Lab4 实验

练习 0：

1、每次都需要用 meld 合并前面的代码，为了以防万一，我将 Lab4 分别与前面的实验用 meld 对比，保证我合并过来的正确性。另外写一些有趣的，如下图所示。

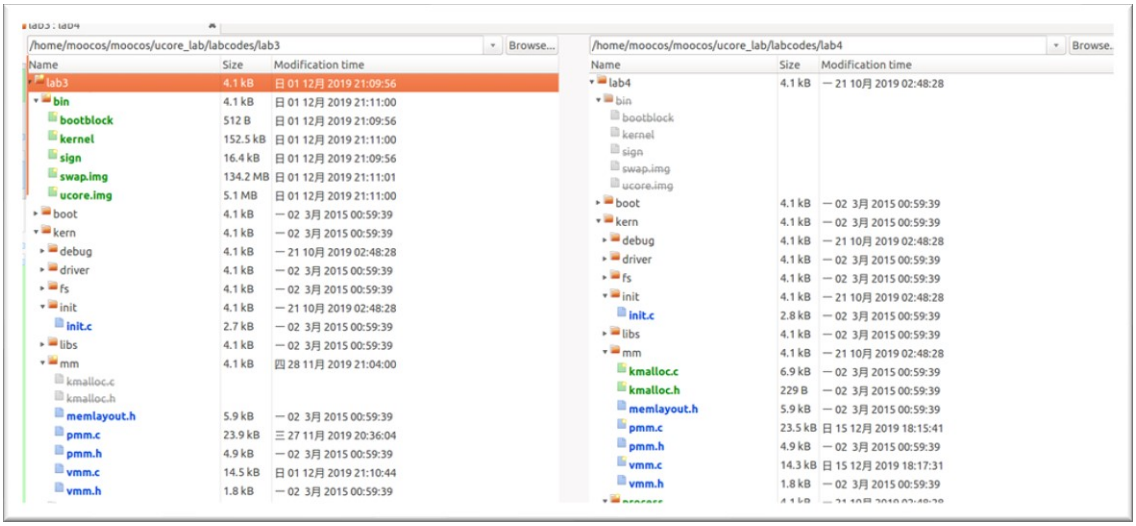


图 1　meld 使用

下次如果有人说他用过 meld，就可以问他绿色和蓝色分别代表什么意思（虽然没什么太大意义），绿色代表有一个文件夹里面没有这个文件，蓝色表示两个文件夹里都有这个文件但是内容不一样。

练习 1：

1、第一眼看到这个练习的时候给我吓坏了，12 个变量的初始化，这有点难了……后来继续往下看实验流程执行概述的分析，又不那么难了，好像还是可以写的。于是乎我就尝试开始写，指针全为 NULL，int、uint32_t（32 位二进制）和 uintptr_t（32 位二进制）全为 0，need_resched 置 0（不能置 1，否则马上就调用 schedule

