

实验总括

本实验将在 xv6 系统中进行修改，实现调整进程占用时间片长度以及不完整的多级反馈队列

实验特点

本实验设计在 xv6 的基础上，着重探究了 xv6 进程调度的相关实现，并且在此之上进行了一定程度的改进，实现了更加复杂的调度方式，加深对 xv6 系统进程调度部分的熟悉以及对多级反馈队列的理解，该实验难度并不高，但要求对 xv6 进程调度有充分了解，同时本实验没有标准的检验程序，而是希望使用者自己编写程序感受效果，同时在此过程中加深对多级反馈队列的理解，本实验也会给这一环节给出自己的样例程序。

具体操作指导

实验一 通过在 PCB 中添加时间片计数器，并修改对应程序以实现调整进程占用时间片长度，编写程序以检验

1. 在 proc.h 的 proc 结构体中添加计数变量

```
83  enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
84
85  // Per-process state
86  struct proc {
87      struct spinlock lock;
88
89      // p->lock must be held when using these:
90      enum procstate state;           // Process state
91      struct proc *parent;            // Parent process
92      void *chan;                     // If non-zero, sleeping on chan
93      int killed;                     // If non-zero, have been killed
94      int xstate;                     // Exit status to be returned to parent's wait
95      int pid;                         // Process ID
96
97      // these are private to the process, so p->lock need not be held.
98      uint64 kstack;                  // Virtual address of kernel stack
99      uint64 sz;                       // Size of process memory (bytes)
100     pagetable_t pagetable;           // User page table
101     struct trapframe *trapframe;     // data page for trampoline.S
102     struct context context;           // swtch() here to run process
103     struct file *ofile[NOFILE];      // Open files
104     struct inode *cwd;                // Current directory
105     char name[16];                   // Process name (debugging)
106     int slot; // 剩余时间片计数
107 };
```

2. 修改 proc.c 中的 allocproc(), 初始化计数 (例子中设置时间片大小为 4)

```

92  static struct proc*
93  allocproc(void)
94  {
95      struct proc *p;
96
97      for(p = proc; p < &proc[NPROC]; p++) {
98          acquire(&p->lock);
99          if(p->state == UNUSED) {
100              goto found;
101          } else {
102              release(&p->lock);
103          }
104      }
105      return 0;
106
107  found:
108      p->pid = allocpid();
109      p->slot=4;//初始化时间片计数
110      // Allocate a trapframe page.
111      if((p->trapframe = (struct trapframe *)kalloc()) == 0){
112          release(&p->lock);
113          return 0;

```

3. 修改 freeproc(), 当释放进程块时将计数归零

```

136  static void
137  freeproc(struct proc *p)
138  {
139      if(p->trapframe)
140          kfree((void*)p->trapframe);
141      p->trapframe = 0;
142      if(p->pagetable)
143          proc_freepagetable(p->pagetable, p->sz);
144      p->pagetable = 0;
145      p->sz = 0;
146      p->pid = 0;
147      p->parent = 0;
148      p->name[0] = 0;
149      p->chan = 0;
150      p->killed = 0;
151      p->xstate = 0;
152      p->state = UNUSED;
153      p->slot=0;//归零计数
154  }

```

4. 在 trap.c 的 usertrap() 和 kerneltrap() 中都有应对 timer interrupt 的语句, 修改它们使得当计数器归零时再放弃 CPU
usertrap()

```

79 // give up the CPU if this is a timer interrupt.
80 if(which_dev == 2){
81     p->slot--;
82     if(p->slot==0){
83         p->slot=4;
84         yield();
85     }
86 }

```

kerneltrap()

```

158 // give up the CPU if this is a timer interrupt.
159 if(which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING){
160     myproc()->slot--;
161     if(myproc()->slot==0){
162         myproc()->slot=4;
163         yield();
164     }
165 }
166
167
168 // the yield() may have caused some traps to occur,
169 // so restore trap registers for use by kernelvec.S's sepc instruction.
170 w_sepc(sepc);

```

5.修改 proc.c 中的 procdump()以显示时间片计数

```

673 void
674 procdump(void)
675 {
676     static char *states[] = {
677         [UNUSED]    "unused",
678         [SLEEPING]  "sleep ",
679         [RUNNABLE]  "runble",
680         [RUNNING]   "run   ",
681         [ZOMBIE]    "zombie"
682     };
683     struct proc *p;
684     char *state;
685
686     printf("\n");
687     for(p = proc; p < &proc[NPROC]; p++){
688         if(p->state == UNUSED)
689             continue;
690         if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
691             state = states[p->state];
692         else
693             state = "???";
694         printf("%d %s %s,剩余时间片%d", p->pid, state, p->name,p->slot);
695         printf("\n");
696     }
697 }

