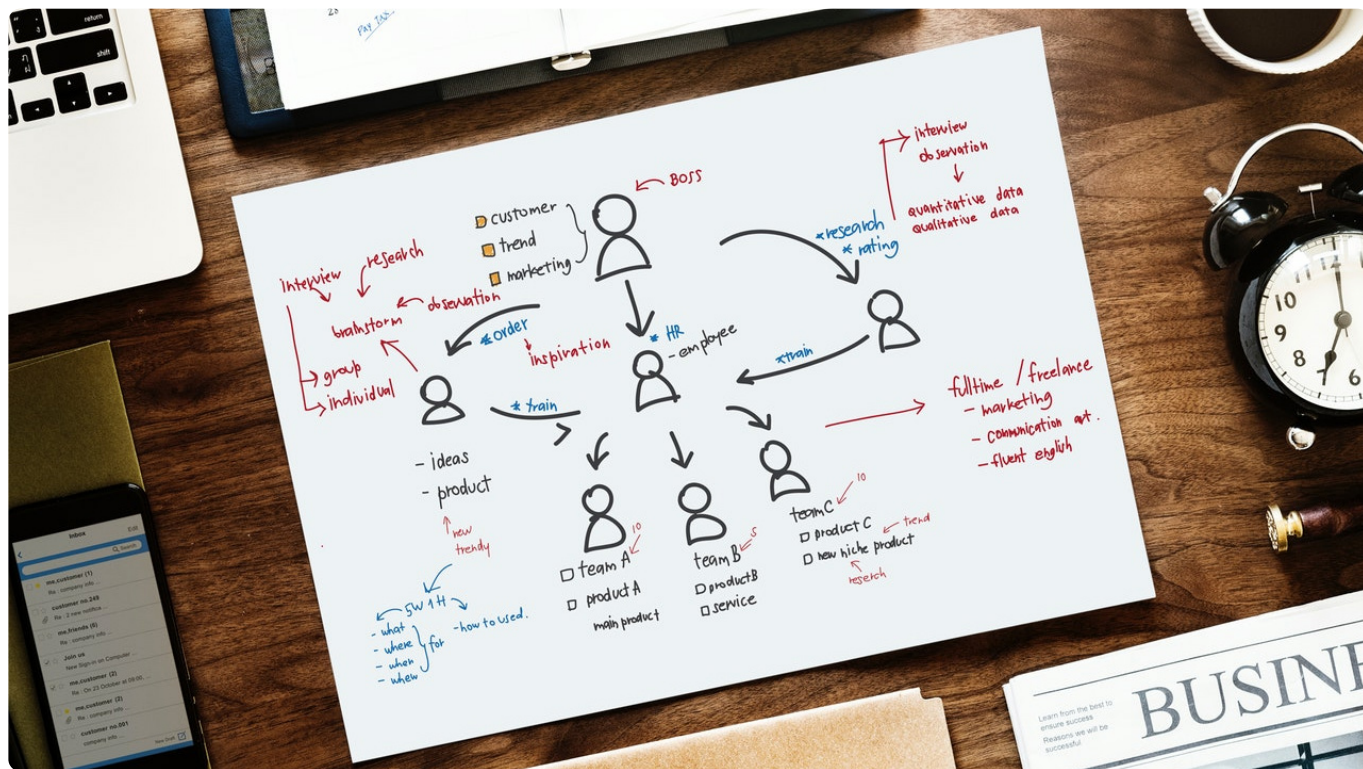


16 | 调度（中）：主动调度是如何发生的？

2019-05-03 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 20:12 大小 18.50M



上一节，我们为调度准备了这么多的数据结构，这一节我们来看调度是如何发生的。

所谓进程调度，其实就是一个人在做 A 项目，在某个时刻，换成做 B 项目去了。发生这种情况，主要有两种方式。

方式一： A 项目做着做着，发现里面有一条指令 sleep，也就是要休息一下，或者在等待某个 I/O 事件。那没办法了，就要主动让出 CPU，然后可以开始做 B 项目。


