南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

操作系统实验报告

Lab4: 进程管理

年级：2023 级

姓名：康志杰，王春晖，彭浩然

2025 年 11 月 23 日

目录

一、 实验目的

- **虚拟内存结构掌握**：理解多级页表（sv39）组织、管理方式，实现地址空间布局，为进程/线程提供独立逻辑空间。
- **内核线程创建/执行**:学习 PCB 分配/初始化、内核栈分配、中断帧复制,通过 do_fork/kernel_thread 实现线程创建。
- **内核线程切换/调度**：掌握上下文切换（switch_to 保存/恢复寄存器）、页表切换（SATP 修改）、FIFO 调度（链表轮转），实现多线程并发。

通过实验，从单执行流扩展多线程内核，奠基用户进程/系统调用/页面置换。

二、 实验原理

(一) 虚拟内存管理原理

虚拟内存机制通过多级页表实现虚拟地址（VA）到物理地址（PA）的映射，支持地址空间隔离和预分配（本实验采用一次性页表映射，无缺页处理）。RISC-V sv39 模式使用三级页表：顶级页目录（PDX1，9 位）、二级页目录（PDX0，9 位）和页表（PTX，9 位），加上页内偏移（PGOFF，12 位）。物理地址为 PPN[2:0]（26+9+9 位）+ PGOFF。

- **页表项（PTE）结构**：包含 PPN（物理页号）、权限位（PTE_V: 有效、PTE_R/W/X: 读/写/执行、PTE_U: 用户）和保留位。宏如 PDX1(la)、PTE_ADDR(pte) 用于地址分解/构造。
- **get_pte 实现**：递归查找 PTE。先定位 pgdir[PDX1(la)]，若无效（!PTE_V）且 create=true，则 alloc_page 分配页、清零（memset(KADDR(pa), 0, PGSIZE)），设置 PTE（pte_create(ppn, PTE_V | PTE_U)）。类似逻辑应用于 PDX0，返回 PTX 位置的 PTE 指针。两段代码相似因 sv39 三级结构对称（PDX1/PDX0 递进），异同：级别不同但检查/分配/清零/设置逻辑一致；sv32（两级）/sv48（四级）可扩展类似循环。
  - **设计评价**：合并查找/分配便利（单函数处理动态页表），但耦合高；建议拆分 find_pte（仅查找）和 create_pte（分配），提升模块化/可测试性，尤其多模式支持。
- **page_insert/page_remove**：page_insert 调用 get_pte(...,1)，增引用（page_ref_inc），替换旧 PTE（page_remove_pte 清引用/释放页），新建 PTE（pte_create + perm），tlb_invalidate 刷新 TLB。page_remove 调用 get_pte(...,0) + page_remove_pte(清 PTE=0，减引用，若 0 则 free_page)。
- **check_pgdir 验证**：分配页（alloc_page），插入/替换映射（page_insert），检查 PTE（get_pte + pte2page）、引用（page_ref ==1/2/0），移除（page_remove），确保多级动态分配/权限（PTE_U/W）正确。

内核空间经 boot_pgdir 静态映射（初始 Giga Page），后续重映射细化段权限（.text: R/X, .data/.bss: R/W）。内核线程共享 pgdir=boot_pgdir，用户进程用 mm_struct 管理 VMA（虚拟内存区域：创建/查找/插入/销毁）。

(二) 内核线程管理原理

内核线程为轻量进程，仅内核态运行，共享地址空间。由 struct proc_struct 描述：状态(PROC_UNINIT/RUNNABLE/SLEEPING/ZOMBIE)、PID、runs（运行次数)、kstack（2 页内核栈)、need_resched（调度标志)、parent（父进程，形成树)、mm（用户内存，内核线程 NULL)、context（ra/sp/s0-s11，上下文切换用)、tf（trapframe，中断帧)、pgdir（页表根 PA)、flags、name、list_link（proc_list 双向链表)、hash_link（PID 哈希表)。

