线程机制实习报告

姓名 **彭广举** 学号 1200012964
日期 2015-3-17

目录

内容一：总体概述

lab1 主要内容是通过阅读 nachos 中线程管理部分的代码，了解 nachos 线程的创建，启动，运行，切换等管理机制，通过和 Linux，Windows 的进程线程管理的对比，更深刻的理解操作系统中的进程和线程的管理机制。

在理解 nachos 线程管理的基础上，对 nachos 的线程类做一些扩充，例如增加线程 ID，用户 ID 信息，并限制系统中最大的线程数。

【用简洁的语言描述本次 lab 的主要内容；阐述本次 lab 中涉及到的重要的概念，技术，原理等，以及其他你认为的最重要的知识点。这一部分主要是看大家对 lab 的总体的理解。

要求：简洁，不需要面面俱到，把重要的知识点阐述清楚即可。】

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6
第一部分	Y	Y	Y	Y		
第二部分	Y/N	Y/N	Y/N			
第三部分	Y/N					
第四部分	Y/N	Y/N	Y/N	Y/N	Y/N	Y/N

具体 Exercise 的完成情况

第一部分

Exercise1

调研 Linux 或 Windows 中进程控制块（PCB）的基本实现方式，理解与 Nachos 的异同。

Linux 的进程控制块（PCB）的具体实现是 task_struct 数据结构。它记录了进程的描述信息，例如进程 ID，进程名，用户 ID 等，还包含进程的控制信息，例如当前的状态，优先级，代码入口地址，程序磁盘地址，进程队列指针等，同时还有所拥有的资源和使用情况，例如打开文件列表，虚拟地址空间的状态等，另外还记录了 CPU 的现场信息，例如寄存器的值，指向赋予该进程的段/页表的指针。

Windows 中的每个 win32 进程都由一个执行体进程块（EPROCESS）表示，其重要内

容有线程块列表，虚拟地址空间描述表和对象句柄列表等，同时还记录有进程 ID，进程状态，进程优先级等基本信息。

Nachos 的线程属性与 Linux 和 Windows 的有共同的部分，例如线程 ID，线程名，堆栈空间等，但没有 Linux 或者 Windows 复杂。在 Nachos 中，线程的控制块是 Thread 类，具体的信息在 Exercise2 中详述。就线程状态来说，Nachos 只有 JUST_CREATE, RUNNING, READY, BLOCKED 四个状态，Windows 的为 7 状态：Ready, Standby, Running, Waiting, Transition, Terminated, Initialized；Linux 的则为：Running, Uninterruptible, Interruptible, Ready, Stopped, Zombie 等。Nachos 创建线程的函数是 Fork，但与 linux 的 fork 函数不太相同，而与 Windows 的 CreateProcess 更相似，因为 Nachos 的 Fork 函数并不包含父子关系。

Exercise2

源代码阅读，理解 Nachos 的线程机制。

Nachos 的线程管理机制中，有两个与机器相关的函数，ThreadRoot 和 SWITCH，前者是所有线程运行的入口，后者负责线程之间的切换。Scheduler 类用于实现线程的调度，目前的策略是 FIFO。Thread 类的对象是线程的控制块相当于进程管理中的 PCB。

Thread 类中包含一些线程的信息，比如线程名，线程的状态，线程的栈指针和一些机器信息等，同时还实现了对线程管理的函数，例如生成一个线程(Fork)，使线程睡眠等待(Sleep)，结束线程(Finish)，设置线程状态(setStatus)，放弃处理机(Yield)。

新建一个线程的方法是：

```
Thread* newThread = new Thread("New Thread");
// 生成一个线程类
newThread->Fork(ThreadFunc,ThreadFuncArg);
// 定义新线程的执行函数及其参数
```

