山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202022300316 | 姓名：王丁 | | 班级：软工20.6 |
| 学号：202000300358 | 姓名：梁思睿 | | 班级：软工20.6 |
| 实验编号：Lab2 | | | |
| 实验题目：具有优先级的线程调度 | | | |
| 实验学时：4 | | 实验日期：2022.xx.xx | |
| 实验目的：  1. 分析说明Nachos原有的线程调度策略。  2. 设计并实现具有静态优先级的非抢占式线程调度策略。  3. 以线程调试模式运行Nachos(./nachos -d t)，研究调试输出信息。上下文切换的次数与被测线程SimpleThread中打印输出的总行数一致吗？多余或缺少的上下文切换次数是什么原因造成的？请修改代码减少上下文切换的次数与被测线程SimpleThread中打印输出的总行数的差距。  4. 在实现了前面优先级调度的基础上，若要求实现优先级调度的老化(aging)，请给出在Nachos中实现的具体方法(不要求实现可运行的代码。在实验报告中用文字描述即可，必要时可在文字中结合关键代码片段、数据结构、对象等说明)。 | | | |
| 硬件环境：  设备名称 LAPTOP-UUHJO71O  处理器　　AMD Ryzen 5 4600U with Radeon Graphics 2.10 GHz  机带 RAM 16.0 GB (15.4 GB 可用)  设备 ID 63412E57-FC9E-4F92-A869-B35FDEF3303C  产品 ID 00342-35837-59360-AAOEM  系统类型 64 位操作系统, 基于 x64 的处理器  硬盘 512GB SSD | | | |
| 软件环境：  宿主机：Windows 10 21H2 64位  虚拟机软件：VMware Workstation Pro 16.1.2 build-17966106  Linux：Ubuntu 14.04.6 LTS Desktop i386 (Trusty Tahr)  gcc/g++：(Ubuntu 4.8.4-2ubuntu1~14.04.4) 4.8.4  MIPS交叉编译器：gcc-2.8.1-mips.tar.gz  Nachos：Nachos-3.4-UALR-2022 | | | |
| 实验步骤与内容：  **概述**  本次实验要求先学会Nachos原有的线程调度策略，再实现具有静态优先级的非抢占式调度，即就绪队列中的线程按照给定的静态优先级排序，并且上处理机的线程直到运行完后才让出处理机。之后要通过查看原有的线程调试信息，分析原有线程在上下文切换方面的不合理之处。最后要  作为第一个改动源代码的正式实验，本次实验需要再次审阅nachos源码、结合自己的理解，并且积极回忆、运用上学期所学操作系统知识才能完成。 ****分析说明Nachos原有的线程调度策略**** 首先需要阅读源码，找到相关文件。  源码中和线程调度相关的文件有Thread类、Scheduler类和List类。   1. Thread   Thread具有四种状态：  enum ThreadStatus { JUST\_CREATED, RUNNING, READY, BLOCKED };  关于线程的调度，我们只需要关注RUNNING, READY 两种状态即可。  栈的分配和线程控制块的初始化工作是由函数 Fork()完成的。即 Fork 一个线程会创建该线程并将其设置为可运行（会调用 schedule 的 ReadyToRun 方法）。  Fork 携带的第一个参数 func 是一个指向将要执行的线程函数的指针。这里用typedef语法规定了函数需要满足的条件。  typedef void (\*VoidFunctionPtr)(\_int arg);  typedef void (\*VoidNoArgFunctionPtr)();  只要函数满足这个条件，就可以包装成这样一个线程来运行。  再来看让出CPU的yield方法：  void Thread::Yield() {      Thread \*nextThread;      IntStatus oldLevel = interrupt->SetLevel(IntOff);      ASSERT(this == currentThread);      DEBUG('t', "Yielding thread \"%s\"\n", getName());      nextThread = scheduler->FindNextToRun();      if (nextThread != NULL) {          scheduler->ReadyToRun(this);          scheduler->Run(nextThread);      }      (void)interrupt->SetLevel(oldLevel);  }  线程在yield的时候，将就绪队列的头指针移动到指向下一个要上处理机的线程， 然后当前线程 this运行完后回到队列，然后调度 第一个就绪的线程上处理机。   1. Scheduler   Scheduler是负责线程调度的类，也是另一个我们需要修改的类。先来看原系统的实现。  基本上只需要关注两个方法即可，一个是ReadyToRun，另一个是FindNextToRun。  void Scheduler::ReadyToRun(Thread \*thread) {      DEBUG('t', "Putting thread %s on ready list.\n",  thread->getName());      thread->setStatus(READY);      readyList->Append((void \*)thread);  }  Thread \*Scheduler::FindNextToRun() {  return (Thread \*)readyList->Remove();  }  ReadyToRun其实就是把一个线程改成就绪的状态，然后放到队列里。通过观察FindNextToRun可知，nachos原来的线程调度只是简单的先来先服务FIFO调度。   