- **全局管理**: current（当前进程)、initproc（首用户进程)、proc_list（所有进程链表)、hash_list（哈希，防冲突)。
- **idleproc 创建（PID=0)**：proc_init 初始化链表，alloc_proc（kmalloc 分配 PCB，清零，state=UNINIT, pid=-1, pgdir=boot_pgdir),后设 pid=0, state=RUNNABLE, kstack=bootstack, need_resched=1, name="idle"。执行 cpu_idle：循环若 need_resched=1 则 schedule。
- **initproc 创建(PID=1)**: kernel_thread(fn, arg, flags) 构造 tf(memset=0, s0=fn, s1=arg, status=(sstatus | SPP | SPIE) & SIE, epc=kernel_thread_entry),do_fork(CLONE_VM)。
  - **do_fork**:alloc_proc → setup_kstack(get_ppages(2) 分配栈)→ copy_mm(mm=NULL) → copy_thread（tf 置新栈顶：*proc->tf=*tf, a0=0 子进程标识, sp=tf 或 esp, context.ra=forkret, sp=tf）→ 插入链表（hash/proc_list）→ state=RUNNABLE → ret pid（next_pid++ 确保唯一：全局递增 + 哈希链)。
- **proc_struct 成员作用**:
  - **context**：保存被调用者寄存器，用于 switch_to（汇编 STORE/LOAD 14 寄存器到 PCB)，编译器处理 caller-saved。
  - **tf**: 保存中断现场 (gpr/epc/status), forkret→___trapret 恢复; kernel_thread_entry: a0=s1, jalr s0(fn), 后 do_exit。
- **调度/切换**: FIFO (schedule: 清 need_resched, 从 proc_list 轮转找 RUNNABLE next, 若无 idleproc; proc_run(next): 若 current==next ret; local_intr_save/restore 关中断; current=next; lsatp(pgdir»12) 切页表; switch_to(&from->context, &to->context) 切上下文; 开中断。执行: idle→schedule→proc_run(init)→switch_to→forkret(sp=tf)→___trapret→kernel_thread_entry→init_main("Hello World")。

本实验运行 2 线程：idleproc（空闲循环)、initproc（打印)。

三、 实验过程

(一) 练习 1

1. 代码实现

首先在代码框架中补充 Lab3 中的代码，主要是补充时钟中断实现的代码。练习 1 需要实现 alloc_proc 函数，该函数主要功能为实现分配并返回进程控制块。结构体 proc_struct 主要属性如下所示。该结构体包含了进程的主要信息，比如进程状态，进程内核栈，父进程，上下文等。

```
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
};
```

根据 proc_struct 结构体的主要属性，alloc_proc 函数实现代码如下。首先调用函数 kmalloc 函数为进程控制块分配内存，内存块大小为 proc_struct 的字节大小。如果内存块分配成功就可以对结构体属性进行初始化。首先进程状态 state 赋值为 PROC_UNINIT，表示进程还没有初始化。合法的 pid 是从 0 开始累加，此时只是初始化进程控制块，所以进程的 pid 赋值为-1。runs 表示进程运行次数，赋值为 0 表示该进程没有执行过。

```
static struct proc_struct *alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        proc->state=PROC_UNINIT;//将进程状态设置为未初始化。
        proc->pid= -1;//进程id设置为-1.合法的PID是从0开始的
        proc->runs=0;//该进程还未运行，所以运行次数为0
```

接着 kstack 表示进程内核栈地址，但此时内核栈还没有分配，所以初始化内核栈地址为 0，表示无效地址。初始化 need_resched 为 0，表示当前进程不需要调度。parent 是该进程的父进程，将父进程指针初始化为 NULL，在后续会设置为创建它的进程。mm 存储内存管理的信息，初始化将其指针赋值为 NULL。context 用于存储进程的上下文信息，即几个关键寄存器的值。初始状态下还没有有效的上下文信息，所以调用函数 memset 将其清空。

```
        proc->kstack=0;//此时内核栈还没有初始化
        proc->need_resched=0;
        proc->parent=NULL;//父进程为空
        proc->mm=NULL;
        memset(&proc->context, 0, sizeof(struct context)); // 上下文清零
```

在上一个实验中已经了解到 tf 为中断帧指针，初始化条件下将指针赋值为 NULL，表示进程还没有有效的中断帧。pgdir 保存进程的页表根节点的物理地址，当发生进程切换时，将 pgdir 存储

