山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202100300063 | 姓名：李彦浩 | | 班级：工软 21 |
| 学号：202100300340 | 姓名：黄幸兒 | | 班级：工软 21 |
| 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：2023.10.29 | |
| 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：  联想Thinkpad笔记本  Intel Core i5-8250 CPU 8核  8GB内存  1907GB 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： 编译条件 可能我们想通过Makefile来实现不同的条件编译，比如说将线程调度的策略修改为Round Robin，或者其他一些策略，我觉得都提到策略了，其实最好是用面向对象的方式去写，比如注入一个Strategy对象，这样不至于一大堆分支判断和处理的代码堆在一起。即便不用OO，那最起码也应该借助抽象来编码，但nachos源码采用的解决方案就是在代码里留一大堆条件编译的语句，然后把逻辑全部堆到一块儿，我个人是很不想这么做的。  我们需要确定三种不同的条件：   1. 是否启用时间片 2. 线程调度的策略（包括一些细节） 3. 是否允许抢占式调度   其实我不确定是否应该把“启用时间片”和“允许抢占式调度”拆开来，因为目前来看，启用时间片本质上就代表了不允许抢占。这是一件显而易见的事情：如果你的时间片没有结束，CPU不会强制剥夺你的使用权，这就已经保证不允许抢占了。  但可能可以这样理解抢占：上学期对于preemptive的定义是，一个线程/进程可能并不能一次性做完他的CPU Burst。所以如果我们在他时间片结束时重新去寻找优先级更高的线程（当然他可能仍然还是优先级最高的线程），然后执行那个优先级更高的线程，这本质上其实也没有让当前的线程做完它的CPU Burst。符合了抢占式的定义。所以如果是这样的话，时间片和抢占式调度其实就没有耦合了。  我最终选择了上面的第二种理解。对于编译条件，当前实现了如下几个。   |  | | --- | | # we should define 3 things  # first,if we use time slice  # second,which CPU scheduling strategy should we choose  # including the details  # third,if the CPU scheduling is preemptive  # first  # -DTIMESLICE decide if we should dispatch time slices for current thread  # second  # 2.1  # -DPRIORITY use priority strategy  # -DAGING after you define we do the CPU scheduling with priority,you should point out whether we use static or dynamic policy.Default is static  # 2.2  # -DROUNDROBIN use round robin strategy  # third  # -DPREEMPTIVE the default sheduling policy is NONPREEMPTIVE |   请注意，里面还是有一些耦合没有处理的。  比如说，RR调度很明显需要时间片，没有时间片还谈什么RR，不就变成FCFS了？但我在实现时没有管这个（也许会影响开发效率），因此，如果你想做RR，请保证条件编译时带了启用时间片的条件。  大概长这样：   |  | | --- | | DEFINES += -DROUNDROBIN -DTIMESLICE |   说实话，如果不加TS的条件，会发生什么还真没测试过。 1.1 原有线程调度策略 原有线程调度策略分为两种：   1. 不带”-rs”选项的**FCFS(First Come First Serve)调度** 2. 带”-rs”选项的**Random Round Robin调度**  1.1.0. 时钟模块注入 如果你在启动nachos时不带-rs选项，比如：   |  | | --- | | ./nachos //假设nachos二进制可执行文件就在你目前cd到的文件夹下 |   此时在nachos虚拟机初始化时，将：   |  | | --- | | //system.cc :: Initialize  //...  bool randomYield = FALSE;  //...  else if (!strcmp(\*argv, "-rs")) {  ASSERT(argc > 1);  RandomInit(atoi(\*(argv + 1))); // initialize pseudo-random  // number generator  randomYield = TRUE;  argCount = 2;  }  //...  if (randomYield) // start the timer (if needed)  timer = new Timer(TimerInterruptHandler, 0, randomYield); |  1. 初始化一个选项开关randomYield，代表是否要让线程根据随机时钟中断放弃CPU 2. 查询nachos启动时是否带随机时间片种子-rs选项，如果是，则重置这个开关为True 3. 最后在初始化为nachos虚拟机注入依赖模块时（比如stats统计模块\interrupt中断模块等），将检查开关，如果打开开关，将注入一个产生随机时钟中断的时钟实例timer；如果没有打开开关，那么本次nachos虚拟机将没有时钟模块。  1.1.1. FCFS调度 根据上一小节的内容，如果nachos虚拟机没有时钟模块，将进行FCFS调度。  首先，如果没有时钟，那么也不会有时钟中断，也就是说每个线程获得的时间片是无限长的，直到它执行完任务为止。  接下来我们看一下一个没有时钟中断的线程的生命周期：  先Fork出来，在Fork时要为这个线程分配栈空间：   |  | | --- | | //thread.cc :: Fork  //...  StackAllocate(func, arg);  //thread.cc :: StackAllocate  //...  