Project2 non-preemptive+kernel 设计文档

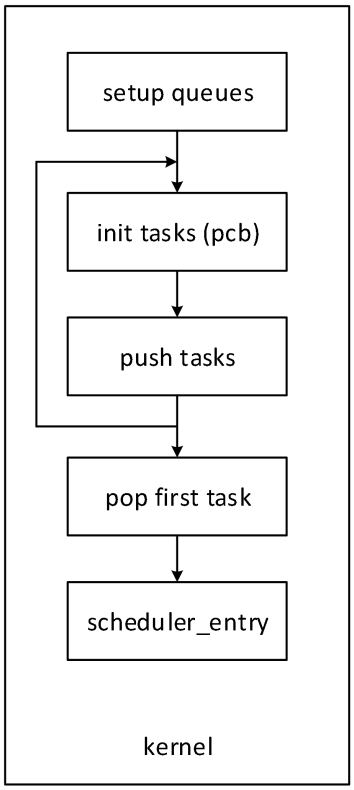
中国科学院大学
段宏键
2017.10.18

1. 1. 多 task 启动与 task switch 设计流程

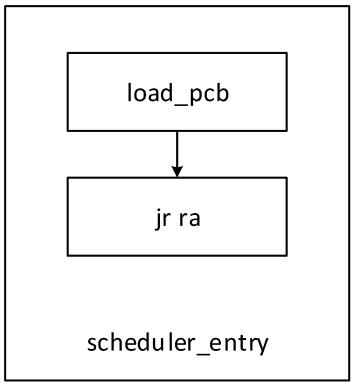
1.1 概述

我们首先看一下“Task 1 – 多 tasks 启动与 context switch”的总体情况，看一下我们这次任务都需要什么东西。首先，我们需要有一个队列，来保存可以运行的任务。

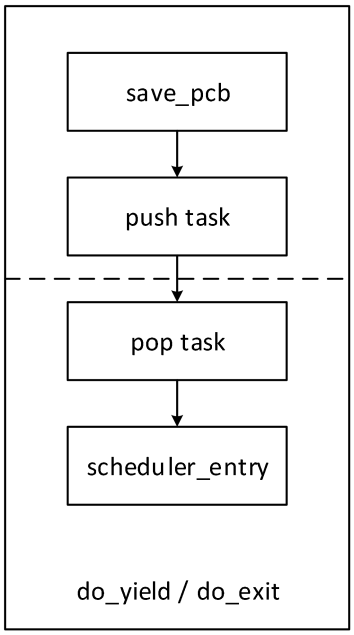
图为对列创建与相关信息初始化：



但是光有队列是不够的，因为操作系统还是不知道我们要先执行哪个任务，所以说我们还得做一个 scheduler 来告诉操作系统下一个执行的任务应该是什么。



当然，队列里的内容也不应该是一成不变的，所以我们还得做一些可以对队列进行修改的调度，就是 do_yield 和 do_exit。



1.2 队列

队列是什么东西呢？首先队列里应该有一些标志，用来显示队列的首尾位置和是否为空这些基本情况。然后，队列里应该有这对于任务的描述，也就是 pcb。所谓 pcb 相当于就是一个任务的档案，拿到这个档案，任务执行到哪里，任务接下来要做什么，操作系统都会一目了然。**pcb 中最重要的东西是一个叫 context 的结构。**这个结构里记录了 比较重要的寄存器和栈指针，有了这个东西我们就可以记录任务运行到哪里。这样就算任务的执行被中断了，我们也可以按图索骥，找到原来运行的位置，继续执行。

1.3 pcb

pcb 要怎么保存哪些信息且 pcb 如何保存呢？我是把 s0 到 s8 与 ra 和 sp 寄存器值中推进 pcb 的 context 的里，并改变 pcb 当前的状态，这样就保存了 pcb。保存的内容就是 pcb 中要有的东西。在这里需要注意一点就是 sp 与 ra 这两个寄存器保存时要小心，因为这两个寄存器一个时跳转相关的寄存器，一个是用与指示栈的寄存器。

1.4 运行流程

我们以 task1 为例，看看这个调度器究竟是怎么运行的。首先，我们跑到 kernel.c 中，完成队列的初始化，也就是将队列的几个标志填好，并要执行的几个任务的 pcb 填到队列里。然后我们就做了一个 schduler_entry，这个 schduler_entry 到位之后，调用 scheduler。Scheduler 会选取队列首位的任务（FIFO）作为我们马上要执行的 current_running。之后，继续 scheduler_entry，把要执行的函数的寄存器信息从栈提取到寄存器中，之后正好 jump 到刚从栈中拿出来的 ra 寄存器指示的地址，开始执行任务。一个进程任务运行到一定阶段，就可能会做一个 yield 或者 exit 操作进而进行系统调用。一个线程任务运行到一定阶段，可能就会做一个 do_yield 或者 do_exit 操作。这个 do_yield 可以保存 pcb，并且通过 schduler_entry 操纵 scheduler，实现任务的轮转。这里需要注意 do_yield 和 do_exit 的区别。Do_yield 操作是将当前运行的操作转移到队尾，而 do_exit 操作是直接将当前运行的操作从队列中踢出去。这样我们的任务就会不断地轮转，直到所有的任务都通过 do_exit 跳出队列为止。

