

OS Lab——进程基础实验报告

一、实验目的

- 了解内核线程创建和执行的管理过程
- 了解内核线程的切换和基本调度过程
- 了解第一个用户进程创建过程

二、实验内容

阅读手册回顾理论课上了解过的进程和线程、用户进程、内核进程、用户线程、内核线程的概念和区别。线程可简单理解为特殊的不用拥有资源的轻量级进程，而在 ucore 的调度和执行管理中并没有区分两者，因此本实验中需要区分的就是用户进程和内核线程。**内核线程和用户进程的区别主要有两个**：内核线程只在内核态运行而用户进程会在用户态和内核态交替运行；ucore 内核中的所有内核线程共享一个内核地址空间和其它资源，从属于同一个唯一的内核进程，即 ucore 本身，而用户进程需要维护各自的用户内存空间。本次实验就是要完成 lab4 和 lab5 中关于内核线程和用户进程创建和执行的相应内容。

开始实验之前需要填写已有实验，考虑到如果使用 diff 直接打补丁会把 lab4 比 lab3 中多出来的部分删除，不可取；使用图形化的 Meld 对比 lab3_result 和 lab4，类似下图：



lab4 中修改的文件包括：debug/kdebug.c, mm/default_pmm.c, mm/pmm.h, mm/swap_fifo.c, mm/vmm.h, trap/trap.c

lab5 修改的文件包括：kdebug.c, init.c, default_pmm.c, kmmalloc.c, kmmalloc.h, memlayout.h, pmm.c, swap_fifo.c, vmm.c, vmm.h, grade.sh. 以及 proc.c 中 alloc_proc(), do_fork() 相较 lab4 都有所改变

1、Lab4 内核线程的创建和执行

从 lab4/kern/init/init.c 中的 **kern_init()** 分析，当完成虚拟内存的初始化工作后，就调用 proc_init() 函数，idleproc 内核线程和 initproc 内核线程的创建工作，idleproc 内核线程在 ucore 没有其他内核线程可执行的情况下才会被调用，它的工作就是不停查询看是否有其他内核线程可执行，如果有马上让调度器选择那个内核线程执行。接着就是调用 kernel_thread() 来创建 initproc 内核线程，它的工作就是显示“Hello World”，表明自己存在且能工作。

调度器执行调度，完成进程切换。在 lab4 中，调度点只有一处，即在 cpu_idle() 函数中，如果发现当前进程 (idleproc) 的 need_resched 置为 1 (初始化时就置为 1 了)，则调用 schedule() 函数，完成进程调度和进程切换。进程调度的过程其实比较简单，就是在进程控制块链表中查找到一个“合适”的内核线程，所谓“合适”就是指内核线程处于“PROC_RUNNABLE”状态。在接下来的 switch_to() 函数完成具体的进程切换过程。一旦切换成功，initproc 内核线程就可以通过显示字符串来表明本次实验成功。

● 分配并初始化一个进程控制块

操作系统是以进程为中心设计的，所以其首要任务是为进程建立档案，进程档案用于表示、标识或描述进程，即进程控制块(PCB)这一数据结构。这里需要完成的就是 ucore 的进程控制块 proc_struct 的初始化

首先找到在 kern/process/proc.h 中定义的结构体 **proc_struct**

```
42 struct proc_struct {
43     enum proc_state state;           // Process state
44     int pid;                         // Process ID
45     int runs;                        // the running times of Proces
46     uintptr_t kstack;               // Process kernel stack
47     volatile bool need_resched;      // bool value: need to be rescheduled to release CPU?
48     struct proc_struct *parent;      // the parent process
49     struct mm_struct *mm;            // Process's memory management field
50     struct context context;          // Switch here to run process
51     struct trapframe *tf;           // Trap frame for current interrupt
52     uintptr_t cr3;                  // CR3 register: the base addr of Page Directroy Table(PDT)
53     uint32_t flags;                 // Process flag
54     char name[PROC_NAME_LEN + 1];  // Process name
55     list_entry_t list_link;         // Process link list
56     list_entry_t hash_link;        // Process hash list
57 };
```

[TODO1] 各参数含义如下：

- **state:** 进程在一个生命周期中所处状态

```
10 // process's state in his life cycle
11 enum proc_state {
12     PROC_UNINIT = 0, // uninitialized
13     PROC_SLEEPING,  // sleeping
14     PROC_RUNNABLE,  // runnable(maybe running)
15     PROC_ZOMBIE,    // almost dead, and wait parent proc to reclaim his resource
16 };
```