machineState[PCState] = (\_int) ThreadRoot;  machineState[StartupPCState] = (\_int) InterruptEnable;  machineState[InitialPCState] = (\_int) func;  machineState[InitialArgState] = arg;  machineState[WhenDonePCState] = (\_int) ThreadFinish; |  1. 先分配这个线程开始执行处的程序计数器，让他指向一段汇编指令ThreadRoot，这代表了系统第一次切换到这个线程时的执行位置 2. 分配是否允许中断的参数 3. 分配这个线程执行的函数的栈空间，以及函数参数的栈空间 4. 分配这个线程结束时调用的函数的栈空间   这里关键点在于4，该线程执行完我们要求它执行的函数，然后该干什么。   |  | | --- | | //thread.cc  static void ThreadFinish() { currentThread->Finish(); }  //thread.cc  void  Thread::Finish ()  {  (void) interrupt->SetLevel(IntOff);  ASSERT(this == currentThread);    DEBUG('t', "Finishing thread \\"%s\\"\\n", getName());    threadToBeDestroyed = currentThread;  Sleep(); // invokes SWITCH  // not reached  }  //thread.cc  void  Thread::Sleep ()  {  Thread \*nextThread;    ASSERT(this == currentThread);  ASSERT(interrupt->getLevel() == IntOff);    DEBUG('t', "Sleeping thread \\"%s\\"\\n", getName());  status = BLOCKED;  while ((nextThread = scheduler->FindNextToRun()) == NULL)  interrupt->Idle(); // no one to run, wait for an interrupt    scheduler->Run(nextThread); // returns when we've been signalled  } |   可以看到ThreadFinish令当前线程调用Finish函数，而Finish函数设置了当前线程将被摧毁，（都结束了当然要释放空间）然后调用了Sleep函数。  Sleep函数阻塞住当前线程，随后查询线程就绪列表中是否有线程要执行，如果没有那么他会反复调用Idle函数，这个Idle函数的执行可能有两种影响：   |  | | --- | | void  Interrupt::Idle()  {  DEBUG('i', "Machine idling; checking for interrupts.\\n");  status = IdleMode;  if (CheckIfDue(TRUE)) { // check for any pending interrupts  while (CheckIfDue(FALSE)) // check for any other pending  ; // interrupts  yieldOnReturn = FALSE; // since there's nothing in the  // ready queue, the yield is automatic  status = SystemMode;  return; // return in case there's now  // a runnable thread  }  // if there are no pending interrupts, and nothing is on the ready  // queue, it is time to stop. If the console or the network is  // operating, there are \*always\* pending interrupts, so this code  // is not reached. Instead, the halt must be invoked by the user program.  DEBUG('i', "Machine idle. No interrupts to do.\\n");  printf("No threads ready or runnable, and no pending interrupts.\\n");  printf("Assuming the program completed.\\n");  Halt();  } |   可以看到，如果此时没有线程，那么它将反复查询是否还有未处理的中断，也就是调用CheckIfDue函数，CheckIfDue函数在上一次实验已经介绍过了，不再赘述。如果连中断也都处理完了，（或者只剩下一个守护时间片了）那么就应该停机。否则就处理中断。  如果停机了，一切都结束了，这个比较直接。但如果还有中断未处理完，那么CheckIfDue将返回False。我们仍处于Sleep函数的循环中。  直到（或者本来就有）一个就绪的线程处于readyList中，我们调度它执行，也就是Run(nextThread)函数。  因此，在不带”-rs”选项的情况下，nachos的线程的生命周期是：   1. 创建并Fork出来，存入readyList 2. 从readyList中取出并得到调度 3. 由于不存在时钟中断，一直执行直到结束 4. 调用Finish，调用Sleep，调用Run切换到其他线程（或调用HALT导致停机）   这完全符合**FCFS**的定义，也就是先到readyList中等待调度，随后直到前面所有线程全部执行结束才轮到它执行，执行完毕后视系统情况选择继续调度下一个还是停机。 1.1.2. Random Round Robin(RRR)调度 为什么叫Random呢，因为和教科书里的不同，时间片长度是随机的，所以是RRR而非RR。  根据1.1.0节中的内容，我们为nachos虚拟机注入了一个生成随机时钟中断的时钟模块。根据上一个实验对时钟以及中断的分析，一个线程的生命周期应该是：   1. 创建并Fork出来，存到readyList中 2. 从readyList中取出并得到调度 3. 在系统时钟推进时，检查时间片是否过期，如果过期，将触发时钟回调函数，进行中断处理 4. 