

华东师范大学数据学院上机实践报告

课程名称： 操作系统

年级： 2022 级

上机实践成绩：

指导教师： 翁楚良

姓名： 李芳

上机实践名称： 彩票调度

学号： 10214602404

上机实践日期： 2024.3.28

一、实验目的

- 熟悉调度程序的工作原理。
- 将调度程序更改为新的彩票调度算法。

二、实验内容

实现彩票调度：根据进程持有的彩票数量的比例，为每个运行的进程分配 CPU。每个时间片，随机抽签决定彩票的赢家；赢家是在那个时间片上运行的过程。进程拥有的彩票越多，运行的次数就越多。

三、使用环境

xv6

四、实验过程及结果

- 准备过程：

下载 <https://github.com/mit-pdos/xv6-public.git> 中的压缩包，并将助教给的三个代码文件 (pstat.h、rand.c、rand.h) 压缩包也上传到该文件夹下

- 知识储备：

- 用户空间&内核空间：

- ◆ 区别：

用户空间：用于执行用户程序的地址空间。在用户空间中，用户程序可以访问其拥有的部分内存空间和系统提供的用户态库，但无法直接访问内核空间的资源。

内核空间：用于执行操作系统内核的地址空间。在内核空间中，操作系统内核可以访问系统的全部资源，包括硬件设备、系统数据结构等。

- ◆ 权限：

用户空间：运行在低特权级别，用户程序只能访问自己的地址空间和一些受限制的系统资源，

无法直接访问内核空间。

内核空间：运行在高特权级别，具有完全的系统访问权限，可以直接访问和管理系统的所有资源。

◆ 联系：

1. 系统调用：用户程序通过系统调用接口向内核空间请求系统提供的服务和功能。当用户程序需要操作系统提供的特权功能时，它会通过系统调用将请求发送到内核空间，并等待内核空间返回结果。

2. 中断和异常：硬件设备和异常事件（如页面错误、除零错误等）会导致处理器从用户空间切换到内核空间执行相应的中断处理程序或异常处理程序。

3. 共享数据：用户程序和内核之间通过共享数据来进行通信。例如，用户程序通过系统调用将参数传递给内核，内核执行完操作后将结果返回给用户程序。

■ 重要文件作用及联系：

◆ proc.c：用于实现进程管理相关的功能。在 proc.c 中，定义了与**进程相关**的数据结构和函数，包括进程的创建、销毁、调度等操作。其中的代码主要负责实现**内核空间中的进程管理功能**，例如创建新进程的 allocproc() 函数、销毁进程的 kill() 函数等。

◆ sysproc.c：用于实现系统调用相关的功能。在 sysproc.c 中，定义了与**系统调用相关**的处理函数，这些函数将**用户程序发起的系统调用转发给内核空间**进行处理。其中的代码主要负责实现用户空间和内核空间之间的通信，即将用户程序发起的系统调用转换为对应的内核函数调用，并将结果返回给用户程序。

◆ syscall.c：用于实现系统调用的核心处理逻辑。在 syscall.c 中，定义了**系统调用号与系统调用处理函数之间的映射关系**，并提供了一个统一的**入口函数 syscall()** 来处理用户程序发起的系统调用。其中的代码主要负责根据用户程序传入的系统调用号，调用对应的系统调用处理函数，以实现用户程序对操作系统的请求。

◆ syscall.h：定义了 syscall.c 文件里要用到的系统调用函数的系统调用号。

◆ usys.S：提供了一种方便的方式来定义系统调用（SYSCALL 宏），并将系统调用与相应的系统调用函数关联起来。用户程序可以直接调用这些系统调用宏来触发相应的系统调用，而无需直接处理系统调用编号和触发系统调用的指令序列，从而简化了系统调用的使用。

- ◆ **联系：**用户程序调用系统调用函数后，然进入 `usys.S` 汇编语言的汇编文件，系统调用函数名称作为变量，通过系统宏定义代码转化为系统调用号后进入内核，调用对应函数，`syscall.c` 接收系统调用请求并根据 `syscall.h` 中的系统调用号调用相应的处理函数，这些处理函数通常在 `sysproc.c` 中实现；`sysproc.c` 中的系统调用处理函数可能需要调用 `proc.c` 中的进程管理函数来完成特定操作。

- 实验操作：

1. 修改 Makefile 文件：

(1) 在 `OBJS` 中添加 `rand.o\`，引入一个新的随机数生成模块或相关功能，以提供随机性的支持。

```
uart.o\
vectors.o\
vm.o\
rand.o\
```

(2) 在 `UPROGS` 中添加 `_testTicket\` 以及 `_testProcinfo\`，为了实现测试彩票调度算法的实现以及进程相关信息的查看。

```
1 _stressfs\
2 _usertests\
3 _wc\
4 _zombie\
5 _testTicket\
6 _testProcInfo\
7
```

