NachOS Lab2 实习报告

史杨勋惟

1200012741 信息科学技术学院

2016,04,12

目录

1 总体概述

这次 lab 的主要内容在于对 nachos 现有调度算法的深入理解和扩展。工作量相对于上次的 lab 多了一些。而工作内容则更偏向于底层。这次 lab 另一个特点是灵活，由于 Challenge 的存在，所以后面的设计是开放性的。不过也需要我们对调度算法有个更深入的理解。我在 Nachos 中整合了四个不同的调度算法，分别是原始的 FIFO，以及新增的基于优先级的抢占式，RR 算法，和一个简单的类似 Linux 的调度算法。

2 任务完成列表

Exercise 1	Exercise 2	Exercise 3	Challenge 1
Y	Y	Y	Y

3 完成情况

Exercise 1

调研 Linux 或 Windows 中采用的进程调度算法

我主要调研的是 Linux 的进程调度算法，因为在我的实现中加入了简单的类似 Linux 的调度算法。Linux 的调度算法为 CFS 算法，其主要设计思想是:

Linux 把所有要调度的进程在一个进程周期先后调度上。把进程的优先级看成两个不同的维度，一个是这个进程在单位调度周期中运行的时间，另一个则是这个进程在这个调度周期中被分配上 CPU 的顺序。其中，

- 执行时间由进程的给定 nice 值决定。
- 执行的先后由这个进程已运行的虚拟运行时间决定，这个虚拟运行时间差不多为实际运行时间除以 nice 值之后的值。

Linux 内核使用红黑树维护所有需要被调度上 CPU 的进程集合。其中红黑树使用地键值就是虚拟运行时间。

Linux 的调度算法还有很多复杂的细节，这里不详述了。把 CFS 算法完全移植到 Nachos 上是十分有难度的，不过我们可以借鉴上述的主要思想，为 Nachos 设计一个类似于 CFS 的调度算法。

Exercise 2

仔细阅读代码，理解 Nachos 现有的线程机制

Nachos 现有的线程机制是 FIFO。调度器 Scheduler 维护了一个就绪队列。每次 Fork 和 Yield 都会将相应的线程移动到这个队列的尾部，这就是一个简单的 FIFO 的调度算法。

当然 Nachos 有时间片的机制，只不过在常规执行下是没有激活的。使用 -rs 命令行参数可以激活。在时间片机制下，main.cc 初始化了一个全局的 timer 计时器，并且为其提供了相应的中断处理程序。观察 timer.h 和 timer.cc 我们可以发现它注册一个时钟中断处理程序，每次时钟中断发生后进入这个程序。这个处理程序会注册下一个时钟中断并且执行构造函数中用户给定的另一个程序。在 main.cc 中，这个程序就是激活 yieldOnReturn 使得每一次时钟中断发生后都会进入当前线程的 Yield，使其让出 CPU 留给下一个线程。

另外 Nachos 的一个时钟是由中断的一次开关触发的，这点坑了我好长的时间，后面会提及。

Exercise 3

线程算法调度扩展，实现基于优先级的抢占式调度算法

从底层设计上来看，基于优先级的抢占式调度算法和 FIFO 最大的不同在于两点: 优先级和抢占。

在 thread.h 和 thread.cc 中我加入并维护了 priority 成员，表示线程的优先级。其构造由可缺省的构造函数参数传入（C++ 默认参数机制）。

优先级体现在调度器的就绪队列的维护上。加入线程不仅仅是简单的插入队尾，而是需要根据优先级将其插入指定的位置。这里只需要修改 scheduler 的 ReadyToRun 函数，把 readlyList->Append 改为 readyList->SortedInsert 即可。

抢占体现在 Fork 的时候，我修改 Fork 代码如下:

```
1 void
2 Thread::Fork(VoidFunctionPtr func, int arg)
```

```
{
        ...
    /** Added **/
    if(sch_mode == 'p') {
        scheduler->PreemptRun(this);
    } else {
        scheduler->ReadyToRun(this);
    }

    (void) interrupt->SetLevel(oldLevel);
}
```

