# lab4

## 实验流程

在lab3实验的基础上，完成vmm_init（）后，通过proc_init()开始进行进程的创建与执行

进程控制块

```
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 cr3;                              // CR3 register: 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
};
```

state：即线程所处状态，有PROC_UNINIT，PROC_SLEEPING，PROC_RUNNABLE，PROC_ZOMBIE四种状态

pid：即线程对应编号，实验中idle线程编号为0，且没有父线程，创建的init线程编号为1

runs：线程被调用的次数，在进行调度的时候会进行增加

kstack：线程所对应的内核栈，idle线程使用已经分配好的内核栈结构，其余线程则在创建时会分配2个页的大小，需要根据kstack的值正确设置tss，以便在进程切换发生中断时能够使用正确的栈，线程结束时，根据kstack快速进行空间的回收。

need_resched：代表当前进程是否需要被立即调度

parent：当前进程的父进程。

mm_struct变量，包含相关内存管理的信息，在lab3中涉及到虚拟内存的管理，但对于本实验的内核线程，内核线程常驻内存，因此不需要考虑换入换出的问题。mm中比较重要的pgdir即一级页表的信息由cr3代为储存。

context：进程上下文，具体的上下文切换位于*kern/process/switch.S*，其中eip即进行上下文切换后需要执行的指令地址，统一为forkets，esp为切换后的栈顶。ebx,ecx,edx为数据寄存器，ebp为栈底，esi和edi为源索引寄存器和目的索引寄存器

```
struct context {
    uint32_t eip;
    uint32_t esp;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t esi;
    uint32_t edi;
    uint32_t ebp;
};
```

tf：中断帧的指针，tf_regs中包含了诸多数据寄存器，gs,fs,es,ds,为段选择子，init线程的段选择子均指向内核的段，代表init线程运行在内核空间。

```
struct trapframe {
    struct pushregs tf_regs;
    uint16_t tf_gs;
    uint16_t tf_padding0;
    uint16_t tf_fs;
    uint16_t tf_padding1;
    uint16_t tf_es;
    uint16_t tf_padding2;
    uint16_t tf_ds;
    uint16_t tf_padding3;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding4;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding5;
} __attribute__((packed));
```

proc_list：管理所有内存块的链表

hash_list：管理所有进程块的哈希链表。

## proc_init函数

proc_init函数中首先初始化了链表头，随后开始进行第0个线程idle的创建，创建分为两部分

```
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;
    //内核栈
    idleproc->kstack = (uintptr_t)bootstack;
    idleproc->need_resched = 1;
    set_proc_name(idleproc, "idle");
    nr_process ++;

    current = idleproc;
    //通过kernel_thread函数来创建一个线程，指定该线程的执行入口和参数，以及写时复制
    //获得线程号
    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);
}
```

第一部分透过alloc_proc函数来完成，在alloc_proc函数中，分配了新的线程块管理结构体，并对状态，线程号，页目录表，调度次数，内核栈，上下文，用户空间，中断帧等信息进行了初步设置。

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {

    }
    //对proc的逐步初始化
    //状态设置为未初始化
    proc->state = PROC_UNINIT;
    //ID设置为-1;
    proc->pid = -1;
    //内核页目录表基址
    proc->cr3 = boot_cr3;
    //运行时间设置为0
    proc->runs = 0;
    //kstack暂未设置
    proc->kstack = 0;
    //不需要进行调度
    proc->need_resched = 0;
    //没有父进程
    proc->parent = NULL;
    //没有对应的用户空间
    proc->mm = NULL;
    //预留了上下文的空间
    memset(&(proc->context), 0, sizeof(struct context));
```

```
    //中断帧为空
    proc->tf = NULL;

    //状态和名字
    proc->flags = 0;
    memset(proc->name, 0, PROC_NAME_LEN);
    return proc;
}
```

第二部分由proc_init函数继续完成，直至最终把current指针指向了idle，代表idle线程已经开始运行。随后开始创建1号进程init。init线程的创建通过kernel_thread函数完成

