Lab1 线程机制 实习报告

【实验目标】

本实习希望通过修改 Nachos 系统平台的底层源代码，达到“扩展线程机制”的目标。

【实习内容】

Exercise 1　调研

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。

在 **Linux** 中 PCB 具体实现是结构体 task_struct，组织成双向环形链表。它记录了以下几个类型的信息：

1.标识信息：

unsigned short pid 为用户标识

int pid 为进程标识

2.状态信息，volatile long state 标识进程的状态，可为下列六种状态之一：

可运行状态(TASK-RUNING)

可中断阻塞状态(TASK-UBERRUPTIBLE)

不可中断阻塞状态(TASK-UNINTERRUPTIBLE)

僵死状态(TASK-ZOMBLE)

暂停态(TASK_STOPPED)

交换态(TASK_SWAPPING)

还有进程的寄存器和栈指针信息。

3.用于进程调度的信息：

long prority 表示进程的优先级

unsigned long rt_prority 表示实时进程的优先级，对于普通进程无效

long counter 为进程动态优先级计数器，用于进程轮转调度算法

unsigned long policy 表示进程调度策略，其值为下列三种情况之一：

SCHED_OTHER(值为 0) 对应普通进程优先级轮转法(round robin)

SCHED_FIFO(值为 1) 对应实时进程先来先服务算法；

SCHED_RR(值为 2) 对应实时进程优先级轮转法

3.资源，资源的链接比如内存，还有资源的限制和权限等。例：

int processor 标识用户正在使用的 CPU,以支持对称多处理机方式

5.组织，例如按照家族关系建立起来的树（父进程，子进程等）。

struct task_struct *next_task,*prev_task 为进程 PCB 双向链表的前后项指针

struct task_struct *next_run,*prev_run 为就绪队列双向链表的前后项指针

struct task_struct *p_opptr,*p_pptr,*p_cptr,*p_ysptr,*p_ptr 指明进程家族间的关系，分别为指向祖父进程、父进程、子进程以及新老进程的指针。

在 **Windows** 中 PCB 具体实现是执行体进程块 EPROCESS，构成双向环形链表。它记录了以下几个类型的信息：

1.进程标示符

内部标示：OS 赋予，唯一的数字标示符

外部标示符：创建者提供，字母+数字，包含父进程标示符，子进程标示符，还有用户标识

2.处理机状态

通用寄存器：用户程序可访问，暂存信息，一般 8~32 个通用寄存器

指令计数器：下一条指令的地址

程序状态字 PSW：含有状态信息条件码、执行方式、中断屏蔽标志

用户栈指针：存放过程和系统调用参数及调用地址

3.进程调度信息

进程状态：调度和对换的依据

进程优先级

其他信息：已等待 CPU 时间总和，已执行时间总和

事件：阻塞原因

4.进程控制信息

程序和数据地址

进程同步和通信机制：消息队列指针，信号量

资源清单：所需资源和已分配到的资源

链接指针：进程队列下一个进程 PCB 的首地址

与 nachos 相比，Windows 和 Linux 的 PCB 信息更全面：nachos 没有存储进程调度信息，没有进程资源信息，只有进程的标识和状态信息。

在组织方式上也不相同：nachos 的 PCB（或者说是 TCB）没有组织，只有一个就绪队列；而 Windows 和 Linux 的 PCB 都组织成环形双向链表，便于维护。

Exercise 2　源代码阅读

仔细阅读下列源代码，理解 Nachos 现有的线程机制。

- code/threads/main.cc 和 code/threads/threadtest.cc
- code/threads/thread.h 和 code/threads/thread.cc
-

main.cc 中有如下代码：首先调用 Initialize(argc, argv)初始化所有数据结构。然后如果定义了 thread，则会从主程序参数中读取测试号，并调用 ThreadTest()

threadtest.cc 中有如下代码：首先判断测试号，如果是 1，则调用 ThreadTest1()，在 ThreadTest1()中创建了一个新线程来执行 SimpleThread(int which)，同时原来的线程也调用 SimpleThread(int which)。SimpleThread(int which)的功能是执行 5 次循环，每次循环都执行 Yield()将当前线程挂起，并打印调试信息。

thread.h 中有如下代码：定义了进程块数据结构，定义了进程状态转移的函数和接口函数。

thread.cc 中有如下代码：实现了进程的状态转移函数和接口函数。

Exercise 3　扩展线程的数据结构

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。

```
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of
    // THEY MUST be in this position fo
    int* stackTop;            // the cur
    int machineState[MachineStateSize];

    int uid;
    int tid;
```

```
Thread::~Thread()
{
    DEBUG('t', "Deleting thr
    freeTid[tid] = 1;
    tNum--;
```

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

    uid = 1;
    for (int i = 1; i <= 128; i++)
    {
        if (freeTid[i])
        {
            freeTid[i] = 0;
            Tpool[i] = this;
            tid = i;
            break;
        }
    }
    tNum++;
```

实现机制：

1.在 Thread 类中增加 uid 和 tid 两个私有变量，在 system.cc 中增加 freeTid[]全局变量。

2.在初始化函数 initialize()中把 freeTid[]设为全 1。

3.在 Thread 构造函数中，把 uid 设为默认值 1；再于 freeTid[]中顺序查找一个值为 1 的位置，把该位置置位 0，并把 tid 设为该位置下标。

4.在 Thread 析构函数中，把 freeTid[tid]置为 1。

注：上面的截图也包含了 Exercise 4 的实现细节。

Exercise 4　增加全局线程管理机制

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；

仿照 Linux 中 PS 命令，增加一个功能 TS(Threads Status)，能够显示当前系统中所有线程的信息和状态。