1. List类   这个类中已经给出了根据SortKey排序的方法：  void List::SortedInsert(void \*item, int sortKey) {      ListElement \*element = new ListElement(item, sortKey);      ListElement \*ptr;       // keep track      if (IsEmpty()) {    // if list is empty, put          first = element;          last = element;      } else if (sortKey < first->key) {      element->next = first;      first = element;      } else {        // look for first elt in list bigger than item          for (ptr = first; ptr->next != NULL; ptr = ptr->next) {              if (sortKey < ptr->next->key) {          element->next = ptr->next;              ptr->next = element;          return;          }      }      last->next = element;       // item goes at end of list      last = element;      }  }  如果需要改进，可以直接利用这个方法来达到非FIFO式的线程调度。 ****设计并实现具有静态优先级的非抢占式线程调度策略**** 在动手之前，需要仔细阅读教程，查看如何修改原来的nachos代码、如何编译已经修改了的nachos系统。通过阅读nachos makefile结构可知，各个lab中的makefile已经有了先在本目录下查找头文件的逻辑。那么我们需要修改哪个，就把哪个部分的h文件和cc文件放到lab里面即可。  思路：首先线程需要有个域用来存放优先级，然后调度算法的FindNextToRun也需要根据优先级有所改动。  这里，我们采用宏的方式定义一种优先级类型。规定最小（最优先）是 0 ，最大（最不优先）是99，超出界限的取界限值。  我们把priority设置为私有变量，所以也设置了相应的get、set函数。  #ifndef THREAD\_PRIORITY  #define THREAD\_PRIORITY int     // type of thread priority  #endif  然后修改默认的构造方法，使得新的线程具有一个默认的中等优先级。    Thread::Thread(const char \*threadName) {      name = (char \*)threadName;      stackTop = NULL;      stack = NULL;      status = JUST\_CREATED;  #ifdef THREAD\_PRIORITY      this->priority = 9;  // default priority is 9  #endif  #ifdef USER\_PROGRAM      space = NULL;  #endif  }  加上一个有参数的构造方法：  Thread::Thread(const char \*threadName, THREAD\_PRIORITY priority\_) {      name = (char \*)threadName;      stackTop = NULL;      stack = NULL;      status = JUST\_CREATED;      // make sure priority is in range [0, 99]      priority\_ = (priority\_ < 0) ? 0 :  (priority\_ > 99) ? 99 : priority\_;      this->priority = priority\_;  }    那么接下来只需要修改一下ReadyToRun即可。  void Scheduler::ReadyToRun(Thread \*thread) {      DEBUG('t', "Putting thread %s on ready list.\n",  thread->getName());      thread->setStatus(READY);      readyList->SortedInsert((void \*)thread,  thread->getPriority());  }  除此之外还要更改Thread::Yield方法，因为这里原来的逻辑是先指定下一个、再把自己重新设置为就绪，这实际上会导致静态优先级不被严格遵守。需要改成先把自己的状态改成就绪，再选择下一个。  修改后的方法如下：  void Thread::Yield() {      Thread \*nextThread;      IntStatus oldLevel = interrupt->SetLevel(IntOff);      ASSERT(this == currentThread);      DEBUG('t', "Yielding thread \"%s\"\n", getName());      scheduler->ReadyToRun(this);      nextThread = scheduler->FindNextToRun();      if (nextThread != NULL) {          scheduler->Run(nextThread);      }      (void)interrupt->SetLevel(oldLevel);  }  还有最后一步工作需要完成，那就是修改测试文件ThreadTest。 ****以线程调试模式运行Nachos(./nachos -d t)，研究调试输出信息**** 更改后的线程测试函数为:  void ThreadTest() {      printf("ThreadTest in LAB2, Starting!\n");      DEBUG('t', "Priority Test in LAB2, Starting!