Thread

Design

uthread

> Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan. When you're done, `make grade` should say that your solution passes the `uthread` test.

This lab is to sibulate multi-thread operation in xv6 user space, the key point is how to save and switch register between cpu switching threads. I copy code from procss swiching code from xvt kernel

```
thread_switch:
        //save old reg
        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)
        //load new regs
        ld ra, 0(a1)
        ld sp, 8(a1)barrier()

        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)

```

```
34          ret
35
```

ph

> To avoid this sequence of events, insert lock and unlock statements in `put` and `get` in `notxv6/ph.c` so that the number of keys missing is always 0 with two threads. The relevant pthread calls are:

```
1   barrier()
2   {
3   // add nthread
4   pthread_mutex_lock(&bstate.barrier_mutex);
5   bstate.nthread++;
6   if (bstate.nthread>=nthread){
7    //add round and wake all threads up
8    bstate.round++;
9    bstate.nthread = 0;
10   pthread_mutex_unlock(&bstate.barrier_mutex);
11   pthread_cond_broadcast(&bstate.barrier_cond);
12  } else{
13   //sleep and wait for the condition call,
14   // in wait(), it will unlock before sleep and lock before wake up
15   pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
16   pthread_mutex_unlock(&bstate.barrier_mutex);
17  }pthread_mutex_t lock;            // declare a lock
18  pthread_mutex_init(&lock, NULL); // initialize the lock
19  pthread_mutex_lock(&lock);       // acquire lock
20  pthread_mutex_unlock(&lock);     // release lock
```

> You're done when `make grade` says that your code passes the `ph_safe` test, which requires zero missing keys with two threads. It's OK at this point to fail the `ph_fast` test.

This lab is to use mutex lock to avoid concurrency races in real linux operation system to achieve multi-threads safety and faster speed.

This is an easy lab about using lock, just add lock in data which can be access by all threads.

Barrier

> Your goal is to achieve the desired barrier behavior. In addition to the lock primitives that you have seen in the `ph` assignment, you will need the following new pthread primitives; look here and here for details.

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

Make sure your solution passes `make grade`'s `barrier` test.

what is barrier?

a point in an application at which all participating threads must wait until all other participating threads reach that point too.

this lab requires me to use condition variable and mutex lock to sync all threads to make sure all threads reach a certain point.

the algrothm is pretty simple:

> if a thread reaches the point, but there are some other threads still working, sleep this thread
>
> if all threads reach the point, wake all threads up

```
1   barrier()
2   {
3     // add nthread
4     pthread_mutex_lock(&bstate.barrier_mutex);
5     bstate.nthread++;
6     if (bstate.nthread>=nthread){
7       //add round and wake all threads up
8       bstate.round++;
9       bstate.nthread = 0;
10      pthread_mutex_unlock(&bstate.barrier_mutex);
11      pthread_cond_broadcast(&bstate.barrier_cond);
12    } else{
13      //sleep and wait for the condition call,
14      // in wait(), it will unlock before sleep and lock before wake up
15      pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
16      pthread_mutex_unlock(&bstate.barrier_mutex);
17    }
```

Result

uthread

1. because the uthread's output depending on the CPU arrangement, I can not test directly in `make grade'`

```
thread_c 60
thread_c 61
thread_c 62
thread_c 63
thread_c 64
thread_c 65
thread_c 66
thread_c 67
thread_c 68
thread_c 69
thread_c 70
thread_c 71
thread_c 72
thread_c 73
thread_c 74
thread_c 75
thread_c 76
thread_c 77
thread_c 78
thread_c 79
thread_c 80
thread_c 81
thread_c 82
thread_c 83
thread_c 84
thread_c 85
thread_c 86
thread_c 87
thread_c 88
thread_c 89
thread_c 90
thread_c 91
thread_c 92
thread_c 93
thread_c 94
thread_c 95
thread_c 96
thread_c 97
thread_c 98
thread_c 99
thread_c: exit after 100
thread_schedule: no runnable threads
```

ph and barrier

```
== Test answers-thread.txt == answers-thread.txt: FAIL
    answers-thread.txt does not seem to contain enough text
== Test ph_safe == make[1]: Entering directory '/home/vielo/code/xv6lab'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/vielo/code/xv6lab'
ph_safe: OK (9.3s)
== Test ph_fast == make[1]: Entering directory '/home/vielo/code/xv6lab'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/vielo/code/xv6lab'
ph_fast: OK (20.4s)
== Test barrier == make[1]: Entering directory '/home/vielo/code/xv6lab'
make[1]: 'barrier' is up to date.
make[1]: Leaving directory '/home/vielo/code/xv6lab'
barrier: OK (2.8s)
== Test time ==
time: OK
```