线程机制实习报告

姓名 张煌昭 学号 1400017707
日期 2017.9.22

目录

内容一：总体概述

本次 Lab 1 对线程控制机制进行了研究和探讨。在 Nachos 模拟系统的基础之上，通过对其线程数据结构——线程控制块（TCB，Thread Control Block），进行用户 ID（UID）和线程 ID（TID）的扩充，了解并掌握了 Nachos 的 Thread 类的数据结构；之后在添加了 TID 的基础之上，进一步加入线程数量控制的机制和仿照 linux 系统 ps 命令的 ts 命令，对 Nachos 的线程控制也有了较深的了解。

本次实验总体而言比较简单，但由于需要理解 Nachos 的线程原理，并进行一定程度的修改，需要对 Nachos 中线程相关的源代码进行通读和修改，对于多文件的项目的编写和修改还不熟练，耗费了很多时间在翻看各个文件上。此外，在 win10 系统和 linux 虚拟机环境下，跨环境开发，也比较具有挑战性。

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4
第一部分	Y	Y	Y	Y

具体 Exercise 的完成情况

Exercise1 调研 PCB

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 PCB 基本实现方式

通过阅读 Linux 的 task_struct 结构体代码，发现 Linux 的 PCB 结构主要包括如下部分：

状态（比如进程的挂起，阻塞，运行等等，此外还有进程号，进程组号等等）；

进程结构（比如进程树的父子指针，进程双向链表指针等等）；

内存资源（比如内存堆栈的分配，内存使用权限等等）；

调度（记录该进程的调度信息，比如何时切换上 CPU 等等）。

比较重要的部分为**进程的状态**，分作运行态（TASK_RUNNING），可中断挂起（TASK_INTERRUPTIBLE），不可中断刮起（TASK_UNINTERRUPTIBLE），暂停态（TASK_STOPPED）和僵尸态（TASK_ZOMBIE）。

运行态指正在被 CPU 运行或者准备就绪的状态，包括了用户运行态（进程正常运行），就绪态（进程在等待队列中等待 CPU 时间片）和内核运行态（比如上下文切换，中断处理等）；可中断挂起指处于等待资源的状态中（比如等待信号量），当该资源空闲时，进程就会进入运行态；不可中断挂起指只能用 wake_up()函数唤醒的挂起状态；暂停态由于进程接收到 SIGTSTP，SIGSTOP 等信号，当接收到 SIGCONT 信号时就从暂停态转为运行态；僵尸态指

进程已经运行终止，但父进程还未询问其状态，一般而言僵尸态进程被内核回收。

与 Nachos 的区别

首先，Nachos 中并没有真正的“进程”，而是使用线程来替代进程的概念；Nachos 中没有线程树（进程树）的组织形式，因而不存在线程组（进程组）等概念。

此外 Nachos 将 Linux 中的两种挂起状态合并，进行简化；同时由于没有线程树，Nachos 线程也没有僵尸态，而是采用 scheduler 定时查询并计时的方式来统一回收僵尸线程。

Exercise2 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**

在 code/threads/目录下运行./nachos 之后，进程运行时其执行的函数的路径如下。

下面将详细说明各个源码文件。

threads/main.cc 文件，为整个./nachos 程序的入口。

与本次 Lab 相关的是 THREADS 宏定义下的内容，包括 testnum 参数。在进入 main 函数之后，首先调用 Initialize 函数（于 threads/system.cc 实现）将整个系统初始化，初始化的过程在下面详细说明；之后 main 函数寻找命令行命令中的“-q”参数，并将紧接其后的参数赋值给 testnum（默认值为 1），可以判断出这里通过命令行命令选择测试模式；得到 testnum 之后，调用 ThreadTest 函数（于 threads/threadtest.cc 实现）运行线程测试；最后调用 currentThread->Finish()函数（此时的 currentThread 实际上是在 Initialize 函数中创建的默认的 main 线程，并且 main 线程也是此时没有唯一的被阻塞的线程），释放全部资源并关机。

threads/system.h 文件，声明了 Nachos 系统中所有相关的组件和控制整个系统的方法。

与本次 Lab 相关的主要是 currentThread 和 threadToBeDestroyed 指针和 Initialize 和 Cleanup 函数。两个指针分别指向当前占用 CPU 的线程和等待回收的线程；两个函数分别为

