练习1 分配并初始化一个进程控制块（需要编码）

补全alloc_proc函数如下

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1 YOUR CODE
    /*
     * below fields in proc_struct need to be initialized
     *       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 cr3;                              // CR3 register: the
base addr of Page Directroy Table(PDT)
     *       uint32_t flags;                             // Process flag
     *       char name[PROC_NAME_LEN + 1];               // Process name
     */
        proc->pid = -1;
        proc->state = PROC_UNINIT;
        proc->kstack = 0;
        proc->runs = 0;
        proc->parent = NULL;
        proc->need_resched = 0;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->cr3 = boot_cr3;
        proc->tf = NULL;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
    }
    return proc;
}
```

函数设计说明如下：

由于进程尚未初始化 所以设置进程参数如下

- 进程ID设置为-1

- 进程状态设置为进程未初始化
- 内核栈尚未分配
- 进程执行时间为0
- 进程父进程为空
- 进程不需要被重新调度
- 进程的内存管理域为空
- 即将进程的上下文进行清空
- 设置进程的页表基地址
- 进程的trapframe设置为空
- 进程标志设置为0
- 清空进程名

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）

struct context context 是用来存储进程当前状态 用于进程切换中上下文的保存与恢复

struct trapframe *tf 是内核态线程返回用户态加载的上下文

练习2 为新创建的内核线程分配资源（需要编码）

添加do_fork函数如下：

```
/* do_fork -     parent process for a new child process
 * @clone_flags: used to guide how to clone the child process
 * @stack:       the parent's user stack pointer. if stack==0, It means to fork a
kernel thread.
 * @tf:          the trapframe info, which will be copied to child process's
proc->tf
 */
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;
    //LAB4:EXERCISE2 YOUR CODE
    /*
     * Some Useful MACROs, Functions and DEFINEs, you can use them in below
implementation.
     * MACROs or Functions:
     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack
```

```
     *   copy_mm:      process "proc" duplicate OR share process "current"'s mm
according clone_flags
     *                 if clone_flags & CLONE_VM, then "share" ; else "duplicate"
     *   copy_thread:  setup the trapframe on the  process's kernel stack top and
     *                 setup the kernel entry point and stack of process
     *   hash_proc:    add proc into proc hash_list
     *   get_pid:      alloc a unique pid for process
     *   wakeup_proc:  set proc->state = PROC_RUNNABLE
     * VARIABLES:
     *   proc_list:    the process set's list
     *   nr_process:   the number of process set
     */

    //    1. call alloc_proc to allocate a proc_struct
    //    2. call setup_kstack to allocate a kernel stack for child process
    //    3. call copy_mm to dup OR share mm according clone_flag
    //    4. call copy_thread to setup tf & context in proc_struct
    //    5. insert proc_struct into hash_list && proc_list
    //    6. call wakeup_proc to make the new child process RUNNABLE
    //    7. set ret vaule using child proc's pid
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = current;

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf);

    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);

    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;
}
```

函数设计说明如下：

原始代码：

首先设置返回值为不能fork新进程

判断进程数量是否到达上限 如果未达到上限 则设置返回值为无内存

添加代码：

首先调用上个问题写好的alloc_proc函数创建新进程

如果返回值为空 则直接用无内存的返回值退出do_fork函数

初始化新进程成功以后 依次执行

设置父进程为当前进程

分配进程的内核栈空间

复制当前进程的内存

复制当前进程的线程

设置新进程的pid并将进程加入hash表 插入进程链表 进程数量加一

最后唤醒新创建的进程 (设置进程状态为就绪状态)

返回新创建进程的pid

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。

##

在刚才编写的do_fork函数中可以看到 一个新创建的进程通过这个函数获取到一个pid

```
// get_pid - alloc a unique pid for process
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;
}
```

todo

- 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。
- 之所以在该函数中维护一个合法的`PID`的区间，是为了**优化时间效率**。如果简单的暴力搜索，则需要搜索大部分PID和所有的线程，这会使该算法的时间消耗很大，因此使用`PID`区间来优化算法。

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）

分析proc_fun函数：

```
// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load  base addr of "proc"'s new PDT
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;
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

首先判断运行的进程是不是当前正在运行的进程

如果是同一个进程 就不做任何操作 直接返回

否则进行下面的进程切换操作：

使用了local_intr_save 和 local_intr_restore保证进程切换操作中不会被中断

首先设置当前运行的进程为proc

然后从proc的结构体中load 栈指针esp 即设置内核栈地址

然后设置页表的基地址

最后切换上下文 执行proc进程

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？**

分析初始化函数可知有两个内核线程

```
// 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");
    }

    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);
}
```

首先创建了idle进程

然后创建了init进程

- **语句 `local_intr_save(intr_flag);....local_intr_restore(intr_flag);`在这里有何作用?请说明理由**

