0. 填写完已有实验,执行结果如下:

```
QEMU
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
 memory: 00002000, [07ffe000, 07fffffff], 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
 I-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  i-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  i-- PTE(00001) fafeb000-fafec000 00001000 -rw
                    ---- END
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check vmm() succeeded.
kernel panic at kern/process/proc.c:353:
    create init_main failed.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

```
发现程序挂在kern/process/proc.c中,正好就是练习1中需要填充的函数对应的文件。
1.
首先查看注释:
  * 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
                                  // CR3 register: the base addr of Page Directroy
       uintptr_t cr3;
Table(PDT)
       uint32_t flags;
                                   // Process flag
       char name[PROC NAME LEN + 1];
                                               // Process name
按照注释的顺序来依次初始化各变量:
由于大多数变量只需要初始化为0,故可调用memset来统一清零,再对个别进行设置。
清零后,首先是设置state,找到定义:
// process's state in his life cycle
enum proc state {
  PROC_UNINIT = 0, // uninitialized
  PROC_SLEEPING, // sleeping
  PROC RUNNABLE, // runnable(maybe running)
  PROC_ZOMBIE,
                  // almost dead, and wait parent proc to reclaim his resource
}:
则此处应为PROC UNINIT;
```

接着是pid设为-1,最后将cr3设为内核页目录表boot_cr3。 全部设置完成后,再执行程序,发现执行结果并无变化(除错误行号)。

2.

按照注释的7步,一步步的翻译,需要注意的是前3步失败后要相应的跳转到对应的位置清除已分配的内存并退出。 可却得到如下结果:

```
● ® QEMU
memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!

    BEGIN

PDE(0e0) c0000000-f8000000 38000000 urw
 I-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
 i-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  !-- PTE(00001) fafeb000-fafec000 00001000 -rw
                     --- END
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
             10000(sectors), 'QEMU HARDDISK'.
262144(sectors), 'QEMU HARDDISK'.
ide 0:
ide 1:
SWAP: manager = fifo swap manager
kernel panic at kern/mm/swap.c:187:
assertion failed: PageProperty(p)
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

调了好久,最后实在是不能理解,然后把lab2中的default_init_memmap、default_alloc_pages、default_free_pages换成标准答案里面的代码就过了,看起来是lab2中的代码有点问题,简直了:

```
📴 🗇 🗇 🛮 make qemu
 kmalloc_init() succeeded!
check_vna_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
 check_pgfault() succeeded!
  check_vmm() succeeded.
ide 0:
ide 1:
 lde 0: 10000(sectors), 'QEMU HARDDISK'.
lde 1: 262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = flfo swap manager
 BEGIN check_swap: count 31986, total 31986
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0-4MB OVER!
setup Page Table Vador 6-4MB UVEKI

set up init env for check_swap begin!

page fault at 0x00001000: K/W [no page found].

page fault at 0x00003000: 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].
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
name fault at 0x00002000.
 page fault at 0x00002000: K/W [no page found].
page rault at 0x00002000: K/W [no page round].
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].
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
write Virt Page d in fifo_check_swap
page fault at 0x00004000; K/W [no page found],
swap_out: 1 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
count is 5, 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:338:
            process exit!!.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

3.

proc_run首先判断当前运行的线程是否是要切换到的线程,是则不用切换了。 然后屏蔽中断,以防止在线程切换过程中被中断而出现问题。

接着,将任务状态栈ts中的内核栈指针esp0设置好,然后将页目录表加载到cr3中,完成页表切换,最后再调用switch_to实现线程上下文的切换,此时所有的切换操作已经准备好,只需要恢复中断即可完成进程切换。

而switch_to所实现的就是,先将原线程执行时的各个寄存器保存到原线程的content的相应位置,然后再将新线程的content中的值恢复到各寄存器中,实现线程上下文的切换。