```

6.编写一个 c 程序测试（要求占用较多时间以及产生数个子进程），并在 xv6 中运行它，在运行中多次用 ctrl+p 调用

procdump()来显示效果

```
C tcostlot.c
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4  #include "stddef.h"
5  int
6  main(void){
7      int i,j,k,n;
8      for(i=1;i<=5;i++){
9          if(fork()==0){
10             for(j=1;j<=i*100000;j++){
11                 for(k=1;k<=100000;k++){
12                     n++;
13                 }
14             }
15             printf("s%d\n",i);
16             exit(0);
17         }
18     }
19     for(j=1;j<=600000;j++){
20         for(k=1;k<=100000;k++){
21             n++;
22         }
23     }
24     printf("f\n",i);
25     exit(0);
26 }
```

```
sikai@LAPTOP-0GM9JLJ: ~/xv6-labs-2020
init: starting sh
$ tcostlot

1 sleep  init, 剩余时间片3
2 sleep  sh, 剩余时间片4
3 run    tcostlot, 剩余时间片2
4 runble tcostlot, 剩余时间片4
5 run    tcostlot, 剩余时间片1
6 runble tcostlot, 剩余时间片4
7 runble tcostlot, 剩余时间片4
8 run    tcostlot, 剩余时间片1

1 sleep  init, 剩余时间片3
2 sleep  sh, 剩余时间片4
3 runble tcostlot, 剩余时间片4
4 runble tcostlot, 剩余时间片4
5 run    tcostlot, 剩余时间片2
6 run    tcostlot, 剩余时间片1
7 runble tcostlot, 剩余时间片4
8 runble tcostlot, 剩余时间片4

1 sleep  init, 剩余时间片3
2 sleep  sh, 剩余时间片4
3 runble tcostlot, 剩余时间片4
4 runble tcostlot, 剩余时间片4
5 run    tcostlot, 剩余时间片4
6 run    tcostlot, 剩余时间片4
7 run    tcostlot, 剩余时间片1
8 runble tcostlot, 剩余时间片4

s1
s2
s3
s4
s5
f
```

实验二 该实验旨在实现一个不完整的多级反馈队列，不完整是因为仍使用原先的 PCB 表并在 PCB 中添加新的标记量以说明其在哪个队列中，并不建立真正的队列，这可能导致一些多级反馈队列的性质无法实现，但并不关键。

1. 修改 proc.h 中的 proc 结构体，添加队列变量

```
105     char name[16];                // Process name (debugging)
106     int slot; //剩余时间片计数
107     int no; //处于第几队列
108     //共三个队列，从1到3时间片长分别是4, 8, 12
109 };
110
```

2. 在 allocproc()中添加队列的初始化

```
108     p->pid = allocpid();
109     p->slot=4; //初始化时间片计数
110     p->no=1; //初始化队列
111     // Allocate a trapframe page.
112     if((p->trapframe = (struct trapframe *)kalloc()) == 0){
```

3. 在 trap.c 中修改 timer interrupt 对应语句，实现队列变化与时间片长度变化

usertrap()

```
80     // give up the CPU if this is a timer interrupt.
81     if(which_dev == 2){
82         p->slot--;
83         if(p->slot==0){
84             //p->slot=4;
85             if(myproc()->no<3){
86                 myproc()->no++;
87             }
88             myproc()->slot=grptlength[myproc()->no];
89             yield();
90         }
91     }
92
```

kerneltrap()

```

163 // give up the CPU if this is a timer interrupt.
164 if(which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING){
165     myproc()->slot--;
166     if(myproc()->slot==0){
167         //myproc()->slot=4;
168         if(myproc()->no<3){
169             myproc()->no++;
170         }
171         myproc()->slot=grptlength[myproc()->no];
172         yield();
173     }
174 }
175
176

```

int grptlength[4]={0,4,8,12};