- **pid:** 进程 ID
- **runs:** 进程分配的时间片，即 CPU 分配给该进程的运行时间
- **kstack:** 记录了分配给该进程的内核栈的位置，对于内核线程就是运行时程序使用的栈对于用户进程用于发生特权级改变时保存被打断的硬件信息。主要作用有两点：
 - ①发生进程切换时，kstack 设置好 tss（保存当前进程的内核栈地址，包括内核态的 ss 和 esp 的值，并将系统当前使用的栈切换成新的内核栈，也就是中断服务程序要使用的栈，而将 ss 和 esp 压到新的内核栈中保存）；
 - ②每个内核线程拥有自己的内核栈，不受 mm 管理，进程退出时能够根据 kstack 的值快速定位栈的位置并进行回收
- **need_resched:** 是否需要调度
- **parent:** 该进程的父进程，即创建它的进程。除了内核创建的第一个内核线程 idleproc，其它进程都有父进程。根据这种父子关系建立进程的树形结构，便于维护一些特殊操作
- **mm:** 内存管理信息的存储，保存页表的物理地址方便进程切换时的页表切换。mm 主要用于实现用户空间的内存管理，内核线程没有用户空间，所以 mm 置为 NULL，cr3=bootcr3，指向 ucore 启动时建立好的栈内核虚拟空间的页表首地址；如果是用户进程用于记录该进程使用的一级页表物理地址的 pgdir 在 proc_struct 中被 cr3 代替
- **context:** 进程上下文(也即环境)，主要保存了前一个进程的现场(各个寄存器的状态)。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 保存寄存器的目的在于在内核态中能够进行上下文之间的切换
- **tf:** 中断帧的指针，指向内核栈某一位置。当进程从用户空间跳到内核空间时记录被中断前的状态；需要跳回用户空间时恢复各寄存器值。此外 ucore 内核允许嵌套中断，所以为了保证嵌套中断发生时 tf 总是能够指向当前的 trapframe，ucore 在内核栈上

维护了 tf 的链。(trapframe 在后面有具体分析)

- **flags:** 进程标志位
- **name:** 进程名
- **list_link:** 进程控制块链表, 便于随时进行插入、删除和查找等进程管理十五
- **hash_link:** 所有进程控制块的哈希表

理解含义后我们需要对成员进行初始化

```
85 static struct proc_struct *
86 alloc_proc(void) {
87     struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
88     if (proc != NULL) {
89         //LAB4:EXERCISE1 YOUR CODE
90         /*
91          * below fields in proc_struct need to be initialized
92          *
93          * enum proc_state state;           // Process state
94          * int pid;                         // Process ID
95          * int runs;                       // the running times of Proces
96          * uintptr_t kstack;               // Process kernel stack
97          * volatile bool need_resched;     // bool value: need to be rescheduled to release CPU?
98          * struct proc_struct *parent;     // the parent process
99          * struct mm_struct *mm;           // Process's memory management field
100         * struct context context;         // Switch here to run process
101         * struct trapframe *tf;           // Trap frame for current interrupt
102         * uintptr_t cr3;                   // CR3 register: the base addr of Page Directroy Table(PDT)
103         * uint32_t flags;                  // Process flag
104         * char name[PROC_NAME_LEN + 1];   // Process name
105         */
106         proc->state=PROC_UNINIT; //状态为未初始化状态
107         proc->pid=-1; //未初始化状态进程id=-1
108         proc->runs=0; //初始化时间片
109         proc->kstack=0; //初始化内存栈地址
110         proc->need_resched=0; //不需要调度
111         proc->parent=NULL;
112         proc->mm=NULL;
113         memset(&(proc->context), 0, sizeof(struct context)); //上下文初始化
114         proc->tf=NULL; //中断帧指针置为空
115         proc->cr3=boot_cr3; //内核页表地址
116         proc->flags=0;
117         memset(proc->name, 0, PROC_NAME_LEN);
118         return proc;
119     }
```

alloc_proc()函数首先通过 **kmalloc()**函数获得 **proc_struct** 结构的一块内存块, 然后将成员变量清零, 某些设置特殊值, 未进行注释的特殊值说明如下:

- **state** 设为初始态 **PROC_UNINIT**, 表示进程诞生但尚未获取资源
- **pid** 在此时设置为 -1, 在完成创建和身份确定以后再设置一个合法值
- 由于内核线程在内核中运行, 所以设置为 **ucore** 内核页表的起始位置 **boot_cr3**。其实与用户进程不同, 所有内核线程都直接从属于 **ucore** 内核, 公用一个映射内核空间的页表, 内核虚地址空间、物理地址空间都相同。

完成进程控制块的建立以后, 就可以创建具体的进程/线程了。由于 **ucore OS** 启动后已经对整个内核内存空间进行了管理, 通过设置页表建立了内核虚拟空间 (即 **boot_cr3** 指向的二级页表描述的空间), 所以内核线程都不需要再建立各自的页表, 只需共享这个内核虚拟空间即可访问整个物理内存。前面提到 **kern_init()**函数通过调用 **proc_init()**函数开始创建内核线程。

● 创建第 0 个内核线程 **idleproc**

从 **kern_init** 启动至今的执行上下文可以看成是 **ucore** 内核中的一个内核线程的上下文, 给他分配一个进程控制块以及对它进行相应初始化, 就可以打造第 0 个内核线程 **idleproc**。在前面初始化的基础上, **proc_init()**函数对 **idleproc** 内核线程进行进一步初始化

```

354     idleproc->pid = 0;
355     idleproc->state = PROC_RUNNABLE;
356     idleproc->kstack = (uintptr_t)bootstack;
357     idleproc->need_resched = 1;
358     set_proc_name(idleproc, "idle");