PreemptRun 如下设计

```
/** Added **/
void Scheduler::PreemptRun(Thread * thread) {
    printf("Trying to let %s preempt.\n", thread->getName());
    DEBUG('t', "Trying to let %s preempt.\n", thread->getName());
    if(currentThread->getPriority() > thread->getPriority()) {
        ReadyToRun(thread);
    } else {
        ReadyToRun(currentThread);
        Run(thread);
    }
}
```

由于此算法不涉及时间片，所以真正的调度是发生在 yield 点的。为了能准确观察算法的执行，我们需要在 threadtest 中添加 yield。同时还要提高 main 线程的优先级至 11。

我的 threadtest 中的代码如下。

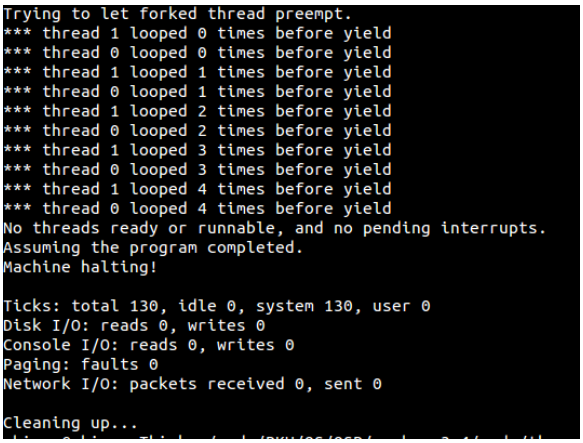
```
void
ThreadTest1()
{
    DEBUG('t', "Entering ThreadTest1");

    Thread *t = new Thread("forked thread", 13);
    // 13 is the priority, higher than "main"

    t->Fork(SimpleThread, 1);
    SimpleThread(0);
}
```

结果如下:

我们看到由于 fork 线程的优先级被提高了。所以先优先进行 fork 线程，抢占式的效果就出来了。



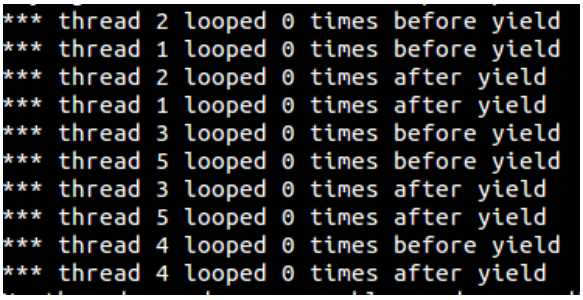
新加几个线程并简单改写 threadtest 如下:

```
void ThreadTest3() {
    printf("Creating thread 1\n");
    Thread *t1 = new Thread("forked 1 with 7", 8);
    t1->Fork(LongThread, 1);
    printf("Creating thread 2\n");
    Thread *t2 = new Thread("forked 2 with 8", 9);
    t2->Fork(SimpleThread, 2);
    printf("Creating thread 3\n");
    Thread *t3 = new Thread("forked 3 with 5", 7);
    t3->Fork(SimpleThread, 3);
    printf("Creating thread 4\n");
    Thread *t4 = new Thread("forked 4 with 2", 5);
    t4->Fork(SimpleThread, 4);
    printf("Creating thread 5\n");
    Thread *t5 = new Thread("forked 5 with 3", 6);
    t5->Fork(SimpleThread, 5);
}

void
SimpleThread(int which)
{
    int num;

    for (num = 0; num < 1; num++) {
        printf("*** thread %d looped %d times before yield\n", which, num);
       currentThread->Yield();
        printf("*** thread %d looped %d times after yield\n", which, num);
    }
}
```

结果如下：



我们可以看到这样的结果就是根据优先级两个高优先级的程序会相互交替执行。仔细分析我们会发现这种情况就是基于优先级的抢占式调度算法的普遍情况。

Exercise 4

增加全局线程管理机制

我由拓展设计了两个不同的调度算法，分别是时间片轮转和简单 CFS 的调度算法。实现这两个算法主要的涉及点是：

- 对于 RR 算法，我们只需要初始化一个 timer 并激活 YieldOnReturn 即可。这一点可以和原来的 rs 模式复用。
- 对于 CFS，我们可以做一下简化。每次调度运行 priority 个周周期，并且 vruntime+1，就绪队列根据 vruntime 进行排序。为 thread 设计一个 getKey 接口后，CFS 就可以和基于优先级的抢占式算法复用优先级接口。

