OS_Lab4_Experimental report

湖南大学信息科学与工程学院

计科 210X 甘晴void （学号 202108010XXX）

实验目的

- 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

实验内容

lab2/3完成了物理和虚拟内存管理，这给创建内核线程（内核线程是一种特殊的进程）打下了提供内存管理的基础。当一个程序加载到内存中运行时，首先通过ucore OS的内存管理子系统分配合适的空间，然后就需要考虑如何分时使用CPU来“并发”执行多个程序，让每个运行的程序（这里用线程或进程表示）“感到”它们各自拥有“自己”的CPU。

本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

- 内核线程只运行在内核态
- 用户进程会在在用户态和内核态交替运行
- 所有内核线程共用ucore内核内存空间，不需为每个内核线程维护单独的内存空间
- 而用户进程需要维护各自的用户内存空间

相关原理介绍可看附录B：【原理】进程/线程的属性与特征解析。

练习**0**：填写已有实验

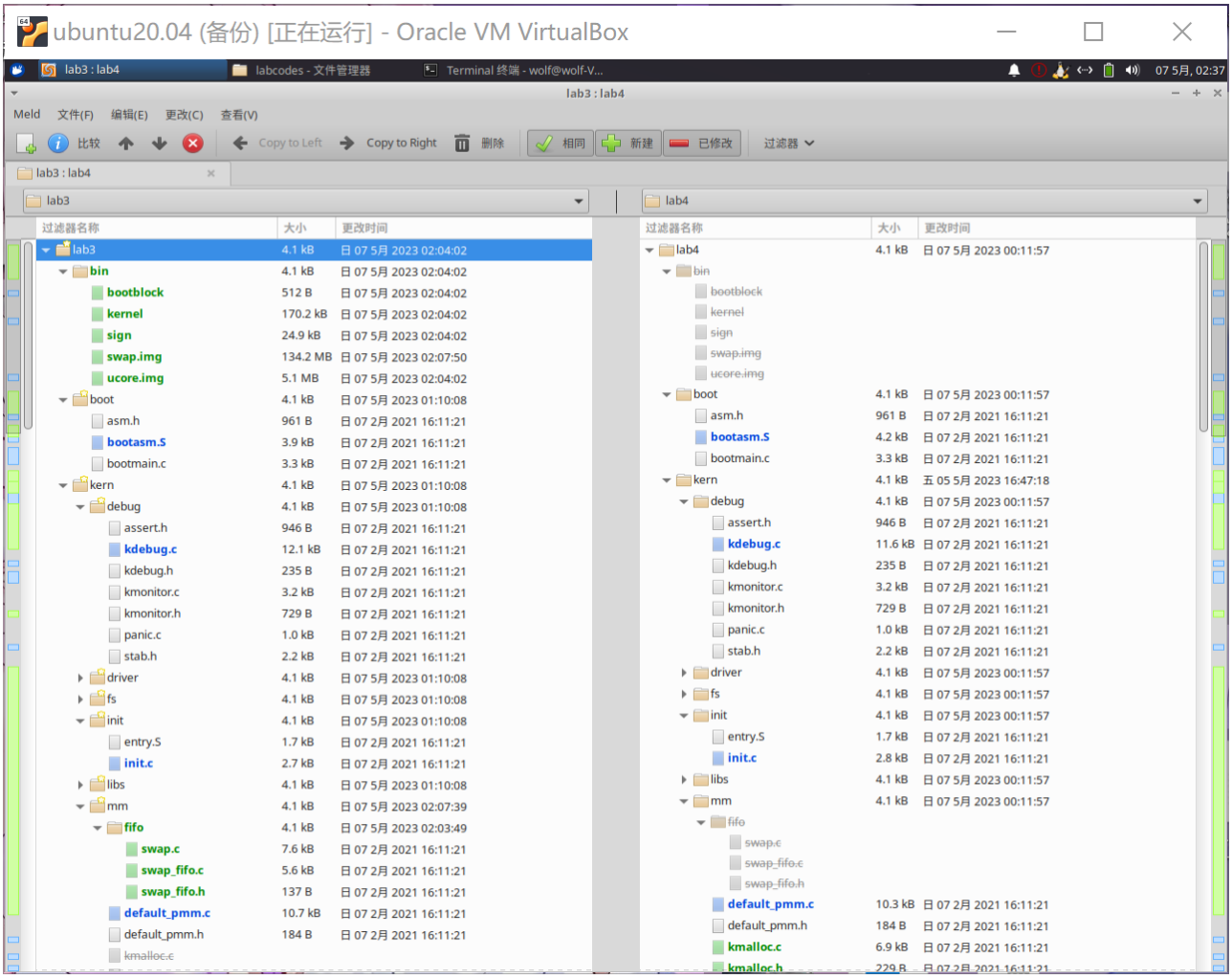
本实验依赖实验1/2/3。请把你做的实验1/2/3的代码填入本实验中代码中有“LAB1”,“LAB2”,“LAB3”的注释相应部分。

使用meld将实验1/2/3的代码中相应的部分填入实验四中的代码中：

使用meld工具可以比较方便地查看Lab4与Lab3的差异，由于Lab1、Lab2已经是被Lab3兼容了，所以不需要再做考虑。

其中，需要修改的部分为：

- default_pmm.c
- pmm.c
- swap_fifo.c
- vmm.c
- trap.c



注意慎重完成迁移工作，不要过多修改代码或少修改代码，这将导致错误。

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。

> 【提示】在*alloc_proc*函数的实现中，需要初始化的*proc_struct*结构中的成员变量至少包括：*state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name*。

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）

1.内核线程及管理

内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

- 内核线程只运行在内核态，用户进程会在在用户态和内核态交替运行；
- 所有内核线程直接使用共同的ucore内核内存空间，不需为每个内核线程维护单独的内存空间，而用户进程需要拥有各自的内存空间。

把内核线程看作轻量级的进程，对内核线程的管理和对进程的管理是一样的。对进程的管理是通过进程控制块结构实现的，将所有的进程控制块通过链表链接在一起，形成进程控制块链表，对进程的管理和调度就通过从链表中查找对应的进程控制块来完成。

2.进程控制块

保存进程信息的进程控制块结构的定义在kern/process/proc.h中定义如下：

```
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; // 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 cr3;              // 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
};
```