2. **编辑 `proc.h` 和 `param.h` 文件：**在 `proc.h` 在文件中的 `proc` 进程结构体中添加 `ticket` 进程彩票数、`tick` 时间片数据，在 `param.h` 文件中 `#define InitialTickets 5` 这一初始进程彩票数的结构体。

```
C proc.h X
C proc.h > proc
37 // Per-process state
38 struct proc {
39     uint sz; // Size of process memory
40     pde_t* pgdir; // Page table
41     char *kstack; // Bottom of kernel stack
42     enum procstate state; // Process state
43     int pid; // Process ID
44     struct proc *parent; // Parent process
45     struct trapframe *tf; // Trap frame for current process
46     struct context *context; // switch() here to run process
47     void *chan; // If non-zero, sleeping on channel
48     int killed; // If non-zero, have been killed
49     struct file *ofile[NFILE]; // Open files
50     struct inode *cwd; // Current directory
51     char name[16]; // Process name (debugging)
52     int ticket; //彩票数
53     int tick; //时间片数
54 };
```

```
C param.h X
C param.h > LOGSIZE
1 #define NPROC 64 // maximum number of processes
2 #define KSTACKSIZE 4096 // size of per-process kernel stack
3 #define NCPU 8 // maximum number of CPUs
4 #define NOFILE 16 // open files per process
5 #define NFILE 100 // open files per system
6 #define NINODE 50 // maximum number of active inodes
7 #define NDEV 10 // maximum major device number
8 #define ROOTDEV 1 // device number of file system root
9 #define MAXARG 32 // max exec arguments
10 #define MAXOPBLOCKS 10 // max # of blocks any FS op can allocate
11 #define LOGSIZE ((MAXOPBLOCKS*3) // max data blocks for log
12 #define NBUF ((MAXOPBLOCKS*3) // size of disk buffer
13 #define FSSIZE 1000 // size of file system in blocks
14 #define InitialTickets 5 //设置默认彩票数值
15
```

3. 编辑 `proc.c` 文件：

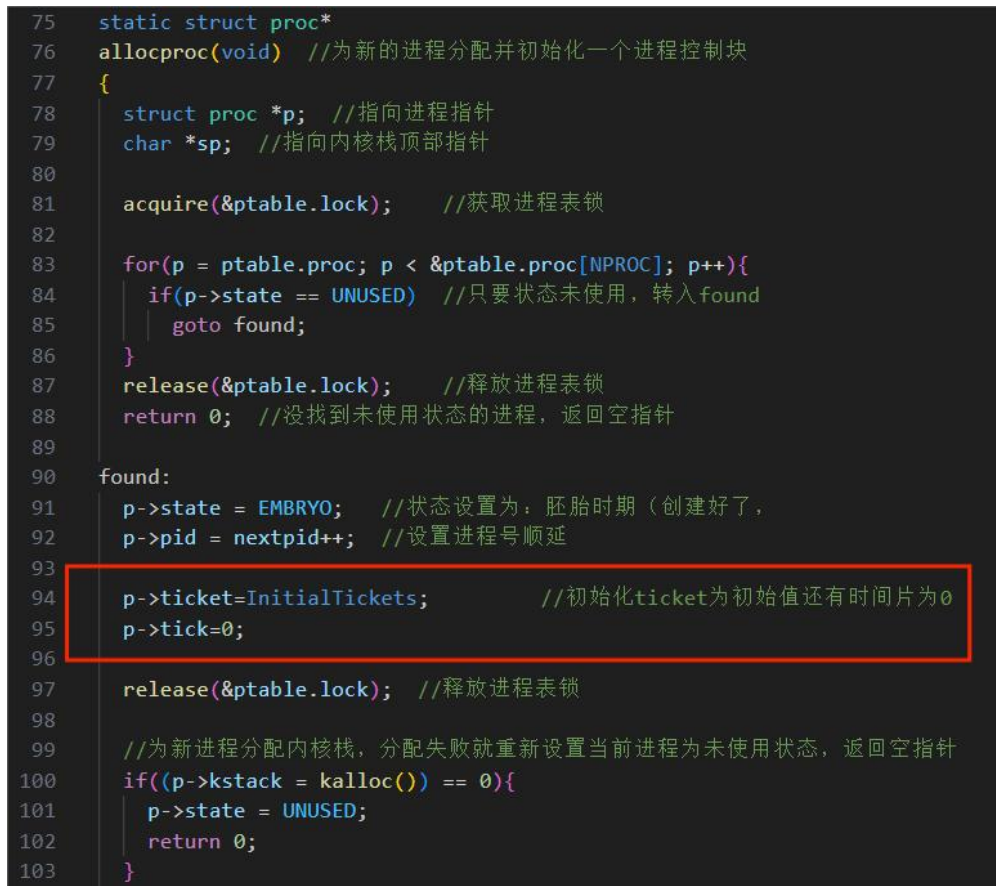
(1) 添加头文件#include "rand.h", 因为后续要用到随机数生成。



```
C proc.c X
C proc.c > settickets(int)
1  #include "types.h"
2  #include "defs.h"
3  #include "param.h"
4  #include "memlayout.h"
5  #include "mmu.h"
6  #include "x86.h"
7  #include "proc.h"
8  #include "spinlock.h"
9  #include "pstat.h"
10 #include "rand.h"
11
```