如果未执行结束，回到readyList，并调度下一个线程进行执行 5. 这个过程持续到该线程执行结束为止   关于系统时钟的数据结构、中断回调函数、时间片过期逻辑、系统时钟推进逻辑，上一个实验写得很详细了，这里不再写一遍，不然又要将近10页的论述。  综上，Nachos原有的调度分为：   1. 带”-rs”的RRR调度 2. 不带”-rs”的FCFS调度  2. Non-Preemptive job scheduling with static priority2.0 数据结构 需要改动的地方如下：   1. 线程控制对象中设置优先级字段与getter/setter 2. ready queue的数据结构List需要表现为一个优先级队列，根据线程的优先级（小端）进行push/pop 3. 测试函数中根据README进行改造，Fork三个额外线程，然后和主线程一起并发执行 4. 线程控制对象中设置优先级字段与getter/setter  |  | | --- | | //thread.h  public:  /\*\*  \* setter and getter of priority  \*\*/  int getPriority(){return (priority);}  void setPriority(int newPriority);  private:  // some of the private data for this class is listed above  /\*\*  \* Lab1:threads schedule with static priority  \*/  int priority; |  1. ready queue的数据结构List需要表现为一个优先级队列，根据线程的优先级（小端）进行push/pop。  |  | | --- | | //scheduler.cc  readyList->SortedInsert((void \*)thread,thread->getPriority())  Thread \*  Scheduler::FindNextToRun ()  {  return (Thread \*)readyList->Remove();  }  //list.cc  void \*  List::Remove()  {  return SortedRemove(NULL); // Same as SortedRemove, but ignore the key  }  void \*  List::SortedRemove(int \*keyPtr)  {  ListElement \*element = first;  void \*thing;  if (IsEmpty())  return NULL;  thing = first->item;  if (first == last) { // list had one item, now has none  first = NULL;  last = NULL;  } else {  first = element->next;  }  if (keyPtr != NULL)  \*keyPtr = element->key;  delete element;  return thing;  }  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) {  // item goes on front of list  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;  }  } |   这里nachos官方实现的链表可以按照升序排列，而优先级队列就可以通过线性搜索插入位置实现。又因为优先级以更小的数字为更高，所以恰好符合小端的需求。只能说他们代码写得确实好，规范，精准，没有错误。   1. 测试函数中根据README进行改造，Fork三个额外线程，然后和主线程一起并发执行  |  | | --- | | //threadtest.cc  //----------------------------------------------------------------------  // SimpleThread  // Loop 5 times, yielding the CPU to another ready thread  // each iteration.  //  // "which" is simply a number identifying the thread, for debugging  // purposes.  //----------------------------------------------------------------------  void  SimpleThread(\_int which)  {  int num;    for (num = 0; num < 5; num++) {  printf("\*\*\* thread %d looped %d times, priority=%d\\n",  (int) which,  num,  currentThread->getPriority()  );  currentThread->Yield();  }  }  //----------------------------------------------------------------------  // ThreadTest  // Set up a ping-pong between two threads, by forking a thread  // to call SimpleThread, and then calling SimpleThread ourselves.  //----------------------------------------------------------------------  void  ThreadTest()  {  DEBUG('t', "Entering SimpleTest");  Thread \*t1 = new Thread("t1");  Thread \*t2 = new Thread("t2");  Thread \*t3 = new Thread("t3");  t1->setPriority(1);  t2->setPriority(2);  t3->setPriority(3);  t1->Fork(SimpleThread, 1);  t2->Fork(SimpleThread,2);  t3->Fork(SimpleThread,3);  SimpleThread(0);  } |  运行结果如下：   这里我只截取了一部分，因为截全太长了。