|  |  |
| --- | --- |
| 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：2022.12.7 |
| 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：  Lenovo XiaoXinPro-13IML2020  Intel Core i5-10TH GEN CPU  16GB内存  512GB SSD | |
| 软件环境：  宿主机：Windows 11 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：  一、分析说明Nachos原有的线程调度策略  **1、当nachos运行时，首先进入thread模块下的main函数，调用Initialize函数：**  Int main(int argc, char \*\*argv)  { int argCount;      DEBUG('t', "Entering main");      (void) Initialize(argc, argv);  …}  **2、进入system.cc中的Initialize函数，做系统初始化工作，new一个scheduler作为全局调度者:**  Void Initialize(int argc, char \*\*argv){...   scheduler = new Scheduler();   // initialize the ready queue  …}  **3、scheduler.cc中使用三个函数完成调度工作：**  **ReadyToRun(Thread\* thread);将thread加入就绪队列readyList中，设置该线程的status=ready：**  Void Scheduler::ReadyToRun (Thread \*thread)  {      DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());      thread->setStatus(READY);      readyList->Append((void \*)thread);  }  **FindNextToRun();找到下一个运行的线程，将它移出就绪队列readyList：**  Thread \*Scheduler::FindNextToRun ()  {      return (Thread \*)readyList->Remove();  }  **Void Scheduler::Run (Thread \*nextThread)；**  ①保存当前线程为旧线程；  Thread \*oldThread = currentThread;  ②如果是用户程序，则保存当前CPU寄存器的状态；  #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  ③检查当前线程是否有存在栈溢出；  oldThread->CheckOverflow();  // check if the old thread                          // had an undetected stack overflow  ④将nextThread的状态设置为运行态，并作为currentThread运行线程；   currentThread = nextThread;         // switch to the next thread  currentThread->setStatus(RUNNING); // nextThread is now running  ⑤如果有threadToBeDestroyed线程，则销毁它，并释放它所占用的空间；  if (threadToBeDestroyed != NULL) {          delete threadToBeDestroyed;      threadToBeDestroyed = NULL;      }  ⑥如果是用户程序，恢复当前CPU寄存器的状态。  #ifdef USER\_PROGRAM      if (currentThread->space != NULL) {     // if there is an address space          currentThread->RestoreUserState();     // to restore, do it.      currentThread->space->RestoreState();      }  #endif  从以上可以看出来，现有的Nachos的调度只是简单的从readyList中取出第一个已经就绪的线程，即先来先服务调度(FCFS)  二、设计并实现具有静态优先级的非抢占式线程调度策略。  **1、设置优先级**  **·Thread.h:**  ①为线程添加一个优先级属性int priority：   private:      // some of the private data for this class is listed above  …  int priority;//为线程添加优先级属性  ②定义一个构造方法和一个获取优先级的方法：  public:  …  Thread(const char\* debugName,int priority);//增加构造方法      // basic thread operations  …  int getPriority();//增加获取优先级的方法    **·Thread.cc:**  实现重载的构造函数：  Thread::Thread(const char\* threadName,int priority)  {      name = (char\*)threadName;      stackTop = NULL;      stack = NULL;      this->priority = priority;      if(priority<0){          this->priority = 0;      }      else if(priority>99){          this->priority = 99;      }//控制优先级范围在0~99      status = JUST\_CREATED;  #ifdef USER\_PROGRAM      space = NULL;  #endif  }  Thread::Thread(const char\* threadName)  {      name = (char\*)threadName;      stackTop = NULL;      stack = NULL;      priority = 9;//默认优先级为9      status = JUST\_CREATED;  #ifdef USER\_PROGRAM      space = NULL;  #endif  }  实现获取优先级函数：  //获取优先级  int  Thread::getPriority(){      return this->priority;  }  **2.获得优先级后即可开始控制调度**  **·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);      readyList->SortedInsert((void\*)thread,thread->getPriority())      //使用SortedInsert实现按优先级调度  }  **3.按照要求，在函数ThreadTest中相继new 3个Thread：t1, t2, t3。