄存器，而临时寄存器的生命周期仅限于本次函数调用，由调用者负责管理，在切换时没有保存的价值。所以，`context`仅保存这部分关键寄存器就足够了，

`trapframe`是用于处理被动强制中断的上下文保存。这种切换是由外部事件或内部异常强行触发的，可能发生在指令执行流的任意一点，并且常常伴随着CPU特权级的变化。这种切换的核心要求是：被中断的进程必须完全感知不到自己被中断过，在恢复时必须能够分毫不差地继续执行。这就要求内核必须完整封存中断发生那一瞬间的整个CPU状态——包括所有通用寄存器、程序计数器、状态寄存器等。

现在我们来完善`alloc_proc`函数，首先分配一个`proc_struct`结构体，然后初始化进程的状态、进程ID、进程的内核栈、进程的上下文、进程的中断帧等信息。

实际编程时，我们在实现时同步参考了`proc_init`函数：

```
// proc_init - set up the first kernel thread idleproc "idle" by itself and
//           - create the second kernel thread init_main
void proc_init(void)
{
    int i;

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

    if ((idleproc = alloc_proc()) == NULL)
    {
        panic("cannot alloc idleproc.\n");
    }

    // check the proc structure
    int *context_mem = (int *)kmalloc(sizeof(struct context));
    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");
    }

    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;

    int pid = kernel_thread(init_main, "Hello world!!", 0);
    if (pid <= 0)
    {
        panic("create init_main failed.\n");
    }

    initproc = find_proc(pid);
    set_proc_name(initproc, "init");

    assert(idleproc != NULL && idleproc->pid == 0);
    assert(initproc != NULL && initproc->pid == 1);
}
```

可以看到，在`proc_init`中，对于`alloc_proc`函数的检查有以下几点：

1. `idleproc->pgdir == boot_pgdir_pa`：进程的页表应该指向内核页表。这不难理解，因为在本实验中，我们的虚拟内存仍采用预映射方式，即在建立地址空间时一次性完成所有需要的页表映射，不涉及按需分配或页面置换。因此，所有进程的页表都指向内核页表`boot_pgdir`。
2. `idleproc->tf == NULL`：进程的中断帧应该为空。
3. `memcmp(&(idleproc->context), context_mem, sizeof(struct context))`：进程的上下文应该为空。
4. `idleproc->state == PROC_UNINIT`：进程的状态应该为未初始化，这也不难理解，因为我们还没有为进程分配资源，进程的初始状态应该是未初始化。
5. `idleproc->pid == -1`：进程的ID应该为-1。
6. `idleproc->runs == 0`：进程的运行次数应该为0。
7. `idleproc->kstack == 0`：进程的内核栈应该为0。

8. `idleproc->need_resched == 0`：进程不需要重新调度。
9. `idleproc->parent == NULL`：进程的父进程应该为空。
10. `idleproc->mm == NULL`：进程的内存管理应该为空。
11. `idleproc->flags == 0`：进程的标志应该为0。
12. `memcmp(&(idleproc->name), proc_name_mem, PROC_NAME_LEN)`：进程的名字应该为空。

根据这些检查，我们可以完善`alloc_proc`函数：

```
static struct proc_struct *
alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        memset(proc, 0, sizeof(struct proc_struct));
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        proc->tf = NULL;
        proc->pgdir = boot_pgdir_pa;
        proc->flags = 0;
        list_init(&proc->list_link);
        list_init(&proc->hash_link);

    }
    return proc;
}
```

因为我们用`memset`设置了`proc`的所有字段为0，所以我们可以省略掉对`context`和`name`的显式赋值。

练习2：为新创建的内核线程分配资源（需要编码）

接下来我们完善`do_fork`函数，为子进程分配资源。在`do_fork`函数中，我们需要为子进程分配进程控制块、内核栈、进程页表、进程上下文等资源。具体来说，我们需要完成以下步骤：