防止中间的语句被时钟中断 保证原子性操作

以进程切换为例，在proc_run中，当刚设置好current指针为下一个进程，但还未完全将控制权转移时，如果该过程突然被一个中断所打断，则中断处理例程的执行可能会引发异常，因为current指针指向的进程与实际使用的进程资源不一致。

扩展练习 实现支持任意大小的内存分配算法

运行说明
在Lab4的源代码中给出的SLOB算法实际上是First-Fit算法，因此我此次实现的任意大小内存分配算法为Best-Fit算法，主要修改了slob_alloc函数。

为了测试代码实现的正确性，仅仅需要执行命令 make score 查看运行结果

实现思路
首先确定所实现kmalloc在uCore内存管理中所处的地位，才能更好地理解函数调用关系。

在内核中，uCore的内存管理分为物理内存管理pmm和虚拟内存管理vmm。虚拟内存管理模块只负责管理页式地址映射关系，不负责具体的内存分配。而物理内存管理模块pmm不仅要管理连续的物理内存，还要能够向上提供分配内存的接口alloc_pages，分配出的物理内存区域可以转换为内核态可访问的区域（只要偏移KERNBASE）即可；也可以做地址映射转换给用户态程序使用。

但是，alloc_pages仅仅提供以页为粒度的物理内存分配，在uCore内核中，会频繁分配小型动态的数据结构（诸如vma_struct和proc_struct），这样以页为粒度进行分配既不节省空间，速度还慢，需要有一个接口能够提供更加细粒度的内存分配与释放工作，这就是slab和slob出现的原因：他们是一个中间件，底层调用alloc_pages接口，上层提供kmalloc接口，内部通过一系列机制管理页和内存小块的关系

仔细阅读Lab4中原有的slob代码，在每个小块内存的头部都存放了该块的大小和下一个空闲块的地址。kmalloc函数会首先判断需要分配的空间是否跨页，如果是则直接调用alloc_pages进行分配，否则就调用slob_alloc进行分配。

具体到Best-Fit算法的实现，实际很简单，仅仅需要在扫描空闲链表的时候动态记录与更新最好的块的地址即可，扫描完成之后，再选出刚刚找到的最合适的空间进行分配即可。无论是First-Fit、Best-Fit还是Worst-Fit，其释放的合并策略都是相同的，因此只需要修改slob_alloc函数即可。

实验代码

```
static void *slob_alloc(size_t size, gfp_t gfp, int align)
{
    assert( (size + SLOB_UNIT) < PAGE_SIZE );
    // This best fit allocator does not consider situations where align != 0
    assert(align == 0);
    int units = SLOB_UNITS(size);

    unsigned long flags;
    spin_lock_irqsave(&slob_lock, flags);

    slob_t *prev = slobfree, *cur = slobfree->next;
    int find_available = 0;
    int best_frag_units = 100000;
    slob_t *best_slob = NULL;
    slob_t *best_slob_prev = NULL;
```

```
    for (; ; prev = cur, cur = cur->next) {
        if (cur->units >= units) {
            // Find available one.
            if (cur->units == units) {
                // If found a perfect one...
                prev->next = cur->next;
                slobfree = prev;
                spin_unlock_irqrestore(&slob_lock, flags);
                // That's it!
                return cur;
            }
            else {
                // This is not a prefect one.
                if (cur->units - units < best_frag_units) {
                    // This seems to be better than previous one.
                    best_frag_units = cur->units - units;
                    best_slob = cur;
                    best_slob_prev = prev;
                    find_available = 1;
                }
            }

        }

        // Get to the end of iteration.
        if (cur == slobfree) {
            if (find_available) {
                // use the found best fit.
                best_slob_prev->next = best_slob + units;
                best_slob_prev->next->units = best_frag_units;
                best_slob_prev->next->next = best_slob->next;
                best_slob->units = units;
                slobfree = best_slob_prev;
                spin_unlock_irqrestore(&slob_lock, flags);
                // That's it!
                return best_slob;
            }
            // Initially, there's no available arena. So get some.
            spin_unlock_irqrestore(&slob_lock, flags);
            if (size == PAGE_SIZE) return 0;

            cur = (slob_t *)__slob_get_free_page(gfp);
            if (!cur) return 0;

            slob_free(cur, PAGE_SIZE);
            spin_lock_irqsave(&slob_lock, flags);
            cur = slobfree;
        }
    }
}
```

```
-> % make grade
Check VMM:                (s)
  -check pmm:                                 OK
  -check page table:                          OK
  -check slab:                                OK
  -check vmm:                                 OK
  -check swap page fault:                     OK
  -check ticks:                               OK
  -check initproc:                            OK
Total Score: 100/100
```