

Project2 Non-Preemptive Kernel 设计文档

中国科学院大学

张远航

2017 年 10 月 18 日

1. Context Switching 设计流程

(1) 进程控制块 (PCB)

进程控制块 (process control block, PCB) 中存储了与操作系统中一个进程所处状态相关的全部信息, 保存的内容通常包括进程状态、程序计数器 (PC)、各通用寄存器、进程号 (PID) 等。每个进程的创建与销毁都由内核负责, 它们都需要在 PCB 中登记信息。在操作系统发生上下文切换时, 我们需要保存和载入相应任务的 PCB。

本次实验中, 我们需要记录在 PCB 中的信息包括: 进程状态 `process_state`, 包括被阻塞 (`PROCESS_BLOCKED`)、就绪待运行 (`PROCESS_READY`)、运行中 (`PROCESS_RUNNING`) 和已退出 (`PROCESS_EXITED`); 各通用寄存器 (含 `$sp` 和 `$ra`); PID 其实在实验中不起什么作用, 但是为了符合一般规范, 我们可以根据任务的类型 (`task_type`) 为其分配 PID, 比如认为内核态线程均是由内核创建的, 分配 PID 为 1, 其他用户态进程 PID 依次自增即可。

(2) 启动首任务

进入内核之后, 我们首先要为待执行的各任务创建 PCB (见关键代码段 (1))。启动首任务发生在其后 (`scheduler.c: 109`), `scheduler_entry` 将启动调度器, 从 FIFO 的就绪队列中取出首个任务, 加载 PCB 并跳转到 `ra` 寄存器中存储的任务入口地址开始运行。

(3) 调度器的调用与执行流程

我们本次实验中实现的是一个非抢占式内核, 因此调度只发生在任务退出或主动让出 CPU 时。调用调度器 (`scheduler_entry`) 时, 首先会跳到 `scheduler()` 函数, 从就绪队列中取出一个任务, 并将 `current_running` 指针指向其 PCB; 接着, 通过汇编指令将 PCB 中存储的各通用寄存器还原, 最后用一条 `jr $ra` 开始执行该任务。

```

Time (in us): 3115444
Thread 1 (time): 7
Thread 2 (lock): 11
Thread 3 (lock): 12

```

Diagram of a lock structure:

```

|-----|
| 0 |
|-----|
| 0 |
|-----|

```

Terminal footer: yuan-hang@caszhang www.caszhang.cn
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Online 0:6 | ttyUSB0

图 1: Task 1 进程上下文切换测试

(4) 任务的上下文切换

如果发生上下文切换是某任务主动让出 CPU 所致，我们就必须保存 PCB，以便下一次切换回该任务时还能正常运行。这一操作是通过汇编完成的：首先，找到 PCB 结构体的指针 `current_running`，接着以其指向的地址为基址在接下来连续的内存单元中写入当前各通用寄存器的值即可。需要注意的是，不管是用户态还是内核态的任务，在进入 `do_yield` 段时，栈指针寄存器 `sp` 被首先减去 3 字节（`addiu sp,sp,-24`），接着 `ra` 被存放在距 `sp` 5 字节处（`sw ra,20(sp)`¹），这才是我们需要保存的真实值。

(5) 调试记录

Task 1 的测试结果如图 1 所示。

栈区域设置 我们需要在内存中为栈分配一块地址空间。最低地址和最高地址我们做如下考虑：进程 3 的入口地址是 `0xa0830000`，故栈至少应在进程 3 的程序段之外，但是也不能太高，否则内存不足。经过试验，最后选择将栈底地址设为 `0xa0890000`。

kernel_entry 最开始忘了修改 SYSCALL 中 `kernel_entry` 的地址导致出错。

2. Context Switching 开销测量设计流程

本节中，代码中开始计时与结束计时的位置分别以黄色和红色加以高亮。

¹我的机器上看汇编代码如此，和课件里还有其他同学的不太一样，他们的是差 4 字节（16）。

(1) 线程-线程切换开销测量

由于被测的 `thread4` 与 `thread5` 是相邻的两个任务，会被调度器相继调度，我们只需在线程 4 让出 CPU 并退出前开始计时，在线程 5 进入时停止计时作差求出切换开销即可。这一操作比较简单。

th3.c

```

1 void thread4(void)
2 {
3     recorded_time = get_timer();
4     do_yield();
5     do_exit();
6 }
7
8 void thread5(void)
9 {
10    uint32_t time = get_timer();
11    time = time - recorded_time;
12    ...
13 }
```

