实验三：用户进程

1.实验目的：

- 了解用户进程的关键信息
- 理解内核管理用户进程的方法
- 理解系统调用的过程

2.实验文件：

相比实验二，这次实验有以下改动值得你注意。

kern/process/

proc.c, proc.h

新增：实现进程、线程相关功能主要文件，内核线程 **[proj10]**

修改：增加载入可执行文件、初始化进程、fork 等功能 **[proj10.1]**

修改：支持 wait, kill, exit 系统调用 **[proj10.2]**

修改：brk 调用处理，堆管理 **[proj10.3]**

修改：支持 sleep，利用 timer **[proj10.4]**

修改：线程支持，内存映射支持，共享内存管理支持 **[proj12]**

entry.S

新增：kernel thread 入口函数 **[proj10]**

switch.S

新增：上下文切换，利用堆栈保存、恢复状态 **[proj10]**

kern/init/

init.c

进程系统初始化，在内核初始化完成之后切入 idle 进程 **[proj10]**

修改：调度器初始化（时钟列表） **[proj10.4]**

修改：kern/mm/

memlayout.h

增加用户态相关内存布局的说明与常数 **[proj10]**

pmm.c, pmm.h

增加用户态内存映射函数 **[proj10]**

新增：shmem.c, shmem.h

共享内存页实现

vmm.c, vmm.h

内存映射、VMA 管理等相关函数 **[proj10]**

brk 调用处理 **[proj10.3]**

内存管理系统锁定接口，用户态 copyin/copyout 接口 **[proj12]**

新增：kern/syscall/

syscall.c, syscall.h

系统调用接口 **[proj10.1]**

支持 wait, kill 系统调用 **[proj10.2]**

支持 brk 系统调用 **[proj10.3]**

支持 sleep, gettime 系统调用 **[proj10.4]**

支持 clone, mmap, munmap, shmem 系统调用 **[proj12]**

kern/trap/

trap.c

修改：处理 syscall，处理用户态导致 page fault **[proj10.1]**

kern/schedule/

sched.c, sched.h

修改：支持增加/删除 timer 操作，处理 timer **[proj10.4]**

user/

新增：用户态程序 **[proj10.1]**

新增：用户态测试程序 **[proj10.2]**

libs/

修改：wait 和 kill 系统调用的用户态接口 **[proj10.2]**

修改：brk 调用处理 **[proj10.3]**

修改：支持 sleep, gettime 系统调用 **[proj10.4]**

修改：支持 clone, mmap, munmap, shmem 系统调用 **[proj12]**

修改：针对共享内存，修改内存分配等操作 **[proj12]**

3.实验内容：

3.1 用户态环境的建立

之前的实验，所有的代码都是 Kernel 的一部分，并不能运行用户进程。这样的操作系统没有什么用处。我们需要给 ucore 添加运行用户程序的能力，首先要建立用户程序运行的环境。

用户态环境，最关键的问题是保护(Protection)。简单的说就是不能让用户态进程执行特定的指令，访问特定的数据。所以需要一套机制隔离用户态进程和内核，Intel 用段和特权级来提供这种机制。

刚进入保护模式时，并没有用户态环境：所有的代码均运行在内核态。注意到，*kern/mm/pmm.c* 中的 pmm_init 函数调用了一个函数 gdt_init。gdt_init，如其名，用来配置好 GDT。相比在 bootloader 时期(*boot/bootasm.S*)建立的段，gdt_init 多建立了两个用户段。注：gdt_init 另外还设立 TSS(Task State Segment)段，这个段用于处理不同权限级间的切换中对用户栈地址和内核栈地址的切换。

有了用户段后，用户态环境便建立好了：只要把段选择子(cs，ds 等)配置成选择相应的段，以及低二位为 3，便运行在用户空间了。事实上，操作系统还需要在切换到用户空间前，做更多准备。

3.2 ucore 如何管理进程

我们将执行中的程序抽象成进程。有内核自己的进程，也有执行用户程序的进程。总的来说，管理进程，有以下几个部分：

- 管理进程的关键信息（记录当前运行状态，页表，处理器上下文，使用的内存空间等等），我们现在把这些信息成为进程信息，使用 struct proc_struct 来记录它们。