的页表地址加载到 satp 寄存器中，从而实现内存空间的切换。ucore OS 这个内核进程管理着所有内核线程，所以内核线程不需要再建立各自的页表，只需共享内核虚拟空间就可以访问整个物理内存。综合前面的分析，pgdir 初始化为内核页目录表的基址 boot_pgdir_pa。后续如果为用户进程，可以切换到用户的页目录。初始化进程标志位为 0，表示无特殊标志。最后调用 memset 将进程名称清空。到此进程控制块初始化结束，返回进程控制块指针即可。

```
        proc->tf=NULL;//
        proc->pgdir=boot_pgdir_pa;//使用内核页目录表的基址
        proc->flags=0;//标志位清零
        memset(&proc->name,0,PROC_NAME_LEN+1);//进程名称清零
    }
    return proc;
}
```

以上即为 alloc_proc 函数实现过程，以下为该函数的整体代码：

```
static struct proc_struct *alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        proc->state=PROC_UNINIT;//将进程状态设置为未初始化。
        proc->pid= -1;//进程id设置为-1.合法的PID是从0开始的
        proc->runs=0;//该进程还未运行，所以运行次数为0
        proc->kstack=0;//此时内核栈还没有初始化
        proc->need_resched=0;
        proc->parent=NULL;//父进程为空
        proc->mm=NULL;
        memset(&proc->context, 0, sizeof(struct context)); // 上下文清零
        proc->tf=NULL;//
        proc->pgdir=boot_pgdir_pa;//使用内核页目录表的基址
        proc->flags=0;//标志位清零
        memset(&proc->name,0,PROC_NAME_LEN+1);//进程名称清零
    }
    return proc;
}
```

2. 问题回答

在前面的分析过程中已知 context 为进程的上下文，包括 ra，sp，s0-s11 这几个寄存器的值。context 保存着进程的状态，是进程切换时需要保存和恢复的核心数据。本次实验会有第 0 个进程 idleproc 切换到第 1 个进程 initproc，在 proc_run 函数中会调用 switch_to 函数保存当前进程 idleproc 的 context，切换到目标进程 initproc 的 context。此时目标进程可以正常执行，在后续实验中可以观察到该进程输出内容，表示进程切换成功。

tf 是进程的中断帧，用于保存进程在运行时发生中断、异常或系统调用时的完整寄存器状态。本次实验会有时钟中断产生，每当时钟中断来临时，进程会将所有寄存器的值存储到 tf 结构体中。内核在处理时钟中断后通过 tf 中的寄存器值恢复到发生中断的位置继续执行之后的代码。

(二) 练习 2

1. 代码实现

练习 2 主要实现 do_fork 函数，该函数的主要作用为从已有的进程创建子进程。该函数主要分为如下步骤:

1, 判断进程数量: 首先将返回值赋值为-E_NO_FREE_PROC 表示没有空闲进程，如果此时进程数量超过最大进程数量 MAX_PROCESS，会直接 goto 到 fork_out 那里返回无空闲进程的错误。

```
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;
    }
```

2, 创建进程控制块: 接着 ret 赋值为-E_NO_MEM 表示内存不足，调用 alloc_proc 为新进程创建进程控制块。如果返回值为 NULL 表示初始化失败，说明内存不足。这时进程控制块的内存未分配，所以直接 goto 到 fork_out，返回内存不足错误。

```
    ret = -E_NO_MEM;
    proc= alloc_proc();//初始化
    if(proc==NULL){
        goto fork_out;
     }
```

3, 构建内核栈: 如果进程控制块初始化成功，则调用函数 setup_kstack 构建进程内核栈，如果构建内核栈因为内存不足导致失败，则 goto 到 bad_fork_cleanup_proc，先调用 kfree 函数回收分配给进程控制块的内存，然后返回内存不足错误。

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

4, 复制内存管理系统: 调用函数 copy_mm 复制内存管理系统，该函数会根据 clone_flags 判断是复制还是共享内存管理系统。mm 描述的是进程用户态空间的情况，本次实验创建的是内核态线程，所以暂且将其赋值为 NULL。如果该函数返回值不是 0 表示没有成功赋值内存管理系统。此时需要 goto 到 bad_fork_cleanup_kstack, 先要调用函数 put_kstack 释放内核栈占有的内存，由于 bad_fork_cleanup_kstack 后面没有 goto 语句，会接着执行 bad_fork_cleanup_proc，调用函数 kfree 释放进程控制块内存，最后返回内存不足错误。