1. 调用`alloc_proc`函数创建一个子进程控制块。
2. 调用`setup_kstack`函数为子进程分配内核栈。
3. 调用`copy_mm`函数复制父进程的内存管理结构，为子进程分配进程页表。
4. 调用`copy_thread`函数复制父进程的上下文，为子进程分配上下文。
5. 把子进程添加到就绪队列中。
6. 调用`wakeup_proc`函数将子进程状态设置为就绪态。
7. 返回子进程的PID。

第一步，我们需要调用`alloc_proc`函数创建一个子进程控制块，如果失败则会使返回的`proc`为`NULL`，此时我们还没有为进程真正分配资源，所以这种情况需要进入`fork_out`标签。

```
if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
```

如果分配成功，我们就可以把当前进程的父进程设置为当前进程`current`了，然后为子进程分配内核栈，我们调用已有的`setup_kstack`函数，这个函数的内容如下：

```
static int
setup_kstack(struct proc_struct *proc)
{
    struct Page *page = alloc_pages(KSTACKPAGE);
    if (page != NULL)
    {
        proc->kstack = (uintptr_t)page2kva(page);
        return 0;
    }
    return -E_NO_MEM;
}
```

可以看到，`setup_kstack`函数会调用`alloc_pages`函数分配一页内存，然后将其地址转换为内核虚拟地址，最后将这个地址赋值给`proc->kstack`。如果分配失败，则返回`-E_NO_MEM`，此时我们还没有为进程真正分配资源，所以这种情况需要进入`bad_fork_cleanup_proc`标签。

```
 proc->parent = current;

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
```

如果分配成功，我们就可以调用`copy_mm`函数复制父进程的内存管理结构，为子进程分配进程页表。

```
if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
```

接下来，我们需要调用`copy_thread`函数复制父进程的上下文，为子进程分配上下文。这个函数的内容如下：

```
static void
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;

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

可以看到，`copy_thread`函数需要传入`esp`和`tf`，然后将`proc->tf`放在了`proc->kstack`的顶部，并将内容设置为我们传入的`tf`的内容，然后将`proc->tf->gpr.a0`设置为0，表示子进程刚刚被创建。

我们在`do_fork`中，将对应的参数传入，即可调用`copy_thread`函数：

```
copy_thread(proc,stack,tf);
```

接下来我们就可以把子进程添加到就绪队列中了，我们首先需要为分配资源成功的子进程申请一个PID，然后将其添加到哈希表中，最后将其添加到就绪队列中：

```
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);
```

在申请PID时，我们需要调用`get_pid`函数，这个函数的内容如下：

```
static int
get_pid(void)
{
    static_assert(MAX_PID > MAX_PROCESS);
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    if (++last_pid >= MAX_PID)
    {
        last_pid = 1;
        goto inside;
    }
    if (last_pid >= next_safe)
    {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        while ((le = list_next(le)) != list)
        {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid)
            {
                if (++last_pid >= next_safe)
                {
                    if (last_pid >= MAX_PID)
                    {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;
                }
```

```
            }
            else if (proc->pid > last_pid && next_safe > proc->pid)
            {
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
}
```

我们可以由此回答指导书的问题2：**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。**

可以看到，我们通过维护 `last_pid` 和 `next_safe` 两个关键变量来实现 PID 的唯一性保障。在分配PID时，我们会对整个进程链表进行完整遍历，确保当前要分配的 PID 没有被任何现有进程使用。当检测到冲突时，算法会递增 PID 并重新扫描，这种重试机制理论上能够避免 PID 重复分配。此外，算法还通过 `next_safe` 变量记录下一个已被占用 PID 的最小值，从而优化了查找效率，避免每次都需要完整扫描链表。

而潜在问题在于 `last_pid` 的递增操作和进程链表的遍历检查不是原子操作。因此可能会出现以下场景：线程 A 执行 `get_pid()` 通过检查后，在返回 PID 之前被线程 B 中断，线程 B 也可能分配相同的 PID，导致两个线程获得重复的 PID。这种竞争条件会破坏 PID 的唯一性保证。

因此，在 `do_fork` 中，我们最好使用关中断操作来保护临界区，这样能够保证 PID 的唯一性。开关中断我们使用了已有的 `local_intr_save` 和 `local_intr_restore` 宏。添加到就绪队列后，我们需要更新 `nr_process` 的值，让当前进程数加一。

这样，分配得到的唯一 PID 会通过 `hash_proc` 函数添加到哈希表中，然后我们再将其添加到就绪队列中。

最后，我们将子进程唤醒，用 `wakeup_proc` 函数将子进程状态设置为就绪态，并返回子进程的PID：