- mm：在Lab3中，该结构用于内存管理。在对内核线程管理时，由于内核线程不需要考虑换入换出，该结构不需要使用，因此设置为NULL。唯一需要使用的是mm中的页目录地址，保存在cr3变量中。
- state：进程状态，有以下几种
    - PROC_UNINIT：未初始化
    - PROC_SLEEPING：睡眠状态
    - PROC_RUNNABLE：可运行（可能正在运行）
    - PROC_ZOMBIE：等待回收
- parent：父进程
- context：进程上下文，用于进程切换
- tf：中断帧指针，用于中断后恢复进程状态
- cr3：页目录的物理地址，用于进程切换时快速找到页表位置
- kstack：线程所使用的内核栈
- list_link：所有进程控制块链接形成的链表的节点
- hash_link：所有进程控制块有一个根据pid建立的哈希表，hash_link是该链表的节点

为了管理系统中的所有进程控制块，ucore还维护了以下全局变量：

- static struct proc *current：当前占用CPU且处于“运行”状态进程控制块指针。通常这个变量是只读的，只有在进程切换的时候才进行修改，并且整个切换和修改过程需要保证操作的原子性，需要屏蔽中断。
- static struct proc *initproc：本实验中，指向一个内核线程。本实验以后，此指针将指向第一个用户态进程。
- static list_entry_t hash_list[HASH_LIST_SIZE]：所有进程控制块的哈希表，proc_struct中的成员变量hash_link将基于pid链接入这个哈希表中。
- list_entry_t proc_list：所有进程控制块的双向线性列表，proc_struct中的成员变量list_link将链接入这个链表中。

★3.分配并初始化一个进程控制块

内核线程创建之前，需要先创建一个进程控制块管理保存进程信息。alloc_proc函数负责分配创建一个proc_struct结构，并进行基本的初始化。此时仅是创建了进程块，内核线程本身还没有创建。这是练习一需要完成的部分，具体的实现如下：

```
//进程状态信息
enum proc_state {
    // 未初始化
    PROC_UNINIT = 0,  // uninitialized
    // 休眠、阻塞状态
```

```
    PROC_SLEEPING,    // sleeping
    // 可运行、就绪状态
    PROC_RUNNABLE,    // runnable(maybe running)
    // 僵尸状态(几乎已经终止，等待父进程回收其所占资源)
    PROC_ZOMBIE,      // almost dead, and wait parent proc to
reclaim his resource
};

// alloc_proc -负责创建并初始化一个新的proc_struct结构存储内核线程信息
static struct proc_struct *
alloc_proc(void)
{
    //为创建的线程申请空间
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
    //LAB4:EXERCISE1 YOUR CODE
    //因为没有分配物理页，故将线程状态初始为初始状态
     proc->state=PROC_UNINIT;
     proc->pid=-1; //id初始化为-1
     proc->runs=0; //运行时间为0
     proc->kstack=0;
     proc->need_resched=0; //不需要释放CPU，因为还没有分配
     proc->parent=NULL;  //当前没有父进程，初始为null
     proc->mm=NULL;      //当前未分配内存，初始为null
     //用memset非常方便将context变量中的所有成员变量置为0
     //避免了一一赋值的麻烦。。
     memset(&(proc -> context), 0, sizeof(struct context));
     proc->tf=NULL;          //当前没有中断帧,初始为null
     proc->cr3=boot_cr3;     //内核线程，cr3 等于boot_cr3
     proc->flags=0;
     memset(proc -> name, 0, PROC_NAME_LEN);
    }
    return proc;
}
```

▲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为？

（1）简单来说：

- context保存了进程的上下文信息，即各个寄存器的值，用于进程切换时恢复上下文。
- tf是中断帧的指针，指向中断帧。中断帧记录了进程被中断前的信息，除寄存器外还有中断号，错误码等信息，用于中断处理后进程状态的恢复。

发生中断时，首先从TSS中找到进程内核栈的指针切换到内核栈，然后在内核栈顶建立trapframe，进入内核态。当中断服务例程运行结束，从中断返回时，再从trapframe恢复寄存器的值，并切换回用户态。用户程序在用户态通过系统调用进入内核态，以及在内核态新创建的进程，都通过tf指向的中断帧恢复寄存器的值，从而回到用户态继续运行。

（2）详细的解题过程

根据提示我们查看相关的代码(通过查找定义tf以及context的函数)：

首先我们找到了kernel_thread函数和copy_thread函数，可知该函数对tf进行了设置，并对context的esp和eip进行了设置(具体设置过程在代码注释中给出)：

```
/*
kernel_thread函数采用了局部变量tf来放置保存内核线程的临时中断帧，并把中断帧的指针传递给do_fork函数，而do_fork函数会调用copy_thread函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间
*/
int kernel_thread(int (*fn)(void *), void *arg, uint32_t
clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));
    //kernel_cs和kernel_ds表示内核线程的代码段和数据段在内核中
    tf.tf_cs = KERNEL_CS;
    tf.tf_ds = tf.tf_es = tf.tf_ss = KERNEL_DS;
    //fn指实际的线程入口地址
    tf.tf_regs.reg_ebx = (uint32_t)fn;
    tf.tf_regs.reg_edx = (uint32_t)arg;
    //kernel_thread_entry用于完成一些初始化工作
    tf.tf_eip = (uint32_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
```

```
}
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct
trapframe *tf)
{
    //将tf进行初始化
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    //设置tf的esp，表示中断栈的信息
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;
    //对context进行设置
    //forkret主要对返回的中断处理，基本可以认为是一个中断处理并恢复
    proc->context.eip = (uintptr_t)forkret;
    proc->context.esp = (uintptr_t)(proc->tf);
}
```

通过上述函数并结合switch.S中对context的操作，将各种寄存器的值保存到context中。我们可以知道context是与上下文切换相关的，而tf则与中断的处理相关。

（3）具体回答：

context作用：

进程的上下文，用于进程切换。主要保存了前一个进程的现场（各个寄存器的状态）。在uCore中，所有的进程在内核中也是相对独立的。使用context 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用context进行上下文切换的函数是在kern/process/switch.S中定义switch_to。

tf作用：

中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，uCore内核允许嵌套中断。因此为了保证嵌套中断发生时tf 总是能够指向当前的trapframe，uCore 在内核栈上维护了 tf 的链。

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。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。