- 给用户进程分配资源：物理内存，处理器时间，文件等。这里主要涉及分配物理内存。分配处理器时间将在下一个有关调度的实验涉及。文件管理等也会在更后的实验讨论。

- 响应用户进程的各种请求。这在 ucore 通过系统调用来实现。其中包括增长内存空间（用于堆增长），创建子进程等。

- 退出进程，释放用户进程的资源。

下面依次介绍。

3.3 管理进程信息

进程信息用 struct proc_struct 记录，在 *kern/process/proc.h* 中定义如下：

```
struct proc {
    enum proc_state state;          // Process state
    int pid;                        // Process ID
    int runs;                       // Scheduled times
    uintptr_t kstack;               // Kernel stack
    volatile bool need_resched;     // Need to be scheduled?
    struct proc_struct *parent;     // Parent process
    struct mm_struct *mm;           // Memory management info
    struct context context;         // Context
    struct trapframe *tf;           // Trap frame
    uintptr_t cr3;                  // CR3 register
    uint32_t flags;
    char name[PROC_NAME_LEN + 1];   // Process name
    list_entry_t list_link;
    list_entry_t hash_link;
};
```

下面解释一下我们关心的几个项：

mm　　内存管理的信息，包括内存映射列表、页表指针等。mm 里有个很重要的项 pgdir，记录的是该进程使用的一级页表的物理地址。

state　　用户进程所处的状态。

parent　　用户进程的父进程（创建它的进程）。在所有进程中，只有一个进程没有父进程，就是内核创建的第一个进程 initproc。内核根据这个父子关系建立进程的树形结构，用于维护一些特殊的操作，例如确定哪些进程是否可以对另外一些进程进行什么样的操作等等。

context　　进程的上下文，用于进程切换（参见 switch.S）。在 ucore 中，所有的进程在内核中也是相对独立的（例如独立的内核堆栈以及上下文等等）。使用 context 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用 context 进行上下文切换的函数是 switch_to，在 *kern/process/switch.S* 中定义。

tf　　中断帧的指针，总是指向内核栈的某个位置：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。当内核需要跳回用户空间时，需要调整中断帧以调整跳回后各寄存器的值。

kern/process/proc.c 中，kenrel 维护了如下全局变量：

```
static struct proc *current;            // 当前用户态进程
static struct proc *initproc;           // 指向第一个用户态进程
static list_entry_t hash_list[HASH_LIST_SIZE];  // 进程结构的哈希表
list_entry_t proc_list;                 // 所有进程的列表
```

3.4 第一个进程的启动过程

我们已经清楚，内核启动以后会在 kern_init (*kern/init/init.c*)中进行一系列的初始化过程，其中 kern_init 函数会通过调用 proc_init 来创建并加载第一个进程。proc_init 函数执行结束以后，进程并没有直接运行，而是通过调用 scheduler（kern_init -> cpu_idle -> scheduler）来调度程序执行的（你可以在 lab3 中简单的熟悉一下进程调度的过程，它会是下一次实验的主要任务）。

由于目前还没有文件系统，内核生成的过程中，我们会通过链接二进制文件的方式将用户程序直接合并到内核中。注意，此时所谓的用户程序实际上还是内核空间中的一部分，在 lab3 中，我们需要将这部分程序复制到用户内存空间中去。因此， 在创建用户进程的时候，需要特殊的方法获得用户程序的具体位置以及大小等信息。观察工程的编译结果 *kernel.asm* （以及其它 .asm 文件），可以发现，对于名为 xxx 的用户程序，编译器生成了形如 _binary_obj___user_xxx_out_start 以及 _binary_obj___user_xxx_out_size 之类的规范的符号名称：前者表示它的起始地址，后者表示它的大小。在 kernel 引用用户程序的时候，我们可以通过这样的名称来获得用户程序的信息。譬如 hello 用户进程，则有 _binary_obj___user_hello_out_start 和 _binary_obj___user_hello_out_start 表示 hello 的起始地址和大小。