```

- idleproc 拥有了合法的 ID，表明线程创建成功和身份的确定
- 此时资源分配完毕，状态由创建改为准备，可以调度执行
- 设置内核栈起始位置，ucore 启动时内核栈直接分配给 idleproc 使用
- idleproc 的主要作用在于 ucore 启动时，或者说不存在其它线程时，所以开始执行后就需要调度器切换到其它进程执行来使用 CPU

● 创建第 1 个内核线程 initproc

idleproc 完成的工作很少，需要创建其它进程来完成工作，于是调用 kernel_thread() 创建内核线程 init_main，在 lab4 中该线程的作用只是输出一些字符串，后续用于创建其它内核线程或用户进程。

创建一个内核线程需要分配和设置好很多资源。initproc 创建过程与 idleproc 不同，通过 kernel_thread()->do_fork() 创建。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。完成在 kern/process/proc.c 中的 do_fork 函数中的处理过程。

首先看 kernel_thread:

```

218 // kernel_thread - create a kernel thread using "fn" function
219 // NOTE: the contents of temp trapframe tf will be copied to
220 //       proc->tf in do_fork-->copy_thread function
221 int
222 kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
223     struct trapframe tf;
224     memset(&tf, 0, sizeof(struct trapframe));
225     tf.tf_cs = KERNEL_CS;
226     tf.tf_ds = tf.tf_es = tf.tf_ss = KERNEL_DS;
227     tf.tf_regs.reg_ebx = (uint32_t)fn;
228     tf.tf_regs.reg_edx = (uint32_t)arg;
229     tf.tf_eip = (uint32_t)kernel_thread_entry;
230     return do_fork(clone_flags | CLONE_VM, 0, &tf);
231 }

```

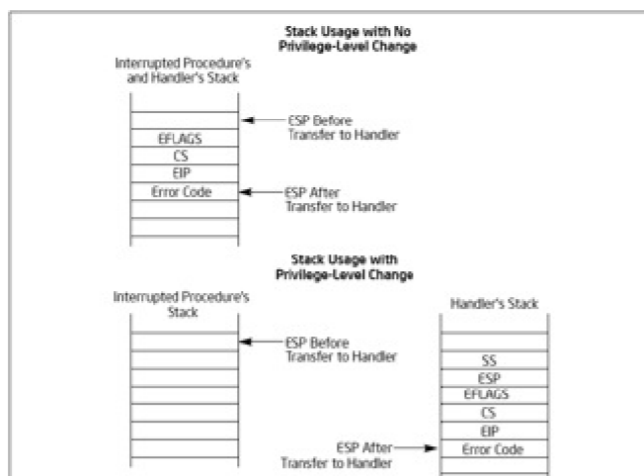
[TODO2] 注意到结构类型 trapframe (lab1 中用于保存当前被打断程序的结构)，作用如下：当 CPU 接收到中断信号以后需要完成一系列工作 ()：

- 硬件中断处理过程1（起始）：从CPU收到中断事件后，打断当前程序或任务的执行，根据某种机制跳转到中断服务例程去执行的过程。其具体流程如下：
 - CPU在执行完当前程序的每一条指令后，都会去确认在执行刚才的指令过程中中断控制器（如：8259A）是否发送中断请求过来，如果有那么CPU就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量；
 - CPU根据得到的中断向量（以此为索引）到IDT中找到该向量对应的中断描述符，中断描述符里保存着中断服务例程的段选择子；
 - CPU使用IDT查到的中断服务例程的段选择子从GDT中取得相应的段描述符，段描述符里保存了中断服务例程的段基址和属性信息，此时CPU就得到了中断服务例程的起始地址，并跳转到该地址；

- CPU会根据CPL和中断服务例程的段描述符的DPL信息确认是否发生了特权级的转换。比如当前程序正运行在用户态，而中断程序是运行在内核态的，则意味着发生了特权级的转换，这时CPU会从当前程序的TSS信息（该信息在内存中的起始地址存在TR寄存器中）里取得该程序的内核栈地址，即包括内核态的ss和esp的值，并立即将系统当前使用的栈切换成新的内核栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将当前程序使用的用户态的ss和esp压到新的内核栈中保存起来；
- CPU需要开始保存当前被打断的程序的现场（即一些寄存器的值），以便于将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息，即依次压入当前被打断程序使用的eflags，cs，eip，errorCode（如果有错误码的异常）信息；
- CPU利用中断服务例程的段描述符将其第一条指令的地址加载到cs和eip寄存器中，开始执行中断服务例程。这意味着先前的程序被暂停执行，中断服务程序正式开始工作。

