

# Lab 7: Multithreading

---

1950787 杨鑫

## 实验目的：

---

- 熟悉线程的概念，对比进程加深理解
- 实现线程的创建和切换功能
- 体会同步和互斥的必要性，并通过使用锁来解决该类问题

## 实验步骤：

---

首先切换分支至 thread branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout thread
make clean
```

### a. Uthread: switching between threads

#### 实验目的：

在本练习中，您将为用户级线程系统设计上下文切换机制，然后实现它。为了让你开始，你的 xv6 有两个文件 user/uthread.c 和 user/uthread\_switch.S，以及 Makefile 中的一个规则来构建一个 uthread 程序。uthread.c 包含大部分用户级线程包，以及三个简单测试线程的代码。threading 包缺少一些用于创建线程和在线程之间切换的代码。

您的工作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。

#### 实验分析：

本实验要求实现线程的创建和切换机制，创建时要记录函数的 pc 位置以确保切换后能找到起始位置并执行，并且要修改栈指针位于栈顶处；切换时要保存旧线程的上下文，同时要恢复新线程的上下文。

#### 实验核心代码：

uthread.c

```
struct thread {
    char    stack[STACK_SIZE]; /* the thread's stack */
    int     state;              /* FREE, RUNNING, RUNNABLE */
}
```

```

    struct context thread_context; /* the context for the thread */
};

void
thread_schedule(void)
{
    struct thread *t, *next_thread;

    /* Find another runnable thread. */
    next_thread = 0;
    t = current_thread + 1;
    for(int i = 0; i < MAX_THREAD; i++){
        if(t >= all_thread + MAX_THREAD)
            t = all_thread;
        if(t->state == RUNNABLE) {
            next_thread = t;
            break;
        }
        t = t + 1;
    }

    if (next_thread == 0) {
        printf("thread_schedule: no runnable threads\n");
        exit(-1);
    }

    if (current_thread != next_thread) {          /* switch threads? */
        next_thread->state = RUNNING;
        t = current_thread;
        current_thread = next_thread;
        /* YOUR CODE HERE
        * Invoke thread_switch to switch from t to next_thread:
        * thread_switch(??, ??);
        */
        thread_switch(&t->thread_context, &current_thread->thread_context);
    } else
        next_thread = 0;
}

void
thread_create(void (*func)())
{
    struct thread *t;

    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == FREE) break;
    }
    t->state = RUNNABLE;
    // YOUR CODE HERE
    t->thread_context.ra = (uint64)func;
    t->thread_context.sp = (uint64)(t->stack) + STACK_SIZE;
}

```

uthread\_switch.S

```

.text

/*
 * save the old thread's registers,
 * restore the new thread's registers.
 */

.globl thread_switch
thread_switch:
/* 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)
ret    /* return to ra */

```

测试:

```
xv6 kernel is booting
```

```
hart 1 starting  
hart 2 starting  
init: starting sh  
$ 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 2  
thread_a 2  
thread_b 2
```

