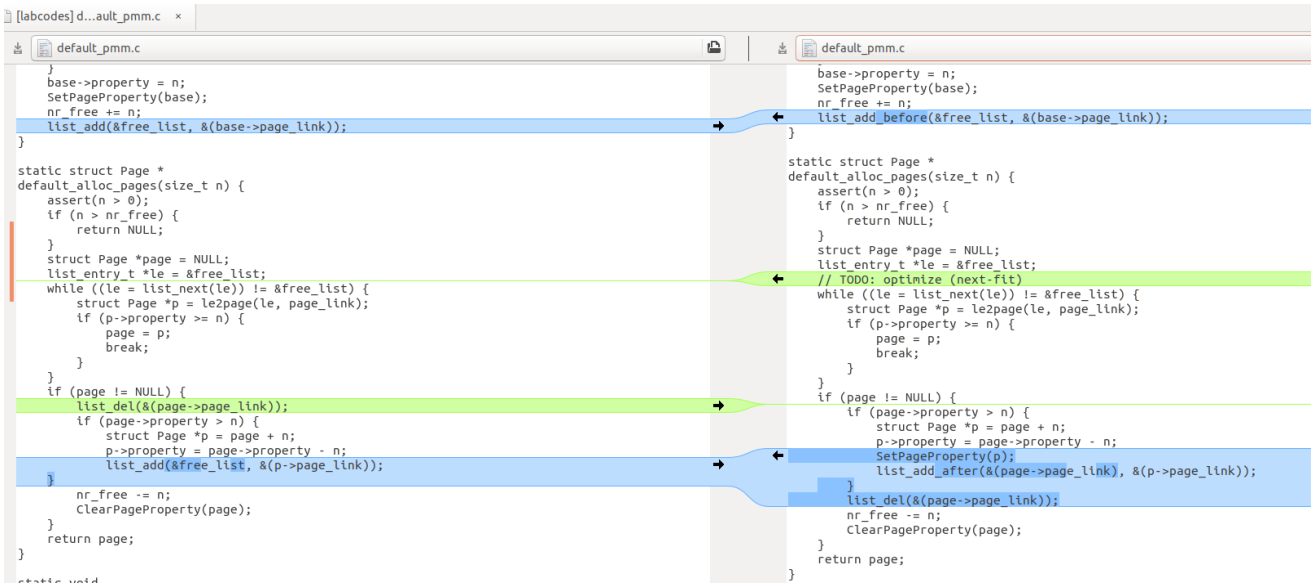
实验四：内核线程管理

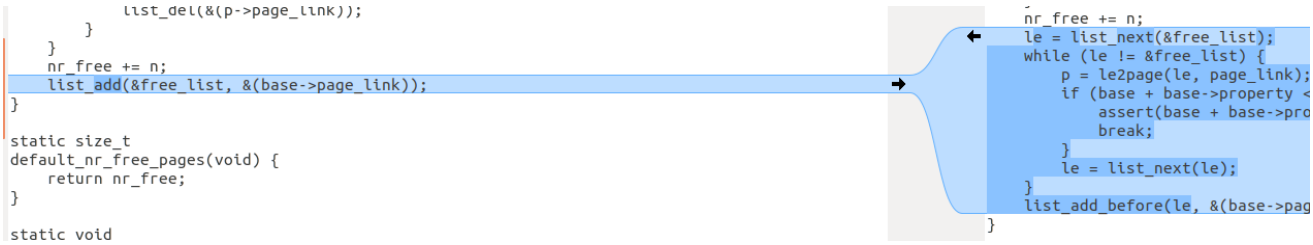
实验中遇到的问题

写完编程部分，执行make qemu之后，报错：assertion failed: (p0 = alloc_page()) == p2 - 1

```
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0123f68 eip:0xc01000d3 args:0x00000000 0xc0123f90 0xffff0000 0xc0123f94
    kern/init/init.c:58: grade_backtrace2+33
ebp:0xc0123f88 eip:0xc01000fc args:0x00000000 0xffff0000 0xc0123fb4 0x0000002a
    kern/init/init.c:63: grade_backtrace1+38
ebp:0xc0123fa8 eip:0xc010011a args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:68: grade_backtrace0+23
ebp:0xc0123fc8 eip:0xc010013f args:0xc0109cdc 0xc0109cc0 0x00003164 0x00000000
    kern/init/init.c:73: grade_backtrace+34
ebp:0xc0123ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:33: 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: 07ee0000, [00100000, 07fdffff], type = 1.
  memory: 00020000, [07fe0000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
kernel panic at kern/mm/default_pmm.c:238:
    assertion failed: (p0 = alloc_page()) == p2 - 1
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

想了一下我在lab4里也没改mm文件里的东西啊，可能是meld操作的时候没处理好？用meld对比了一下lab4和答案的default_pmm.c，不知道是不是没有设置property位的原因.....





修改了mm文件夹里和answer不一样的地方，执行make qemu之后还是报错：assertion failed: !PageReserved(p) && !PageProperty(p)，这个错的原因好像还是因为我用到了一个不能用的页，或者是初始化的时候一些标志位没有设置好，但是我也再找不到哪里写的有问题了，所以make clean，再make qemu之后，结果正确了……可能是最开始生成的bin、obj文件下的文件有问题吧……

