Project2 Non-Preemptive Kernel 设计文档

中国科学院大学
付琳晴
10.17.2017

1. Context Switching 设计流程

1.1 PCB

PCB 即进程控制块，是用来存放进程的控制信息的数据结构。该数据结构中存储的信息有：进程起始地址，进程所占大小，上下文信息（记录了重要寄存器和栈指针的位置），栈底地址，进程的状态，进程标识符等。xv6 源码中 pcb 结构体定义如图 1. 而在本次实验中，其实可以进行简化，因此在我的代码中 PCB 结构如图 2.

```
30  struct proc {
31    char *mem;                  // Start of process memory (kernel address)
32    uint sz;                    // Size of process memory (bytes)
33    char *kstack;               // Bottom of kernel stack for this process
34    enum procstate state;       // Process state
35    volatile int pid;           // Process ID
36    struct proc *parent;        // Parent process
37    struct trapframe *tf;       // Trap frame for current syscall
38    struct context *context;    // Switch here to run process
39    void *chan;                 // If non-zero, sleeping on chan
40    int killed;                 // If non-zero, have been killed
41    struct file *ofile[NOFILE]; // Open files
42    struct inode *cwd;          // Current directory
43    char name[16];              // Process name (debugging)
44  };
45
```

图 1. xv6 源码中 PCB 结构体

```
63  typedef struct pcb {
64
65      int pid;//program ID
66      process_state state; //program state
67      uint32_t s0;
68      uint32_t s1;
69      uint32_t s2;
70      uint32_t s3;
71      uint32_t s4;
72      uint32_t s5;
73      uint32_t s6;
74      uint32_t s7;
75      uint32_t fp;//frame pointer
76      uint32_t sp;//stack pointer
77      uint32_t ra;// return address
78  } pcb_t;
```

图 2. 实验代码

1.2 初始化 task 过程

在 kernel.c 中进行初始化。初始化过程分为两个部分：初始化任务队列和初始化各个任务的 PCB 并压入 ready 队列。初始化任务队列很简单，只需要分配两个空间分别用来存 ready 队列和 blocked 队列。

初始化各个任务的 PCB 需要将 pcb_t 结构体中的各个域分别初始化，然后调用 queue_push 函数将该 task 压入 ready 队列。需要注意的是将 ra 域（用来存放返回地址）初始化成该 task 的第一条语句地址，即 entry_point，这点非常重要，这是执行任务的必要条件，也是 task 数组与我们所写代码唯一且重要的联系。初始化代码如图 3.

初始化完成之后就可以开始执行任务了，通过调用 scheduler_entry 函数，从 task 数组中按照顺序取出任务（task 数组在 tasks.c 中定义，并可以根据不同要求更改）开始执行。第一个任务就是通过 scheduler_entry 函数中调用的 schedular 函数被启动。

```
76    for(i = 0; i < NUM_TASKS; i++){
77      //PCBs[i].sz = STACK_SIZE;
78      //PCBs[i].kstack = STACK_SIZE - (STACK_SIZE * (i+1));
79      PCBs[i].state = PROCESS_READY;
80      PCBs[i].pid = i;    //process ID
81      stack_top += STACK_SIZE;
82      PCBs[i].s0 = 0;
83      PCBs[i].s1 = 0;
84      PCBs[i].s2 = 0;
85      PCBs[i].s3 = 0;
86      PCBs[i].s4 = 0;
87      PCBs[i].s5 = 0;
88      PCBs[i].s6 = 0;
89      PCBs[i].s7 = 0;
90      PCBs[i].sp = stack_top;
91      PCBs[i].fp = stack_top;
92      PCBs[i].ra = task[i]->entry_point;
93
94      queue_push(ready_queue,PCBs+i);
95      //PCBs[i].context = (struct context*)(STACK_MAX - (STACK_SI
96    }
```

图 3. Task 初始化

1.3 schedular 调用和执行流程

任务初始化过后，通过 schduler_entry 函数开始执行任务。scheduler_entry 函数首先调用了 scheduler，该函数会 pop 出队列中第一个任务作为将要执行的任务，并将这个任务的 pcb_t 结构指针存在 current_running 中，方便我们之后对当前任务的 pcb_t 结构信息进行修改。Scheduler_entry 函数继续执行，将栈中保存的寄存器数据重新恢复给寄存器，这里最重要的就是每个任务第一次执行时，在初始化中我们对 ra 的初始化将派上用场，schedular 执行完会根据 ra 中的返回地址（即 task 的 entry_point）开始执行第一个任务。当前任务执行完之后会继续选取 ready 队列中下一个任务执行。