```
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;
    tf.tf_regs.reg_ebx = (uint32_t)fn;
    tf.tf_regs.reg_edx = (uint32_t)arg;
    tf.tf_eip = (uint32_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

kernel_thread函数要返回的内容和随后do_fork函数要返回的内容相同，即创建出的线程号。kernel_thread创建线程的同时还指定了该线程的任务，即执行函数init_main，所提供的函数参数为字符串"Hello world!!"即向控制台打印字符串。在kernel_thread中，创建了init线程的中断帧。其代码段，数据段均指向了内核的代码段和数据段，代表着init线程本身也是运行在内核空间的。随后在其中的数据寄存器ebx中保存了任务函数init_main的地址，在edx寄存器中保存了调用时的参数字符串，设置tf.eip即中断返回后要运行的指令为kernel_thread_entry。完成了中断帧的初步设置，随后调用do_fork函数开始真正进行init线程块的创建。

```
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.call alloc_proc
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
        //现在current处于idle状态，
    proc->parent = current;
    //2.call setup_kstack
    //分配了两个页
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    //3.call copy_mm
```

```
        //不涉及要用户空间，即current的mm为空，
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    //4.call copy_thread
    copy_thread(proc, stack, tf);

    //5.把设置好的进程控制块放入hash_list和proc_list两个全局进程链表中；原子操作？
    bool intr_flag;
    //关闭中断
    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);
    //state=PROC_RUNNABLE
    wakeup_proc(proc);

    ret = proc->pid;
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
```

创建init线程块按照一定的步骤进行，首先完成了相关线程块的创建，分配其内核栈空间（setup_kstack），设置用户空间，本实验的两个线程均为用户进程，不涉及相关内容。随后进行**中断帧的设置**。然后关闭中断，在不会被打断的情况下获取了唯一的线程编号并将线程块接入到了对应的哈希链表和普通链表中。最后将线程块设置为PROC_RUNNABLE状态，代表准备运行。

中断帧的设置部分，在线程块的内核栈栈顶开辟出一片空间用作中断帧的存储。将预先设置好的临时中断帧进行了拷贝。随后补充了eax寄存器值为0，中断栈栈顶为传入的参数0，并开启了线程的中断使能。随后设置了上下文context的eip与esp，此时分别有两个eip与esp,分别为中断帧存储的eip为kern_thread_entry，esp为0，和上下文存储的eip为forkret（用以进行中断的函数入口）和esp（内核栈栈顶）。在进行上下文切换后，会从context.eip开始执行，转化到中断帧管理的状态，再由中断返回跳转到kern_thread_entry，准备进行任务函数的执行。

```
//分配两个页作为内核栈空间
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;
}
```

```
//init线程块中断帧的设置及完善
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;
    //中断使能
    proc->tf->tf_eflags |= FL_IF;
    //一个是中断帧中的eip，一个是context中的eip
    //前者指向kern_thread_entry
    //后者指向forkret

    //一个是中断帧的tf_esp,一个是contest中的esp
    //前者指向0
    //后者指向可用的kstack顶部proc->tf
    proc->context.eip = (uintptr_t)forkret;
    proc->context.esp = (uintptr_t)(proc->tf);
}
```

## 关于中断的关闭

```
bool intr_flag;
local_intr_save(intr_flag);
{

}
local_intr_restore(intr_flag);
```

```
static inline bool
__intr_save(void) {
    if (read_eflags() & FL_IF) {
        intr_disable();
        return 1;
    }
    return 0;
}

static inline void
__intr_restore(bool flag) {
    if (flag) {
        intr_enable();
    }
}

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

通过local_intr_save进行关闭中断操作。首先读取了eflags寄存器和中断使能信号，在当前中断使能的情况下，通过cli来关闭中断。同时传入的intr_flag也被赋值为1.通过local_intr_restore恢复中断。

至此，完成了init线程的创建，回到proc_init函数中将其线程名设置为init，proc_init函数也结束。在kern_init的最后，执行了cpu_idle函数，开始进行线程的调度。

## 线程调度

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

cpu_idle也可视为idle线程的任务函数，如果当前线程处于需要调度的状态，就执行schedule函数开始进行调度

```
void
schedule(void) {
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    //关闭中断
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        //当前是idle线程的话，就从头开始遍历，否则从当前线程开始向后继续寻找
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
        } while (le != last);
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

同样也是先关闭了中断，避免调度过程被打断导致程序崩溃。随后从进程链表中选取一个PROC_RUNNABLE的线程切换到执行状态，通过函数proc_run来完成。

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        //关闭中断
```

```
        local_intr_save(intr_flag);
        {
            current = proc;
            //pmm.c
            load_esp0(next->kstack + KSTACKSIZE);
            //x86.c
            lcr3(next->cr3);
            //上下文切换
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

