练习0：填写已有实验

以下操作都是将**内容**复制到 lab4 里，**不要复制整个文件！！**

将 lab1 的 `kern/debug/kdebug.c`、`kern/init/init.c` 以及 `kern/trap/trap.c` 复制到 lab4 里

再将 lab2 的 `kern/mm/pmm.c` 和 `kern/mm/default_pmm.c` 复制到 lab4 里

最后将 lab3 的 `kern/mm/vmm.c` 和 `kern/mm/swap_fifo.c` 复制到 lab4 里

练习1：分配并初始化一个进程控制块

alloc_proc 函数源码

写于：**kern/process/proc.c**

```c
// 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
     */
    }
    return proc;
}
```

alloc_proc 函数答案

照着注释里给的元素一个个填就行

相比于视频里给的元素来讲，这个函数的注释里没有 `list_link` 和 `hash_link` 元素，也就是这俩不需要初始化

```c
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->pid = -1;                                     // 进程ID
        memset(&(proc->name), 0, PROC_NAME_LEN);            // 进程名
        proc->state = PROC_UNINIT;                          // 进程状态
        proc->runs = 0;                                     // 进程时间片
        proc->need_resched = 0;                             // 进程是否能被调度
        proc->flags = 0;                                    // 标志位
        proc->kstack = 0;                                   // 进程所使用的内存栈地址
        proc->cr3 = boot_cr3;                               // 将页目录表地址设为内核页目录表基址
        proc->mm = NULL;                                    // 进程所用的虚拟内存
        memset(&(proc->context), 0, sizeof(struct context));  // 进程的上下文
        proc->tf = NULL;                                    // 中断帧指针
        proc->parent = NULL;                                // 父进程
    }
    return proc;
}
```

练习2：为新创建的内核线程分配资源

do_fork 函数源码

写于：**kern/process/proc.c**

```c
/* 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
fork_out:
    return ret;

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

E_NO_FREE_PROC 宏

写于：**libs/error.h**

```
#define E_NO_FREE_PROC      5   // Attempt to create a new process beyond
```

这个意思是再申请一个新进程的话就会超过规定的进程总数

MAX_PROCESS 宏

写于：**kern/process/proc.h**

```
#define MAX_PROCESS                 4096
```

规定最大的进程总数为 0x1000

E_NO_MEM 宏

写于：**libs/error.h**

```
#define E_NO_MEM            4   // Request failed due to memory shortage
```

由于内存不足，请求失败

setup_kstack 函数

写于：**kern/process/proc.c**

```
// setup_kstack - alloc pages with size KSTACKPAGE as process kernel stack
static int
setup_kstack(struct proc_struct *proc) {
    struct Page *page = alloc_pages(KSTACKPAGE);   // 申请 2 个连续的物理页
    if (page != NULL) {                            // 如果能获取到物理页地址
        proc->kstack = (uintptr_t)page2kva(page);  // 用 kstack 记录物理页的虚拟地址
        return 0;
    }
    return -E_NO_MEM;                              // 否则返回 -4，表示内存不足，申请失败
}
```

该函数的作用就是创建内核堆栈，会申请两个连续的物理页

KSTACKPAGE 宏

写于：**kern/mm/memlayout.h**

```
#define KSTACKPAGE          2                           // # of pages in kernel stack
```

规定内核堆栈中页的数目

kfree 函数

写于：**kern/mm/kmalloc.c**

```
void kfree(void *block)
{
    bigblock_t *bb, **last = &bigblocks;
    unsigned long flags;

    if (!block)
        return;

    if (!((unsigned long)block & (PAGE_SIZE-1))) {
        /* might be on the big block list */
        spin_lock_irqsave(&block_lock, flags);
        for (bb = bigblocks; bb; last = &bb->next, bb = bb->next) {
            if (bb->pages == block) {
                *last = bb->next;
                spin_unlock_irqrestore(&block_lock, flags);
                __slob_free_pages((unsigned long)block, bb->order);
                slob_free(bb, sizeof(bigblock_t));
                return;
            }
        }
        spin_unlock_irqrestore(&block_lock, flags);
    }
```