1.4 context switching 保存 PCB

当一个任务运行到一个阶段时，调用 do_yield 函数，意味这该任务将被移到 ready 队列的最后，实现任务的轮转。在 do_yield 函数中，需要保存 PCB，才能保证下一次该任务能

够继续从当前地方执行。而保存当前任务的 PCB 信息，将下一个任务的 PCB 信息赋给寄存器，这个过程称为 context switching。

do_yield 函数代码如图 4. 为了使得任务再切换回来时能顺利执行，save_pcb 将保存当前任务的重要寄存器中的数据于 PCB 结构中。在 MIPS 里是$16-$23 寄存器和$29-$31 寄存器。存的顺序要和 scheduler_entry 函数中取的顺序一样。Save_pcb 函数执行完后 scheduler_entry 函数将下一个需要执行的任务的寄存器信息从栈提取到寄存器中。

```
33  void do_yield(void)
34  {
35      save_pcb();
36      /* push the qurrently running process on ready queue */
37      /* need student add */
38      current_running->state = PROCESS_READY;
39      queue_push(ready_queue, (pcb_t*)current_running);
40
41      // call scheduler_entry to start next task
42      scheduler_entry();
43
44      // should never reach here
45      ASSERT(0);
46  }
47
```

图 4. do_yield 函数代码

1.5 对进程与内核线程区别的理解

内核线程可以直接访问内核数据，调用内核函数，但进程只能通过系统调用才可以达到此目的。二者的差别在代码中有明显的体现。

线程可以直接调用 do_yield 函数。但是进程执行内核中的函数需要通过系统调用，例如 process1 中使用了 yield 函数，查看 yield 函数代码如图 5，这里通过 SYSCALL 跳转到 kernel_entry 函数（见 entry_mips.S），而在 kernel_entry 函数中调用了 do_yield 函数，部分代码见图 6。可见进程执行内核中的函数更复杂。

```
15  yield :
16      addiu sp,sp,-24
17      sw ra,0(sp)
18      sw a0,8(sp)
19      sw a1,16(sp)
20      SYSCALL(0)
21      lw a1,16(sp)
22      lw a0,8(sp)
23      lw ra,0(sp)
24      addiu sp,sp,24
25      jr ra
26      nop
27
```

图 5. yield 函数汇编代码

```
16  kernel_entry:
17      addiu sp, sp, -24
18      sw ra, 0(sp)
19      bnez $4, 1f
20      nop
21
22      jal do_yield
23      nop
24      beqz $0,2f
25      nop
26
```

图 6. kernel_entry 函数部分汇编代码

1.6 遇到的困难

Task1 真的好难写好难写，虽然写出来觉得没什么，但是文件太多，要搞清各个部分之间的关联很花时间，而且任何一个文件中小小的变动都会导致跑不出来小飞机。总结错误点如下。

（1）kernel_entry 没有根据自己编译出的文件进行修改，导致 task1 能显示小飞机但时间全是 0，而且两个线程的信息根本没有打印出来（可能根本没执行）。

解决过程：事实上我一直有意识在用 objdump 查看 kernel_entry 地址并在 syslib.S 文件中修改，但是没有改对位置。应该修改下图中的地址部分，但是我一直在改这个文件最后的 ENTER_POINT，所以即使错的样子和同学的一模一样，同学也给我强调了无数遍要记得修改 kernel_entry，还是一直没搞对。

```
1  #include "regs.h"
2
3  .text
4  /* address of kernel_entry function may be modified, zxj */
5  #define SYSCALL(i) \
6      li a0,i; \
7      li $8,0xa0800480; \
8      jal $8; \
9      nop
10     nop
```

