深 圳 大 学 实 验 报 告

课程名称：操作系统

实验项目名称：实验 1 并发程序设计

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：谭舜泉

报告人：黎浩然 冯海月 学号：2018112061 2018191116

实验时间：2021 年 3 月 23 日-2021 年 4 月 11 日

实验报告提交时间：2021 年 4 月 11 日

教务部制

实验目的与要求：

实验目的：

(1)、掌握计算机操作系统管理进程、处理机、存储器、文件系统的基本方法。

(2)、了解进程的创建、撤消和运行，进程并发执行；自行设计解决哲学家就餐问题的并发线程，了解线程（进程）调度方法；掌握内存空间的分配与回收的基本原理；通过模拟文件管理的工作过程，了解文件操作命令的实质。

(3)、了解现代计算机操作系统的工作原理，具有初步分析、设计操作系统的能力。

(4)、通过在计算机上编程实现操作系统中的各种管理功能，在系统程序设计能力方面得到提升。

实验要求：

(1) 在 xv6 环境下开发应用程序

题目：仿照 echo，写一个命令 echo_reversal，实现以下功能：把输入的每个参数中的字符次序颠倒输出。

例如：$ echo_reversal Hello World!

则输出： olleH !dlroW

(2) 回答以下问题：

Xv6 中并发进程有几种状态，在源码中分别以什么常量代表，试解释每种状态的意义。

Xv6 中 PCB 是以什么方式存放的，链表还是数组？系统最多允许同时运行多少个进程？

Xv6 是否支持多核 cpu? 如果支持的话，是通过哪个数据结构支持的？

系统启动的第一个进程，其入口函数在哪个文件第几行？它主要实现什么功能？（提示：阅读《xv6 中文文档》第 1 章“第一个进程”）

说明：

（1）本次实验课作业满分为 100 分，占总成绩的比例（待定）。

（2）本次实验课作业截至时间 2021 年 4 月 11 日（周日）23:59。

（3）报告正文：请在指定位置填写，本次实验不需要单独提交源程序文件。

（4）个人信息：WORD 文件名中的“姓名”、“学号”，请改为你的姓名和学号；实验报告的首页，请准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“实验报告提交时间”等信息。

（5）提交方式：请在 BLACKBOARD 平台中按时提交；延迟提交不得分。

（6）发现抄袭（包括复制&粘贴整句话、整张图），该次作业记零分。

（7）期末考试阶段补交无效。

(1). 在 xv6 环境下开发应用程序

题目：仿照 echo，写一个命令 echo_reversal，实现以下功能：把输入的每个参数中的字符次序颠倒输出。

例如：$ echo_reversal Hello World!

则输出： olleH !dlroW

在 xv6-public 路径下编写 echo_reversal.c。

```
[root@localhost xv6-public]# pwd
/home/seamoon/qemu/xv6-public
```

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

int
main(int argc, char *argv[])
{
  int i, j;

  for(i = 1; i < argc; i++){
    // reverse the arguments
    char c;
    int len = strlen(argv[i]);
    for(j = 0; j < len/2; j++){
      c = argv[i][j];
      argv[i][j] = argv[i][len-j-1];
      argv[i][len-j-1] = c;
    }

    printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
  }
  exit();
}
```

在此处将echo_reversal每个参数逆置

//---------- 补充：

home > xv6-public > C echo_reversal.c

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

int main(int argc, char *argv[]) {
    int i, j;
    for(i = 1; i < argc; i++) {
        int len = strlen(argv[i]);
        for(j = 0; j < len/2; j++) {
            argv[i][j] ^= argv[i][len-j-1];
            argv[i][len-j-1] ^= argv[i][j];
            argv[i][j] ^= argv[i][len-j-1];
        }
        printf(1, "%s%s", argv[i], i + 1 < argc ? " " : "\n");
    }
    exit();
}
```

//----------

接着修改 Makefile 中的 UPROGS 变量，添加_echo_reversal。然后执行 make，结果如下所示。

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_echo_reversal\

fs.img: mkfs README $(UPROGS)
	./mkfs fs.img README $(UPROGS)
```

```
[root@localhost xv6-public]# make
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer
-fno-stack-protector      -c -o echo_reversal.o echo_reversal.c
ld -m       elf_i386 -N -e main -Ttext 0 -o _echo_reversal echo_reversal.o ulib.o usys.o printf.o umalloc.o
objdump -S _echo_reversal > echo_reversal.asm
objdump -t _echo_reversal | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > echo_reversal.sym
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests
_wc _zombie _echo_reversal
used 29 (bit 1 ninode 26) free 29 log 10 total 1024
balloc: first 414 blocks have been allocated
balloc: write bitmap block at sector 28
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0295785 s, 173 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 8.3553e-05 s, 6.1 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
248+1 records in
248+1 records out
127012 bytes (127 kB) copied, 0.000382451 s, 332 MB/s
rm echo_reversal.o
```

第六行显示的命令使用 mkfs 程序生成 fs.img 磁盘影像文件，其中可执行文件列表的最后一个就是我们刚生成的_echo_reversal。

启动 xv6 系统后，执行 echo_reversal 程序，输出结果达到预期。