我们通过先执行一个内核进程，再进行一种特殊的 execve 操作，将其转化为一个用户态进程。具体来说，这个内核进程从 proc.c 的 init_main 开始运行，随后调用 kernel_execve，这个函数模拟一次 exec 的 syscall，传入刚才获得的用户程序的开始位置和大小，以及用户进程的名字。当 exec 系统调用完成之后，开始执行用户程序。

一般程序通过 fork 来创建进程。fork 系统调用会执行 do_fork，它通过 alloc_proc 创建一个新进程，随后通过 setup_kstack 初始化进程的内核栈，调用 copy_mm 复制内存管理信息，调用 copy_thread 复制上下文信息，随后将其加入进程列表并返回。这个进程已经有了基本的资源：有内核的结构记录它的信息，有内核栈，有页表。随后，程序通过 exec 系统调用来将自己替换成新的进程。exec 系统调用会执行

do_execve，它会调用 load_icode 载入新进程的代码与数据，替换原程序的数据。这样，新的程序就得到了运行。

3.5 练习 1 获取一个空闲的进程

alloc_proc 函数负责分配并返回一个新的 struct proc_struct 结构，用于存储新建立的进程的数据。它需要对这个结构进行最基本的初始化。这个函数位于 *kern/process/proc.c*。

- 现在由你来完成这个函数。
- 随着系统功能的增多，alloc_proc 需要做的工作也会变多。因此在将其实现从第一个 proj 向之后的复制时，请进行适当的修改。具体区别可以参考注释。

3.6 练习 2 为新创建的进程分配资源

一个进程执行需要很多资源：它需要使用 CPU 的寄存器；CPU 的各个控制寄存器也要恰当设置；它需要一个页表；它需要必要的内存空间存放它的代码和数据；它还需要一个栈；由于它要跳到内核空间去，它还需要一个内核栈。

你需要完成 fork 系统调用。它在 *kern/process/proc.c* 中的 do_fork 中处理。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。一般程序通过 fork 系统调用创建新进程。fork 系统调用的作用是，创建当前进程的一个副本，它们的上下文、代码、数据都一样，但是存储位置不同。如果实现了 copy on write，那么在新进程修改数据之前，它们会共享同一片内存。在 fork 的过程中，需要给新进程分配资源，并且复制原进程的状态。它应该：

- 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
- 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

- 根据注释，完成这个函数。
- 在第一个 proj 之后，这个函数的内容有所变化，为了适应系统的其他部分的变化。请参照注释，在将实现复制到后面的 proj 时进行相应修改。

以上两个函数在各个 proj 中均有出现，内容稍有不同，基本上可以在一个 proj 中完成之后合并到另一个 proj 之中。对于各个 proj 的特别要求详见注释。

3.7 创建进程

参考 gdt_init 函数与 *kern/mm/pmm.c*，可以知道用户进程使用 UTEXT 和 UDATA 段。下面我们以线性地址来讲解用户进程的内存分布：

起始地址	终结地址(闭区间)	用途
0xC0000000 KERNBASE	0xF8000000 KERNTOP	重映射物理内存区
…		
0xB0000000–0x100000(USTACKSIZE)	0xB0000000 USTACKTOP	用户栈

…	…	
0x00800000 UTEXT	0x00800000 + proc size	代码段+数据段+堆
…	…	

当 ucore 需要创建一个进程时，会使用 kernel_thread 函数。kernel_thread 函数会制造一个 trap frame，将其中的 eip 设为 kernel_thread_entry（定义在 *kern/process/entry.S*），ebx 设为参数中的函数指针，edx 设为函数参数，各个段设为内核段，直接调用 do_fork，相当于模拟了一次 syscall。do_fork 会创建一个新进程，试图复制原进程的数据与状态。由于原进程为内核线程，内存管理信息不用复制，而其他的寄存器与执行状态等都得到了复制，也有了新的内核堆栈。之后，当 do_fork 返回时，会返回到制造出来的 trap frame 的 eip 处，也就是 kernel_thread_entry。kernel_thread_entry 仅仅将 edx 中的参数压入栈中，随后调用 ebx 中的函数指针。从此处内核进程即开始运行。