4. 修改 scheduler(), 实现多级反馈队列, 注意到这种方法并没有实现 FCFS,不是严格的多级反馈队列, 但因为新进入的进程属于最高优先度, 而最高优先度的时间片最短, 因而不会造成很严重的饥饿现象。


```

458 void
459 scheduler(void)
460 {
461     struct proc *p,*m;
462     struct cpu *c = mycpu();
463
464     c->proc = 0;
465     for(;;){
466         // Avoid deadlock by ensuring that devices can interrupt.
467         intr_on();
468
469         int nproc = 0;
470         int key=99;//记录进程中最高优先级
471         for(p = proc; p < &proc[NPROC]; p++) {
472             acquire(&p->lock);
473             if(p->state != UNUSED) {
474                 nproc++;
475             }
476             //当每次切换进程时扫描一次PCB表，找到最高优先级
477             if(key==99){
478                 for(m = proc; m < &proc[NPROC]; m++){
479                     if(m->state == RUNNABLE&&m->no<key){
480                         key=m->no;
481                     }
482                 }
483             }
484             if(p->state == RUNNABLE && p->no==key) {
485                 // Switch to chosen process. It is the process's job
486                 // to release its lock and then reacquire it
487                 // before jumping back to us.
488                 p->state = RUNNING;
489                 c->proc = p;
490                 key=99;
491                 swtch(&c->context, &p->context);
492
493                 // Process is done running for now.
494                 // It should have changed its p->state before coming back.
495                 c->proc = 0;
496             }
497             release(&p->lock);
498         }
499         if(nproc <= 2) { // only init and sh exist
500             intr_on();
501             asm volatile("wfi");
502         }
503     }

```

5. 修改 procdump(),并编写 C 程序，运行并在过程中使用 ctrl+p 展示效果

```

684 void
685 procdump(void)
686 {
687     static char *states[] = {
688         [UNUSED]    "unused",
689         [SLEEPING]  "sleep ",
690         [RUNNABLE]  "runble",
691         [RUNNING]   "run   ",
692         [ZOMBIE]    "zombie"
693     };
694     struct proc *p;
695     char *state;
696
697     printf("\n");
698     for(p = proc; p < &proc[NPROC]; p++){
699         if(p->state == UNUSED)
700             continue;
701         if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
702             state = states[p->state];
703         else
704             state = "???";
705         printf("%d %s %s,剩余时间片%d,进程优先级%d", p->pid, state, p->name,p->slot,p->no);
706         printf("\n");
707     }
708 }

```



```

C tcostlot.c
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4  #include "stddef.h"
5  int
6  main(void){
7      int i,j,k,n;
8      for(j=1;j<=50000;j++){
9          for(k=1;k<=100000;k++){
10             n++;
11         }
12     }
13     for(i=1;i<=5;i++){
14         if(fork()==0){
15             for(j=1;j<=i*100000;j++){
16                 for(k=1;k<=100000;k++){
17                     n++;
18                 }
19             }
20             printf("s%d\n",i);
21             exit(0);
22         }
23     }
24     for(j=1;j<=600000;j++){
25         for(k=1;k<=100000;k++){
26             n++;
27         }
28     }
29     printf("f\n",i);
30     exit(0);
31 }

```

```

xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ tcostlot

1 sleep  init, 剩余时间片4, 进程优先级1
2 sleep  sh, 剩余时间片4, 进程优先级1
3 run    tcostlot, 剩余时间片5, 进程优先级2

1 sleep  init, 剩余时间片4, 进程优先级1
2 sleep  sh, 剩余时间片4, 进程优先级1
3 run    tcostlot, 剩余时间片8, 进程优先级3
4 run    tcostlot, 剩余时间片4, 进程优先级1
5 run    tcostlot, 剩余时间片4, 进程优先级1
6 runble tcostlot, 剩余时间片4, 进程优先级1
7 runble tcostlot, 剩余时间片4, 进程优先级1
8 runble tcostlot, 剩余时间片4, 进程优先级1

1 sleep  init, 剩余时间片4, 进程优先级1
2 sleep  sh, 剩余时间片4, 进程优先级1
3 runble tcostlot, 剩余时间片12, 进程优先级3
4 run    tcostlot, 剩余时间片5, 进程优先级2
5 run    tcostlot, 剩余时间片5, 进程优先级2
6 runble tcostlot, 剩余时间片8, 进程优先级2
7 runble tcostlot, 剩余时间片8, 进程优先级2
8 run    tcostlot, 剩余时间片1, 进程优先级1

```

可以观察到多个优先级与进程切换

如果要实现 FCFS，可以考虑在 proc 中添加一个变量来表示进入队列时间，或者建立三个真正的队列来利用 fifo 的性质