```

    slob_free((slob_t *)block - 1, 0);
    return;
}
```

具体内容以后分析，看函数名可知该函数的作用是释放内核堆块

copy_mm 函数

写于：**kern/process/proc.c**

```
// copy_mm - process "proc" duplicate OR share process "current"'s mm according clone_flags
//         - if clone_flags & CLONE_VM, then "share" ; else "duplicate"
static int
copy_mm(uint32_t clone_flags, struct proc_struct *proc) {
    assert(current->mm == NULL);
    /* do nothing in this project */
    return 0;
}
```

将父进程的内存信息复制到子进程

put_kstack 函数

写于：**kern/process/proc.c**

```
// put_kstack - free the memory space of process kernel stack
static void
put_kstack(struct proc_struct *proc) {
    free_pages(kva2page((void *)(proc->kstack)), KSTACKPAGE);
}
```

释放内核栈

copy_thread 函数

写于：**kern/process/proc.c**

```
// copy_thread - setup the trapframe on the  process's kernel stack top and
//             - setup the kernel entry point and stack of process
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);
}
```

复制父进程的中断帧和上下文

KSTACKSIZE 宏

写于：**kern/mm/memlayout.h**

```
#define KSTACKSIZE          (KSTACKPAGE * PGSIZE)       // sizeof kernel stack
```

KSTACKSIZE 等于 0x2000

local_intr_save 宏

写于：**kern/sync/sync.h**

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

返回值是 1 代表 eflags 上有中断标志位，此时内核禁止中断发生，保护代码运行

__intr_save 函数

写于：**kern/sync/sync.h**

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

read_eflags 函数

写于：**libs/x86.h**

```
static inline uint32_t
read_eflags(void) {
    uint32_t eflags;
    asm volatile ("pushfl; popl %0" : "=r" (eflags));
    return eflags;
}
```

返回 eflags 寄存器里的值

FL_IF 宏

写于：**kern/mm/mmu.h**

```
#define FL_IF           0x00000200  // Interrupt Flag
```

中断标志位

intr_disable 函数

写于：**kern/driver/intr.c**

```c
/* intr_disable - disable irq interrupt */
void
intr_disable(void) {
    cli();
}
```

禁止中断发生，保护代码运行

cli 函数

写于：**libs/x86.h**

```c
static inline void
cli(void) {
    asm volatile ("cli" ::: "memory");
}
```

禁止中断发生，保护代码运行

get_pid 函数

写于：**kern/process/proc.c**

```c
// get_pid - alloc a unique pid for process
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);        // 静态断言，要求 MAX_PID 必须大于 MAX_PROCESS
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    static int next_safe = MAX_PID, last_pid = MAX_PID;  // 一定要注意这俩是 static 修饰的变量
    /* *
     * 如果有严格的 next_safe > last_pid + 1，那么就可以直接取 last_pid + 1 作为新的 pid
     * （需要 last_pid 没有超出 MAX_PID 从而变成 1）
     * */
    if (++ last_pid >= MAX_PID) {                // 要是 last_pid >= MAX_PID
        last_pid = 1;                            // 使 last_pid 变为 1
        goto inside;
    }
    if (last_pid >= next_safe) {                 // 如果 last_pid >= next_safe
    inside:
        next_safe = MAX_PID;                     // 使 next_safe 等于 MAX_PID
    repeat:
        le = list;
        while ((le = list_next(le)) != list) {   // 遍历进程控制块列表
            proc = le2proc(le, list_link);       // 从 le 得到对应的 proc_struct 结构体基地址
            if (proc->pid == last_pid) {         // 如果遍历的进程控制块的 pid 等于 last_pid
                if (++ last_pid >= next_safe) {  // 如果满足 next_safe > last_pid + 1
                    if (last_pid >= MAX_PID) {   // 要是 last_pid >= MAX_PID
```