```
cpu1: starting
cpu0: starting
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 1929
cat            2 3 9576
echo           2 4 9100
forktest       2 5 5888
grep           2 6 10716
init           2 7 9432
kill           2 8 9100
ln             2 9 9092
ls             2 10 10668
mkdir          2 11 9168
rm             2 12 9152
sh             2 13 16444
stressfs       2 14 9644
usertests      2 15 37468
wc             2 16 9904
zombie         2 17 8880
echo_reversal  2 18 9440
console        3 19 0
$ echo_reversal hello world
olleh dlrow
$ echo_reversal a0a0 a1a1 a2a2
0a0a 1a1a 2a2a
$
```

(2). 回答以下问题：

Xv6 中并发进程有几种状态，在源码中分别以什么常量代表，试解释每种状态的意义。

Xv6 中 PCB 是以什么方式存放的，链表还是数组？系统最多允许同时运行多少个进程？

Xv6 是否支持多核 cpu？如果支持的话，是通过哪个数据结构支持的？

系统启动的第一个进程，其入口函数在哪个文件第几行？它主要实现什么功能？（提示：阅读《xv6 中文文档》第 1 章“第一个进程”）

1. 并发进程有以下几种状态（参见 proc.h）：

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

① UNUSED

进程未被使用。在 proc 的表中标记为 UNUSED 的槽位；实际上此时并没有对应的进程存在，它仅仅标记一个 proc 的表的槽位未被使用。

② EMBRYO

进程初始态。userinit()和 fork()系统调用会调用 alloproc()在 proc 的表中寻找一个标记为 UNUSED 的槽位，并将其设置为 EMBRYO；该状态表示操作系统正在为其分配系统资源、初始化 PCB。

