南京航空航天大学

操作系统实践

总结报告

班级： 1618001

学号： 161840230

姓名： 王可

Proj0

一、 题目简述

1. 添加用户命令(40%)

实现命令pxy Z（xy为学号最后两位），使得屏幕输出OS Lab $学号$: Z 并换行。

2. 添加内核输出语句(10%)

在系统启动输出的信息中，加入自己的学号姓名，具体位置在” cpu: starting 0” 与 “sb : size 1000 nblocks...” 之间。

3. 添加系统调用(50%)

添加用户命令shutdown，由于功能需要调用特权指令outw(0x604,0x2000)，应实现系统调用来完成用户命令，使得输入shutdown后，推出qemu到ssh窗口。

二、 实验过程

1. 添加用户命令

(1) 添加用户命令首先需要定义命令的逻辑，实现一个名为p30.c的文件（文件名与命令名称对应）并通过main函数参数接收Z的内容，首先输出"OS Lab 161840230: "，然后对除命令本身的其他输入依次输出，用空格隔开，最后输出一个换行符。

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc,char* argv[]){
    printf(1,"OS Lab 161840230: ");
    if(argc > 1){
        for(int i = 1; i < argc;++i){
            printf(1,"%s ",argv[i]);
        }
    }
    printf(1,"\n");
    exit();
}

```

(2) 其次，需要将该用户命令注册到用户命令列表中。 对此，需要对Makefile中的 UPROGS量添加一个_p30\进行注册。

(3) 完成后进行make生产，再次打开系统就可以成功执行命令p30。

2. 添加内核输出语句

(1) 通过阅读源码可以发现，"cpu starting"在mpmain(void)函数的第一行被输出，而通过测试，下一句系统输出在scheduler()函数中，或者之后，因此，在这两者之间任意地方增加语句：

```
cprintf("wangke 161840230\n");
```

即可。

3. 添加系统调用

(1) 首先，添加系统调用语句，命名为sdown，在proc.c中进行编写注册，sdown中调用特权指令outw(0x604,0x2000)。

```
int
sdown(void){
  outw(0x604,0x2000);
  return 22;
}
```

(2) 在user.h,defs.h中注册sdown函数，用于后续绑定(会被其他定义处引用)。

(3) 在syscall.h中将SYS_sdown系统命令与命令号22绑定，作为sdown的返回值，且将此命令号在syscall.c中与系统调用sys_sdown进行绑定，并注册sys_sdown命令。

(4) 在sysproc.c文件中定义 sys_sdown函数，

(5) 按照实验一的方式添加用户命令shutdown，并调用系统调用sdown，即可完成关机指令。

(6) 打开系统，输入shutdown，正常退出。

三、 实验结果

1. 添加用户命令

在git bash中输入make qemu指令，启动xv6，输出如下信息：

```
xv6...
cpu0: starting 0
wangke 161840230
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
```

2. 添加内核输出语句

输入p30以及一个含空格的字符串，输出指定结果：OS Lab (学号): Z

```
$ p30 hello world
OS Lab 161840230: hello world
```

3. 添加系统调用

在xv6系统中输入shutdown，成功进行关机操作，退出到ubuntu界面：

```
$ shutdown
wangke@VM-199-186-ubuntu:~/proj0-revise$
```

四、 实验心得

1. 添加用户命令时，起初使用return 0返回出错，考虑可能是返回值没有给良好的处理，所以进trap了，改为exit(0)后解决。
2. 考虑内核输出时，和用户空间输出的差别在于，从用户空间输出时，需要中断调用输出，切换状态，切换上下文，需要更多代价，而从内核态输出，只需要函数调用即可，因此速度更快，对计算机资源的浪费更低。 此外，从内核输出，可以减少用户命令的绑定，使得命令空间更加简洁。
3. 操作系统调试，报错时信息量比较少，修改时应该做版本控制或记录修改处。

Proj1

一、 题目简述

1. 带参数的系统调用(30%)

修改系统调用使得shutdown可以接受一个整形参数，当用户输入shutdown命令后，屏幕上打印一行字并退出，示例如下：

```
$ shutdown 20
Leaving with code 20.
```

2. 理解进程切换(20%)

阅读源码并理解，回答一系列问题：

- CPU在执行 `scheduler()` 时运行在用户态还是内核态？运行在哪个栈上面？
- 当你在命令行敲下 `shutdown` 时，系统会创建一个进程执行shutdown.c中的代码，当CPU执行以下三条指令时 `movl $SYS_shutdown, %eax; int $T_SYSCALL; ret` 时，CPU运行在用户态还是内核态？运行在哪个栈上面？（注意：CPU在执行这三条指令时的状态和栈是一样的）
- 在执行命令 `shutdown` 的过程中，当cpu执行到涉及特权指令的函数 `outw()` 时，CPU运行在用户态还是内核态？运行在哪个栈上面？
- 为何在执行完 `swtch` 函数后，cpu没有像普通函数调用一样返回到 `scheduler` 函数中？

3. 子进程优先的fork(50%)

阅读proc.c中的int fork(void)函数。xv6执行完fork后，父进程先运行，然后子进程运行。

修改这个行为，使得子进程优先运行。实现一个系统调用，以控制是否启用算法。

```
void fork_winner(int winner);
```

回答问题：

- 在父进程优先的情况下，偶尔会有子进程先于父进程打印到屏幕的情况出现；在子进程优先的情况下，偶尔也会有父进程先于子进程打印到屏幕的情况。解释可能的原因。

二、 实验过程

1. 带参数的系统调用

1. 阅读源码得知，shutdown用户函数调用了名为shutdown的系统调用(该函数在usys.S文件中被定义)，在中断trap中，通过syscall调用到sys_shutdown函数，在此函数中，需要实现获取一个参数，并将其输出到屏幕的功能。
2. 仿照sleep函数，使用了argint函数获取int型参数，argint有两个参数
    - 其一为**n**，从外部意义上控制了获取函数第n个变量，而从函数内部是控制了栈地址，取得地址空间为(myproc()->tf->esp) + 4 + 4*n，即栈指针所指位置加4(n+1)个字节，系数是4是因为一个整形数所占存储空间为4个字节，而之所以使用n+1即第一个参数需要向上跨过4个字节是由函数调用时栈的组织结构决定的，根据xv6官方指南Figure2-3中的结构所示，函数栈中，底部为函数返回地址，上面为被调用者参数自左向右依次入栈，即第一个参数在最底部，也就是esp+4的位置，因此，所寻地址为esp+4+4*n.
    - argint 第二个参数即为所获取的数据存放的地址，通过调用fetchint判断地址是否越界后获得数据。
    - 通过这种方法，即完成了目标需求。
3. 此外，考虑到不使用Leaving code的情况，本使用判断输入参数是否为0的方法决定直接退出系统还是先输出信息再退出系统，但由于给出的参数范围为止，即存在Leaving code本身为0的情况，因此在用户函数获取命令行参数时，根据参数数量生成另一系统调用参数ctrl用以控制。实现代码如下：

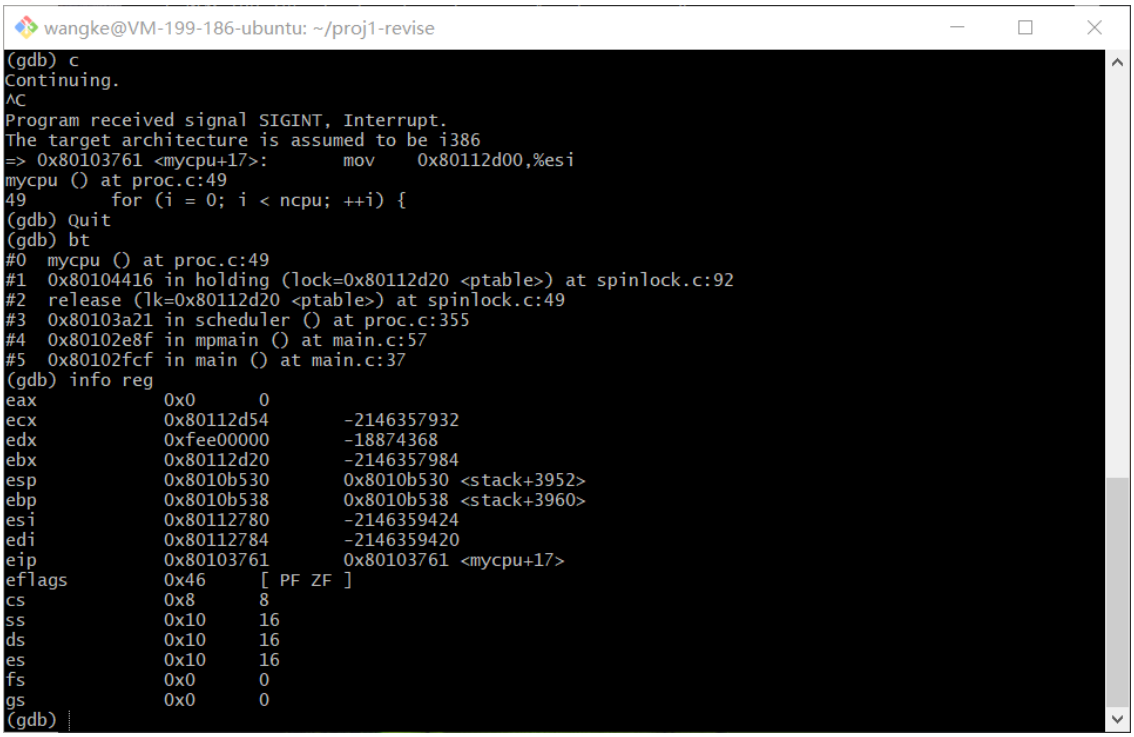
```
// sysproc.c
int
sys_shutdown(void){
  int arg;
  int ctrl;
  // for(int i = 0 ; i < 40;i++){
  //   argint(i,&arg);
  //   cprintf("%d  :  %d\n",i,arg);
  // }
  if(argint(0,&ctrl) < 0){
    return -1;
  }
  if(argint(1,&arg) < 0){
    return -1;
  }  // the function argument is defined in file syscall.c
  if(ctrl == 1){
    cprintf("Leaving with code %d\n",arg);
  }
  outw(0x604, 0x2000);
  return 0;
}

// shutdown.c --main
int
main(int argc, char *argv[])
{
  int arg = 0;
  int ctrl = 0;
  char *number = "";
  if(argc > 1){
    number = argv[1];
    ctrl = 1;          // the variable ctrl is initialized as 0
  }
  while(strlen(number)>0){  // transform the char* argument into int
    arg = arg * 10;
    arg+=number[0]-'0';
    number += 1;
```