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

    }
```

5, 设置进程中断帧和上下文:do_fork 函数形参列表中给出了父进程的内核栈指针和中断帧 tf 指针。所以调用 copy_thread 函数设置中断帧和上下文。此时是创建子进程，所以 copy_thread 函

数的栈指针 esp 为 0。在该函数中进程的中断帧被放置在内核栈栈顶位置。子进程将父进程的 tf 复制了一份。tf 中的 a0 寄存器被赋值为 0，表示该进程为子进程。

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

6, 将进程添加到 hash 列表和进程列表: 首先调用函数 get_id 得到一个进程 ID，并且将其赋值给 pid。接着调用函数 hash_proc，该函数根据宏 pid_hashfn 和进程的 pid 计算进程所在的哈希桶位置，接着使用 list_add 将该进程指针链接到对应哈希桶中的链表上。接着调用 list_add 将进程链接到进程链表 proc_list 中，此时进程数量 nr_process 自增 1。

```
    int pid=get_pid();
    proc->pid=pid;
    hash_proc(proc);
    list_add(&proc_list,&proc->list_link);
    nr_process++;
    wakeup_proc(proc);
```

7, 唤醒进程并返回新进程 ID: 调用函数 wakeup_proc 将进程的 state 设置为 PROC_RUNNABLE，表示进程处于就绪状态。接着将 ret 赋值为新进程的 pid。最后会执行到 fork_out 并返回新进程的 ID。

```
    wakeup_proc(proc);
     ret=proc->pid;
fork_out:
    return ret;

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

2. 问题回答

已知进程的 ID 是从 get_id 函数中获取的。所以要判断 ucore 是否为每个新进程分配唯一的 ID，就是要判断该函数是否每次返回的 ID 号是唯一存在的。MAX_PID 是 pid 的最大值，MAX_PROCESS 是进程的最大数量，首先断言确保 pid 要大于最大进程数量，否则 pid 不能保证分配到每个进程。last_id 记录上一次分配的 id，next_safe 记录当前已分配的 PID 中，大于 last_pid 的最小 PID 值。

第一次调用该函数会将两个静态变量都赋值为 MAX_PID，尝试分配的 last_id 从 1 开始。之后每次调用函数时尝试分配 last_id 自增 1 后的 pid。如果满足后面的 if 分支条件说明发生冲突，将 last_pid 赋值为 1 并跳转到 inside 检查冲突。

```
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;
    }
```

区间 [last_pid, next_safe) 是安全范围，无需遍历进程列表就可以直接分配 PID。如果 last_pid >= next_safe，说明可能存在冲突。先将 next_safe 赋值为 MAX_PID，接着需要遍历进程列表。while 循环中通过 list_next 遍历 proc_list 进程列表，le2proc 会将链表节点转换为进程控制块节点 (实现原理与前面将链表节点转换为页节点一样)。

如果当前尝试分配的 last_pid 与进程链表中的某个进程 pid 重复，则递增尝试分配下一个 pid，但是如果新的 last_id 超出了安全范围将 last_id 赋值为 1，重置 next_safe 为 MAX_PID，跳转到 repeat 再次遍历链表。如果进程 pid 大于将要分配的 last_pid 并且安全范围大于进程的 pid 则更新 next_safe。之后返回不冲突的 last_id。冲突处理保证了每一次分配的 pid 是唯一的。所以给每个进程分配的 pid 都是唯一的。

```
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;
```

(三) 练习 3

1. 代码实现

本练习需要实现 proc_run 代码。该函数作用为将指定的进程切换到 CPU 上运行，实现的代码如下。实现进程上下文切换必须是两个不同的进程，如果 proc 和 current 是同一个进程就没有切换的必要。进程上下文切换是一个原子操作，所以必须调用宏 local_intr_save 在上下文切换之前关闭中断。切换完成之后调用宏 local_intr_restore 再打开中端。

进程上下文切换首先需要将 proc 设置为当前运行的进程 current，接着调用函数 lsatp 将 satp 寄存器的值修改为当前进程的页目录地址，切换到新的内存空间。最后调用函数 switch_to 将 pre 进程的上下文保存起来，将 next 进程的上下文进行切换。到此该函数已经完成了上下文切换。