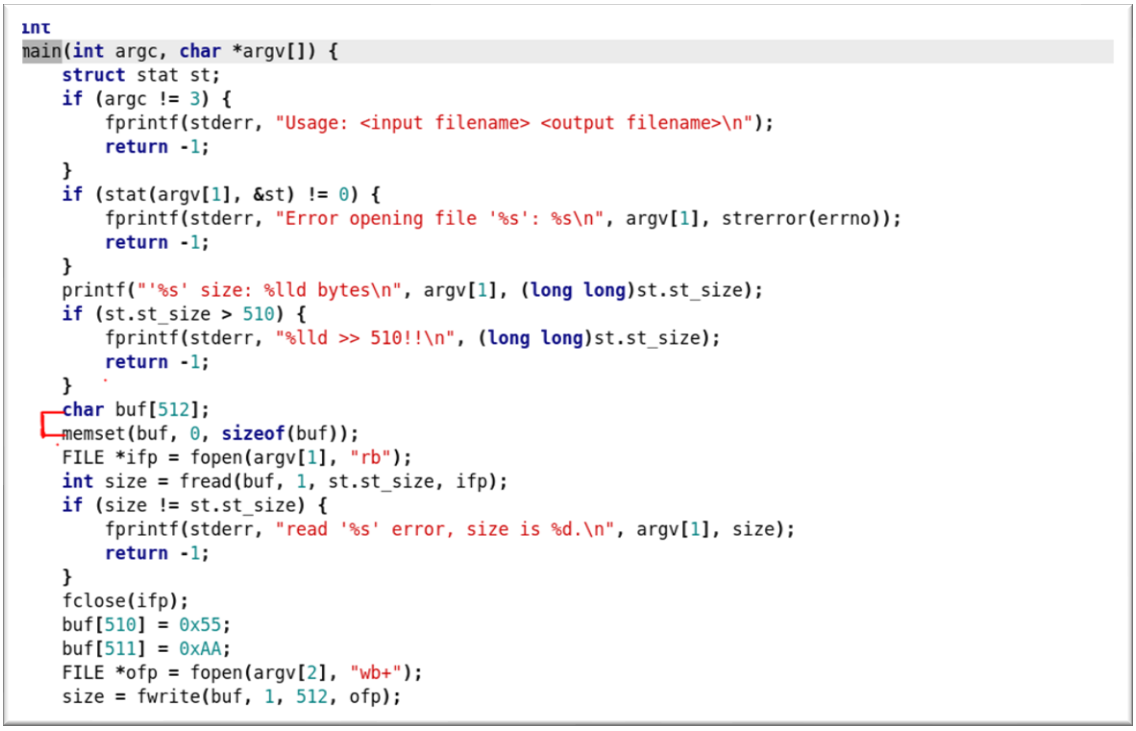
函数切换其他进程执行，这个进程无法执行)，cr3 等于 boot_cr3(明由于该内核线程在内核中运行，故采用为 ucore 内核已经建立的页表，即设置为在 ucore 内核页表的起始地址 boot_cr3)，还有两个类型不会初始化。

```
//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(PD
 *       uint32_t flags;                             // Process flag
 *       char name[PROC_NAME_LEN + 1];               // Process name
 */
 proc->state = PROC_UNINIT;
 proc->pid = -1;
 proc->runs = 0;
 proc->kstack = 0;
 proc->need_resched = 0;
 proc->parent = NULL;
 proc->mm = NULL;
 proc->context =
 proc->tf = NULL;
 proc->cr3 = boot_cr3;
 proc->flags = 0;
 proc->name =
}
```

图 2　练习 1 第一版代码

于是我开始思考,，context 是一个结构类型，name 是一个数组，第一个想法把 context 里面所有变量拿出来单独初始化，name 写个循环即可，但是这样做实在是有点太丢人了，刚学语言编程的时候就是这样做的……

最后我去翻了一下前面的一些初始化的例子，在 main 函数里面发现了数组初始化的样例。



```
int
main(int argc, char *argv[]) {
    struct stat st;
    if (argc != 3) {
        fprintf(stderr, "Usage: <input filename> <output filename>\n");
        return -1;
    }
    if (stat(argv[1], &st) != 0) {
        fprintf(stderr, "Error opening file '%s': %s\n", argv[1], strerror(errno));
        return -1;
    }
    printf("'%s' size: %lld bytes\n", argv[1], (long long)st.st_size);
    if (st.st_size > 510) {
        fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
        return -1;
    }
    char buf[512];
    memset(buf, 0, sizeof(buf));
    FILE *ifp = fopen(argv[1], "rb");
    int size = fread(buf, 1, st.st_size, ifp);
    if (size != st.st_size) {
        fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
        return -1;
    }
    fclose(ifp);
    buf[510] = 0x55;
    buf[511] = 0xAA;
    FILE *ofp = fopen(argv[2], "wb+");
    size = fwrite(buf, 1, 512, ofp);
```

图 3　main 函数部分截图

没错，就是这个标红的 buf，用了 memset 函数来进行初始化；接着又找了到结构类型用 memset 进行初始化的例子（kernel_thread 函数），于是乎练习一的任务就大功告成。顺便说一句，用 understand 找 memset 函数在哪里使用过非常方便，只需点击一次鼠标。

```
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->state = PROC_UNINIT;
 proc->pid = -1;
 proc->runs = 0;
 proc->kstack = 0;
 proc->need_resched = 0;
 proc->parent = NULL;// type * is NULL
 proc->mm = NULL;
 memset(&(proc->context), 0, sizeof(struct context));
 proc->tf = NULL;
 proc->cr3 = boot_cr3;
 proc->flags = 0;
 memset(proc->name, 0, sizeof(proc->name));
}
```

图 4　练习 1 最后代码

2、context 是进程的上下文，用于保存当前被打断的寄存器信息，使得下一次该进程被执行时可以恢复程序之前的“执行现场”；tf 是中断帧的指针，总是指向内核的某个位置，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。

