Lab5 Multithreading 实验要求

一、实验简介

本次实验目的是熟悉多线程。实验将在用户级线程包中实现线程切换机制，并通过多线程来加速程序，并实现 Barrier。

二、实验准备

[1] 阅读 xv6 book 第 7 章 Scheduling，并阅读相关源代码
[2] 准备 xv6 Lab5 源码：

```
$ git fetch
$ git checkout thread
$ make clean
```

三、实验要求

3.1 Uthread: switching between threads

在本实验中，你将为用户级线程系统设计上下文切换机制，然后实现它。xv6 下有两个文件 $user/uthread.c$ 和 $user/uthread\_switch.S$，以及 $Makefile$ 中的一个规则用于构建一个$uthread$程序。$uthread.c$包含大部分用户级线程包，以及三个测试线程的代码。但线程包中缺少一些用于**创建线程**和**线程间切换**的代码。

TODO：

你的任务是制定一个计划来**创建线程**并**保存/恢复寄存器以在线程之间切换**，并实施该计划。完成后，在 xv6 上运行$uthread$时应该会看到以下输出（三个线程可能以不同的顺序启动）：

```
$ make qemu
...
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
...
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

此输出来自三个测试线程，每个测试线程有都有一个循环，循环的任务是打印一行信息，然后将 CPU 交给其他线程。但是，此时，如果没有上下文切换代码，你将看不到任何输出。

你需要在 $user/uthread.c$ 中的 **thread_create()**和 **thread_schedule()**以及 $user/uthread_switch.S$中的 **thread_switch** 中添加代码。一个目标是**确保当 thread_schedule()运行给定线程时，该线程在自己的堆栈上执行传递给 thread_create()的函数。**另一个目标是**确保 thread_switch 保存被切换到的线程的寄存器，恢复被切换到的线程的寄存器，并返回到后一个线程的指令中上次停止的点**。你必须决定**在哪里保存/恢复寄存器；例如修改 struct thread 以保存寄存器。**你需要在 **thread_schedule 中添加对 thread_switch 的调用**；你可以将所需的任何参数传递给 thread_switch，来从线程 t 切换到 next_thread。

Hints：

- thread_switch 只需要保存/恢复 callee-save registers。思考下为什么
- 可以在 $user/uthread.asm$ 中查看 uthread 的汇编代码来方便调试
- 可以参考文档中的调试步骤测试自己的代码

完成后，**make grade** 来通过 **uthread** 测试。

3.2 Using threads

本实验将使用 hash table 探索线程和锁的并行编程。注意需要在真正的 Linux 或 MacOS 多核计算机（**不是 xv6，不是 qemu**）上完成这个实验。

本实验使用 Unix pthread 线程库。可以通过 man pthreads 找到相关手册。也可以查看文档上的网站。

文件 $notxv6/ph.c$ 含有一个$hash\ table$，在单线程下正常运行，但多线程中不可以。在 xv6 主目录下，键入：

```
$ make ph
$ ./ph 1
```

注意，要构建 ph, $Makefile$使用的是操作系统的 gcc，而不是 6.S081 工具。ph 程序运行的参数是对 hash table 执行 put 和 get 操作的线程的数量。ph 1 运行的结果与下列类似：

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

本地运行结果与样例结果有可能因为计算机运行速度、core 数量等原因相差两倍甚至更多。

ph 运行两个基准。第一：通过调用 put()来添加大量的 keys 到 hash table，并打印出每秒 put 的次数。第二：通过 get()从 hash table 中取出 keys，打印因 put()出现在 hash table 中但丢失的 keys 的数量（本例为 0），并打印出每秒能达到的 get()数量。

用多线程操作 hash table 可以尝试 ph 2：

```
$ ./ph 2
100000 puts, 1.885 seconds, 53044 puts/second
1: 16579 keys missing
0: 16579 keys missing
200000 gets, 4.322 seconds, 46274 gets/second
```

ph 2 表示两个线程并发向 hash table 中添加表项，理论上速率可以达到 ph 1 的两倍，获得良好的并行加速（parallel speedup）。

但是，两行 16579 keys missing 表明很多 keys 在 hash table 中不存在，put() 应该将这些 keys 加入了 hash table，但是有些地方出错了。需要关注 $notxv6/ph.c$的 put()和 insert()。

TODO：

1. 为什么两个线程会丢失 keys，但是一个线程不会？确定一种两个线程的执行序列，可以使得 key 丢失。

2. 为了避免这种情况，需要在 $notxv6/ph.c$ 中的 put()和 get()中添加 lock 和 unlock 语句，使得两个线程丢失的 keys 数量为 0。相关的线程函数如下：

```
pthread_mutex_t lock;                  // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock);             // acquire lock
pthread_mutex_unlock(&lock);           // release lock
```

完成后用 **make grade** 来通过 **ph_safe** 测试。

3. 内存中没有交集的并发读写操作不需要锁相互制约，利用这个特性提高并发加速（Hint：每个 hash bucket 一个锁）。修改你的代码，使一些 put 操作在保持正确性的同时并行运行。

完成后用 **make grade** 来通过 **ph_fast** 测试。ph_fast 测试要求两个线程每秒产生的 put 次数至少是一个线程的 1.25 倍。

3.3 Barrier

本实验中将实现一个 barrier：当一个线程到这个点后，必须等待其余所有线程都到达这点。使用 pthread 条件变量，类似于 xv6 的 $sleep/wakeup$ 的序列协调技术。注意本实验同样应在真正的计算机上完成（不是 xv6，不是 qemu）

文件 $notxv6/barrier.c$ 含有一个不完整的 barrier。

```
$ make barrier
$ ./barrier 2
barrier: notxv6/barrier.c:42: thread: Assertion `i == t' failed.
```

2 表示在 barrier 上同步线程的数量（是 barrier.c 中的 nthread）。每个线程运行一个循环。循环的每次迭代调用 barrier()，然后睡眠一段随即时间。当一个线程在另一个线程到达 barrier 之前就越过了 barrier，则 assert 触发。理想的情况是每个线程都阻塞在 barrier()，直到 nthreads 个线程都调用 barrier()。

TODO：

本实验应完成理想的 barrier 行为。除了上个实验的锁原语，还需要新的 pthread 原语（可以参考文档的链接）。

```
pthread_cond_wait(&cond, &mutex);   // go to sleep on cond, releasing
lock mutex, acquiring upon wake up
pthread_cond_broadcast(&cond);        // wake up every thread sleeping on
cond
```

Hints：

- 调用 pthread_cond_wait 时释放 mutex，返回之前重新获得 mutex。

- 实验已经给出了 barrier_init()，需要我们实现 barrier()来防止 panic，struct barrier 已经被定义好。

有两个问题需要注意下：

- 必须处理一连串的 barrier 调用，每次调用为一次 round。bstate.round 记录当前的 round。你需要在每次当所有的线程到达 barrier 之后，将 bstate.round 加一。
- 需要注意一种情况：在其他线程退出 barrier 之前，一个线程进入了循环（特别是，从一个 round 到另一个 round，正在重新使用 bstate.nthread）。确保之前的 round 正在使用时，一个线程离开 barrier，再进入循环时不会增加 bstate.nthread。
- Test your code with one, two, and more than two threads.

完成后用 **make grade** 来通过 **barrier** 测试。

3.4 Optional challenges for uthread

感兴趣的同学可以自行阅读并实现文档上的相关挑战。

四、实验提交

参考 PPT 中提交部分的详细说明