方式二： A 项目做着做着，旷日持久，实在受不了了。项目经理介入了，说这个项目 A 先停停，B 项目也要做一下，要不然 B 项目该投诉了。

主动调度

我们这一节先来看方式一，主动调度。


这里我找了几个代码片段。**第一个片段是 Btrfs，等待一个写入。**[Btrfs](#) (B-Tree) 是一种文件系统，感兴趣你可以自己去了解一下。

这个片段可以看作写入块设备的一个典型场景。写入需要一段时间，这段时间用不上 CPU，还不如主动让给其他进程。

 复制代码

```
1 static void btrfs_wait_for_no_snapshoting_writes(struct btrfs_root *root)
2 {
3     .....
4     do {
5         prepare_to_wait(&root->subv_writers->wait, &wait,
6                         TASK_UNINTERRUPTIBLE);
7         writers = percpu_counter_sum(&root->subv_writers->counter);
8         if (writers)
9             schedule();
10        finish_wait(&root->subv_writers->wait, &wait);
11    } while (writers);
12 }
```


另外一个例子是，**从 Tap 网络设备等待一个读取。**Tap 网络设备是虚拟机使用的网络设备。当没有数据到来的时候，它也需要等待，所以也会选择把 CPU 让给其他进程。

 复制代码

```
1 static ssize_t tap_do_read(struct tap_queue *q,
2                             struct iov_iter *to,
3                             int noblock, struct sk_buff *skb)
4 {
5     .....
6     while (1) {
7         if (!noblock)
8             prepare_to_wait(sk_sleep(&q->sk), &wait,
9                             TASK_INTERRUPTIBLE);
10    .....
11        /* Nothing to read, let's sleep */
12        schedule();
13    }
14    .....
15 }
```


你应该知道，计算机主要处理计算、网络、存储三个方面。计算主要是 CPU 和内存的合作；网络和存储则多是和外部设备的合作；在操作外部设备的时候，往往需要让出 CPU，就像上面两段代码一样，选择调用 `schedule()` 函数。

接下来，我们就来看 **schedule 函数的调用过程**。

 复制代码

```
1 asmlinkage __visible void __sched schedule(void)
2 {
3     struct task_struct *tsk = current;
4
5
6     sched_submit_work(tsk);
7     do {
8         preempt_disable();
9         __schedule(false);
10        sched_preempt_enable_no_resched();
11    } while (need_resched());
12 }
```

这段代码的主要逻辑是在 `__schedule` 函数中实现的。这个函数比较复杂，我们分几个部分来讲解。


 复制代码

```
1 static void __sched notrace __schedule(bool preempt)
2 {
3     struct task_struct *prev, *next;
4     unsigned long *switch_count;
5     struct rq_flags rf;
6     struct rq *rq;
7     int cpu;
8
9
10    cpu = smp_processor_id();
11    rq = cpu_rq(cpu);
12    prev = rq->curr;
13    .....
```

首先，在当前的 CPU 上，我们取出任务队列 rq。

task_struct *prev 指向这个 CPU 的任务队列上面正在运行的那个进程 curr。为啥是 prev？因为一旦将来它被切换下来，那它就成了前任了。


接下来代码如下：

 复制代码

```
1 next = pick_next_task(rq, prev, &rf);
2 clear_tsk_need_resched(prev);
3 clear_preempt_need_resched();
```

第二步，获取下一个任务，task_struct *next 指向下一个任务，这就是**继任**。

pick_next_task 的实现如下：

 复制代码

```
1 static inline struct task_struct *
2 pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
3 {
4     const struct sched_class *class;
5     struct task_struct *p;
6     /*
7      * Optimization: we know that if all tasks are in the fair class we can call th
8      */
9     if (likely((prev->sched_class == &idle_sched_class ||
10                prev->sched_class == &fair_sched_class) &&
11              rq->nr_running == rq->cfs.h_nr_running)) {
12         p = fair_sched_class.pick_next_task(rq, prev, rf);
13         if (unlikely(p == RETRY_TASK))
14             goto again;
15         /* Assumes fair_sched_class->next == idle_sched_class */
16         if (unlikely(!p))
17             p = idle_sched_class.pick_next_task(rq, prev, rf);
18         return p;
19     }
20 again:
21     for_each_class(class) {
22         p = class->pick_next_task(rq, prev, rf);
23         if (p) {
24             if (unlikely(p == RETRY_TASK))
25                 goto again;
```


```

26             return p;
27         }
28     }
29 }

```

我们来看 again 这里，就是咱们上一节讲的依次调用调度类。但是这里有了一个优化，因为大部分进程是普通进程，所以大部分情况下会调用上面的逻辑，调用的就是 `fair_sched_class.pick_next_task`。

