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：  目录  [1. 源码阅读 4](#_Toc22634)  [1.1. 怎样创建多个线程 4](#_Toc10530)  [1.2. 线程之间是怎样切换的？ 6](#_Toc6450)  [2. 编译条件 11](#_Toc6603)  [3. 原有线程调度策略 13](#_Toc5785)  [3.1. 时钟模块注入 13](#_Toc28663)  [3.2. FCFS调度 14](#_Toc5869)  [3.3. Random Round Robin(RRR)调度 16](#_Toc9935)  [4. Non-Preemptive job scheduling with static priority 17](#_Toc20237)  [4.1. 数据结构 17](#_Toc23572)  [5. ./nachos -d t下的线程切换日志信息 21](#_Toc15733)  6.[Aging](#Aging) 23  6.1. [提升未执行的线程的优先级](#提升未执行的线程的优先级) 23  6.2. [当前执行线程老化](#当前执行线程老化) 26  6.3. [优先级调度的熔断机制](#优先级调度的熔断机制) 27  7.[额外的CPU\_Scheduling机制](#额外的CPU_Scheduling机制) 31  7.1. [时间片机制](#时间片机制) 31  7.1.1. [时间片机制对优先级调度的兼容](#时间片机制对优先级调度的兼容) 31  7.1.2. [Yield函数的不同语义](#Yield函数的不同语义) 32  7.1.3. [三种不同语义下的FindNextToRun](#三种不同语义下的FindNextToRun) 33  7.1.4. [时间片刷新](#时间片刷新) 33  7.1.5. [综合机制下的CPU\_Scheduling](#综合机制下的CPU_Scheduling) 35  7.2. [占先式调度](#占先式调度) 35  7.2.1. [占先式调度的检查时机](#占先式调度的检查时机) 35  7.2.2. [占先式调度测试](#占先式调度测试) 36 **源码阅读****怎样创建多个线程** 既然要多线程并发，那么在虚拟机的某个进程中需要开辟至少两个线程。在这点上，发行的nachos在内核程序的启动入口中提供了一个样例。   |  | | --- | | //main.cc  int  main(int argc, char \*\*argv)  {  int argCount; // the number of arguments  // for a particular command  DEBUG('t', "Entering main");  (void) Initialize(argc, argv); //in system.cc  #ifdef THREADS  ThreadTest(); //in threadtest.cc  #if 0  SynchTest();  #endif  #endif  //...  //system.cc  void  Initialize(int argc, char \*\*argv)  {  //...  // We didn't explicitly allocate the current thread we are running in.  // But if it ever tries to give up the CPU, we better have a Thread  // object to save its state.  currentThread = new Thread("main");  currentThread->setStatus(RUNNING);  //...  //threadtest.cc  void  SimpleThread(\_int which)  {  int num;    for (num = 0; num < 5; num++) {  printf("\*\*\* thread %d looped %d times\n", (int) which, num);  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 \*t = new Thread("forked thread");  t->Fork(SimpleThread, 1);  SimpleThread(0);  } |   在main中，如果运行的是THREADS中的可执行产物，那么将执行ThreadTest()这个函数。这就是一个样例，创建一个新的线程，然后令该线程执行SimpleThread函数，随后在当前线程中也同样执行一遍这个SimpleThread函数。  那么问题来了，上述的“当前线程是谁呢？其实早在ifdef THREADS中的内容执行前，有这么一段   |  | | --- | | (void) Initialize(argc, argv); //in system.cc  //system.cc  void  Initialize(int argc, char \*\*argv)  {  //...  // We didn't explicitly allocate the current thread we are running in.  // But if it ever tries to give up the CPU, we better have a Thread  // object to save its state.  currentThread = new Thread("main");  currentThread->setStatus(RUNNING);  //... |   在main中调用了一个Initialize函数，这个函数创建了一个线程对象，取名为”main”，同时将已经声明的全局变量currentThread指向了该线程对象。这就在ThreadTest中Fork新线程前创建了一个线程，只不过由于它其实就是main线程，不用显式地为其分配栈空间。  同时，我们也没有使用Fork来让他执行任何用户程序，因此该线程只是RUNNING态，但实际上没有使用CPU。  这样一旦我们在当前进程下又开辟了一个新的线程控制对象，并编写代码让这两个线程能够交替执行，就实现了线程并发。 **线程之间是怎样切换的？** 在这个样例中，从代码的角度来看，实际上是某个线程在执行完一些操作后，主动放弃了CPU。   |  | | --- | | void  SimpleThread(\_int which)  {  int num;    for (num = 0; num < 5; num++) {  printf("\*\*\* thread %d looped %d times\n", (int) which, num);  currentThread->Yield();  }  } |   当前正在执行的线程在一次printf后，主动Yield。nachos官方的注释对此的解释是：Relinquish the CPU if any other thread is ready to run.（如果有其他线程处于READY态，主动放弃CPU）  来看一下Yield的实现:   |  | | --- | | //thread.cc  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);  } |   把便于开发的断言和调试去了，其实就是做了这样一件事：   1. 尝试获取下一个ready queue中的线程 2. 如果能够获取到 3. 将当前线程的状态设为READY 4. 运行获取到的下一个线程   而根据nachos官方给出的Round Robin调度策略，实际上FindNextToRun就是简单地获取ready queue的头部元素。   |  | | --- | | //scheduler.cc  Thread \*  Scheduler::FindNextToRun ()  {  return (Thread \*)readyList->Remove(); //这里不多关注链表类对remove的实现了  } |   而ReadyToRun就是将当前线程状态设为READY，然后根据轮转调度的规则，把它放回到ready queue队尾。   |  | | --- | | //scheduler.cc  void  Scheduler::ReadyToRun (Thread \*thread)  {  DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());  thread->setStatus(READY);  readyList->Append((void \*)thread);  } |   Run函数执行时考虑的事情要多一些，首先是保存现场的工作。   |  | | --- | | //scheduler.cc  void  Scheduler::Run (Thread \*nextThread)  {  Thread \*oldThread = currentThread;    #ifdef USER\_PROGRAM // ignore until running user programs  if (currentThread->space != NULL) { // if this thread is a user program,  currentThread->SaveUserState(); // save the user's CPU registers  currentThread->space->SaveState();  }  #endif |   在线程控制对象中，如果该对象控制一个用户线程，那么将开辟一组寄存器存放数据（有别于一般的寄存器）。   |  | | --- | | //thread.h  //defination of Thread  #ifdef USER\_PROGRAM  // A thread running a user program actually has \*two\* sets of CPU registers --  // one for its state while executing user code, one for its state  // while executing kernel code.  int userRegisters[NumTotalRegs]; // user-level CPU register state  public:  void SaveUserState(); // save user-level register state  void RestoreUserState(); // restore user-level register state  AddrSpace \*space; // User code this thread is running.  #endif |   nachos编写了一个Machine类来模拟CPU（包括CPU的寄存器组）。其中包括了一个字段：   |  | | --- | | int registers[NumTotalRegs]; // CPU registers, for executing user programs |   官方对其解释是：用来执行用户程序的CPU寄存器组。  那么在运行新的线程前，我们需要将这组寄存器的内容保存到线程控制对象中保存用户程序数据的寄存器组。  随后检查栈溢出，至于为什么是这个时机我也不太清楚。  然后是最关键的部分，切换线程：SWITCH函数   |  | | --- | | ;scheduler.cc  SWITCH(oldThread, nextThread);  ;线程切换时，保存现场的步骤和参数对于每种不同的处理器是不一样的，所以这个函数是由  ;汇编语言编写的  ;以MIPS架构为例  /\* Symbolic register names \*/  #define z $0 /\* zero register \*/  #define a0 $4 /\* argument registers \*/  #define a1 $5  #define s0 $16 /\* callee saved \*/  #define s1 $17  #define s2 $18  #define s3 $19  #define s4 $20  #define s5 $21  #define s6 $22  #define s7 $23  #define sp $29 /\* stack pointer \*/  #define fp $30 /\* frame pointer \*/  #define ra $31 /\* return address \*/  # a0 -- pointer to old Thread  # a1 -- pointer