根据注释提示我们了解几个函数用途以及用法：

```
//创建一个proc并初始化所有成员变量
void alloc_proc(void)
//为一个内核线程分配物理页
static int setup_kstack(struct proc_struct *proc)
//暂时未看出其用处，可能是之后的lab会用到
static int copy_mm(uint32_t clone_flags, struct proc_struct *proc)
//复制原进程上下文到新进程
static void copy_thread(struct proc_struct *proc, uintptr_t esp,
struct trapframe *tf)
//返回一个pid
static int get_pid(void)
//将proc加入到hash_list
static void hash_proc(struct proc_struct *proc)
// 唤醒该线程，即将该线程的状态设置为可以运行
void wakeup_proc(struct proc_struct *proc);
```

下面是具体的实现过程：

根据要求可知，do_fork()函数的实现大致步骤包括七步，然后根据注释大致实现过程如下：

①调用alloc_proc()函数申请内存块，如果失败，直接返回处理。

alloc_proc（）函数在练习一中实现过，如果分配进程PCB失败，也就是说，进程一开始就是NULL，那么就会被if（proc！=NULL）判定为否，那么就不会分配初始化资源，连初始化资源都没有了，那么就会返回NULL。

②调用setup_kstack()函数为进程分配一个内核栈。

从下面此函数代码中可以看到，如果页不为空的时候，会return 0，也就是说分配内核栈成功了（这样推测的根据在于，最后一个return -E_NO_MEM，大概推测就是一个初始化的或者错误的状态，因为在这个函数最开始不需要实现的部分，这个值就赋值给了ret），那么就会返回0，否则返回一个奇怪的东西。因此，我们调用该函数分配一个内核栈空间，并判断是否分配成功。

```
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;
}
```

③调用copy_mm()函数，复制父进程的内存信息到子进程。

对于这个函数可以看到，进程proc复制还是共享当前进程current，是根据clone_flags来决定的，如果是clone_flags & CLONE_VM（为真），那么就可以拷贝。这个函数里面似乎没有做任何事情，仅仅是确定了一下current当前进程的虚拟内存是否为空，那么具体的操作，只需要传入它所需要的clone_flag就可以，其余事情不需要我们去做。

```
static int
copy_mm(uint32_t clone_flags, struct proc_struct *proc) {
    assert(current->mm == NULL);
    /* do nothing in this project */
    return 0;
}
```

④调用copy_thread()函数复制父进程的中断帧和上下文信息。

copy_thread()函数需要传入的三个参数，第一个是比较熟悉，练习一中已经实现的PCB模块proc结构体的对象，第二个参数，是一个栈，判断的依据是它的数据类型，在练习一中的PCB模块中，为栈定义的数据类型就是uintptr_t，第三个参数也很熟悉，它是练习一PCB中的中断帧的指针。

```
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct
trapframe *tf) {
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;

    proc->context.eip = (uintptr_t)forkret;
    proc->context.esp = (uintptr_t)(proc->tf);
}
```

⑤将新进程添加到进程的（hash）列表中。

调用hash_proc这个函数可以将当前的新进程添加到进程的哈希列表中，分析hash函数的特点，直接调用hash（proc）即可。

```
hash_proc(struct proc_struct *proc) {
    list_add(hash_list + pid_hashfn(proc->pid), &(proc-
>hash_link));
}
```

⑥唤醒新进程。

wakeup_proc(proc);

⑦返回新进程pid。

ret = proc->pid;

do_fork函数的实现：

```
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;
    //1: 调用alloc_proc()函数申请内存块，如果失败，直接返回处理
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    //2.将子进程的父节点设置为当前进程
    proc->parent = current;
    //3.调用setup_stack()函数为进程分配一个内核栈
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    //4.调用copy_mm()函数复制父进程的内存信息到子进程
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    //5.调用copy_thread()函数复制父进程的中断帧和上下文信息
    copy_thread(proc, stack, tf);
    //6.将新进程添加到进程的hash列表中
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc); //建立映射
        nr_process ++;  //进程数加1
        list_add(&proc_list, &(proc->list_link));//将进程加入到进程的链
表中
    }
    local_intr_restore(intr_flag);
    //      7.一切就绪，唤醒子进程
    wakeup_proc(proc);
    //      8.返回子进程的pid
    ret = proc->pid;
```

```
fork_out:
    return ret;

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

▲问题：ucore是否做到给每个新fork的线程一个唯一的id？

uCore中，每个新fork的线程都存在唯一的一个ID，理由如下：

在函数get_pid中，如果静态成员last_pid小于next_safe，则当前分配的last_pid一定是安全的，即唯一的PID。
但如果last_pid大于等于next_safe，或者last_pid的值超过MAX_PID，则当前的last_pid就不一定是唯一的PID，此时就需要遍历proc_list，重新对last_pid和next_safe进行设置，为下一次的get_pid调用打下基础。

ucore通过调用get_pid函数分配pid，我们可以对get_pid函数进行分析。

```
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);//分配pid前，get_pid会先确认可用进程号大于最大进程数。
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    //该函数定义了两个静态全局变量，next_safe最初被设置为最大进程号，last_pid最初设置为1，
    //[last_pid，next_safe]就是合法的pid区间。
    if (++ last_pid >= MAX_PID) {
        //如果last_pid++在这个区间内，就可以直接返回last_pid作为新分配的进程号。
        last_pid = 1;
        goto inside;
    }
    //区间合法性判断
    if (last_pid >= next_safe) {
        //如果last_pid>=next_safe，就将next_safe设置为MAX_PID，遍历链表确保last_pid和已有进程的pid不相同，并更新next_safe。
```

```
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        //遍历进程链表
        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;              //区间不合法，重新遍历链表
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid)
{
                next_safe = proc->pid;       //更新next_safe
            }
        }
    }
    return last_pid;
}
```

维护last_pid到next_safe这个区间将可用的pid范围缩小，以提高了分配的效率，如果区间不合法，也会重新更新区间，并排除和已有进程进程号相同的情况，因此最终产生的进程的pid是唯一的。但是需要注意的是进程链表是全局变量，如果有另一个进程get_pid后还没有把进程加入链表，调度到了当前进程，而当前进程又需要遍历链表排除进程号相同的情况，就可能产生错误，因此要在get_pid和将进程加入链表的位置添加互斥。保证互斥的方法为在do_fork中分配进程号和进程加入进程链表的部分关中断，避免进程调度。

练习3：阅读代码，理解 **proc_run** 函数和它调用的函数如何完成进程切换的。（无编码工作）

请在实验报告中简要说明你对proc_run函数的分析。并回答如下问题：

- 在本实验的执行过程中，创建且运行了几个内核线程？
- 语句`local_intr_save(intr_flag);....local_intr_restore(intr_flag);`在这里有何作用?请说明理由

根据实验指导书，uCore中，内核的第一个进程idleproc会执行cpu_idle函数，并从中调用schedule函数，准备开始调度进程，完成进程调度和进程切换。