根据上一节对于 `fair_sched_class` 的定义，它调用的是 `pick_next_task_fair`，代码如下：


 复制代码

```

1 static struct task_struct *
2 pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
3 {
4     struct cfs_rq *cfs_rq = &rq->cfs;
5     struct sched_entity *se;
6     struct task_struct *p;
7     int new_tasks;

```

对于 CFS 调度类，取出相应的队列 `cfs_rq`，这就是我们上一节讲的那棵红黑树。

 复制代码


```

1     struct sched_entity *curr = cfs_rq->curr;
2     if (curr) {
3         if (curr->on_rq)
4             update_curr(cfs_rq);
5         else
6             curr = NULL;
7     }
8     se = pick_next_entity(cfs_rq, curr);
9

```

取出当前正在运行的任务 `curr`，如果依然是可运行的状态，也即处于进程就绪状态，则调用 `update_curr` 更新 `vruntime`。`update_curr` 咱们上一节就见过了，它会根据实际运行时间算出 `vruntime` 来。


接着，`pick_next_entity` 从红黑树里面，取最左边的一个节点。这个函数的实现我们上一节也讲过了。

 复制代码

```
1      p = task_of(se);
2
3
4      if (prev != p) {
5          struct sched_entity *pse = &prev->se;
6          .....
7          put_prev_entity(cfs_rq, pse);
8          set_next_entity(cfs_rq, se);
9      }
10
11
12      return p
```

`task_of` 得到下一个调度实体对应的 `task_struct`，如果发现继任和前任不一样，这就说明有一个更需要运行的进程了，就需要更新红黑树了。前面前任的 `vruntime` 更新过了，`put_prev_entity` 放回红黑树，会找到相应的位置，然后 `set_next_entity` 将继任者设为当前任务。

第三步，当选出的继任者和前任不同，就要进行上下文切换，继任者进程正式进入运行。

 复制代码

```
1 if (likely(prev != next)) {
2     rq->nr_switches++;
3     rq->curr = next;
4     ++*switch_count;
5     .....
6     rq = context_switch(rq, prev, next, &rf);
```

进程上下文切换

上下文切换主要干两件事情，一是切换进程空间，也即虚拟内存；二是切换寄存器和 CPU 上下文。

我们先来看 `context_switch` 的实现。

```

1  /*
2   * context_switch - switch to the new MM and the new thread's register state.
3   */
4  static __always_inline struct rq *
5  context_switch(struct rq *rq, struct task_struct *prev,
6                struct task_struct *next, struct rq_flags *rf)
7  {
8      struct mm_struct *mm, *oldmm;
9      .....
10     mm = next->mm;
11     oldmm = prev->active_mm;
12     .....
13     switch_mm_irqs_off(oldmm, mm, next);
14     .....
15     /* Here we just switch the register state and the stack. */
16     switch_to(prev, next, prev);
17     barrier();
18     return finish_task_switch(prev);
19 }

```

这里首先是内存空间的切换，里面涉及内存管理的内容比较多。内存管理后面我们会有专门的章节来讲，这里你先知道有这么一回事就行了。

接下来，我们看 `switch_to`。它就是寄存器和栈的切换，它调用到了 `__switch_to_asm`。这是一段汇编代码，主要用于栈的切换。


对于 32 位操作系统来讲，切换的是栈顶指针 `esp`。

```

1  /*
2   * %eax: prev task
3   * %edx: next task
4   */
5  ENTRY(__switch_to_asm)
6  .....
7      /* switch stack */
8      movl    %esp, TASK_threadsp(%eax)
9      movl    TASK_threadsp(%edx), %esp
10     .....
11     jmp     __switch_to
12 END(__switch_to_asm)


```

对于 64 位操作系统来讲，切换的是栈顶指针 `rsp`。

 复制代码

```
1  /*
2   * %rdi: prev task
3   * %rsi: next task
4   */
5  ENTRY(__switch_to_asm)
6  .....
7      /* switch stack */
8      movq    %rsp, TASK_threadsp(%rdi)
9      movq    TASK_threadsp(%rsi), %rsp
10 .....
11      jmp     __switch_to
12 END(__switch_to_asm)
```

最终，都返回了 `__switch_to` 这个函数。这个函数对于 32 位和 64 位操作系统虽然有不同的实现，但里面做的事情是差不多的。所以我这里仅仅列出 64 位操作系统做的事情。