我用一个全局变量来区分调度算法，由命令行参数决定。由于之前提到的 yieldOnReturn 机制，所以我们对代码的改写只需要在 yield 中进行即可。

修改后的主要代码如下：

```
1 Thread::Thread(char* threadName, int _priority, int _uid)
2 {
3         ...
4     if(!strcmp(name, "main")) {
5         priority = 11;
6     }
7     lastTick = currentTick = 0;
```

```
8       if(sch_mode == 'l') {
9           tickStep = priority;
10      } else {
11          tickStep = 1;
12      }
13  }
14
15  int Thread::getKey() {
16      if(sch_mode == 'r' || sch_mode == 'f') return 0;
17      if(sch_mode == 'p') return -priority;
18      if(sch_mode == 'l') {
19          return (currentTick / priority);
20      }
21  }
22
23  void
24  Thread::Yield ()
25  {
26          ...
27      /** Added **/
28      bool change = false;
29      TS();
30      currentTick++;
31      if(currentTick - lastTick == tickStep) {
32          lastTick = currentTick;
33          change = true;
34      } else {
35          change = false;
36      }
37      if(sch_mode == 'f' || sch_mode == 'p') {
38          change = true;
39      }
40      DEBUG('t', "Cur, Lst = %d %d\n", currentTick, lastTick);
41      if(change) {
42          nextThread = scheduler->FindNextToRun();
43          if (nextThread != NULL) {
44          scheduler->ReadyToRun(this);
45          scheduler->Run(nextThread);
46          }
47      } else {
48          scheduler->Run(this);
49      }
50      DEBUG('t', "Before SET");
51      (void) interrupt->SetLevel(oldLevel);
52      DEBUG('t', "End Yielding");
53  }
54
```

```
55 void
56 Scheduler::ReadyToRun (Thread *thread)
57 {
58     DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());
59
60     thread->setStatus(READY);
61     readyList->SortedInsert((void *)thread, thread->getKey());
62 }
```

这里有个小 trick，就是对 yield 的理解。yield 主要功能是主动让出 CPU，但是这个功能是留给用户的还是留给系统的有待斟酌。我的理解是后者，这个 yield 并不是 sleep，不应该由用户发起，所以 yield 的主要作用就应该是留给系统在调度中使用的。所以在 threadtest 中我并没有插入 yield。我的第一个 threadtest 如下设计：

我的测试 threadtest 如下设计：

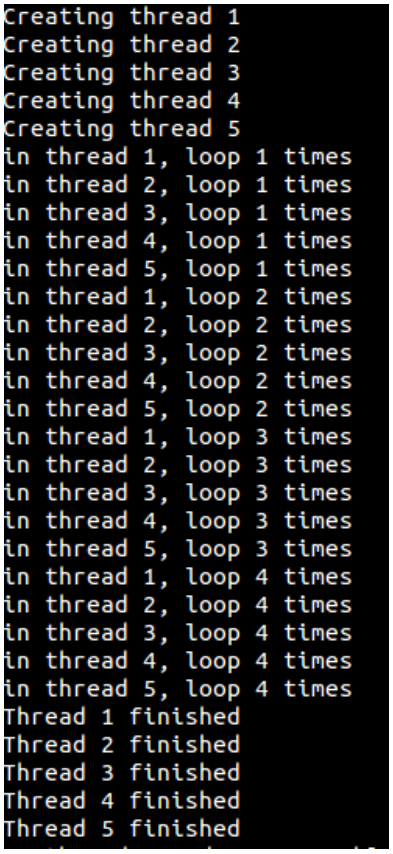
```
1  void ThreadTest3() {
2      printf("Creating thread 1\n");
3      Thread *t1 = new Thread("forked 1 with 7", 8);
4      t1->Fork(LongThread, 1);
5      printf("Creating thread 2\n");
6      Thread *t2 = new Thread("forked 2 with 8", 9);
7      t2->Fork(LongThread, 2);
8      printf("Creating thread 3\n");
9      Thread *t3 = new Thread("forked 3 with 5", 7);
10     t3->Fork(LongThread, 3);
11     printf("Creating thread 4\n");
12     Thread *t4 = new Thread("forked 4 with 2", 5);
13     t4->Fork(LongThread, 4);
14     printf("Creating thread 5\n");
15     Thread *t5 = new Thread("forked 5 with 3", 6);
16     t5->Fork(LongThread, 5);
17 }
18
19 void LongThread(int tid) {
20     DEBUG('t', "begin %d\n", tid);
21     while(currentThread->getCurrentTick() < 5) ;
22     printf("Thread %d finished\n", tid);
23     return;
24 }
```