```
void cpu_idle(void) {
    while (1)
        if (current->need_resched)
            schedule();
}
```

对schedule函数代码进行分析：

```
/* 宏定义:
   #define le2proc(le, member)         \
    to_struct((le), struct proc_struct, member)*/
void
schedule(void) {
    bool intr_flag; //定义中断变量
    list_entry_t *le, *last; //当前list，下一list
    struct proc_struct *next = NULL; //下一进程
    local_intr_save(intr_flag); //中断禁止函数
    {
        current->need_resched = 0; //设置当前进程不需要调度
      //last是否是idle进程(第一个创建的进程),如果是，则从表头开始搜索
      //否则获取下一链表
        last = (current == idleproc) ? &proc_list : &(current-
>list_link);
        le = last;
        do { //一直循环，直到找到可以调度的进程
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);//获取下一进程
                if (next->state == PROC_RUNNABLE) {
                    break; //找到一个可以调度的进程，break
                }
            }
        } while (le != last); //循环查找整个链表
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc; //未找到可以调度的进程
        }
```

```
        next->runs ++; //运行次数加一
        if (next != current) {
            proc_run(next); //运行新进程,调用proc_run函数
        }
    }
    local_intr_restore(intr_flag); //允许中断
}
```

可以看到ucore实现的是FIFO调度算法：

1 调度开始时，先屏蔽中断。

2 在进程链表中，查找第一个可以被调度的程序

3 运行新进程，允许中断

★注意到这里的调度也有不允许中断的操作

chedule函数会先清除调度标志，并从当前进程在链表中的位置开始，遍历进程控制块，直到找出处于就绪状态的进程。

之后执行proc_run函数，将环境切换至该进程的上下文并继续执行。

提到上下文切换，就需要使用switch_to函数：

```
switch_to:                      # switch_to(from, to)
    # save from's registers
    movl 4(%esp), %eax          #保存from的首地址
    popl 0(%eax)                #将返回值保存到context的eip
    movl %esp, 4(%eax)          #保存esp的值到context的esp
    movl %ebx, 8(%eax)          #保存ebx的值到context的ebx
    movl %ecx, 12(%eax)         #保存ecx的值到context的ecx
    movl %edx, 16(%eax)         #保存edx的值到context的edx
    movl %esi, 20(%eax)         #保存esi的值到context的esi
    movl %edi, 24(%eax)         #保存edi的值到context的edi
    movl %ebp, 28(%eax)         #保存ebp的值到context的ebp

    # restore to's registers
    movl 4(%esp), %eax          #保存to的首地址到eax
    movl 28(%eax), %ebp         #保存context的ebp到ebp寄存器
    movl 24(%eax), %edi         #保存context的ebp到ebp寄存器
```

```
    movl 20(%eax), %esi         #保存context的esi到esi寄存器
    movl 16(%eax), %edx         #保存context的edx到edx寄存器
    movl 12(%eax), %ecx         #保存context的ecx到ecx寄存器
    movl 8(%eax), %ebx          #保存context的ebx到ebx寄存器
    movl 4(%eax), %esp          #保存context的esp到esp寄存器
    pushl 0(%eax)               #将context的eip压入栈中
    ret
```

所以switch_to函数主要完成的是进程的上下文切换，先保存当前寄存器的值，然后再将下一进程的上下文信息保存到对于寄存器中。

```
//proc_run函数
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;//定义中断变量
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag); //屏蔽中断
        {
            current = proc;//修改当前进程为新进程
            load_esp0(next->kstack + KSTACKSIZE);//修改esp
            lcr3(next->cr3);//修改页表项,完成进程间的页表切换
            switch_to(&(prev->context), &(next->context));//上下文切
换
        }
        local_intr_restore(intr_flag); //允许中断
    }
}
```

实现思路：

- 让 current 指向 next 内核线程 initproc；
- 设置任务状态段 ts 中特权态 0 下的栈顶指针 esp0 为 next 内核线程 initproc 的内核栈的栈顶，即 next->kstack + KSTACKSIZE ；
- 设置 CR3 寄存器的值为 next 内核线程 initproc 的页目录表起始地址 next->cr3，这实际上是完成进程间的页表切换；
- 由 switch_to函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当 switch_to 函数执行完“ret”指令后，就切换到 initproc 执行了

▲问题一：在本实验的执行过程中，创建且运行了几个内核线程？

两个，分别是idleproc和initproc。

- idleproc：第一个内核进程，完成内核中各个子系统的初始化，之后立即调度，执行其他进程。
- initproc：用于完成实验的功能而调度的内核进程。

idlepro是0号内核线程。kern_init调用了proc_init，在proc_init中会创建该线程。该线程的need_resched设置为1，运行cpu_idle函数，总是要求调度器切换到其他线程。

```
//proc_init中创建idle_proc
    if ((idleproc = alloc_proc()) == NULL) {
        panic("cannot alloc idleproc.\n");
    }
    //线程初始化
    idleproc->pid = 0;                                  //0号线程
    idleproc->state = PROC_RUNNABLE;                    //设置为可运行
    idleproc->kstack = (uintptr_t)bootstack;            //启动后的内核栈被
设置为该线程的内核栈
    idleproc->need_resched = 1;
    set_proc_name(idleproc, "idle");
    nr_process ++;
    current = idleproc;
//kern_init最后会运行该内核线程，调度到其他线程
void cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
    }
}
```

initproc是第1号线程，未来所有的进程都是由该线程fork产生的。init_proc也是在proc_init中创建的，通过调用kernel_thread创建，该线程运行init_main并输出字符串。

```
//init_proc的创建
    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");
```

kernel_thread中定义了一个trapframe结构，然后将该结构传入do_fork完成线程的建立。