```
thread_a 96  
thread_b 96  
thread_c 97  
thread_a 97  
thread_b 97  
thread_c 98  
thread_a 98  
thread_b 98  
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  
$
```

可以准确的执行 uthread 测试

**评分:**

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-thread Uthread  
make: "kernel/kernel"已是最新。  
== Test uthread == uthread: OK (2.9s)  
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

可以通过所有测试

## b. Using threads

**实验目的:**

在本作业中，您将使用哈希表探索线程和锁的并行编程。您应该在具有多个内核的真实 Linux 或 MacOS 计算机（不是 xv6，不是 qemu）上执行此任务。大多数最新的笔记本电脑都有多核处理器。

这里提供了一个测试程序，使用哈希表来添加元素并取出。要在这个程序上实现多线程以提高运行速度。但是在刚开始时执行这个测试程序时，若选择两个线程会导致大量的 key missing，这是因为在多线程并发处理时没有处理好同步互斥关系，所以需要加上互斥锁来确保程序能够正确的执行而不会出现 key missing 的情况。

但是如果全部加锁的话，会导致执行速度变慢，而达不到多线程加速的效果，所以还需要进一步分析什么时候需要加锁而什么时候不必。

## 实验分析：

当两个线程向哈希表的同一个bucket分别插入不同的元素的时候，如果当他们都找到了插入位置（即该bucket）的最后面，但是两者都还没插入，此时无论哪个先插入都会被后插入的“替换”掉。

这样会导致 key missing，所以要通过锁建立一个合适的同步互斥机制来避免这种情况的发生。我们先可以考虑在每次添加键的时候都设置一个互斥信号量，必须等待前一个线程的添加完成后，下一个才能继续添加。这样可以解决上述的冲突，可以保证添加键后不会出现键的缺失情况；然后考虑一下：这样做在每次添加键都要开关锁，会造成比较大的开销。实际上，当不同线程添加的键的哈希值不同时，他们是无关的，是可以完全并行的，只有当两个键的哈希值相同才可能出现错误。所以我们可以修改互斥信号，为每个 bucket 添加一个互斥信号，每次向某个 bucket 插入键时只锁住该 bucket 的互斥信号即可，只样不仅可以解决冲突，还可以大大提高并行度，从而提高执行速度。

## 解决 ph\_safe :

### 实验核心代码：

ph.c

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <sys/time.h>

#define NBUCKET 5
#define NKEYS 100000

struct entry {
    int key;
    int value;
    struct entry *next;
};

struct entry *table[NBUCKET];
int keys[NKEYS];
int nthread = 1;
pthread_mutex_t lock;

double
now()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}
```

```

static void
insert(int key, int value, struct entry **p, struct entry *n)
{
    struct entry *e = malloc(sizeof(struct entry));
    e->key = key;
    e->value = value;
    e->next = n;
    *p = e;
}

static
void put(int key, int value)
{
    int i = key % NBUCKET;

    // is the key already present?
    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key)
            break;
    }
    if(e){
        // update the existing key.
        e->value = value;
    } else {
        // the new is new.
        insert(key, value, &table[i], table[i]);
    }
}

static struct entry*
get(int key)
{
    int i = key % NBUCKET;

    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key) break;
    }

    return e;
}

static void *
put_thread(void *xa)
{
    int n = (int) (long) xa; // thread number
    int b = NKEYS/nthread;

    for (int i = 0; i < b; i++) {
        pthread_mutex_lock(&lock);
        put(keys[b*n + i], n);
        pthread_mutex_unlock(&lock);
    }

    return NULL;
}

```

```

}

static void *
get_thread(void *xa)
{
    int n = (int) (long) xa; // thread number
    int missing = 0;

    for (int i = 0; i < NKEYS; i++) {
        struct entry *e = get(keys[i]);
        if (e == 0) missing++;
    }
    printf("%d: %d keys missing\n", n, missing);
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t *tha;
    void *value;
    double t1, t0;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s nthreads\n", argv[0]);
        exit(-1);
    }
    nthread = atoi(argv[1]);
    tha = malloc(sizeof(pthread_t) * nthread);
    srand(0);
    assert(NKEYS % nthread == 0);
    for (int i = 0; i < NKEYS; i++) {
        keys[i] = random();
    }

    //
    // first the puts
    //
    pthread_mutex_init(&lock, NULL);
    t0 = now();
    for(int i = 0; i < nthread; i++) {
        assert(pthread_create(&tha[i], NULL, put_thread, (void *) (long) i) == 0);
    }
    for(int i = 0; i < nthread; i++) {
        assert(pthread_join(tha[i], &value) == 0);
    }
    t1 = now();

    printf("%d puts, %.3f seconds, %.0f puts/second\n",
        NKEYS, t1 - t0, NKEYS / (t1 - t0));

    //
    // now the gets
    //
    t0 = now();
    for(int i = 0; i < nthread; i++) {
        assert(pthread_create(&tha[i], NULL, get_thread, (void *) (long) i) == 0);
    }
}

```

```

    for(int i = 0; i < nthread; i++) {
        assert(pthread_join(thr[i], &value) == 0);
    }
    t1 = now();

    printf("%d gets, %.3f seconds, %.0f gets/second\n",
           NKEYS*nthread, t1 - t0, (NKEYS*nthread) / (t1 - t0));
}

```

测试:

```

yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./ph 2
100000 puts, 9.066 seconds, 11030 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 6.896 seconds, 29001 gets/second