```
34  static struct proc*
35  allocproc(void)
36  {
37    struct proc *p;
38    char *sp;
39
40    acquire(&ptable.lock);
41    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
42      if(p->state == UNUSED)
43        goto found;
44    release(&ptable.lock);
45    return 0;
46
47  found:
48    p->state = EMBRYO;
49    p->pid = nextpid++;
50    release(&ptable.lock);
51
52    // Allocate kernel stack.
53    if((p->kstack = kalloc()) == 0){
54      p->state = UNUSED;
55      return 0;
56    }
57    sp = p->kstack + KSTACKSIZE;
```

EMBRYO 是一个过渡状态，紧接着 alloproc()会尝试为这个进程分配一个独有的 pid，并且为其分配内核栈。如果分配失败了，会将槽位又设置为 UNUSED 并返回 0 表示失败；如果接下来的一系列初始化都成功，那么该进程最终会进入 RUNNABLE 状态。

③ SLEEPING

进程进入睡眠状态。RUNNING 状态的进程通过调用 sleep 使得进程转化为 SLEEPING 状态并以释放 CPU，wakeup 调用会寻找一个睡眠进程并将其标志位 RUNNABLE。处于 SLEEPING 状态的进程不会被调度；该状态又被称为阻塞态，指进程不具备运行的条件，正在等待某个事件的完成。

④ RUNNABLE

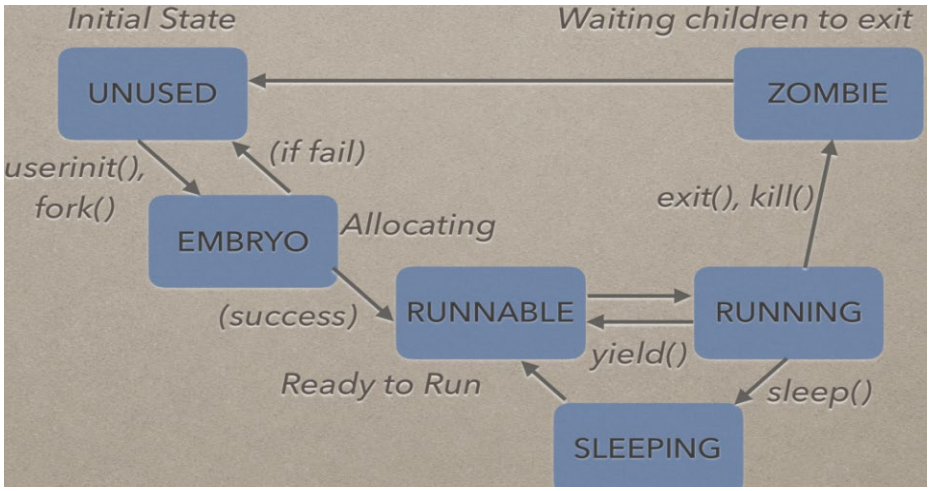
进程就绪状态。此时进程等待获得 CPU 时间片，进程能够被操作系统所调度。

⑤ RUNNING

进程运行中。此时进程正占用 CPU 时间片，CPU 正在执行该进程的代码。

⑥ ZOMBIE

僵尸状态。此时进程已经结束运行，等待父进程通过调用 wait 回收资源。wait 接着会重新查看进程表并找到 state == ZOMBIE 的已退出子进程。它会记录该子进程的 pid 然后清理其 struct pro，释放相关的内存空间。



2. PCB 的内容在 proc.h 中有以下定义：

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

其中，PCB 是以数组的方式存放的，在 proc.c 中有如下源码。

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

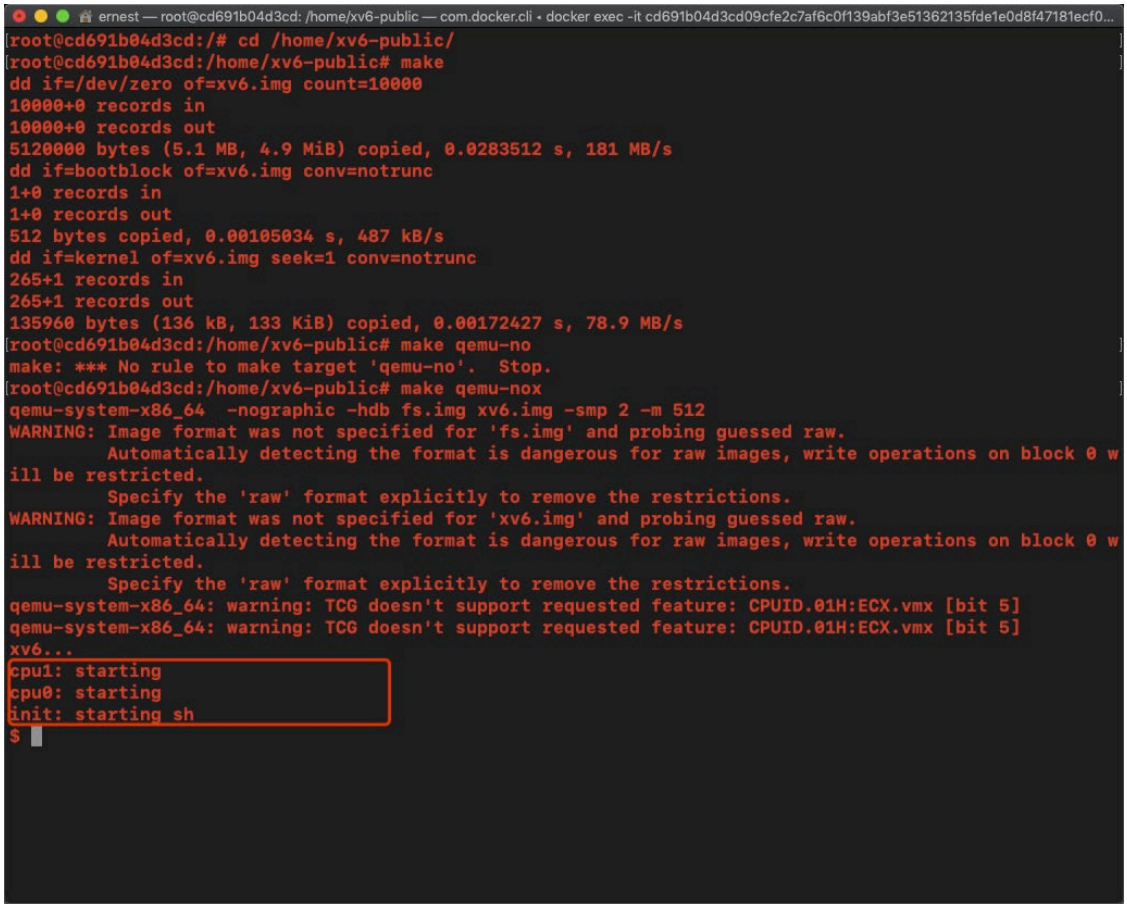
可以看到，NPROC 定义了系统最多同同时运行的进程数，在 param.h 文件中，NPROC 被定义为 64。