其中 4、5 步一个重要功能就是向内核栈中压入各种寄存器，既可以起到保护现场的作用，又能让 ISR（中断服务程序）知道中断的各种信息

下图显示了给出相同特权级和不同特权级情况下中断产生后的堆栈变化示意图



除了 CPU 要压入的各种寄存器，我们还需要压入其他一些寄存器用于保存现场和提供给 ISR 中断信息。在 ucore 中，我们使用结构体 **trapframe** 来将保存的寄存器传给 ISR。下面来看 trapframe (kern/trap/trap.c)：

```
50 /* registers as pushed by pushal */
51 struct pushregs {
52     uint32_t reg edi;
53     uint32_t reg esi;
54     uint32_t reg ebp;
55     uint32_t reg oesp;           /* Useless */
56     uint32_t reg ebx;
57     uint32_t reg edx;
58     uint32_t reg ecx;
59     uint32_t reg eax;
60 };
61
62 struct trapframe {
63     struct pushregs tf_regs;
```



```

64  uint16_t tf_gs;
65  uint16_t tf_padding0;
66  uint16_t tf_fs;
67  uint16_t tf_padding1;
68  uint16_t tf_es;
69  uint16_t tf_padding2;
70  uint16_t tf_ds;
71  uint16_t tf_padding3;
72  uint32_t tf_trapno;

73  /* below here defined by x86 hardware */
74  uint32_t tf_err;
75  uintptr_t tf_eip;
76  uint16_t tf_cs;
77  uint16_t tf_padding4;
78  uint32_t tf_eflags;
79  /* below here only when crossing rings, such as from user to kernel */
80  uintptr_t tf_esp;
81  uint16_t tf_ss;
82  uint16_t tf_padding5;
83 } __attribute__((packed));

```

其中 pushregs 中的寄存器都是 pushal 中需要压入栈的所有寄存器。有了这个数据结构后，我们就可以在中断后获取中断的信息，并将它传给 ISR，ISR 会根据传入的 trapframe 来进行相应的操作。

```

proc.c x  proc.h x  trap.h x  trapentry.S x  vectors.S x
12  pushl $1
13  jmp  _alltraps
14  .globl vector2
15  vector2:
16  pushl $0
17  pushl $2
18  jmp  _alltraps

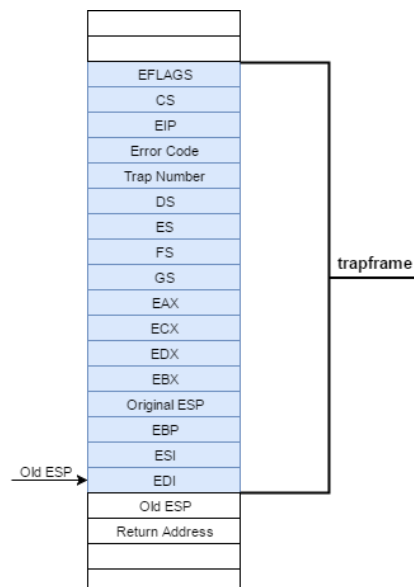
```

```

proc.c x  proc.h x  trap.h x  trapentry.S x  vectors.S x  trap.c x
1 #include <memlayout.h>
2
3 # vectors.S sends all traps here.
4 .text
5 .globl _alltraps
6 _alltraps:
7  # push registers to build a trap frame
8  # therefore make the stack look like a struct trapframe
9  pushl %ds
10 pushl %es
11 pushl %fs
12 pushl %gs
13 pushal
14
15 # load GD_KDATA into %ds and %es to set up data segments for kernel
16 movl $GD_KDATA, %eax
17 movw %ax, %ds
18 movw %ax, %es
19
20 # push %esp to pass a pointer to the trapframe as an argument to trap()
21 pushl %esp
22
23 # call trap(tf), where tf=%esp
24 call trap
25
26 # pop the pushed stack pointer
27 popl %esp
28
29 # return falls through to trapret...

```

在第 6 步时 CPU 首先压入 0 和 2，0 是 error code(对于没有 error code 的中断，ISR 会压入 0 作为 error code；如果中断有 error code，这里就不会压入 0，2 是**中断向量号**)。在压入 error code 和中断向量号后，CPU 跳到 **__alltraps**，__alltraps 会将所有中断需要保存的寄存器存到内核栈，然后将此时栈顶的地址(\$esp)作为参数传给 trap()，trap()会将此时栈中压入的各种寄存器整体当成 trapframe 来处理。trap()会根据 trapframe 中的内容，对中断进行相应的处理。完成对 trapframe 赋值后内核栈情况为：



当 trap()运行结束后，我们需要将寄存器恢复到中断前的状态。在这里，我们只需要将内核栈中的内容分别弹出，并保存到相应的寄存器即可。

回到 kernel_thread()，采用局部变量 tf 来放置保存内核线程的临时中断帧，给中断帧分配空间后构造新进程的中断帧，首先进行清零初始化，然后由于 initproc 内核线程在内核空间中执行，设置中断帧代码段和数据段为内核空间的段，tf.tf_eip 指出 initproc 开始执行的位置 kernel_thread_entry，也即 fn 函数
最后把中断帧的指针传递给 do_fork()函数完成具体内核线程的创建工作，而 do_fork()会调用 copy_thread 来在新创建的进程内核栈上给进程的中断帧分配一块空间

do_fork()函数实现主要分为如下 7 步：

1. 分配并初始化进程控制块（alloc_proc函数）；
2. 分配并初始化内核栈（setup_stack函数）；
3. 根据clone_flag标志复制或共享进程内存管理结构（copy_mm函数）；
4. 设置进程在内核（将来也包括用户态）正常运行和调度所需的中断帧和执行上下文（copy_thread函数）；
5. 把设置好的进程控制块放入hash_list和proc_list两个全局进程链表中；
6. 自此，进程已经准备好执行了，把进程状态设置为“就绪”态；
7. 设置返回码为子进程的id号。