to new Thread  .globl SWITCH  .ent SWITCH,0  SWITCH:  sw sp, SP(a0) # save new stack pointer  sw s0, S0(a0) # save all the callee-save registers  sw s1, S1(a0)  sw s2, S2(a0)  sw s3, S3(a0)  sw s4, S4(a0)  sw s5, S5(a0)  sw s6, S6(a0)  sw s7, S7(a0)  sw fp, FP(a0) # save frame pointer  sw ra, PC(a0) # save retu rn address    lw sp, SP(a1) # load the new stack pointer  lw s0, S0(a1) # load the callee-save registers  lw s1, S1(a1)  lw s2, S2(a1)  lw s3, S3(a1)  lw s4, S4(a1)  lw s5, S5(a1)  lw s6, S6(a1)  lw s7, S7(a1)  lw fp, FP(a1)  lw ra, PC(a1) # load the return address  j ra  .end SWITCH |   关于sw,lw这些汇编指令的用法，具体参照Assembly Lang 中的一些常用指令  这里要注意一件事，SWITCH函数保存的现场和SaveUserState保存的现场不是一个东西！  在userRegisters这个类字段上有这样的注释说明：   |  | | --- | | // A thread running a user program actually has \*two\* sets of CPU registers --  // one for its state while executing user code, one for its state  // while executing kernel code.  int userRegisters[NumTotalRegs]; // user-level CPU register state |   实际上，无论什么线程都有machineState这个寄存器组，用来执行kernel code。  但只有用户线程有userRegisters这个寄存器组，用来执行user code。  SWITCH函数本质上做的是对kernel code的现场保存。  这一通sw,lw，意思就是把当前kernel寄存器组中的值保存到对应线程控制对象的machineState数组里，然后把新线程的machineState数组中的数据加载到kernel寄存器组里。  随后是一个跳转指令   |  | | --- | | j ra |   ra是return address的缩写，也就是函数返回地址寄存器。j在MIPS指令集里是无条件跳转的指令，也就是说这里简单地把程序指针跳到了ra寄存器中对应的内存地址上。  那么现在ra里面的地址是谁？注意lw ra, PC(a1)这个指令，意思就是把a1寄存器中的值加上PC，得到一个新的内存地址，然后保存到ra寄存器里，所以ra = mem[$a1 + E(PC)]。  那么mem[$a1+E(PC)]到底是什么呢？a1其实是\*newThread，也就是新线程栈空间的开始部分   |  | | --- | | # a0 -- pointer to old Thread  # a1 -- pointer to new Thread |   而PC这个立即数的含义并非是程序指针的内存地址，而是代表了newThread这个线程的PC指针在kernel code对应的寄存器组的偏移量。  这一点可以从switch.h头文件中看出来   |  | | --- | | /\* Registers that must be saved during a context switch.  \* These are the offsets from the beginning of the Thread object,  \* in bytes, used in switch.s  \*/  #define SP 0  #define S0 4  #define S1 8  #define S2 12  #define S3 16  #define S4 20  #define S5 24  #define S6 28  #define S7 32  #define FP 36  #define PC 40 |   注意官方给的注释:in bytes。这其实就意味着0,4,8,…这些数本质上是字节数。比如S1这个寄存器在对应寄存器组中的字节偏移量是8，而一个\_int(分不同机器，可能是条件编译了int，也可能是long)是4个字节，也就是说其数组元素索引为8/4=2。其余同理。  所以$a1是基址，PC是偏移量，加上之后我们就得到了新线程程序指针的真实内存空间了。  那么也就是说ra里的值是新线程PC指针的内存地址。当无条件跳转到该地址后，接下来PC顺序执行，就应该执行新线程了。具体内容需要根据Fork中传递的函数来决定。 2. 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。如下所示   |  | | --- | | DEFINES += -DROUNDROBIN -DTIMESLICE |  如果在-DROUNDROBIN的同时并没有加入-DTIMESLICE，那么就会变成FCFS。以上的几种CPU Scheduling机制可以任意组合，比如，你可以使用：  |  | | --- | | DEFINES += -DPRIORITY -DAGING -DTIMESLICE |   来使用带有老化和时间片机制的优先级调度。  又或者是：   |  | | --- | | DEFINES += -DPRIORITY -DAGING -DPREEMPTIVE |   可以展示带有老化机制的占先式优先级调度。  你可以自由选用并组合这些机制，来构造一种除了实验要求之外的CPU Scheduling。 3. 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**  3.1. 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。  3.2. 