

ucoreOS操作系统实验——Lab4

问题发现与改进

- 这次像lab3一样的打patch出现了问题，然后使用git提交版本号打也出现了小的问题，然后通过labcodes_answer打patch就成功了。。并不知道为什么，可能answer的patch和Makefile就是对的吧_(:з」∠)_不过反正这个代码是为了make grade通过，所以不是我以前写了一大堆注释的不简洁的代码，肯定效果更好，不会因为多的乱七八糟的注释而变得混乱
- 第一次运行 `make qemu` 以后得到了理想的输出，但是和示例输出不一样的是，我多了中断的堆栈信息

```
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:345:
process exit!!.
```

```
stack traceback:
ebp:0xc0330fa8 eip:0xc0101e01 args:0xc01093a8 0xc012e044 0xc032e0c0 0xc0330fdc
kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0330fc8 eip:0xc01017d4 args:0xc010ca35 0x00000159 0xc010ca84 0xc012e044
kern/debug/panic.c:27: __panic+107
ebp:0xc0330fe8 eip:0xc0109926 args:0x00000000 0xc010cb04 0x00000000 0x00000010
kern/process/proc.c:345: do_exit+28
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

```

++ 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:316:

process exit!!.

Welcome to the kernel debug monitor!!

Type 'help' for a list of commands.

K>

```

然后尝试了一下 `make grade` 发现只有90分，显示是少了check_slab的部分

但是我明明就有

```

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 31919
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!

```

又看了看grade.sh，估计是固定检查 `make qemu` 的输出是否有 函数 `succeeded !` 这一句，大概是语句写错了

找了半天都没找到哪里有check_slab(甚至都想去改grade.sh了。)，最后在kern/mm/kmalloc.c里面找到了，然后改成succeeded!就对了，这都什么莫名其妙的错。。。

练习0

见问题改进

练习一

- 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。
- 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。

• 关键数据结构

根据提示先行查看proc结构的成员变量都是什么以及功能是什么

```
//kern/process/proc.h    *表示本lab比较重要
struct proc_struct {
    enum proc_state state; //进程所处的状态 未初始化、sleep、Zombie、运行  *
    int pid;                //进程ID  *
    int runs;               //运行时间
    uintptr_t kstack;       //内核栈，记录了分配给该线程的内核栈的位置。对内核线程是运行时的程序使用的栈；而对是
    发生特权级改变的时候使保存被打断的硬件信息用的栈  *
    volatile bool need_resched; // 需要被调度，即如果为1则需要调度器调度其他的process
    struct proc_struct *parent; // 父进程
    struct mm_struct *mm; // 内存管理的信息，包括内存映射列表、页表指针等等，这里其实不用考虑换页
    struct context context; // 进程的上下文，用于进程切换，通用寄存器等  *
    struct trapframe *tf; // 中断帧的指针 指向当前中断状态  *
    uintptr_t cr3; // 保存页表的物理地址PDT 进程切换的时候方便直接使用lcr3实现页表切换
    uint32_t flags; // 标志位
    char name[PROC_NAME_LEN + 1]; // 进程名
    list_entry_t list_link; // 进程链表
    list_entry_t hash_link; // 哈希?
};
```

• 实现

再根据proc.c里面的help_comment可以很容易实现练习一

除了特定的几个属性其它都赋0/NULL即可

```

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 Proc
 *     uintptr_t kstack;                // Process kernel stack
 *     volatile bool need_resched;      // bool value: need to be s
 *     struct proc_struct *parent;      // the parent process
 *     struct mm_struct *mm;            // Process's memory manager
 *     struct context context;          // Switch here to run proce
 *     struct trapframe *tf;            // Trap frame for current s
 *     uintptr_t cr3;                   // CR3 register: the base a
 *     uint32_t flags;                  // Process flag
 *     char name[PROC_NAME_LEN + 1];    // Process name
 */
proc->state = PROC_UNINIT; // 状态尚未初始化
proc->cr3 = boot_cr3;      // pmm.c
proc->pid = -1;
proc->runs = 0; // 对其他成员变量清零处理
proc->kstack = 0;
proc->need_resched = 0;
proc->parent = NULL;
proc->mm = NULL;
memset(&proc->context, 0, sizeof(struct context));
// 使用memset函数清零占用空间较大的成员变量，如数组，结构体等
proc->tf = NULL;
proc->flags = 0;
memset(proc->name, 0, PROC_NAME_LEN);
}

```

● 回答问题

- 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）

- context：进程上下文,用于在上下文切换时保存当前通用寄存器(除%eax)及eip的值

除了为了简化切换模式而省略掉的返回寄存器%eax(可以在栈上对应找到)，保存其它所有通用寄存器以及eip的值