```
Thread* createThread(char* threadName)
{
    Thread* newThread;
    if (tNum <= 128)
    {
        newThread = new Thread(threadName);
    }
    else newThread = NULL;
    return newThread;
}
```

```
void TS() {
    if (tsFlag == 0) return;
    for (int i = 1; i <= 128; ++i) {
        if (freeTid[i] == 0) {
            printf("uid: %d\ttid: %d\
        }
    }
}
```

1.增加全局变量 tNum 统计新建线程数，增加全局变量线程指针数组 Tpool[]。

2.增加包装的构造函数 createThread()，用来判断线程数量是否超过 128，如果超过 128，则返回空指针。增加这个包装函数的好处是不需要处理过多创建的线程的析构。另外还有一种解决办法是在 Fork()中判断线程数量，但是这并不安全，一旦攻击者只构造 Thread 而不 Fork()，那么系统无法阻止这种非法操作，从而造成栈溢出。

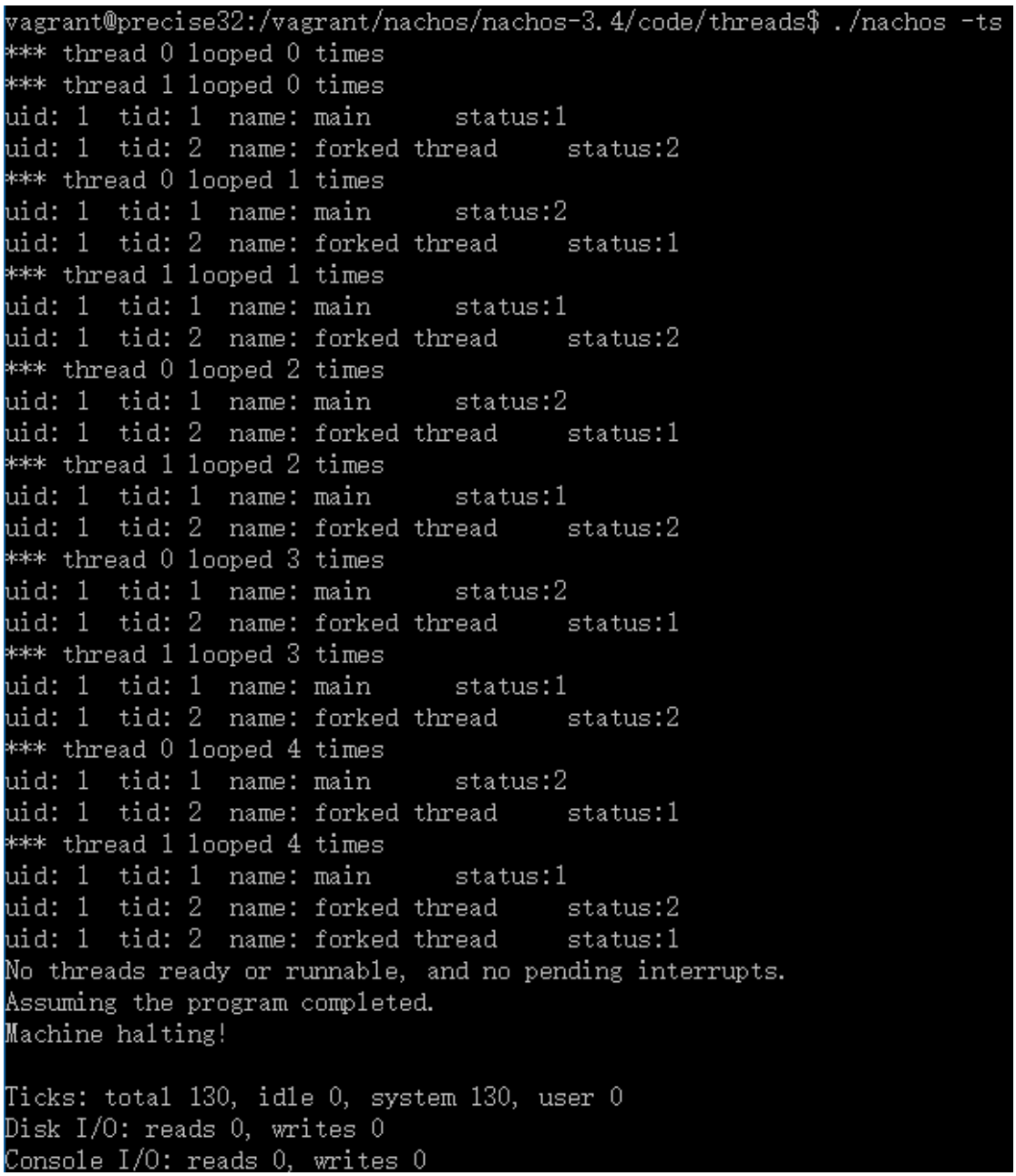
3.在 Thread 构造函数中，把线程赋给 Tpool 中对应 tid 的下标的线程指针。

4.在主程序中的#ifdef THREADS 中加入对”-ts”的支持：设置一个 tsFlag，如果参数有-ts 则置为 1。

5.增加 TS()函数，首先判断 tsFlag 是否为 1，如果不是就直接返回；然后依次打印所有线程的 uid, tid 和状态信息。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 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
```



上面两个截图分别是不带-ts 和带-ts 的执行结果。通过第二张图可以看出，由于各自调用了Yield()函数，两个进程的状态在交替切换。

【实习总结】

通过这次实验，我对进程的理解更加深刻了，同时，我了解了操作系统的编程风格，为后面的实验打好了基础。

杨东升

1400012898