```

可以看到，所有的 key missing 已经被消除，但是执行速度比较慢

评分:

```

== Test ph_safe == make[1]: 进入目录"/home/yangxin/桌面/xv6-labs-2020"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/yangxin/桌面/xv6-labs-2020"
ph_safe: OK (14.7s)

```

可以通过所有测试

## 解决 ph\_fast:

实验核心代码:

ph.c

```

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <sys/time.h>

#define NBUCKET 5
#define NKEYS 100000

struct entry {
    int key;
    int value;
    struct entry *next;
};

struct entry *table[NBUCKET];
int keys[NKEYS];
int nthread = 1;
pthread_mutex_t lock[NBUCKET];

```



```

double
now()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

static void
insert(int key, int value, struct entry **p, struct entry *n)
{
    struct entry *e = malloc(sizeof(struct entry));
    e->key = key;
    e->value = value;
    e->next = n;
    *p = e;
}

static
void put(int key, int value)
{
    int i = key % NBUCKET;
    pthread_mutex_lock(&lock[i]);

    // is the key already present?
    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key)
            break;
    }
    if(e){
        // update the existing key.
        e->value = value;
    } else {
        // the new is new.
        insert(key, value, &table[i], table[i]);
    }
    pthread_mutex_unlock(&lock[i]);
}

static struct entry*
get(int key)
{
    int i = key % NBUCKET;

    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key) break;
    }

    return e;
}

static void *
put_thread(void *xa)
{

```

```

int n = (int) (long) xa; // thread number
int b = NKEYS/nthread;

for (int i = 0; i < b; i++) {
    put(keys[b*n + i], n);
}

return NULL;
}

static void *
get_thread(void *xa)
{
    int n = (int) (long) xa; // thread number
    int missing = 0;

    for (int i = 0; i < NKEYS; i++) {
        struct entry *e = get(keys[i]);
        if (e == 0) missing++;
    }
    printf("%d: %d keys missing\n", n, missing);
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t *tha;
    void *value;
    double t1, t0;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s nthreads\n", argv[0]);
        exit(-1);
    }
    nthread = atoi(argv[1]);
    tha = malloc(sizeof(pthread_t) * nthread);
    srand(0);
    assert(NKEYS % nthread == 0);
    for (int i = 0; i < NKEYS; i++) {
        keys[i] = random();
    }

    //
    // first the puts
    //
    for (int i = 0; i < NBUCKET; i++)
        pthread_mutex_init(&lock[i], NULL);
    t0 = now();
    for(int i = 0; i < nthread; i++) {
        assert(pthread_create(&tha[i], NULL, put_thread, (void *) (long) i) == 0);
    }
    for(int i = 0; i < nthread; i++) {
        assert(pthread_join(tha[i], &value) == 0);
    }
    t1 = now();

    printf("%d puts, %.3f seconds, %.0f puts/second\n",

```

```

        NKEYS, t1 - t0, NKEYS / (t1 - t0));

    //
    // now the gets
    //
    t0 = now();
    for(int i = 0; i < nthread; i++) {
        assert(pthread_create(&tha[i], NULL, get_thread, (void *) (long) i) == 0);
    }
    for(int i = 0; i < nthread; i++) {
        assert(pthread_join(tha[i], &value) == 0);
    }
    t1 = now();

    printf("%d gets, %.3f seconds, %.0f gets/second\n",
        NKEYS*nthread, t1 - t0, (NKEYS*nthread) / (t1 - t0));
}

```

测试:

```

yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./ph 2
100000 puts, 4.911 seconds, 20361 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 7.262 seconds, 27540 gets/second

```

可以看到，在没有 key missing 的情况下，速度也提高了很多

评分:

```

== Test ph_fast == make[1]: 进入目录"/home/yangxin/桌面/xv6-labs-2020"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/yangxin/桌面/xv6-labs-2020"
ph_fast: OK (26.1s)

```

可以通过所有测试

## c. Barrier

### 实验目的:

在这个任务中，您将实现一个屏障：应用程序中的一个点，所有参与的线程都必须等待，直到所有其他参与的线程也到达该点。您将使用 pthread 条件变量，这是一种类似于 xv6 的睡眠和唤醒的序列协调技术。