在proc_run中加载了新的esp0寄存器和页目录表，然后开始进行上下文的切换

```
switch_to:                      # switch_to(from, to)

    # save from's registers
    movl 4(%esp), %eax          # eax points to from
    popl 0(%eax)                # save eip !popl
    movl %esp, 4(%eax)          # save esp::context of from
    movl %ebx, 8(%eax)          # save ebx::context of from
    movl %ecx, 12(%eax)         # save ecx::context of from
    movl %edx, 16(%eax)         # save edx::context of from
    movl %esi, 20(%eax)         # save esi::context of from
    movl %edi, 24(%eax)         # save edi::context of from
    movl %ebp, 28(%eax)         # save ebp::context of from

    # restore to's registers
    movl 4(%esp), %eax          # not 8(%esp): popped return address already
                                # eax now points to to
    movl 28(%eax), %ebp         # restore ebp::context of to
    movl 24(%eax), %edi         # restore edi::context of to
    movl 20(%eax), %esi         # restore esi::context of to
    movl 16(%eax), %edx         # restore edx::context of to
    movl 12(%eax), %ecx         # restore ecx::context of to
    movl 8(%eax), %ebx          # restore ebx::context of to
    movl 4(%eax), %esp          # restore esp::context of to

    pushl 0(%eax)               # push eip

    ret
```

从proc_run进入到switch_to函数的时候，已经完成了一部分栈的工作，即将参数从右向左入栈，并将返回地址也就是当前线程的eip入栈。因此此时esp处即当前线程的eip,esp+4即prev->context，esp+8即next->context。首先存储当前的上下文状态，将prev->context存储到eax中，此时栈顶即为current->eip,所以直接出栈。依次存储相关寄存器的值，完成旧进程寄存器状态的保存。

以同样的方式逆序进行，将新进程的上下文状态恢复到内核中，最后一步push即压入新进程的eip到栈顶作为返回地址。此处一点是获取新进程的上下文时采用的时esp+4,是因为在保存前一进程上下文时进行过了一次pop操作。

根据上文所提到，init进程上下文中所保存的eip为forkret，因此程序会跳转到forket处继续执行，forket依赖相关的中断帧来执行

```
static void
forkret(void) {
    //上下文切换后来到这里
    //forkrets位于trapentry.S中，以汇编实现
    forkrets(current->tf);
}
```

```
.globl __trapret
__trapret:
    # restore registers from stack
    popal

    # restore %ds, %es, %fs and %gs
    popl %gs
    popl %fs
    popl %es
    popl %ds

    # get rid of the trap number and error code
    addl $0x8, %esp
    iret

.globl forkrets
forkrets:
    # 把当前的函数调用的栈切换成了中断栈
    # set stack to this new process's trapframe
    # esp+4正好是current->tf
    movl 4(%esp), %esp
    jmp __trapret
```

在forkrets处，首先将当前栈切换到了中断栈处，根据中断帧中的内容依次还原了相关寄存器的数值，使得这里的相关段选择子指向内核空间的位置。在这里进行iret的时候，根据中断帧的eip进行返回，中断帧的eip设置为了***kernel_thread_entry***处，开始准备进行任务函数的执行。

```
.globl kernel_thread_entry
kernel_thread_entry:        # void kernel_thread(void)

    pushl %edx              # push arg
    call *%ebx              # call fn

    pushl %eax              # save the return value of fn(arg)
    call do_exit            # call do_exit to terminate current thread
```

根据在kernel_thread函数处对中断帧的设计。edx即传入的参数字符串，ebx为需要执行的函数的入口地址，最终的返回值保存在eax中也进行压栈等待返回，可以开始进行init线程的真正执行了。

# 实验习题

1.**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）**

context中保存的时线程的数据寄存器中的信息，其中的eip初始负责跳转到中断处，由中断返回再跳转到真正的任务函数，esp保存的则是线程的内核栈栈顶地址。其余的寄存器为执行过程中可能要保存的寄存器信息。

而trapframe中则保存了相关的段选择子，代表着当前线程是运行在用户空间还是内核空间。同时也包含eip代表中断返回时要跳转回的地址。trapframe中还保存了标志寄存器。

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。**

可以的，线程号的分配通过函数get_pid来完成

```
static int
get_pid(void) {
    //静态检查
    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;
}
```

last_pid保存了上一次分配的id值，next_safe则代表了小于next_safe范围内的值尚处在可用状态。具体的过程讲解再说

3.**在本实验的执行过程中，创建且运行了几个内核线程?**

两个线程吧，其中有一个是通过proc_run函数进行运行的

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

中断机制的关闭于打开，目的的避免调度过程中被打断导致的并发崩溃。具体的分析在上面有。

# challenge

在进行线程块对象分配的时候，调用了kmalloc函数

```
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
```

