|  |
| --- |
| 实验目的：  1. 熟悉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。 |
| 硬件环境：  惠普品牌型号笔记本  Intel Core i5-8300 CPU  8GB内存  512GB SSD |
| 软件环境：  宿主机：Windows 10 21H1 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-LW |
| 实验步骤与内容：  目录：  [Lab2 具有优先级的线程调度](#lab2-具有优先级的线程调度)  [2.1 实验内容](#X3605793e8709e4bf92c87064126c736b72345d9)  [2.2 实验思路](#X534f8515422f814ad861cd5530a5cedff878c05)  [2.3 实验代码](#X1a10d4b9aafd0dcc6a6b1edef9fb47604f3c10c)  [2.3.1 Thread](#Xce56401cc27c5e83b0150608d9aafe61e7c942c)  [2.3.2 List.cc](#X27a54bb38e427728c2c37e506d04882acac2be3)  [2.3.3 scheduler.cc](#X208efc9e0ffe3273fc7bf5b328ab18bdd156c56)  [2.3.4 threadtest.cc](#X5a29ebfae49926a6efc74acaf4eb865bb22c51b) Lab2 具有优先级的线程调度2.1 实验内容  1. 熟悉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。   **提示**：List类中已有的SortedInsert方法可加以利用。 2.2 实验思路 我们需要给所有线程维护一个优先级属性，每次调用scheduler->FindNextToRun()选出队列中优先级最大的线程出队列。因此由两个思路：   * 维护一个优先队列，每次出队列时选择队首元素。 * 按照先来先进队列顺序，但每次出队列时遍历整个队列。   为了利用队列中原有的SortedInsert()方法，同时又因为维护优先队列的时间复杂度由于遍历队列。我们采用方案一。 2.3 实验代码2.3.1 Thread 给每个线程添加一个int priority属性，编写对应的set与get方法，同时重载一个构造函数Thread(char\* name,int threadPriority)：   * Thread.h： * private:  int priority; // 进程的优先级 public:  void SetPriority(int priority){  //优先级-20~20  ASSERT(priority>=-20 && priority<=20);  priority=prior;  }  int getPriority(){return priority;} * Thread.cc：重载的构造函数 * Thread::Thread(const char\* threadName,int threadPriority) {  name = (char\*)threadName;  priority = threadPriority;  stackTop = NULL;  stack = NULL;  status = JUST\_CREATED; #ifdef USER\_PROGRAM  space = NULL; #endif }  2.3.2 List.cc 修改List.cc中SortedInsert()方法中判断权重大小部分的符号，从sortKey < first->key、sortKey < ptr->next->key改为：sortKey > first->key、sortKey > ptr->next->key  // 改为按照从大到小的优先级插入 void List::SortedInsert(void \*item, int sortKey) {  ListElement \*element = new ListElement(item, sortKey);// 创建一个队列元素  ListElement \*ptr; // keep track   if (IsEmpty()) { // if list is empty, put  first = element;  last = element;  } else if (sortKey > first->key) {// 将插入线程放在队列首  // 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) {// 迭代寻找合适的位置  if (sortKey > ptr->next->key) {  element->next = ptr->next;  ptr->next = element;  return;  }  }  last->next = element; // item goes at end of list  last = element;  } } 2.3.3 scheduler.cc 修改scheduler.cc中的ReadyToRun()与FindNextToRun ()方法，使得线程进入与弹出等待队列按照优先级。  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());// 按照优先级插入，维护一个优先队列 }  Thread\* Scheduler::FindNextToRun () {  return (Thread \*)readyList->Remove();// 取出优先队列队首的元素 } 2.3.4 threadtest.cc 修改测试类测试优先级调度  //传入参数为线程编号 void SimpleThread(\_int which){  int num;    for (num = 0; num < 5; num++) {  printf("\*\*\* thread %d looped %d times\n", (int) which, num);  currentThread->Yield();  } }  void ThreadTest(){  DEBUG('t', "Entering SimpleTest");   currentThread->SetPriority(1);// 给当前运行的线程赋优先级  printf("name=%s,priority=%d\n",currentThread->getName(),currentThread->getPriority());   Thread\* t1 = new Thread("Thread1", 2);  printf("name=%s,priority=%d\n",t1->getName(),t1->getPriority());  t1->Fork(SimpleThread, 1);   Thread\* t2 = new Thread("Thread2", 3);  printf("name=%s,priority=%d\n",t2->getName(),t2->getPriority());  t2->Fork(SimpleThread, 2);   Thread\* t3 = new Thread("Thread3", 4);  printf("name=%s,priority=%d\n",t3->getName(),t3->getPriority());  t3->Fork(SimpleThread, 3);   SimpleThread(0); } |
| 结论分析与体会：  赋值修改的文件到lab2文件夹中，修改makefile.local的INCPATH为INCPATH += -I- -I../lab2 -I../threads -I../machine。切换到lab2文件夹下，依次执行命令编译：  make clean make ./nachos  实验二截图  可以看到我们额外创建了三个线程，分别赋予其优先级，nachos操作系统默认执行main线程，SimpleThread(0)方法被执行，main线程调用Yield()方法，main线程重新回到等待队列中，scheduler对象执行FindNextToRun()方法寻找优先级最高的下一个线程（Thread3），scheduler对象调用run(Thread3)方法切换上下文，执行Thread3中的SimpleThread(1)，重复此过程.... |