可以看到，thread 1在执行50次循环之后才把CPU交给thread2（49是因为从0开始算的）。这是因为thread1比thread2优先级高，它在执行完成前是不会让出CPU的。 2.1 时间片过期判断 在这期间我调试程序时发现一个小问题，关于时间片过期的，之前检查时间片是否过期的代码的部分如下：   |  | | --- | | if(interrupt->nextTimeSlice()->when>stats->systemTicks){  tsExpired = false;  } |   这会导致时间片永远不会过期！想一下为什么：   1. 如果时间片截止时间要严格大于当前系统时间就算不过期，那么假设一个线程执行到还剩一次循环时间片就过期的时刻，这个判断一定为真，因为最后一次循环的时候的系统时间肯定小于时钟中断的时间。 2. 那么时间片没过期，它继续执行，推进系统时钟，发现此时有一个时钟中断（别忘了我们在1中假设还剩一次循环就中断），然后执行回调，分配一个新的时间片。 3. 那么这个线程就得到了一个新的时间片！它将继续执行下去，这与逻辑不符。   解决方法也很简单，我们只要让最后一次循环一定是做线程切换的就可以了，也就是说看似最后一次循环还属于这个线程，其实非也，而是确保让这个线程做切换的。换句话说，假设一个线程分配到10个ticks，那么它只能实际执行其中9个，最后一个是用来做线程切换的（做完线程切换时间片正好过期）  修改代码如下：   |  | | --- | | if(interrupt->nextTimeSlice()->when-SystemTick>stats->systemTicks){  tsExpired = false;  } |   现在需要保证减去一个tick仍然大于当前系统时间，才算时间片没过期，这样就解决了上述问题。 2.2 时间片刷新 在调试的途中发现的这个bug，当一个线程没有用完它的时间片就结束执行时，对于下一个要执行的线程，我们需要为其刷新一个新的时间片。举个例子：  Untitled  比如像这里，thread 1执行完后执行完之后的系统时间是70，而下一个时钟中断的时间是100。也就是说它抢先于时间片过期而结束了执行（或者说执行完了），但下一个线程的时钟中断还是100，这就不符合逻辑了，下一个时钟中断的时间应该是上一个线程结束时间加一个时间片的长度，也就是70+100 = 170。  为了修正这个逻辑，我们需要在线程Finish时更新分配给下一个线程的时间片。让他不要从上一个时间片结束时间开始算，而是从上一个线程结束的时间开始算。   |  | | --- | | //thread.cc :: Finish  interrupt->nextTimeSlice()->when = stats-> systemTicks + timer->TimeOfNextInterrupt() ;  interrupt->getPendingList()->top()->setKey(interrupt->nextTimeSlice()->when); |   首先我们先更新时钟中断实体中的截止时间，也就是interrupt→nextTimeSlice()→when。让他等于当前系统时间加上一个时间片的长度。  然后要注意，pending本质上是一个有序列表，所以我们要更新这个实体对应的key，也为同样的截止时间。也就是interrput→getPendingList()→top()中的key，用一个setter更新为新key。  之后重新编译运行，可以看到：  IMG_256  这回逻辑正确无误了。 3. ./nachos -d t 这里仅仅测试上下文切换次数与打印次数，不测试时间片以及线程调度算法，后面两部分分别在1.2以及1.4节进行了测试。这里只关注本题的重点。  在threadtest.cc中编写测试程序，为了让整段控制台输出总行数稍微少一点以便于调试，我选择减少每个线程循环打印的次数：   |  | | --- | | void  SimpleThread(\_int which)  {  int num;    for (num = 0; num < 2; num++) {  #ifdef PRIORITY  printf("\*\*\* thread %d looped %d times, priority=%d\\n",  (int) which,  num,  currentThread->getPriority()  );  #endif  #ifndef PRIORITY  printf("\*\*\* thread %d looped %d times\\n",  (int) which,  num  );  #endif  currentThread->Yield();  }  } |   基于这段测试程序，编译，随后带上debug选项-d，运行。  结果如下：  Untitled 1  怎么统计上下文切换的次数呢？就是通过这个   |  | | --- | | Switching from thread "%d" to thread "%d" |   的debug信息。这句DEBUG发生在scheduler.cc中Run函数中Switch函数之前，而Switch函数就是上下文函数。  我们可以看到总共有9条Switch信息。这与事实不符，因为：   1. 共计4个线程，每个线程执行两次print然后结束，则2\*4=8次上下文切换 2. 每个线程结束后还要切换到下一个线程，但注意，最后一个线程是直接结束的！不需要切换到下一个线程！因此不同线程之间的切换次数为4-1=3次   故总共应有8+3=11次线程切换。为什么这里只打印了9次呢？  翻了下代码，发现不是逻辑的问题：   |  | | --- | | //thread.cc :: Yield  //...  nextThread = scheduler->FindNextToRun();  if (nextThread != NULL) {  //...  scheduler->Run(nextThread);  }  //... |   可以看到，如果就绪队列中没有其他就绪线程，我们是不会运行Run函数去进行上下文切换的。所以本质上逻辑上没有问题，这里并没有发生上下文切换。  但题目要求我们要缩短上下文切换和打印调试信息条数的差距，那么只能在不进行上下文切换的地方来一个“模拟上下文切换”了，也就是自己切换到自己。（所以说是模拟的）   |  | | --- | | //thread.cc :: Yield  //...  