```
wakeup_proc(proc);
ret = proc->pid;
```

练习3：编写proc_run 函数（需要编码）

最后我们需要实现 `proc_run` 函数，`proc_run` 用于将指定的进程切换到CPU上运行。我们首先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。接下来在切换过程中，我们需要禁用中断，切换页表，切换上下文，最后允许中断。

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            lsatp(next->pgdir);
            switch_to(&prev->context,&next->context);
```

```
            }
            local_intr_restore(intr_flag);
        }
    }
```

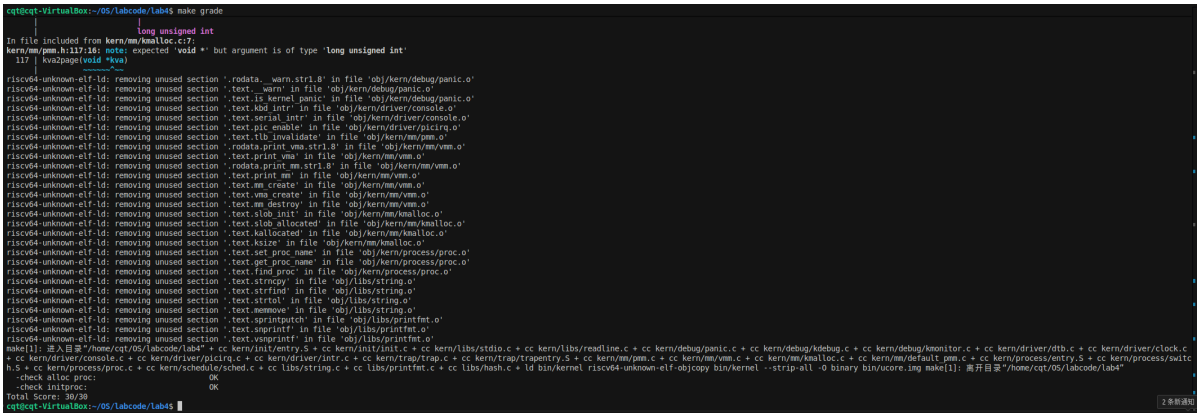
我们用`proc != current`这个条件来检查要切换的进程是否与当前正在运行的进程相同。在切换页表时，我们调用预先定义的宏`lsatp`，传入我们要切换的进程的页表，然后使用`switch_to`函数切换上下文，`switch_to`需要传入两个参数，第一个参数是前一个进程的上下文，第二个参数是后一个进程的上下文。最后我们再允许中断。这样就完成了`proc_run`函数的实现。

现在我们回答指导书的问题3：**在本实验的执行过程中，创建且运行了几个内核线程?**

通过阅读代码，我们可以看到，在本实验的执行过程中，共创建且运行了两个内核线程。一个是空闲进程`idleproc`，这是第一个内核线程，在`proc_init()`函数中创建，它的PID为0，状态设置为`PROC_RUNNABLE`，负责在系统没有其他可运行进程时占用CPU，并执行调度检查。我们在`proc_init()`中直接设置为当前进程，并立即运行。另一个是初始化进程initproc，这是第二个内核线程，在`proc_init()`中通过`kernel_thread()`函数创建，它的PID为1，入口函数为`init_main`，参数为`"Hello world!!"`。创建后状态设置为`PROC_RUNNABLE`，并通过`wakeup_proc()`唤醒。

在`cpu_idle()`循环中，当调度器被触发时，`initproc`会被选中并运行，然后执行初始化任务并打印消息。

练习题测试结果



Challenge1：说明语句`local_intr_save(intr_flag);....local_intr_restore(intr_flag);`是如何实现开关中断的?