练习 2：

1、do_fork 函数的相关作用在实验指导书中写的非常清楚，该题的注释也非常清楚，于是我很快就写出了第一版代码，如下图所示。

```
    //    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 wakup_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;
    }
    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);
    proc->pid = get_pid();
    hash_proc(proc);
    list_add(&proc_list,&(proc->list_link));
    wakeup_proc(proc);
    ret = proc->pid;
    nr_process++;
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
```

图 5　练习 2 第一版代码

运行之后出现问题，报错显示“kernel panic at kern/process/proc.c:384: create init_main failed.”。于是我去看了第 384 行，发现这个错误出现的原因是因为 pid<=0。仔细分析了一下，如果代码运行到 get_pid 函数这里，get_pid 肯定会返回一个大于 0 的 pid，现在 pid<=0，说明根本没有运行到这里。这说明前面有 if 语句判断错误，直接跳转到错误的位置。我找到 setup_kstack 和 copy_kstack 两个函数，并记录返回值，找到问题所在：这俩函数如果都正确无误返回 0，否则返回其他值，于是修改得到最终代码，成功运行。

```
    //    6. call wakup_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;
    }
    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);
    proc->pid = get_pid();
    hash_proc(proc);
    list_add(&proc_list,&(proc->list_link));
    wakeup_proc(proc);
    ret = proc->pid;
    nr_process++;
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
```

图 6　练习 2 最终代码

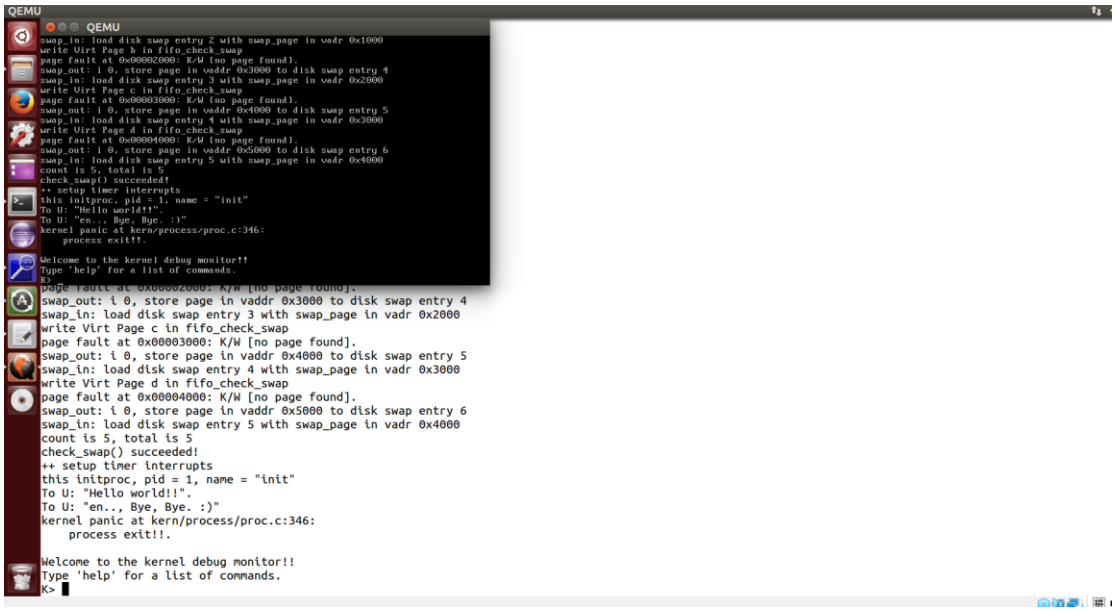
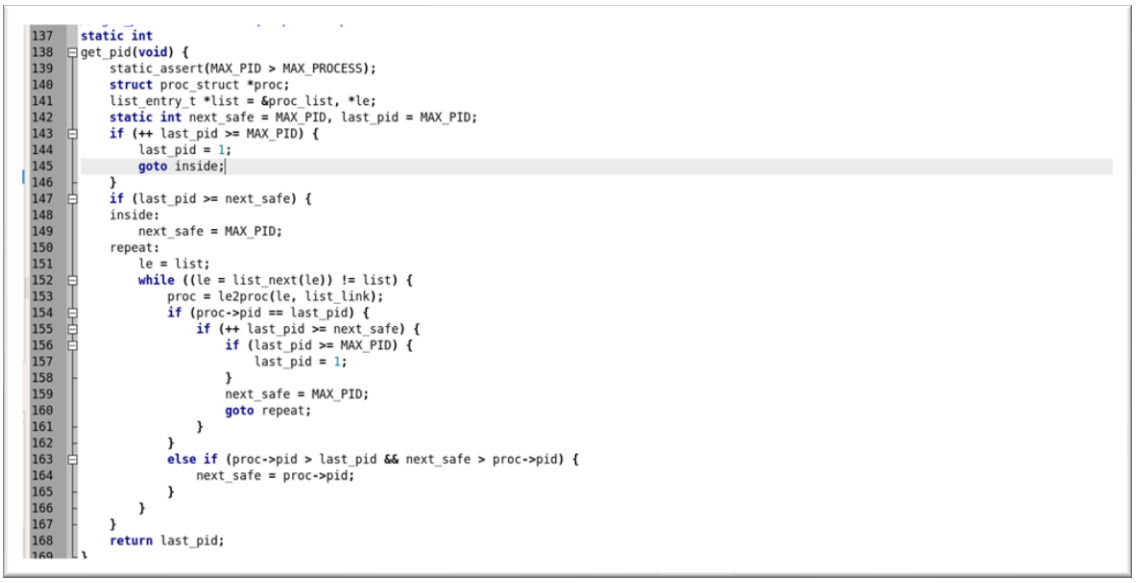