Fork 方法分配一块固定大小的内存作为线程的堆栈,在栈顶放入 ThreadRoot 的地址。当新
线程被调上 CPU 时,要用 SWITCH 函数切换线程图像,SWITCH 函数返回时,会从栈顶取出返回地址,于是将 ThreadRoot 放在栈顶,在 SWITCH 结束后就会立即执行 ThreadRoot 函数。ThreadRoot 是所有线程的入口,它会调用 Fork 的两个参数,运行用户指定的函数;
Yield 方法用于本线程放弃处理机。Sleep 方法可以使当前线程转入阻塞态,并放弃 CPU,直到被另一个线程唤醒,把它放回就绪线程队列。在没有就绪线程时,就把时钟前进到一个中断发生的时刻,让中断发生并处理此中断,这是因为在没有线程占用 CPU 时,只有中断处理程序可能唤醒一个线程,并把它放入就绪线程队列。

Finish 函数用来结束一个线程，Finish 中先关中断，然后设置 threadToDestoryed 为当前进程，然后调用 SLEEP，去调度下一个线程，在 Scheduer.cc 中 run 里 delete 掉 threadToDestoryed 的进程，即当一个线程要 Finish 时，需要在调度另外一个线程时才完成 Finsih 的工作。

Exercise3

增加“用户 ID，线程 ID”两个数据成员，并在 Nachos 现有的线程管理机制中增加对这两个数据成员的维护机制。

用户 ID 和线程 ID 都是 Thread 的属性，所以直接在 Thread 类里进行添加，如下图，在 Thread 类中添加如下代码：

```
int threadID;    // 线程ID
int userID;      // 用户ID
```

```
void SetThreadID(int _id) { threadID = _id; }   // 设置线程ID
int GetThreadID() { return threadID; }          // 获取线程ID

void SetUserID(int _id) { userID = _id; }       // 设置用户ID
int GetUserID() { return userID; }              // 获取用户ID
```

因为还没有涉及到用户进程，所以对于 userID 只做了简单处理。对于 threadID，由于 Exercise4 中对线程的数量进行了限制，要求系统中最多有 128 个线程，所以在 system.h 中定义了如下变量：

```
const int MaxThreadNum = 128;                   // 线程的最大数量

extern int allThreads[MaxThreadNum];          // 对应ID的线程是否存在,大小为128的数组
extern int threadCount;                       // 已经存在多少线程
```

设定 const 变量 MaxThreadNum 来限制线程的最大数量，数组 allThreads 用于分配线程 ID，当 allThreads[i]=0 时，表示 i 可以作为 threadID，否则，若 allThreads[i]=1 时，表示 i 已经是一个 threadID 了，threadCount 用来统计当前系统中有多少个线程，在 Fork 中执行加一操作，所以在 system.cc 中，将其初值设为 1,因为 main 是第一个线程并且没有调用 Fork 函数。对 threadID 进行管理的代码如下：

```
Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;

    for (int i = 0; i < MaxThreadNum; ++i)
    {

        if (allThreads[i] == 0)  // 这里需要对数组加个锁，待以后实现
        {
            threadID = i;
            allThreads[i] = 1;
            break;
        }
    }
#ifdef USER_PROGRAM
    space = NULL;
#endif
}
```

每次 new 一个 Thread 对象时，便利 allThreads 数组，看是否有可用的 id 用于分配。为了使测试结果更加清晰，修改 threadtest.cc 里的 SimpleThread 和 ThreadTest 函数如下：

```
void
SimpleThread(int which)
{
    int num;

    for (num = 0; num < 5; num++) {
    printf("*** thread %d  name:\"%s\" looped %d times\n", which, currentThread->getName(),  num);
        currentThread->Yield();
    }
}
```

```
void
ThreadTest1()
{
    DEBUG('t', "Entering ThreadTest1");

    //Thread *t = new Thread("forked thread");
    Thread *t1 = new Thread("thread1");
    Thread *t2 = new Thread("thread2");
    Thread *t3 = new Thread("thread3");

    //t->Fork(SimpleThread, 1);
    t1->Fork(SimpleThread, t1->GetThreadID());
    t2->Fork(SimpleThread, t2->GetThreadID());
    t3->Fork(SimpleThread, t3->GetThreadID());

    SimpleThread(0);
}
```