```
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:354:
    process exit!!.

stack trackback:
ebp:0xc0330f98 eip:0xc0101f43 args:0xc010b6c1 0xc0330fdc 0x00000162 0xc0330fcc
    kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0330fc8 eip:0xc01018ed args:0xc010d575 0x00000162 0xc010d589 0xc012e044
    kern/debug/panic.c:27: __panic+105
ebp:0xc0330fe8 eip:0xc010a3df args:0x00000000 0xc010d608 0x00000000 0x00000010
    kern/process/proc.c:354: do_exit+33
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

其他的还没有遇见什么问题，只是觉得自己还是 对这些过程不是那么的明晰，需要参考网上的代码才能完成作业。

实验目的

- 了解内核线程创建/执行的管理过程
- 了解内核线程的切换和基本调度过程

实验内容

内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

- 内核线程只运行在内核态
- 用户进程会在在用户态和内核态交替运行
- 所有内核线程共用ucore内核内存空间，不需为每个内核线程维护单独的内存空间
- 而用户进程需要维护各自的用户内存空间

练习0：填写已有实验

经过lab2和lab3之后，这就不是什么事了，使用meld即可。

练习1：分配并初始化一个进程控制块（需要编码）

alloc_proc函数（位于kern/process/proc.c中）负责分配并返回一个新的struct proc_struct结构，用于存储新建立的内核线程的管理信息。ucore需要对这个结构进行最基本的初始化。

> 【提示】在alloc_proc函数的实现中，需要初始化的proc_struct结构中的成员变量至少包括：state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name。

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1 YOUR CODE
        proc->state = PROC_UNINIT; // 进程状态，PROC_UNINIT，表示未初始化；
        proc->pid = -1; // 进程ID，-1，表示未分配；
        proc->runs = 0; // 进程时间片
        proc->kstack = 0; // 进程所使用的内存栈地址
        proc->need_resched = NULL; // 进程是否能被调度
        proc->parent = NULL; // 父进程
        proc->mm = NULL; // 进程所用的虚拟内存
        memset(&(proc->context), 0, sizeof(struct context)); // 进程的上下文
        proc->tf = NULL; // 中断帧指针
        proc->cr3 = boot_cr3; // 页目录表地址 设为 内核页目录表基址
        proc->flags = 0; // 标志位
        memset(&(proc->name), 0, PROC_NAME_LEN); // 进程名
    }
    return proc;
}
```

回答如下问题：

- 请说明proc_struct中`struct context context`和`struct trapframe *tf`成员变量含义和在本实验中的作用？

  context：进程的上下文，用于进程切换（参见switch.S），在上下文切换时保存当前EBX、ECX、EDX、ESI、EDI、ESP、EBP、EIP八个寄存器。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在*kern/process/switch.S*中定义switch_to。

  ```
  //kern/process/proc.h
  struct context {
    uint32_t eip;
    uint32_t esp;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t esi;
    uint32_t edi;
    uint32_t ebp;
  };
  ```

trapframe *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，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，uCore内核允许嵌套中断。因此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，可以参考trap.c::trap函数做进一步的了解。

```
//kern/trap/trap.h
struct trapframe {
  struct pushregs tf_regs;
  uint16_t tf_gs;
  uint16_t tf_padding0;
  uint16_t tf_fs;
  uint16_t tf_padding1;
  uint16_t tf_es;
  uint16_t tf_padding2;
  uint16_t tf_ds;
  uint16_t tf_padding3;
  uint32_t tf_trapno;
  /* below here defined by x86 hardware */
  uint32_t tf_err;
  uintptr_t tf_eip;
  uint16_t tf_cs;
  uint16_t tf_padding4;
  uint32_t tf_eflags;
  /* below here only when crossing rings, such as from user to kernel */
  uintptr_t tf_esp;
  uint16_t tf_ss;
  uint16_t tf_padding5;
} __attribute__((packed));
```

练习2：为新创建的内核线程分配资源

创建一个内核线程需要分配和设置好很多资源。kernel_thread函数通过调用**do_fork**函数完成具体内核线程的创建工作。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。你需要完成在kern/process/proc.c中的do_fork函数中的处理过程。它的大致执行步骤包括：

- 首先调用 `alloc_proc` 来申请一个初始化后的进程控制块；
- 调用 `setup_kstack` 为内核进程（线程）建立栈空间；
- 调用 `copy_mm` 拷贝或者共享内存空间；
- 调用 `copy_thread` 建立trapframe以及上下文；
- 调用 `get_pid()` 为进程分配一个PID；
- 将进程控制块加入哈希表和链表；
- 最后，返回进程的PID。