```

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;
};

```

参考proc_run()和copy_thread()

- tf：中断帧，调度往往发生在时钟中断的时候，所以调度执行进程的时候，需要进行中断返回

tf变量的作用在于在构造出了新的线程的时候，如果要将控制权交给这个线程，是使用中断返回的方式进行的，因此需要构造出一个伪造的中断返回现场，即trapframe，使得可以正确地将控制权转交给新的线程

具体切换到新的线程的做法为，调用switch_to(switch.S)函数，然后在该函数中进行函数返回，直接跳转到forkret函数，最终进入中断返回函数__trapret，之后便可以根据tf中构造的中断返回地址切换到新的线程

练习二

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

● 相关宏即函数定义

```
alloc_proc//proc.c刚完成的 分配一个进程
setup_kstack//proc.c给线程内核栈分配一个大小为KSTACKPAGE(2Page 8KB)的页
copy_mm//proc.c 根据clone_flags对虚拟内存空间进行拷贝 如果和CLONE_VM(pmm.h)一致则共享否则赋值
copy_thread//proc.c 拷贝设置tf以及context(返回eip和中断帧中的栈顶指针esp)中堆栈的信息
hash_proc//proc.c 把进程加到哈希表里
get_pid//proc.c 为新进程创建一个pid
wakeup_proc//sched.c 通过将状态置为runable达到唤醒进程的目的
```

● 实现

同样照着comment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;
    //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

    // 为新线程分配PCB
    if ((proc = alloc_proc()) == NULL)
        goto fork_out; // 判断是否分配到内存空间
    assert(setup_kstack(proc) == 0);
    // 设置内核栈
    assert(copy_mm(clone_flags, proc) == 0);
    // 对虚拟内存空间进行拷贝, lab4内核线程之间共享一个虚拟内存空间, 所以并不需要进行任何操作
    copy_thread(proc, stack, tf);
    // 在内核栈上面设置伪造好的tf, 便于利用iret命令将控制权转移给新的线程
    proc->pid = get_pid(); // 创建pid
    hash_proc(proc); // 将线程放入使用hash表, 加速查找
    nr_process++; // 全局线程数加1
    list_add(&proc_list, &proc->list_link); // 将线程加入链表
    wakeup_proc(proc); // 唤醒线程
    ret = proc->pid; // 返回新线程的pid

fork_out:
    return ret;

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

```

● 回答问题

- 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。

能做到

查看 `get_pid` 代码

```
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;
    //如果next_safe > last_pid + 1成立, 以下两个if判断之后直接返回即可
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;
        goto inside;
    }
    if (last_pid >= next_safe) { //不成立
inside:
        next_safe = MAX_PID;
repeat:
        le = list;
        while ((le = list_next(le)) != list) {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid) { //循环找一个(last_pid,next_safe)之间的
                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;
}
```

- 在该函数中使用了两个静态局部变量 `next_safe` 和 `last_pid`，在每次进入 `get_pid` 函数的时候，这两个变量的数值之间的取值均是合法(尚未使用)的 `pid`，如果有严格的 `next_safe > last_pid + 1`，就可以直接取 `last_pid + 1` 作为新的 `pid`（`last_pid` 就是上一次分配的 `PID`）
- 如果 `next_safe > last_pid + 1` 不成立，则在循环中通过 `if (proc->pid == last_pid)` 确保不存在任何进程的 `pid` 与 `last_pid` 相同，再通过 `if (proc->pid > last_pid && next_safe > proc->pid)` 保证了不存在任何已经存在的 `pid` 满足：`last_pid < pid < next_safe`，这样就保证最后能够找到一个满足条件的区间，来获得合法的 `pid`