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
        bool flags=0;
        struct proc_struct*pre=current,*next=proc;
        local_intr_save(flags);
        current=proc;
        lsatp(proc->pgdir);//修改satp寄存器中的值
        switch_to(&pre->context,&next->context);
        local_intr_restore(flags);
    }
}
```

2. 问题回答

在本次实验中创建且运行了两个进程，分别为 idleproc 和 initproc。首先在函数 proc_init 函数中调用 alloc_proc 为 idleproc 函数创建进程控制块，并且为该进程的参数进行赋值。值得关注的是 idleproc 进程的 pid 为 0，state 为 PROC_RUNNABLE 状态，need_resched 赋值为 1。表明 idle 为内核创建的第 0 个进程。将 current 赋值为 idleproc，当前运行的进程为 idleproc。

```
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);
```

```
14
15      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)
16      {
17          cprintf("alloc_proc() correct!\n");
18      }
19
20      idleproc->pid = 0;
21      idleproc->state = PROC_RUNNABLE;
22      idleproc->kstack = (uintptr_t)bootstack;
23      idleproc->need_resched = 1;
24      set_proc_name(idleproc, "idle");
25      nr_process++;
26
27      current = idleproc;
```

接着会调用函数 kernel_thread 创建了第二个内核进程 initproc，进程的 pid 为 1。而 proc_init 会在 init 函数中调用。调用时就会创建这两个进程。

```
1   int pid = kernel_thread(init_main, "Hello world!!", 0);
2       if (pid <= 0)
3       {
4           panic("create init_main failed.\n");
5       }
```

如下代码所示，最后在 init 函数中调用函数 cpu_idle，此时 current 为 idleproc，而 idleproc 的 need_resched 等于 1。此时会进入 if 分支并调用 schedule 函数，该函数会找到下一个 state 为 PROC_RUNNABLE 的进程，此时只有 initproc 的为就绪状态，所以空闲进程 idleproc 会将运行权交给 initproc 进程，此时 initproc 进程处于运行状态。