图 7. 需要修改的 kernel_entry 地址

（2）save_pcb 和 scheduler_entry 函数代码写错导致 TLB miss

解决过程：最开始我想在 pcb_t 结构体里存指针，指针指向该 task 存在栈中的寄存器信息。因此最初我的 pcb_t 结构体定义是这样的，如图 8. 这样的后果就是 save_pcb 里怎么写代码都不对，理想中不过是多用几次 lw 指令取到正确数据，但是现实总是残酷的，程序跑一半总是因为 sp 成 0 或者其他原因而停止。至今不知原因。最后采用了跑出飞机的同学的结构设置。

```
36 struct context {
37     uint32_t s0;
38     uint32_t s1;
39     uint32_t s2;
40     uint32_t s3;
41     uint32_t s4;
42     uint32_t s5;
43     uint32_t s6;
44     uint32_t s7;
45     uint32_t sp;
46     uint32_t s8;
47     uint32_t ra;
48 };
49
50
51 typedef struct pcb {
52     //char *mem;                //start of process memory(kernel address)
53     struct context *context; //switch here to run process :edi esi ebx ebp eip
54     process_state state;      //process state
55     int pid;          //process ID
56
57 } pcb_t;
58
```

图 8. 没有 debug 成功的 pcb_t 结构

2. Context Switching 开销测量设计流程

2.1 测量线程与线程的切换时间

我们需要编写 th4 和 th5 两个线程，在其中使用 get_timer 函数记录 context switching 时间并打印。当然，记录时间的变量需要是个全局变量。为了时间不被其他函数影响导致时间增加，我们需要在 th4 调用 do_yield 函数前记录一次时间，并在 th5 开始时记录时间，这样时间恰好记录的就是 th4 切换到 th5 需要的时间。

2.2 测量线程与进程的切换时间

采用在 th4 线程中记下时间，第一步记录从 th4 切换到 process3 前的时间，第二步记录从 process3 切换到 th4 后的时间，相减除 2 就可看做是线程切进程的时间。

2.3 遇到的困难

最开始测试的是进程切换到进程的时间，最后让 process3 一直自己 yield 再调用该任务。后来写实验报告过程中，发现要求测试的是线程切进程的时间，于是临时改了一下代码，把 process3 的测试时间的部分加到 thread4 了。

（1）打印出线程线程切换和线程进程切换时间后产生 TLB miss

解决过程：发现 TLB miss 后打印出的信息，其中用到的一个寄存器里的数竟然是 0。想了很久，意识到自己从来没想过任务全部执行完会发生什么，因为在 task1 里我们的小飞机是一直循环在执行的。当 task 数组里没有东西，取出的 current_running 是 NULL，而我之前没有考虑 NULL 时函数的情况，因此继续取东西，但全是 0. 因此我最后在 schedule 中加入了判断，如图 9. 当没有任务后会打印出没有任务的字样。同时我在 th4 和 process3 中加入了循环，使任务多执行几次。

```
16  void scheduler(void)
17  {
18      ++scheduler_count;
19
20      // pop new pcb off ready queue
21      /* need student add */
22
23      current_running = queue_pop(ready_queue);
24      while (!current_running){
25          print_str(0, 0, "absense of tasks:\n");;
26      }
27      current_running->state = PROCESS_RUNNING;
28          //print_str(0,0,"no task");
29          //return;
30
31  }
```

图 9. Sheduler 函数代码

3. Mutual lock 设计流程

3.1 自旋锁和互斥锁

自旋锁不会引起调用者睡眠，如果自旋锁已经被别的线程保持，那么调用者就会一直循

环查看锁是否被释放。这会一直占用 CPU。在老师提供的代码里，采用的是一直 yield 的方法，只要锁还没被释放，就一直扔回队列最后面。代码如图 10. 而互斥锁不会一直循环看锁有没有被释放，当互斥锁被别的线程保持，调用者整个任务会被扔到 blocked 队列里，过一段时间会重新放入 ready 队列。

```
28  void lock_acquire(lock_t * l)
29  {
30      if (SPIN) {
31          while (LOCKED == l->status)
32          {
33              do_yield();
34          }
35          l->status = LOCKED;
```

图 10. 自旋锁获取代码

用助教的话通俗的解释二者的区别其实是，自旋锁会一直看锁有没有被释放，而互斥锁是过一段时间再过来看锁有没有被释放。

3.2 互斥锁设计

获取到锁时，意味着当前任务将占有这个资源，其他任务不能占有，因此会将锁的状态改为 LOCKED。直到这个任务将锁释放，其他的任务才能使用这个资源。

未获取到锁，意味着互斥锁已经被别的线程保持，那么当前任务会被扔到 blocked 队列里，schedule 会再从 ready 队列 pop 出下一个任务执行。被扔到 blocked 里的任务将会在锁被释放的时候放一个到 ready 队列里，该功能在 lock_release 函数里。如图 11.

```
45  void lock_release(lock_t * l)
46  {
47      if (SPIN) {
48          l->status = UNLOCKED;
49      } else {
50          /* need student add */
51          if(blocked_tasks()){/*Returns TRUE if there are blocked tas
52              unblock();
53          }
54          l->status = UNLOCKED;
55
56      }
57  }
58
```