```
  }
  shutdown(ctrl,arg);
  exit();
}
```

2. 理解进程切换

阅读源码并理解，回答一系列问题：

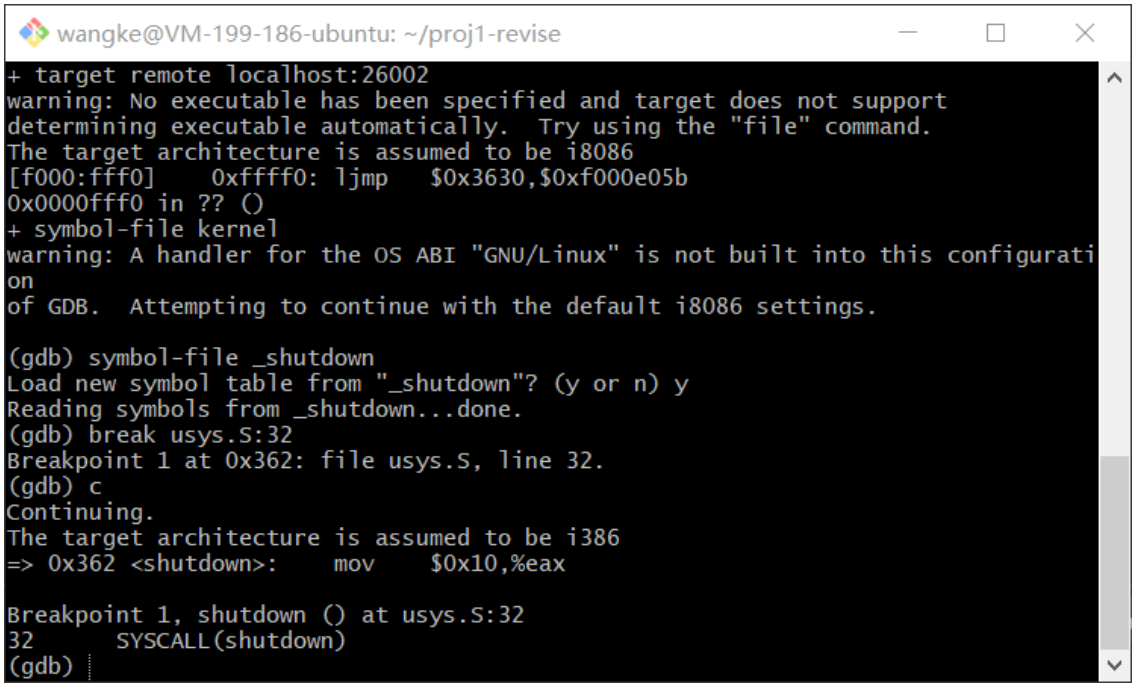
- CPU在执行`scheduler()`时运行在用户态还是内核态？运行在哪个栈上面？
  - 运行时在gdb中使用ctrl+C打断程序执行，进入scheduler()函数，通过bt命令确认后，确实进入了该函数：

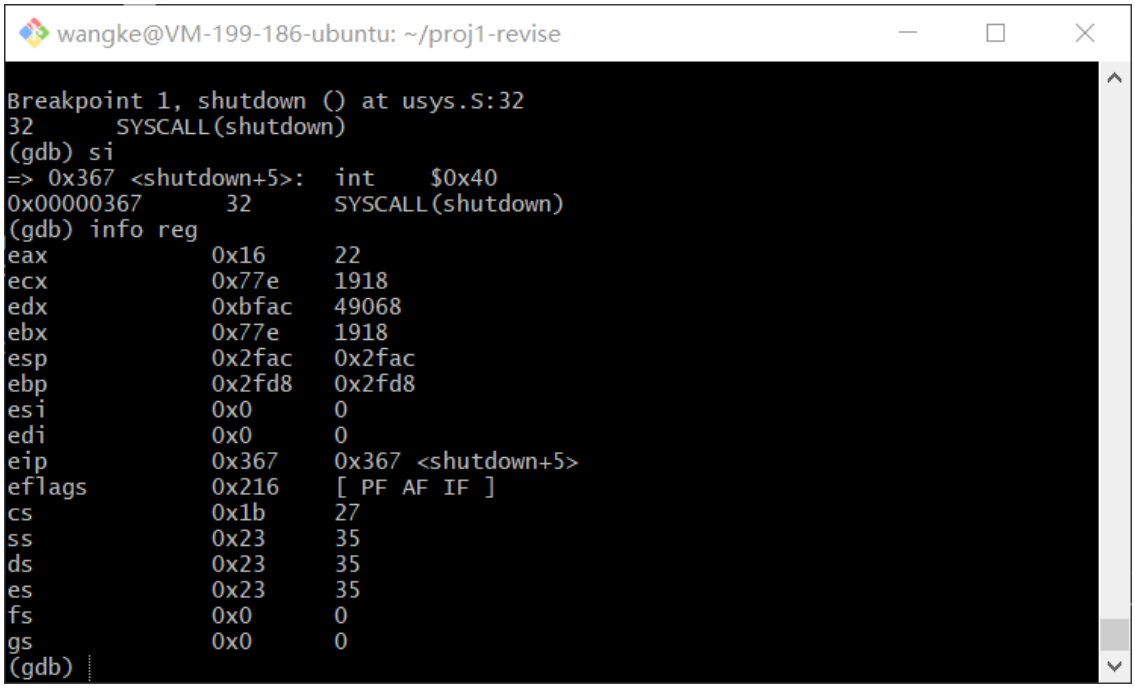


    - 查看寄存器可以发现：cs值为8，即1000，最后两位是00，即处于内核态中，同时esp寄存器中的值为0x8010b530，大于0x8000 0000。由此可见当前栈地址在内核空间，函数scheduler()运行在内核栈中。
- 当你在命令行敲下`shutdown`时，系统会创建一个进程执行shutdown.c中的代码，当CPU执行以下三条指令时`movl $SYS_shutdown, %eax; int $T_SYSCALL; ret`时，CPU运行在用户态还是内核态？运行在哪个栈上面？（注意：CPU在执行这三条指令时的状态和栈是一样的）
  - 使用如下命令加载_shutdown并将断点设置于如题三条指令前：

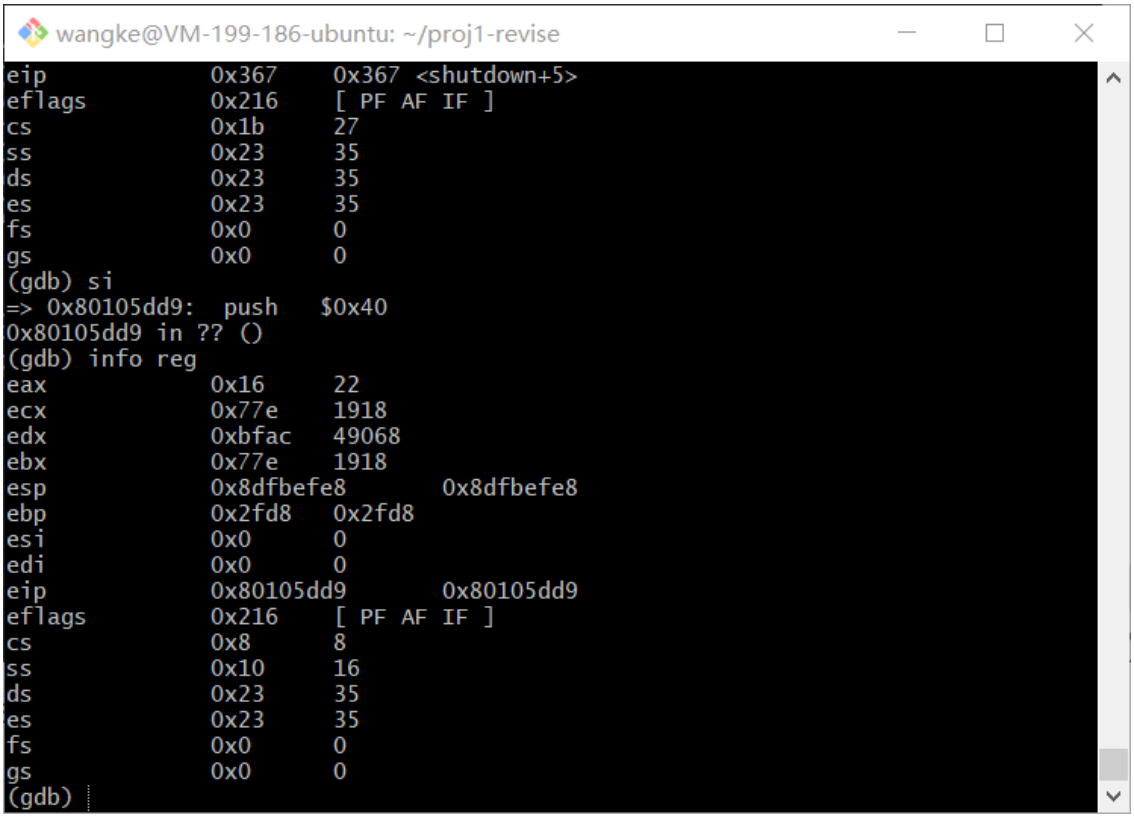
```
symbol-file _shutdown
break usys.S:32  # 需要将断点打到函数中而不是define中
```

    - 多次使用continue进入执行后，输入shutdown命令，进入断点并使用info reg查看寄存器，结果如下：



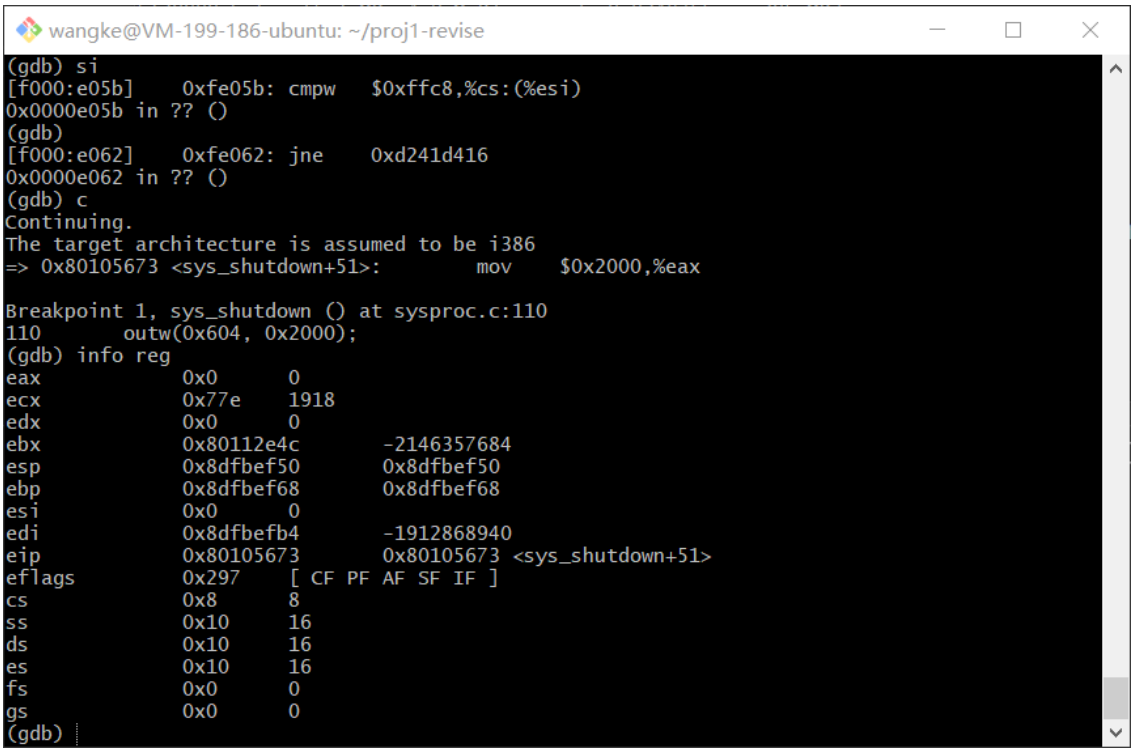


- %eax寄存器值为22，正是syscall.h中设置的系统调用号，断点正确。



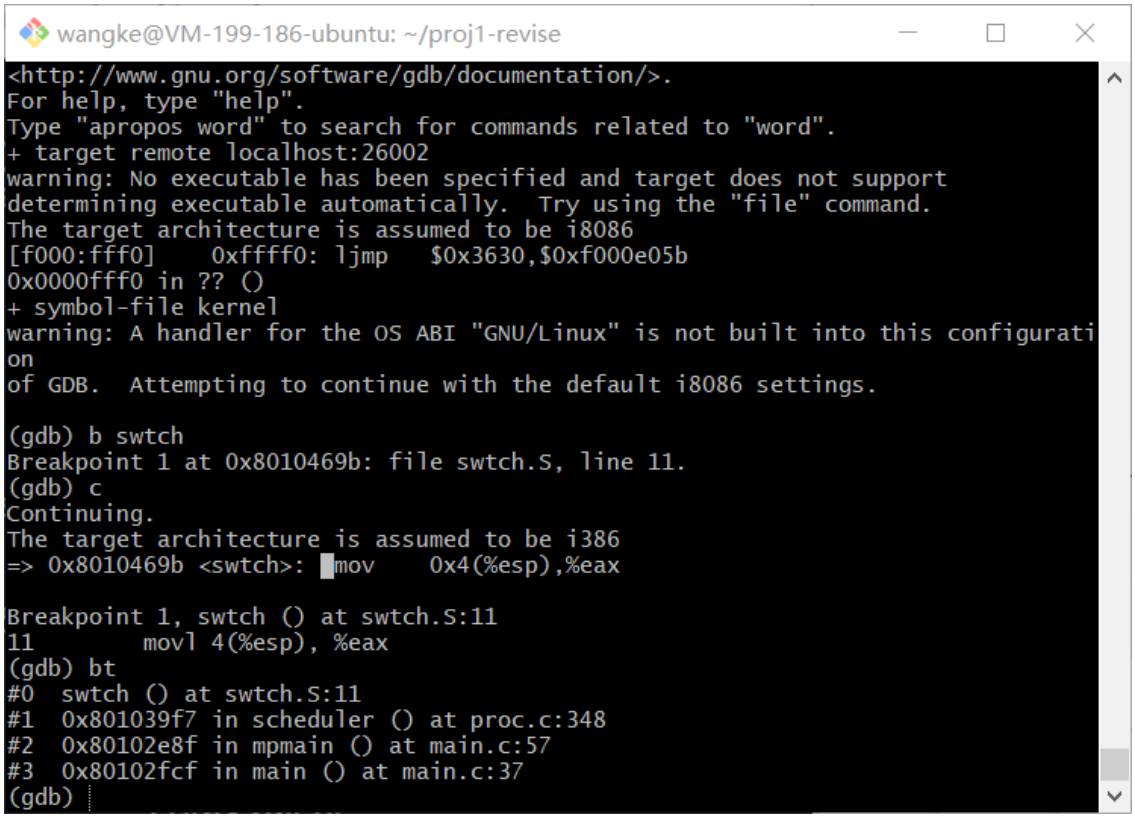
- 进入中断后cs寄存器值变为8，最后两位为00，进入了内核态，而进入终端之前，值为27(00011011) 在用户态中。进入中断后esp为0x8dfb efe8 大于0x8000 0000 ，也证明栈在内核空间中。因此这段代码本身运行在用户态用户栈中，而进入中断后，进入内核态内核栈。

- 在执行命令`shutdown`的过程中，当cpu执行到涉及特权指令的函数`outw()`时，CPU运行在用户态还是内核态？运行在哪个栈上面？
  - 使用命令break x86.h:29 设置断点进入outw函数如下：

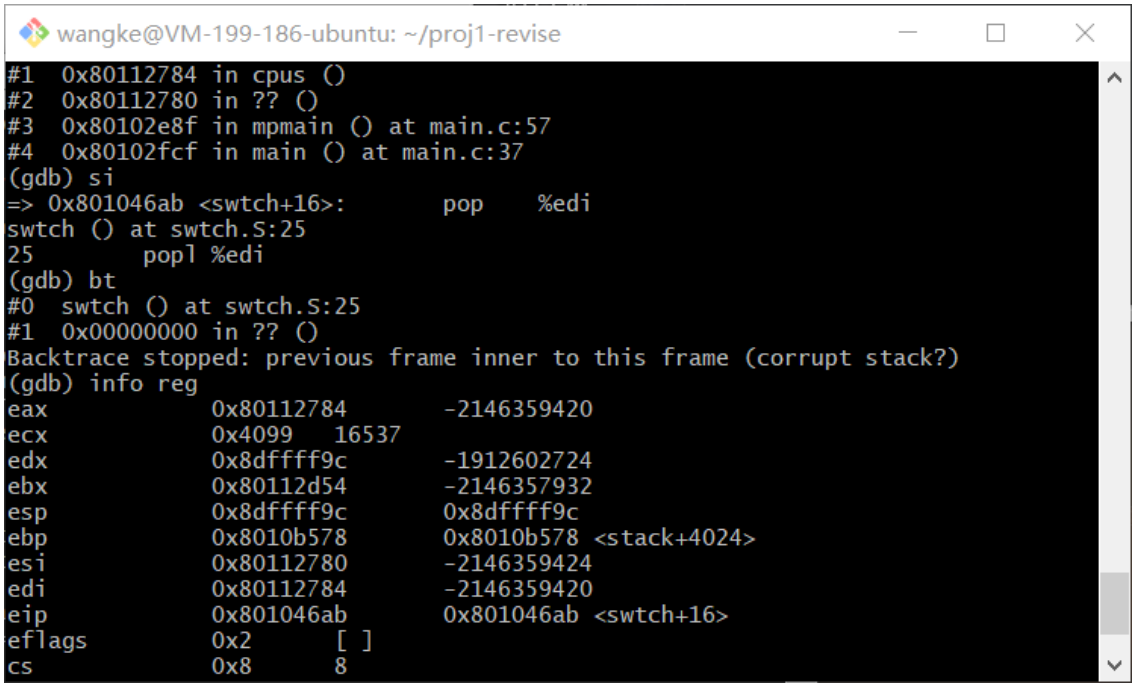


- 查看寄存器状况得到cs为8，运行在内核态，同时esp为0x8dfg ef50，大于0x8000 0000 因此栈处于内核空间中。

- 为何在执行完`swtch`函数后，cpu没有像普通函数调用一样返回到`scheduler`函数中？
  - 使用“ b swtch”设置断点，触发断点后查看如下：



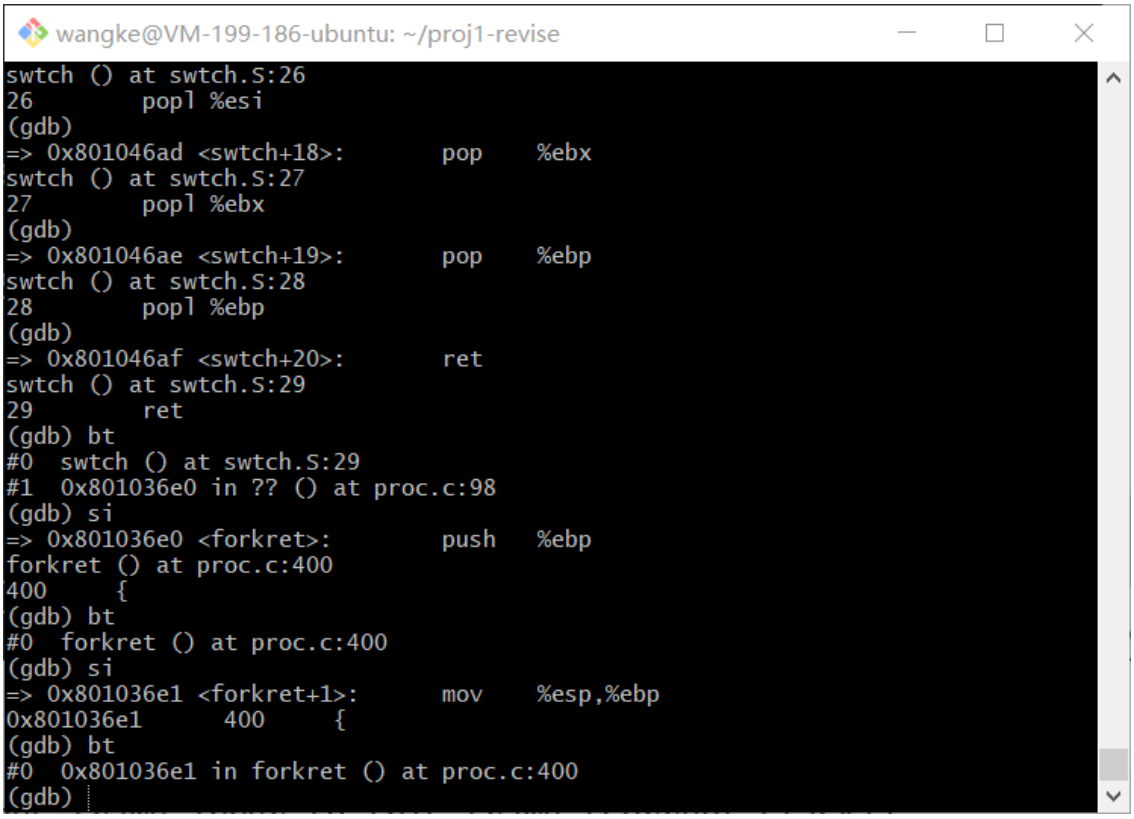
- 不断使用si单步执行并查看寄存器内容，与调用层次，均为main()->mpmain()---，esp变化幅度不大，直到一条pop指令后：



- esp明显变化，且调用层次提示

```
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

- 继续单步执行，至ret后，返回到forkret()函数中，如下所示:



- 在程序中，创建进程空间时将其上下文中eip设置为forkret()函数的起始地址，因此进程运行返回时，会继续执行forkret()代码，而其上还有一个指向trapret的指针，因此forkret()执行完毕返回时，将继续执行trapret()函数，而该函数恢复用户寄存器并跳转到process代码。由于这样的机制，完成swtch函数后，cpu不会返回到scheduler()函数中。