点开此代码跳转到

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

然后发现是定义的代码，最原始的代码应该是sync.h代码文件最上面的代码

```
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();
    }
}
```

详细分析代码：

`__intr_save` 函数

```
static inline bool __intr_save(void) {
```

`static inline`：表示这是内联函数，编译时直接展开，减少函数调用开销

`bool`：返回布尔类型，**根据之前的中断状态，禁用中断**

`void`：无参数

```
if (read_csr(sstatus) & SSTATUS_SIE) {
```

`read_csr(sstatus)`：读取控制状态寄存器（CSR）中的sstatus寄存器

`SSTATUS_SIE`：中断使能位的掩码（Supervisor Interrupt Enable）

`&`：位与操作，检查中断使能位是否为1

如果条件为真，说明当前中断是启用的

```
intr_disable(); // 禁用中断
return 1; // 返回true，表示之前中断是启用的
```

返回true，表示之前中断是启用的

```
return 0; // 返回false，表示之前中断已是禁用的
```

返回false，告诉调用者之前中断状态已经是禁用的

`__intr_restore` 函数

```
static inline void __intr_restore(bool flag) {
```

接收一个布尔参数，表示**要恢复的中断状态**

```
if (flag) {
    intr_enable();
}
```

如果flag为true，表示**之前中断是启用的，则重新启用中断**

如果flag为false，什么都不做，保持中断禁用状态

中断开关的实现机制

工作原理

1. **保存状态**：`__intr_save` 检查当前中断状态并立即禁用中断，同时返回之前的状态
2. **执行关键代码**：在中断禁用的状态下执行需要原子性的操作
3. **恢复状态**：`__intr_restore` 根据保存的状态恢复原来的中断设置

典型例子

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

Challenge2：深入理解不同分页模式的工作原理（思考题）

一、get_pte()中两段类似代码的原因及与sv32/sv39/sv48的关系

get_pte()函数中有两段形式类似的代码，分别用于处理不同级别的页表（如sv39有三级页表，sv32有两级，sv48有四级）。每一级页表都需要判断当前页表项是否有效（PTE_V），如果无效则分配新页并初始化，然后进入下一级页表。代码结构如下：

```
pde_t *pdep1 = &pgdir[PDX1(la)];
if (!(*pdep1 & PTE_V)) {
    // 分配新页，初始化
}
pde_t *pdep0 = &((pte_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
if (!(*pdep0 & PTE_V)) {
    // 分配新页，初始化
}
return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
```

这种结构相似，是因为无论是sv32、sv39还是sv48，页表查找和分配的流程都是“逐级查找，每级无效则分配新页”。不同的只是页表的级数和每级索引的位数。例如：

- sv32：两级页表（一级PDE，二级PTE）
- sv39：三级页表（PDE1、PDE0、PTE）
- sv48：四级页表

所以，代码结构会随着级数增加而递归/循环地重复，表现为“形式类似”。本质上是多级页表机制的通用实现。

二、查找和分配合并在一个函数里是否合理？是否需要拆分？

优点：

1. 使用方便：调用者只需一个接口即可完成查找和分配，减少了代码量，简化了调用流程。
2. 自动化：在需要分配新页表项时，自动完成分配，适合内核自动扩展页表的场景。
3. 统一管理：减少了接口数量，便于维护。

缺点：

1. 灵活性不足：有些场景只需要查找页表项，不希望自动分配新页（比如只读查询），此时合并接口会导致不必要的内存分配，浪费资源。
2. 可读性降低：查找和分配逻辑混在一起，代码不易理解，后续维护和调试变得困难。
3. 错误处理复杂：分配失败和查找失败的处理方式不同，合并后不易区分和处理异常。
4. 不利于扩展：如果后续需要支持更多页表操作（如只查找、只分配、批量分配等），合并接口会限制功能扩展。

因此，将查找和分配功能拆分为两个独立的函数更好，这样可以实现：

- 灵活控制：调用者可以根据实际需求选择只查找或查找+分配，避免不必要的内存分配。
- 代码清晰：每个函数职责单一，易于理解和维护。
- 错误处理明确：查找失败和分配失败可以分别处理，提升健壮性。
- 便于扩展：后续可以根据需要增加更多功能接口。