```c
                        last_pid = 1;           // 使 last_pid 变为 1
                    }
                    next_safe = MAX_PID;         // 使 next_safe 变为 MAX_PID
                    goto repeat;                 // 跳转到 repeat
                }
            }
            // 如果遍历的进程控制块的 pid 大于 last_pid 并且 next_safe 大于遍历的进程控制块的 pid
            else if (proc->pid > last_pid && next_safe > proc->pid) {
                next_safe = proc->pid;           // 更新 next_safe 为遍历的进程控制块的 pid
            }
        }
    }
    return last_pid;
}
```

如果在进入函数的时候，这两个变量之后没有合法的取值，也就是说 `next_safe > last_pid + 1` 不成立，那么进入循环

在循环之中首先通过 `if (proc->pid == last_pid)` 这一分支确保了不存在任何进程的 pid 与 last_pid 重合

然后再通过 `if (proc->pid > last_pid && next_safe > proc->pid)` 这一判断语句

保证了不存在任何已经存在的 pid 满足：`last_pid < pid < next_safe`

这样就确保了最后能找到这么一个满足条件的区间，从而得到合法的 pid

MAX_PROCESS 宏

写于：**kern/process/proc.h**

```c
#define MAX_PROCESS                 4096
```

MAX_PROCESS 等于 0x1000

MAX_PID 宏

写于：**kern/process/proc.h**

```c
#define MAX_PID                     (MAX_PROCESS * 2)
```

MAX_PID 等于 0x2000

le2proc 宏

写于：**kern/process/proc.h**

```c
#define le2proc(le, member)         \
    to_struct((le), struct proc_struct, member)
```

依靠作为 proc_struct 结构体中 member 成员变量的 le 变量，得到 le 成员变量所对应的 proc_struct 结构体的基地址

hash_proc 函数

写于：**kern/process/proc.c**

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

在哈希列表中的 `proc->hash_link` 节点后面添加新节点

hash_list 结构体数组

写于：**kern/process/proc.c**

```
// has list for process set based on pid
static list_entry_t hash_list[HASH_LIST_SIZE];
```

一共有 0x400 个元素

HASH_SHIFT 宏

写于：**kern/process/proc.c**

```
#define HASH_SHIFT          10
```

HASH_LIST_SIZE 宏

写于：**kern/process/proc.c**

```
#define HASH_LIST_SIZE      (1 << HASH_SHIFT)
```

HASH_LIST_SIZE 等于 0x400

pid_hashfn 宏

写于：**kern/process/proc.c**

```
#define pid_hashfn(x)       (hash32(x, HASH_SHIFT))
```

取高 HASH_SHIFT 位，这个值做为表头数组的索引

hash32 函数

写于：**libs/hash.c**

```c
/* *
 * hash32 - generate a hash value in the range [0, 2^@bits - 1]
 * @val:    the input value
 * @bits:   the number of bits in a return value
 *
 * High bits are more random, so we use them.
 * */
uint32_t
hash32(uint32_t val, unsigned int bits) {
    uint32_t hash = val * GOLDEN_RATIO_PRIME_32;
    return (hash >> (32 - bits));
}
```

就是做散列运算的函数，找出对应的散列下标

GOLDEN_RATIO_PRIME_32 宏

写于：**libs/hash.c**

```c
/* 2^31 + 2^29 - 2^25 + 2^22 - 2^19 - 2^16 + 1 */
#define GOLDEN_RATIO_PRIME_32       0x9e370001UL
```

这个值作为 hash 的乘数能够较广泛地使关键字平均存在，这样引起的冲突最小

local_intr_restore 宏

写于：**kern/sync/sync.h**

```c
#define local_intr_restore(x)   __intr_restore(x);
```

如果 flag 不为 0，则允许中断发生

__intr_restore 函数

写于：**kern/sync/sync.h**

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

如果 flag 不为 0，则允许中断发生

intr_enable 函数

写于：**kern/driver/intr.c**

```c
/* intr_enable - enable irq interrupt */
void
intr_enable(void) {
    sti();
}
```

允许中断发生

sti 函数

写于：**libs/x86.h**

```
static inline void
sti(void) {
    asm volatile ("sti");
}
```

允许中断发生

wakeup_proc 函数

写于：**kern/schedule/sched.c**

```
void
wakeup_proc(struct proc_struct *proc) {
    assert(proc->state != PROC_ZOMBIE && proc->state != PROC_RUNNABLE);
    proc->state = PROC_RUNNABLE;
}
```

将该进程的状态设置为可以运行

proc_state 枚举

写于：**kern/process/proc.h**