3. 子进程优先的fork

阅读源码可知，父进程创建完子进程空间并复制内容后，并未进行额外操作，只是将子进程状态修改为RUNNABLE，使其进入就绪状态，等待时间片轮转。因此大多时间父进程需要执行完毕后才会将时间片释放交给子进程执行（父进程执行所需时长小于时间片长度）。若想要子进程优先执行，有两种方案：

- 一是让父进程调用sleep进入休眠，待到子进程执行完毕后，使用wakeup函数将父进程唤醒继续执行。然而因为test程序给定无法修改，因此只能使用第二种方法。
- 二是设置一个信号值，使父进程创建完子进程后交出时间片，这样就可以使得子进程相对父进程而言优先执行。完成子进程优先级更高的需求。

实现sys_fork_winner(void)系统调用的过程中需要传入一个整型参数winner，采用第一题中的方法用argint函数读取栈内存，获得forktest函数中给出的参数，并将其存入一个全局变量地址空间中(该全局变量winner在proc.h中被定义，由于该头文件被proc.c和sysproc.c同时引用，所以可以直接定义)。同时，在fork函数创建完子进程后，若winner为1，则调用yield()函数交还时间片，由此完成需求。代码如下:

```
// proc.c
// int fork(void)
if(winner)
    yield();

//sysproc.c
int
sys_fork_winner(void){

  if(argint(0,&winner)<0)
    return -1;
  return 0;
}
```

回答问题:

- 父进程优先时，子进程先于父进程打印的原因：
  - 若父进程fork后恰好时钟发出中断，使得父进程丢失时间片，则子进程将优先完成输出。
- 子进程优先时，父进程先于子进程打印的原因：

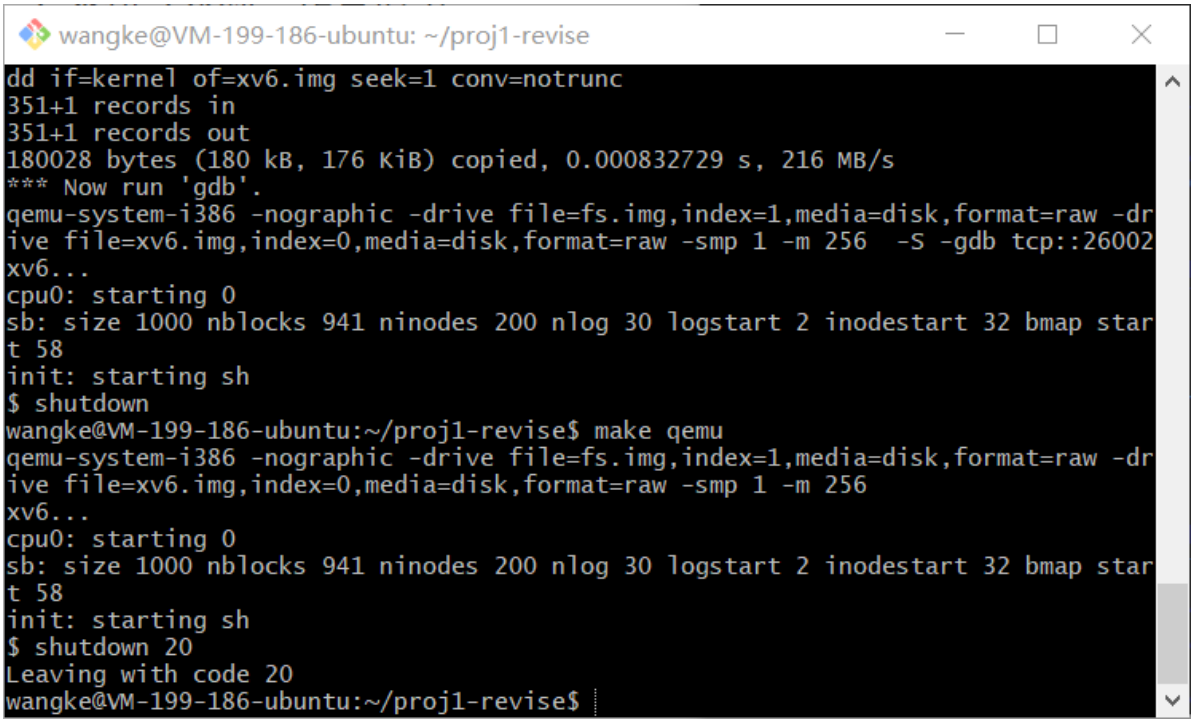
```
  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);
  if(winner)
    yield();
```

  - 对于如上代码，若编译器进行优化，调换第7和第3行语句执行顺序，或者运行时出错导致第七行优先执行，则父进程退出时子进程在这一轮时间片分配中未处于就绪状态，不会执行，等到父进程执行完毕后，才进入就绪，开始执行。

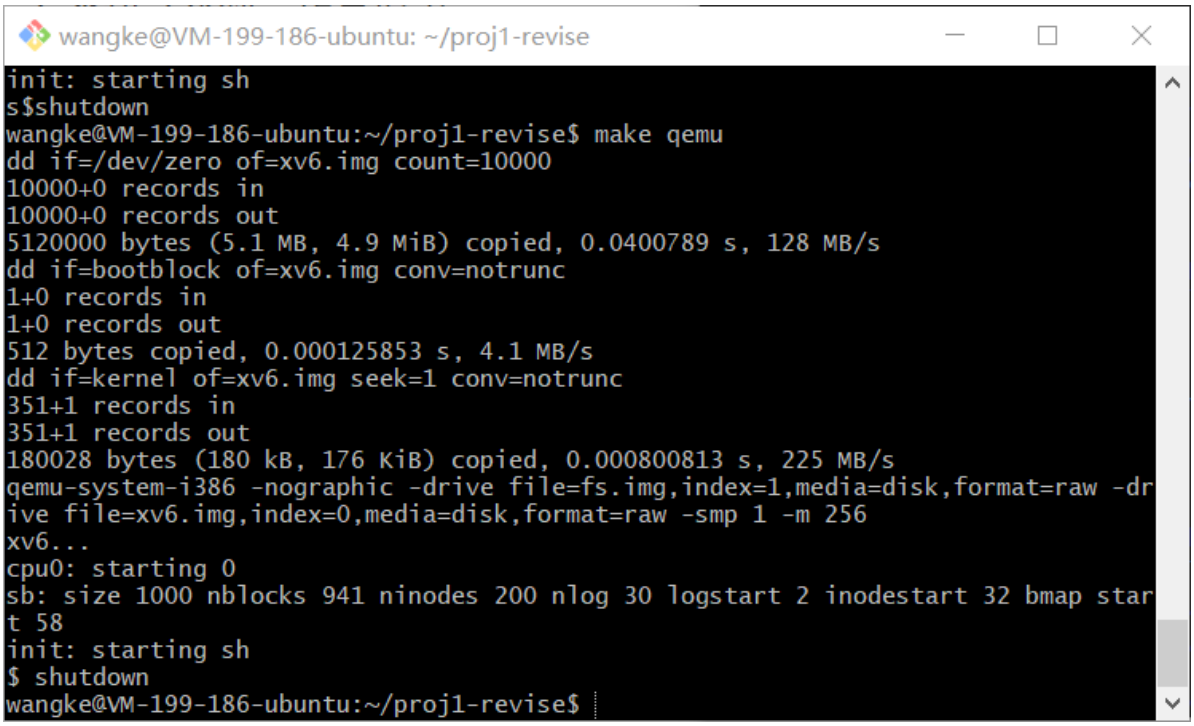
三、 实验结果

1. 带参数的系统调用

在xv6中输入shutdown 20，按要求输出了指定内容：

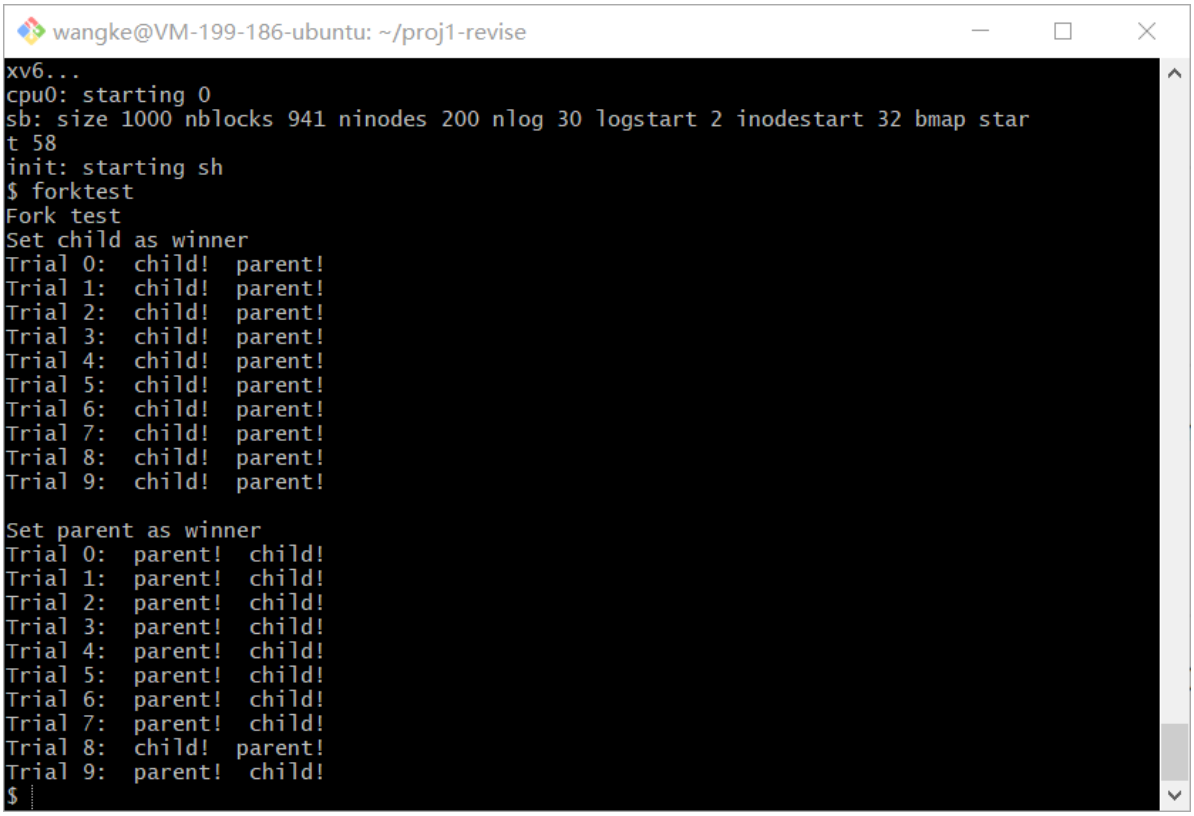


若不包含参数，正常退出：



3. 子进程优先的fork

在xv6中输入forktest，输出以下信息：



四、 实验心得

本次实验更多的阅读了xv6系统的源码，对其运行机制，特别是进程调度的方法有了进一步的认识。同时学习了gdb调试方法，增加了新的知识。

Proj2

一、 题目简述

1. 线程支持(60%)

修改xv6系统使得系统支持多线程，任务分为两部分，第一部分为库函数，第二部分为系统调用（为使得编程和阅读方便，我在proc.c 文件中实现三个用户函数，使用系统调用包装）。

1.1 库函数

根据pthread库，实现下列三个函数：

```c
// 1
int xthread_create(int *tid, void *(*start_routine)(void *), void *arg);
// 2
void xthread_exit(void *ret_val_p);
// 3
void xthread_join(int tid, void ** retval);
```

(1) 创建线程

该函数主要用来控制线程的创建， 返回成功与否，主要功能有获得线程tid（本实验中用pid代替），使线程执行函数设置为start_routine，并传入参数arg，需要考虑函数主动退出和运行完成被动退出时返回值的获取问题。

(2) 等待线程

此函数功能与**wait()**函数类似，调用者将等待线程号为tid的线程执行完成并将返回值传回，再继续工作，考虑参考wait，使用**sleep**与**wakeup**机制完成函数。

(3) 退出线程

函数主要目标是主动结束线程，获得返回值（退出码）以供其他线程获得（调用join函数）。

1.2 系统调用

实现以下三个系统调用辅助完成：

```c
// 1
int clone(void *(*fn)(void *), void *stack, void *arg);
// 2
void join(int tid, void **ret_p, void **stack);
// 3
void thread_exit(void *ret);
```

(1) clone

- clone 使得新的线程共享创建者的地址空间，可以继续使用PCB数据结构；
- 通过设置新线程的初试状态trapframe使得线程运行函数fn；
- 通过传入create函数malloc获得的地址空间，设置用户栈(注意malloc返回空间低地址)。

(2) join

- 通过join调用完成等待，并将终止线程的返回值存入参数，将用户栈释放（通过free低地址的方式）。

(3) thread_exit

- 主动终止线程，需要记录返回值

1.3 约定

- 主线程才执行clone（不需要考虑多层父子关系）；
- 主线程调用exit不调用thread_exit（不需要判断是主线程还是子线程）；
- 主线程调用exit时，终止所有未终止线程并释放资源；
- 主线程才会fork
- PCB不可以添加太多字段

---

2. 优先级调度(40%)

在xv6原生的时间片轮转调度算法的基础上实现一个优先级调度算法：

- 线程具有三个优先级：1最高，3最低；
- 线程在创建时默认优先级为2；
- 调度时优先调度优先级最高的进程
- 同一优先级的多个进程采用轮转调度

实现两个系统调用来辅助完成任务：

```
int enable_sched_display(int i);
int set_priority(int pid, int prior);
```

二、 实验过程(1)

1. 线程支持

1.1 全局考虑

1. 线程使用PCB数据结构，将tid存入pid字段，创建者为其parent，pgdir等字段均继承自其创建者。起初我以为用户的内核栈也应公用地址空间，因此自己重新实现了allocproc函数，并未使用内置的kalloc函数。然而实现时才考虑到上下文及初始化等问题，重新使用allocproc分配内存空间。

   既然使用PCB，那么自然也是存入ptable数组中，即最多产生64个进程（线程）。

2. 创建时，主线程并不主动放弃时间片，只有等到join的时候才会进入sleep，等待子线程完成工作并唤醒主线程。

3. 通过阅读源码可知，线程自身结束后（无论是主动结束还是被动结束），均进入僵尸状态，只有在主线程调用exit或者其他线程调用join时才会被彻底释放，因此有可能存在同时几个线程结束并产生返回值的情况，因此，每一个线程都需要一个独立的存放返回值的空间，而不可以共用一个（这容易导致先结束的进程返回值被覆盖）。因此，设置一个PCB字段存放返回值是比较方便的选择。

4. 然而由于分数限制只能开辟一个新的字段，因此这个字段要尽可能多加利用。每个子线程自己拥有的资源只有一个用户栈。而栈由于仅由esp指针控制，具有较大的局限性，而若标记其起始或者终止位置，则可以将其化为类似数组的，可以随机存储访问的数据结构，大大提高其利用效率和便利性。因此选择在PCB中添加stakc_top字段（实际上是分配的低地址，取名top是因为栈自高向低的特性使然）。

1.2 创建线程

1.2.1 主要思路

调用allocproc函数获得PCB并分配内核栈，将相关的PCB字段由创建者继承，并根据malloc分配一个用户栈，将stack_top字段赋值，并在栈顶端导入参数和返回地址(0xffffffff)，并初始化eip和esp指针。最后将子线程的状态设为RUNNABLE可运行。

接下来讨论返回地址问题，根据试验任务书提醒，返回地址设置为0xffffffff可以触发PAGE FAULT自陷，通过trap函数进行处理，中断号为14。实现一简单函数辅助完成被动退出后线程的处理任务如下。首先从trapframe的eax寄存器中将返回值取出，放入ret中，并存到用户栈空间底部（我认为此处可以视作无法到达，或者如果到达了，也可以在最后时间覆盖，不做使用）。随后，将该线程状态置为僵尸，唤醒其父进程，然后关闭线程资源并进入调度。

```
// in trap.c
case T_PGFLT:    // 14 PAGE FAULT
  {
```

```
    int ret = 0;
    ret = myproc()->tf->eax;
    int *ret_temp;
    ret_temp = (int*)myproc()->stack_top;
    *ret_temp = (int)ret;
    myproc()->state = ZOMBIE;
    if(myproc()->parent->state == SLEEPING)
      myproc()->parent->state = RUNNABLE;
    sched_return();
    break;
  }

// in proc.c
void
sched_return(void){
  struct proc *curproc = myproc();
  int fd;

  //close all open files
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }
  acquire(&ptable.lock);
  sched();
}
```