运行结果如下：

```
one@ONE-Y400: ~/nachos/nachos-3.4/code/threads
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ ./nachos
*** thread 0  name:"main" looped 0 times
*** thread 1  name:"thread1" looped 0 times
*** thread 2  name:"thread2" looped 0 times
*** thread 3  name:"thread3" looped 0 times
*** thread 0  name:"main" looped 1 times
*** thread 1  name:"thread1" looped 1 times
*** thread 2  name:"thread2" looped 1 times
*** thread 3  name:"thread3" looped 1 times
*** thread 0  name:"main" looped 2 times
*** thread 1  name:"thread1" looped 2 times
*** thread 2  name:"thread2" looped 2 times
*** thread 3  name:"thread3" looped 2 times
*** thread 0  name:"main" looped 3 times
*** thread 1  name:"thread1" looped 3 times
*** thread 2  name:"thread2" looped 3 times
*** thread 3  name:"thread3" looped 3 times
*** thread 0  name:"main" looped 4 times
*** thread 1  name:"thread1" looped 4 times
*** thread 2  name:"thread2" looped 4 times
*** thread 3  name:"thread3" looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 270, idle 0, system 270, 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...
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ gnome-screenshot -w
```

Exercise4

1. 在 Nachos 中增加对线程数量的限制，使得 Nachos 中最多能够同时存在 128 个线程；

在 Exercise3 已经增加了全局变量对线程数量进行了控制，还需要修改 Fork 的代码如下：

```
void
Thread::Fork(VoidFunctionPtr func, int arg)
{
    DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",
      name, (int) func, arg);

    // 由于没有关中断，所以对threadCount的操作也需要加锁
    threadCount++;
    if (threadCount > MaxThreadNum)
        return;

    StackAllocate(func, arg);

    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts
                    // are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

还需要在 Finish 中加入对 threadID 的回收操作，如下：

```
void
Thread::Finish ()
{
    (void) interrupt->SetLevel(IntOff);
    ASSERT(this == currentThread);

    DEBUG('t', "Finishing thread \"%s\"\n", getName());

    // 关了中断，所以后面是原子操作，不需要加锁
    allThreads[currentThread->GetThreadID()]=0;
    threadCount--;

    threadToBeDestroyed = currentThread;

    Sleep();                    // invokes SWITCH
    // not reached
}
```

为了更好的测试对线程数量的限制，将 MaxThreadNum 设为 2，运行结果如下：

```
one@ONE-Y400: ~/nachos/nachos-3.4/code/threads
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ ./nachos
*** thread 0  name:"main" looped 0 times
*** thread 1  name:"thread1" looped 0 times
*** thread 0  name:"main" looped 1 times
*** thread 1  name:"thread1" looped 1 times
*** thread 0  name:"main" looped 2 times
*** thread 1  name:"thread1" looped 2 times
*** thread 0  name:"main" looped 3 times
*** thread 1  name:"thread1" looped 3 times
*** thread 0  name:"main" looped 4 times
*** thread 1  name:"thread1" looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 130, idle 0, system 130, 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...
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ gnome-screenshot -w
```

可以看到 thread2 和 thread3 都没有运行，说明测试成功。

这里有些地方需要对全局变量实现互斥访问，已经在代码中注释说明，但由于 synch.cc 中信号量和锁的实现还不完整，所以还没有写到代码里。

2. 增加 ts 命令显示当前系统的线程信息

一般来说，如果类似于 Linux 的 PS 命令的话，执行 TS 应该是在 shell 中输入命令，但目前还无法实现，所以选取了另一种方法体现 ts 命令。

在 main.cc 中增加如下的代码：