练习三

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

`proc_run` 函数的作用是让线程在CPU上运行起来,即将CPU的控制权交给指定线程

• 执行过程

```
void proc_run(struct proc_struct *proc) {
    if (proc != current) { // 判断指定线程是否正在运行
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
```



```

    local_intr_save(intr_flag); //sync.h 关闭中断
    {
        current = proc; //current->将要执行的process
        load_esp0(next->kstack + KSTACKSIZE); //pmm.c 设置TSS
        lcr3(next->cr3); // libs/x86.h 内联编译 修改当前cr3为需要运行线程的页目录表PDT
        switch_to(&(prev->context), &(next->context)); //切换到新的线程
    }
    local_intr_restore(intr_flag); //恢复中断
}
}

```

- 保存FL_IF(中断标志位intr_flag)并禁止中断
- 将current指针指向将要执行的进程，设置 任务状态段ts 中特权态0下的栈顶指针 esp0 为将要执行线程(next)的内核栈栈顶，即next->kstack + KSTACKSIZE
- 加载新的页表，设置CR3寄存器的值为 next->cr3，完成进程间的页表切换(由于lab4都是内核进程，所以这一步其实没用)
- 调用switch_to进行切换，switch_to执行到 ret 后就算切换成功了
- 当执行proc_run的进程恢复执行之后，恢复FL_IF

• 回答问题

■ 查看init_proc及运行结果可得

- 两个：idleproc(pid=0)和initproc(pid=1)
 - idleproc 最初的内核线程，在完成新的内核线程的创建以及各种初始化工作之后，进入死循环不断寻找可以调度的任务执行
 - initproc 用于打印"Hello World"的线程
- 该语句作用是关闭中断，使得在这个语句块内的执行内容不会被中断打断，是一个原子操作
- 在进程切换过程需要避免中断干扰以免产生不必要的错误，所以在切换进程期间将FL_IF(中断标志位)保存并禁止中断，等到进程切换完毕之后再恢复FL_IF

实验结果

完成之后运行 make qemu 和 make grade 可以得到如下结果

```

VNC server running on 127.0.0.1:5900
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xc0100036 (phys)
  etext 0xc010a84e (phys)
  edata 0xc012b000 (phys)
  end    0xc012e158 (phys)
Kernel executable memory footprint: 185KB
ebp:0xc0127f48 eip:0xc0101e01 args:0x00010094 0x00010094 0xc0127f78 0xc01000cc
  kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0127f58 eip:0xc01020fd args:0x00000000 0x00000000 0x00000000 0xc0127fc8
  kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0127f78 eip:0xc01000cc args:0x00000000 0xc0127fa0 0xffff0000 0xc0127fa4
  kern/init/init.c:57: grade_backtrace2+19
ebp:0xc0127f98 eip:0xc01000ee args:0x00000000 0xffff0000 0xc0127fc4 0x0000002a
  kern/init/init.c:62: grade_backtrace1+27
ebp:0xc0127fb8 eip:0xc010010b args:0x00000000 0xc0100036 0xffff0000 0xc0100079
  kern/init/init.c:67: grade_backtrace0+19

```



```

ebp:0xc0127fd8 eip:0xc010012c args:0x00000000 0x00000000 0x00000000 0xc010a860
kern/init/init.c:72: grade_backtrace+26
ebp:0xc0127ff8 eip:0xc0100086 args:0xc010acb8 0xc010acc0 0xc0102086 0xc010acdf
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, 000ffffff], 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 31919
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]

```

```

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:345:
process exit!!.
```

```

user@ubuntu:~/Desktop/labcodes_answer/lab4_result$ make grade
Check VMM: (1.3s)
-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
```

Challenge

- 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。。

看到是slab相关其实我就不想做。。。

参考kern/mm/kmalloc.c中关于SLOB和SLAB的解释以及Linux的实现文档

大概看懂了ucore是怎么做的，但要让我移植到first fit/best fit上一时半会还想不出来，所以就先空着，如果后面lab都比较顺利的话回来看看SLAB的实现