并把t1, t2, t3 的优先级priority依次设置为 1, 2, 3**  Void ThreadTest()  {      DEBUG('t', "Entering SimpleTest");      Thread \*t1 = new Thread("t1",1);      Thread \*t2 = new Thread("t2",2);      Thread \*t3 = new Thread("t3",3);      t1->Fork(SimpleThread, 1);      t2->Fork(SimpleThread, 2);      t3->Fork(SimpleThread, 3);      SimpleThread(0);  }  **4、进行编译，查看结果**    三、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。  **①以线程调试模式运行Nachos(./nachos -d t)，研究调试输出信息：**  **被测线程**SimpleThread中打印输出的总行数：20  **上下文切换次数：23**  ②**多余或缺少的上下文切换次数是什么原因造成的**  **根据观察输出信息，我觉得是因为这一段：当最后一次loop，thread t1已经完成，但还不能删，因为此时还在旧的栈里，还需进行额外的上下文切换**      **③**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。  //将最后一次循环改为结束  Void SimpleThread(\_int which)  {      int num;      for (num = 0; num < 5; num++) {          if(num<4){               printf("\*\*\* thread %d looped %d times,priority=%d\n", (int) which, num,currentThread->getPriority());               currentThread->Yield();              }else if (num = 4)              {                 printf("\*\*\* thread %d looped %d times,priority=%d\n", (int) which, num,currentThread->getPriority());                 currentThread->Finish();              }      }  }  **修改后结果：**  **被测线程**SimpleThread中打印输出的总行数：20  **上下文切换次数：19**    四、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)。  ①在thread.h里定义一个函数  void setPriority();  ②在thread.cc里实现上述函数：当优先级最高的thread每执行一次，其余的thread优先级增高一级  Void Thread::setPriority(){      this->priority=this->priority-1;  }  ③在scheduler.cc里，简单实现了一下：  修改了一下run（）函数，在当前函数开始running的时候，先将readylist里剩余的thread取出来，将其优先级增高，然后又全部SortedInsert进去  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      oldThread->CheckOverflow();         // check if the old thread                          // had an undetected stack overflow      currentThread = nextThread;         // switch to the next thread      currentThread->setStatus(RUNNING);      // nextThread is now running        //感觉可以在这里添加代码实现aging：每执行一次，将未执行的thread优先级提高，即优先数减1        Thread \*t\_1=(Thread \*)readyList->Remove();        t\_1->setPriority();        Thread \*t\_2=(Thread \*)readyList->Remove();        t\_2->setPriority();        Thread \*t\_3=(Thread \*)readyList->Remove();        t\_3->setPriority();        readyList->SortedInsert((void\*)t\_1,t\_1->getPriority());        readyList->SortedInsert((void\*)t\_2,t\_2->getPriority());        readyList->SortedInsert((void\*)t\_3,t\_3->getPriority());        DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",        oldThread->getName(), nextThread->getName());      SWITCH(oldThread, nextThread)      DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());      if (threadToBeDestroyed != NULL) {          delete threadToBeDestroyed;      threadToBeDestroyed = NULL;  }  运行了一下，发现确实有老化。 | |
| 结论分析与体会：  通过本次实验，我对nachos原有的调度策略有了一定的认识。Nachos现有的调度只是简单的从readyList中取出第一个已经就绪的线程，即先来先服务调度(FCFS)。通过为每个线程增加优先数，调用nachos原本就有的SortedInsert函数，实现具有静态优先级的非抢占式线程调度策略。以线程调试模式运行Nachos(./nachos -d t)，研究调试输出信息，发现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，我通过观察，感觉是因为run（）函数中，当最后一次loop，当前thread已经完成，但还不能删，因为此时还在旧的栈里，还需进行额外的上下文切换。所以我将SimpleThread(\_int which)函数中，最后一次循环就改为结束，经调试，发现有效减少了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。最后关于优先级调度的老化(aging)，当优先级最高的thread每执行一次，其余的thread优先级增高一级修改了一下run（）函数，在当前函数开始running的时候，先将readylist里剩余的thread取出来，将其优先级增高，然后又全部SortedInsert进去，有效实现优先级调度的老化。 | |