华东师范大学计算机科学与技术学院上机实践报告

课程名称：操作系统	**年级**：2022 级	**上机实践日期**：2024 年 5 月 26 日
指导教师：李东	**姓名**：岳锦鹏	**学号**：10213903403
实验名称：实验五　用户进程管理		

一、 研读理解相关代码后，回答以下问题：

1、 用户进程与内核线程有何不同？为何要引入用户进程？

用户进程所在的段的特权级是 3，而内核进程所在的段的特权级是 0。

引入用户进程是为了确保安全。一些代码和数据只能在内核态访问而不能在用户态访问，例如操作系统进行页面管理和进程管理的代码。如果用户的代码出错，错误地修改了操作系统的代码段或数据段，会造成严重的错误。而引入用户进程后，用户进程在特权级 3 下运行，无法修改内核进程所在段，只要操作系统不崩溃，用户进程出现错误仍不影响操作系统正常运行。

2、 uCore 是如何实现系统调用的？这样的设计有何优点？

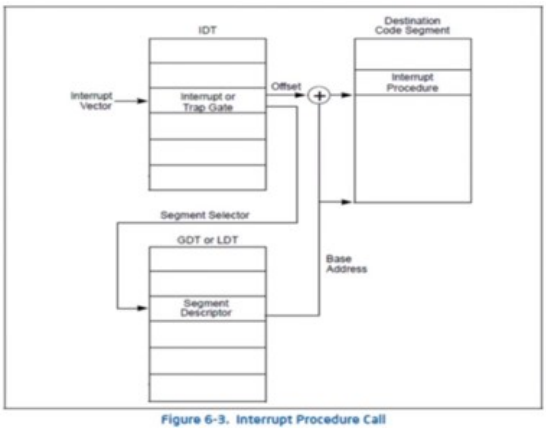

Figure 6-3. Interrupt Procedure Call

在用户态（特权级 3）通过 `int 0x80` 指令，根据中断描述符表找到中断处理程序的段选择子以及偏移量，根据段选择子在段描述符表中找到对应的段描述符，再把段描述符的地址加上偏移量即可找到最终需要执行的代码，并且以特权级 0 执行。其中参数（即中断号）和返回值的传递通过 `eax` 寄存器实现。

这样设计的优点是在用户进程处于用户态的情况下能以内核态执行系统调用，也就是说操作系统可以在切换到内核态之前先进行检查用户是否有权限执行此系统调用，如果没有权限可以提前拦截，避免用户进程错误地或恶意地切换到内核态。

3、 分析 uCore 中用户进程的创建、调度以及执行的过程。(要求说明处理流程，相关的主要函数、它们的功能以及调用关系。)

在 init_main 中加入了一行 int pid = kernel_thread(user_main, NULL, 0); , user_main 的定义如下：

```
// user_main - kernel thread used to exec a user program
static int
user_main(void *arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
#else
    KERNEL_EXECVE(exit);
#endif
    panic("user_main execve failed.\n");
}
```

可以看到这个进程是用来创建用户进程的，其中调用了 kernel_execve ：

```
// kernel_execve - do SYS_exec syscall to exec a user program called by user_main kernel_thread
static int
kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int ret, len = strlen(name);
    asm volatile (
        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL), "0" (SYS_exec), "d" (name), "c" (len), "b" (binary), "D" (size)
        : "memory");
    return ret;
}
```

其中又调用了系统调用子功能 SYS_exec ，它对应的函数为

```
static int
sys_exec(uint32_t arg[]) {
    const char *name = (const char *)arg[0];
    size_t len = (size_t)arg[1];
    unsigned char *binary = (unsigned char *)arg[2];
    size_t size = (size_t)arg[3];
    return do_execve(name, len, binary, size);
}
```

其中又调用了 do_execve

```
// do_execve - call exit_mmap(mm)&put_pgdir(mm) to reclaim memory space of current process
//           - call load_icode to setup new memory space accroding binary prog.
int
do_execve(const char *name, size_t len, unsigned char *binary, size_t size) {
    struct mm_struct *mm = current->mm;
    if (!user_mem_check(mm, (uintptr_t)name, len, 0)) {
        return -E_INVAL;
    }
    if (len > PROC_NAME_LEN) {
        len = PROC_NAME_LEN;
    }

    char local_name[PROC_NAME_LEN + 1];
```

```
    memset(local_name, 0, sizeof(local_name));
    memcpy(local_name, name, len);

    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit;
    }
    set_proc_name(current, local_name);
    return 0;

execve_exit:
    do_exit(ret);
    panic("already exit: %e.\n", ret);
}
```

这里的 do_execve 就释放了当前内核进程的内存空间，并且调用了 load_icode

```
/* load_icode - load the content of binary program(ELF format) as the new content of current process
* @binary:  the memory addr of the content of binary program
* @size:  the size of the content of binary program
*/
static int
load_icode(unsigned char *binary, size_t size) {
    ...
}
```

这里 load_icode 就会创建一个新的页目录，把内核空间中共用的部分复制过去，之后从程序文件（ELF 格式）中读取程序头，根据各个段的地址在内存中分配空间，并相应设置权限，然后从程序文件中的内容复制到内存中，之后将当前进程的页目录设置为新创建的这个页目录。

最后设置当前中断帧的代码段，由于这里的代码段的特权级为 3，所以（通过 eax ）返回之后继续执行代码就是以用户的特权级执行了。