```
#ifdef THREADS
    for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {
      argCount = 1;
      switch (argv[0][1]) {
      case 'q':
        testnum = atoi(argv[1]);
        argCount++;
        break;
      case 't':                          // 判断是否是ts命令
        if (argv[0][2] == 's')
        {
            testnum = 2;
            argCount++;
            argCount++;
        }
        else
        {
            testnum = 1;
            argCount++;
        }
        break;
      default:
        testnum = 1;
        break;
      }
    }
```

当输入./nachos -ts 执行时，会设定 testnum=2。然后修改 threadtest.cc，增加 ShowThreads 函数，来打印线程信息：

```
void
ShowThreads()
{
    printf(" TID  UID  NAME    STATUS   \n");
    for (int i = 0; i < MaxThreadNum; ++i)
    {
        if (threadsInfo[i] != NULL)
        {
            Thread* temp = threadsInfo[i];
            printf("  %d    %d   %s    %s   \n", temp->GetThreadID(), temp->
GetUserID(), temp->getName(), threadStatusStr[temp->getStatus()]);
        }
    }
```

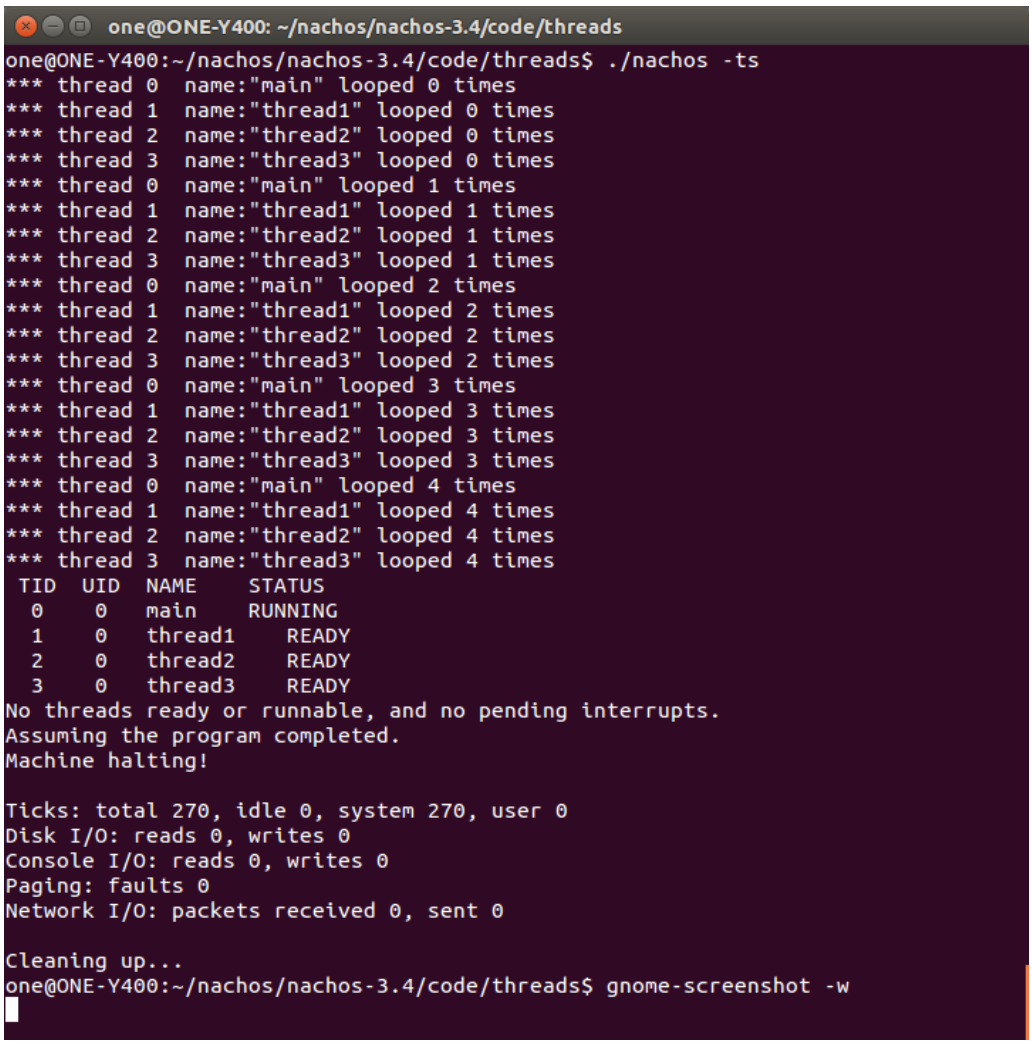
然后在 ThreadTest 中增加下面代码：