```
if ((proc = alloc_proc()) == NULL)
    goto fork_out;
if ((ret = setup_kstack(proc)) != 0)
    goto fork_out;
if ((ret = copy_mm(clone_flags, proc)) != 0)
    goto fork_out;
copy_thread(proc, stack, tf);
ret = proc->pid = get_pid();
hash_proc(proc);
list_add(&proc_list, &(proc->list_link));
wakeup_proc(proc);
```

回答如下问题：

- 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。

  线程的PID由`get_pid`函数产生，该函数中包含了两个静态变量`last_pid`以及`next_safe`。`last_pid`变量保存上一次分配的PID，而next_safe和last_pid一起表示一段可以使用的PID取值范围$(last_pid, next_safe)$，同时要求PID的取值范围为$[1, MAX_{PID}]$，`last_pid`和`next_safe`被初始化为`MAX_PID`。每次调用`get_pid`时，除了确定一个可以分配的PID外，还需要确定`next_safe`来实现均摊以此优化时间复杂度，PID的确定过程中会检查所有进程的PID来确保PID是唯一的。

练习3：阅读代码，理解 proc_run 函数和它调用的函数如何完成进程切换的。

`proc_run`的执行过程为：

- 保存IF位并且禁止中断；
- 将current指针指向将要执行的进程；
- 更新TSS中的栈顶指针；
- 加载新的页表；
- 调用switch_to进行上下文切换；
- 当执行proc_run的进程恢复执行之后，需要恢复IF位

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，一个为 idle_proc， 第 0 个内核线程，完成内核中的初始化，调度执行其他进程或线程。另一个为 init_proc，本次实验的内核线程，只用来打印字符串。

- 语句`local_intr_save(intr_flag);....local_intr_restore(intr_flag);`在这里有何作用？

  进行进程切换的时候，需要避免出现中断干扰这个过程，所以需要在上下文切换期间清除IF位屏蔽中断，并且在进程恢复执行后恢复IF位。

```
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);// 关闭中断
        {
            current = proc;// 将当前进程换为 要切换到的进程
            // 设置任务状态段tss中的特权级0下的 esp0 指针为 next 内核线程 的内核栈的栈顶
            load_esp0(next->kstack + KSTACKSIZE);
```

```
            // 重新加载 cr3 寄存器(页目录表基址) 进行进程间的页表切换
            lcr3(next->cr3);
            // 调用 switch_to 进行上下文的保存与切换
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);//恢复IF位
    }
}
```

```
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:354:
    process exit!!.

stack trackback:
ebp:0xc0330f98 eip:0xc0101f43 args:0xc010b6c1 0xc0330fdc 0x00000162 0xc0330fcc
    kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0330fc8 eip:0xc01018ed args:0xc010d575 0x00000162 0xc010d589 0xc012e044
    kern/debug/panic.c:27: __panic+105
ebp:0xc0330fe8 eip:0xc010a3df args:0x00000000 0xc010d608 0x00000000 0x00000010
    kern/process/proc.c:354: do_exit+33
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

扩展练习Challenge：实现支持任意大小的内存分配算法

这不是本实验的内容，其实是上一次实验内存的扩展，但考虑到现在的slab算法比较复杂，有必要实现一个比较简单的任意大小内存分配算法。可参考本实验中的slab如何调用基于页的内存分配算法（注意，不是要你关注slab的具体实现）来实现first-fit/best-fit/worst-fit/buddy等支持任意大小的内存分配算法。。

【注意】下面是相关的Linux实现文档，供参考

SLOB

http://en.wikipedia.org/wiki/SLOB http://lwn.net/Articles/157944/

SLAB

https://www.ibm.com/developerworks/cn/linux/l-linux-slab-allocator/

这里需要借助实验二扩展练习实现的Slub算法，但是我实验而实现的是buddy system，所以在这里，我放弃了。