[TODO3] 使用提供的函数和宏定义补全 do_fork()函数

```
392 // 1. call alloc_proc to allocate a proc_struct
393 if ((proc = alloc_proc()) == NULL) {
394     cprintf("allocate failed!");
395     goto fork_out;
396 }
397 proc->parent = current;
398 assert(current->wait_state == 0);
399
400 // 2. call setup_kstack to allocate a kernel stack for child process
401 if (setup_kstack(proc) != 0) {
402     cprintf("allocate failed!");
403     goto bad_fork_cleanup_proc;
404 }
405 // 3. call copy_mm to dup OR share mm according clone_flag
406 if (copy_mm(clone_flags, proc) != 0) {
407     cprintf("copy failed!");
408     goto bad_fork_cleanup_kstack;
409 }
410 // 4. call copy_thread to setup tf & context in proc_struct
411 copy_thread(proc, stack, tf);
412 // 5. insert proc_struct into hash_list && proc_list
413 bool intr_flag;
414 local_intr_save(intr_flag);
415 {
416     proc->pid = get_pid();
417     hash_proc(proc);
418     set_links(proc);
419 }
420 local_intr_restore(intr_flag);
421 // 6. call wakeup_proc to make the new child process RUNNABLE
422 wakeup_proc(proc);
423 // 7. set ret vaule using child proc's pid
424 ret = proc->pid;
```

● 调度并执行内核线程 initproc

ucore 在 lab4 中只实现了一个简单的 FIFO 调度器，核心即 **schedule()**函数

```
13 void
14 schedule(void) {
15     bool intr_flag;
16     list_entry_t *le, *last;
17     struct proc_struct *next = NULL;
18     local_intr_save(intr_flag);
19     {
20         current->need_resched = 0;
21         last = (current == idleproc) ? &proc_list : &(current->list_link);
22         le = last;
23         do {
24             if ((le = list_next(le)) != &proc_list) {
25                 next = le2proc(le, list_link);
26                 if (next->state == PROC_RUNNABLE) {
27                     break;
28                 }
29             }
30         } while (le != last);
31         if (next == NULL || next->state != PROC_RUNNABLE) {
32             next = idleproc;
33         }
34         next->runs ++;
35         if (next != current) {
36             proc_run(next);
37         }
38     }
39     local_intr_restore(intr_flag);
40 }
```


- 1) 设置当前内核线程 `current->need_resched` 为 0;
- 2) 在 `proc_list` 队列中查找下一个处于就绪态的线程或进程 `next`;
- 3) 找到这样的进程后, 就调用 `proc_run` 函数, 保存当前进程 `current` 的执行现场(进程上下文), 恢复新进程的执行现场, 完成进程切换。

因为当前只有两个内核线程, 所以 `schedule()`函数通过查找 `proc_list` 进程队列, 只有处于就绪状态的 `initproc` 内核线程

进入 `proc_run()`函数

```

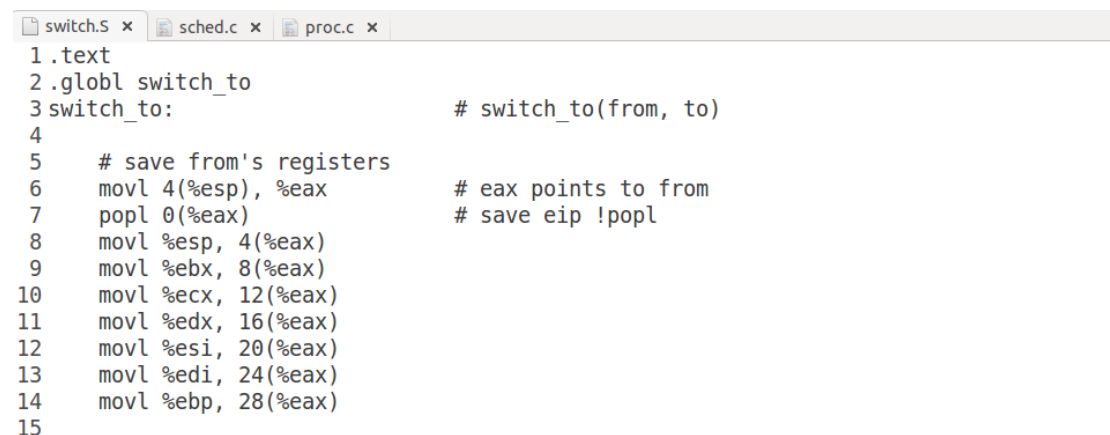
159 // proc_run - make process "proc" running on cpu
160 // NOTE: before call switch_to, should load base addr of "proc"'s new PDT
161 void
162 proc_run(struct proc_struct *proc) {
163     if (proc != current) {
164         bool intr_flag;
165         struct proc_struct *prev = current, *next = proc;
166         local_intr_save(intr_flag);
167         {
168             current = proc;
169             load_esp0(next->kstack + KSTACKSIZE);
170             lcr3(next->cr3);
171             switch_to(&(prev->context), &(next->context));
172         }
173         local_intr_restore(intr_flag);
174     }
175 }