1.5 注意事项

这个任务中，遇到的主要问题集中在 save pcb 这里，我们在进行栈操作的时候，一定要将 pcb 中的对应 context 的寄存器按顺序对应好。而且 do_yield()、block()和 unblock()都需要执行 save_pcb()，但 3 个函数创建栈帧时栈指针 sp 减小的量不同，ra 在栈中的位置也不同，所以需要用嵌入式汇编手动调整。

1.6 实验结果

实验完成后，可以看到小飞机飞过。

2. Context switch 开销测量设计流程

2.1 概述

这个任务比较简单，我们只需要在 process 和 thread 中用 get_timer 函数把时间记录下来，然后使用 util 的函数里定义的 printstr 函数和 printint 函数把测得的时间输出 到屏幕中就可以，其中要注意**用 printlocation 函数确定输出位置，避免后输出的内容覆盖掉先输出内容的信息。**

2.2 时间记录的位置

因为在这个试验中我们一共是有两个线程和一个进程。所以线程设置记录时间的位置可以定在第一个线程 do_yield 之前和第二个线程开始运行的时候。而 对于进程，我们再第二个线程 do_yield 之前到调用的进程一开始之间测量时间并求出差。我的代码在这一部分设计的不太好。测出的线程到进程的时间特别大，很有可能是我的设计中，进程与线程的执行顺序并不是向我想的那样按照 thrend4、thrend5 再到 process3 这个过程，我需要在进一步的调查。

2.3 实验结果

完成后可以在屏幕左上角看到函数运行的时间。

3. 互斥锁设计流程

3.1 概述

操作系统一般是有好几个进程和线程在同时跑的。如果这些进程/线程如果可以随意的互相干扰，那就一定会产生错误。比如如果我们两个进程同时对一个共享的资源进行修改，那我们就会难以预测修改之后的结果是什么。这样显然是不行的。所以我们要设计原子操作，使用锁的结构将不应该被打断的操作保护起来，从而使进程可以正确地运行。

3.2 具体操作

在我们的这个任务中，就是设计了一个简单的互斥锁。在 实验 task3 中，有几

个简单的操作，这些操作主要是进行了 lock_aquire 和 lock_release 这两个操作。Lock_aquire 操作会将锁的状态标记成 locked ;如果我们的锁此时已经是 locked 的了，那么我们就会将当前运行的任务扔进 block 队列里面。而 lock_release 操作则可以将 block 队列中的第一个任务取到 ready 队列的第一个来运行。在这个具体实现的过程中，我是直接使用 queue_pop 与 queue_push 函数进行操作的，且每当解开 lock 后，block 队列中全部的任务会 pop 出来并依次 push 进 ready 队列中。

3.3 注意事项

在这里我们需要注意一个问题就是 ,我们在将 block_queue 中第一个任务取到 ready 队列的时候需要先判断一下 block 队列是否为空。如果我们不进行判断，就会导 致程序提前结束。还有就是 block 队列和 ready 队列一定要赋好初值，如果不进行处理，那么 block 和 ready 就是指向同一地址的，这样相当于我们只有一个 ready 队列。

3.4 执行结果

设置 SPIN=FALSE ,task 中选任务 3 的 test ,可以得到正确的飞机图像。最终 test all 时在一定时间后屏幕上变化到数字 200 后会变为 pass，表明设计正确。

4. 关键函数

_Start： 对 pcb 和队列进行一些初始化，并最终跳到 scheduler_entry，开始调度

Scheduler： 会在 ready_queue 中取出运行时间最短的任务加载到 current_running 上

Scheduler_entry：会保存当前的 eip 并调用 scheduler，并把函数要用的参数传到通用寄存器

Save_pcb：会将通用寄存器中的值按顺序压进栈 ,并将其地址记录到 context 指针中

Do_yield：将正在运行的任务 push 到队尾，然后调用 scheduler_entry

Do_exit：将当前运行的任务标记为 exit，并调用 scheduler_entry

Lock_ acquire: 如果当前没锁，就获得锁。否则将当前操作放到 block 队列中

Lock_release：如果此时 block 队列里有任务，就将之取到 ready 队列队首

Block：把当前进行的操作放进 block 队列里

Unblock：如果 block 队列为空就什么都不做，如果非空就将任务取到 ready 队列头部

5. 参考文献

[1] Xv6-public-master