同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 6

2151767 薛树建

日期：2023 年 7 月 20 日

目录

Lab6: Multithreading

1. Uthread: switching between threads (moderate)

1) 实验目的

本实验目的是让我们实现用户级线程的切换，实验已经提供了 user/uthread.c 和 user/uthread_switch.S 两个文件，根据提示我们可以知道实现用户线程切换主要缺少的部分是创建线程与在线程之间切换的代码。

在开始本实验之前，我们需要先了解一下 xv6 上下文切换的机制。线程切换应该遵循的是时间片轮转策略，目前来看在内核时钟中断部分实现线程切换。

当发生时钟中断时，调用 yield，在该函数中将进程状态设置为 runable(等待)，接着调用 sche 函数，调用 swtch，保存此时进程上下文，并将调度器的上下文载入 cpu，接着 cpu 会运行调度器(scheduler 函数)，寻找一个等待的进程，在此调用 swtch，保存调度器的上下文，并载入新进程的上下文。由于每次切换时进程都为于 sche 函数中的 swtch 代码下，因而新的进程也会在该位置运行，此时完成了线程的切换。

2) 实验步骤

了解了上下文切换的机制之后，我们可以很块的完成该实验。我们本次实验的代码基本上不涉及内核，仅仅是在用户代码中去仿照内核实现而言。

a) 首先是如何去创建一个线程，我们需要先定义线程的上下文 context 结构体，并在线程控制块中加入 context，如下：

```
struct context {//上下文
  uint64 ra;
  uint64 sp;

  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

```
struct thread {
  char        stack[STACK_SIZE]; /* the t
  int         state;             /* FREE,
  struct context contexts;
};
```

b) 接着我们需要修改一下其 thread_create 函数，主要是对创建线程时的一些变量的初始化。

```
  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->context.ra = (uint64)func;                  // 设定函数返回
  t->context.sp = (uint64)t->stack + STACK_SIZE;  // 设定栈指针
}
```

大体逻辑就是从 all_thread 中找到一个为 FREE 的线程控制块，对它的变量进行修改，并将其设置为 runable(等待)。

c) 我们创建线程结束后，main 函数调用 thread_schedule，该函数逻辑与内核态的还不太一样，在该函数中直接寻找下一个等待的线程，并直接将两个线程的 context 进行交换了。我们需要加上一行代码去调用 thread_switch。

```
    current_thread = next_thread;
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    thread_switch((uint64)&t->context, (uint64)&current_thread->context);//
  } else
    next_thread = 0;
```

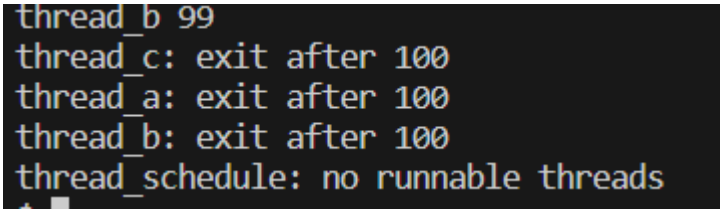
d) 最后我们在 uthread_switch.S 加入相应的汇编代码，可以仿照内核态中的实现（其实代码一样）。

```
/* YOUR CODE HERE */
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)
```

基本上就是先 sd 保存，在 ld 载入相应寄存器。

e) 测试如下：



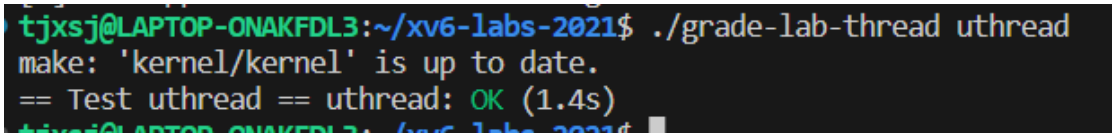
3) 实验中遇到的问题和解决方法

本实验主要问题便是需要先了解内核态中是如何进行线程的切换，这些可以通过阅读实验指导书来解决。

4) 实验心得

通过完成本实验，我对于内核态中线程的切换有了一定的了解，该切换实际上时同时时钟中断来进行的（时间片轮转），发生时钟中断时便进入内核并将 cpu 放弃，转换到 scheduler，scheduler 寻找一个新的等待的线程，去恢复其上下文，然后实现线程的切换。

评分如下：



2. Using threads (moderate)

1) 实验目的

在这个实验中，已经提前帮助我们写好了一个向哈希桶中插入若干键并获取插入的键，二者结果比较来判断插入是否正确。

目前的程序在单线程下工作时正确的，但是在多线程下会缺失一部分键。要求我们调用 pthread 库中的部分接口，来保证多线程执行的正确性。大概是关于多线程编程的，比较容易

2) 实验步骤

首先为什么会缺少键?

假设现在有两个线程 T1 和 T2，二者同时在执行插入操作，假设它们两个插入的桶也是相同的，那么在 T1 完成插入时，T2 也进行插入，此时就会出现互相覆盖的情况进而造成数据丢失。解决方法便是借助 pthread 库有关的锁的函数，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
```

