MultiThreading 实验报告

专　业：软　件　工　程

指　导　老　师：王　冬　青

年　级：大　学　二　年　级

学　号：2153061

姓　名：谢　嘉　麒

1. Uthread: switching between threads (moderate)

1.1.实验目的

本次实验要求为用户层面的线程系统设置一个切换机制，并且将其实现。实验提供了基本的Xv6代码，包括user/uthread.c和user/uthread_Switch.S以及Makefile。uthread.c包括大多数用户级线程包以及三种简单测试线程的代码。然而这些线程包缺少部分创建线程和线程切换的代码。

实验者需要想出办法来创建线程并切换线程，并且实现这个计划。

1.2.实验步骤

1.2.1.分析已有代码

实验中给出了部分代码，在实验开始前先对他们进行分析。

查看user/uthread.c中的线程数据结构，可以看出struct thread中用stack[STACK_SIZE]作为线程的栈，用state表示线程的状态。根据实验要求，需要添加一个数据结构来保存每个进程的上下文。

1.2.2.完成上下文切换

在user/uthread.c文件中新增添数据结构strcut context，根据内核中关于进程的上下文来确定其中内容。查阅Xv6实验手册可以知道，除sp，s0，和ra寄存器，只需要保存callee-saved寄存器。包括uint64 ra，uint64 sp...uint64 s10，uint64 s11。

在user/uthread_switch.S文件中添加sd和ld内容，具体代码如下：

```
sd ra, 0(a0)
sd sp, 8(a0)
sd s0, 16(a0)
sd s1, 24(a0)
sd s2, 32(a0)
sd s3, 40(a0)
sd s4, 48(a0)
sd s5, 56(a0)
sd s6, 64(a0)
sd s7, 72(a0)
sd s8, 80(a0)
sd s9, 88(a0)
sd s10, 96(a0)
sd s11, 104(a0)

ld ra, 0(a1)
ld sp, 8(a1)
ld s0, 16(a1)
ld s1, 24(a1)
ld s2, 32(a1)
ld s3, 40(a1)
ld s4, 48(a1)
ld s5, 56(a1)
ld s6, 64(a1)
ld s7, 72(a1)
ld s8, 80(a1)
ld s9, 88(a1)
ld s10, 96(a1)
ld s11, 104(a1)
ret
```

通过以上代码来完成上下文切换的功能

1.2.3.创建和调度线程

实验已经给出thread_create()函数作为线程的创建函数，也在其中给出了基础的代码。我们需要设置线程的栈，并且保证线程在调度运行时pc进行合适的跳转。

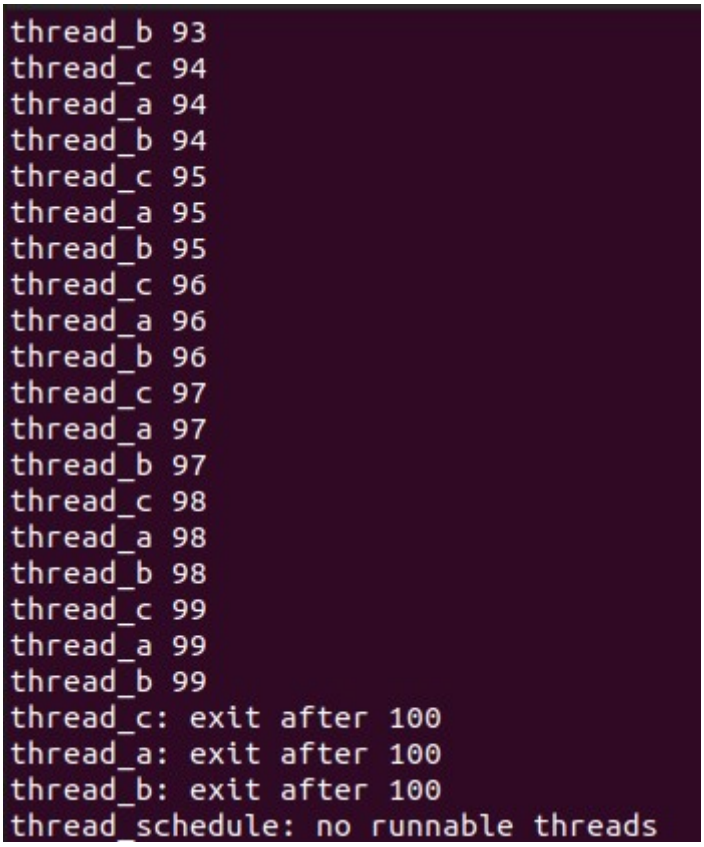
```
t->context.ra = (uint64)func;
t->context.sp = (uint64)&t->stack[STACK_SIZE];
```

在thread_schedule()函数中添加调度线程的语句，选中下一个可运行的线程并且切换上下文即可：

```
thread_switch((uint64)&t->context, (uint64)&current_thread->context);
```