```
param.h:1:#define NPROC        64  // maximum number of processes
```

3. mpmain 中的 mp 是多处理器（MultiProcessor）的英文简写。它是操作系统的 main 函数中的最后一个函数，它的代码如下：

```
55  // Common CPU setup code.
56  static void
57  mpmain(void)
58  {
59    cprintf("cpu%d: starting\n", cpu->id);
60    idtinit();       // load idt register
61    xchg(&cpu->started, 1); // tell startothers() we're up
62    scheduler();     // start running processes
63  }
```

可以看到他输出了一个 CPU 的信息。并且，在编译并运行 xv6 的时候输出的信息表示有两个 CPU 在工作：



此外，Xv6 支持多核 cpu，在 proc.c 中定义，通过数组结构实现，其中 NCPU=8，表示 xv6 系统最多支持 8 个 cpu。

```
extern struct cpu cpus[NCPU];
extern int ncpu;
```

4. 入口函数在 main.c 的第 42 行，如下。

```c
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  // Finish setting up this processor in mpmain.
  mpmain();
```

在 main.c 中查看 xv6 操作系统的主函数，main 函数首先调用了一系列名字带有 init 的函数初始化 BIOS 和子系统等，然后通过 userinit 函数创建第一个进程，注释中说明这是第一个（用户）进程的创建。查看 userinit 函数的代码：

```c
//PAGEBREAK: 32
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
}
```

如前面所说，调用 alloproc()从 proc.h 申请一个处于 UNUSED 的进程槽位，并在 alloproc()中对其进行初始化，设置进程号 pid 为 1。然后，userinit 继续进行下列工作：

1. 调用 setupkvm 分配第一个进程的内核栈，并设置其大小为 KSTACKSIZE，即 4096 字节

2. 第一个进程会先运行一段程序，这段程序由汇编写成（initcode.s）。不过在此之前需要找到物理内存来存放这段代码，并设置页表来指向那一段内存。_binary_initcode_start 和 _binary_initcode_size 表示了这段二进制码的位置和大小。

3. 然后 userinit 调用 inituvm。分配一页物理内存，并将虚拟地址 0 指向这段内存中，然后设置 TrapFrame 结构体。TrapFrame 结构体保存着从内核态返回用户态的时候需要恢复的寄存器等快照，也就是 initcode.s 刚开始执行时的寄存器状态。

```
182  inituvm(pde_t *pgdir, char *init, uint sz)
183  {
184    char *mem;
185
186    if(sz >= PGSIZE)
187      panic("inituvm: more than a page");
188    mem = kalloc();
189    memset(mem, 0, PGSIZE);
190    mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
191    memmove(mem, init, sz);
192  }
```

4. 接下来设置 TrapFrame 中的寄存器，第一个进程需要将 TrapFrame 设置为用户态，%cs 寄存器保存着特权级别，所以需要设置为 DPL_USER，%ds 寄存器也需要设为该特权级别。接着还需要设置中断有关的寄存器 eflags 表示为允许硬件中断（FL_IF）。

5. 接着需要设置栈指针 esp 为进程的第一页的最大有效虚拟地址。程序计数器 eip 指向初始化代码入口，即地址 0。然后设置首个进程的名字为"initcode"、工作目录为根目录下。

```
  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
```

6. 将进程状态修改为就绪态，等待进程被调度。

经过一系列的操作之后，第一个进程已经创建完毕了。接下来由 mpmain 调用 scheduler 函数来最终调度第一个进程，执行第一个进程最开始的代码 initcode.s。

```
55  // Common CPU setup code.
56  static void
57  mpmain(void)
58  {
59    cprintf("cpu%d: starting\n", cpu->id);
60    idtinit();       // load idt register
61    xchg(&cpu->started, 1); // tell startothers() we're up
62    scheduler();     // start running processes
63  }
```

Initcode.s 代码如下。

```
# Initial process execs /init.