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。 3.3. 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  4. Non-Preemptive job scheduling with static priority4.1. 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。   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);  } |  这里我不认为要用Yield来主动切换线程。因为一个线程极少有主动放弃CPU的时刻，大多数都是发生在时间片过期时调用的Yiled函数。因此这里为了推进系统时钟，我直接采用Interrupt类中专门用来推进系统时钟的函数OneTick。后续在我实现了时间片机制后，这个测试程序仍然可以复用，通过手动推动时钟前进的方式来测试时间片机制逻辑是否正确。运行结果如下： IMG_256  观察打印的优先级。9→1→2→3。先执行优先级为9的线程是因为main线程在调度执行时，readyList中还没有thread 1/2/3这三个线程呢。等到thread 1/2/3 Fork出来后，由于我们是非占先调度，main线程会一直持有CPU，直到它执行结束。 5. ./nachos -d t下的线程切换日志信息 如果我们要统计上下文切换次数，应该先知道上下文切换发生在哪里。这个之前提到过，即汇编语言编写的上下文切换函数SWITCH。而这个函数在Scheduler的类成员函数Run中被调用：   |  | | --- | | DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",  oldThread->getName(), nextThread->getName());  //...  SWITCH(oldThread, nextThread); |  在这个函数上面，还有一个DEBUG的日志信息，因此只要打印了这个Switching from …的日志，就代表上下文发生切换。因此我们只要统计该日志信息即可。带上-d t调试选项，运行结果如下：IMG_256 可以看到，4个线程每个循环五次，总共20次。20次之间共有19次切换，因为第20次执行时发现ready list中没有线程了，结束执行（而不是切换线程）。因此也是相符的。  这里需要解释一件事情，可能你会发现，怎么这里线程在交替执行？  这就要分析一个很重要的概念，Yield的目的是什么？Nachos源码的官方作者在注释中这样写：   |  | | --- | | // Thread::Yield  // Relinquish the CPU if any other thread is ready to run. |  目的是让当前线程放弃CPU，调度其他线程执行。也就是说Yield函数如果被一个线程主动调用，那么意味着这个线程暂时不想再执行下去了。即便它是整个队列中优先级最高的线程，也要暂停执行，而是调度次高优先级的线程执行。所以这个结果和优先级调度其实是不相悖的。6. 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。   1. 对于优先级的调整实际上涉及两个层面： 2. 当一个时间片结束时，提升ready queue中所有线程的优先级   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。  **6.1.** **提升未执行的线程的优先级**  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。 **6.2. 当前执行线程老化** 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。  从直观上来说，一个线程的执行的时间越久，它的优先级就应该越低，因此我选择在系统时钟推进的OneTick函数中进行老化：   |  | | --- | | //interrupt.cc  void  Interrupt::OneTick()  {  //...  /\*\*  \* 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  //...  } |  **6.3. 优先级调度的熔断机制** 这里需要注意一个问题：我们更新优先级实在是太频繁了！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，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。抛弃了这个熔断机制。  **7. 额外的CPU Scheduling机制**  上文我提到过，我还额外实现了时间片和占先式这两个机制。  首先我们要明确一点的是，这两个机制在逻辑上是互斥的，有时间片就不能占先；有占先就不能有时间片。因为占先是每个系统Tick都调度优先级最高的线程执行。但时间片的含义是，某个线程在这个时间片覆盖的时间范围内是不能被打断的（不然时间片也没用了）。  **7.1. 时间片机制**  在Lab1中我们分析过如何注入时钟模块，这里就不再赘述了。之后的论述都是基于时钟模块已经完成注入之上的。  我直接复用了Nachos官方提供的时钟模块，因此在时间片实体上，即TimerInt类型的PendingInterrupt，与源码无异。  但是要注意的是，我们要使得时间片机制兼容前面已经实现过的优先级调度以及老化机制。让他们可以一起使用。来想想可能出现的矛盾在哪里吧。  **7.1.1. 时间片机制对优先级调度的兼容**  我先举个例子，就可以很容易地看出这种兼容工作的必要性了。假设现在有两个线程A和B。A的优先级是1，B的优先级是10（小端）。那么优先级更高的A进行执行：   1. A执行直到时间片过期，时钟模块执行回调函数，声明当前线程时间片过期，需要进行切换。 2. A调用Yield函数。Scheduler中的FindNextToRun函数发现readyList中还有B线程，则调度B线程执行。A线程进入ReadyList。   现在问题已经看出来了，A明明是优先级最高的线程，但为什么反而调度了B进行执行呢？这就和优先级调度产生了矛盾。因此我们要做的就是在时间片过期时，调度所有线程中那个优先级最高的执行，而不是随便从ready list队首挑一个进行切换。  可能有人会问，既然每个时间片都会被安排给优先级最高的线程，那为什么还需要时间片呢？