(2) 修改 static struct proc* allocproc(void)函数:

- ① 添加初始化彩票数和时间片值代码: 在 found:体中添加 p->ticket = InitiaTickets; p->tick = 0;
- ② allocproc 函数: 在操作系统内核中分配一个进程控制块给新创建的进程, 同时为其分配内核栈并初始化必要的上下文信息, 以便该进程可以被调度执行, 结果返回指向新分配的进程控制块的指针。



```
75 static struct proc*
76 allocproc(void) //为新的进程分配并初始化一个进程控制块
77 {
78     struct proc *p; //指向进程指针
79     char *sp; //指向内核栈顶部指针
80
81     acquire(&ptable.lock); //获取进程表锁
82
83     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
84         if(p->state == UNUSED) //只要状态未使用, 转入found
85             goto found;
86     }
87     release(&ptable.lock); //释放进程表锁
88     return 0; //没找到未使用状态的进程, 返回空指针
89
90 found:
91     p->state = EMBRYO; //状态设置为: 胚胎时期 (创建好了,
92     p->pid = nextpid++; //设置进程号顺延
93
94     p->ticket=InitialTickets; //初始化ticket为初始值还有时间片为0
95     p->tick=0;
96
97     release(&ptable.lock); //释放进程表锁
98
99     //为新进程分配内核栈, 分配失败就重新设置当前进程为未使用状态, 返回空指针
100    if((p->kstack = kalloc()) == 0){
101        p->state = UNUSED;
102        return 0;
103    }
```

```

105  sp = p->kstack + KSTACKSIZE; //新进程的内核栈指针指向内核栈顶部
106  // Leave room for trap frame.
107  sp -= sizeof *p->tf; //为新进程trap frame分配空间
108  p->tf = (struct trapframe*)sp; //发生异常或中断时保存处理器状态的数据结构,以便在异常或中断处理完成后能够正确地恢复程序的执行
109
110  // Set up new context to start executing at forkret,
111  // which returns to trapret.
112  //设置执行上下文
113  sp -= 4; //sp指针下移,为指令地址预留空间
114  *(uint*)sp = (uint)trapret; //sp指向trapret函数地址,该地址作为新进程的context的eip寄存器的初始值,新进程从此开始
115
116  sp -= sizeof *p->context; //sp指针下移一个context结构体大小,为新进程执行上下文结构体context分配空间
117  p->context = (struct context*)sp; //context指向当前sp地址
118  memset(p->context, 0, sizeof *p->context); //使用memset函数将context结构体初始为0
119  p->context->eip = (uint)forkret; //context的eip寄存器存储forkret函数返回地址,接下来继续执行父进程
120
121  return p;
122 }
123

```

(3) 修改 int fork(void)函数:

- ① 在第二个 if 判断完成后,让子进程继承父进程的彩票数: np->ticket = curproc->ticket;
- ② fork 函数: 创建一个新的子进程,子进程会复制父进程的执行环境、内存映射和文件描述符等信息,并将子进程设置为可运行状态。最后,它返回子进程的进程 ID 给父进程,如果创建失败则返回 -1。

```

186  int
187  fork(void)
188  {
189      int i, pid;
190      struct proc *np; //指针指向新进程
191      struct proc *curproc = myproc(); //指针指向myproc()返回的当前CPU进程地址
192      if((np = allocproc()) == 0){ //调用allocproc函数尝试给新进程分配进程控制块,分配失败返回-1
193          return -1;
194      }
195
196      if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){ //复制当前进程用户地址空间给新进程
197          kfree(np->kstack); //复制失败时释放新进程占用资源,状态设置为未使用,返回-1
198          np->kstack = 0;
199          np->state = UNUSED;
200          return -1;
201      }
202      np->sz = curproc->sz; //新进程大小设置为当前进程大小
203      np->parent = curproc; //新进程父进程指针指向当前进程
204      *np->tf = *curproc->tf; //父进程的trapframe复制给新进程
205      np->ticket = curproc->ticket; //新进程继承当前进程彩票数
206
207      np->tf->eax = 0; //初始新进程eax寄存器为0.
208
209      //文件操作: 遍历当前进程的文件描述符数组,为新进程复制文件描述符表和当前工作目录
210      for(i = 0; i < NOFILE; i++)
211          if(curproc->ofile[i])
212              np->ofile[i] = filedup(curproc->ofile[i]);
213      np->cwd = idup(curproc->cwd);
214
215      safestrcpy(np->name, curproc->name, sizeof(curproc->name)); //安全复制当前进程名字作为新进程名字
216      pid = np->pid; //获取新进程ID
217      acquire(&table.lock); //获取进程表锁
218      np->state = RUNNABLE; //当前进程设置为可运行状态
219      release(&table.lock); //释放进程表锁
220      return pid; //返回新进程ID
221  }
222