```
1   void cpu_idle(void)
2   {
3       while (1)
4       {
5           if (current->need_resched)
6           {
7               schedule();
8           }
9       }
10  }
```

3. 实现结果

到此本次实验的三个函数均已实现，现在输入 make qemu 命令得到如图1所示结果，输出结果显示 alloc_proc 成功，说明第 0 号进程 idleproc 的进程控制块创建成功。还显示第一个进程

init 运行成功，pid 也如预期的为 1，当 initproc 进程运行时会调用 init_main 函数输出如图字符串。

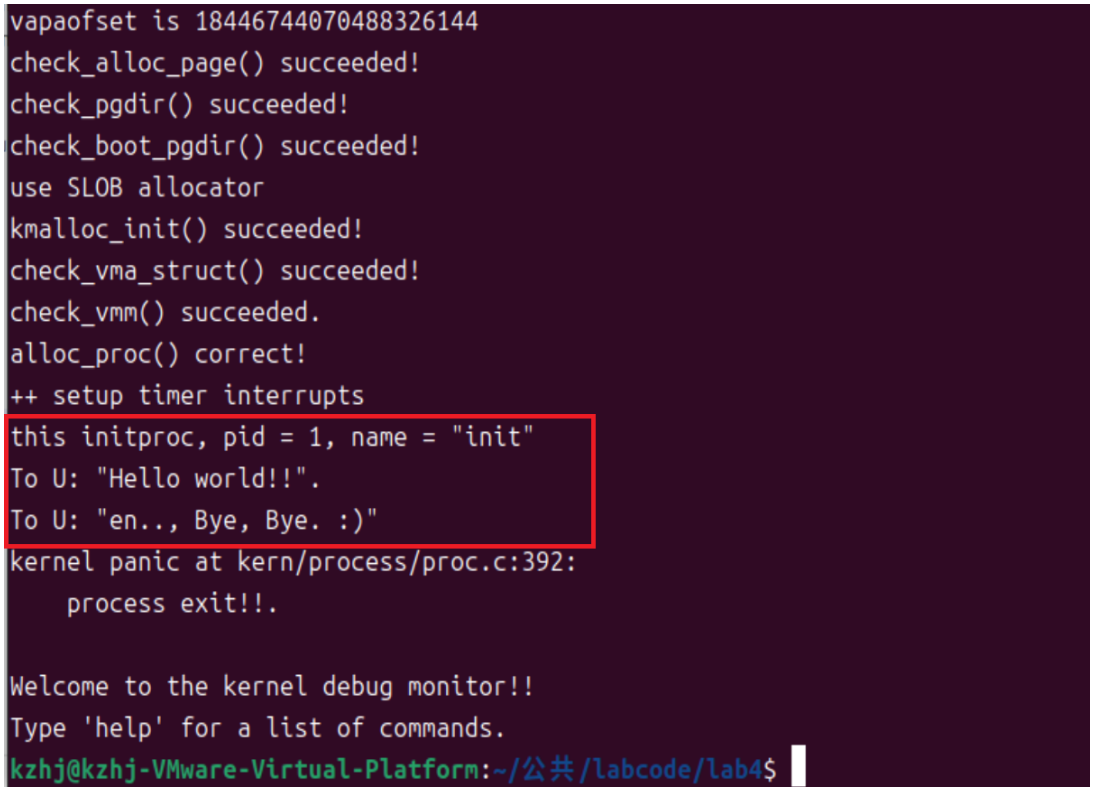

图 1: make qemu 运行结果

接着在命令行中输入 make grade 命令，得到如图2所示结果，结果显示得分为 30 分，说明输出结果正确，即代码实现正确。

图 2: make grade 运行结果

(四) 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?

local_intr_save(intr_flag)：csrr t0, sstatus 保存 eflags=*intr_flag；if(SIE) csrc sstatus, SIE

(禁用)。首先读取当前处理器状态寄存器（如 RISC-V 中的 sstatus 寄存器）的值，该寄存器中包含中断使能标志位（如 sstatus 中的 SIE 位，用于控制 supervisor 模式下的中断使能)。将读取到的状态值保存到 intr_flag 变量中，以记录中断的原始状态。清除状态寄存器中的中断使能标志位，从而关闭中断，防止在临界区执行期间被中断干扰。

local_intr_restore：if(eflags SIE) csrs sstatus, SIE（恢复)。将保存在 intr_flag 变量中的原始状态值写回到处理器状态寄存器中。这样就恢复了中断的原始使能状态，如果之前中断是开启的，恢复后中断继续保持开启；如果之前是关闭的，恢复后仍然关闭。

原理：eflags 原子捕获/恢复 SIE，确保嵌套安全/原子性（proc_run 中禁用 → 切换 → 恢复，避免中断竞态)。

2. 深入理解不同分页模式的工作原理,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。

(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。

get_pte () 函数中两段形式类似的代码，核心原因是 SV32、SV39、SV48 虽存在虚拟地址位数、页表级数、各级 VPN 位数的差异（如 SV32 为 32 位虚拟地址、两级页表，SV39 为 39 位虚拟地址、三级页表，SV48 为 48 位虚拟地址、四级页表)，但分页机制的核心逻辑具有一致性——均通过“按级索引页表”实现虚拟地址到物理地址的转换，每一级页表的结构（如每个页表包含 512 个页表项)、操作逻辑（根据对应级别的 VPN 索引页表项，判断页表项是否存在，不存在则需分配新页表）完全相同。两段类似代码本质是对应不同级别页表的处理流程，仅索引的 VPN 片段不同，核心操作（查找页表项、检查有效性、分配新页表并更新映射）高度一致，因此代码形式呈现出明显相似性。

(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？

将页表项的查找和分配合并在 get_pte () 函数中的写法并不理想，有必要拆分为两个独立函数。这种合并写法虽能减少函数调用开销、让完整流程集中呈现，但违背了单一职责原则，导致函数功能冗余，降低了代码的可读性和可维护性——修改查找逻辑可能影响分配逻辑，反之亦然；同时缺乏复用性，若其他场景仅需查找页表项（无需分配）或仅需分配页表项（无需查找)，无法直接复用该函数，需重复编写代码。拆分后，一个函数专注于页表项查找（仅返回已有页表项或提示不存在)，另一个函数专注于页表项分配（在查找失败时创建新页表并生成页表项)，能让函数职责更明确，提升代码的可维护性、复用性和灵活性，适配不同场景的使用需求。