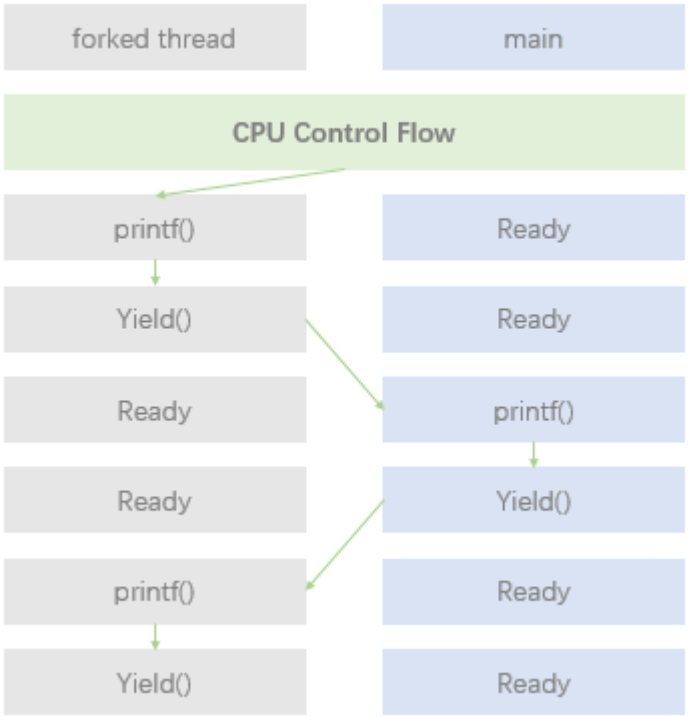
初始化 Nachos 系统和清除 Nachos 系统并关机的函数。此外，还有 interrupt 指针用于指示中断，status 指针用于指明各个线程的操作权限，timer 指针指向硬件时钟。

threads/system.cc 文件，对 threads/system.h 中的变量和函数声明进行了定义。

Initialize 函数初始化整个 Nachos 系统，与本次 Lab 相关的部分主要位于函数的后半部分。在函数中，对 Debug 信息进行初始化，创建并初始化 stats，interrupt 和 scheduler 等。此外最重要的是，Initialize 函数中创建了一个 main 线程，用于在 CPU 空闲时占据 CPU。（不妨将 main 线程理解为 Linux 中的 0 号进程，但由于 Nachos 中没有父子线程的机制，这一理解实际也是不恰当的）至于 Cleanup 函数十分简单，直接删除所有组件，之后 Exit 退出。