else{  DEBUG('t', "Switching from thread \\"%s\\" to thread \\"%s\\"\\n",  currentThread->getName(),currentThread->getName());  } // else 指nextThread == NULL的情况  //... |   在Yield里的nextThread == NULL的分支中加入DEBUG，重新编译，运行:  打印结果如下：  Untitled 2  可以看到，多了三次Switch，但理应应该多两次啊？  这是因为我们的逻辑是每次循环最后Yield，而最后一次循环时应该Finish，而非先Yield再Finish，不然就浪费了一次上下文切换。  因此修改测试程序，让循环进入到最后一次时直接Finish:   |  | | --- | | if(num!=1)  currentThread->Yield();  else currentThread->Finish(); |   再重新编译运行：  Untitled 3  可以发现上下文切换次数只有7次了，这是为什么？  其实原因很简单，我们修改测试程序的时候，不仅对最后一个执行的线程生效，还对前面所有线程生效，其实现在的上下文切换次数已经不是11次了，就是7次，计算过程如下：   1. 每个线程执行两次print，但注意，第二次执行时，我们现在改成了直接Finish，而不是先Yield再Finish，所以本质上少了一次Yield，也就少了一次上下文切换。所以总计4\*(2-1)=4次。 2. 不同线程之间的切换次数和之前的计算方式一致，4-1=3次   所以总计4+3=7次。正好就是我们的打印次数。 1.4. Aging(老化) 这里我就参照Nachos\_操作系统教程.pdf这个文档实现了，主要是想看看比较权威的解决方案是怎样的。  实际上Aging只是一个动态调整优先级的策略，他并不涉及到调度策略，文档里采用的调度策略是Round Robin+Dynamic Priority。利用时钟中断进行线程优先调度。  对于优先级的调整实际上涉及两个层面：   1. 当一个时间片结束时，提升ready queue中所有线程的优先级 2. 当线程调度时，降低当前线程的优先级（当前线程指的是需要被切换的旧线程）   一开始我一直认为Aging应该只针对于1中的内容。但后来细想了以下，如果我们一直提升那些Ready态的线程的优先级，那么最后可能所有的线程优先级都达到最高，然后就跟Round Robin或者FCFS没啥区别了（具体是哪个取决于ready queue的结构）。所以在参考了指导书后我觉得加上2其实是合理的。  对于1：  设置一个调整的步长，在时间片过期时刷新一遍ready queue中的线程的优先级   |  | | --- | | //thread.h  /\*\*  \* Lab2/Aging  \*/  #define AGING\_PACE -1;  //scheduler.h  #ifdef AGING  private:  /\*\*  \* Lab2/Aging  \* increase the priority of a specific thread  \*/  void increPriority(Thread\* thread);  /\*\*  \* Lab2/Aging  \* maintain the last time of thread switching,to adjust the priority  \*/  int lastSwitchTick;  public:  /\*\*  \* Lab2/Aging  \* flush the priority of all the ready threads  \*/  void FlushPriority();  /\*\*  \* Lab2/Aging  \* getter/setter of lastSwitchTick  \*/  int getLastSwitchTick();  int setLastSwitchTick();  #endif  //scheduler.cc  /\*\*  \* Lab2/Aging  \* increase the priority of a specific thread  \*/  void  increPriority(Thread\* thread){  int changePri;  changePri = thread->getPriority()+AGING\_PACE;  if (changePri<0)  {  changePri=0;  }  thread->setPriority(changePri);  }  /\*\*  \* Lab2/Aging  \* flush the priority of all the ready threads  \*/  void  FlushPriority(){  readyList->MapCar((VoidFunctionPtr)increPriority);  }  这里调用了nachos源码中实现的迭代器MapCar，传入一个句柄，然后处理列表元素。  //list.cc  //----------------------------------------------------------------------  // List::Mapcar  // Apply a function to each item on the list, by walking through  // the list, one element at a time.  //  // Unlike LISP, this mapcar does not return anything!  //  // "func" is the procedure to apply to each element of the list.  //----------------------------------------------------------------------  void  List::Mapcar(VoidFunctionPtr func)  {  for (ListElement \*ptr = first; ptr != NULL; ptr = ptr->next) {  DEBUG('l', "In mapcar, about to invoke %x(%x)\\n", func, ptr->item);  (\*func)((\_int)ptr->item);  }  } |   为了刷新所有线程的优先级，当然要传入对某个指定线程优先级提升的句柄啦。  时间片过期时调用FlushPriority这个函数。注意刷新时机只在时间片过期的时候，而不是每次在时钟推进的时候。因为如果时间片没有过期，即便把ready queue中的线程的优先级提升到非常高，执行的也是当前的线程。所以，在时间片过期时刷新是一个比较省事的选择。  