1.2.4.编译与检测

完成以上步骤后，进入Xv6目录终端并且执行make qemu指令来进行编译，编译成功后执行uthread，可以看到预期输出如下：



1.3.实验中遇到的问题和解决方法

1.3.1.增加保存线程上下文的数据结构

根据实验要求需要添加保存进程上下文的数据结构，因此需要对进程上下文的具体内涵和内核对于进程上下文的代码实现有一定的学习和理解。

通过对kernel中关于进程上下文的代码的参考，可以设置uint64类型的变量来进行存储，设置寄存器ra,sp,s0~s11来存放。

1.4.实验心得

通过本次实验，笔者对于用户态线程库有了进一步的了解，同时对于线程的代码实现，包括数据结构，上下文保存，寄存器读取等进行了学习和实现。

2. Using threads (moderate)

2.1.实验目的

本实验要求使用线程和锁，使用哈希表来探索并行程序。需要在真实的Linux或MaxOS系统上（需要多核）运行而不是Xv6和qemu。

本实验需要修改notxv6/ph.c文件来保证多线程读写一个哈希表并产生正确结果。

2.2.实验步骤

2.2.1.观察输出结果

进入终端运行make ph来编译运行notxv6/ph.c，运行./ph 1可以看到输出如下：

```
100000 puts, 3.991 seconds, 25056 puts/second
0: 0 keys missing
100000 gets, 3.981 seconds, 25118 gets/second
```

读写哈希表并且产生了正确结果。

运行./ph 2，看到输出结果如下：

```
100000 puts, 1.885 seconds, 53044 puts/second
1: 16579 keys missing
0: 16579 keys missing
200000 gets, 4.322 seconds, 46274 gets/second
```

发现存在遗漏，需要进行完善。

2.2.2.完善实验

判断./ph 2出现的遗漏是因为多线程同时读写的情况下，部分数据发生了竞争访问，没有被正确写入哈希表，因此可以尝试使用锁来处理共享数据结构。

（1）首先为哈希表定义一个保护的互斥锁：pthread_mutex_t lock;

（2）定义锁后需要进行初始化：pthread_mutex_init(&lock, NULL);

（3）在写入哈希表前获得锁，写入完成后释放锁。

2.2.3.编译与检测

完成以上步骤后再次make ph编译notxv6/ph.c，运行./ph 2，得到了预期结果：

```
cinderlord@ubuntu:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
cinderlord@ubuntu:~/xv6-labs-2021$ ./ph 2
100000 puts, 2.321 seconds, 43087 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 4.444 seconds, 45004 gets/second
```

2.3.实验中遇到的问题和解决方法

2.3.1.理解数据的竞争访问

对实验前进行的测试内容要进行合理分析。根据实验的提示，笔者联想到数据遗漏的的问题可能是写入或读出时发生了竞争而未能成功读写。因此笔者对数据的竞争访问进行了调研和学习，并且了解到给共享数据加锁能够解决这一问题。通过查阅线程加锁的资料实现了这一实验。

2.4.实验心得

本次实验中，笔者对数据的竞争访问进行了调研和学习，同时也在判断数据遗漏的问题中有了进一步的理解和认识。通过实际的加锁操作，结合收集到的thread加锁资料，笔者在多线程读写的代码实现方面也更加熟练。

3. Barrier (moderate)

3.1.实验目的

本次实验要求实现一个线程屏障：程序中的一个关节点，所有线程必须在此处等待直到别的线程也到达这个关节点。当所有线程都到达屏障后，才能都继续运行。

实验提供了一个不完善的屏障，通过运行make barrier以及./barrier 2可以得到如下语句：

```
cinderlord@ubuntu:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
cinderlord@ubuntu:~/xv6-labs-2021$ ./barrier 2
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
Aborted (core dumped)
```

3.2.实验步骤

3.2.1.屏障数据结构的分析

观察barrier.c可以看到屏障的数据结构，包括一个互斥锁，一个条件变量，一个记录到达线程屏障的线程数的变量，一个记录屏障轮次的变量。因此可以判断在线程到达屏障后，需要先获取锁，修改nthread。当到达线程数足够后，清空nthread并且轮次增加，唤醒线程；结束后释放锁。

3.2.2.barrier()实现

根据3.2.1.的分析，barrier()的实现可以分为以下几个部分：

（1）获取屏障的互斥锁；

（2）nthread++；

（3）判断nthread此时的数量是否符合预期；

（4）当足够后，round++，nthread = 0，唤醒线程；

（5）不足的话让线程继续等待；

（6）完成以上步骤后释放互斥锁。

3.2.3.编译与检测

完成以上步骤后执行make barrier进行编译，运行./barrier 2检测，获得预期结果：

```
cinderlord@ubuntu:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
cinderlord@ubuntu:~/xv6-labs-2021$ ./barrier 2
OK; passed
```

3.3.实验中遇到的问题和解决方法

3.3.1.通过线程屏障来实现同步

本次实验线程屏障的实现并不困难，但是需要具体理解这一功能设置的用途和意义。因此笔者在搜集资料，整理并学习后了解到各线程协作时，可能需要代码在某个位置线程同步，因此需要使用条件变量实现barrier来进行线程同步。

3.4.实验心得

本实验作为MultiThreading实验的收尾内容，与其上两个实验一起体现了线程的三个概念，即互斥，上下文切换和同步。通过本实验，笔者对于线程这一概念有了具体而清晰的理解。