1.2.2 实现代码

```
// in xthread.c
int xthread_create(int * tid, void * (* start_routine)(void *), void * arg)
{
    // add your implementation here ...
    int flag;
    void *stack;
    stack = (void *)malloc(4096);
    flag = clone(start_routine, (void *)((int)stack+4096), arg);
    if(flag == -1){
        return -1;
    }
    *tid = flag;
    return 1;
}
// in proc.c
int clone(void* (*fn)(void *), void *stack, void *arg){
  int i, pid;
  struct proc *curproc = myproc();

  struct proc *p;
  if((p = allocproc()) == 0){
    return -1;
  }
  p->sz = curproc->sz;
  p->pgdir = curproc->pgdir;
  p->parent = curproc;
```

```
  p->stack_top = stack-4096;
  *p->tf = *curproc->tf;

  //Clear %eax so that return 0
  //np->tf->eax = 0;
  stack -= 4;
  *((int *)stack) = (int)arg;
  stack -= 4;
  *((int *)stack) = 0xffffffff;
  p->tf->eip = (uint)fn;
  p->tf->esp = (uint)stack;

  //similiar fork()
  for(i = 0; i < NOFILE; i++){
    if(curproc->ofile[i]){
      p->ofile[i] = filedup(curproc->ofile[i]);
    }
  }
  p->cwd = idup(curproc->cwd);

  safestrcpy(p->name, curproc->name, sizeof(curproc->name));

  pid = p->pid;
  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);
  return pid;
}
```

1.3 等待线程

1.3.1 主要思路

对于一个线程是否满足等待要求，有以下两点：

- 该线程存在可以等待的其他线程（根据测试样例中，不存在调用兄弟线程的情况，即存在孩子线程）
- 该线程在等待结束后可以继续执行，即处于正常运行状态（否则没有等待的意义）

因此设置haveKid和使用killed字段进行判断join调用的成功与否。

而函数具体逻辑为：

- 对所有PCB进行扫描，找到自己的目标孩子并且已经执行完毕的线程，将其资源释放，状态改为UNUSED释放PCB，并从stack_top位置取得返回值，将返回值置入参数地址中。若没有找到符合条件的线程，调用sleep函数进入睡眠状态等待孩子将自己唤醒。而若唤醒自己的不是目标孩子，则该线程会继续陷入睡眠直到目标执行完毕为止。

1.3.2 实现代码

```
int join(int tid, void **ret_p, void **stack){
  struct proc *curproc = myproc();
  struct proc *p;
  int haveKid, pid;
  acquire(&ptable.lock);
  for(;;){
    haveKid = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc){
```

```
        continue;
      }
      haveKid = 1;
      if(p->state == ZOMBIE && p->pid == tid){
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        *stack = p->stack_top;
        *ret_p = *(void**)p->stack_top;
        release(&ptable.lock);
        return pid;
      }
    }
    if(!haveKid || curproc->killed){
      release(&ptable.lock);
      return -1;
    }
    sleep(curproc, &ptable.lock);
  }
  return 0;
}
```

1.4 退出线程

1.4.1 主要思路

大体退出思路与被动退出线程大同小异，关闭文件等释放所有文件资源，并将返回值置入栈底部位置，将栈低地址返回给库函数进行空间释放（使用free函数），最后将线程状态设置为僵尸，进入调度。

1.4.2 实现代码

```
// xthread.c
void xthread_join(int tid, void ** retval)
{
    // add your implementation here ...
   void *stack;
    join(tid, retval, &stack);
    free(stack);

}

// proc.c
int
thread_exit(void *ret)
{
  struct proc *curproc = myproc();
  int fd;

  if(curproc == initproc){
    panic("init exiting");
  }

  //close files
```

```
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;
  acquire(&ptable.lock);

  //Take the test file into account, the thread that call the function
  // join must be the main thread ( the parent process ), so we just need to
  // wake up the parent process
  wakeup1(curproc->parent);

  *(int*)curproc->stack_top = (int)ret;
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

1.5 回答问题

- 在你的设计中，struct proc增加了几个字节？（优先级调度部分增加的不算）

  在PCB中，我增加了4个字节 stack_top（一个Int型字段），该字段既可以用来表示应该释放的栈地址空间，也可以用于随机访问栈地址空间，同时，还用于存放了返回值。

2. 优先级调度

实现优先级调度的主要思路为，对于所有PCB列表，按照优先级由高到低的顺序遍历，并在观察到更高优先级的线程时，从该优先级重新遍历，以确保高优先级的任务可以优先执行。

同时，在proc.c 定义一个全局变量用于控制sched()函数是否输出调度信息。以此控制测试的实现。实现代码如下：

```
// proc.c
int display_enabled = 0;

int
set_priority(int pid, int prior)
{
  struct proc *p;

  acquire(&ptable.lock);  //  to ensure the setter's success, hold the lock
  for(p = ptable.proc; p< &ptable.proc[NPROC]; p++){
    if(p->pid != pid)
      continue;
    p->priority = prior;
    release(&ptable.lock);
    return -1;
  }
  release(&ptable.lock);
  return pid;
}
```

```
20
21  int
22  enable_sched_display(int i)
23  {
24    display_enabled = i;
25    return display_enabled;
26  }
27
28  void
29  scheduler(void)
30  {
31    struct proc *p;
32    struct cpu *c = mycpu();
33    c->proc = 0;
34    int flag = 1;
35    for(;;){
36      // Enable interrupts on this processor.
37      sti();
38
39      // Loop over process table looking for process to run.
40      acquire(&ptable.lock);
41      for(int i = 1; i <= 3;){
42        flag = 1;
43        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
44          if(p->state != RUNNABLE || p->priority > i)
45            continue;
46          if(p->state == RUNNABLE && p->priority < i){
47            i = p->priority;
48            flag = 0;
49            break;
50          }
51
52          // Switch to chosen process.  It is the process's job
53          // to release ptable.lock and then reacquire it
54          // before jumping back to us.
55          c->proc = p;
56          switchuvm(p);
57          p->state = RUNNING;
58
59          swtch(&(c->scheduler), p->context);
60          switchkvm();
61
62          // Process is done running for now.
63          // It should have changed its p->state before coming back.
64          c->proc = 0;
65        }
66        if(flag) i++;
67      }
68
69      release(&ptable.lock);
70
71    }
72  }
73
74
75  void
76  sched(void)
77  {
```

```
  int intena;
  struct proc *p = myproc();
  if(display_enabled)
    cprintf("%d - ",p->pid);
 //.....
}
```

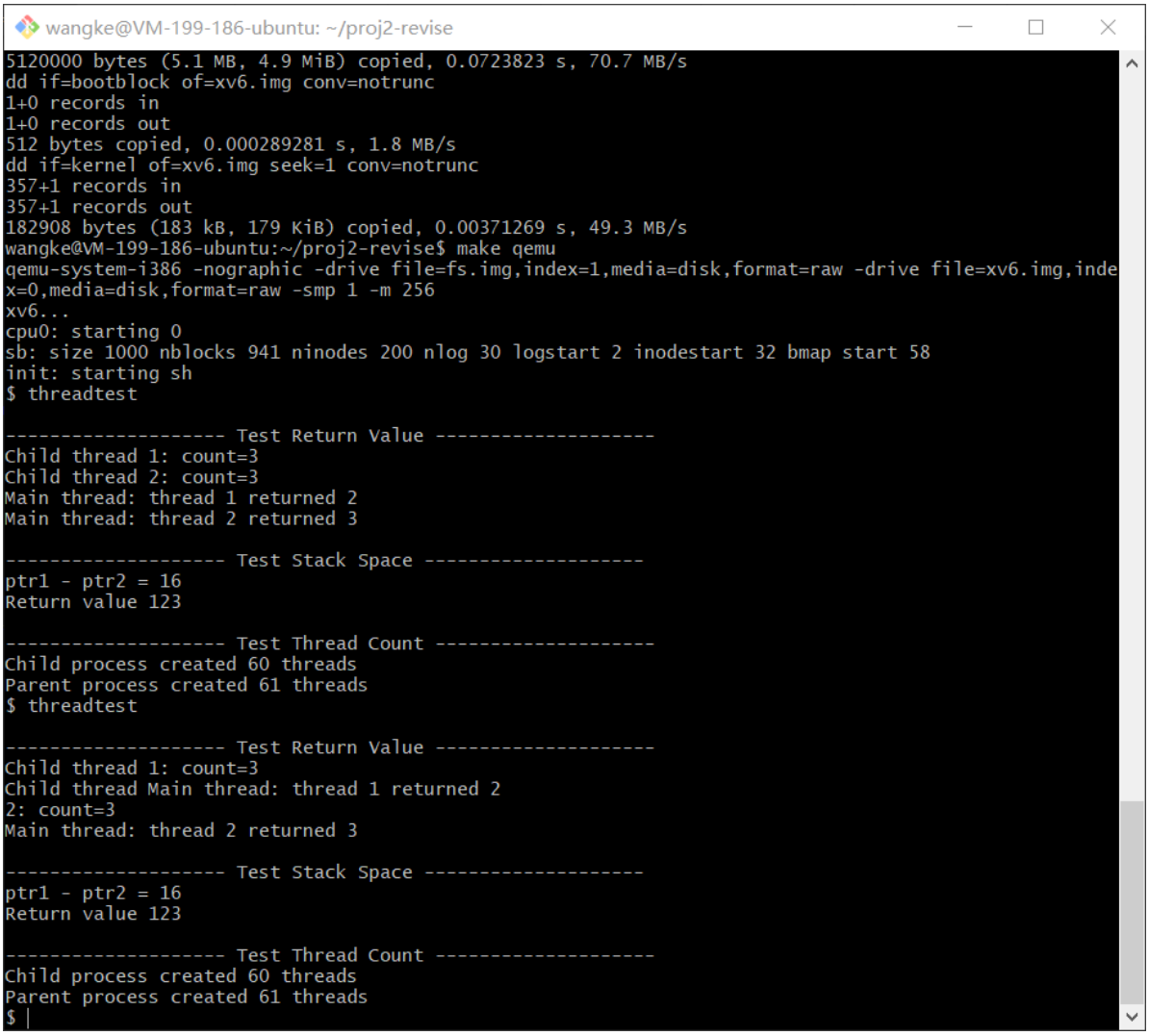
父进程的优先级为1，为何有时优先级低的子进程会先于它执行？父进程似乎周期性出现在打印列表中，为什么？（你需要阅读schedtest.c）set_priority系统调用会否和scheduler函数发生竞争条件（race condition）？你是如何解决的?

1. 在测试函数中，进行调度时，父进程循环7次调用wait()函数，进入sleeping状态，而调度函数只调度状态为RUNNABLE的进程。因此此时若没有与父进程优先级相同的进程，则调度算法将会将时间片交给低优先级的子进程，从而产生优先级低的子进程先于父进程执行的情况。
2. 父进程周期性出现在打印列表中也是出于上面解释的原因，当父进程执行wait()后，任一执行完成的子进程都将唤醒它，而因为父进程优先级高，被唤醒后又会很快进入wait()，因此会周期性出现在打印列表中。
3. 会发生竞争，因为两个函数都会访问ptable，并且对ptable中的信息进行修改（状态和优先级），若同时操作有可能出现数据丢失。我仿照其他使用ptable的函数，在进入set_priority循环前对ptable加锁，函数结束时再释放锁。

三、 实验结果

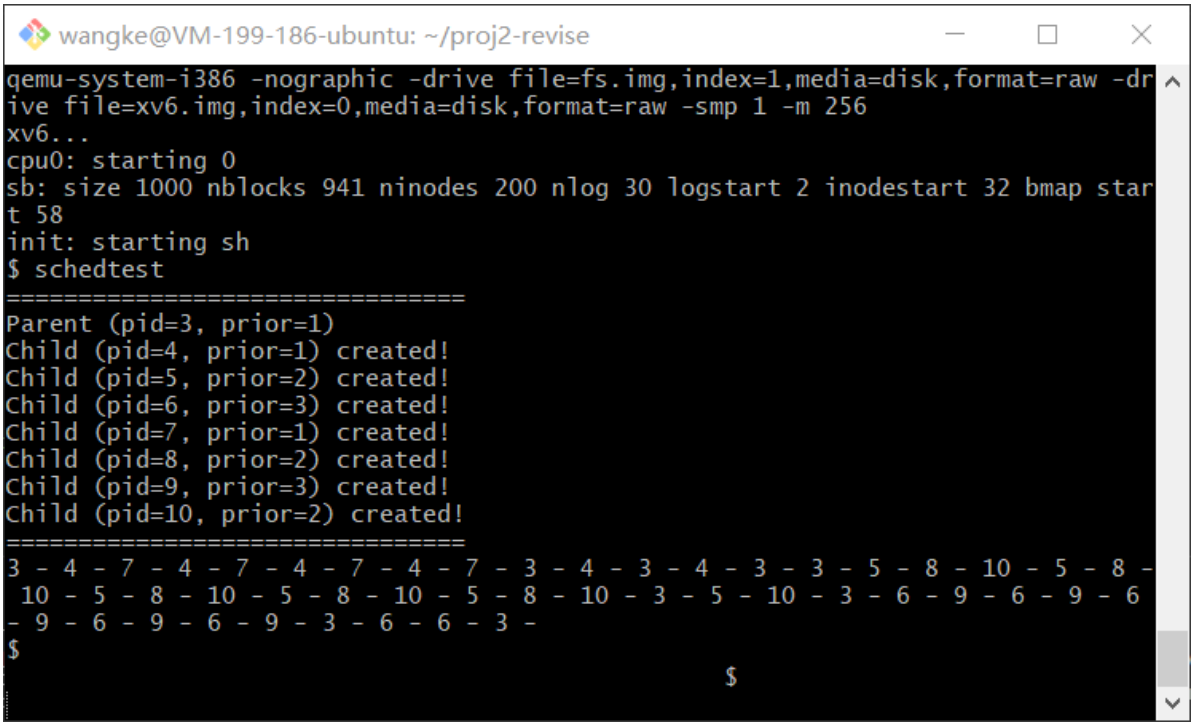
1. 线程支持

在xv6中连续输入两次threadtest，按要求输出了指定内容：



3. 优先级调度

在xv6中输入schedtest，输出以下信息：



四、实验心得

本次实验对进程和线程的内存空间理解更加深刻，起初希望使用纯使用内核栈的方式进行实现，达到完全不修改PCB的目的，但是由于时间问题，最终没能完美地实现，因此更改为在PCB中增加一个字段stack_top。而在整个实验中，给我印象比较深的其实是过程中对实验不要求的内容的畅想，比如实验限制非主线程不会再调用thread_create()函数，那么如果其他线程也能创建会是什么样的情况？又或者说，将PCB和TCB分开，将TCB放入PCB中，是不是能扩展系统的能力等等，这些拓展的想法使得我更加思考深入的内容。

需要进行一点说明的是，这一次实验的代码我参考了部分林新智同学的实现，虽然整体思路都是我和他共同思考讨论得出的。我本来意图实现另一套思路，在PCB里不添加任何字段，全部使用父线程的内核栈完成，在context下先使用一个Int型存储现存的子线程数量，再用64*3* sizeof(Int)的大小存储大小为64的结构体数组，用来存储子线程的TID，返回类型（主动退出为0，被动退出为1），和返回值。通过这种方式，不仅父线程可以随时取到返回值和自己的子线程，其他线程对兄弟线程也有访问和查询的能力。并且，可以通过这样的方式实现递归exit()，只要对每个子线程的tid，找到其内核栈，就能再递归查找到其子线程，从而进行树形搜索资源释放。同时，由于PCB中有parent字段，这样的线程树还是一棵双向树，更容易实现线程之间的联系。在实现过程中，测试点1和测试点2都顺利完成，但测试点3中，却遇到了无法创建60个线程就遭遇各种trap，panic的状况，努力查找bug后也仅仅发现不存储子线程数量可以使得线程全部创建，但是这是因噎废食的方案，并不能解决问题，由于时间的限制，在gdb调试手段的协助下我也未能完成所有的测试。因此我转而使用了和林新智同学一样的实现方法，未来有机会的话再尝试进行debug.

Proj3

一、 题目简述

1. 信号量(50%)

- 借助spinlock，为xv6添加信号量的支持。

  定义一个信号量结构体`struct semaphore`，成员自定，然后在内核里开辟一个包含100个信号量的空间，假如是s[100]，提供以下**系统调用**给用户程序：