四、 实验总结

(一) 实验核心知识点与 OS 原理对应分析

1. **实验知识点：多级页表（SV39）实现虚拟内存映射**

   - **OS 原理知识点：**虚拟内存与多级页表机制
   - **含义理解：**
     - 实验中：通过 `getpte()`、`pageinsert()`、`pageremove()` 等函数，基于 SV39 三级页表结构，实现虚拟地址到物理地址的映射建立、删除与查找，支持页表项的动态分配与回收，同时通过页表项权限位（PTE_R/PTE_W/PTE_X/PTE_U）实现内存访问保护。

  - 原理中：虚拟内存是操作系统为进程提供的抽象地址空间，通过多级页表减少页表占用的物理内存，实现地址空间隔离与访问控制；按需分页、页表映射是虚拟内存的核心机制。

- **关系与差异：**
  - 关系：实验是原理的具体实现，严格遵循“虚拟地址拆分 → 多级页表索引 → 物理地址转换”的核心逻辑，权限位设计直接对应原理中的内存保护机制。
  - 差异：实验聚焦 SV39 架构的三级页表实现，未涉及原理中的页换入换出（Swap）、缺页异常处理等完整虚拟内存功能，仅实现静态映射与手动映射管理。

2. **实验知识点：进程控制块（proc_struct）与进程管理**

- **OS 原理知识点：**进程控制块（PCB）与进程描述
- **含义理解：**
  - 实验中：`proc_struct` 存储进程状态（state）、PID、内核栈（kstack）、上下文（context）、页表基址（pgdir）等关键信息，是内核管理进程的核心数据结构；通过 `proc_list` 链表和哈希表维护所有进程的生命周期。
  - 原理中：PCB 是操作系统感知进程存在的唯一标识，包含进程的所有运行状态信息，是进程调度、资源分配与回收的基础。
- **关系与差异：**
  - 关系：实验中的 `proc_struct` 完全对应原理中的 PCB，成员变量（状态、PID、资源指针等）与原理定义高度一致，链表管理方式是 PCB 组织的典型实现。
  - 差异：实验仅实现内核线程的 PCB 管理，未涉及用户进程的资源（如文件描述符、信号量）管理，成员变量（如 `mm`）暂未完全启用。

3. **实验知识点：进程上下文切换（switch_to 函数）**

- **OS 原理知识点：**进程上下文切换
- **含义理解：**
  - 实验中：通过汇编函数 `switch_to` 保存当前进程的被调用者保存寄存器（ra、sp、s0 s11），恢复目标进程的寄存器上下文；结合 `context` 结构体和内核栈，实现 CPU 执行权的切换。
  - 原理中：上下文切换是调度器将 CPU 从一个进程切换到另一个进程的过程，需保存进程的 CPU 状态（寄存器、程序计数器等），恢复目标进程的状态以保证执行连续性。
- **关系与差异：**
  - 关系：实验严格遵循原理中的上下文切换逻辑，利用“调用者保存寄存器由编译器处理、被调用者保存寄存器手动保存”的优化策略，是原理的高效实现。
  - 差异：实验仅实现内核态上下文切换，未涉及用户态与内核态切换时的特权级转换细节（如 SSTATUS 寄存器的完整配置）。

4. **实验知识点：内核线程创建（kernel_thread/do_fork 函数）**

- **OS 原理知识点：**进程创建与 fork 系统调用
- **含义理解：**

  - 实验中：通过 `kernel_thread` 构造中断帧（`trapframe`），`do_fork` 分配 PCB、初始化内核栈、复制上下文，最终创建内核线程；`idleproc`（第 0 个线程）继承内核初始上下文，`initproc`（第 1 个线程）通过手动构造中断帧创建。
  - 原理中：进程创建通过复制父进程资源（PCB、地址空间等）实现，fork 系统调用是典型接口，子进程继承父进程的资源并独立运行。
- **关系与差异：**
  - 关系：`do_fork` 对应原理中 fork 的核心逻辑（分配 PCB、复制状态），中断帧构造对应子进程执行上下文的初始化。