```

(4) 添加 int getRunnableProcTickets(void)函数:

- ① 首先定义了一个指向进程控制块的指针变量 `pre`，以及一个初始化为 0 的整型变量 `totalticket`，用于存储可运行进程的总彩票数。接着利用 `for` 循环语句，遍历进程表中的所有进程，只要满足状态为可运行状态。就将其彩票数加入到 `totalticket` 中，最后返回所有可运行进程的总彩票数。
- ② `getRunnableProcTickets` 函数：遍历进程表中的所有进程，计算所有可运行进程的总彩票数，并将其作为返回值返回。

```

322 int
323 getRunnableProcTickets(void){ //统计所有可运行进程的彩票数
324     struct proc *pre;
325     int totalticket=0;
326     for(pre=ptable.proc;pre<&ptable.proc[NPROC];pre++){
327         if(pre->state==RUNNABLE){
328             totalticket+=pre->ticket;
329         }
330     }
331     return totalticket;
332 }
333

```

(5) 添加 `int settickets(int num)` 函数：

- ① 首先获取当前调用该函数的进程的进程控制块，并获取其进程 ID，接着获取进程表的锁，确保在修改进程信息时不会被其他线程干扰。然后利用 `for` 循环遍历进程表中的所有进程，如果找到了与当前进程 ID 相匹配的进程，则将该进程的彩票数设置为给定的 `num` 值，并释放进程表的锁，然后返回 0 表示成功执行；如果没有找到匹配的进程，则释放进程表的锁，并返回 0 表示成功执行。
- ② `settickets` 函数：用于设置调用该函数的进程的彩票数为给定的 `num` 值。

```

557 //由内核调用，用于管理和操作系统中的进程。
558 int
559 settickets(int num){
560     struct proc *pr=myproc(); //指向当前CPU正在执行进程
561     int myprocpid=pr->pid; //获取进程pid
562     acquire(&ptable.lock); //获取进程表锁，遍历之前，锁住进程表
563     struct proc *pre;
564     for(pre=ptable.proc;pre<&ptable.proc[NPROC];pre++){ //遍历所有进程
565         if(pre->pid==myprocpid){ //找到myproc进程
566             pre->ticket=num; //设置进程彩票数
567             release(&ptable.lock); //设置完对应进程彩票数之后，释放锁，直接退出程序
568             return 0;
569         }
570     }
571     release(&ptable.lock); //防止没找到对应进程，无法设置彩票数情况下，锁一直未释放
572     return 0;
573 }

```

(6) 添加 `int getpinfo(struct pstat*ps)` 函数：

- ① 首先获取进程表的锁，确保在获取进程信息时不会被其他线程干扰，再定义一个指向进程控制块的指针变量 `pre`，以及一个计数器 `i`，用于遍历进程表并记录进程信息的索引。相同道理遍历进程表中的所有进程。将当前进程 `p` 的进程 ID、是否在使用中的状态、彩票数以及已运行的时间片数分别存储到 `struct pstat` 结构体 `ps` 对应位置的数组中，并递增计数器 `i`。最后，释放进程表的锁，允许其他线程访问进程表，返回 0 表示成功执行。
- ② `getpinfo` 函数：获取系统中所有进程的相关信息，包括进程 ID、状态、彩票数以及已运行的时间片数，并填充到传入的 `struct pstat` 结构体中。

```

575 int
576 getpinfo(struct pstat *ps){
577     acquire(&ptable.lock);    //获取进程表锁，遍历之前，锁住进程
578     struct proc *pre;         //创建遍历承接当前所有进程的结构体
579     int i=0;
580     for(pre=ptable.proc;pre<&ptable.proc[NPROC];pre++){
581         ps->pid[i]=pre->pid;    //承接pid
582         ps->inuse[i]=(pre->state!=UNUSED); //承接进程状态。因为pstat.h声明在使用为1，未使用为0；
583         ps->ticket[i]=pre->ticket; //承接票数
584         ps->tick[i]=pre->tick;   //承接时间片
585         i++;
586     }
587     release(&ptable.lock);    //释放进程表锁
588     return 0;
589 }