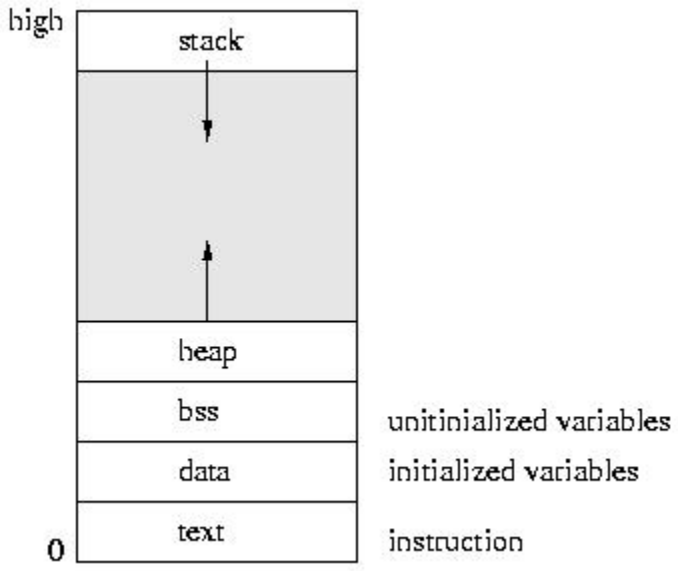
如果要创建用户态进程，ucore 会将 user_main 作为参数。user_main 中用到了 KERNEL_EXECVE 宏，它的最终作用就是获取被调用程序的开始位置与长度，传入 kernel_execve。kernel_execve 会进行一次 exec 系统调用。在 do_execve 中，原进程的内存管理信息会被释放，并且调用 load_icode。在 load_icode 中，会分配新的内存管理信息，elf 程序被分析并载入，用户栈被映射，并且返回时的 trap frame 被重新改写，将各个段设为用户段，将栈设为映射的用户栈顶，eip 设为程序的入口点。这样，当 exec 系统调用返回时，就会自动切换到用户态。

3.7.1 load_icode

load_icode 将用户数据读取到内存中来。由于执行这个函数时，当前 CPU 所用的页表可能和目标进程的页表不同，函数中进行了页表切换。load_icode 同时映射了用户栈。

3.8 用户进程内存的动态分配

用户进程使用的内存空间逻辑上可以分为代码段，数据段，BSS 段，堆和用户栈四个部分



（内核栈对用户是透明的），如图所示：

其中只有堆和用户栈的空间是需要动态增长的。当代码调用 malloc 时，如果当前堆不够，可以通过 brk 系统调用来获得更多空间。brk 接受一个非负整数为参数，返回内核给用户进程分配的内存的逻辑地址。具体可以参照 *user/libs/malloc.c*。brk 在内核中的处理函数是 do_brk，它通过 mm_brk 来增长进程分配的内存。

3.9 内核态到用户态的切换

进程建立后，需要跳到进程空间执行。由于涉及权限切换，这个过程并不只是 ljmp 或者 call 这么简单。

之前提到，在 load_icode 最后，会将 trap frame 的 eip 设置为程序入口点，并且将各个段都设为用户段。在 load_icode 返回到 do_execve 之后，会返回到 trap 函数，之后继续返回到 *kern/trap/trapentry.S* 中的__alltraps 函数。这里会执行 iret，返回到 trap frame 指定的 trap 发生位置，在这里也就是程序的入口点，并且切换到了用户态。

3.10 复制进程

实际上，内核在系统启动后只会设定极少的初始进程，以后的进程都是有进程执行进行系统调用创建的。fork 便是这样的系统调用：它会完全按照按照复制一个和调用进程完全一样的进程。对子进程来说，它好像刚刚从 fork 返回。