直接调度优先级最高的线程，直到该线程执行结束不就可以了？这就涉及到线程老化的问题，当一个线程执行一个时间片的时间后，它会老化，从而可能不再是优先级最高的线程，这时时间片的作用就体现出来了。  **7.1.2. Yield函数的不同语义**  刚刚我只是从理论角度说明了这个兼容工作的必要性。还没有真正开始改造代码。本节开始将真正编码。  首先我们要考虑的事情是，时间片过期后，OneTick会使当前的线程调用Yield：   |  | | --- | | // interrupt.cc :: OneTick  if (yieldOnReturn) { // if the timer device handler asked  // for a context switch, ok to do it now  yieldOnReturn = FALSE;  status = SystemMode; // yield is a kernel routine  currentThread->Yield(Compelled);  status = old;  } |   可是我们刚才提过，Yield函数本质上是线程主动放弃CPU，让其他线程执行。但是在优先级+时间片机制下，时间片过期时如果当前线程仍是优先级最高的线程，那么它不会放弃CPU，而是继续执行。相当于我们又给当前CPU分配了一个时间片（如果它的优先级仍是最高的）。如果当前线程不是目前优先级最高的线程（由于老化），那么就要让当前线程被迫放弃CPU了。  此时Yield函数就产生了不同的语义：被迫放弃CPU。为了解决这一矛盾，我把线程Yield分为三种语义：   1. 线程主动放弃CPU。 2. 线程执行结束，放弃CPU。 3. 线程由于时间片过期，被迫放弃CPU。   其实1和2本质上都是线程主动放弃CPU，不过1可能更偏向于系统调用方面，2是线程知道自己执行结束了所以放弃CPU。之所以细分1和2是因为：   1. 如果线程还没执行结束，它应该回到ready list。 2. 如果线程已经结束，则不需要回到ready list。   因为这方面逻辑的差异，我最终将1和2的行为划分为不同的语义。  综上，我创建了一个枚举类，存放了上述的三种语义：   |  | | --- | | //scheduler.h  enum{Relinquish,Complete,Compelled}; |   分别对应了1、2、3三种语义。  像OneTick检查时间片是否过期，如果过期则Yield，此时的Yield中的参数就应该是：   |  | | --- | | currentThread->Yield(Compelled); |  **7.1.3. 三种不同语义下的FindNextToRun** 我们知道，一个线程Yield后，需要通过FindNextToRun来找到下一个执行的线程。所以既然我们修改了Yield函数，FindNextToRun函数自然而然也要修改。  面对三种不同的语义，FindNextToRun的行为自然也不一样。   1. 当语义是Relinquish和Complete时：  |  | | --- | | // scheduler.cc :: FindNextToRun  // current thread is voluntary to relinquish the cpu to others,  // it's time for next thread to run(if there still has any)  if(status != Compelled){  /\*\*  \* Lab2/TIMESLICE  \* When a thread volunteered to relinquish the cpu  \* We should ensure that the time slice is reallocated  \* otherwise,the next thread would have the rest of the time slice prepared for the dying thread!  \*/  #ifdef TIMESLICE  DEBUG('t',"current sys tick = %d\n",stats->systemTicks);  interrupt->nextTimeSlice()->when = stats-> systemTicks + timer->TimeOfNextInterrupt() ;  interrupt->getPendingList()->top()->setKey(interrupt->nextTimeSlice()->when);  DEBUG('t',"next time slice = %d\n",interrupt->nextTimeSlice()->when);  #endif  if(status== Relinquish){  scheduler->ReadyToRun(currentThread);  }  return nextThread;  } |   ifdef中的代码我会放到下一个小节解释。这里只关注其他代码。正如我之前提到的，如果线程还没执行结束（Relinquish），它应该回到ready list。如果线程已经结束（Complete），则不需要回到ready list。   1. 当语义是Compelled：  |  | | --- | | Thread\* nextThread = (Thread \*)readyList->Remove();  /\*\*  \* Lab2  \* we initially suppose the current thread would be the loser  \* when compete with the one we pick from the ready queue  \*/  Thread\* loser = currentThread;  if(nextThread!