Lab - Multithreading 目录

1. Uthread: switching between threads (moderate)

1.1 实验目的

- 设计和实现上下文切换机制，支持用户级线程系统

1.2 实验步骤

1. 线程是运行在进程上下文中的逻辑流，每个线程都有自己的上下文：唯一的整数线程ID、栈、栈指针、程序计数器、通用目的寄存器和条件码。所有运行在一个进程里的线程共享该线程的整个虚拟地址空间。

   当用户级线程系统要进行切换时，应当把线程的现场保存下来，这一现场就被称作上下文（`context`），在线程切换时上下文也发生切换。

2. 在`user/uthread.c`中添加上下文`thread_context`的定义，用于保存现场。那么每个线程都应该维护一个自己的`context`用于保存和恢复：

```
struct thread_context {
    // 这与proc结构体中的context是一样的，都需要保存下列寄存器的值
    // 不同之处在于而thread_context是用于线程切换的
    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 thread's stack */
    int state;              /* FREE, RUNNING, RUNNABLE */

    struct thread_context context; // 线程的上下文
};
```

3. 在`user/uthread.c`中补充`thread_create()`函数，补充在创建函数过程中应当完成的操作，该函数的逻辑是找到一个空闲线程并为其分配，而后将其状态设置为占用（运行），那么接下来应该添加的操作是完成现场的保存；函数接受了一个传入的函数指针`func`，那么接下来要完成的便是调用该函数并复制`stack pointer`：

```
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->context.ra = (uint64)func;                    // 执行传入的函数
（thread_switch）
    t->context.sp = (uint64)(t->stack + STACK_SIZE); // 复制栈顶的stack
pointer
}
```

4. 在`user/uthread.c`中补充`thread_schedule()`函数，找到可用线程`t`，并把当前线程`current_thread`的`context`保存在其中，具体的保存过程调用函数`thread_switch()`实现：

```
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((uint64)&t->context, (uint64)&current_thread-
>context);
    }
    else
        next_thread = 0;
}
```

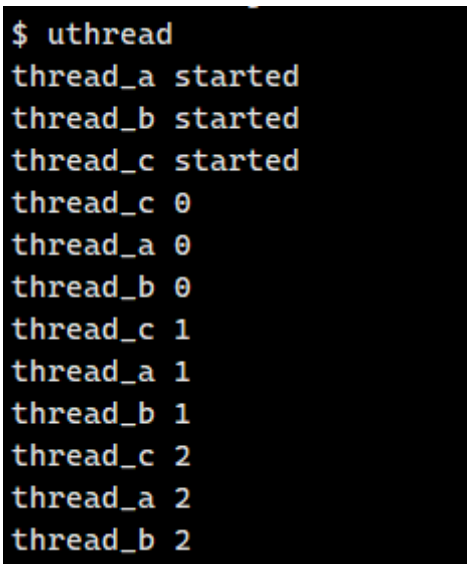
5. 在`user/uthread_switch.S`中用汇编实现`thread_switch`函数，实现寄存器的保存与读取：

```
thread_switch:
    /* YOUR CODE HERE */
    /* copy the content in kernel/swtch.S */
swtch:
        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 /* ret to ra*/
```