```

分析代码可知:

- 1) 让 `current` 指向 `next` 内核线程 `initproc`;
- 2) 设置任务状态段 `ts` 中特权态 0 下的栈顶指针 `esp0` 为 `next` 内核线程 `initproc` 的内核栈的栈顶, 即 `next->kstack + KSTACKSIZE` ;
- 3) 设置 `CR3` 寄存器的值为 `next` 内核线程 `initproc` 的页目录表起始地址 `next->cr3`, 这实际上是完成进程间的页表切换;
- 4) 由 `switch_to` 函数完成具体的两个线程的执行现场切换, 即切换各个寄存器, 当 `switch_to` 函数执行完“ret”指令后, 就切换到 `initproc` 执行了。

接下来分析 `switch_to()`函数:



```

switch.S x sched.c x proc.c x
1 .text
2 .globl switch_to
3 switch_to:                                # switch_to(from, to)
4
5     # save from's registers
6     movl 4(%esp), %eax                    # eax points to from
7     popl 0(%eax)                          # save eip !popl
8     movl %esp, 4(%eax)
9     movl %ebx, 8(%eax)
10    movl %ecx, 12(%eax)
11    movl %edx, 16(%eax)
12    movl %esi, 20(%eax)
13    movl %edi, 24(%eax)
14    movl %ebp, 28(%eax)
15

```

```

16     # restore to's registers
17     movl 4(%esp), %eax          # not 8(%esp): popped return address already
18                                 # eax now points to to
19     movl 28(%eax), %ebp
20     movl 24(%eax), %edi
21     movl 20(%eax), %esi
22     movl 16(%eax), %edx
23     movl 12(%eax), %ecx
24     movl 8(%eax), %ebx
25     movl 4(%eax), %esp
26
27     pushl 0(%eax)              # push eip
28
29     ret

```

首先保存前一个进程的执行现场，前两条汇编指令保存在了进程在返回 switch_to() 函数后的指令地址到 context.eip 中；保存前一个寄存器的其它 7 个寄存器到 oncontext 相应成员变量中。然后恢复下一个进程的执行现场，即保存的你过程，从 context 高地址成员变量 ebp 开始把相关值赋给相应寄存器。最后把 context 中保存的下一个进程要执行的指令地址 context.eip 放到栈顶，ret 指令把栈顶赋给 eip，切换完成。

[问题回答]

？ 请分析通过 kernel_thread()->do_fork() 创建的 initproc 和仅适用了 init_proc() 创建的 idleproc 的区别

通过上面的过程分析，可以看到两个内核线程的

创建过程不同：0 号内核线程代表 ucore os 来完成一系列管理工作，包括后面 1 号内核线程的创建和执行工作。创建过程只是进行了简单的控制块初始化工作，包括 id, state, kstack, nees_resched, name 的设置。有了 0 号以后才可以创建 1 号线程，后者初始化比较复杂，需要做好各种准备，包括中断帧 trapframe、上下文 context 等来保证进程间切换顺利进行，前面 alloc_init() 函数只是找到了一小块内存来存储信息，而 do_fork() 函数的作用才是给内核线程实际分配资源。

作用不同：idleproc 的作用只是 ucore 启动时，或者说不存在其它线程时，所以开始执行后就需要调度器切换到其它进程执行来使用 CPU；然后是在此基础上创建 initproc，后续的各项工都交给 initproc 来实现，两者目的不同。

？ 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。
是。在分配 PID 时会设置一个保护锁，暂时不允许中断，以保证唯一分配

？ 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？
两个内核线程：idleproc 和 initproc
- 语句 local_intr_save(intr_flag);...local_intr_restore(intr_flag); 在这里有何作用？
保护进程切换不会被中断，避免进程切换时其它进程再进行调度