 复制代码

```
1  __visible __notrace_funcgraph struct task_struct *
2  __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
3  {
4      struct thread_struct *prev = &prev_p->thread;
5      struct thread_struct *next = &next_p->thread;
6      .....
7      int cpu = smp_processor_id();
8      struct tss_struct *tss = &per_cpu(cpu_tss, cpu);
9      .....
10     load_TLS(next, cpu);
11     .....
12     this_cpu_write(current_task, next_p);
13
14
15     /* Reload esp0 and ss1. This changes current_thread_info(). */
16     load_sp0(tss, next);
17     .....
18     return prev_p;
19 }
```


这里面有一个 Per CPU 的结构体 tss。这是个什么呢？

在 x86 体系结构中，提供了一种以硬件的方式进行进程切换的模式，对于每个进程，x86 希望在内存里面维护一个 TSS（Task State Segment，任务状态段）结构。这里面有所有的寄存器。


另外，还有一个特殊的寄存器 TR（Task Register，任务寄存器），指向某个进程的 TSS。更改 TR 的值，将会触发硬件保存 CPU 所有寄存器的值到当前进程的 TSS 中，然后从新进程的 TSS 中读出所有寄存器值，加载到 CPU 对应的寄存器中。

下图就是 32 位的 TSS 结构。

I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

但是这样有个缺点。我们做进程切换的时候，没必要每个寄存器都切换，这样每个进程一个 TSS，就需要全量保存，全量切换，动作太大了。

于是，Linux 操作系统想了一个办法。还记得在系统初始化的时候，会调用 `cpu_init` 吗？这里面会给每一个 CPU 关联一个 TSS，然后将 TR 指向这个 TSS，然后在操作系统的运行过程中，TR 就不切换了，永远指向这个 TSS。TSS 用数据结构 `tss_struct` 表示，在 `x86_hw_tss` 中可以看到和上图相应的结构。

 复制代码

```
1 void cpu_init(void)
2 {
3     int cpu = smp_processor_id();
4     struct task_struct *curr = current;
5     struct tss_struct *t = &per_cpu(cpu_tss, cpu);
6     .....
7     load_sp0(t, thread);
8     set_tss_desc(cpu, t);
9     load_TR_desc();
10    .....
11 }
12
13
14 struct tss_struct {
15     /*
16      * The hardware state:
17      */
18     struct x86_hw_tss    x86_tss;
19     unsigned long        io_bitmap[IO_BITMAP_LONGS + 1];
20 }
```

在 Linux 中，真的参与进程切换的寄存器很少，主要的就是栈顶寄存器。

于是，在 `task_struct` 里面，还有一个我们原来没有注意的成员变量 `thread`。这里面保留了要切换进程的时候需要修改的寄存器。

 复制代码

```
1 /* CPU-specific state of this task: */
```

所谓的进程切换，就是将某个进程的 thread_struct 里面的寄存器的值，写入到 CPU 的 TR 指向的 tss_struct，对于 CPU 来讲，这就算是完成了切换。

例如 __switch_to 中的 load_sp0，就是将下一个进程的 thread_struct 的 sp0 的值加载到 tss_struct 里面去。

指令指针的保存与恢复

你是不是觉得，这样真的就完成切换了吗？是的，不信我们来**盘点**一下。

从进程 A 切换到进程 B，用户栈要不要切换呢？当然要，其实早就已经切换了，就在切换内存空间的时候。每个进程的用户栈都是独立的，都在内存空间里面。

那内核栈呢？已经在 __switch_to 里面切换了，也就是将 current_task 指向当前的 task_struct。里面的 void *stack 指针，指向的就是当前的内核栈。

内核栈的栈顶指针呢？在 __switch_to_asm 里面已经切换了栈顶指针，并且将栈顶指针在 __switch_to 加载到了 TSS 里面。

用户栈的栈顶指针呢？如果当前在内核里面的话，它当然是在内核栈顶部的 pt_regs 结构里面呀。当从内核返回用户态运行的时候，pt_regs 里面有所有当时在用户态的时候运行的上下文信息，就可以开始运行了。

唯一让人不容易理解的是指令指针寄存器，它应该指向下一条指令的，那它是如何切换的呢？这里有点绕，请你仔细看。


这里我先明确一点，进程的调度都最终会调用到 __schedule 函数。为了方便你记住，我姑且给它起个名字，就叫“**进程调度第一定律**”。后面我们会多次用到这个定律，你一定要记住。

我们用最前面的例子仔细分析这个过程。本来一个进程 A 在用户态是要写一个文件的，写文件的操作用户态没办法完成，就要通过系统调用到达内核态。在这个切换的过程中，用户