```
case 2:
ThreadTest1();
ShowThreads();
break;
```

另外需要在 system.cc/.h 中增加记录线程信息的数组 threadsInfo，类型是 Thread*指向特定线程，并在新建一个线程和结束一个线程时，对该数组进行维护。

这样会调用 ShowThreads 打印线程信息，但是这种做法会使得，每次打印信息都是 main 进程的状态为 RUNNING，因为这个函数是在 main 函数中调用的。

此时，输入指令./nachos -ts 会按照 ShowThreads 中规定的格式打印出线程信息，如下图所示：



第二部分

Exercise1

……

Exercise2

……

Exercise3

……

第三部分

Exercise1

……

第四部分

Exercise1

……

Exercise2

……

Exercise3

……

Exercise4

……

Exercise5

……

Exercise6

……

【对于阅读代码类的 exercise，请对其中你认为重要的部分（比如某文件，或某个类、或某个变量、或某个函数……）做出说明。

对于要编程实现的 exercise，请对你增加或修改的内容作出说明。如果增加或修改了某个类，请写出这个类写在那个文件中，它的的功能是什么，你自己添加或修改的成员变量以及成员函数有哪些，它们的功能是什么；特别的，对于复杂的函数，请说明你的实现方法。不需要贴具体的实现代码。

要求：表述清楚即可， 可采用图表来辅助说明，不需要贴代码】

内容三：遇到的困难以及解决方法

困难 1

这个 lab 的第一部分的难点是对代码的理解，需要对 C++的知识比较熟悉。

困难 2

……

【描述你在实习过程中遇到的困难，是与实习直接相关的技术方面的难题。突出说明你是如何攻克这些难关的。

要求：只需写一下有较深体会的困难。如果觉得整个过程都比较简单的话此部分可不用写。】

内容四：收获及感想

1. 熟悉在 Ubuntu 下的编程环境，对 C++的一些基本知识的又一次学习。
2. 更深刻理解线程的管理机制

【自己的收获，任何关于实习的感想，可以是技术方面的或非技术方面的，可随意发挥。
要求：内容不限，体裁不限，字数不限，多多益善，有感而发。】

内容五：对课程的意见和建议

我个人的一点小不解：这门实习课本来就是一个人完全可以完成的，就像编译实习一样，我不理解为什么还需要小组，如果只是为了讨论的话，那在课堂上可以请助教讲解，大神讲解，然后同学们提一下问题的方式就完全可以，如果为了大家合作的话，软工课的小组就更实际一些 现在的小组好像只是为了调动课堂气氛和让大家花更多的时间而存在的，并且和成绩扯上关系，对一些水平不太好又不太爱发言的小组（例如我们）来说，压力很大。。。纯属个人理解，说得不对的地方还请谅解~

【请写下你认为课程需要改进的地方，任何方面，比如进度安排、难易程度、课堂讲解、考核方式、题目设置.....甚至如果你认为源代码哪里写得不好也欢迎提出。
各位同学反馈的信息对课程建设会有极大帮助。】

内容六：参考文献

1. 操作系统原理课课件：线程进程模型
2. Nachos 中文版文档：
   http://read.pudn.com/downloads101/doc/project/413940/NachosT.pdf
3. 参考资料

【我们希望大家不要使用复制粘贴来拼凑你的报告。详细地列出你在完成 lab 的过程中引用的书籍，网站，讲义，包括你咨询过的大牛们。
要求：诚实，格式尽量按照论文的要求，请参考“论文参考文献格式.doc”】