```

(7) 修改 void scheduler(void)函数：

- ① 首先定义了指向进程控制块的指针变量 `p`，以及指向当前 CPU 的指针变量 `c`，并将当前 CPU 上运行的进程置为空。接着，利用 `for(;;) { ... }` 语句实现无限循环，持续进行进程调度，紧接着，利用 `sti();` 启用处理器的中断，允许中断发生。接下来，就开始对进程进行操作：第一步、获取进程表的锁，确保在遍历进程表时不会被其他线程干扰。第二步、获取所有可运行进程的总彩票数，并生成一个随机数 `wticket`，范围在 0 到总彩票数之间，用于选择一个胜出的进程。第三步、遍历进程表中的所有进程：如果进程是可运行状态，就把其彩票数累加到 `totalticket` 变量中，否则继续下一个进程的检查。**如果当前彩票累加值超过了随机生成的胜出彩票数，则选择该进程进行调度：**将当前 CPU 的运行进程设置为选择的进程 `p`，切换到该进程的用户虚拟内存空间，并将其状态设置为运行中，并且要记录进程运行的时间片数量，并更新进程的 `tick` 数量，这之后再切换回内核虚拟内存空间，并将当前 CPU 的运行进程重置为空。最后释放进程表的锁，允许其他线程访问进程表。
- ② `scheduler` 函数：是负责彩票调度逻辑的代码，根据进程的彩票数来选择一个胜出的进程进行调

度，并记录其运行时间片。

```
C proc.c > scheduler(void)
334 void
335 scheduler(void) //scheduler()是彩票调度的逻辑代码，从可运行的进程中选择出来一个进程运行
336 {
337     struct proc *p; //指针变量指向进程控制块
338     struct cpu *c = mycpu(); //指向当前CPU的指针变量
339     c->proc = 0; //将当前CPU运行进程置空
340
341     for(;;){ //无限循环，持续进行进程调度
342         sti(); //用于将处理器的中断标志为1 表示允许中断
343         long ctotat=0; //存放当前彩票累加值
344         acquire(&ptable.lock); //获取进程表锁
345         long total=getRunnableProcTickets()*1LL; //获取可运行进程的总彩票数，1LL代表长长整型1
346         long wticket=random_at_most(total); //随机数，用来选择胜出的进程
347
348         for(p=ptable.proc;p<&ptable.proc[NPROC];p++){ //遍历进程表所有进程
349             if(p->state==RUNNABLE){ //可运行状态进程，就累加彩票数
350                 ctotat+=p->ticket;
351             }else{
352                 continue;
353             }
354             if(ctotat>wticket){ //当当前累加数超过胜出彩票数时，进行进程调度
355                 c->proc=p; //切换CPU正在运行的进程为胜出进程
356                 switchvm(p); //切换到胜出进程的用户虚拟空间
357                 p->state=RUNNING; //修改胜出进程状态为运行中
358
359                 int tickstart=ticks; //记录胜出进程开始运行时间
360                 swtch(&(c->scheduler),p->context);
361                 int tickend=ticks; //记录胜出进程结束运行时间
362                 p->tick+=(tickend-tickstart); //更新胜出进程时间片数
363
364                 switchkvm(); //切换回内核虚拟内存空间
365                 c->proc=0; //CPU正在运行进程置空
366                 break;
367             }else{continue;}
368         }
369         release(&ptable.lock); //释放进程表锁，允许线程访问
370     }
371 }
```

4. 编辑 defs.h 和 user.h 文件：在 defs.h 的 proc.c 目录下添加函数声明：int settickets(int);以及 int getpinfo(struct pstat*)，并且由于 getpinfo 用到了结构体 pstat，所以要在该文件最上面声明 struct pstat;。在 user.h 的 syscall 目录下，进行相同的函数和结构体声明操作。

```
C defs.h > ...
1 struct buf;
2 struct context;
3 struct file;
4 struct inode;
5 struct pipe;
6 struct proc;
7 struct rtcdate;
8 struct spinlock;
9 struct sleeplock;
10 struct stat;
11 struct superblock;
12 struct pstat;
13
14 // bio.c
15 void binit(void);
```

```
95 //PAGEBREAK: 16
96 // proc.c
97 int cpuid(void);
98 void exit(void);
99 int fork(void);
100 int growproc(int);
101 int kill(int);
102 struct cpu* mycpu(void);
103 struct proc* myproc();
104 void pinit(void);
105 void procdump(void);
106 void scheduler(void) __attribute__((noreturn));
107 void sched(void);
108 void setproc(struct proc*);
109 void sleep(void*, struct spinlock*);
110 void userinit(void);
111 int wait(void);
112 void wakeup(void*);
113 void yield(void);
114
115 //定义内核的一些常量、类型、全局变量以及一些系统级别的函数声明
116 int settickets(int);
117 int getpinfo(struct pstat*);
```



```

C user.h
C user.h > ...
1 struct stat;
2 struct rtdat;
3 struct pstat; //声明后面要用到的pstat结构体
4
5 // system calls
6 int fork(void);
7 int exit(void) __attribute__((noreturn));
8 int wait(void);
9 int pipe(int*);
10 int write(int, const void*, int);
11 int read(int, void*, int);
12 int close(int);
13 int kill(int);
14 int exec(char*, char**);
15 int open(const char*, int);
16 int mknod(const char*, short, short);
17 int unlink(const char*);
18 int fstat(int fd, struct stat*);
19 int link(const char*, const char*);
20 int mkdir(const char*);
21 int chdir(const char*);
22 int dup(int);
23 int getpid(void);
24 char* sbrk(int);
25 int sleep(int);
26 int uptime(void);
27 //定义用户空间的系统调用接口和用户程序需要使用的一些常量、类型和函数声明
28 int settickets(int); //声明函数
29 int getpinfo(struct pstat*);
30