#include "syscall.h"
#include "traps.h"


# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

第一段代码触发 exec 系统调用，将从文件系统中获取的 /init 的二进制代码代替 initcode 的代码。如果 exec 失败并且返回了，initcode 会不断调用一个不会返回的系统调用 exit。接着查看 init.c 的代码，如下。

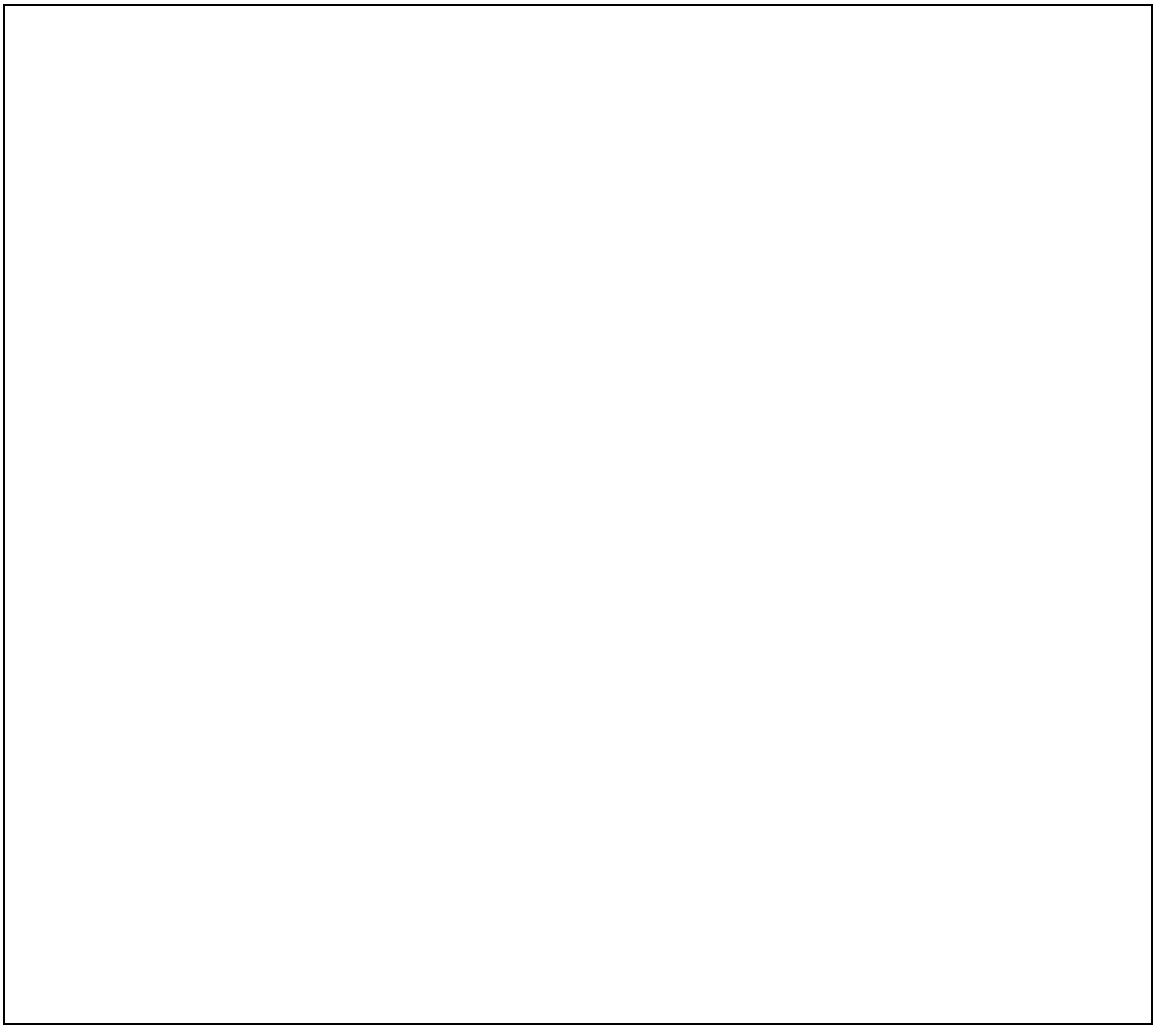
```
// init: The initial user-level program

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

char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

init 创建一个新的控制台设备文件，然后把它作为描述符 0，1，2 打开。接下来它将不断循环，开启控制台 shell，处理没有父进程的僵尸进程，直到 shell 退出，然后再反复。

++

其他（例如感想、建议等等）。

通过本次试验，初步了解了 xv6 操作系统内核的基本框架以及所使用的数据结构，如 proc 的表和 TrapFrame 结构，用户态和内核态的转换、锁的机制。知道如何编译、运行一个 xv6 内核。通过编写一个小程序懂得如何在 xv6 环境下编写并运行程序，从 xv6 第一个进程的创建到第一个进程的运行所经历的过程。



指导教师批阅意见：   成绩评定：   指导教师签字：谭舜泉 2021 年 4 月 18 日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。