```
int kernel_thread(int (*fn)(void *), void *arg, uint32_t
clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));
    tf.tf_cs = KERNEL_CS;
    tf.tf_ds = tf.tf_es = tf.tf_ss = KERNEL_DS;          //使用内核的
代码和数据段
    tf.tf_regs.reg_ebx = (uint32_t)fn;                   //函数地址
    tf.tf_regs.reg_edx = (uint32_t)arg;                  //参数
    tf.tf_eip = (uint32_t)kernel_thread_entry;
//kernel_thread_entry中将进入ebx指定的函数执行
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

该线程创建完成后，proc_init也完成了工作，返回到kern_init，kern_init会运行idle_proc的cpu_idle，进行进程调度，从而切换运行init_proc。切换线程是调度器schedule函数完成的，该函数会在进程链表中寻找一个就绪的进程，调用proc_run切换到改进程。proc_run会进行上下文切换，而在do_fork中调用的copy_thread函数中，将context.eip设置为了forkret，进程切换完成后从forkret开始运行。forkret实际上是forkrets，forkrets会从当前进程的trapframe恢复上下文，然后跳转到设置好的kernel_thread_entry。

▲问题二：语句`local_intr_save(intr_flag);....local_intr_restore(intr_flag);`在这里有何作用？请说明理由。

原子化的操作，用于哈希表写入的过程中。

作用分别是屏蔽中断和打开中断，以免进程切换时其他进程再进行调度。也就是保护进程切换不会被中断，以免进程切换时其他进程再进行调度，相当于互斥锁。之前在第六步添加进程到列表的时候也需要有这个操作，是因为进程进入列表的时候，可能会发生一系列的调度事件，比如我们所熟知的抢断等，加上这么一个保护机制可以确保进程执行不被打乱。

代码准确性核验

完成代码编写后，编译并运行代码：make qemu

如果可以得到如 附录A所示的显示内容（仅供参考，不是标准答案输出），则基本正确。

输出结果如下，发现与参考答案大致相同。

```
wolf@wolf-VB:~/桌面/wolf/os_kernel_lab-master/labcodes/lab4$ make
qemu
+ cc kern/init/entry.S
+ cc kern/init/init.c
kern/init/init.c: In function ‘kern_init’:
kern/init/init.c:32:5: warning: implicit declaration of function
‘grade_backtrace’ [-Wimplicit-function-declaration]
   32 |     grade_backtrace();
      |     ^~~~~~~~~~~~~~~~
kern/init/init.c: In function ‘grade_backtrace2’:
kern/init/init.c:57:5: warning: implicit declaration of function
‘mon_backtrace’; did you mean ‘grade_backtrace2’? [-Wimplicit-
function-declaration]
   57 |     mon_backtrace(0, NULL, NULL);
      |     ^~~~~~~~~~~~
      |     grade_backtrace2
kern/init/init.c: At top level:
kern/init/init.c:71:1: warning: conflicting types for
‘grade_backtrace’
   71 | grade_backtrace(void) {
      | ^~~~~~~~~~~~~~~~
```

```
kern/init/init.c:32:5: note: previous implicit declaration of
‘grade_backtrace’ was here
   32 |     grade_backtrace();
      |     ^~~~~~~~~~~~~~~
kern/init/init.c:104:1: warning: ‘lab1_switch_test’ defined but not
used [-Wunused-function]
  104 | lab1_switch_test(void) {
      | ^~~~~~~~~~~~~~~~
+ cc kern/libs/stdio.c
+ cc kern/libs/rb_tree.c
+ cc kern/libs/readline.c
+ cc kern/debug/panic.c
kern/debug/panic.c: In function ‘__panic’:
kern/debug/panic.c:27:5: warning: implicit declaration of function
‘print_stackframe’; did you mean ‘print_trapframe’? [-Wimplicit-
function-declaration]
   27 |     print_stackframe();
      |     ^~~~~~~~~~~~~~~
      |     print_trapframe
+ cc kern/debug/kdebug.c
+ cc kern/debug/kmonitor.c
+ cc kern/driver/ide.c
+ cc kern/driver/clock.c
+ cc kern/driver/console.c
+ cc kern/driver/picirq.c
+ cc kern/driver/intr.c
+ cc kern/trap/trap.c
kern/trap/trap.c: In function ‘print_trapframe’:
kern/trap/trap.c:108:16: warning: taking address of packed member
of ‘struct trapframe’ may result in an unaligned pointer value [-
Waddress-of-packed-member]
  108 |     print_regs(&tf->tf_regs);
      |                ^~~~~~~~~~~
+ cc kern/trap/vectors.S
+ cc kern/trap/trapentry.S
+ cc kern/mm/pmm.c
kern/mm/pmm.c:279:1: warning: ‘boot_alloc_page’ defined but not
used [-Wunused-function]
  279 | boot_alloc_page(void) {
      | ^~~~~~~~~~~~~~~
+ cc kern/mm/swap_fifo.c
+ cc kern/mm/vmm.c
```

```
kern/mm/vmm.c: In function ‘check_vmm’:
kern/mm/vmm.c:165:12: warning: unused variable
‘nr_free_pages_store’ [-Wunused-variable]
  165 |     size_t nr_free_pages_store = nr_free_pages();
      |            ^~~~~~~~~~~~~~~~~~
kern/mm/vmm.c: In function ‘check_vma_struct’:
kern/mm/vmm.c:177:12: warning: unused variable
‘nr_free_pages_store’ [-Wunused-variable]
  177 |     size_t nr_free_pages_store = nr_free_pages();
      |            ^~~~~~~~~~~~~~~~~~
+ cc kern/mm/kmalloc.c
kern/mm/kmalloc.c: In function ‘__slob_free_pages’:
kern/mm/kmalloc.c:93:23: warning: passing argument 1 of ‘kva2page’
makes pointer from integer without a cast [-Wint-conversion]
   93 |   free_pages(kva2page(kva), 1 << order);
      |                       ^~~
      |                       |
      |                       long unsigned int
In file included from kern/mm/kmalloc.c:7:
kern/mm/pmm.h:106:16: note: expected ‘void *’ but argument is of
type ‘long unsigned int’
  106 | kva2page(void *kva) {
      |          ~~~~~~^~~
+ cc kern/mm/swap.c
+ cc kern/mm/default_pmm.c
+ cc kern/fs/swapfs.c
+ cc kern/process/entry.S
+ cc kern/process/switch.S
+ cc kern/process/proc.c
+ cc kern/schedule/sched.c
+ cc libs/string.c
+ cc libs/printfmt.c
+ cc libs/hash.c
+ cc libs/rand.c
+ ld bin/kernel
+ cc boot/bootasm.S
+ cc boot/bootmain.c
+ cc tools/sign.c
+ ld bin/bootblock
'obj/bootblock.out' size: 446 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
记录了10000+0 的读入
```

```
记录了10000+0 的写出
5120000字节（5.1 MB，4.9 MiB）已复制，0.0149466 s，343 MB/s
记录了1+0 的读入
记录了1+0 的写出
512字节已复制，7.141e-05 s，7.2 MB/s
记录了409+1 的读入
记录了409+1 的写出
209440字节（209 kB，205 KiB）已复制，0.000446185 s，469 MB/s
记录了128+0 的读入
记录了128+0 的写出
134217728字节（134 MB，128 MiB）已复制，0.125741 s，1.1 GB/s
WARNING: Image format was not specified for 'bin/ucore.img' and
probing guessed raw.
         Automatically detecting the format is dangerous for raw
images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the
restrictions.
WARNING: Image format was not specified for 'bin/swap.img' and
probing guessed raw.
         Automatically detecting the format is dangerous for raw
images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the
restrictions.
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc0100036 (phys)
  etext  0xc010ab8d (phys)
  edata  0xc012e000 (phys)
  end    0xc0131158 (phys)
Kernel executable memory footprint: 197KB
ebp:0xc012af48 eip:0xc0101e9d args:0x00010094 0x00010094 0xc012af78
0xc01000d0
    kern/debug/kdebug.c:308: print_stackframe+25
ebp:0xc012af58 eip:0xc01021b1 args:0x00000000 0x00000000 0x00000000
0xc012afc8
    kern/debug/kmonitor.c:129: mon_backtrace+14
ebp:0xc012af78 eip:0xc01000d0 args:0x00000000 0xc012afa0 0xffff0000
0xc012afa4
    kern/init/init.c:57: grade_backtrace2+23
ebp:0xc012af98 eip:0xc01000f6 args:0x00000000 0xffff0000 0xc012afc4
0x0000002a
```

```
    kern/init/init.c:62: grade_backtrace1+31
ebp:0xc012afb8 eip:0xc0100117 args:0x00000000 0xc0100036 0xffff0000
0xc0100079
    kern/init/init.c:67: grade_backtrace0+23
ebp:0xc012afd8 eip:0xc010013c args:0x00000000 0x00000000 0x00000000
0xc010aba0
    kern/init/init.c:72: grade_backtrace+30
ebp:0xc012aff8 eip:0xc0100086 args:0xc010aff8 0xc010b000 0xc010212e
0xc010b01f
    kern/init/init.c:32: kern_init+79
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07ee0000, [00100000, 07fdffff], type = 1.
  memory: 00020000, [07fe0000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
use SLOB allocator
check_slab() success
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 1, total 31916
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
```

```
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/R [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
count is 0, total is 5
check_swap() succeeded!
++ 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:354:
    process exit!!.

stack trackback:
ebp:0xc0333fa8 eip:0xc0101e9d args:0xc010962a 0xc0131044 0xc03310c0
0xc0333fdc
    kern/debug/kdebug.c:308: print_stackframe+25
ebp:0xc0333fc8 eip:0xc010185e args:0xc010cdf5 0x00000162 0xc010ce09
0xc0131044
    kern/debug/panic.c:27: __panic+111
ebp:0xc0333fe8 eip:0xc0109bfd args:0x00000000 0xc010ce88 0x00000000
0x00000010
    kern/process/proc.c:354: do_exit+32
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

使用make grade查看分数