```

5. 编辑 sysproc.c 文件：（注意添加相应头文件：pstat.h）

(1) 添加 int sys_settickets(void) 函数：用于设置进程的票数（tickets），从而影响进程在调度时的优先级。

- ① 首先从用户程序传递的参数中获取要设置的票数值，并进行合法性检查：如果传递的票数小于 0，则返回错误码 -1；如果传递的票数为 0，则将进程的票数设置为初始化值（可能是 InitiaTickets）；否则，将进程的票数设置为传递的值。
- ② 最后，函数返回设置操作的结果，0 表示成功，-1 表示失败。

(2) 添加 int sys_getpinfo(void) 函数：用于获取系统中所有活跃进程的信息，例如它们的 PID（进程 ID）、状态等。

- ① 首先从用户程序传递的参数中获取一个指向 struct pstat 结构的指针，该结构用于保存进程信息。如果参数获取失败，则返回错误码 -1，否则，调用 getpinfo() 函数来获取系统中活跃进程的信息，并将结果保存在传入的 struct pstat 结构中。
- ② 最后，函数返回 0 表示成功。

```

93 //用户空间系统调用实现，用户程序调用的接口，将用户程序的系统调用请求传递给内核处理，调用proc.c中内核管理进程的函数
94
95 int sys_settickets(void){ //传递settickets参数
96     int number;
97     argint(0,&number); //从用户空间获取系统调用的第一个参数，并将其存储到number变量中
98     if(number<0){
99         return -1;    //无意义
100     }
101     else if(number==0){
102         return(settickets(InitialTickets)); //默认值
103     }
104     else{
105         return settickets(number);
106     }
107 }
108
109 int sys_getpinfo(void){
110     struct pstat *ps;
111     //往ps指向的位置存储系统调用参数中提取的第一个参数，(char**)&ps用来把ps地址转换成char**类型的指针
112     if(argptr(0,(char**)&ps,sizeof(struct pstat))<0){ //提取参数失败，参数无效
113         return -1;
114     }
115     getpinfo(ps);
116     return 0;
117 }

```

6. 编辑 syscall.c、syscall.h 以及 usys.S 文件:

- (1) syscall.c 中添加函数声明: extern int sys_settickets(void);和 extern int sys_getpinfo(void);, 并且在 static int (*syscalls[])(void) = {}中添加[SYS_settickets] sys_settickets,[SYS_getpinfo] sys_getpinfo,

```

105     extern int sys_uptime(void);
106     extern int sys_settickets(void);
107     extern int sys_getpinfo(void);
108
109     //从当前进程控制块中获取系统调用号码，然后根据该号码执行相应的系统调用函数。
110     static int (*syscalls[])(void) = {
111         [SYS_fork]    sys_fork,
112         [SYS_exit]    sys_exit,
113         [SYS_wait]    sys_wait,
114         [SYS_pipe]    sys_pipe,
115         [SYS_read]    sys_read,
116         [SYS_kill]    sys_kill,
117         [SYS_exec]    sys_exec,
118         [SYS_fstat]    sys_fstat,
119         [SYS_chdir]    sys_chdir,
120         [SYS_dup]    sys_dup,
121         [SYS_getpid]    sys_getpid,
122         [SYS_sbrk]    sys_sbrk,
123         [SYS_sleep]    sys_sleep,
124         [SYS_uptime]    sys_uptime,
125         [SYS_open]    sys_open,
126         [SYS_write]    sys_write,
127         [SYS_mknod]    sys_mknod,
128         [SYS_unlink]    sys_unlink,
129         [SYS_link]    sys_link,
130         [SYS_mkdir]    sys_mkdir,
131         [SYS_close]    sys_close,
132         [SYS_settickets] sys_settickets, //声明系统调用函数
133         [SYS_getpinfo] sys_getpinfo,
134     };

```

- (2) syscall.h 文件定义两系统调用函数的调用号#define SYS_settickets 22 #define SYS_getpinfo 23

```
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_settickets 22 //定义系统调用函数的系统调用号，供syscall.c使用
24 #define SYS_getpinfo 23
```

(3) Usys.S 文件添加系统调用宏定义：SYSCALL(settickets)和 SYSCALL(getpinfo)

```
31 SYSCALL(uptime)
32 SYSCALL(settickets)
33 SYSCALL(getpinfo)
```

7. 编写 testTicket.c 和 testProcInfo.c 文件:

(1) testTicket.c 用于设置当前进程的彩票数:

- ① #include "types.h"、#include "stat.h"、#include "user.h"这些头文件的引用，包含了所需的类型定义、系统调用声明等。
- ② 首先将命令行参数 argv[1] 转换为整数，并将其赋值给变量，然后调用 settickets 系统调用，将参数作为当前进程的彩票数进行设置。彩票数决定了进程在 CPU 资源竞争中的优先级，票数越多，优先级越高。
- ③ 利用 while(1)这个无限循环，让进程持续运行，直到手动终止。
- ④ 最后调用 exit 系统调用，退出当前进程。

```
C testTicket.c X
C testTicket.c > main(int, char * [])
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int main(int argc, char *argv[])
6 {
7     int number = atoi(argv[1]); // 接收命令行输入彩票数;atoi()函数将字符转化为整数
8     settickets(number);
9
10    while (1)
11    {
12        // 在服务器程序中，可能会有监听连接请求的代码，当有连接请求时，程序会执行相应的操作，
13        // 而当没有请求时，程序可能会休眠一段时间以节省系统资源
14    }
15    exit();
16 }
```

(2) testprocInfo.c 用于获取系统中所有进程的调度信息，并打印出来:

- ① #include "types.h"、#include "stat.h"、#include "user.h": 这些头文件的引用，包含了所需的类型定义、系统调用声明等; #include "pstat.h"、#include "param.h": 这两个头文件用于

获取进程状态信息和参数的定义。

- ② 首先定义一个 count 变量用于计算所有进程的时间片总数，再定义了一个 pstat 结构体变量 ps，用于存储获取的进程状态信息。然后调用 getpinfo 系统调用，获取系统中所有进程的状态信息，并将其保存在 ps 结构体中。
- ③ 接着就开始打印进程信息：
 - 1) 先打印初始分配的调度票数
 - 2) 再打印表头，包括进程 ID、是否可运行、调度票数和已运行的时间片数
 - 3) 接着遍历所有可能的进程 ID：检查进程是否存在且其调度票数大于 0，满足就打印进程的 ID、是否可运行、调度票数和已运行的时间片数，并累加进程的已运行时间片数到 count 变量中。
- ④ 最后调用 exit 系统调用，退出当前进程。

```
testProcInfo.c ×
testProcInfo.c > main(int, char * [])
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #include "pstat.h"
5  #include "param.h"
6
7  int main(int argc, char *argv[])
8  {
9      int count = 0; //计数时间片总数量
10     struct pstat ps; //创建承接结构体，由pstat.h定义
11     getpinfo(&ps); //获取当前所有进程的信息，由proc.c编辑
12
13     printf(1, "\n----- Initially assigned Tickets = %d ----- \n", InitialTickets); //打印默认票数数值，由param.h定义
14     printf(1, "\nProcessID\tRunnable(0/1)\tTickets\t\tTicks\n");
15     for (int i = 0; i < NPROC; i++) //遍历最大进程数，由param.h定义
16     {
17         if (ps.pid[i] && ps.ticket[i] > 0) //只有进程存在并且拥有彩票数大于0才会被打印
18         {
19             printf(1, "%d\t%d\t%d\t%d\t%d\n", ps.pid[i], ps.inuse[i], ps.ticket[i], ps.tick[i]);
20             count += ps.tick[i]; //计数总时间片
21         }
22     }
23     printf(1, "\n----- Total Ticks = %d ----- \n\n", count);
24     exit(); //直接退出程序
25 }
```

● 系统调用过程（以 testTicket.c 为例）：

- 1) 首先，testTicket.c 程序执行后会调用系统调用函数 settickets，此时，会进入到 usys.S 中汇编代码部分，该文件定义了系统调用宏定义 SYSCALL()，所以 settickets 会作为 SYSCALL()函数的变量，通过下面这段汇编代码，根据 settickets 在 syscall.h 中定义的系统调用号去内核空间调用该系统调用函数，并把 settickets 的系统调用号存到进程结构体 eax 寄存器中：

```

#define SYSCALL(name) \

.globl name; \

name: \

    movl $SYS_ ## name, %eax; \ ;将系统调用号码加载到 %eax 寄存器中, SYS_ ## name 宏定义的方式将系统调用函数名转
                                换为系统调用号码

    int $T_SYSCALL; \ ;触发软中断 T_SYSCALL, 以便将控制权交给内核, 执行相应的系统调用处理程序。

    ret ;返回到用户空间

```

- 2) 然后, syscall.c 中的 syscall() 函数从 eax 寄存器中取出 settickets 的系统调用号, 通过该文件定义的 static int (*syscalls[])(void) 去调用函数 sys_settickets()

```

//从当前进程控制块中获取系统调用号码
//调用的系统调用函数在sysproc.c中实现
void
syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax; //从当前进程控制块 (struct proc) 中 eax 寄存器中获取系统调用号码
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) { //检查系统调用号码是否有效, 如果有效则执行对应的系统调用函数, 将返回值存储到进程的寄存器中
        curproc->tf->eax = syscalls[num]();
    } else { //系统调用号码无效或者对应的系统调用函数不存在, 则输出错误信息, 并将返回值设为 -1
        cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}

