实验项目二：Nachos进程管理

2.1实验目的：

完成Nachos的进程管理模块的扩展，掌握操作系统中进程管理和进程调度扩展实现。

2.2实验环境：

描述所用实验环境，包括计算机硬件和软件资源的情况，如选用的操作系统、机器配置、编译器版本等。

Galliumos OS LINUX，gcc编译器。

2.3实验内容：

对实践过程的详细说明，如对Nachos平台的哪些代码进行了什么样的修改，采用了何种算法或思想等。

List核心的代码，如以文件为单位一一进行描述。

可以结合适当的流程图或者类图来辅助描述。

1.为线程这个类定义静态变量`thread_count`来记录（限制）线程总数。

```
void Finish();       // The thread is done executing

void CheckOverflow();     // Check if thread stack has overfl
void setStatus(ThreadStatus st) { status = st; }
char* getName() { return (name); }
void Print() { cout << name; }
void SelfTest();    // test whether thread impl is working
static int thread_count;
int thread_priority;
```

2.修改`thread.cc`中`Thread`类的构造函数，限制线程启用总数。

```
Thread::Thread(char* threadName)
{
    if (++thread_count > 128){
        printf("Error,Number of thread is over 128.\n");
        exit(1);
        ASSERT(thread_count<128);
    }
```

3.修改`thread.cc`中`Thread`类的`SelfTest`函数，创建多个线程。

```
void
Thread::SelfTest()
{
    DEBUG(dbgThread, "Entering Thread::SelfTest");
    // Test threads overflow.
    for (int i=0;i<4;i++){
        Thread *t = new Thread("forked thread");
        //t->thread_id = i;
        printf("Thread: %d has been created.\n \
            The thread priority is %d.\n",i,t->thread_priority);
        t->Fork((VoidFunctionPtr) SimpleThread, (void *) i);
    }
}
```

4.修改`thread.h`以及`thread.cc``Thread`类，为每个线程加入随机优先级，同时，为了避免间隔过短时，rand()相等，创建一个线程完后，阻塞1秒。

```
    void Finish();      // The thread is done executing

    void CheckOverflow();     // Check if thread stack has overfl
    void setStatus(ThreadStatus st) { status = st; }
    char* getName() { return (name); }
    void Print() { cout << name; }
    void SelfTest();    // test whether thread impl is working
    static int thread_count;
    int thread_priority;
```

```
Thread::Thread(char* threadName)
{
    if (++thread_count > 128){
        printf("Error,Number of thread is over 128.\n");
        exit(1);
        ASSERT(thread_count<128);
    }
    srand(time(NULL));
    this->thread_priority = rand() % 10 + 1; // Get rand priority;
    sleep(1);
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
    machineState[i] = NULL;     // not strictly necessary, since
                        // new thread ignores contents
                        // of machine registers
    }
    space = NULL;
}
```

5.修改`scheduler.h`,`scheduler.cc`，将调度器调度策略由FIFO变为优先级调度。
因为lib库中list类中有实现好的sortedlist类，所以我们将scheduler.h中readyList由List变为SortedList。同时，重写调度器构造函数，加入compare函数，使SortList以每个线程的优先级由高到低排序。

```
  private:
    SortedList<Thread *> *readyList;  // queue of threads that are ready to run,
    //SortedList<Thread *> privilege_thread_list;
        // but not running
    Thread *toBeDestroyed;  // finishing thread to be destroyed
            // by the next thread that runs
};
```

```
    // SelfTest for scheduler is implemented in class Thread
    static int Scheduler::compare(Thread* x,Thread* y){
      int res =  x->thread_priority - y->thread_priority;
      return res<0?1:-1;
      //return x->thread_priority > y->thread_priority;
    }
```

```
Scheduler::Scheduler()
{
   //readyList = new List<Thread *>;
   readyList = new SortedList<Thread*>(&compare);
   toBeDestroyed = NULL;
}
```

6.修改`thread.cc`中`Yield`函数，将原来的先从等待队列寻找下一个运行的线程然后在将当前进程放入等待队列逻辑变为先将当前进程放入等待队列，然后再从等待队列寻找下一个运行的线程。

```
void
Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Yielding thread: " << name);

    kernel->scheduler->ReadyToRun(this);
    nextThread = kernel->scheduler->FindNextToRun();

    if (nextThread != NULL) {
    // kernel->scheduler->ReadyToRun(this);
    kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

7.修改`thread.cc`中`SelfTest`函数，完成线程调度测试。

```
void
Thread::SelfTest()
{
    DEBUG(dbgThread, "Entering Thread::SelfTest");
    // Test threads overflow.
    for (int i=0;i<4;i++){
        Thread *t = new Thread("forked thread");
        //t->thread_id = i;
        printf("Thread: %d has been created.\n \
            The thread priority is %d.\n",i,t->thread_priority);
        t->Fork((VoidFunctionPtr) SimpleThread, (void *) i);
    }
}
```

2.4实验结果：

截图说明运行效果，并且辅助相关描述信息。

测试结果：

```
            The thread priority is 6.
Thread: 125 has been created.
            The thread priority is 6.
Thread: 126 has been created.
            The thread priority is 6.
Thread: 127 has been created.
            The thread priority is 6.
Error,Number of thread is over 128.
ssorryqaq@sorry /u/l/n/c/build.linux>
```

```
tests summary: ok:0
Thread: 0 has been created.
            The thread priority is 1.
Thread: 1 has been created.
            The thread priority is 7.
Thread: 2 has been created.
            The thread priority is 9.
Thread: 3 has been created.
            The thread priority is 8.
*** thread 2 looped 0 times
*** thread 2 looped 1 times
*** thread 2 looped 2 times
*** thread 2 looped 3 times
*** thread 2 looped 4 times
*** thread 3 looped 0 times
*** thread 3 looped 1 times
*** thread 3 looped 2 times
*** thread 3 looped 3 times
*** thread 3 looped 4 times
*** thread 1 looped 0 times
*** thread 1 looped 1 times
*** thread 1 looped 2 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
```

```
*** thread 1 looped 0 times
*** thread 1 looped 1 times
*** thread 1 looped 2 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
Machine halting!

Ticks: total 2440, idle 0, system 2440, 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
ssorryqaq@sorry /u/l/n/c/build.linux>
```

创建4个线程，按照优先级次序执行，且保证了时间分片(Yeild策略)。

2.5实验总结：

总结本实验的完成情况，包括代码是否编写完成，是否调试通过，能否正常运行，实现了实验要求中要求的哪些项（对实验要求的满足程度；）；

总结本实验所涉及到的知识点。

代码编写完成，调试通过，正常运行。完成实验要求：

1. 限制线程数量（128）
2. 修改扩充Nachos的线程调度机制，改为“优先级调度”的抢占式调度。

知识点：

1. 调度器调度线程
2. 时间分片。