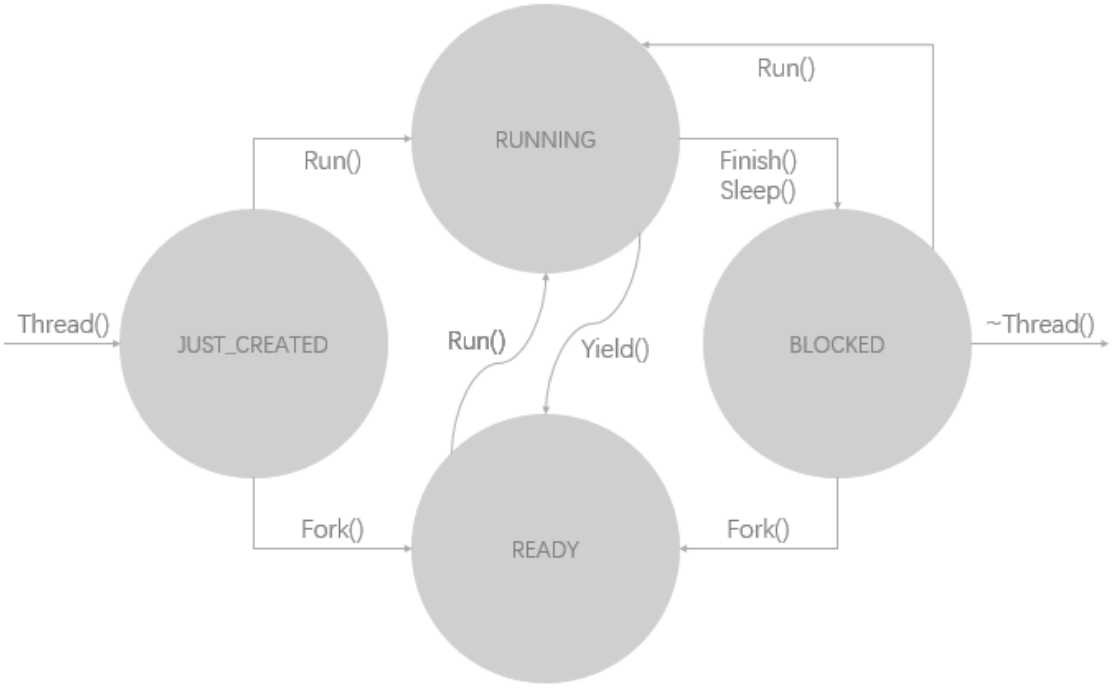
threads/threadtest.cc 文件，定义了测试线程机制的函数等。

所有线程测试函数由 ThreadTest 函数进行统一管理，ThreadTest 函数被调用后，通过 testnum 参数选择使用那一个测试函数（或者测试模式）。Nachos 本身只定义了 ThreadTest1 这一个测试函数，这一函数简单的创建一个新的 forked thread 线程，之后让 forked thread 和 main 两个线程分别执行 SimpleThread 函数，最终两个函数轮换输出，循环五次，其执行的逻辑如下图。



threads/thread.h 文件，声明了 Thread 类（即 TCB）的数据结构和操作方法。

Thread 类中包括栈顶指针 stackTop，寄存器堆 machineState[]，栈底指针 stack，线程状态 status，线程名字 name；Thread 的寄存器堆大小由宏定义 MachineStateSize 定义，默认为 18，栈的大小由宏定义 StackSize 定义，默认为 4*1024 个字。一个线程的状态 status 有四种，分别为 JUST_CREATED（刚被创建），RUNNING（占据 CPU 运行中），READY（位于等待 CPU 空闲的队列中，队列排到后转为 RUNNING），BLOCKED（被阻塞），各个状态在 Nachos 中的转换如下图所示。



threads/thread.cc 文件，定义了 threads/thread.h 中声明的方法，与本次 Lab 相关的有下列函数。

Thread 构造函数，构造一个新的 Thread 线程对象，将其栈指针置为 NULL，命名，并将 status 设定为 JUST_CREATED。

~Thread 析构函数，断言检查该线程是否为占据 CPU 的 currentThread，若是则报错，保证占据 CPU 的线程不会被析构。之后将该 Thread 对象的栈删除。

Fork 函数，类似于 linux 中的 exec 函数而不是 fork 函数，该函数不会将线程分裂为两个父子线程，而是直接令该 Thread 对象执行某个函数。为了保证原子操作，其先将中断阻塞后，再将该 Thread 对象放入 READY 等待队列，之后再开放中断。

Yield 函数，首先断言检查该线程是否为占据 CPU 的 currentThread，若否则报错，保证只有占据 CPU 的线程会被 Yield。如果有线程在 READY 等待队列中，则该线程释放 CPU，进入 READY 等待队列，队列中的第一个线程由 READY 切换为 RUNNING；否则 READY 等待队列为空，该线程继续占据 CPU（本质是先放弃 CPU 进入 READY 等待队列尾，之后再从队列首出队占据 CPU，这一过程可以直接简化为不进行操作）。

Sleep 函数，同 Yield 函数一样先进行断言检查。将该线程 status 置为 BLOCKED 阻塞态，直至被某个信号唤醒，重新进入 READY 等待队列（该信号可能来自其他争夺信号量的线程，也可能该线程不会被唤醒）；而 READY 等待队列中队首线程出队，占据 CPU，如果队列为空，则循环执行 interrupt->Idle()函数，等待中断。

Finish 函数，同 Yield 函数一样先进行断言检查。将该线程标记为待回收，之后调用 Sleep 函数将其阻塞，这种情况就是永远不会被唤醒的情况。

除去上面的与线程 status 相关的函数，还有其它的一些辅助使用的函数。StackAllocate 函数给线程分配栈，CheckOverFlow 函数检查是否发生了栈溢出，Print 函数打印线程信息，getName 和 getStatus 函数返回线程的名字和状态，setStatus 函数设置线程状态。

threads/scheduler.h 和 threads/scheduler.cc 两个文件，对 CPU 上线程的控制和等待队列，以及待回收线程等的控制，进行了定义和说明。threads/thread.cc 中状态控制的一些函数，使用了 threads/scheduler.cc 中的 Idle 函数，Run 函数和 ReadyToRun 函数。由于与本次 Lab 没有很大的关联，不进行赘述。

Exercise3 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。

增加用户 ID（UID）十分简单。对于在 Exercise2 中已经提到过的./nachos 命令的执行过程再进行研究，发现 Nachos 并没有多用户机制。Nachos 的本质就是一个运行在 linux 环境中的进程，该进程模拟了一个操作系统，它用线程来模拟 linux 系统中的进程概念。