```
wolf@wolf-VB:~/桌面/wolf/os_kernel_lab-master/labcodes/lab4$ make grade
Check VMM:                (1.4s)
  -check pmm:                                 OK
  -check page table:                          OK
  -check slab:                                WRONG
   !! error: missing 'check_slab() succeeded!'

  -check vmm:                                 OK
  -check swap page fault:                     OK
  -check ticks:                               OK
  -check initproc:                            OK
Total Score: 90/100
make: *** [Makefile:255: grade] 错误 1
```

可以看到几乎正确，这里的slab项未拿全分数是因为这是下一项扩展练习需要用到的。

扩展练习**Challenge**：实现支持任意大小的内存分配算法

这不是本实验的内容，其实是上一次实验内存的扩展，但考虑到现在的slab算法比较复杂，有必要实现一个比较简单的任意大小内存分配算法。可参考本实验中的slab如何调用基于页的内存分配算法（注意，不是要你关注slab的具体实现）来实现first-fit/best-fit/worst-fit/buddy等支持任意大小的内存分配算法。

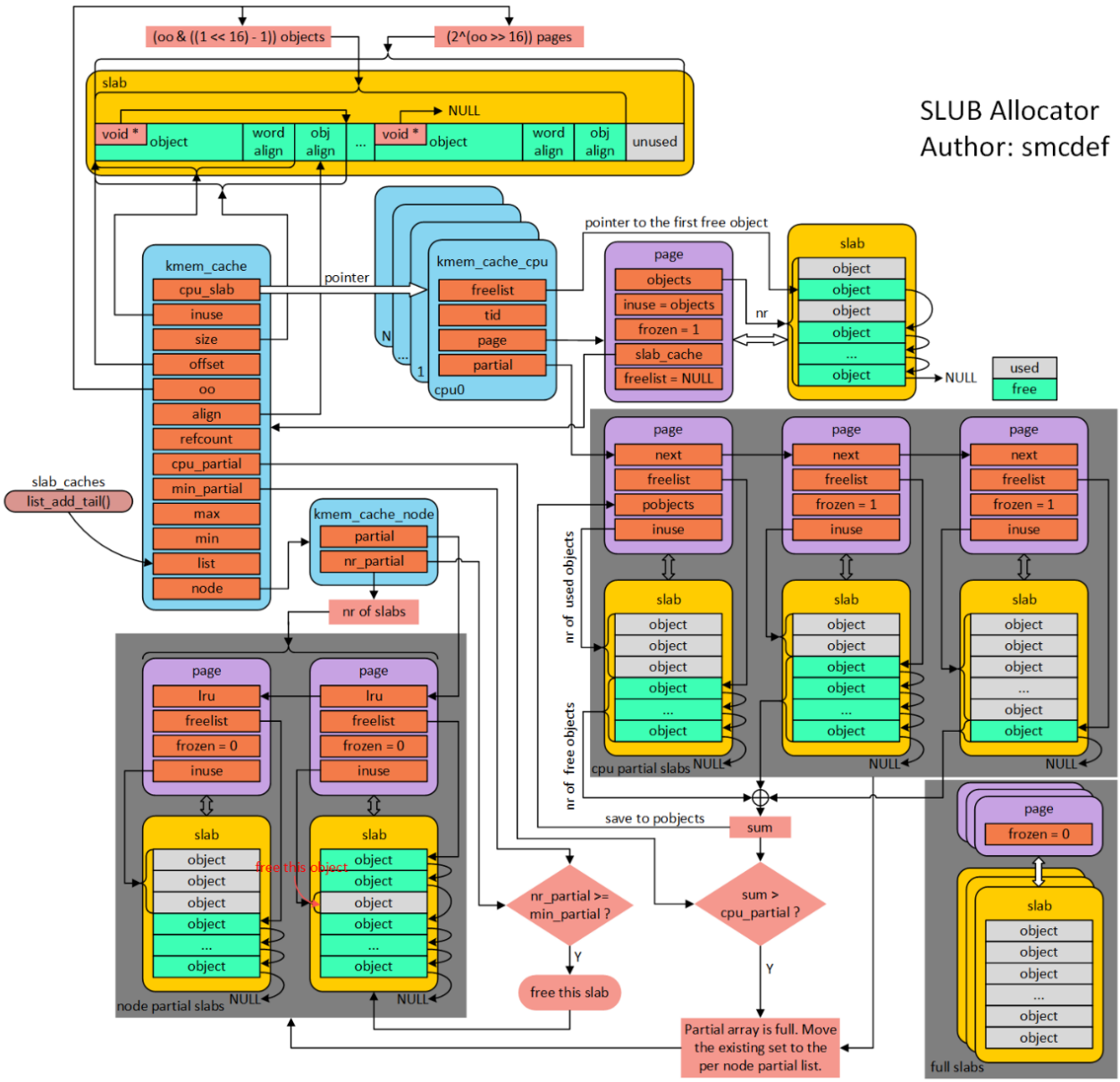
1.对比first-bit/best-fit/worst-fit/slab以及buddy这几种算法的特点

- 首次适应算法（First Fit）：该算法从空闲分区链首开始查找，直至找到一个能满足其大小要求的空闲分区为止。然后再按 照作业的大小，从该分区中划出一块内存分配给请求者，余下的空闲分区仍留在空闲分区链 中。
    - 特点： 该算法倾向于使用内存中低地址部分的空闲区，在高地址部分的空闲区很少被利用，从而保留了高地址部分的大空闲 区。显然为以后到达的大作业分配大的内存空间创造了条件。
    - 缺点：低地址部分不断被划分，留下许多难以利用、很小的空闲区，而每次查找又都从低地址部分开始，会增加查找的开销。
- 循环首次适应算法（Next Fit）：该算法是由首次适应算法演变而成的。在为进程分配内存空间时，不再每次从链首开始查 找，直至找到一个能满足要求的空闲分区，并从中划出一块来分给作业。
    - 特点：使内存中的空闲分区分布的更为均匀，减少了查找时的系统开销。
    - 缺点：缺乏大的空闲分区，从而导致不能装入大型作业。
- 最佳适应算法（Best Fit）：该算法总是把既能满足要求，又是最小的空闲分区分配给作业。为了加速查找，该算法要求将 所有的空闲区按其大小排序后，以递增顺序形成一个空白链。这样每次找到的第一个满足要求的空闲区，必然是最优的。孤立 地看，该算法似乎是最优的，但事实上并不一定。因为每次分配后剩余的空间一定是最小的，在存储器中将留下许多难以利用 的小空闲区。同时每次分配后必须重新排序，这也带来了一定的开销。
    - 特点：每次分配给文件的都是最合适该文件大小的分区。
    - 缺点：内存中留下许多难以利用的小的空闲区。
- 最坏适应算法(Worst Fit)：该算法按大小递减的顺序形成空闲区链，分配时直接从空闲区链的第一个空闲区中分配（不能 满足需要则不分配）。很显然，如果第一个空闲分区不能满足，那么再没有空闲分区能满足需要。这种分配方法初看起来不太 合理，但它也有很强的直观吸引力：在大空闲区中放入程序后，剩下的空闲区常常也很大，于是还能装下一个较大的新程序。
    - 最坏适应算法与最佳适应算法的排序正好相反，它的队列指针总是指向最大的空闲区，在进行分配时，总是从最大的空闲 区开始查寻。该算法克服了最佳适应算法留下的许多小的碎片的不足，但保留大的空闲区的可能性减小了，而且空闲区回收也和最佳适 应算法一样复杂。
    - 特点：给文件分配分区后剩下的空闲区不至于太小，产生碎片的几率最小，对中小型文件分配分区操作有利。
    - 缺点：使存储器中缺乏大的空闲区，对大型文件的分区分配不利。
- 伙伴算法