为了区分子进程和父进程，他们唯一的区别就是调用 fork 的返回值：

父进程的返回值是子进程的进程号，子进程的返回值是 0。

3.11 延时操作

有时候，进程需要等待一段时间，之后再进行某些操作。这段时间之内，进程进入睡眠状态，调度器不再激活该进程。直到延时时间到，或者睡眠被其他事件所中断，进程才会再次苏醒，继续运行。

在 ucore 中，延时操作通过 sleep 系统调用完成。进程进入 sleep 状态之后，进程状态位中会表明进程正在睡眠，因此调度器就不会调度这个进程运行。同时，进程将一个定时器加入系统的定时器列表。每次时钟中断时，系统会处理定时器列表，对那些已经超时的定时器，系统会清除其睡眠状态，将其唤醒，并且删除这些定时器。

定时器的增加、删除以及处理操作在 *kern/schedule/sched.h* 中实现，而 sleep 系统调用的处理在 *kern/process/proc.c* 中的 do_sleep 函数中实现，其中调用了定时器相关的函数。

练习 3 编写 sleep 系统调用处理函数

sleep 系统调用在 *kern/process/proc.c* 中的 do_sleep 中处理。在这个函数中，需要实现：

- 确定睡眠时间有效
- 初始化一个定时器，设好超时时间
- 关中断
- 把自己设为睡眠状态，并且正在等待延时
- 添加定时器到系统定时器列表
- 开中断
- 进入睡眠
- 从睡眠中苏醒，删除定时器

这个函数在 proj10.4 第一次出现，之后也有，可以合并进去。

其后各个 proj 中的这个函数实现与 proj10.4 中的相同，可以直接复制。

3.12 等待所有子进程执行结束

父进程创建了子进程后，可能希望等它的子进程都结束后自己才继续运行。这里可以使用 wait 系统调用。wait 会等待它的一个子进程退出后，才会返回，并且返回这个子进程的进程号。

参考 *kern/process/proc.c* 中的 do_wait 函数，可以发现，wait 还进行了资源释放工作。这个问题在后面会提及。

3.13 进程生命周期的结束

进程执行完了后，操作系统需要进行清理工作：

- 通知可能在等待它退出的父进程，让父进程继续执行。
- 释放进程占用的各种资源，其中主要是页表和数据等占用的内存资源。

3.13.1 进程主动退出

进程可以通过 exit 系统调用主动退出。用户进程调用 exit 系统调用后，最终会跳到 *kern/process/proc.c* 的 do_exit 函数。通过阅读函数，可以发现，它会首先唤醒可能在等他退出的父进程。把退出的进程的子进程们“过继”给父进程。把自己的状态设置成 ZOMBIE 状态，并且释放它占用的代码，数据和用户栈。

- **问题：** 为什么这时不释放内核栈？

这时，这个进程并没有完全释放：它的基本信息还在进程链表里面，内核栈还没释放，页表也没有完全被释放。这时因为这个它的基本信息还需要被它的父进程使用。父进程调用 wait 得知它退出了，在 do_wait 函数中就会释放这个它的资源。

回到 3.13 提到的 do_wait 函数，可以知道它会释放如下资源：

- 进程的内核栈
- 进程的一级页表
- 进程的基本信息（释放基本信息结构）

3.13.2进程被杀死

进程也可以被别的进程用 kill 系统调用杀死。kill 函数实现很简单：在目标进程的标志位中设置正在退出标记，并唤醒它就可以了。

这样做的原因在于：所有的进程被切换成不执行都在 schedule 函数中。schedule 调用 proc_run，它调用 switch_to 将当前上下文保存，并切换到目标进程的上下文。所以进程返回运行时，必然从 proc_run 中 switch_to 返回的一刻开始。也就是接着执行 proc_run。最后都会回到 trap。trap 中会检查这个进程是否被 kill 了，若是，则执行 exit 退出。