态的指令指针寄存器是保存在 `pt_regs` 里面的，到了内核态，就开始沿着写文件的逻辑一步一步执行，结果发现需要等待，于是就调用 `__schedule` 函数。

这个时候，进程 A 在内核态的指令指针是指向 `__schedule` 了。这里请记住，A 进程的内核栈会保存这个 `__schedule` 的调用，而且知道这是从 `btrfs_wait_for_no_snapshoting_writes` 这个函数里面进去的。

`__schedule` 里面经过上面的层层调用，到达了 `context_switch` 的最后三行指令（其中 `barrier` 语句是一个编译器指令，用于保证 `switch_to` 和 `finish_task_switch` 的执行顺序，不会因为编译阶段优化而改变，这里咱们可以忽略它）。

 复制代码

```
1 switch_to(prev, next, prev);
2 barrier();
3 return finish_task_switch(prev);
```

当进程 A 在内核里面执行 `switch_to` 的时候，内核态的指令指针也是指向这一行的。但是在 `switch_to` 里面，将寄存器和栈都切换到成了进程 B 的，唯一没有变的就是指令指针寄存器。当 `switch_to` 返回的时候，指令指针寄存器指向了下一条语句 `finish_task_switch`。

但这个时候的 `finish_task_switch` 已经不是进程 A 的 `finish_task_switch` 了，而是进程 B 的 `finish_task_switch` 了。

这样合理吗？你怎么知道进程 B 当时被切换下去的时候，执行到哪里了？恢复 B 进程执行的时候一定在这里呢？这时候就要用到咱的“进程调度第一定律”了。

当年 B 进程被别人切换走的时候，也是调用 `__schedule`，也是调用到 `switch_to`，被切换成为 C 进程的，所以，B 进程当年的下一个指令也是 `finish_task_switch`，这就说明指令指针指到这里是没有错的。

接下来，我们要从 `finish_task_switch` 完毕后，返回 `__schedule` 的调用了。返回到哪里呢？按照函数返回的原理，当然是从内核栈里面去找，是返回到 `btrfs_wait_for_no_snapshoting_writes` 吗？当然不是了，因为

btrfs_wait_for_no_snapshoting_writes 是在 A 进程的内核栈里面的，它早就被切换走了，应该从 B 进程的内核栈里面找。


假设，B 就是最前面例子里面调用 tap_do_read 读网卡的进程。它当年调用 __schedule 的时候，是从 tap_do_read 这个函数调用进去的。

当然，B 进程的内核栈里面放的是 tap_do_read。于是，从 __schedule 返回之后，当然是接着 tap_do_read 运行，然后在内核运行完毕后，返回用户态。这个时候，B 进程内核栈的 pt_regs 也保存了用户态的指令指针寄存器，就接着在用户态的下一条指令开始运行就可以了。

假设，我们只有一个 CPU，从 B 切换到 C，从 C 又切换到 A。在 C 切换到 A 的时候，还是按照“进程调度第一定律”，C 进程还是会调用 __schedule 到达 switch_to，在里面切换成为 A 的内核栈，然后运行 finish_task_switch。

这个时候运行的 finish_task_switch，才是 A 进程的 finish_task_switch。运行完毕从 __schedule 返回的时候，从内核栈上才知道，当年是从 btrfs_wait_for_no_snapshoting_writes 调用进去的，因而应该返回 btrfs_wait_for_no_snapshoting_writes 继续执行，最后内核执行完毕返回用户态，同样恢复 pt_regs，恢复用户态的指令指针寄存器，从用户态接着运行。

到这里你是不是有点理解为什么 switch_to 有三个参数呢？为啥有两个 prev 呢？其实我们从定义就可以看到。

 复制代码

```
1 #define switch_to(prev, next, last) \
2 do { \
3     prepare_switch_to(prev, next); \
4 \
5     ((last) = __switch_to_asm((prev), (next))); \
6 } while (0)
```

在上面的例子中，A 切换到 B 的时候，运行到 __switch_to_asm 这一行的时候，是在 A 的内核栈上运行的，prev 是 A，next 是 B。但是，A 执行完 __switch_to_asm 之后就被切换走了，当 C 再次切换到 A 的时候，运行到 __switch_to_asm，是从 C 的内核栈运行的。