  ```
  1  int alloc_sem (int v);
  ```

  创建一个初始值为v的信号量，返回信号量的下标。若返回值为-1，表明分配失败。分配失败的原因可能是初始值为负数或者系统中信号量资源不足。

  ```
  1  int wait_sem(int i);
  ```

  对信号量s[i]执行wait操作；成功返回1，出错返回-1。出错的原因可能是该信号量不存在，如i不在0~99之间，或者信号量尚未分配。

```
int signal_sem(int i);
```

对信号量s[i]执行signal操作；成功返回1，出错返回-1。出错的原因同上。

```
int dealloc_sem(int i);
```

删除信号量s[i]，将s[i]标记为未分配。同时将所有等待s[i]的进程终止（终止进程时参考kill的实现）。成功返回1，失败返回-1。失败的情况包括下标不在合法范围内以及信号量未被分配。

---

2. 消息传递(50%)

xv6没有实现消息传递机制，你需要添加这个支持，实现以下系统调用。

```
int msg_send (int pid, int a, int b, int c);
```

将a,b,c三个整数发送给编号为pid的进程，直到接收者进程执行完receive才返回。若接收者尚未执行receive，则进入阻塞状态。成功返回1，失败返回-1。失败的情况包括系统资源不足或者不存在编号为pid的进程。

```
int msg_receive(int *a, int *b, int *c);
```

接收一个消息(三个整数)，返回发送者的进程编号。如果未接收到任何消息，则陷入阻塞状态。

你可以用前面实现的信号量，但请注意:

- 信号量应当用时分配，不可以提前分配
- 当进程消亡时，它所占用的信号量资源应当释放
- 先执行receive，则receive会阻塞，直到有进程send
- 先执行send，则send会阻塞，直到有进程receive

二、 实验过程

1. 信号量

1.1 全局考虑

1. 信号量的实现较为简单，观察测试程序可以得出，信号量的测试主要是考察如下两点
    - 对于错误情况的处理
    - 阻塞与唤醒的控制

    因此，事实上信号量实验的核心就是实现题给四个函数调用，完成介绍中描述的功能即可。
2. 在本次实验中，需要设计一个数据结构用于表示信号量并且完成一定功能。包括分配，使用和删除。
3. 为了使得删除时杀死所有相关进程，那么需要存储进程相关内容，考虑的方法有数组和链表两种，出于习惯，选择了链表的方式（虽然这给后面的工作带来了很多很多麻烦）。

1.2 数据结构

建立的数据结构主要考虑：需要记录信号量值，需要对信号量加锁，需要记录信号量当前状态，同时使用链表结构记录相关进程，并用一个整型数记录相关进程数量（等待阻塞状态的进程数量）。对于链表中的进程，使用另一个数据结构proc_list,存储是否被使用，进程PCB，下一个结点。

```
//semaphore
```

```
struct semaphore{
  struct spinlock lock;
  int value;
  int used;
  struct proc_list *procList;
  int length;
};

struct proc_list{
  int id;
  int used;
  struct proc *proc;
  struct proc_list *next;
};
```

1.3 分配信号量

1.3.1 主要思路

对于信号量的分配主要关注两点

1. 逻辑的实现：

   首先，规定了SEM_SIZE,信号量的最大数量，控制最多能访问的信号量数量，同时检测信号量初始值不能为负。当找到可用的信号量时，将其标记为不可用并进行初始化。

2. 资源的保护

   由于信号量表是所有进程共享的资源，容易存在并发问题，因此需要对其加以保护。使用spinlock对其进行加锁，在更改完表项后解锁。

1.3.2 实现代码

```
int
alloc_sem(int v)
{
  if(v < 0) return -1;
  int index;
  acquire(&semaphores_lock);
  for(index = 0; index < SEM_SIZE; index++){
    if(semaphores[index].used == 0){
      semaphores[index].used = 1;
      semaphores[index].value = v;
      semaphores[index].procList = NULL;
      semaphores[index].length = 0;
      initlock(&semaphores[index].lock,"sem");
      release(&semaphores_lock);
      return index;
    }
  }
  release(&semaphores_lock);
  return -1;
}
```

1.4 wait与signal

1.4.1 主要思路

wait和signal是信号量处理的主要机制，在实现上也有对应的关系：

wait时，首先将信号量值减一，若不足0，则进入阻塞，将自己加入到信号量的等待列表中，并增加链表长度，过程中需要对信号量加锁。

而signal时，先将信号量加一，若非正，则说明有任务在等待，所以将等待列表中的第一项释放，唤醒，减少链表长度后退出。

1.4.2 实现代码

```
int
wait_sem(int i)
{
  if(i < 0 || i >= SEM_SIZE) return -1;
  if(semaphores[i].used == 0) return -1;
  acquire(&semaphores[i].lock);
  semaphores[i].value--;
  if(semaphores[i].value < 0){
    struct proc_list *p = semaphores[i].procList;
    if(semaphores[i].length == 0) {
      semaphores[i].procList= PLalloc();
      p = semaphores[i].procList;
    }else{
      while(p->next != NULL) p = p->next;
      p->next = PLalloc();
      p = p->next;
    }
    if(p == NULL) return -1;
    semaphores[i].length += 1;
    p->proc = myproc();
    p->next = NULL;
    sleep(myproc(), &semaphores[i].lock);
  }

  release(&semaphores[i].lock);
  return 1;
}

int
signal_sem(int i)
{
  if(i < 0 || i >= SEM_SIZE) return -1;
  if(semaphores[i].used == 0) return -1;
  acquire(&semaphores[i].lock);
  semaphores[i].value++;
  if(semaphores[i].value <= 0){
    wakeup(semaphores[i].procList->proc);
    struct proc_list *p = semaphores[i].procList;
    semaphores[i].procList = p->next;
    PLfree(p);
    semaphores[i].length -- ;
  }
  release(&semaphores[i].lock);
  return 1;
}
```

1.5 回收信号量

1.5.1 主要思路

回收信号量，实际上就是将信号量标为未使用状态，但因为还需要同时杀死所有相关进程，所以要遍历信号量的进程链表，找到所有相关进程，用其进程号将其杀死，并回收链表节点资源。

1.5.2 实现代码

```
int
dealloc_sem(int i)
{
  if(i < 0 || i >= SEM_SIZE) return -1;
  if(semaphores[i].used == 0) return -1;
  acquire(&semaphores[i].lock);
  if(semaphores[i].used == 1){
    semaphores[i].used = 0;
    while(semaphores[i].procList != NULL)
    {
      kill(semaphores[i].procList->proc->pid);
      struct proc_list *p = semaphores[i].procList;
      semaphores[i].procList = p->next;
      PLfree(p);
    }
  }
  release(&semaphores[i].lock);
  return 1;
}
```

1.6 问题简述

关于你实现的wait_sem，如果多个进程同时执行wait_sem，但wait的具体信号量不同，它们能并

发执行吗？（你应该用一个spinlock去保护一个信号量，另外用一个spinlock保护资源分配）

- 是可以并发执行的。在分配信号量时，由于需要遍历整个表，我使用了semaphores_lock来进行加锁保护，而分配结束后就进行了释放。对于单个信号量的锁，只保护自身。所以多个信号量之间并不涉及到竞争，是可以并发执行的。

2. 消息传递

2.1 全局考虑

消息的传递分为发送和接受，由于是在不同进程内，实现消息传递的同步需要信号量协同完成。当发送方发送但对方未接收时，或者接收放意图接收但无进程发送消息时，都将进入阻塞状态等待同步。这里面同样关键的一点在于，设计数据结构使得每一次交互得以被记录。我采用双端记录的方式，每个进程只考虑与自己相关的内容，不需要对所有信息进行遍历，一定程度上提高了效率。

2.2 数据结构

设计如下数据结构：

在每一个记录的MSGstate结构中包含许多字段， 其中pid表示当前处理的进程号，记录下进程号是因为未来表中是以PCB为单位进行记录而不是进程号，因此若进程出现切换，可能导致同一个PCB下其实是不同的进程，如果保留原来的接收块则容易出错。used字段标识资源是否被使用，positive为主动标识，表示该块记录的是pid字段中进程主动发出的行为还是其他进程发出的行为。target为目标，记录发送方或者接收方的PCB id或者pid根据使用需求填写。sid为当前通讯所使用的信号量id。最后还记录了数据和下一条指针。

链表表头以PCB为单位，因为进程号是向上无穷增长的，而xv6中进程PCB是有限的，因此，只需要创建NPROC数量的表块即可，分为发送和接收分别进行记录。事实上，经过分析可以发现，每一个进程至多发送一个消息，即只需要一个发送块，因此事实上发送表并不需要链表的结构，其实数组就可以实现了，并且这些发送都是主动行为，其positive字段均应该为1。

```
// Massage trans
struct MSGstate{
  int id;
  int pid;
  int used;
  int positive;
  int target;
  int sid; //semaphore id
  int data[3];
  struct MSGstate *next;
};


struct MSGstate * Msend[NPROC];
struct MSGstate * Mrecv[NPROC];
```

2.3 发送消息

2.3.1 主要思路

在进程发送消息时，首先要考虑对方是否已经在等待接收消息，如果已经在等待，则直接将数据填入结构中并通过信号量通知对方接收即可，若对方无请求，则需要在己方和对方处均进行记录（将自己的发送块设置为使用并将对方信息填入块中，同时为对方建立一个被动接收块），代码如下：

2.3.2 实现代码

```
int
msg_send(int pid, int a, int b, int c){
  int found = getProcNum(pid);
  int semaphore;
  if(found < 0)
    return -1;

  acquire(&Msg_lock);
  int now_pid = myproc()->pid;
  int myNum = getProcNum(now_pid);
  struct MSGstate *p = Mrecv[found];

  for(; p != NULL ; p = p->next){  // if the target is waiting for receiving

    if(p->positive == 1 && p->pid == pid){

      p->data[0] = a;
      p->data[1] = b;
      p->data[2] = c;
      p->target = now_pid;   //set the target to be the process
      signal_sem(p->sid);
      release(&Msg_lock);
      return 1;
    }
  }
  struct MSGstate *myp = Msend[myNum];
  p = Mrecv[found];
  semaphore = alloc_sem(0);
  if(semaphore == -1) return -1;

  if(Mrecv[found] == NULL){   // set a nagetive receive for the target
```

```
    Mrecv[found] = MSalloc();
    p = Mrecv[found];
  }else{

    for(; p->next !=NULL; p = p->next);
    p->next = MSalloc();
    p = p->next;
  }
  if(p == NULL) return -1;
  p->pid = pid;
  p->positive = 0;
  p->data[0] = a;
  p->data[1] = b;
  p->data[2] = c;
  p->target = myNum;
  p->sid = semaphore;
  p->next = NULL;

  Msend[myNum] =  MSalloc();
  myp = Msend[myNum];
  if(myp == NULL) return -1;
  myp->pid = now_pid;  // set a positive send for the process itself
  myp->positive = 1;
  myp->data[0] = a;
  myp->data[1] = b;
  myp->data[2] = c;
  myp->target = found;
  myp->sid = semaphore;
  release(&Msg_lock);

  wait_sem(semaphore);

  acquire(&Msg_lock);
  MSfree(myp);
  Msend[myNum] = NULL;
  dealloc_sem(semaphore);
  release(&Msg_lock);

  return 1;
}
```

2.3 接收消息

2.3.1 主要思路

接收消息时的思路与发送较为类似，首先考虑是否有进程已经向自己发送消息，即检查自己的MRecv表中是否有被动消息，若检查有，则将其中数据和target取出，获得数据并返回对方pid。若没有，则给自己的MRecv表中添加一个被动接收块，以供其他进程查找，自身进入阻塞等待消息。

2.3.2 实现代码

```
int
msg_receive(int* a, int* b, int* c){
  int i;
  int target;
  int now_pid= myproc()->pid;
  int flag = 0;
  int semaphore;
```

```
  int myNum = getProcNum(now_pid);
  struct MSGstate * pre_p = Mrecv[myNum];
  struct MSGstate * p = Mrecv[myNum];

  acquire(&Msg_lock);

  for(i = 0; p != NULL; p = p->next ,flag = 1, i++){  // if other process has sent some msg
      if(flag) pre_p = pre_p->next;
      if(p->pid == now_pid && p->positive == 0){
        *a = p->data[0];
        *b = p->data[1];
        *c = p->data[2];
        // cprintf("now recieve %d \n",*a);
        target = p->target;
        semaphore = p->sid;
        if(i == 0) Mrecv[myNum] = p->next;
        else{
          pre_p->next = p->next;
        }
        MSfree(p);
        signal_sem(semaphore);
        release(&Msg_lock);
        return target;
      }
  }

  for(i = 0; p != NULL; p = p->next ,flag = 1, i++){  // if other process has sent some msg
      if(flag) pre_p = pre_p->next;
      if(p->pid != now_pid){
        if(i == 0) Mrecv[myNum] = p->next;
        else{
          pre_p->next = p->next;
        }
        MSfree(p);
      }
  }
  semaphore = alloc_sem(0);
  if(semaphore == -1){
    release(&Msg_lock);
    return -1;
  }
  if(Mrecv[myNum]==NULL){
    Mrecv[myNum] = MSalloc();
    // cprintf("\n----------------want things pid: %d myNUM: %d\n\n",now_pid, myNum);
    p = Mrecv[myNum];
  }else{
    pre_p->next = MSalloc();
    p = pre_p->next;
  }
  if(p == NULL) {
    cprintf("allocate wrong\n");
    return -1;
  }

  p->pid = now_pid;
```

```
  p->positive = 1;
  p->sid = semaphore;
  p->target = myNum;
  p->next = NULL;

  // cprintf("in %d recv is %d   \n",now_pid,Mrecv[myNum]->pid);
  release(&Msg_lock);
  // cprintf("%d  in waiting\n",p->pid);
  wait_sem(semaphore);
  acquire(&Msg_lock);
  // cprintf("%d  finish waiting\n",p->pid);
  *a = p->data[0];
  *b = p->data[1];
  *c = p->data[2];
  target = p->target;
  if(i == 0) Mrecv[myNum] = p->next;
  else{
       pre_p->next = p->next;
      }
  MSfree(p);
  dealloc_sem(semaphore);
  release(&Msg_lock);
  return target;
}
```

2.4 辅助函数

在如下函数中，进行了锁和表的初始化，并且实现一个函数，通过pid值获得该进程存储于PCB中的偏移量，以此确定存储位置。

```
// initialize the spinlocks and the tables
void initMsg(){
  initlock(&Msg_lock,"msg_lock");
  initlock(&semaphores_lock,"sem");
  initlock(&PLlock,"proclist_lock");
  for(int i = 0 ; i < 1000; i++){
    PL[i].used = 0;
    PL[i].id = i;
    MS[i].id = i;
    MS[i].used = 0;
  }
  initlock(&MSlock,"massage_lock");
  for(int i = 0; i < NPROC; i++){
      Msend[i] = NULL;
      Mrecv[i] = NULL;
  }
  return;
}

// get the location in ptable by pid
int
getProcNum(int pidd)
{
  struct proc *p;

  acquire(&ptable.lock);
  int num = 0;
```

```
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++,num++){
    if(p->pid == pidd){
      release(&ptable.lock);
      return num;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

2.4 问题简述

如果receive先执行，但send未执行，你是如何让receive阻塞的（或者说阻塞到哪个信号量上

了）？如果send先执行，但receive未执行，你是如何让send阻塞的?