因此综合以上想法，在 Nachos 中增加 UID，只需要对代码进行一点微调即可：在 Thread 类中增加私有成员变量 UID，添加公有 getUID 函数以便从外部读 UID；最后在构造函数中，对 UID 赋初值即可。起初我采用的是直接赋 UID 为 0，但后来考虑到 linux 父子进程会继承 UID，因此将线程的 UID 赋值为 Nachos 进程的 UID。

代码修改的展示，在下面的增加 TID 的部分中进行。

增加线程 ID（TID）相对复杂。TID 类似于 UID，需要将数据存放在 TCB 之中，在构造时赋值并且之后不会再更改；但因为 TID 不可重复，同时由于线程会被回收等原因，TID 也需要进行统一的管理和发放。

因此综合上面的想法，将 TID 的内容分作两部分：一部分是线程 TCB 中的 TID，可以通过修改 threads/thread.h 和 threads/thread.cc 完成；另一部分是所有线程 TID 的管理，可以通过在 threads/system.h 和 threads/system.cc 中添加线程 TID 管理的部分完成。

TCB 中的修改为：在 Thread 类中增加私有变量 TID，添加公有 getTID 函数以便从外部读 TID，在构造函数中对 TID 进行赋值。TID 管理需要添加的部分为记录所有线程的数组 allThread，最大线程数宏定义 MaxThread，分配 TID 的 allocTID 函数；所有线程都被用指针的形式存入 allThread 之中，allocTID 会在 allThread 中寻找为空的项并将其下标返回作为分配的 TID；Thread 构造函数会调用 allocTID 得到一个为空的 TID，之后将 allThread 中对应位置指向自己这个线程；额外考虑一下线程 TID 回收，由于线程被回收后，其 TCB 空间被删除，因而 allThread 中指向它的指针也自动变为了 NULL，因此 TID 不需要显式地回收，当线程被回收之后，TID 会被自动释放出来。

测试结果

threads 目录下运行命令./nachos -q 1 进行测试，结果如下。可见两个进程的 UID 和 TID 按预期进行了打印。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 1
*** thread 0 looped 0 times
name: main, UID: 1000, TID: 0
*** thread 1 looped 0 times
name: forked thread, UID: 1000, TID: 1
*** thread 0 looped 1 times
name: main, UID: 1000, TID: 0
*** thread 1 looped 1 times
name: forked thread, UID: 1000, TID: 1
*** thread 0 looped 2 times
name: main, UID: 1000, TID: 0
*** thread 1 looped 2 times
name: forked thread, UID: 1000, TID: 1
*** thread 0 looped 3 times
name: main, UID: 1000, TID: 0
*** thread 1 looped 3 times
name: forked thread, UID: 1000, TID: 1
*** thread 0 looped 4 times
name: main, UID: 1000, TID: 0
*** thread 1 looped 4 times
name: forked thread, UID: 1000, TID: 1
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...
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$
```

Exercise4 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。

增加线程数量限制在上一部分已经完成。当在 allThread 数组中找不到一个为 NULL 的位置时，allocTID 函数将返回-1，标明已经达到线程数量上限，否则返回为 NULL 的位置的下标作为分配的 TID；在构造函数中，断言检查 allocTID 的返回值，若为-1 则报错并打印线程数量超出限值，否则将该返回值赋给 TID 并正常运行。将 threads/system.h 宏定义中的 MaxThread 的值设定为 128 即符合要求。

完成修改后，为了方便测试，在 threads/threadTest.cc 中添加 threadTest2 函数。该函数循环循环 129 次，每次尝试创建一个新的线程，若增加的数量限制无误，则该函数必定超出数量限制，被断言检查出后，Nachos 直接报错退出。

增加 TS 功能，实质上就是增加一个打印 TCB 数据的函数。考虑到 TS 功能属于系统管理所有线程的功能，并且线程数组 allThread 位于 threads/system.h 和 threads/system.cc 之中，因而该 ts 函数添加在这两个文件之中。

ts 函数遍历 allThread 数组，当数组元素非 NULL 时则打印该元素的 UID，TID，名字和 status 状态。为了更好地模仿 linux 中的 ps 命令，在遍历数组之前打印出"UID TID NAME STATUS"这样一行字符串即可。

考虑到让 Nachos 系统有可能像真正的操作系统一样进行命令行输入，在 threads/threadTest.cc 中加入了命令行输入的函数 ThreadTest0：这函数可以循环读入命令，若命令为“ts”，则调用 ts 函数；若命令为“exit”，则运行原代码中最后一行 currentThread->Finish()函数关机；若命令为“nt”，则创建一个新的线程，命名为 forked thread；否则输出找不到该命令，并等待下一次输入。将 threads/main.cc 中的默认的 testnum 改为 0，threads/threadTest 中的 ThreadTest 函数添加 case 0 的情况为 ThreadTest0。

此外，为了方便测试，还在 threads/threadTest.cc 中添加了 threadTest3 函数，该函数创建两个新的线程，之后令这两个执行名为 rawThread 的简单的函数，在这一过程中不断调用 ts 函数，输出各个时间点的线程状态。

测试结果

threads 目录下运行命令./nachos -q 2 对数量限制进行测试，结果如下。可见当线程 TID 达到 127 之后，再次创建新的线程，断言检查发现线程数量超限，报错退出。