应该是要分配一个线程块结构体大小的空间

kmalloc调用了_kmalloc

```
void *
kmalloc(size_t size)
{
  return __kmalloc(size, 0);
}
```

```
static void *__kmalloc(size_t size, gfp_t gfp)
{
    slob_t *m;
    bigblock_t *bb;
    unsigned long flags;

    if (size < PAGE_SIZE - SLOB_UNIT) {
        m = slob_alloc(size + SLOB_UNIT, gfp, 0);
        return m ? (void *)(m + 1) : 0;
    }

    bb = slob_alloc(sizeof(bigblock_t), gfp, 0);
    if (!bb)
        return 0;

    bb->order = find_order(size);
    bb->pages = (void *)__slob_get_free_pages(gfp, bb->order);

    if (bb->pages) {
        spin_lock_irqsave(&block_lock, flags);
        bb->next = bigblocks;
        bigblocks = bb;
        spin_unlock_irqrestore(&block_lock, flags);
        return bb->pages;
    }

    slob_free(bb, sizeof(bigblock_t));
    return 0;
}
```

其中的gfp_t就是普通的int对象，赋值为0,SLOB_UNIT的值为slob_t结构体的大小，slob_t结构体包含了一个整数unit和一个指针。当要分配的空间大小小于（PAGE_SIZE - SLOB_UNIT）时，直接调用了slob_alloc函数进行分配，现在更名为了first_fit_alloc函数。

```
static void *first_fit_alloc(size_t size, gfp_t gfp, int align)
{
    //在size+SLOB_UNIT的基础上又加了一个SLOB_UNIT
  assert( (size + SLOB_UNIT) < PAGE_SIZE );

    slob_t *prev, *cur, *aligned = 0;
    //此处units应该时算出了总共需要多少个slob_t对象？
    int delta = 0, units = SLOB_UNITS(size);
    unsigned long flags;

    //与传入的参数slob_lock无关，只是根据flags来关闭了中断
    spin_lock_irqsave(&slob_lock, flags);
    prev = slobfree;
    //开始遍历arena，暂时还不清楚什么时候向里面存了东西
    for (cur = prev->next; ; prev = cur, cur = cur->next) {
        //给来的align为0
        if (align) {
            aligned = (slob_t *)ALIGN((unsigned long)cur, align);
            delta = aligned - cur;
        }
        //当前slob_t中剩余的unit数量多于想要分配的数量
        if (cur->units >= units + delta) { /* room enough? */
            //不对齐的话delta就为0
            if (delta) { /* need to fragment head to align? */
                aligned->units = cur->units - delta;
                aligned->next = cur->next;
                cur->next = aligned;
                cur->units = delta;
                prev = cur;
                cur = aligned;
            }

            if (cur->units == units) /* exact fit? */
                //相当于把当前的slob_t从链上摘掉了
                prev->next = cur->next; /* unlink */
            else { /* fragment */
                //进行分割
                prev->next = cur + units;
                prev->next->units = cur->units - units;
                prev->next->next = cur->next;
                cur->units = units;
            }
            //下一次的分配将从这里开始
            slobfree = prev;
            //恢复中断
            spin_unlock_irqrestore(&slob_lock, flags);
            return cur;
        }
        if (cur == slobfree) {
            //已经遍历完了一轮
            spin_unlock_irqrestore(&slob_lock, flags);

            if (size == PAGE_SIZE) /* trying to shrink arena? */
                return 0;
            //分配了空余的页出来
```

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

            slob_free(cur, PAGE_SIZE);
            spin_lock_irqsave(&slob_lock, flags);
            //开始继续寻找
            cur = slobfree;
        }
    }
}
```

现在开始尝试使用best-fit函数进行分配。输入的参数相同

```
static void *best_fit_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;
    //一直到这里都还和first-fit保持一致
    int find_available = 0;
    int best_frag_units = 100000;
    slob_t *best_slob = NULL;
    slob_t *best_slob_prev = NULL;

    //还是在现在的slobfree开始遍历
    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.
                    //分配后剩下的部分
                    best_frag_units = cur->units - units;
                    //从哪里分走
                    best_slob = cur;
                    //prev是谁
                    best_slob_prev = prev;
                    find_available = 1;
                }
```

```
            }

        }

        // Get to the end of iteration.
        if (cur == slobfree) {
            //遍历完了一轮
            if (find_available) {
                //和first-fit中分割不太适合的块的方法一样
                // 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.
            //开启下一轮循环的方式也一样，即没有大小富余的块，进行获取。
            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;
        }
    }
}
```