\n");      for (int i = 0; i < 4; i++) {          char\* name = new char[10];          sprintf(name, "Thread %d", i);          Thread \*t = new Thread(name, i);          t->Fork(SimpleThread, i);      }      // SimpleThread(0);  }  运行 SDU-OS-Project/code/lab2$ ./nachos -d t，观察结果。输出比较长，具体内容详见附录1。  观察输出，会发现一共Switch了24次，但所有线程加起来一共只loop了20次，上下文切换的次数与被测线程SimpleThread中打印输出的总行数不一致。  　　仔细观察不难发现，存在这种情形的调度：  Switching from thread "Thread 0" to thread "Thread 0"  Now in thread "Thread 0"  Finishing thread "Thread 0"  Sleeping thread "Thread 0"  Switching from thread "Thread 0" to thread "Thread 1"  究其原因，是因为在每个线程的最后一轮循环结束之前，还会调用一下Yield方法，等到下次轮到自己的时候其实已经无事可做了，下次轮到自己便直接跳出循环、结束了线程，没有任何输出。  要想使得上下文切换的次数与被测线程SimpleThread中打印输出的总行数一致，只需修改SimpleThread，当这次循环是最后一次的时候直接退出即可：  void SimpleThread(\_int which) {      int num;      int times\_run = 5;      for (num = 0; num < times\_run; num++) {          printf("\*\*\* thread %d looped %d times\n",  (int)which, num);          if (num != times\_run - 1) {          // if not the last time, yield the CPU to another thread              currentThread->Yield();          }      }  }  再次查看输出（详见附录2），线程上下文切换次数已经和loop总次数一致。 ****若要求实现优先级调度的老化，给出在Nachos中实现的具体方法**** 思路：线程的老化是指随着时间不断增加，线程的优先级也逐渐提高，以防止优先级低的线程一直等待而无法执行（starvation）。  在nachos中，要想实现线程的aging，首先要考虑这个时间该如何获取.  如果是Windows或者Linux系统，我们可以调用类似于gettimeofday()的方法，来获取一个墙上时钟时间，进而根据这个提升线程的优先级。  但是nachos显然不能这么做。通过阅读machine.timer可知，并没有提供墙上时钟的方法，其他地方也未能发现类似的时间函数，不过有一个TimeOfNextInterrupt的方法，可以知道下次中断的相对时间。  利用这个来实现线程的老化，其实还不如在每次Scheduler调用FindNextToRun的时候，给自己的一个int私有变量+1，如果私有变量达到了某个限度就归0，并且执行aging操作。  这个思路转换为代码也不难。只不过是测试的时候，涉及aging，需要写很多测试代码才能验证。  具体如下：  Scheduler.h class Scheduler {     public:      .. .. ..     private:      List\* readyList;  // queue of threads that are ready to run,                        // but not running     #ifdef THREAD\_AGING        int agingTicks;     #endif  };    Scheduler.cc  Thread \*Scheduler::FindNextToRun() {  #ifdef THREAD\_AGING      agingTicks++;      if (agingTicks >= 100) {          agingTicks = 0;          ListElement \*e = readyList->getFirst();          while (e != NULL) {              Thread \*t = (Thread \*)e->item;  // priority is in range [0, 99],  //smaller number has higher priority.              bool isHighestPriority = t->getPriority() == 0;              if (!isHighestPriority) {                  t->setPriority(t->getPriority() - 1);              }              e = e->next;          }      }  #else      return (Thread \*)readyList->Remove();  #endif  }  这里使用 agingTicks >= 100 而不是 agingTicks == 100 ，以防止由于未知原因跳过 100 这个数字发生、产生逻辑错误的情况。  至此，实验2的全部内容已经完成。 | | | |
| 结论分析与体会：  通过这次实验,我们对于nachos的线程调度有了彻底的认识、了解和使用了线程相关的系统调用、成功修改了nachos的线程调度算法并正确运行，并且优化了Simple Thread线程工作以减少不必要的上下文切换、提出了切实可行的线程老化优化思路、给出了可用于实战的代码实现。  在实验之前，我们花费了大量时间再次仔细阅读源码、修改Makefile、利用git和GitHub私有代码库进行协作，不仅对OS的内核有了进一步的认识，还同时学习了git、makefile等知识，可谓收获良多。 | | | |