编译并运行代码 make qemu，输出 hello world

```

[~/moocos/ucore_lab/labcodes/lab4]
moocos-> make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xc010002a (phys)
  etext 0xc010b216 (phys)
  edata 0xc0128a90 (phys)
  end 0xc012bc18 (phys)
Kernel executable memory footprint: 175KB
ebp:0xc0127f38 eip:0xc0101e6a args:0x00010094 0x00000000 0xc0127f68 0xc01000d3
  kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0127f48 eip:0xc0102159 args:0x00000000 0x00000000 0x00000000 0xc0127fb8
  kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0127f68 eip:0xc01000d3 args:0x00000000 0xc0127f90 0xffff0000 0xc0127f94
  kern/init/init.c:57: grade_backtrace2+33
ebp:0xc0127f88 eip:0xc01000fc args:0x00000000 0xffff0000 0xc0127fb4 0x0000002a
  kern/init/init.c:62: grade_backtrace1+38
ebp:0xc0127fa8 eip:0xc010011a args:0x00000000 0xc010002a 0xffff0000 0x0000001d
  kern/init/init.c:67: grade_backtrace0+23
ebp:0xc0127fc8 eip:0xc010013f args:0xc010b23c 0xc010b220 0x00003188 0x00000000
  kern/init/init.c:72: grade_backtrace+34
ebp:0xc0127ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
  kern/init/init.c:32: kern_init+84
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfdf], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [ffffc000, 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 31950, total 31950
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 vaddr 0x1000

```

2、Lab5 用户进程的创建和执行

首先阅读说明了解了**用户进程管理**和之前 lab4 **内核线程管理**的关系和区别。在内存管理部分增加了用户态虚拟内存的管理, 对页表的内容进行扩展, 能够把部分物理内存映射为用户态虚拟内存, CPU 在用户态下执行可以访问本进程页表描述的用户态虚拟内存, 而不能访问内核态虚拟内存; 并且由于不同进程拥有各自的页表, 所以即使虚拟地址相同相互之间也无法访问。在进程管理方面主要是进程控制块 (ucore 中进程控制块与之前的线程控制块相同) 中与内存管理相关的部分。

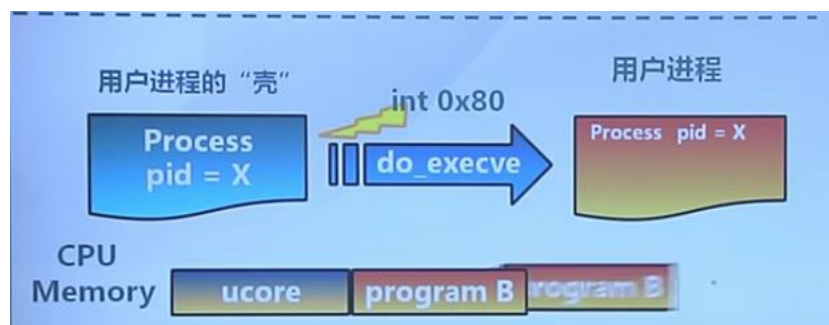
下图即为用户进程的内存虚拟空间，包括两部分：**用户态虚拟地址空间**，虽然虚拟地址范围一样，但是映射到不同且没有交集的物理内存空间；**核心态虚拟地址空间**与之前内核线程一样，是所有用户进程共享的内核虚拟地址空间，映射到同样的物理内存空间中。

```

28 /* *
29 * Virtual memory map:
30 *
31 *
32 * 4G -----> +-----+
33 * |               |
34 * | Empty Memory (*) |
35 * |               |
36 * +-----+-----+ 0xFB000000
37 * | Cur. Page Table (Kern, RW) | RW/-- PTSIZE
38 * VPT -----> +-----+ 0xFAC00000
39 * | Invalid Memory (*) | --/--
40 * KERNTOP -----> +-----+ 0xF8000000
41 * |               |
42 * | Remapped Physical Memory | RW/-- KMEMSIZE
43 * |               |
44 * +-----+-----+ 0xC0000000
45 * | Invalid Memory (*) | --/--
46 * USERTOP -----> +-----+ 0xB0000000
47 * | User stack |
48 * |               |
49 * |               |
50 * | : |
51 * | : |
52 * | : |
53 * | : |
54 * | ~~~~~ |
55 * |               |
56 * | User Program & Heap |
57 * UTEXT -----> +-----+ 0x00800000
58 * | Invalid Memory (*) | --/--
59 * | - - - - - |
60 * | User STAB Data (optional) |
61 * USERBASE, USTAB -----> +-----+ 0x00200000
62 * | Invalid Memory (*) | --/--
63 * 0 -----> +-----+ 0x00000000

```

- 创建并执行用户进程



lab5 中已经将 ucore 和 program 放到 memory 中，**do_execve()**的作用就是把用户进程创建好，并且把相应程序放到 process 中去执行（类比做寄居蟹的形成原理）。首先把用户态内存空间原有的内容清空，然后加载应用程序到当前进程的新创建的用户态虚拟空间中。**load_icode()**函数主要工作就是给用户进程建立一个能够让其正常运行的用户环境。