- receive先执行的情况中，接收方进程在自己的MRecv表中建立一个positive字段值为1的接收块，并将自己分配到的信号量id（初始值为0）存入块中，接收方便对此信号量进行wait操作。发送方检查到接收方有主动接收块，则将数据存入其中，并取出semaphore id字段，对其执行signal来唤醒接收方进程。
- 反之类似，send先执行的情况中，发送方将给接收方在MRecv字段中创建一个被动接收块，并将分配得到的Sid存入，进行wait，接收方在receive时先检测到自身MRecv中有被动接受块，则从其中取出数据与Sid, 执行signal操作唤醒发送方。

3. 其他设计

在整个流程中，因为初期想要使用的是链表，因此第一时间想到动态内存分配，可以节省大量空间。

代码完成后，编译意识到在proc中不能使用malloc，我进一步尝试重新编写malloc，换引用关系，甚至添加系统调用，都无法实现动态的内存分配，出于时间原因，我模仿alloc_proc的做法，先将所有资源分配完成，建立1000个可供分配的资源块，在程序申请时进行分配。以此实现了伪分配的效果，代码如下:

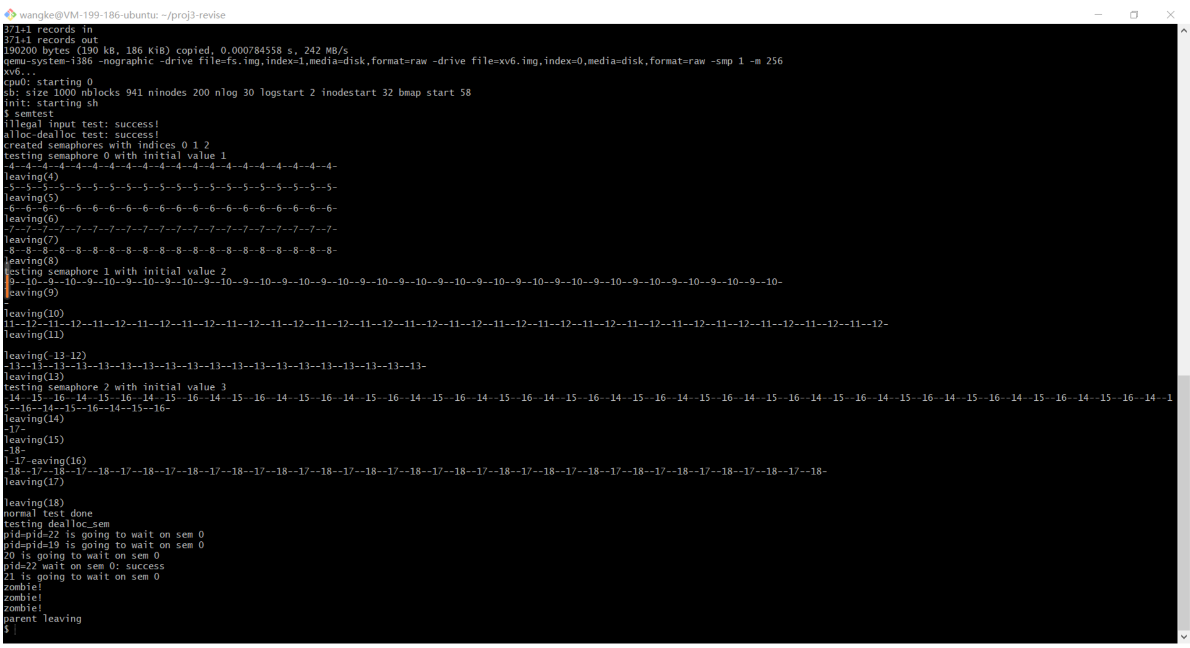
```
struct proc_list PL[1000];
struct MSGstate MS[1000];

struct proc_list *
PLalloc(){
  acquire(&PLlock);
  int i;
  for(i = 0 ; i < 1000; i++){
    if(PL[i].used == 0){
      goto found;
    }
  }
  release(&PLlock);
  return NULL;

found:
  PL[i].used = 1;
  release(&PLlock);
  return &PL[i];
}

struct MSGstate *
MSalloc(){
  acquire(&MSlock);
  int i;
```

```c
  for(i = 0 ; i < 1000; i++){
    if(MS[i].used == 0){
      goto found;
    }
  }
  release(&MSlock);
  return NULL;

found:
  MS[i].used = 1;
  release(&MSlock);
  return &MS[i];
}

int
PLfree(struct proc_list *pl){
  acquire(&PLlock);
  pl->used = 0;
  release(&PLlock);
  return 1;
}

int
MSfree(struct MSGstate *ms){
  acquire(&MSlock);
  ms->used = 0;
  release(&MSlock);
  return 1;
}
```

三、 实验结果

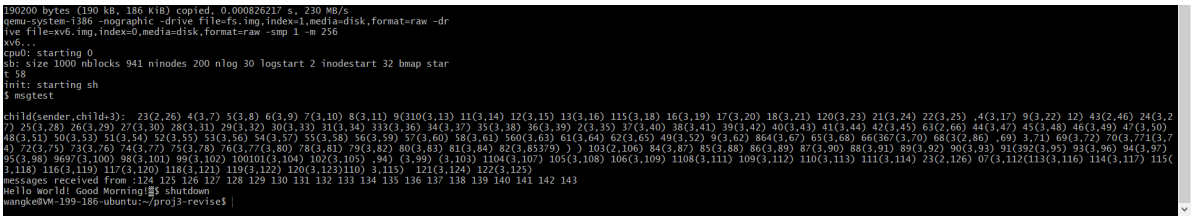
1. 信号量

在xv6中输入semtest，按要求输出了指定内容：



2. 消息传递

在xv6中输入msgtest，输出以下信息：



四、实验心得

本次实验中，我在动态分配内存上花的时间比较多，虽然最后没有能真正实现内存的分配，但是通过模仿alloc_Proc, 也完成了目标需求，在指针操作的编写和调试时，出现了一些问题，比如指针指向方向并没有连接到链表中，出现了trap等等。指针的错也只能回去一行行看代码，好在进行实验前，我已经对于整体框架的设计比较细节，虽然还有一些没有考虑到的点，但在出错后能很快通过输出标识定位，并解决问题。

Proj4

一、 题目简述

1. 写时复制的fork（90%）

fork产生的子进程和父进程有很大的相似性：代码段一样，数据段一样，栈段一样，堆段也一样。xv6在实现fork时，会为子进程的代码、数据、堆、栈均分配物理内存，这有一些浪费，尤其是，若子进程接着调用了exec，则刚分配的物理内存又需要释放。你需要完成以下任务（请参阅教材9.3节的内容）。

- fork执行时，只为子进程创建页表，这个页表完全复制了父进程的页表（包括页框号）。同时，将子进程和父进程的可写页面标记为只读，这些页面称为写时复制页面，它们本来可以被写，只是因为写时复制技术才标记为只读，需要将这些页面与本来就是只读的页面做区分（缺页中断时需要区分处理）。可以通过页表项中的第9-11位来做标记（这几位硬件未使用，是供操作系统用的）
- 当父进程（或子进程）试图写这些页面时，会触发缺页中断，核实原因为写内存（通过error code判断）并且所写页面为写时复制页面时，再为其分配一个页框，并将原有的内容复制到此页框中，然后修改当前进程的页表项为可写。提示：将页表项的权限改为只读 *pte&=~PTE_W ； 将其设置为可写 *pte|=PTE_W 。

- 当进程消亡时，需要释放页框。但如果两个进程共享一个页框，则提前消亡的进程在释放页框后会造成存活进程无法正常运行。一种策略是为每个已分配的页框记录一个引用数，记录该页框被几个进程的页表所指，每次释放会将引用数减1，当引用数为0时再真正释放页框。**你需要在哪些地方维护这个引用数？请写到课设报告中。**
- xv6最多支持64个进程，所以引用数用1个字节表示即可（c语言中可用 char 类型）。Makefile中指明物理内存大小为256M（QEMUOPTS中的 -m 256 ），据此计算出页框总数，从而得出需要多少字节来存储引用数。为简单起见，你可以在内核里定义一个全局的数组来记录系统启动后页框被进程引用的数量（被内核引用多少次倒不必记录，这里记录的引用数以能满足copy-on-write fork的要求为准）
- 由于一个页框可能分配给两个进程，所以有可能出现两个进程同时释放页框的情况，即，在对页框的引用数做修改时可能出现竞争条件。复用kmem.lock来解决这个问题。

---

2. 开放任务(10%，未实现)

二、 实验过程

1. 写时复制的fork

1.1 全局考虑

1. 在进行fork时，会调用一个函数进行页目录的创建，包括四个部分

- 建立页目录表（setupkvm）
- 建立页目录项（walkpgdir）
- 分配页框（kalloc， mappages）
- 复制页框内容（memmove）

  实际上复制页框内容的memmove出现在分配页框的mappages之前，但为了方便理解，做如上排序。

  其中，为了实现写时复制，即是要修改后面两个部分的内容，取消新页框的分配，也就不需要复制页框内容，而为了满足页表要求，需要将页目录项重定向到父进程的页框上，并且相对修改一些Flag标志位，并且根据要求修改Val空闲位，以供后续中断时识别使用。

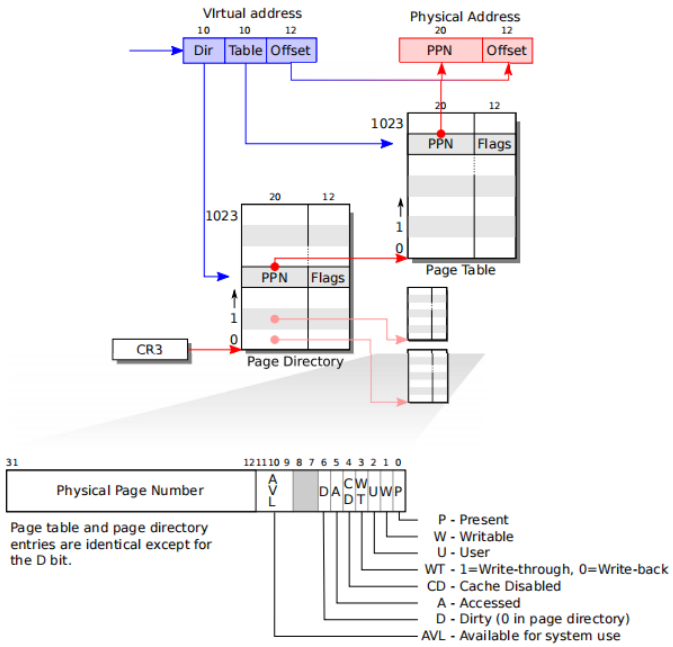


Figure 2-1. x86 page table hardware.

2. 关于写时复制，在实验前，我以为是在父进程或者子进程修改时，为子进程重新分配一个页框并将原本内容复制进去，同时把父进程的不可写状态改为可写。而在实现时，构想如何区分父进程和子进程，发现太过困难，因为对于树结构的进程结构而言，若出现三层以上的继承关系，父进程同时也是子进程，在其被写时很难判断修改的先后，因此重新阅读题目和书后发现是在某个写时复制进程时（无论父子）给其分配一个新的页框，而若父子都失去对原有页框的引用时，将原页框释放即可。
3. 中断处理时，通过查阅题目中给的网页链接，可以发现error code中，用户进程访问不可写页面的特征为末三位为111，因此需要在trap中对其进行判断，同时，我将写时复制的val设置为001，这一部分也需要进行判断。同时，引起错误的地址在CR2中获得，使用rcr2函数。

Page Faults

A page fault exception is caused when a process is seeking to access an area of virtual memory that is not mapped to any physical memory, when a write is attempted on a read-only page, when accessing a PTE or PDE with the reserved bit or when permissions are inadequate.

Handling

The CPU pushes an error code on the stack before firing a page fault exception. The error code must be analyzed by the exception handler to determine how to handle the exception. The bottom 3 bits of the exception code are the only ones used, bits 3-31 are reserved.

```
Bit 0 (P) is the Present flag.
Bit 1 (R/W) is the Read/Write flag.
Bit 2 (U/S) is the User/Supervisor flag.
```

The combination of these flags specify the details of the page fault and indicate what action to take:

```
US RW  P - Description
0  0   0 - Supervisory process tried to read a non-present page entry
0  0   1 - Supervisory process tried to read a page and caused a protection fault
0  1   0 - Supervisory process tried to write to a non-present page entry
0  1   1 - Supervisory process tried to write a page and caused a protection fault
1  0   0 - User process tried to read a non-present page entry
1  0   1 - User process tried to read a page and caused a protection fault
1  1   0 - User process tried to write to a non-present page entry
1  1   1 - User process tried to write a page and caused a protection fault
```

When the CPU fires a page-not-present exception the CR2 register is populated with the linear address that caused the exception. The upper 10 bits specify the page directory entry (PDE) and the middle 10 bits specify the page table entry (PTE). First check the PDE and see if it's present bit is set, if not setup a page table and point the PDE to the base address of the page table, set the present bit and iretd. If the PDE is present then the present bit of the PTE will be cleared. You'll need to map some physical memory to the page table, set the present bit and then iretd to continue processing.

4. 在维护页框引用量时需要注意的是

- 页框数量为物理存储空间大小除以页面大小即256M/4KB，即为0x10000
- 当使用pa作为下表时，应该注意pa需要向右移动12位，否则将会获得基地址的数值（远远大于数组大小）
- 访问页框引用数组应当使用物理地址，若获得的是逻辑地址，需要使用V2P函数转为物理地址。

1.2 fork

1.2.1 主要思路

在对进程进行fork时，调用了copyuvm函数进行页表的创建，因此修改此函数，使之符合上面第一条全局考虑的要求即可。

我仿照PTE_ADDR宏定义了四个辅助函数于mmu.h，用于管理pte的符号位，其实现如下：

```
// modify the flags into unwritable,  retain other flags
#define PTE_Clear_writable(pte) ((uint)(pte) & 0xFFFFFFFD)

// clear the flags units of AVL, and get ready to set these units to show "Copy on Writing"
#define PTE_Clear_Val(pte) ((uint)(pte) & 0xFFFFF1FF)

// set the flags units of AVL to be normal
#define PTE_Set_Val_Normal(pte) ((uint)(pte) | 0x0)

// set the flags units of AVL to be "Copy on Writing"
#define PTE_Set_Val_Copy_on_Writing(pte) ((uint)(pte) | 0x200)
```

通过使用以上函数，可以将原有的flag更改为我们需要的，去除了Writable位并且赋值了Val的标记，将新的标记与原有物理地址合并，可以获得新的父进程PTE，而对于子进程，则通过mappages进行映射，生成新的pte指向父进程的页框。

1.2.2 实现代码

```
pde_t*
  copyuvm_new(pde_t *pgdir, uint sz)
  {
    pde_t *d;
    pte_t *pte;
    uint pa, i, flags;
    // make a new kernal page table directory
    if((d = setupkvm()) == 0)
      return 0;

    // copy all the PTE from parent
    for(i = 0; i < sz; i += PGSIZE){
      if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
        panic("copyuvm: pte should exist");
      if(!(*pte & PTE_P))
        panic("copyuvm: page not present");
      /* ------------------------------------do not allocate new pages ------*/


      pa = PTE_ADDR(*pte);

      flags = PTE_FLAGS(*pte);

      flags = PTE_Clear_writable(flags);
      flags = PTE_Clear_Val(flags);
      flags = PTE_Set_Val_Copy_on_Writing(flags);
      // if((mem = kalloc()) == 0)
      //   goto bad;
      // memmove(mem, (char*)P2V(pa), PGSIZE);
      // *pte = V2P(pa) | flags;
      *pte = pa | flags;

```

```
      if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
        goto bad;

      // add the reference number of the page
      kaddRefer(pa);

      lcr3(V2P(pgdir));

    }
    return d;

  bad:
    lcr3(V2P(pgdir));
    freevm(d);
    return 0;
  }
```

1.3 trap

1.3.1 主要思路

首先需要判断自陷类型是否为目标类型

- PageFault 页访问错误
- 通过查询tf->err得知是用户访问不可写页面
- 通过查询pte得知是写时复制的页面报错

确认目标类型后即可展开操作，先获取当前指向的目标页，将其引用删除，同时创建新页表，复制原页表内容并重定向pte到新的页框，并且更改flags位，使得可写（因为可写之后不会进入二重判断，所以不修改Val也可以）。增加新页面的引用。

在这个过程中，我本来使用mappages重新定向，但报错remap，仔细检查代码后发现，mappages禁止向以及存在并PTE_P位有效的页面重定向。因此改为手动修改PTE的值。

1.3.2 实现代码

```
// trap.c
case T_PGFLT:
    // cprintf("get pgflt\n");
    if((tf->err & 0b111) == 0b111){   // a user process are to write on a protected page
      char *a = (char*)rcr2();
      pde_t* pgdir = myproc()->pgdir;
      Handle_trap_copy_on_writing(pgdir, a);
      // cprintf("write err end in pid: %d \n", myproc()->pid);
      }
      break;


// vm.c
int Handle_trap_copy_on_writing(pde_t *pgdir, char *a){
  pte_t *pte;
  uint pa, flags;
  char *mem;
  if((pte = walkpgdir(myproc()->pgdir, a, 0)) == 0){
        return -1;
  }