  - 差异：实验仅支持内核线程创建，无父进程资源复制（如地址空间共享而非复制），未实现 fork 的用户态接口与写时复制（Copy-on-Write）机制。

5. **实验知识点：FIFO 进程调度（schedule 函数）**
   - **OS 原理知识点：**进程调度算法
   - **含义理解：**
     - 实验中：`schedule` 函数遍历 `proc_list` 链表，选择处于 `PROC_RUNNABLE` 状态的下一个进程，通过 `proc_run` 触发上下文切换，实现简单的 FIFO 调度。
     - 原理中：调度算法是操作系统分配 CPU 资源的策略，FIFO 是最简单的非抢占式调度算法，按进程就绪顺序分配 CPU。
   - **关系与差异：**
     - 关系：实验的调度逻辑完全对应原理中的 FIFO 算法，是调度器的最小可行实现。
     - 差异：实验无抢占式调度（仅通过 `need_resched` 标志触发调度），未实现时间片轮转、优先级调度等复杂算法。
6. **实验知识点：内存保护（页表项权限位设置）**
   - **OS 原理知识点：**内存访问控制与隔离
   - **含义理解：**
     - 实验中：通过页表项的 PTE_U（用户态访问）、PTE_R/W/X（读写执行权限）控制虚拟地址的访问权限，避免越权访问（如修改内核.text 段）。
     - 原理中：内存保护是操作系统保证进程地址空间隔离的核心机制，通过页表权限位、段寄存器等限制进程对内存的访问范围与操作类型。
   - **关系与差异：**
     - 关系：实验直接采用原理中的页表权限位设计，是内存保护的硬件辅助实现（依赖 RISC-V 的 MMU 对权限位的检查）。
     - 差异：实验仅实现静态权限配置，未涉及动态权限修改（如进程运行时调整访问权限）。
7. **实验知识点：Slab 分配器简化版（kmalloc/kfree）**
   - **OS 原理知识点：**内存分配算法（Slab 分配器）
   - **含义理解：**
     - 实验中：`kmalloc`/`kfree` 基于简化的 Slab 算法，为内核对象（如 `proc_struct`、页表项）提供高效的小内存分配与回收，避免内存碎片。

  - 原理中：Slab 分配器是为内核对象设计的缓存分配机制，将相同大小的对象分组管理，提高分配效率并减少碎片。

- **关系与差异：**
  - 关系：实验是 Slab 分配器的核心思想简化实现，聚焦“对象缓存、批量分配”的核心逻辑。
  - 差异：实验未实现 Slab 分配器的高级特性（如对象构造/析构钩子、缓存收缩策略）。

（二） OS 原理中重要但实验未涉及的知识点

1. **用户进程管理：**原理中用户进程是操作系统的核心管理对象，涉及用户态地址空间、用户栈、系统调用接口等；实验仅实现内核线程，未涉及用户进程的创建、切换与资源管理。
2. **缺页异常与按需分页：**原理中虚拟内存的核心特性之一，通过缺页异常触发物理内存分配与页映射；实验仅实现静态映射，无缺页异常处理流程。
3. **页换入换出（Swap）：**原理中通过磁盘扩展虚拟内存空间，将不常用页面换出到磁盘，需用时换入；实验未涉及磁盘交互，无 Swap 机制。
4. **复杂调度算法：**原理中时间片轮转、优先级调度、多级反馈队列等调度算法是进程调度的核心；实验仅实现简单 FIFO 调度，无抢占式调度与时间片管理。
5. **进程同步与通信：**原理中进程间通过信号量、管道、消息队列等机制同步与通信；实验中进程（内核线程）无交互，未涉及同步与通信机制。
6. **死锁检测与避免：**原理中死锁是多进程资源竞争的重要问题，需通过算法检测或避免；实验进程数量少且无资源竞争，未涉及死锁相关机制。
7. **内存碎片整理：**原理中内存分配会产生内碎片与外碎片，需通过紧凑、伙伴系统等机制整理；实验的 Slab 简化版未涉及碎片整理策略。
8. **用户态与内核态切换细节：**原理中用户进程通过系统调用、中断进入内核态，需完整保存用户态上下文；实验仅涉及内核态线程切换，无用户态与内核态的切换流程。