(2) 线程-进程切换开销测量

为了记录线程与进程间切换的开销，我们需要记录线程执行完毕退出的时刻，并在用户态的进程开始执行时读取这一值作差。这一过程穿过了内核边界，严格来讲应当增加一个 `syscall`，通过系统调用获取内核记录下的数据，这样用户不会直接访问内核使用的地址空间，保证用户不会发生越权访问；但是本次实验较为简单，线程与进程从行为上看没有太大的区别。此外，`util.h` 中已经为我们定义好了一个变量 `recorded_time`，考虑到 `process3.c` 中也引用了这一头文件，我们可以暂时直接利用这一全局变量存储线程执行完毕时的 CPU 周期数，然后在进入进程 3 时作差求出切换开销。

th3.c

```

1 void thread5(void)
2 {
3     ...
4     recorded_time = get_timer();
5     do_exit();
6 }
```

process3.c

```

1 void _start(void)
```

```

yuan-hang@caszhang: 查看(V) 搜索(S) 终端(T) 帮助(H)
Scheduler: Nothing left to do.

Context switch time: (in ms)
Thread:      Thread / process:  Process:

1           2           2
1           4           6
           9           6
           8           6
           2           6
           1           6
           0           6

yuan-hang@caszhang www.caszhang.cn
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Online 0:29 | ttyUSB0

```

图 2: Task 2 上下文切换开销测试

```

2 {
3     uint32_t time = get_timer();
4     ...
5     print_int(15, 10, (int)(time - recorded_time) / MHZ);
6     ...
7 }

```

(3) 进程-进程切换开销测量

如下所示，测量进程-进程切换的开销，只要从 `yield` 前开始计时，再次回到进程时停止计时即可（此前线程 4 和线程 5 已经退出，因此调度器再次回到的一定还是进程 3）。

process3.c

```

1 void _start(void)
2 {
3     ...
4     time = get_timer();
5     yield();
6     ...
7     print_int(35, 10, (int)(get_timer() - time) / MHZ);
8     ...
9 }

```

(4) 调试记录

调度器无任务导致异常 由于 Task 2 中测试的两个线程均在执行完毕后退出（我设置成了只运行一次，以便读数），调度器没有找到下一个可以执行的任务（即 `ready_queue` 为空），但 `queue_pop` 本身是没有边界检查的，结果调度器不断地从队列中取出空指针，引发了 TLB miss 异常。修改为从队列中取任务前先检查队列是否非空，若为空则进入自旋状态，问题解决。

scheduler.c（错误）

```

1  // pop new pcb off ready queue
2  current_running = queue_pop(ready_queue);
3  // do not run exited processes
4  while (current_running->state == PROCESS_EXITED)
5      current_running = queue_pop(ready_queue);
6  while (1)
7      print_str(0, 0, "Nothing left to do...");
8  current_running->state = PROCESS_RUNNING;
```

scheduler.c（正确）

```

1  if (ready_queue->isEmpty) {
2      while (1)
3          print_str(0, 0, "Scheduler: Nothing to do.");
4  }
5  else {
6      // pop new pcb off ready queue
7      current_running = queue_pop(ready_queue);
8      // do not run exited processes
9      while (current_running->state == PROCESS_EXITED) {
10         if (ready_queue->isEmpty) {
11             while (1)
12                 print_str(0, 0, "Scheduler: Nothing to do.");
13         }
14         else current_running = queue_pop(ready_queue);
15     }
16     current_running->state = PROCESS_RUNNING;
17 }
```

测试结果 Task 2 切换开销测量在板上的运行效果如图 2 所示。可见，线程-线程上下文切换（`do_yield`）的开销最小，为 11 毫秒；进程-进程切换（`yield`）次之，为 2666 毫秒；线程-进程上下文切换（调度器按 RR 轮转选择下一任务）的开销最大，比前两者要大 2 ~ 5 个数量级。

3. Mutual lock 设计流程

(1) 自旋锁与互斥锁

自旋锁 (spin-lock) 与互斥锁 (mutex) 都是为了保证多线程在同一时刻只能有一个线程操作临界区 (critical section) 而引入的。不同点在于, 对于互斥锁, 如果目前是加锁状态, 那么后面的线程就会进入“休眠”状态, 解锁之后, 又会被唤醒继续执行; 如果是自旋锁, 那么后面的线程会一直等待, 直到锁被释放后立刻执行。因此, 自旋锁比较适合执行较短的任务, 否则会产生较大的性能消耗。

(2) 互斥锁的处理流程

互斥锁的主要操作有三个: `lock_init` (创建锁)、`lock_acquire` (获得锁) 和 `lock_release` (释放锁), 具体实现见关键函数段 ??。 `lock.c` 顶部注释中给出了以下几个假设:

- 要求满足 FIFO 公平性原则;
- 只有一个进程会尝试创建锁;
- 进程只会尝试获得已创建的锁;
- 进程一定会释放锁。

创建锁时, 我们只需将锁的状态设为未加锁即可。当一个任务申请锁时, 如果能获取到锁, 我们就加锁, 然后继续执行任务的剩余部分; 否则调用 `block()`, 将该任务加入阻塞队列并修改 PCB 记录为阻塞状态。某任务释放锁时, 我们判断阻塞队列是否非空, 如果非空则调用 `unblock()` 从阻塞队列中取出首个未执行的任务将其加入就绪队列, 这样待当前任务和就绪队列中其后不需要锁的任务执行完成后, 调度器便会开始运行这些之前被阻塞的任务; 否则只要解锁即可。²

(3) 调试记录

互斥锁的调试比较顺利, Task 3 的测试结果如图 3 所示。所有任务的综合测试效果如图 4 所示。

²注: 此处有争议, 有同学认为必须取出队首元素。——张远航 11/16/17

```

Hlock initialized by thread 1!
lock acquired by thread 1! yielding...
thread 3 in context! yielding...
thread 4 in context! acquiring lock...
thread 1 in context!
Hthread 1 releasing lock
thread 1 exiting
thread 3 acquiring lock!
thread 3 in context! releasing lock!!
thread 3 exiting
thread 4 in context! releasing lock...
thread 4 exiting!