```

```
  if((((*pte) & PTE_W) == 0 ) && ((((*pte) >> 9) & 0b111) == 0b001)){  // unwritable && copy_on_writing

    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    flags = PTE_Clear_Val(flags);
    flags = flags | PTE_W;

    if((mem = kalloc()) == 0){
      kfree(mem);
      return 0;
    }
    memmove(mem, (char*)P2V(pa), PGSIZE);
    kfreeRefer(pa);
    //   remap
    *pte = V2P(mem) | flags | PTE_P;
    kaddRefer(PTE_ADDR(*pte));
    // kshowRefer(PTE_ADDR(*pte));
    lcr3(V2P(pgdir));
    return 1;
  }
  return 0;
}
```

1.4 Reference Number

使用一个字符型数组char refer[65540]来记录引用量，在kalloc.c文件中分装四个相关函数，分别用于增加引用量，减少饮用量，检查引用量为0时释放页框，和显示引用量（用于调试）代码如下：

其中，对于refer数组的争用，复用kmem.lock进行加锁处理，在初期debug时发现会卡死，经过修改尝试后判断可能是kmem.lock还没生成的时候就使用了，因此导致出错，因此模仿其他kalloc中的函数加上了对kmem.use_lock的值判断。同时kfree自带有加锁，所以提前解锁。

```
int kaddRefer(uint pa){
  if(kmem.use_lock)
    acquire(&kmem.lock);
  pa = (pa >> 12);
  refer[pa]++;
  if(kmem.use_lock){
    // if(refer[pa]>2)
    // cprintf("the ++ number of %d is %d\n",pa,refer[pa]);
    release(&kmem.lock);
  }

  return 1;
}

// Minus the reference number of a page
int kfreeRefer(uint pa){
  uint paa = (pa >> 12);
  if(kmem.use_lock)
    acquire(&kmem.lock);
  refer[paa]--;

  if(kmem.use_lock){
    // cprintf("the -- number of %d is %d\n",paa,refer[paa]);
```

```
    release(&kmem.lock);
  }


  return kcheckPage(pa);
}

// Check if the reference number of page is zero, which means the page
should be released.
int kcheckPage(uint pa){
  if(kmem.use_lock)
    acquire(&kmem.lock);
  if(refer[pa >> 12] <= 0){
    char *v = P2V(pa);
    if(kmem.use_lock)
      release(&kmem.lock);
    kfree(v);

    return 1;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return 0;
}

int kshowRefer(uint pa){
  pa = (pa >> 12);
  if(kmem.use_lock)
    acquire(&kmem.lock);
  cprintf("the reference number of %d is %d\n",pa,refer[pa]);
  if(kmem.use_lock)
    release(&kmem.lock);
  return 1;
}
```

其中引用量添加的操作，实现于inituvm,allocuvm,copyuvm_new以及trap处理函数中。 而减少操作则在trap处理函数和deallocuvm中出现。

三、 实验结果

在xv6中先后输入forktest和stresstest，按要求输出了指定内容：

MINGW64:/e/SchoolStudy/2020-2021_spring/操作系统

```
182496 bytes (182 kB, 178 KiB) copied, 0.000959468 s, 190 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -dr
ive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 256
xv6...
cpu0: starting 0
number of free frames: 56917
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ forktest
fork test
fork test OK
$ stresstest
created 61 child processes
pre: 56770, post: 52562
$ forktest
fork test
fork test OK
$ stresstest
created 61 child processes
pre: 56770, post: 52562
```

四、 实验心得

本次实验中，在debug上花了很多多余的时间，因为一开始的错在初始化部分，什么报错信息都无法显示，后期出现很多死循环和页引用错，用了gdb和输出信息来确定错误位置。通过本次实验，我对操作系统如何分配页面有了进一步了解。

而关于post值发生改变，有以下考虑：

首先考虑页面数量变化的可能原因：

1. fork 中初始化写时复制错误
2. 中断处理时错误

对以上两种可能分别排查

（1） 可能1：copyuvm

在copyuvm中做如下改动，在setupkvm前后以及函数完成前分别记录空闲页面数，输出其变化（为了方便观察，更改stresstest文件中的循环次数为5）

代码(改动处已做标记）以及结果如下所示：

```
1  extern int free_frame_cnt;
2    pde_t*
3    copyuvm_new(pde_t *pgdir, uint sz)
4    {
5      pde_t *d;
6      pte_t *pte;
7      uint pa, i, flags;
8      // make a new kernal page table directory
9  //=============================================================
10     int pre,p1,p2;                              //================
11     pre = free_frame_cnt;                       //================
12 //=============================================================
13     if((d = setupkvm()) == 0)
14       return 0;
15 //=============================================================
16     p1 = free_frame_cnt;                        //================
17 //=============================================================
18
19     for(i = 0; i < sz; i += PGSIZE){
```

```
20
21        if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
22          panic("copyuvm: pte should exist");
23
24        if(!(*pte & PTE_P))
25          panic("copyuvm: page not present");
26        pa = PTE_ADDR(*pte);
27
28        flags = PTE_FLAGS(*pte);
29
30        flags = PTE_Clear_writable(flags);
31        flags = PTE_Clear_Val(flags);
32        flags = PTE_Set_Val_Copy_on_Writing(flags);
33
34        *pte = pa | flags;
35
36        if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
37          goto bad;
38
39        kaddRefer(pa);
40
41        lcr3(V2P(pgdir));
42
43      }
44  //=============================================================
45      p2 = free_frame_cnt;                    //================
46      cprintf("in %d copy: pre-p1  %d,   p1-p2:  %d
47         \n",myproc()->pid,pre-p1,p1-p2);
48  //=============================================================
49      return d;
50
51    bad:
52      lcr3(V2P(pgdir));
53      // lcr3(V2P(d));
54      freevm(d);
55      return 0;
56    }
```

```
xv6...
cpu0: starting 0
number of free frames: 56917
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start
init: starting sh
in 1 copy: pre-p1  65,   p1-p2:  1
$ stresstest
in 2 copy: pre-p1  65,   p1-p2:  1
in 3 copy: pre-p1  65,   p1-p2:  1
in 3 copy: pre-p1  65,   p1-p2:  1
in 3 copy: pre-p1  65,   p1-p2:  1
in 3 copy: pre-p1  65,   p1-p2:  1
in 3 copy: pre-p1  66,   p1-p2:  1
created 5 child processes
pre: 56770, post: 56426
$ stresstest
in 2 copy: pre-p1  65,   p1-p2:  1
in 9 copy: pre-p1  65,   p1-p2:  1
in 9 copy: pre-p1  65,   p1-p2:  1
in 9 copy: pre-p1  67,   p1-p2:  1
in 9 copy: pre-p1  66,   p1-p2:  1
in 9 copy: pre-p1  65,   p1-p2:  1
created 5 child processes
pre: 56770, post: 56425
$
```

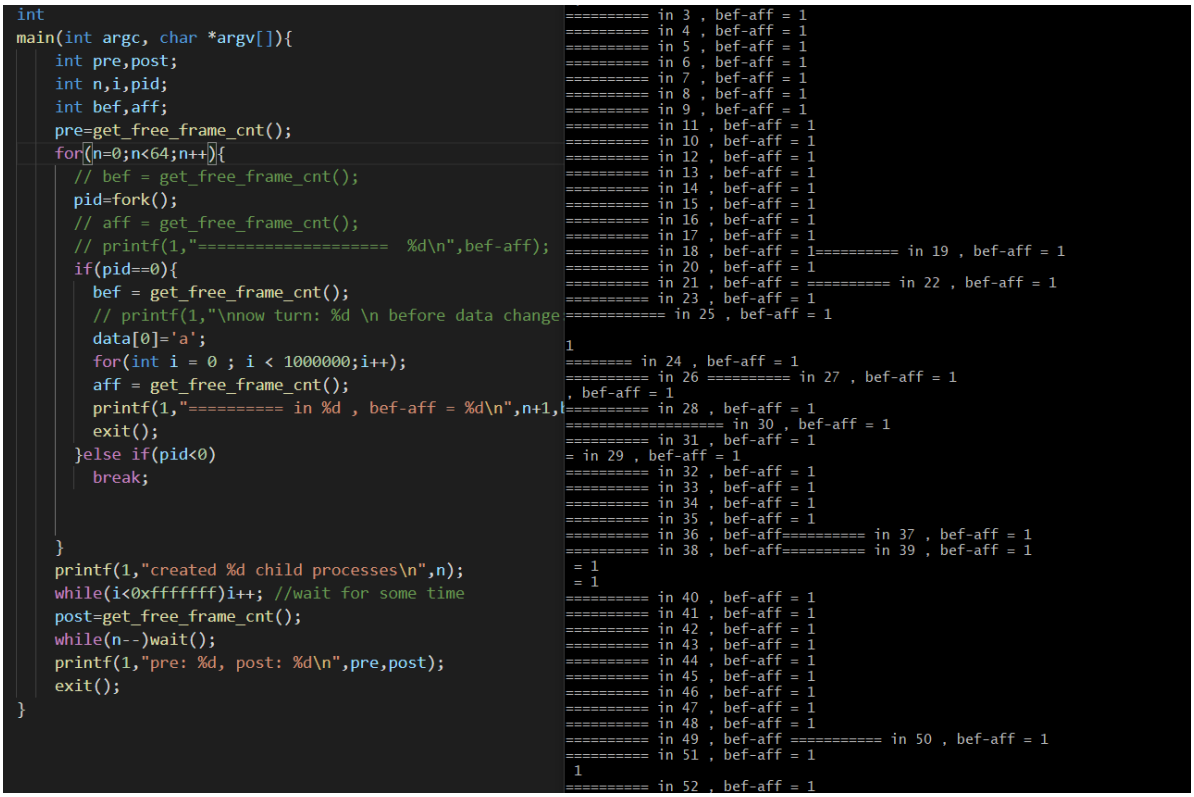
可以看出，post结果不一致时，在setupkvm之后的操作对空闲页面数量的改变是一致的，均为-1，而setupkvm本身却会导致不同数量的页面被使用。

（2）可能2： 中断处理

在测试文件中，仅一行测试了中断，即更改data值，因此修改代码如下（新增的代码已顶格写）：

```
int pre,post;
    int n,i,pid;

int bef,aff;

    pre=get_free_frame_cnt();
    for(n=0;n<5;n++){
      pid=fork();
      if(pid==0){

bef = get_free_frame_cnt();

        data[0]='a';

for(int i = 0 ; i < 1000000;i++);
aff = get_free_frame_cnt();
printf(1,"========== in %d , bef-aff = %d\n",n+1,bef-aff);

        exit();
      }else if(pid<0)
        break;

```

测试结果如下：



从右边可以看出，每一次data的更改都固定减去了一个空闲页面，属于稳定操作。

结论

综上所述，post值的改变主要是由于setupkvm函数使用页面数量不稳定导致的。由于setupkvm中无法输出，所以无法通过测试得到更具体的数据。

同时，也考虑是测试文件中，等待一段时间所执行的循环指令被编译器自动优化了，导致没有等待，影响了最终结果。

Proj5

一、 题目简述

添加文件系统校验

当xv6启动时，在mpmain()调用scheduler()函数之前，你需要检验文件系统的一致性，并修正不一致的

部分。本实验中你仅需要考虑一种不一致性：**有些inode已经被使用了，但无法从根目录搜寻到它。**比如，/home/xzhu/1.txt对应4个inode，其中，根目录、home目录、xzhu目录、1.txt文件分别对应一个inode。此时，如果根目录的内容中删除了home目录的条目，则home目录、xzhu目录、1.txt文件的inode均有问题；反之，如果仅是home目录中的xzhu条目被删除，则xzhu目录、1.txt文件的inode有问题。修正方式是，归还这些有问题的inode，并归还inode指向的文件所占用的磁盘空间。或者说，将删除操作执行完。

- **问题解释在第二部分的第五小点展开**

二、 实验过程

1. 添加文件系统校验

1.1 全局考虑

1. 在完成文件系统的遍历时，首先需要了解xv6中FILE SYSTEM的结构，如下图，第一个分区为boot block，接下来是superblock，存储了整个文件系统的大小和分区结构，而后存储了题目中提到的200个inode，bitmap以及存放数据的data blocks：

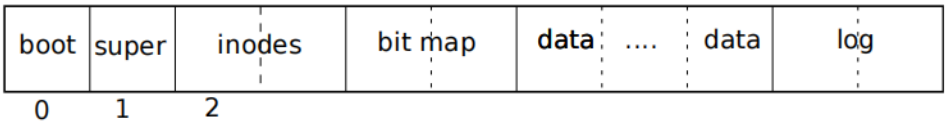


Figure 6-2. Structure of the xv6 file system. The header `fs.h` (3650) contains constants and data structures describing the exact layout of the file system.

然而，通过查询官方文档，我发现了如下描述(仅截取有用片段）：

The fifile system does not use block 0 (it holds the boot sector). Block 1 is called the superblock; it contains metadata about the fifile system (the fifile system size in blocks, the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold inodes, with multiple inodes per block.

该段表示xv6中并没有使用第一个分区，即boot block，因此在进行磁盘读写获取superblock的时候，需要注意偏移量offset为0即可。

2. 关于inode无法一次性放入内存的问题，考虑使用了栈缓冲的做法，栈的优势在于不需要程序员主动释放空间，这里算是偷了个小懒，但后面发现出了一点小小的意外（但针对此测试文件无影响）。事实上使用堆的方法与我使用的方法实现差别不大，因此改动几行代码便可以将此实现改为堆实现，所以可以理解为我这里栈的实现方式仅仅是为了尝试练习对指针、地址的控制而使用的。

3. 完成本次实验很重要的一点是需要了解inode的结构，从文档中了解结构如下图所示——除去前面的一些标志外，存放了13个addr。其中前12个为direct address，最后一个为single indirect block，而single所指向的块存放了另外128个addr，因此一个inode最多可以指向140个地址。**若深入考虑，如果inode被释放时，indirect block存在，则需要将此块也释放。**出于时间原因，暂时没有实现。

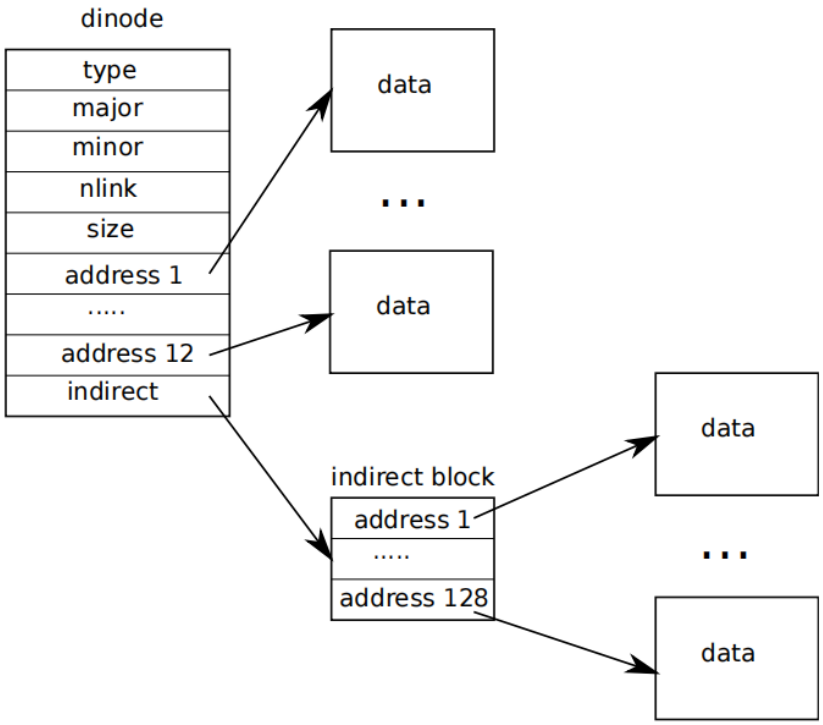


Figure 6-4. The representation of a file on disk.