结果是：**死机了！**

于是我打开各种调试信息，发现当一个线程进入 while 循环后就没有时钟中断发生了。我百思不得其解只能深入探索 Nachos 的中断机制。终于

找到了原因所在，那就是时钟是由中断开关触发的（我觉得非常不合理）。

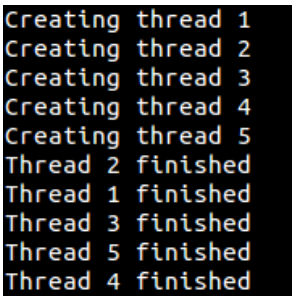
于是我在 LongThread 中加入了两句 SetLevel，成功输出了。RR 的结果（加入了 printf 细节）是：



可见轮转正常。RR 不需要优先级，否则又是两个高优先级的轮着运行了。

CFS 的结果（把结束判断的 5 改成了 99）是：

可见高优先级的线程的确是最先结束的。这里由于篇幅限制我省略了详细的 printf 输出，不过输出表示整个运转是正常的。



	FIFO	Preempt	RoundRobin	CFS
命令行参数	f	p	r	l
排序键值	无	priority	无	currentTick/priority
是否触发时间片	N	N	Y	Y

各调度算法总结

4 遇到的困难以及解决办法

底层代码

之前助教也提到过，Nachos 是建立在虚拟机上的，所以和一个真实操作系统还是有所差距的。尤其到了底层代码上，虚拟机并不直接操作硬件，所以理解上我也碰到了一些问题。其中最主要的就是对 Nachos 中断机制。之前也提到了，我在 Nachos 的时钟上卡了好久，这也是我在整个代码设计中最大的困难了。

参数调整和测试设计

由于不同的参数导致的结果会有不同，所以到底选择哪些参数比较好还是需要斟酌一下的。我最后选择的 TimerSlice 是 20，用户线程的优先级是 1-10，main 线程的优先级是 11。由于这些参数主要是宏定义，所以修改起来比较简单，但调试和测试的过程是比较麻烦的。

调度算法的细节理解

虽然在操作系统课上我们学过了很多的调度算法，也对调度算法有了一个比较全面的了解。但是这种了解到了实习课上还是远远不够的。由于涉及的代码更底层，所以我们还需要对每个算法细节又理解。需要深入理

解 readylist 上个进程的具体执行顺序，以及各种操作都会触发调度算法的哪些行为。还是有一些预备知识需要积累的。

各算法的整合

另外一个需要考虑的就是整个 nachos 上每个调度算法的横向兼容性。在我的代码里，这些算法的决定是由命令行参数决定的。我为 nachos 设计了一个全局变量 schmode 来标识使用的调度算法。我觉得这样的设计是最符合常规的。

5 收获与感想

总的来说，这次的 lab 还算是一个比较复杂的 lab。需要设计的代码量比较大，也需要比较仔细地去分析。与此同时由于更接近底层，所以调试时间也比较长。我还是花了不少时间在这个 lab 上的。

不过虽然复杂，但是由于其涉及的内容仅仅是调度算法，所以问题的广度不算太大。如果在设计之前对每个调度算法先进行简单的调研以及思考，代码设计的整个过程也还是比较清晰的。

当然，虽然我整合了四个调度算法，但是都只是进行了简单的实现。效率也暂时没有进行深入评测。如果再给我写一次，或许我会深入钻研一种调度算法并进行一些高效的优化。

6 意见与建议

无。

7 参考资料

- 《现代操作系统》
- 《Linux 内核设计与实现》