



# 华中科技大学

## 操作系统原理课程实验报告

姓 名： 董玲晶  
学 院： 计算机科学与技术学院  
专 业： 计算机科学与技术  
班 级： CS2005 班  
学 号： U202090063  
指导教师： 石柯

分数	
教师签名	

2022 年 12 月 31 日

## 目 录

<b>实验一 lab1_3 外部中断 .....</b>	<b>1</b>
1.1 实验目的.....	1
1.2 实验内容.....	1
1.3 实验调试及心得.....	2
<b>实验二 lab2_3 缺页异常 .....</b>	<b>3</b>
1.1 实验目的.....	3
1.2 实验内容.....	3
1.3 实验调试及心得.....	4
<b>实验三 lab3_1 进程创建 .....</b>	<b>5</b>
1.1 实验目的.....	5
1.2 实验内容.....	5
1.3 实验调试及心得.....	5
<b>实验三 lab3_2 进程 yield .....</b>	<b>6</b>
1.1 实验目的.....	6
1.2 实验内容.....	6
1.3 实验调试及心得.....	6
<b>实验三 lab3_3 循环轮转调度 .....</b>	<b>7</b>
1.1 实验目的.....	7
1.2 实验内容.....	7
1.3 实验调试及心得.....	7

# 实验一 lab1\_3 外部中断

## 1.1 实验目的

完善 `handle_mtimer_trap()` 函数，实现 PKE 操作系统的时钟中断处理，使得程序可以正确实现外部时钟中断的功能，同时执行 `obj/app_long_loop` 时获得要求的正确输出。

## 1.2 实验内容

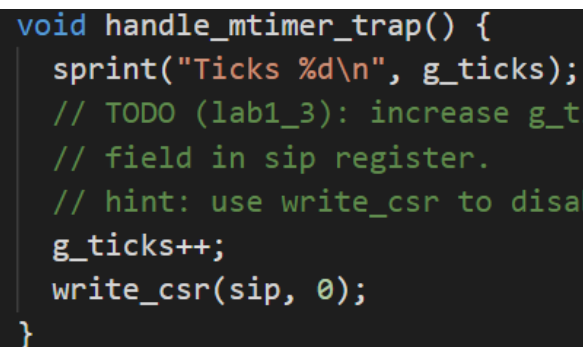
可以看到待完善的函数 `handle_mtimer_trap()` 上面一行定义了一个全局变量 `g_ticks`，此变量用于记录时钟中断的次数。根据任务描述里的打印要求，如图 1.1 所示，间隔 10000000 个周期就会进行一次时钟中断，同时打印一次 `g_ticks`。



```
Ticks 0
wait 15000000
wait 20000000
Ticks 1
wait 25000000
wait 30000000
wait 35000000
Ticks 2
```

图 1.1 lab1\_1 输出要求

根据打印的内容可知，`g_ticks` 的值每次增加 1，因此我们就要对 `g_ticks` 进行加一操作。同时要将中断清除，因此要将 SIP 的 `SIP_SSIP` 位清零，保证下次中断的正常发生，即调用 `write_csr()` 函数将 `sip` 设置为 0，如图 1.2 所示。



```
void handle_mtimer_trap() {
    sprintf("Ticks %d\n", g_ticks);
    // TODO (lab1_3): increase g_t
    // field in sip register.
    // hint: use write_csr to disa
    g_ticks++;
    write_csr(sip, 0);
}
```

图 1.2 lab1\_3 通关代码

## 1.3 实验调试及心得

一开始起步会觉得实验很难，尤其顺序看文档的话，实验文档的基础知识内容非常多，在入手的时候要看很多内容，而且看了之后也觉得似是而非不太理解，对 `elf` 文件一头雾水。但是真正接触实验之后觉得还好，因为代码注释写得很好，同时实验文档的教程也十分清晰，只要根据教程在对应的地方补充代码即可，总体比较简单。

## 实验二 lab2\_3 缺页异常

### 1.1 实验目的

完善 `handle_user_page_fault()` 函数，实现 PKE 操作系统的栈缺页异常处理，使得执行 `obj/app_sum_sequence` 时能够递归求解数字 0~1000 的和，且当 `n` 过大时用户态栈空间管理能够正确处理用户进程的“压栈”请求。

### 1.2 实验内容

由构造结果可知，用户态栈的栈底 `USER_STACK_TOP` 在 `0x7fff000`，页面大小位 4KB，因为 `n` 太大了“压爆”了用户态栈。为了解决该问题，补充 `handle_user_page_fault(uint64 mcause, uint sepc, uint stval)` 函数：

首先通过输入的最后一个参数 `stval`（根据注释该参数存放的是发生缺页异常时程序要访问的逻辑地址）来判断缺页的逻辑地址在用户进程逻辑地址空间中的位置，看这个地址是不是超过了 `USER_STACK_TOP`，且比预设的最大空间小。若满足，则为合法地址。

而若为合法地址就分配一个物理页，将所分配的物理页面映射到 `stval` 所对应的虚拟地址上。方法如前面实验所示（`sys_user_allocate_page` 中），使用 `user_vm_map()` 函数，代码如图 2.1 所示。

```
void handle_user_page_fault(uint64 mcause, uint64 sepc,
uint64 stval) {
    sprintf("handle_page_fault: %lx\n", stval);
    switch (mcause) {
        case CAUSE_STORE_PAGE_FAULT:
            // TODO (lab2_3): implement the operations that solve
            // the page fault to
            // dynamically increase application stack.
            // hint: first allocate a new physical page, and then,
            // maps the new page to the
            // virtual address that causes the page fault.
            map_pages(current->pagetable, ROUNDDOWN(stval, PGSIZE),
PGSIZE, (uint64)alloc_page(), prot_to_type(PROT_READ |
PROT_WRITE, 1));
            break;
```