这个时候，prev 是 C，next 是 A，但是 __switch_to_asm 里面切换成为了 A 当时的内核栈。

还记得当年的场景 “prev 是 A，next 是 B”，__switch_to_asm 里面 return prev 的时候，还没 return 的时候，prev 这个变量里面放的还是 C，因而它会把 C 放到返回结果中。但是，一旦 return，就会弹出 A 当时的内核栈。这个时候，prev 变量就变成了 A，next 变量就变成了 B。这就还原了当年的场景，好在返回值里面的 last 还是 C。

通过三个变量 switch_to(prev = A, next=B, last=C)，A 进程就明白了，我当时被切换走的时候，是切换成 B，这次切换回来，是从 C 回来的。

总结时刻

这一节我们讲主动调度的过程，也即一个运行中的进程主动调用 __schedule 让出 CPU。在 __schedule 里面会做两件事情，第一是选取下一个进程，第二是进行上下文切换。而上下文切换又分用户态进程空间的切换和内核态的切换。



课堂练习

你知道应该用什么命令查看进程的运行时间和上下文切换次数吗？

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，**反复研读**。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院
云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 调度（上）：如何制定项目管理流程？

下一篇 17 | 调度（下）：抢占式调度是如何发生的？

精选留言 (13)

写留言



why

2019-05-03

15

- 调度, 切换运行进程, 有两种方式
 - 进程调用 sleep 或等待 I/O, 主动让出 CPU
 - 进程运行一段时间, 被动让出 CPU
- 主动让出 CPU 的方式, 调用 schedule(), schedule() 调用 __schedule()
 - __schedule() 取出 rq; 取出当前运行进程的 task_struct...

展开 ▾



安排

2019-05-03

6

proc文件系统里面可以看运行时间和切换次数，还可以看自愿切换和非自愿切换次数。

老师请教一个问题，A切到B, B切到C, C切到A, 当最后切换回A的时候，A要知道自己是从C切换过来的，也就是last，这样做的目的是什么呢？A要对C做什么善后操作吗？

展开 ▾



刘強

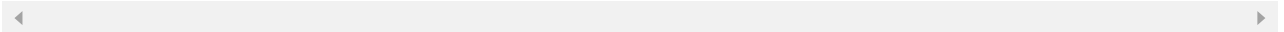
2019-05-05

👍 2

看了三遍，因为有一些基础，大概明白了。我觉得有个地方很巧妙。当函数返回的时候，由于切换了上下文，包括栈指针，所以一个进程函数执行return返回到了另一个进程，也就是完成了进程的切换。由此也可以看出，cpu也是比较"笨的"，它只提供了基本的机制，至于如何利用这种机制，玩出花样，那就是各个操作系统自由发挥了。

展开 ▾

作者回复: 是的，这一点比较绕



coyang

2019-05-03

👍 2

vmstat、pidstat 和 /proc/interrupts可以查看进程的上下文切换。

展开 ▾



youyui

2019-05-28

👍

用户态可以操作寄存器进行cpu上下文切换么

展开 ▾

作者回复: 不可以



憨人

2019-05-17

👍

进程切换需要搞明白：我从哪里来，我要到哪里去

展开 ▾

作者回复: 这句话赞



周平

2019-05-14



Linux 为关联一个 TSS用于进程切换，它在最底层有用到X86的x86 体系结构中，提供的以硬件的方式进行进程切换的模式吗？还是纯软件实现的？

作者回复: 没有用到硬件切换



尚墨

2019-05-08



刘老师，每个用户的进程都会被分配一个内核栈吗？

展开 ∨



川云

2019-05-07



这节看出老师真是功力深厚啊

展开 ∨



青石

2019-05-05



进程切换的过程，可不可以这么理解：

1. 在task_struct中的tss_struct是记录TSS段内容的，结构与寄存器结构相同，thread中保留了切换进程的时候需要修改的寄存器，当前任务寄存器TR指向的是当前运行进程的tss_struct; ...

展开 ∨

作者回复: 不是，tss启动后就不变了，在硬件看来没切换，软件仅仅修改变的几个寄存器



小龙的城堡





2019-05-04

老师讲的很清楚! 👍

展开 ▾



tiankonghe...

2019-05-04

学习了

展开 ▾



一笔一画

2019-05-03

老师, 我还是对三个参数不解, $A \rightarrow B \rightarrow C$, 如果再来一个D怎么办?

展开 ▾

作者回复: 不影响, 这里只站在a的角度看问题, 从a到b, 让后中间经历一万个进程, 然后到c再到a, 也是这个样子的