    - 伙伴算法会浪费大量的内存,(如果需要大小为9的内存块必须分配大小为16的内存块).而优点也是明显的,分配和合并算法都很简单易行.但是,当分配和回收较快的时候,例如分配大小为9的内存块,此时分配16,然后又回收,即合并伙伴内存块,这样会造成不必要的cpu浪费,应该设置链表中内存块的低潮个数,即当链表中内存块个数小于某个值的时候,并不合并伙伴内存块,只要当高于低潮个数的时候才合并。

- slab算法
    - 采用buddy算法，解决了外碎片问题，这种方法适合大块内存请求，不适合小内存区请求，与传统的内存管理模式相比， slab缓存分配器提供了很多优点。首先，内核通常依赖于对小对象的分配，它们会在系统生命周期内进行无数次分配。slab缓存分配器通过对类似大小的对象进行缓存而提供这种功能，从而避免了常见的碎片问题。slab分配器还支持通用对象的初始化，从而避免了为同一目而对一个对象重复进行初始化。最后，slab分配器还可以支持硬件缓存对齐和着色，这允许不同缓存中的对象占用相同的缓存行，从而提高缓存的利用率并获得更好的性能。

2.Slub算法详解

什么是slab缓存池呢？我的解释是使用struct kmem_cache结构描述的一段内存就称作一个slab缓存池。一个slab缓存池就像是一箱牛奶，一箱牛奶中有很多瓶牛奶，每瓶牛奶就是一个object。分配内存的时候，就相当于从牛奶箱中拿一瓶。总有拿完的一天。当箱子空的时候，你就需要去超市再买一箱回来。超市就相当于partial链表，超市存储着很多箱牛奶。如果超市也卖完了，自然就要从厂家进货，然后出售给你。厂家就相当于伙伴系统。



（1） per cpu freelist

针对每一个cpu都会分配一个struct kmem_cache_cpu的结构体。可以称作是本地缓存池。当内存申请的时候，优先从本地cpu缓存池申请。在分配初期，本地缓存池为空，自然要从伙伴系统分配一定页数的内存。内核会为每一个物理页帧创建一个struct page的结构体。kmem_cacche_cpu中page就会指向正在使用的slab的页帧。freelist成员指向第一个可用内存obj首地址。处于正在使用的slab的struct page结构体中的freelist会置成NULL，因为没有其他地方使用。struct page结构体中inuse代表已经使用的obj数量。这地方有个很有意思的地方，在刚从伙伴系统分配的slab的inuse在分配初期就置成obj的总数，在分配obj的时候并不会改变。你是不是觉得很奇怪，既然表示已经使用obj的数量，为什么一直是obj的总数呢？你想想，slab中的对象总有分配完的时候，那个时候就直接脱离kmem_cache_cpu了。此时的inuse不就名副其实了嘛！对于full slab就像图的右下角，就像无人看管的孩子，没有任何链表来管理。

（2） per cpu partial

当图中右下角full slab释放obj的时候，首先就会将slab挂入per cpu partial链表管理。通过struct page中next成员形成单链表。per cpu partial链表指向的第一个page中会存放一些特殊的数据。例如：pobjects存储着per cpu partial链表中所有slab可供分配obj的总数，如图所示。当然还有一个图中没有体现的pages成员存储per cpu partial链表中所有slab缓存池的个数。pobjects到底有什么用呢？我们从full slab中释放一个obj就添加到per cpu partial链表，总不能无限制的添加吧！因此，每次添加的时候都会判断当前的pobjects是否大于kmem_cache的cpu_partial成员，如果大于，那么就会将此时per cpu partial链表中所有的slab移送到kmem_cache_node的partial链表，然后再将刚刚释放obj的slab插入到per cpu partial链表。如果不大于，则更新pobjects和pages成员，并将slab插入到per cpu partial链表。

（3） per node partial

per node partia链表类似per cpu partial，区别是**node**中的**slab**是所有**cpu**共享的，而**per cpu**是每个**cpu**独占的。假如现在的slab布局如上图所示。假如现在如红色箭头指向的obj将会释放，那么就是一个empty slab，此时判断kmem_cache_node的nr_partial是否大于kmem_cache的min_partial，如果大于则会释放该slab的内存。

3.实现Slub算法

通过少量的修改，即可使用实验2扩展练习实现的 Slab 算法。

- 初始化 Slub 算法：在初始化物理内存最后初始化 Slub ；