6. 运行`uthread`，进行多线程测试，得到结果如下：



7. 运行`grade`测试，结果如下：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-
thread
make: “kernel/kernel”已是最新。
== Test uthread == uthread: OK (1.4s)
```

1.3 实验中遇到的问题和解决方法

- **问题**：在 `thread_schedule()` 中如何切换到下一个可运行的线程

  解决方法：在 `thread_schedule()` 中，需要选择一个可运行的线程，并切换到它。可以使用一个全局变量或其他数据结构来维护线程队列，并选择下一个可运行的线程。一旦找到下一个线程，可以通过调用 `thread_switch()` 来进行实际的线程切换。

- **问题**：如何在 `thread_switch()` 中保存和恢复寄存器状态

  解决方法： 在 `thread_switch()` 中，只需要保存和恢复 `callee-save` 寄存器，因为这些寄存器在函数调用过程中需要被调用者保存。可以使用类似于函数调用的方式来保存和恢复这些寄存器。同时，需要注意恢复栈指针和程序计数器等寄存器，以确保线程可以从上次离开的地方继续执行。

1.4 实验心得

在本次实验中，我着重学习和理解了用户级线程的设计和实现。这是一个相对复杂的任务，需要在操作系统中实现上下文切换机制，以便支持多个用户级线程的运行。在开始实验之前，我花了很多时间研究用户级线程的概念和工作原理。我深入阅读了相关的文档和代码，特别是 `user/uthread.c` 和 `user/uthread_switch.S` 文件。这帮助我对用户级线程的运行机制有了全面的认识，并为我实现线程切换提供了指导。在实现线程切换的过程中，我遇到了一些困难，特别是关于寄存器状态的保存和恢复。我利用 gdb 和单步调试的方式，逐步跟踪线程切换的过程，确保每一步都正确执行，并且寄存器状态得到正确保存和恢复。通过本次实验，我对用户级线程的实现和上下文切换有了更深入的理解，这对我日后工作学习有很大的帮助。

2. Using threads (moderate)

2.1 实验目的

- 使用线程和锁探索并行程序

2.2 实验步骤

1. 使用 `man pthreads` 了解 `pthreads` 的使用方法，部分如下：

```
NAME
       pthreads - POSIX threads

DESCRIPTION
       POSIX.1 specifies a set of interfaces (functions, header files) for
threaded pro-
       gramming commonly known as POSIX threads, or Pthreads.  A single
process can con-
       tain  multiple  threads,  all  of  which  are  executing the same
program.  These
       threads share the same global memory (data and heap segments),  but
each  thread
       has its own stack (automatic variables).

       POSIX.1 also requires that threads share a range of other attributes
(i.e., these
       attributes are process-wide rather than per-thread)
```

主要方法是使用下面四个函数调用：

```
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
```

2. 理解`notxv6/ph.c`中的哈希表实现和测试方法，创建多个散列桶并不断`put()`入新元素，使用`get()`来检验。
3. 使用`make ph`，将`ph.c`编译成可执行文件，执行`./ph 1`，开启单线程测试，结果如下：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-multithreading$ ./ph 1
100000 puts, 6.326 seconds, 15808 puts/second
0: 0 keys missing
100000 gets, 6.408 seconds, 15606 gets/second
```

4. 执行`./ph 2`，开启多线程测试，结果如下，发现产生了错误，键本该生成在哈希表中，但是发生了丢失：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-multithreading$ ./ph 2
100000 puts, 2.930 seconds, 34135 puts/second
1: 16531 keys missing
0: 16531 keys missing
200000 gets, 6.673 seconds, 29971 gets/second
```

5. 原因在于同时访问哈希表时发生了冲突，因此考虑使用锁来解决，先在`ph.c`中为每个哈希桶定义一个锁：

```
pthread_mutex_t lock[NBUCKET]; // 为每个散列桶设置一个锁
```

6. 修改`put()`函数，使得多线程间不再发生冲突：

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

    // is the key already present?
    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
```

```
            // 找到插入位置
            if (e->key == key)
                break;
        }
        if (e) {
            // update the existing key.
            e->value = value;
        }
        else {
            // the new is new.
            pthread_mutex_lock(&lock[i]); // 对第i个bucket加锁
            insert(key, value, &table[i], table[i]);
            pthread_mutex_unlock(&lock[i]); // 解锁
        }
    }
```

7. 重新编译生成可执行文件，执行`./ph 2`结果如下：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-multithreading$ ./ph 2
100000 puts, 3.195 seconds, 31303 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 6.380 seconds, 31347 gets/second
```

8. 运行grade的phsafe测试和phfast测试，结果如下：

```
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录“/home/zheng-linux/xv6-labs-2022/xv6-lab-multithreading”
ph_safe: OK (9.7s)
```