=NULL){  /\*\*  \* 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  \*/  bool priorityCmp = false;  #ifdef PRIORITY  //lower number means higher priority  priorityCmp = currentThread->getPriority()<nextThread->getPriority();  #endif  // necessary condition for switching  // the new thread's priority is higher than the currentThread  if(priorityCmp){  loser = nextThread;  nextThread = currentThread;  }  #ifdef PRIORITY  scheduler->ReadyToRun(loser);  #else  scheduler->BackToTop(loser);  #endif  }  return nextThread; |   这里就涉及我在6.1.1节中提到的兼容工作。当时间片过期时，我们要把下一个时间片分配给当前优先级最高的线程。流程如下：   1. 先认为当前线程在优先级上低于ready list中优先级最高的线程  |  | | --- | | Thread\* loser = currentThread; |  1. 然后比较当前线程和ready list中优先级最高的线程的优先级  |  | | --- | | bool priorityCmp = false;  #ifdef PRIORITY  //lower number means higher priority  priorityCmp = currentThread->getPriority()<nextThread->getPriority();  #endif |  1. 如果当前线程优先级较高，则更换loser  |  | | --- | | if(priorityCmp){  loser = nextThread;  nextThread = currentThread;  } |  1. 将loser放回readyList。注意，如果不带优先级调度，我们是把该线程从ready list头部取出来的，因此要原样放回头部。  |  | | --- | | #ifdef PRIORITY  scheduler->ReadyToRun(loser);  #else  scheduler->BackToTop(loser);  #endif |   这样我们在不同场合下使用Yield或FindNextToRun时，就可以根据具体的语义来调用了。  比如，在线程执行结束时的FindNextToRun中，语义就是Complete：   |  | | --- | | while ((nextThread = scheduler->FindNextToRun(Complete)) == NULL) |   **7.1.4. 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 3  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。  同样的事情发生在线程主动Yield，也就是上文中提到的Relinquish语义下。  为了修正这个逻辑，我们需要在线程FindNextToRun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。   |  | | --- | | //scheduler :: FindNextToRun  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：  Untitled 4  这回逻辑正确无误了。  **7.1.5.** **综合机制下的CPU Scheduling**  现在，我把之前实现的优先级调度、老化和现在实现的时间片机制放在一起，看看测试逻辑是否正确。  Untitled 5  可以看到，由于老化，优先级最高的线程t1在一个时间片结束后不再是优先级最高的了，则下一个调度的线程将是拥有当前最高优先级的t2。这和我们期望的逻辑相符。  **7.2.** **占先式调度**  之前我们提到过，占先式调度和时间片机制本质上在逻辑上是矛盾的。但在具体实现上，占先式调度不代表没有时钟模块，恰恰相反的是，占先式调度反而是通过巧妙设计时间片长度实现的。  **7.2.1.** **占先式调度的检查时机**  我们知道，占先式调度总是在每个系统Tick后检查当前线程是否是优先级最高的。如果是，那么当前线程继续执行；如果不是，那么线程切换给当前优先级最高的线程。  因此，我们只要把时间片长度设置成一个系统Tick，就能在每个Tick时检查优先级了。   |  | | --- | | #ifdef PREEMPTIVE  #define TimerTicks 2\*SystemTick // (average) time between timer interrupts  #else |   但是在实现时，我们要注意，Yield函数本身会推动一次系统时钟，所以我们要预留一个Tick给实际的执行，所以总计是两个系统Tick。 **7.2.2.** **占先式调度测试** 好了，占先式调度就这么完成了，就是这么简单！只要改一下时间片的长度即可，现在让我们看一下我们的设计是否正确吧。  Untitled 6  上图的结果是占先式的带有老化机制的优先级调度。可以看到线程执行的次序错乱了很多。你可能会注意到：  Untitled 7  明明线程2的优先级是2，为什么线程1老化到优先级为4的时候还没有切换到线程2呢？  我们来看一下DEBUG日志信息就知道为什么了。  Untitled 8  实际上在t1老化到4时，线程2还没有被创建出来。注意，Fork函数也算一次执行！   |  | | --- | | void  Thread::Fork(VoidFunctionPtr func, \_int arg)  {  //...  IntStatus oldLevel = interrupt->SetLevel(IntOff);  //...  (void) interrupt->SetLevel(oldLevel);  } |   可以看到Fork函数开关了中断，从而推进了时钟。   |  | | --- | | // threadtest.cc  t1->Fork(SimpleThread, 1);  t2->Fork(SimpleThread,2);  t3->Fork(SimpleThread,3); |   而main线程Fork了t1之后，发现t1的优先级高于自己，随后直接把CPU交给t1了，此时t2还没有Fork出来！因此我们的逻辑是对的。 | | | |
| 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。 | | | |