对于2：  参考书中给出的优先级降低的策略需要维护一个字段：lastSwitchTick，也即是上一次线程进行切换的时间。  计算公式如下：   |  | | --- | | Priority = Priority - ( systemTicks - lastSwitchTick) / SystemTick |   其中systemTicks为系统当前时间，SystemTick为时钟推进一次的步长。  如果要维护上一次线程进行切换的时间，那么就要在Run函数中维护了：   |  | | --- | | //scheduler.cc  /\*\*  \* Lab2/Aging  \* update the lastSwitchTick before we switch the thread  \*/  #ifdef AGING  lastSwitchTick = stats->systemTicks;  #endif  SWITCH(oldThread, nextThread); |   这只是维护了我们进行优先级计算的公式中的一个变量而已。别忘了我们维护这个变量用来干什么：计算优先级，在调度时选择优先级最高的线程执行。  而Run函数本身已经挑选了一个合适的线程进行切换，因此重新计算当前线程的优先级的这个任务就不能由他完成。而是需要由他的上游：Yield函数完成。   |  | | --- | | //thread.cc  void  Thread::Yield ()  {  //...  /\*\*  \* Lab2/Aging  \* when a running thread asked to relinquish the CPU,  \* we should adjust its priority to a lower level  \*/  #ifdef AGING  int changePriority = (stats->systemTicks-scheduler->getLastSwitchTick())/SystemTick;  this->setPriority(this->getPriority()+changePriority);  #endif  //...  } |   Yield不仅对当前线程的优先级进行了维护，还改造了挑选下一个执行的线程这一部分的代码。这是因为当前线程的优先级已经被改变了（准确地说是降低了），即便此时没有到时间片过期的时刻，也就是说即便此时其他Ready态的线程的优先级没有得到提升，当前线程也很有可能不再是优先级最高的线程了。所以我们需要把他放回ready队列中，重新挑选一个线程出来。（有可能再次把当前线程又挑出来了）   |  | | --- | | //thread.cc  void  Thread::Yield{  //...  nextThread = scheduler->FindNextToRun();  if (nextThread != NULL) {  DEBUG('t', "Next thread \\"%s\\"\\n", nextThread->getName());  /\*\*  \* Lab2/NONPREEMPTIVE  \* if we want to make sure a thread won't be kicked out of CPU before it's time slice expired  \* we need to compare the system time and the expiring time  \*/  bool tsExpired = checkTsExpired();  #ifdef TIMESLICE  DEBUG('t',"Next time slice is %d,current time is %d\\n",interrupt->nextTimeSlice()->when,stats->systemTicks);  #endif  /\*\*  \* Lab2  \* when we are scheduling the threads with priority(no matter static or dynamic ones)  \* we should compare the one we select from the ready queue with current thread  \* and pick one with higher priority to execute while the other should be put back into ready queue  \*/  DEBUG('t',"Time slice expired? %d\\n",tsExpired);  bool priorityCmp = false;  #ifdef PRIORITY  priorityCmp = this->getPriority()<nextThread->getPriority();  #endif  if(!tsExpired||priorityCmp){  loser = nextThread;  nextThread = this;  }  #ifdef PRIORITY  scheduler->ReadyToRun(loser);  #endif  #ifndef PRIORITY  if(!tsExpired){  scheduler->BackToTop(loser);  }else{  scheduler->ReadyToRun(loser);  }  #endif  scheduler->Run(nextThread);  }  //...  } |   随后我们要分两种情况了：   1. 抢占式的调度 2. 非抢占式的调度   先来看抢占式的调度，这里的抢占式并不是指当一个新线程加入时立刻检查它和当前线程的优先级然后选择一个更高的执行，而是仅当时间片将要过期时检查（距离时间片过期还有1个Tick）。如果我们想修改抢占式调度的规则，让他在系统时钟每推进一个tick时都选择当前优先级最高的线程，那么简单地把这个if去掉，将tsExpired恒置为true即可。  我们先暂且认为当前线程的优先级已经不是所有Ready态线程中最高的了，所以先用一个\*loser指针指向当前线程，然后我们比较重新挑选出的优先级最高的线程的优先级和当前线程的优先级，如果当前线程赢了我们再将\*loser指针指向那个挑选出的新线程也不迟。  随后我们将loser加入ready queue，执行winner即可。  然后是非抢占式的。则会把过程简化非常多，因为非抢占式调度下一个线程拥有无限多的时间片，所以无论如何时间片都不会过期的（tsExpired=false,ts short for time slice)。  