图 11. lock_release 函数

3.3 遇到的问题

（1）打印出的信息和预想中不太一样

解决过程：原本错误的打印如图 12. 可以看到 th4 在 th3 明明锁了互斥锁的情况下诡异的拿到了锁。查看代码发现明明对的啊，在 lock_acquire 中明明将锁状态改成 LOCKED 之后 return 啊。最后发现好像函数不认这个 return…又把 thread3 block 了。因此我加上了 if..else 结构，就正常了。

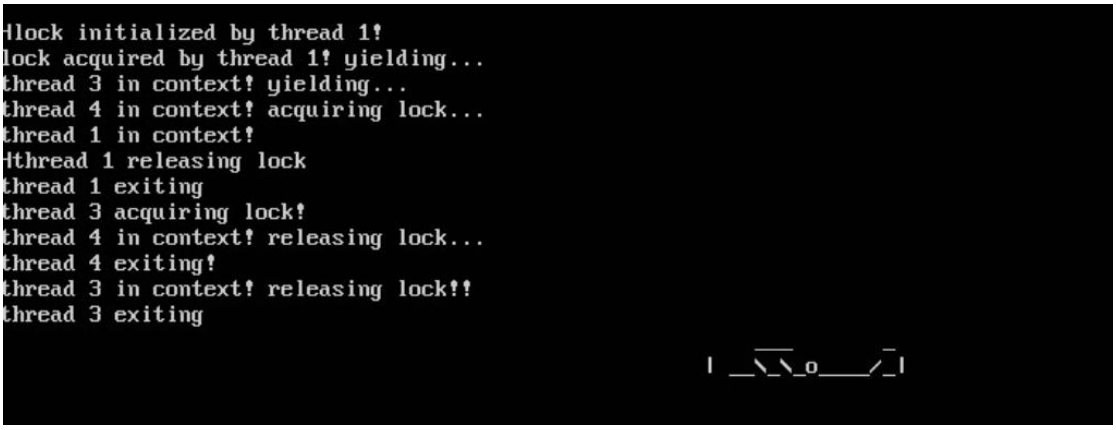

图 12. 错误版本

（2）task all 最终打印出 Failed

解决过程：unblock 函数每次释放出一个。并且要判断一下 block 队列是否为空！这很重要！从 block 队列中取出的任务放在 ready 队列的前面或最后面没有影响。

4. 关键函数功能

4.1 save_pcb

将关键寄存器中的数据存入 pcb_t 结构体中。这里存 sp 寄存器数据的时候需要注意，根据反汇编，进入 do_yield 函数时 sp 首先减了 24，如图 13. 在 do_yield 中首先调用的 save_pcb，因此想保存之前的 sp 需要将 sp 加 24。

```
281  a0800608 <do_yield>:
282  a0800608:   27bdffe8    addiu   sp,sp,-24
283  a080060c:   afbf0010    sw  ra,16(sp)
284  a0800610:   0c20013f    jal a08004fc <save_pcb>
285  a0800614:   00000000    nop
286  a0800618:   3c05a080    lui a1,0xa080
287  a080061c:   8ca32118    lw  v1,8472(a1)
```

图 13. 反汇编

4.2 write_file

是 Createimage.c 文件中用于写文件可执行代码到 image 文件的函数，其实是根据 write_block 和 write_kernel 修改的，都合成了一个函数。这里添加了一个新的参数 sector_num，用来记录要写多少个扇区，从而保证将除可执行代码部分外其他都填充 0.

4.3 do_yield

保存 PCB 信息并选取下一个任务执行，代码如图 14. 需要记得把这个任务扔回队列最后。

```
33  void do_yield(void)
34  {
35      save_pcb();
36      /* push the qurrently running process on ready queue */
37      /* need student add */
38      current_running->state = PROCESS_READY;
39      queue_push(ready_queue, (pcb_t*)current_running);
40
41      // call scheduler_entry to start next task
42      scheduler_entry();
43
44      // should never reach here
45      ASSERT(0);
46  }
```

图 14. Do_yield 代码

■