a) 整体操作还是较为简单的，首先就是为每一个桶配一把锁并初始化。

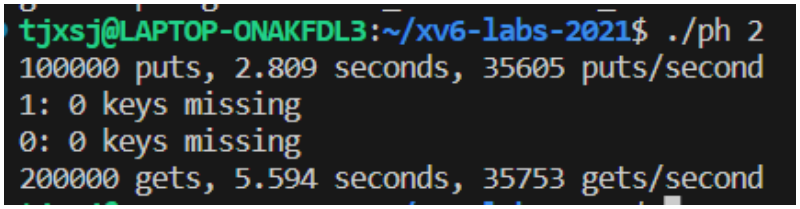
```
int nthread = 1;
pthread_mutex_t lock[NBUCKET]; // 每个散列桶一把锁
```

```
double t1, t0;
//对每一把锁进行初始化
for (int i = 0; i < NBUCKET; ++ i)
    pthread_mutex_init(&lock[i], NULL);
```

b) 接着在 put 函数插入时上锁解锁就行了。

```
} else {
  // the new is new.
  pthread_mutex_lock(&lock[i]);
  insert(key, value, &table[i], table[i]);
  pthread_mutex_unlock(&lock[i]);
}
```

c) 测试如下：



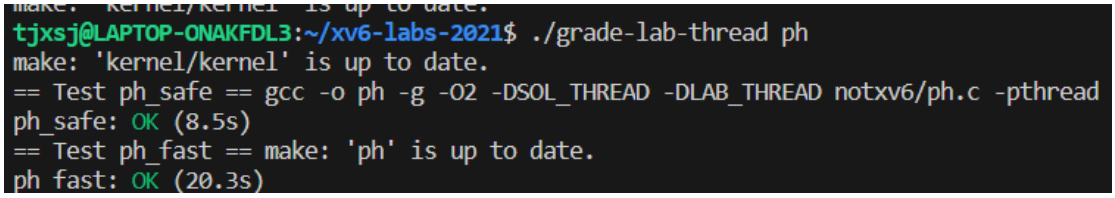
3) 实验中遇到的问题和解决方法

本实验主要就是多线程编程注意对临界区的访问，可以通过锁来实现。

4) 实验心得

通过完成本实验，我对于操作系统的中的临界区的互斥访问有了一定理解，同时也提高了我多线程编程的代码能力

评分如下：



3. Barrier (moderate)

1) 实验目的

在这个实验中，给出了一个不完成的 barrier.c，要求我们去完善它，实际上就是让我们去实现一个屏障，在程序中指定一个点，所有的线程执行到此处时都要被阻塞，只有全部的线程都到达之后，才会继续运行。

这是一些可能用到的接口：

```
pthread_cond_wait(&cond, &mutex);
pthread_cond_broadcast(&cond);
```

2) 实验步骤

a) pthread_cond_wait 接收两个参数，第一个参数为 pthread_cond_t 为条件变量，第二个参数为 pthread_mutex_t，调用函数时会释放传入的锁，并使线程进入等待；pthread_cond_wait 接受一个参数为 pthread_cond_t，让那些因为该条件变量而等待线程被唤醒，继续执行。

b) 只需要在每一个线程执行到 barrier 时，先获取锁，接着记录此时等待的线程数量，若线程数量与总数一致，则直接唤醒所有线程尝试获取锁并继续执行，如果不是则调用 pthread_cond_wait 进入等待并释放锁。最后注意去释放锁。

```
// 申请持有锁
pthread_mutex_lock(&bstate.barrier_mutex);

bstate.nthread++;
if(bstate.nthread == nthread) {
  // 所有线程已到达
  bstate.round++;
  bstate.nthread = 0;
  pthread_cond_broadcast(&bstate.barrier_cond);//
} else {
  // 等待其他线程,并释放锁
  pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
}
// 释放锁
pthread_mutex_unlock(&bstate.barrier_mutex);
```

c) 测试如下：

```
ph_fast: OK (20.3s)
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthrea
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./barrier 2
OK; passed
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$
```

3) 实验中遇到的问题和解决方法

本实验主要对提示中的两个函数不太熟悉，通过在网上查找得知其作用。
pthread_cond_wait(&cond, &mutex)让当前进程等待并释放锁；
pthread_cond_broadcast(&cond)则是唤醒等待的进程并让它们重新获取锁。

4) 实验心得

通过完成本实验，提高了我的多线程编码的能力，对 c 的多线程库有了一定的了解
评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-thread barrier
make: 'kernel/kernel' is up to date.
== Test barrier == make: 'barrier' is up to date.
barrier: OK (3.0s)
```

总结

通过完成本次实验，我对于内核部分线程的切换流程有了进一步的了解，同时后面两个多线程编程实验也是进一步提高了我的编码能力。

本实验总体评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (2.8s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (8.2s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (20.1s)
== Test barrier == gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_TH
barrier: OK (3.2s)
== Test time ==
time: OK
Score: 60/60
```