附录1：更改调度策略之后，修改SimpleThread之前，测试文件的调试模式输出：

**beihai@ubuntu:/mnt/hgfs/SDU-OS-Project/code/lab2$ ./nachos -d t**

ThreadTest in LAB2, Starting!

Priority Test in LAB2, Starting!

Forking thread "Thread 0" with func = 0x5655a30b, arg = 0

Putting thread Thread 0 on ready list.

Forking thread "Thread 1" with func = 0x5655a30b, arg = 1

Putting thread Thread 1 on ready list.

Forking thread "Thread 2" with func = 0x5655a30b, arg = 2

Putting thread Thread 2 on ready list.

Forking thread "Thread 3" with func = 0x5655a30b, arg = 3

Putting thread Thread 3 on ready list.

Finishing thread "main"

Sleeping thread "main"

Switching from thread "main" to thread "Thread 0"

\*\*\* thread 0 looped 0 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

Deleting thread "main" with priority 9

\*\*\* thread 0 looped 1 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

\*\*\* thread 0 looped 2 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

\*\*\* thread 0 looped 3 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

\*\*\* thread 0 looped 4 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

Finishing thread "Thread 0"

Sleeping thread "Thread 0"

Switching from thread "Thread 0" to thread "Thread 1"

\*\*\* thread 1 looped 0 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

Deleting thread "Thread 0" with priority 0

\*\*\* thread 1 looped 1 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

\*\*\* thread 1 looped 2 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

\*\*\* thread 1 looped 3 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

\*\*\* thread 1 looped 4 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

Finishing thread "Thread 1"

Sleeping thread "Thread 1"

Switching from thread "Thread 1" to thread "Thread 2"

\*\*\* thread 2 looped 0 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

Deleting thread "Thread 1" with priority 1

\*\*\* thread 2 looped 1 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

\*\*\* thread 2 looped 2 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

\*\*\* thread 2 looped 3 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

\*\*\* thread 2 looped 4 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

Finishing thread "Thread 2"

Sleeping thread "Thread 2"

Switching from thread "Thread 2" to thread "Thread 3"

\*\*\* thread 3 looped 0 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

Deleting thread "Thread 2" with priority 2

\*\*\* thread 3 looped 1 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

\*\*\* thread 3 looped 2 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

\*\*\* thread 3 looped 3 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

\*\*\* thread 3 looped 4 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

Finishing thread "Thread 3"

Sleeping thread "Thread 3"

No threads ready or runnable, and no pending interrupts.

Assuming the program completed.

Machine halting!

Ticks: total 290, idle 0, system 290, user 0

Disk I/O: reads 0, writes 0

Console I/O: reads 0, writes 0

Paging: faults 0

Network I/O: packets received 0, sent 0

Cleaning up...

附录2：更改调度策略、更改SimpleThread之后，测试文件的输出

**beihai@ubuntu:/mnt/hgfs/SDU-OS-Project/code/lab2$ ./nachos -d t**

ThreadTest in LAB2, Starting!

Priority Test in LAB2, Starting!

Forking thread "Thread 0" with func = 0x5660630b, arg = 0

Putting thread Thread 0 on ready list.

Forking thread "Thread 1" with func = 0x5660630b, arg = 1

Putting thread Thread 1 on ready list.

Forking thread "Thread 2" with func = 0x5660630b, arg = 2

Putting thread Thread 2 on ready list.

Forking thread "Thread 3" with func = 0x5660630b, arg = 3

Putting thread Thread 3 on ready list.

Finishing thread "main"

Sleeping thread "main"

Switching from thread "main" to thread "Thread 0"

\*\*\* thread 0 looped 0 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

Deleting thread "main" with priority 9

\*\*\* thread 0 looped 1 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

\*\*\* thread 0 looped 2 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

\*\*\* thread 0 looped 3 times

Yielding thread "Thread 0"

Putting thread Thread 0 on ready list.

Switching from thread "Thread 0" to thread "Thread 0"

Now in thread "Thread 0"

\*\*\* thread 0 looped 4 times

Finishing thread "Thread 0"

Sleeping thread "Thread 0"

Switching from thread "Thread 0" to thread "Thread 1"

\*\*\* thread 1 looped 0 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

Deleting thread "Thread 0" with priority 0

\*\*\* thread 1 looped 1 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

\*\*\* thread 1 looped 2 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

\*\*\* thread 1 looped 3 times

Yielding thread "Thread 1"

Putting thread Thread 1 on ready list.

Switching from thread "Thread 1" to thread "Thread 1"

Now in thread "Thread 1"

\*\*\* thread 1 looped 4 times

Finishing thread "Thread 1"

Sleeping thread "Thread 1"

Switching from thread "Thread 1" to thread "Thread 2"

\*\*\* thread 2 looped 0 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

Deleting thread "Thread 1" with priority 1

\*\*\* thread 2 looped 1 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

\*\*\* thread 2 looped 2 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

\*\*\* thread 2 looped 3 times

Yielding thread "Thread 2"

Putting thread Thread 2 on ready list.

Switching from thread "Thread 2" to thread "Thread 2"

Now in thread "Thread 2"

\*\*\* thread 2 looped 4 times

Finishing thread "Thread 2"

Sleeping thread "Thread 2"

Switching from thread "Thread 2" to thread "Thread 3"

\*\*\* thread 3 looped 0 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

Deleting thread "Thread 2" with priority 2

\*\*\* thread 3 looped 1 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

\*\*\* thread 3 looped 2 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

\*\*\* thread 3 looped 3 times

Yielding thread "Thread 3"

Putting thread Thread 3 on ready list.

Switching from thread "Thread 3" to thread "Thread 3"

Now in thread "Thread 3"

\*\*\* thread 3 looped 4 times

Finishing thread "Thread 3"

Sleeping thread "Thread 3"

No threads ready or runnable, and no pending interrupts.

Assuming the program completed.

Machine halting!

Ticks: total 250, idle 0, system 250, user 0

Disk I/O: reads 0, writes 0

Console I/O: reads 0, writes 0

Paging: faults 0

Network I/O: packets received 0, sent 0

Cleaning up...