```
== Test ph_fast == make[1]: 进入目录“/home/zheng-linux/xv
”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/zheng-linux/xv6-labs-2022/xv6-lab
ph_fast: OK (22.3s)
```

2.3 实验中遇到的问题和解决方法

- **问题**：为何运行多线程测试`./ph 2`时会出现大量的键值对丢失的情况

  解决方法：多线程环境下，由于没有加锁保护，多个线程可能会同时访问哈希表，导致竞争，竞争条件会导致一些`put()`操作被覆盖或丢失，从而导致部分键值对没有成功插入哈希表。

2.4 实验心得

在本次实验中我学会了如何使用线程和锁来实现并行编程以及如何处理多线程环境下的竞争问题、学习了如何使用`pthread`线程库，掌握了线程的创建和销毁、锁的使用以及线程同步的方法。同时通过调试和测试，我更深入地理解了并行编程中的一些常见问题，比如竞争条件和死锁等。实验中我使用了具有多核处理器的物理计算机，这使得并行线程得到了更好的执行效果。通过对比单线程和多线程的生成速度，发现多线程使得程序效率得到了大幅提升，但也同时带来了一系列的控制问题。总体而言，通过探索并行编程和锁的使用，我加深了对多线程编程的理解，并提升了在并发环境下开发正确且高效程序的能力。

3. Barrier(moderate)

3.1 实验目的

- 实现屏障（Barrier），使得所有线程必须在此等待

3.2 实验步骤

1. 理解`notxv6/barrier.c`中的实现和测试方法，程序创建多个线程，每个线程执行一个循环。在每次循环迭代中，每个线程调用`barrier()`，而后在随机休眠一段时间。要让每个线程都阻塞在`barrier()`处，只有当所有线程都调用了`barrier()`再继续执行。实验中会用到`pthread`中的两个函数调用：

```
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
```

其中`cond_wait`使得线程在`cond`条件下睡眠并释放互斥锁，当醒来的时候又会获取锁，而`cond_broadcast`会唤醒所有因`cond`条件而睡眠的线程。

对于一系列的`barrier`调用，每一连串的调用为一轮（`round`）。`bstate.round`记录当前轮数。每次当所有线程都到达屏障时，都应增加`bstate.round`。

```
struct barrier {
    pthread_mutex_t barrier_mutex;
    pthread_cond_t barrier_cond;
    int nthread; // Number of threads that have reached this round of the
barrier
    int round;   // Barrier round
} bstate;
```

2. 修改`barrier()`函数，使其在所有线程调用`barrier()`之前都保持阻塞状态，并且增加`round`，细节见注释：

```
static void barrier()
{
    // YOUR CODE HERE
    //
    // Block until all threads have called barrier() and
    // then increment bstate.round.
    //

    pthread_mutex_lock(&bstate.barrier_mutex); // 上锁
    bstate.nthread++;                          // 线程数+1

    if (bstate.nthread == nthread) {
        // 当所有线程都到达时（bstate.nthread == nthread）
        bstate.round++;                                 // 新增一轮
        bstate.nthread = 0;                             // 重置线程数
        pthread_cond_broadcast(&bstate.barrier_cond); // 唤醒所有线程
    }
    else {
        // 当前线程进入睡眠状态等待
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }

    pthread_mutex_unlock(&bstate.barrier_mutex); // 一轮结束，解锁
}
```

3. 使用`make barrier`编译生成可执行文件，使用单线程测试，结果如下：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-multithreading$ ./barrier 1
OK; passed
```

4. 使用2线程、4线程测试，结果分别如下：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-multithreading$ ./barrier 2
OK; passed
```

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-multithreading$ ./barrier 4
OK; passed
```

5. 运行`grade`测试，结果如下：

```
== Test barrier == make[1]: 进入目录“/home/zheng-linux/xv6-labs-2022/xv6-l
”
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录“/home/zheng-linux/xv6-labs-2022/xv6-lab-multithreading”
barrier: OK (2.9s)
```

3.3 实验中遇到的问题和解决方法

- **问题**：运行 `./barrier 2` 时出现断言失败的错误

  解决方法：多个线程在并发地通过`barrier`，导致一个线程离开`barrier`时其他线程还没有到达，从而导致断言失败。使用`cond`和互斥锁来确保所有线程都能正确地等待在`barrier`处即可。

3.4 实验心得

在本次实验中，我使用了条件变量和互斥锁来实现线程的同步。通过正确使用条件变量和互斥锁，我解决了多线程并发访问`barrier`的问题，确保了所有线程都能正确地等待在`barrier`处，直到所有线程都到达`barrier`后再一起继续执行。在处理连续的`barrier`调用时，使用一个轮次计数器来跟踪当前的`barrier`轮次，并避免了多个`barrier`调用之间的混淆。通过实现`barrier`，我更深入地理解了线程同步的概念和使用条件变量和互斥锁的技术，对多线程编程有了更深入的认知。