```
// process's state in his life cycle
enum proc_state {
    PROC_UNINIT = 0,  // uninitialized
    PROC_SLEEPING,    // sleeping
    PROC_RUNNABLE,    // runnable(maybe running)
    PROC_ZOMBIE,      // almost dead, and wait parent proc to reclaim his resource
};
```

do_fork 函数答案

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;                      // 设置返回值为 -5
    struct proc_struct *proc;                       // 定义 proc_struct 结构体指针变量 proc（子进程）
    if (nr_process >= MAX_PROCESS) {                // 进程总数（全局变量）如果 >= 0x1000
        goto fork_out;                              // 就跳转到 fork_out 地址，此时返回值是 -5
    }
    ret = -E_NO_MEM;                                // 更换返回值为 -4
    if ((proc = alloc_proc()) == NULL) {            // 申请一个 proc_struct 结构体
        goto fork_out;                              // 若是申请失败就跳转到 fork_out 地址，此时返回值是 -4
```

```
    }

    proc->parent = current;                          // 设置当前进程的父进程地址

    if (setup_kstack(proc) != 0) {                   // 分配并初始化内核栈，返回值是 0 则成功分配
        goto bad_fork_cleanup_proc;                  // 创建失败，即内存不足，则跳转到 bad_fork_cleanup_proc
    }
    if (copy_mm(clone_flags, proc) != 0) {           // 将父进程的内存信息复制到子进程
        goto bad_fork_cleanup_kstack;                // 出错就跳转到 bad_fork_cleanup_kstack
    }
    copy_thread(proc, stack, tf);                    // 复制父进程的中断帧和上下文

    bool intr_flag;
    local_intr_save(intr_flag);                      // 禁止中断发生，保护代码运行
    {
        proc->pid = get_pid();                       // 获取该程序的 pid
        hash_proc(proc);                             // 将该节点添加进哈希列表
        list_add(&proc_list, &(proc->list_link));    // 将该节点添加进双向链表
        nr_process ++;                               // 记录总进程数的变量自增 1
    }
    local_intr_restore(intr_flag);                   // 如果 intr_flag 不为 0，则允许中断发生

    wakeup_proc(proc);                               // 唤醒该进程

    ret = proc->pid;                                 // 更新返回值为该进程的 pid
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);                                // 释放内核栈
bad_fork_cleanup_proc:
    kfree(proc);                                     // 释放申请的物理页，跳转到 fork_out
    goto fork_out;
}
```

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

proc_run 函数

写于：**kern/process/proc.c**

```
// 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);                         // 禁止中断发生，
保护代码运行
        {
            current = proc;                                 // 将当前进程换为
要切换到的进程
            load_esp0(next->kstack + KSTACKSIZE);           // 将 tf->esp0
设置为内核栈地址
            lcr3(next->cr3);                                // 将 next->cr3
变量的值存储到 cr3 寄存器中
            switch_to(&(prev->context), &(next->context));  // 进行上下文切换
        }
        local_intr_restore(intr_flag);
    }
}
```

load_esp0 函数

写于：**kern/mm/pmm.c**

```
/* *
 * load_esp0 - change the ESP0 in default task state segment,
 * so that we can use different kernel stack when we trap frame
 * user to kernel.
 * */
void
load_esp0(uintptr_t esp0) {
    ts.ts_esp0 = esp0;
}
```

lcr3 函数

写于：**libs/x86.h**

```
static inline void
lcr3(uintptr_t cr3) {
    asm volatile ("mov %0, %%cr3" :: "r" (cr3) : "memory");
}
```

将 cr3 变量的值存储到 cr3 寄存器中

switch_to 函数

写于：**kern/process/switch.S**

```
.text
.globl switch_to
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
```

这个函数是保存前一个进程的其他 7 个寄存器到 context 中，后面的指令和前面的相反

这个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应的寄存器中

扩展练习Challenge：实现支持任意大小的内存分配算法

后面再写

参考链接

https://www.jianshu.com/p/50dd281a82f0

https://yuerer.com/操作系统-uCore-Lab-4/

https://blog.csdn.net/weixin_43995093/article/details/105975763