$n$  指定在屏障上同步的线程数（barrier.c 中的 nthread）。每个线程执行一个循环。在每次循环迭代中，一个线程调用 barrier()，然后休眠随机数微秒。断言触发，因为一个线程在另一个线程到达屏障之前离开了屏障。期望的行为是每个线程都阻塞在屏障（）中，直到它们的所有 nthread 都调用了屏障（）。

您的目标是实现所需的屏障行为。

## 实验分析：

本次实验实际上是另一种同步互斥机制的实现。在程序中让多个线程反复执行一个 barrier() 函数，但是要求必须等待所有线程执行完一次 barrier() 后，他们才能继续执行后面的代码。也就是说，所有线程必须阻塞在 barrier() 中，除非所有的线程都已经到达该处。

我们只需要在 barrier() 中每次执行时增加一次 nthread，如果 nthread 还没到达线程总数，则让其加入等待队列并阻塞；若达到线程总数，则释放所有等待队列中的线程并重新计数。

## 实验核心代码：

barrier.c

```
static void
barrier()
{
    pthread_mutex_lock(&bstate.barrier_mutex);
    bstate.nthread++;
    if (bstate.nthread == nthread) {
        bstate.nthread = 0;
        bstate.round++;
        pthread_cond_broadcast(&bstate.barrier_cond);
    } else {
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }
    pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

## 测试：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./barrier 2
OK; passed
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./barrier 3
OK; passed
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

## 评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-thread barrier
make: "kernel/kernel"已是最新。
== Test barrier == make: "barrier"已是最新。
barrier: OK (11.8s)
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

可以通过所有测试

## 实验评分：

---

按照实验要求对本次实验的所有小实验进行评分，结果如图：

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (4.2s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录"/home/yangxin/桌面/xv6-labs-2020"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/yangxin/桌面/xv6-labs-2020"
ph_safe: OK (11.6s)
== Test ph_fast == make[1]: 进入目录"/home/yangxin/桌面/xv6-labs-2020"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/yangxin/桌面/xv6-labs-2020"
ph_fast: OK (26.0s)
== Test barrier == make[1]: 进入目录"/home/yangxin/桌面/xv6-labs-2020"
make[1]: "barrier"已是最新。
make[1]: 离开目录"/home/yangxin/桌面/xv6-labs-2020"
barrier: OK (11.3s)
== Test time ==
time: OK
Score: 60/60
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

通过了所有测试

## 问题以及解决办法：

### a.多线程为何会造成初始的哈希表键缺失

原始的插入键代码如下：

```
static
void put(int key, int value)
{
    int i = key % NBUCKET;

    // is the key already present?
    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key)
            break;
    }
    if(e){
        // update the existing key.
        e->value = value;
    } else {
        // the new is new.
        insert(key, value, &table[i], table[i]);
    }
}
```

可以看到，首先计算出插入的 bucket 的序号  $i$  以后，就要寻找该 bucket 的插入位置（一般是在最后）。如果两个线程一前一后执行，先插入一个键，然后插入另一个键，则不会发生任何错误；但是如果两个线程同时完成了找到插入位置这一过程，并且都还没插入时，此时二者的插入位置是同一个（即当前桶中最后一个元素的后面），然后无论以何种顺序完成插入后，最先插入的那个键一定会被最后插

入的键替代掉，这样就会导致键的缺失。这就是为什么需要在插入键时加入互斥锁以防止错误产生的原因。

## 实验心得：

---

1. 本次实验体会到了线程的概念，熟悉了线程的创建，切换等操作，对于上下文的回复也有了一定的体会
2. 本次实验还发现并尝试解决了多线程中的同步互斥问题：由于多线程的出现，一些必须保证一定先后执行次序的地方不能得到保证，从而会导致出现一系列错误，为了避免这种错误的产生，可以通过使用锁来保证一定的同步和互斥关系
3. 在使用锁时，要注意尽量在最小的地方使用互斥锁，即某些不需要维持同步互斥的地方尽量让他们不受锁的影响，否则会导致多线程加速程序执行效率的优势得不到体现
4. 了解了一些常见的原子操作和锁