练习 4 编写 kill 系统调用处理函数

kill 系统调用在 *kern/process/proc.c* 中的 do_kill 中处理。在这个函数中，需要实现：

- × 找到目标进程
- × 将其标志位设置为正在退出
- × 如果它的状态处于被中断状态，则唤醒之
- × 如果找不到，返回错误

这个函数在 proj10.2 第一次出现，之后也有，可以合并进去。
其后各个 proj 中的这个函数实现与 proj10.2 中的相同，可以直接复制。

3.14 系统调用

之前一直都有提到系统调用，那么系统调用的全过程是怎样的呢？下面的一系列实验，我们将实现一个简单的系统调用：

`int print(char *str)`

将 `str` 所指的字符串的内容由内核输出到屏幕上去，并得到字符串的长度。

3.14.1 练习 5 为新的系统调用分配系统调用号

每个系统调用都有一个系统调用号。参考 *libs/unistd.h*，可以看到，`fork` 的系统调用号是 2。我们现在需要为 `print` 系统调用定义一个新宏 `SYS_print`，表示它的系统调用号。

- 为 `print` 在 *libs/unistd.h* 分配一个新的系统调用号。

3.14.2 练习 6 用户空间的封装

我们已经知道，系统调用通过 `int 0x80` 指令实现，其中系统调用号需要事先传到 `eax` 中。但实际我们用 `c` 写程序时，并不会这样加入内嵌汇编，而是用一个函数将这些工作都包装起来。

- **练习 6.1** 参照 *user/libs/syscall.h* 已有的系统调用入口函数和上面的 `print` 系统调用原型，添加一个新的系统调用入口。

在头文件有了定义后，还需要这个封装的实现（也在用户空间）。参照，*user/libs/syscall.c, user/libs/ulib.c，user/libs/ulib.h*，可以发现这个封装很简单：每个系统调用之间不同的地方就是指定不同的系统调用号，然后再调用 `syscall()`。

- **练习 6.2** 模仿 `fork`，为 `print` 添加封装的实现。

至此，用户空间的工作就结束了。

3.14.3 练习 7 内核的入口

当执行 `0x20`，经过一系列权限切换的操作，执行到了 `trap` 函数。`trap` 函数会调用 `trap_dispatch` 函数，它会调用 `syscall` 函数(*kern/syscall/syscall.c*)。`syscall` 函数根据系统调用号（也就是内核帧的 `eax` 项，也就是用户进程进入内核态之前 `eax` 的值），来调用相应的系统调用函数。这跟中断向量表类似。而我们的系统调用函数表就是 `syscalls`。熟悉代码中对数组初始化的语法，

- 为 `SYS_print` 在系统调用函数表中添加一项，指向 `sys_print` 函数。

3.14.4 练习 8 处理系统调用

注意到，`sys_print` 函数并没有具体参数。这是因为 `syscall` 函数只负责重定向函数入口，并不知道各个函数对参数的具体要求。而获取参数唯有靠各个处理函数自己了。
每个系统调用都会在用户栈中找自己的需要的参数，这通过直接对参数 `arg` 数组元素进行类型转换所得。这里涉及到操作系统设计的一个安全性问题：地址检查。

系统调用作为用户态进程和内核交流的重要途径，需要做好安全检查：并不是所有进程都是正确的，这其中可能有恶意程序，也可能有编写不善的程序。

拿我们的 `print` 系统调用举例，如果用户进程传一个没有被映射的地址，则会导致内核读一个没有被映射的地址，造成 `Page Fault`：内核导致的 `Page Fault` 则是一个很严重的问题。而如果它传一个内核空间的地址，则内核会把这部分内容打印到屏幕上，造成内核数据泄漏。