图 7　练习 2 运行成功截图

2、ucore 能做到给每个 fork 线程一个新 fork 的线程一个唯一的 id，ucore 通过 get_pid 函数来为新线程分配 pid，所以我们来分析 get_pid 函数。为了分析的方便，将行号也一并截图。



```
137 static int
138 get_pid(void) {
139     static_assert(MAX_PID > MAX_PROCESS);
140     struct proc_struct *proc;
141     list_entry_t *list = &proc_list, *le;
142     static int next_safe = MAX_PID, last_pid = MAX_PID;
143     if (++ last_pid >= MAX_PID) {
144         last_pid = 1;
145         goto inside;
146     }
147     if (last_pid >= next_safe) {
148     inside:
149         next_safe = MAX_PID;
150     repeat:
151         le = list;
152         while ((le = list_next(le)) != list) {
153             proc = le2proc(le, list_link);
154             if (proc->pid == last_pid) {
155                 if (++ last_pid >= next_safe) {
156                     if (last_pid >= MAX_PID) {
157                         last_pid = 1;
158                     }
159                     next_safe = MAX_PID;
160                     goto repeat;
161                 }
162             }
163             else if (proc->pid > last_pid && next_safe > proc->pid) {
164                 next_safe = proc->pid;
165             }
166         }
167     }
168     return last_pid;
169 }
```

图 8　get_pid 函数

该函数中有两个静态局部变量 last_pid 和 next_safe，通过这个名字我们可以想到这两个变量之间的值应该是可以作为新进程 pid 的值，即已存在的进程的 pid 不在其中。我们可以发现，143、147、155 和 163 行的判断语句都是为了保证 last_pid<next_safe，而 154 和 163 的判断语句是为了保证 last_pid 与 next_safe 之间一定没有当前已经存在的进程的 pid，此时选择 last_pid+1 作为新进程的 pid（这步操作在 143 和 155 判断的时候已经进行了）。理解这个函数需要明白：这个函数设计的目的就是维护一个未曾分配的 pid 值区间，由于静态局部变量只初始化一次，所以每一次调用这个函数 last_pid 和 next_safe 的值都能保留，这样下一次调用就能利用上一次的两个变量值，大大提高了时间效率，与暴力穷举的方法形成鲜明对比。

练习 3：

1、练习 3 主要是理解 proc_run 函数和它所调用的函数如何完成进程切换的工

作，首先我们截图 proc_run 函数如下。

```
// 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);
    }
}
```

图 9　proc_run 函数

从图中可以看到，该函数首先将 prev 设置成 current，即当前正在占用 CPU 的内存，将 next 设置为参数传递过来即将要执行的进程；第二步将任务状态段 ts 中特权态 0 下的栈顶指针 esp0 指向 next 进程内核栈的栈顶，即 next->kstack+KSTACKSIZE；第三步将 CR3 寄存器的值设置为 next 进程的页目录起始地址，完成进程间的页表切换；第四步调用 switch_to 切换执行现场，保存前一个进程的执行现场，然后恢复后一个进程的执行现场，最终完成进程的切换。

2、本实验创建并执行了两个进程。一个是 idleproc，最初的内核线程，在完成新的内核线程创建初始化和调度之后，进入“退休”状态；一个是 initproc，被创建并打印“Hello World”的进程。

3、这条语句的作用是保证括号内的代码执行过程中不会被中断打断，是一个原子操作，不难想象，进程切换过程是不允许被打断的，否则系统便会崩溃。