[TODO4] 讲义中提到该函数需要完成 7 项工作。需要补充的代码主要是 `proc_struct` 结构中 **tf 结构体的设置**，以便于从内核态切换到用户态以后的执行。当进程从内核空间切换到用户空间时，需要恢复之前保存的各个寄存器的状态，因此只需将代码段寄存器、数据段寄存器等恢复即可。同时程序入口地址也需重新设置并需要打开中断。根据提示，这里的 `tf_cs` 即代码段设置为 `USER_CS`，`tf->tf_ds`，`tf->tf_es`，`tf->tf_ss` 都设置为 `USER_DS`，参考上面的虚拟内存空间分布图，`ts_esp` 设为用户栈栈顶，`eip` 指向 `entry`

```
596 /* LAB5:EXERCISE1 YOUR CODE
597 * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
598 * NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE from kernel. So
599 *         tf_cs should be USER_CS segment (see memlayout.h)
600 *         tf_ds=tf_es=tf_ss should be USER_DS segment
601 *         tf_esp should be the top addr of user stack (USTACKTOP)
602 *         tf_eip should be the entry point of this binary program (elf->e_entry)
603 *         tf_eflags should be set to enable computer to produce Interrupt
604 */
605 tf->tf_cs=USER_CS;//切换至用户态代码段
606 tf->tf_ds=tf->tf_es=tf->tf_ss=USER_DS;//ds,es,ss全部设置为用户态数据段
607 tf->tf_esp=USTACKTOP;//0xB0000000
608 tf->tf_eip=elf->e_entry;//指向程序路口地址
609 tf->tf_eflags=FL_IF;//重设标志位，中断打开
610 ret = 0;
```

● 系统调用实现

在 `kern_init()` 中调用了 `idt_init()` 函数来初始化中断描述符表，并设置一个特定中断号的中断门，专门用于用户进程访问系统调用，建立了特权级为 `DPL_USER` 的特殊中断。

当用户进程执行 `SYSCALL` 后，CPU 会从用户态切换到内核态，保存相关寄存器，并跳转到 `_vectors[T_SYSCALL]` 处开始执行

`SYSCALL` 的执行过程：通过 `INT` 指令发起调用，通过 `IRET` 指令完成调用返回用户态，通过 `trapframe` 保存用户进程的执行现场

[TODO5] 下面分析 `hello` 应用程序通过系统中断完成用户态→内核态→用户态切换的过程

```
1 #include <stdio.h>
2 #include <ulib.h>
3
4 int
5 main(void) {
6     cprintf("Hello world!!.\n");
7     cprintf("I am process %d.\n", getpid());
8     cprintf("hello pass.\n");
9     return 0;
10 }
```

`hello` 应用程序只是输出一些字符串，并通过系统调用 `sys_getpid` 输出代表 `hello` 应用程序执行的用户进程的进程标志 `pid`。

用户进程调用 `getpid` 函数，最终执行到“`INT T_SYSCALL`”指令，CPU 根据中断描述符进入内核态并跳转到 `vector128` 处，开始操作系统的系统调用执行过程。调用，路径：

```
vector128(vectors.S)-->
\_alltraps(trapentry.S)-->trap(trap.c)-->trap\_dispatch(trap.c)--
-->syscall(syscall.c)-->sys\_getpid(syscall.c)-->.....-->\_trapret(trapentry.S)
```

在执行 `trap` 函数之前，需要保存执行系统调用前的现场，即中断帧的赋值。将相关寄存器等内容填入 `trapframe` 后，操作系统开始完成具体的系统调用服务。

```
49 static int
50 sys_getpid(uint32_t arg[]) {
51     return current->pid;
52 }
```

`sys_getpid()` 直接把当前进程的 `pid` 返回。

完成服务后，操作系统按调用关系的路径原路返回到 `_alltraps`，然后根据中断帧内容恢

复现场，也即把 `trapframe` 的一部分内容保存到寄存器中。调整内核堆栈指针到中断帧的 `tf_eip` 处，执行 `IRET` 指令后返回到用户态，最后把 `EIP` 指向 `tf_eip` 就完成整个系统调用，也即用户态→内核态→用户态切换的过程。

三、实验感悟

通过此次实验加深了对于内核线程和用户进程的了解，之前理论课上的认识只停留在文字上。实验过程中出现了不少问题：比如是真的看不懂，这次时间主要花在了阅读手册和代码框架上，两个 lab 的讲义前后看了好多遍，代码也是，有一些很复杂的函数调用和宏定义导致阅读非常困难；一开始拷代码的时候不只是简单地把上一个 Lab 的答案补充到下一个 Lab 里面，而是有一些小改动，需要一个一个文件地对比；然后因为分配的硬盘容量比较小，一直在提醒剩余空间不足，然后在某次操作以后…ubuntu 就黑屏了…只能使用原来的镜像文件新建虚拟机，重新对比补充、写代码…

```
[~/moocos/ucore_lab/labcodes/lab5]
moocos-> make qemu
dd: writing to 'bin/ucore.img': No space left on device
4025+0 records in
4024+0 records out
2060288 bytes (2.1 MB) copied, 0.0174059 s, 118 MB/s
make: *** [bin/ucore.img] Error 1
make: *** Deleting file `bin/ucore.img'
```

然后因为虚拟机连接不上网络，无法下载文件到主机上，所以从 github 上面下载了 `proc.c` 文件进行修改，接下来要赶紧解决硬盘容量不足和网络的问题！

总之，感觉还是有很大收获的！但是虽然完成了实验，但还有一些困惑的地方，比如说内存虚拟空间就不怎么明白，希望在后续实验中能够得到解答。