这里为了简化，不考虑安全检查的问题。在以后的代码中，会利用 copy_from_user() 从用户段拷数据到内核段。

- 补充 sys_print：取得参数，打印字符串，返回字符串的长度。

至此，便添加了一个新的系统调用。

以上关于系统调用的各个实验必须全都完成，才能通过测试。

练习 5-练习 8 只需要在 proj12 中实现就可以了。

3.15 Challenge A 自动变长的用户栈 (0.01)

之前的实验，我们给用户分配的是固定大小的栈，只有一个页。对于很多应用来说，是不够用的。但是如果每个进程都分配很多页，就会造成浪费：又有很多进程用不了那么大的栈。

- 添加一个处理 Page Fault 的功能，当探测到是因为栈增长造成的 Page Fault 时，增长用户栈。

注意，内核不能无限的满足进程的要求，需要有个上限，这个上限统一为 32 个页：当用户已经使用 32 个页的栈还想要更多的栈时，那就让它崩溃吧。

提醒：

- 当你没有实现 Challenge A，不要更改用户栈的默认大小。

3.16 Challenge B 系统调用的更快的实现 (0.02)

Intel 设计了 sysenter 和 sysexit 两条指令来达到比 int/iret 更快的进行系统调用速度。它们通过用寄存器而不是栈传递数据来达到这一点。由于这种切换方式和 int 这种软中断很不相同，请参阅[2B]获取更多详情。

在ucore中添加这个机制的最简单的办法是在*kern/trap/trapentry.S*中添加处理这种情况的函数：你需要维护好内核的环境和系统调用的参数。同时，在*kern/init/init.c*中，你还要在初始化的时候设置好相应的Model Specific Registers(MSRs)。参考这个文件(http://www.garloff.de/kurt/linux/k6mod.c)来了解如何修改MSRs。可以参考http://en.wikipedia.org/wiki/Machine_state_register以及下面的链接来了解相关资料。

当做内联汇编时，请参照 GCC 内联汇编指南，维护好寄存器的值，不要让内联汇编影响到 c 代码执行使用的寄存器。

由于这个方法并不使用内核帧，可能会和有些用到内核帧的系统调用不兼容。

你可能要修改 Makefile 已添加你新建的.c 或.S 文件。

- 实现这个机制，让 sbrk 系统调用使用这个机制。

3.17 检查你的完成情况

在各个 proj 目录，执行

```
$ make grade
```

检查你的完成情况。如果看到得分为 x/y，其中 x,y 相同，则基本上表示你完成了所有的必做部分。由于每个测试用户程序都需要重新编译，时间较长，还有可能出现超时的情况。如果编译超时了，可尝试重新执行 make grade。

3.18 用用户程序测试你的内核

和之前的实验不同，我们现在无法在内核内部测试你的实现是否正确：只有靠用户进程

来测试。在一开始我们能看到本次试验提供了多个用户进程。执行$ make grade 会测试他们全部。

如果需要测试单个进程，譬如 hello，则可以执行：

```
$ make clean
$ make TESTPROG=hello
```

用这些命令可以得到启动某个用户进程的 ucore.img，用 make qemu 即可测试。

如果不定义 TESTPROG，则默认以 hello 为第一个用户程序。

也可以运行

```
$ make run-hello
```

来直接启动系统并运行 hello 程序。

3.18.1 添加新的用户程序

你也可以添加新的用户程序。在 user 中写一个你自己的.c 程序，譬如 myprog.c：参照别的程序，用 umain 来做程序入口，包含必要的头文件，显式地调用 exit 退出即可。用上面的方法可以测试该程序的运行结果。

如果你写了一些有趣的程序，请告诉助教。

4．报告提交要求

在代码目录 lab3 下完成实验。在实践中完成实验中的各个练习。在各个 proj 目录中，执行 make handin 得到 proj*-handin.tar.gz。

在报告中回答所有练习中提出的问题和简要描述如何完成练习，练习中碰到的问题/困难，已经如何解决的等。实验报告文档命名为 lab3-学生 ID.txt 或.pdf，和各个 proj 目录中 make handin 得到的压缩包放在同一个目录下，运行（以 lab3-yourid.txt 为例）：

```
$ tar -zcvf lab3-yourid.tar.gz lab3-yourid.txt proj*-handin.tar.gz
```

会得到 lab3-yourid.tar.gz 的文件。譬如你的学号为 2008011234，则执行：

```
$ tar -zcvf lab3-2008011234.tar.gz lab3-2008011234.txt proj*-handin.tar.gz
```

得到 lab3-2008011234.tar.gz。

最后把这个文件按时提交到网络学堂上。

如果你做了 Challenge，除了要在报告上写明外，还要在网络学堂提交窗口的作业内容一栏注明。