1.2 读写

1.2.1 主要思路

读写磁盘的代码基本仿照了bootmain.c中的实现，具体代码如下，可以看出几处有限的修改：

1. readsec/writesec中将0xE0改为了0xF0,这是修改的磁盘目标，改为对img1的读写。
2. 写磁盘的部分将insl改为了outsl并将第二个参数改为了0x30.

1.2.2 实现代码

```
// copy from bootmain.c
void
waitdisk(void)
{
  // Wait for disk ready
```

```
  while((inb(0x1F7) & 0xC0) != 0x40);
}

// copy from bootmain.c
// Read a single sector at offset into dst.
void
readsec(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  //----------------------------------------
  // E0->F0  Read from disk img 1
  outb(0x1F6, (offset >> 24) | 0xF0);
  //----------------------------------------
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}

// Write a single sector at offset into dst.
void
Writesec(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  //----------------------------------------
  // E0->F0  Read from disk img 1
  outb(0x1F6, (offset >> 24) | 0xF0);
  //----------------------------------------
  outb(0x1F7, 0x30);  // cmd 0x30 - write sectors

  // Read data.
  waitdisk();
  outsl(0x1F0, dst, SECTSIZE/4);
}

// copy from bootmain.c
void
readblock(char* pa, uint count, uint offset)
{
  char* epa;

  epa = pa + count;

  // Round down to sector boundary.
  pa -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
```

```
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We'd write more to memory than asked, but it doesn't matter --
  // we load in increasing order.
  for(; pa < epa; pa += SECTSIZE, offset++)
    readsec(pa, offset);
}

int flag[250];

void
writeblock(char* pa, uint count, uint offset)
{
  char* epa;

  epa = pa + count;

  // Round down to sector boundary.
  pa -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We'd write more to memory than asked, but it doesn't matter --
  // we load in increasing order.
  for(; pa < epa; pa += SECTSIZE, offset++)
    Writesec(pa, offset);
}
```

1.3 第一次扫描

1.3.1 主要思路

进行第一次扫描时，我需要获取所有通过正常路径能够访问到的文件（inode），即从根目录可以访问到的文件，题目已给信息提到，根目录root存放在inum为1的dinode里，因此需要从此处进行遍历查找，获得所有合法inode，并将它们进行标记。此处使用了递归的方法，对于目录文件，首先遍历其直接地址，再考虑一次间接地址。代码的含义在下面有具体的注释：其一般流程为先获取inode中存放的地址，获取地址指向的block后，从中读取directory entry并搜索其后继，获取新的inode进行递归查找。

在实现中发现，递归层次若太多，则会产生栈溢出的问题。这个问题可以通过开全局变量（但是这样不加分）或者使用堆分配动态空间的方法解决该问题。

1.3.2 实现代码

```
void
dfs(const char* name, const struct dinode* inode,int depth){
  // address of dataBlock going to access
  uint addr;
  // dataBlock
  char db[512];
  // single block

  // inode
  struct dinode INODES[8];
  // cprintf("before if\n");
```

```
  if(inode->type == 0) return;
  if(inode->type == T_DIR){
    // 12 direct address
    for(int i = 0; i < NDIRECT; i++){
      addr = inode->addrs[i];
      if(0 == addr) {
        continue;
      }
      // get the data block on the address
      readblock(&db[0],BSIZE,(addr-1) * BSIZE);
      struct dirent* entry = (struct dirent*)(&db);
      for(int j = 0; j < ENTNUM; j++){
        // get access to each directory entry saved in the data block
        if(entry->name[0]=='.'||
        entry->inum<=0||
        entry->inum>200){
          entry++;
          continue;
        }
        // get the inode metioned in the entry
        readblock((char*)(&INODES),BSIZE, BSIZE + logNum * BSIZE + (entry->inum/8 * BSIZE));
        struct dinode* next_inode = &INODES[entry->inum%8];
        // dfs into the newly got inode and search for more legal files
        if(next_inode==inode) continue;
        // flag the found dir/file
        flag[entry->inum] = 1;
        dfs(entry->name,next_inode,depth+1);
        entry++;
      }
    }

    addr = inode->addrs[NDIRECT];
    if(addr == 0) return;
    // get the single indirect block
    readblock(&single[0],4096,(addr-1) * BSIZE);
    // for the 128 addresses in the block, find all avaliable ones
    // the address is 4-Byte long, so the number i adds itself by 4 one time
    for(int i = 0;i < NINDIRECT;i+=4){
      addr = (uint)*(&single + i);
      if(addr == 0) continue;
      readblock(&db[0],BSIZE,(addr - 1) * BSIZE);
      struct dirent* entry = (struct dirent*)(&db);
      // if the address is avaliable, look for new files and the following is some familiar actions
      for(int j = 0; j < ENTNUM; j++,entry++){
        if(entry->name[0]=='.'||
        entry->inum==0||
        entry->inum>200){
          continue;
        }
        readblock((char*)(&INODES),BSIZE, BSIZE + logNum * BSIZE + (entry->inum/8 * BSIZE));
        struct dinode* next_inode = &INODES[entry->inum%8];
        flag[entry->inum] = 1;
        dfs(entry->name,next_inode,depth+1);
      }
    }
```

```
  }else{
      // cprintf("name :  %s\n",name);
  }
}
```

1.4 第二次扫描

第二次扫描的目标是对所有inode进行扫描，对于那些自身生效，但是其指向的块儿没有在第一次扫描中被标记的inode，进行释放操作，具体操作为将其type置为0，并重新写回到磁盘中。

```
void check_fs()
{
  struct superblock sb;
  struct dinode INODE[8];
  struct dinode *check;

  flag[ROOTINO] = 1;
    // check
  int checkFlag = 0;
    // False Num
  int Fnum = 0;

  readblock((char*)(&sb),BSIZE,0);
  readblock((char*)(&INODE),BSIZE,BSIZE + (sb.nlog) * BSIZE);
  logNum = sb.nlog;
  struct dinode * root = &INODE[1];
  dfs("/",root,1);


  for(int i = 1; i < sb.ninodes; i++){
    if(i % 8 == 0){
      readblock((char*)(&INODE),BSIZE,BSIZE + (sb.nlog) * BSIZE + i / 8 * BSIZE);
    }
    check = &INODE[i % 8];
    if(flag[i] == 0 && check->type != 0){
      check->type = 0;
      Fnum = Fnum + 1;
      checkFlag = 1;
      cprintf("\nfsck: inode %d is allocated but is not referenced by any dir!",i);
      writeblock((char*)(&INODE),BSIZE,BSIZE + (sb.nlog) * BSIZE + (i / 8) * BSIZE);
      cprintf("Fixing ... done");
    }
  }
  if(checkFlag){
    cprintf("\nfsck completed. Fixed %d inodes and freed %d disk blocks.\n",Fnum,Fnum);
  }else{
    cprintf(" fsck: no problem found. ");
  }
}
```

1.5 问题解释

> 1. 为何OS尚未切换到scheduler时，读写磁盘不能用中断的方式进行?

在main函数中，操作系统正在进行初始化，若产生中断将使得部分功能未开启的环境下进入中断处理程序（理应是健全的），因此此时系统将中断使能关闭，而在scheduler函数中（如下图）调用了sli函数打开中断，此时才能触发中断。我们所进行的磁盘读取是在scheduler函数调用之前，因此不能使用中断。

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();
```

> 2. 写出你遍历目录的伪代码。

函数： Search

输入： dinode* root

操作：

```
if(root->type == T_DIR)
  |for addr:root->addr[]:
  |  if addr != 0:
  |     db = getDataBlock();
  |     for entrys in db by sizeof(entry):
  |        if inum != 0 and name != "." or ".." :
  |           findNewInode();
  |           Search(new_inode);
  |if root->addr[13]!=0:
  |  single = getIndirectBlock();
  |   for addr in single by 4:
  |      if addr != 0:
  |        db = getDataBlock();
  |        for entrys in db by sizeof(entry):
  |           if inum != 0 and name != "." or ".." :
  |              findNewInode();
```

```
|               Search(new_inode);
```

> 3. 你是使用全局变量存储inode，还是使用局部变量或者是用kalloc和kfree?

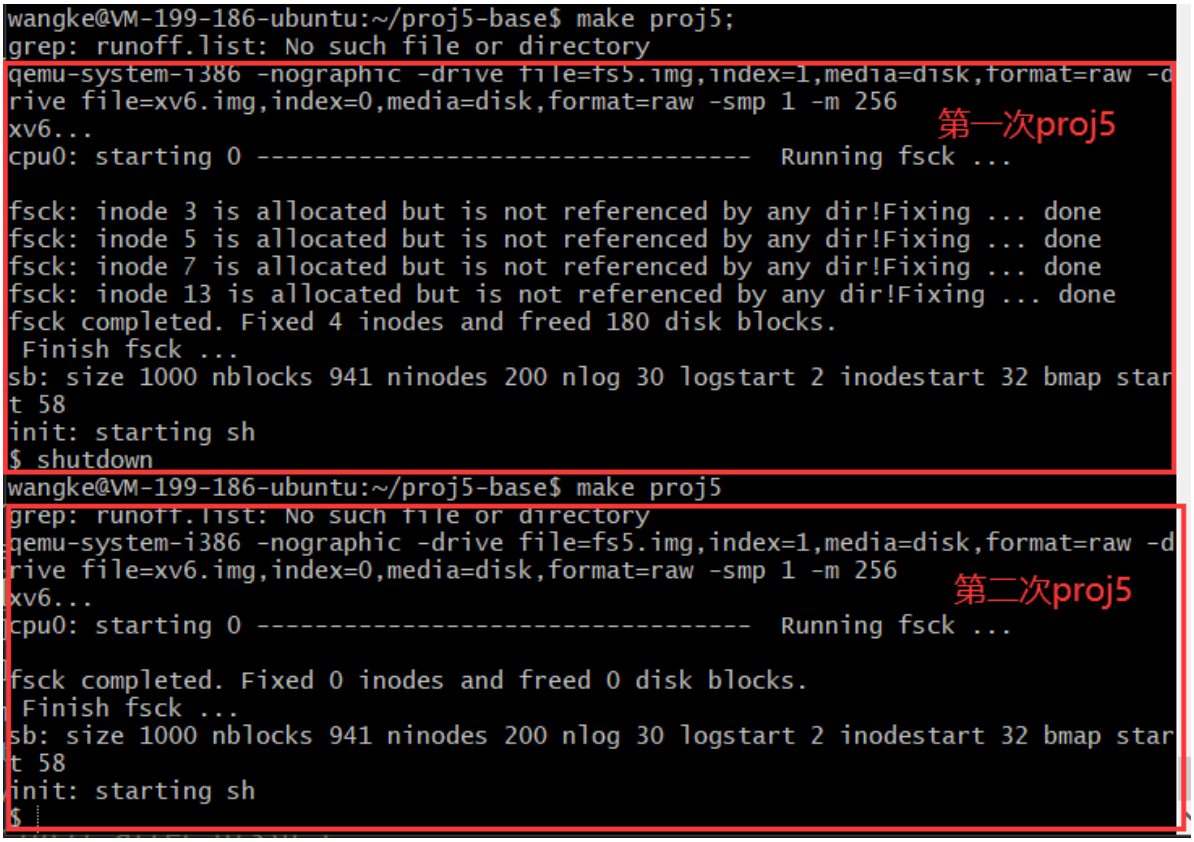
我使用的是局部变量存储inode，但事实上使用kalloc进行存储在代码上的区别不大，使用局部变量只是为了熟悉*和&操作符。

> 4. 你需要再实验报告中说明，make proj5和make proj5i分别修复了多少个inode，释放了多少个 disk blocks。

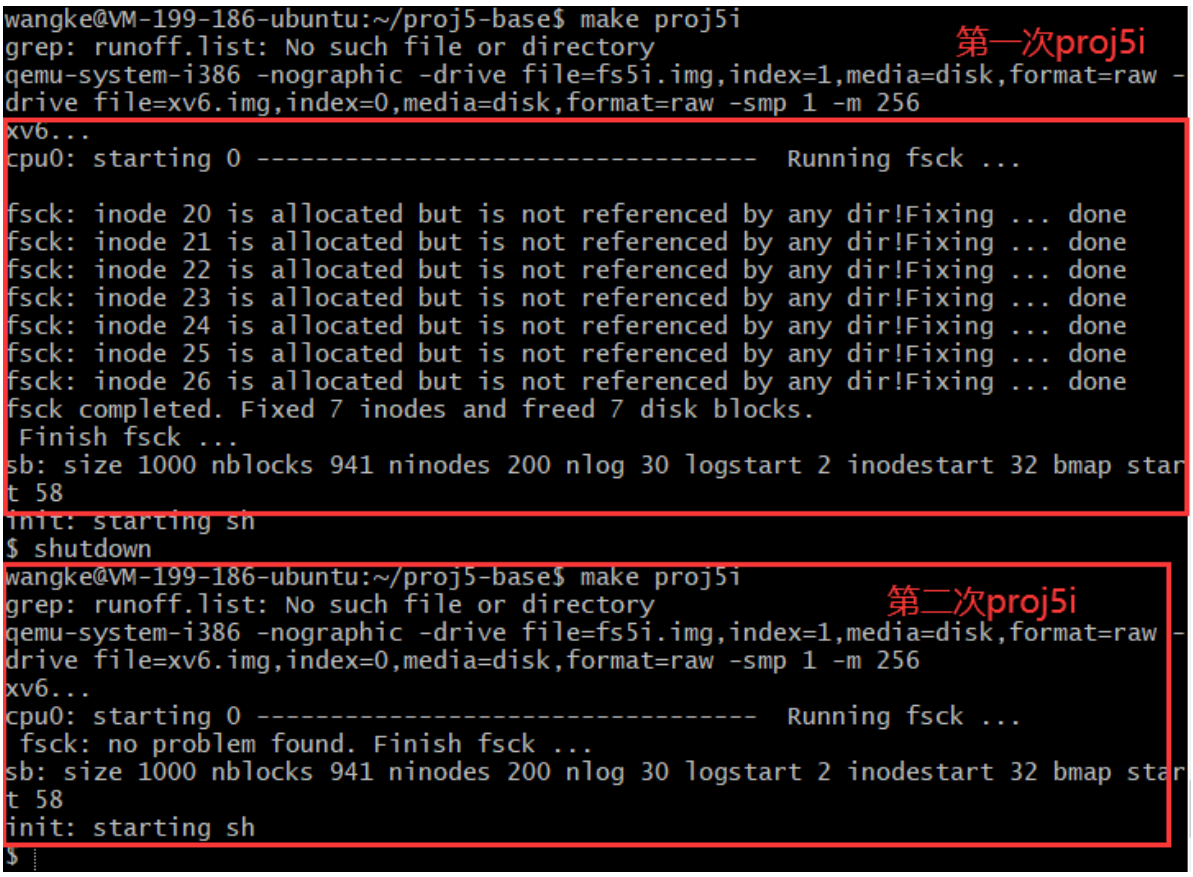
	修复inode数量	释放diskblocks数量
make proj5	4	180?
make proj5i	7	7

三、 实验结果

1. 在git bash中输入两次make proj5，按要求输出了指定内容：



2. 在git bash中输入两次make proj5i，按要求输出了指定内容：



四、实验心得

本次实验中，所花的时间大多在资料的阅读还有代码的阅读上，实际编写代码的时间较少，思路比较清晰。后期debug花费时间比较长且遇到比较离奇的错误就是爆栈，事实上存放inode的缓冲区在栈中还是可以安全存放，但存放singleIndirectBlock的4KB则太大了，因此最后我把这部分空间开成了全局变量，但这部分也可以作为动态使用，因为使用次数并不多。而本次实验我也是为了方便一次性缓冲了8个inode，事实上如果一次读取一个inode的话，应该可以使得递归层数限制翻3翻。

实验心得

完成了一共六个操作系统试验后，我对于课程内学习的内容有了更加实际的把握，了解了scheduler函数到底是如何调度选择下一个执行的进程，了解了线程的实现需要考虑哪些内容，尝试实现了进程间的同步和通信方式，明白了文件系统中的存储结构。并且，对于GDB调试方法有了深刻的认识和体会。此外，除了试验任务之外，通过阅读官方文档和代码，我也对内核/用户态，虚拟内存结构等内容有了进一步的了解。尽管xv6只是一个精简的操作系统，但是它依然让我感到了许多的困难和复杂性，让我对操作系统的认知深入的同时保持一颗敬畏的心。