图 2.1 lab2\_3 通关代码

其中 `PROT_WRITE`, `PROT_READ` 代表此页面是否可写、可读，`USER` 位为 1 表示在用户模式下可以访问该页。

## 1.3 实验调试及心得

这关一开始感觉有点懵，后来又回头看了一下前面两关的指导才有了眉目，上手敲代码之后因为脑子不清楚在很简单的地方卡了很久很久，现在回头想想自己都不理解的程度。做出来之后认为这关还是比较简单的（做不出来就是难死了，做出来就是还行/很简单 笑）。

# 实验三 lab3\_1 进程创建

## 1.1 实验目的

完善 process.c 中的 do\_fork()函数，在 PKE 内核中实现子进程到父进程代码段的映射，将子进程中对应的逻辑地址空间映射到其父进程中装载代码段的物理页面，完成子进程的创建并将子进程投入运行。

## 1.2 实验内容

观察 do\_fork()函数中的 switch 函数，前两种 case 下（即 trapframe 段和堆栈段）都是进行复制的方法将父进程的这两个段拷贝给子进程。然而对于代码段来说，为了减少系统的开销（注释也说了不要拷贝），应该通过的映射的方法将子进程中对应的逻辑地址空间映射到其父进程中装载代码段的物理页面。

由 lab2 可知，仍然使用 user\_vm\_map()函数来实现映射，如图 3.1.1 所示。

```
// DO NOT COPY THE PHYSICAL PAGES, JUST MAP THEM.
user_vm_map(
    child -> pagetable,
    parent -> mapped_info[i].va,
    parent -> mapped_info[i].npages*PGSIZE,
    lookup_pa(parent -> pagetable,
        parent -> mapped_info[i].va),
    prot_to_type(PROT_EXEC | PROT_READ, 1)
);
```

图 3.1.1 lab3\_1 通关代码

将子进程的代码段映射到父进程的代码段的物理页面，而 lookup\_pa 是为了获取父进程代码段的物理地址并将其虚拟地址存储在 mapped\_info[i].va 中；同时设置 READ、WRITE 访问权限（已在 lab2\_3 说明）。

## 1.3 实验调试及心得

这关还是比较简单的，通过一个小例子不仅了解了子进程的创建及其过程，同时也再次巩固了内存管理部分的内容，熟悉 user\_vm\_map()函数的使用。

# 实验三 lab3\_2 进程 yield

## 1.1 实验目的

完善 sys\_user\_yield()函数，实现进程执行过程中的主动释放 CPU 的动作。

## 1.2 实验内容

实验指导说明和对应代码如下：

- 将当前进程设置为就绪状态（READY）  
⇒ current -> status = READY;
- 将当前进程加入到就绪队列的队尾  
⇒ Insert\_to\_ready\_queue(current);
- 转进程调度  
⇒ schedule()

以上即为答案，整体代码如图 3.2.1 所示。

```
current->status = READY;
insert_to_ready_queue( current );
schedule();
return 0;
```

图 3.2.1 lab3\_2 通关核心代码

## 1.3 实验调试及心得

嗯，谢谢实验指导文档。但是还是学会了怎么具体实现进程释放 CPU。



# 实验三 lab3\_3 循环轮转调度

## 1.1 实验目的

完善 `rrsched()` 函数，使单个进程能够主动放弃对 CPU 的掌控，从而解决单个进程长期霸占 CPU、执行体过长的问题，实现进程的循环轮转调度。

## 1.2 实验内容

查找 `sched.h` 文件，可看到有这样一行代码，如图 3.3.1 所示。可得知时间片的长度为 2ticks，即每隔两个周期触发一次重新调度。

```
//length of a time slice, in number of ticks
#define TIME_SLICE_LEN 2
```

图 3.3.1 时间片长度定义

已知定义了一个 `tick` 计数器变量 `tick_count`，用来实现对 `tick` 的计数，从而来判断是否达到一个时间片的长度，从而判断是否要触发一次重新调度。实现的思路为：判断下一个 `tick` 是否达到时间片的长度：若达到，则进行 lab3\_2 中的 CPU 释放工作（详情见上一章），并且要对计数器变量进行清零操作；反之则直接令计数器自增并直接返回。

核心代码如图 3.3.2 所示，只是比 lab3\_2 多增加了一个计数器的功能和 if-else 判断语句，总体来说是 lab3\_2 的延伸。

```
if(current->tick_count+1 >= TIME_SLICE_LEN){
    current->tick_count = 0;
    current->status = READY;
    insert_to_ready_queue(current);
    schedule();
} else {
    current->tick_count++;
}
```

图 3.3.2 lab3\_2 通关核心代码

## 1.3 实验调试及心得

如上所说，该实验是上一个子实验的延伸，让我巩固了 CPU 释放的方法，

也在查看代码的过程中加深了对时间片的理解，最后实现了对两个进程的循环轮转调度。虽然实验比较简单，但是却让一个抽象的概念更加具象化和清晰，加深了我的理解。