```
void pmm_init(void) {
    ...
    kmem_int();
}
```

- 在 vmm.c 中使用 Slub 算法：

为了使用Slub算法，需要声明仓库的指针。

```
struct kmem_cache_t *vma_cache = NULL;
struct kmem_cache_t *mm_cache = NULL;
```

在虚拟内存初始化时创建仓库。

```
void vmm_init(void) {
    mm_cache = kmem_cache_create("mm", sizeof(struct mm_struct),
NULL, NULL);
    vma_cache = kmem_cache_create("vma", sizeof(struct vma_struct),
NULL, NULL);
    ...
}
```

在 mm_create 和 vma_create 中使用 Slub 算法。

```
struct mm_struct *mm_create(void) {
    struct mm_struct *mm = kmem_cache_alloc(mm_cache);
    ...
}

struct vma_struct *vma_create(uintptr_t vm_start, uintptr_t vm_end,
uint32_t vm_flags) {
    struct vma_struct *vma = kmem_cache_alloc(vma_cache);
    ...
}
```

在 mm_destroy 中释放内存。

```
void
mm_destroy(struct mm_struct *mm) {
    ...
    while ((le = list_next(list)) != list) {
        ...
        kmem_cache_free(mm_cache, le2vma(le, list_link));  //kfree
vma
    }
    kmem_cache_free(mm_cache, mm); //kfree mm
    ...
}
```

- 在 proc.c 中使用 Slub 算法：

声明仓库指针。

```
struct kmem_cache_t *proc_cache = NULL;
```

在初始化函数中创建仓库。

```
void proc_init(void) {
    ...
    proc_cache = kmem_cache_create("proc", sizeof(struct
proc_struct), NULL, NULL);
    ...
}
```

在 alloc_proc 中使用 Slub 算法。

```
static struct proc_struct *alloc_proc(void) {
    struct proc_struct *proc = kmem_cache_alloc(proc_cache);
    ...
}
```

本实验没有涉及进程结束后 PCB 回收，不须要回收内存。

实验总结

本实验主要是内核线程创建与切换的具体实现。在ucore中，首先创建idle_proc这个第0号内核线程，然后调用kernel_thread建立init_proc第1号内核线程，最后回到kern_init执行idle_proc线程，idle_proc总是调度到其他线程。线程具体的创建是由do_fork完成的，do_fork调用alloc_proc等函数，完成进程控制块的创建，内核栈和pid的分配，父进程上下文和中断帧的复制，还会进行一些设置，如将上下文的eip设置为fork_ret，在trapframe中将返回值设置为0等。创建完毕后返回pid，当调度器调度该线程时，调度器调用proc_run完成上下文切换后就会执行fork_ret，恢复中断帧，从而开始执行指定的程序。

重要知识点

- 内核线程和用户进程的区别
- 进程控制块
- 内核线程的创建
- 内核线程资源分配

- 进程(线程)切换的过程
- 进程控制块
- 进程状态
- 进程挂起
- 用户线程与内核线程
- 线程与进程的比较

参考文献

lab4实验的进行以及知识点的理解参考了

- https://blog.csdn.net/Aaron503/article/details/130453791?spm=1001.2014.3001.5501
- https://blog.csdn.net/sfadjlha/article/details/124859514?spm=1001.2014.3001.5502

对比first-bit/best-fit/worst-fit/slab以及buddy这几种算法的特点参考了

- https://blog.csdn.net/weixin_42637204/article/details/90968731

实现slub算法部分【强烈推荐】

- http://www.noobyard.com/article/p-edwelvki-cr.html