```

yuan-hang@caszhang www.caszhang.cn
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Online 0:6 | ttyUSB0

图 3: Task 3 互斥锁测试

4. 关键函数功能

(1) 创建 PCB (kernel.c : 87-104)

读取 `tasks.c` 中各任务的入口地址，存入 PCB 记录的 `$ra` 寄存器中，并从栈底地址 `STACK_MIN` 开始连续分配栈空间。内核中的线程只具有内核态的上下文，无需保存；因此我们为用户态和内核态的上下文分配一个公共的栈。PCB 表设置完成后，还要将任务加入就绪队列，等待被调度器调度。

```

1   pcb_t *newtask = &pcbs[0];    // pointer to first PCB entry
2   int nproc = 1;                // kernel is the only running process
3   int i, j;
4
5   // set up each PCB entry and push to queue
6   for (i = 0 ; i < NUM_TASKS; ++i) {
7       struct task_info *ptask = task[i];
8       stack_top += STACK_SIZE;
9       if (ptask->task_type == KERNEL_THREAD)
10          newtask->pid = 1;
11       else
12          newtask->pid = ++nproc;
13       for (j = 0; j < NUM_REGISTERS; ++j)
14          newtask->pcb_context[j] = 0;
15       newtask->pcb_reg_ra = task[i]->entry_point;
16       newtask->pcb_reg_sp = stack_top;
17       newtask->state = PROCESS_READY;
18
19       bool_t init_success = queue_push(ready_queue, newtask);

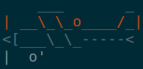
```

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Time (in us): 7536813
Thread 1 (time) : 9

Thread 2 (lock) : 25
Thread 3 (lock) : 26

Did you know that 1 + ... +6 = 21
Process 2 (Math) : 6

Hlock initialized by thread 1!
lock acquired by thread 1! yielding...
thread 3 in context! yielding...
thread 4 in context! acquiring lock...
thread 1 in context!
Hthread 1 releasing lock
thread 1 exiting
thread 3 acquiring lock!
thread 3 in context! releasing lock!!
thread 3 exiting
thread 4 in context! releasing lock...
thread 4 exiting!
```



```
yuan-hang@caszhang www.caszhang.cn
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Online 0:8 | ttyUSB0
```

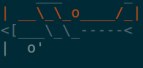
(a) 运行过程中

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Time (in us): 7536813
Thread 1 (time) : 9

Thread 2 (lock) : Exited
Thread 3 (lock) : Passed

Did you know that 1 + ... +100 = 5050
Process 2 (Math) : Exited

Hlock initialized by thread 1!
lock acquired by thread 1! yielding...
thread 3 in context! yielding...
thread 4 in context! acquiring lock...
thread 1 in context!
Hthread 1 releasing lock
thread 1 exiting
thread 3 acquiring lock!
thread 3 in context! releasing lock!!
thread 3 exiting
thread 4 in context! releasing lock...
thread 4 exiting!
```



```
yuan-hang@caszhang www.caszhang.cn
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Online 0:15 | ttyUSB0
```

(b) 运行结束

图 4: 整体测试


```

20     ASSERT(init_success == TRUE);
21     ++newtask;
22 }

```

(2) 内核和进程间区域填零 (createimage.c: 44-50)

每次写镜像后，保持文件指针不动；直接从下一可执行文件中读出程序将被加载到的起始内存地址，减去 0xa0800000 得到在镜像内的相对偏移，然后补零。

```

1  // pad between executables; assume at least one program section
2  padding = p_hdr[0].p_vaddr - 0xa0800000 - ftell(*image_file);
3  printf("Target at %x, now at %x; to pad %x\n", p_hdr[0].p_vaddr -
         0xa0800000, ftell(*image_file), (unsigned int)padding);
4  while (padding > 0) {
5      fputc(0, *image_file);
6      padding--;
7  }

```

(3) 锁的获取与释放 (lock.c: 28-52)

原理解释见第 3 节。

```

1  void lock_acquire(lock_t * l)
2  {
3      if (SPIN) {
4          while (LOCKED == l->status) {
5              do_yield();
6          }
7          l->status = LOCKED;
8      } else {
9          if (l->status == UNLOCKED)
10             l->status = LOCKED;
11             else block();
12     }
13 }
14
15 void lock_release(lock_t * l)
16 {
17     if (SPIN) {
18         l->status = UNLOCKED;
19     } else {
20         if (blocked_tasks())
21             unblock();
22         else

```

```
23         l->status = UNLOCKED;  
24     }  
25 }
```