这部分的代码里还有RR（Round Robin)调度的处理部分。  这里需要注意一个问题：我们更新优先级实在是太频繁了！实际上每次时钟推移时我们都需要将当前线程推入ready queue中，然后把他取出来，这个过程会调用Run函数，Run函数会更新这个优先级。当优先级频繁更新时，我们就要面临频繁切换线程的结果，指导书认为我们不应该承担这一开销。所以他维护了一个常量叫做MinSwitchPace（最小切换间隔）。当我们尝试去寻找下一个要切换到的线程时，先检查当前系统时间和上一次切换线程的时间之间的间隔是否小于了这个阈值，如果是则返回NULL。我们知道Yield对NULL的处理逻辑：如果找不到下一个应该切换的线程，那么就维持当前线程的运行。   |  | | --- | | //scheduler.cc  Thread \*  Scheduler::FindNextToRun ()  {  #ifdef AGING  /\*\*  \* Lab2/Aging  \* in case of we switching the threads too frequently,  \* we should calculate the duration between the last switch and the current switch  \* if they are too close,just return NULL  \*/  // int duration = stats->systemTicks - this->lastSwitchTick;  // int msp = MinSwitchPace;  // if(duration<msp){  // DEBUG('t',"Thread switch duration is %d ,too frequently,current systemTicks,current totalTicks=%d\\n",duration,stats->systemTicks,stats->totalTicks);  // return NULL;  // }  #endif  return (Thread \*)readyList->Remove();  } |   但可以看到的是，我把这里全部注释了。这是因为这个指导书想法虽然好，但是想得有点简单了。我们不能简单地在频繁切换线程时就抛弃与这个代价相对应的责任。因为可能系统就是需要频繁地切换线程。比方说，现在有两个线程t1,t2。t1执行了1个Tick后结束了任务，他想Finish并把CPU转交给t2。但这是做不到的。  这是因为，线程栈空间维护了一个线程PC在执行完所有任务后结束时的终止任务，叫做ThreadFinish函数。   |  | | --- | | //thread.cc  void  Thread::StackAllocate (VoidFunctionPtr func, \_int arg)  {  //...  machineState[WhenDonePCState] = (\_int) ThreadFinish;  //...  }  static void ThreadFinish() { currentThread->Finish(); }  void  Thread::Finish ()  {  (void) interrupt->SetLevel(IntOff);  ASSERT(this == currentThread);    DEBUG('t', "Finishing thread \\"%s\\"\\n", getName());    threadToBeDestroyed = currentThread;  Sleep(); // invokes SWITCH  // not reached  } |   在我们传递给这个线程的任务句柄执行结束后，线程的PC就跳到了ThreadFinish的首条语句所在的内存地址上。这样接下来就要执行这个ThreadFinish了。  这样，t1调用Finish，Finish会把当前这个线程（t1）标记为待删除的线程，然后进入Sleep。   |  | | --- | | //thread.cc  void  Thread::Sleep ()  {  Thread \*nextThread;    ASSERT(this == currentThread);  ASSERT(interrupt->getLevel() == IntOff);    DEBUG('t', "Sleeping thread \\"%s\\"\\n", getName());  status = BLOCKED;  while ((nextThread = scheduler->FindNextToRun()) == NULL)  interrupt->Idle(); // no one to run, wait for an interrupt    scheduler->Run(nextThread); // returns when we've been signalled  } |   Sleep会把当前这个线程挂起（status=BLOCKED)。然后搜寻下一个将要执行的线程（本来应该是t2)。  现在，因为1个Tick小于了最短切换时间的阈值，那么FindNextToRun将返回NULL。于是我们将进入Idle状态等待一个时钟中断的发生。   |  | | --- | | //interrupt.cc  void  Interrupt::Idle()  {  DEBUG('i', "Machine idling; checking for interrupts.\\n");  status = IdleMode;  if (CheckIfDue(TRUE)) { // check for any pending interrupts  while (CheckIfDue(FALSE)) // check for any other pending  ; // interrupts  yieldOnReturn = FALSE; // since there's nothing in the  // ready queue, the yield is automatic  status = SystemMode;  return; // return in case there's now  // a runnable thread  }  // if there are no pending interrupts, and nothing is on the ready  // queue, it is time to stop. If the console or the network is  // operating, there are \*always\* pending interrupts, so this code  // is not reached. Instead, the halt must be invoked by the user program.  