```

- 3) 接着, sysproc.c 中编辑的 sys_settickets 函数被调用, 通过 argint(0,&number); 从用户空间获取系统调用的第一个 settickets(number) 的 number 参数, 并将其存储到 sys_settickets 定义的 number 变量中, 如果彩票数有实际意义, 就会将该 number 参数传递给内核空间的进程管理函数, 即被 proc.c 文件定义的 settickets 函数。

```

//用户空间系统调用实现, 用户程序调用的接口, 将用户程序的系统调用请求传递给内核处理, 调用proc.c中内核管理进程的函数
int sys_settickets(void) { //传递settickets参数
    int number;
    argint(0, &number); //从用户空间获取系统调用的第一个参数, 并将其存储到number变量中
    if(number < 0) {
        return -1; //无意义
    }
    else if(number == 0) {
        return(settickets(InitialTickets)); //默认值
    }
    else {
        return settickets(number);
    }
}

```

- 4) 最后, proc.c 文件中定义的 settickets 接收到传递来的 num 参数, 开始对正在 CPU 运行的进程进行彩票数的设置。至此, 整个调用过程完成, 彩票数被设置完成。


```
//由内核调用，用于管理和操作系统中的进程。
int
settickets(int num){
    struct proc *pr=myproc(); //指向当前CPU正在执行进程
    int myprocpid=pr->pid; //获取进程pid
    acquire(&ptable.lock); //获取进程表锁，遍历之前，锁住进程表
    struct proc *pre;
    for(pre=ptable.proc;pre<&ptable.proc[NPROC];pre++){ //遍历所有进程
        if(pre->pid==myprocpid){ //找到myproc进程
            pre->ticket=num; //设置进程彩票数
            release(&ptable.lock); //设置完对应进程彩票数之后，释放锁，直接退出程序
            return 0;
        }
    }
    release(&ptable.lock); //防止没找到对应进程，无法设置彩票数情况下，锁一直未释放
    return 0;
}
```

- 实验结果截图：设置了相同彩票数，运行了两回：

```
$ testTicket 10&;testTicket 15&;testTicket 20&;testTicket 25&;testTicket 30&;
$ testProcInfo
----- Initially assigned Tickets = 5 -----
ProcessID    Runnable(0/1)  Tickets    Ticks
1            1              5          2
2            1              5          3
25           1              5          0
5            1              10         2461
7            1              10         2393
16           1              10         116
18           1              15         146
20           1              20         217
22           1              25         231
24           1              30         254
----- Total Ticks = 5823 -----
$
```

```
$ testProcInfo
----- Initially assigned Tickets = 5 -----
ProcessID    Runnable(0/1)  Tickets    Ticks
1            1              5          2
2            1              5          1
16           1              5          0
5            1              10         651
7            1              15         1012
9            1              20         1352
11           1              25         1728
13           1              30         2017
----- Total Ticks = 6763 -----
```

五、总结

通过本次实验，我复习了操作系统一些理论知识，比如：

- 进程调度：理解了彩票调度算法的原理和特点，能够实现 scheduler 调度器以在多个就绪进程中选择下一个运行的进程。
- 系统调用：理解了系统调用的概念和作用，能够在操作系统中实现新的系统调用，类似 settickets 和 getpinfo，以使用户程序可以使用新功能。
- 进程管理：巩固了进程的概念，包括进程的创建、销毁和状态转换等，也学会如何实现进程控制块（PCB）以及相关的数据结构，用于跟踪和管理进程。
- 并发控制：能够处理并发访问共享资源的问题，如临界区、互斥锁、信号量等，确保在多个进程之间正确地共享和同步数据。
- 调试和测试：可以根据助教提供的测试用例来自己编写测试用例来验证实现的正确性。使用调试工具来排查代码中的错误和问题。

在此次实验过程中中，我也遇到了一些错误：

- 对进程状态转换（如就绪、运行、阻塞）的管理不当，导致进程无法正确地被调度和执行：
 - v 在实现 proc.c 文件中的 scheduler 函数时，没有正确处理正在调度的进程运行状态，误将 RUNNING 设置成了 RUNNABLE，导致最后测试一直卡在 testTicket 票数；这一步
- 彩票调度算法的实现，需要确保彩票数量的合法性和公平性，避免某些进程过度占据资源等问题：
 - v 在实现 proc.c 文件中的 fork 函数时，np->ticket = curproc->ticket;后面加上 np->tick= curproc->tick，最后检查代码要求时发现只需要继承父进程的彩票数即可，添加时间片继承，导致逻辑不通，随后修改

总之，这次上机实验我的收获很大：对操作系统调度算法的理解更加深入，特别是彩票调度算法的原理和应用；加深了对系统调用和进程管理的理解，让我能够更好地设计和实现新的系统功能；提高了调试和测试的能力，能够更有效地定位和解决代码中的问题和错误。