```
Attempting to creat thread122
name: thread122, UID: 1000, TID: 123
Attempting to creat thread123
name: thread123, UID: 1000, TID: 124
Attempting to creat thread124
name: thread124, UID: 1000, TID: 125
Attempting to creat thread125
name: thread125, UID: 1000, TID: 126
Attempting to creat thread126
name: thread126, UID: 1000, TID: 127
Attempting to creat thread127
Out of threads. Failed to create a new thread.
System panic!
Assertion failed: line 51, file "../threads/thread.cc"
Aborted
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$
```

threads 目录下运行命令./nachos -q 3 对 ts 功能进行测试，结果如下。各个时间点的 ts 打印结果均正确。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 3
Test checkpoint 1
Thread Status:
UID             TID             Name            Status
1000            0               main            RUNNING

Test checkpoint 2
Thread Status:
UID             TID             Name            Status
1000            0               main            RUNNING
1000            1               forked thread           JUST CREATED
1000            2               forked thread           JUST CREATED

name: forked thread, UID: 1000, TID: 1
Thread Status:
UID             TID             Name            Status
1000            0               main            READY
1000            1               forked thread           RUNNING
1000            2               forked thread           READY

name: forked thread, UID: 1000, TID: 2
Thread Status:
UID             TID             Name            Status
1000            0               main            READY
1000            1               forked thread           READY
1000            2               forked thread           RUNNING

Test checkpoint 3
Thread Status:
UID             TID             Name            Status
1000            0               main            RUNNING
1000            1               forked thread           READY
1000            2               forked thread           READY

No test specified.
```

threads 目录下运行命令./nachos -q 0 对 ts 的“命令行功能”进行测试，结果如下。Ts 命令的结果正确。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 0
$ >> ts
UID             TID             Name            Status
1000            0               main            RUNNING
$ >> nt
$ >> ts
UID             TID             Name            Status
1000            0               main            RUNNING
1000            1               forked thread           JUST CREATED
$ >> nt
$ >> ts
UID             TID             Name            Status
1000            0               main            RUNNING
1000            1               forked thread           JUST CREATED
1000            2               forked thread           JUST CREATED
$ >> exit
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 10, idle 0, system 10, 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...
```

内容三：遇到的困难以及解决方法

暂无。

内容四：收获及感想

本次 Lab 相对简单，主要工作是在阅读 Nachos 源码，阅读 Linux 相关资料，和熟悉使用 Nachos 以及其它工具上；实际编写代码只有不到 100 行。

对我来说，最大的收获是在熟悉使用 GIT 版本控制和 Sublime 文档编辑和使用方面的。

从在刚刚开始修改第一行 Nachos 源码时，小心翼翼，没改一行就想 git commit 提交一次版本，到现在完整实现某个功能之后再进行提交，慢慢的我也收到这种工程开发思想的启迪，对于“版本”有了比之前更深的理解。

此外，我对于我向 Nachos 源码中插入的 python 脚本十分满意。使用该脚本之后，可以简单地用一句符合格式（ python onekeyrun.py [-m or -n] {Nachos args} ）的命令完成 make + run Nachos 等一系列的操作，希望我可以在之后的几次 Lab 之中，继续完善这一脚本，做“跨语言开发”的程序员: P

内容五：对课程的意见和建议

我认为这次的 Lab 相对简单，题目本身都没有很大的挑战性，exercise 2 几乎完全是课堂讨论时讲授过的内容。为了将此次 Lab 做得更有趣一些，我扩展了 exercise 4 的 ts 功能的题目，将在仅仅实现 ts 功能的基础之上，简单地实现了命令行的功能。

希望之后的 Lab 可以“普惠中拔尖”，在相对难度较低的题目之中，留取一两道比较有趣和有挑战性的题目。