DEBUG('i', "Machine idle. No interrupts to do.\\n");  printf("No threads ready or runnable, and no pending interrupts.\\n");  printf("Assuming the program completed.\\n");  Halt();  } |   Idle中调用地CheckIfDue中先检查是否能把时钟提前到当前即将发生中断的发生时间，这当然是可以的，于是我们把totalTicks提前到了when上，但并没有改动systemTicks。  如果当前机器状态为空闲，且即将发生的中断是中断任务列表中唯一的一个中断（也即时钟中断守护线程），那么代表nachos虚拟机没有事情要做了，那自然没有过期的时间片了，通知调用者false。   |  | | --- | | //interrupt.cc  bool  Interrupt::CheckIfDue(bool advanceClock)  {  MachineStatus old = status;  int when;  ASSERT(level == IntOff); // interrupts need to be disabled,  // to invoke an interrupt handler  if (DebugIsEnabled('i'))  DumpState();  PendingInterrupt \*toOccur =  (PendingInterrupt \*)pending->SortedRemove(&when);  if (toOccur == NULL) // no pending interrupts  return FALSE;  if (advanceClock && when > stats->totalTicks) { // advance the clock  stats->idleTicks += (when - stats->totalTicks);  stats->totalTicks = when;  } else if (when > stats->totalTicks) { // not time yet, put it back  pending->SortedInsert(toOccur, when);  return FALSE;  }  // Check if there is nothing more to do, and if so, quit  if ((status == IdleMode) && (toOccur->type == TimerInt)  && pending->IsEmpty()) {  pending->SortedInsert(toOccur, when);  return FALSE;  }  DEBUG('i', "Invoking interrupt handler for the %s at time %d\\n",  intTypeNames[toOccur->type], toOccur->when);  #ifdef USER\_PROGRAM  if (machine != NULL)  machine->DelayedLoad(0, 0);  #endif  inHandler = TRUE;  status = SystemMode; // whatever we were doing,  // we are now going to be  // running in the kernel  (\*(toOccur->handler))(toOccur->arg); // call the interrupt handler  status = old; // restore the machine status  inHandler = FALSE;  delete toOccur;  return TRUE;  } |   而Idle接到了这一通知之后，就会通知检查中断的代码块无需继续检查了，所以就执行下面的代码，也就是Halt()。这是个很熟悉的指令，停机。  那t2呢？永远无法醒来。因为已经停机了，nachos虚拟机进程已经Exit了。  所以我选择了承担全部的线程切换的代价。但其实这没什么影响，因为现在是非抢占式啊，也不会频繁地在两个不同线程之间切换。所有切换线程的部分其实都只是为了推进系统时钟而已。  最终运行结果如下：    每个线程跑50次循环。可以看到在这个过程中，线程的优先级一直在降低（我们是小端法优先级，数字越小优先级越高）。因为优先级的降低，当失去时间片时它将根据优先级插入到readyList。随后调度的线程就是其他优先级高的线程了。这和静态优先级的结果是截然不同的。 | | | |
| 结论分析与体会：  本次Lab我动手实现了具有优先级的线程调度内核。实验要求的内容比较简单，首先非抢占式这一点就能够保证一个线程拥有时间片时不会被迫放弃CPU，将CPU让给优先级高的线程，而静态优先级和老化（动态优先级）也无非就是修改几个成员函数就能做到。  但这些都和实际的操作系统相去甚远。在实际的操作系统中，有着RR、Preemptive CPU scheduling、multilevel feedback queue这些调度算法，面对不同的任务，操作系统会选择不同的算法来应对。比如对于前台方面交互性强的线程，可能会采用Preemptive + Priority CPU scheduling来进行调度，而对于后台里计算密集型的任务，可能采用RR或FCFS来进行调度，不同的调度算法导致的平均等待时间也是不一样的。  这次的Lab让我初步了解了一些简单的调度算法，以及调度模块与时钟、中断模块的交互，希望以后有机会我可以自己动手去实现更加复杂的算法与系统架构。  在本次实验中，我接触到nachos虚拟机的各种选项，比如-d的debug选项，在编写程序时，我自己会在代码中加上DEBUG调试语句，有助于在思路不够清晰或发生逻辑错误时通过日志信息来调试。通过./nachos -d t,i，我打印出了DEBUG日志，并根据这些信息，调通了时间片过期检查以及时间片刷新这两个功能。具体内容和图片都放在上面的实验内容的叙述中了。  平日里我调试bug，都有IDE中集成的断点调试工具帮助我分析代码，但现在没有这些工具，这些轮子都需要自己造，让我感受到system的底层架构以及实现这个方向的难度之高。同时坚定了只有编写出行为精准、良好的操作系统，才能开发出优秀的上层应用。 | | | |