Lab6-Multithreading

Uthread: switching between threads

实验目的

- 设计上下文切换机制
  - 实现创建线程的功能,实现线程之间的上下文切换机制
- 保存和恢复寄存器
  - 在`thread_switch`中保存被切换出去的线程的寄存器。
  - 恢复被切换进来的线程的寄存器

实验步骤

1. **修改`struct thread`**：在`struct thread`中增加保存寄存器的字段，同时新建线程上下文结构体的定义`thread_context`

```
// 线程的上下文
struct thread_context {

    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
};
```

2. **实现** `thread_create()`：创建线程并初始化其栈和寄存器。为新创建的线程初始化必要的上下文，使其能够正确执行传入的函数并在自己的栈空间上运行。这样，当调度器第一次调度该线程时，线程会从传入的函数开始执行，并且使用分配给它的栈空间。

```
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); // 复制栈顶指针
}
```

3. **实现** `thread_schedule()`：调度可运行的线程，调用 `thread_switch` 进行上下文切换。

```
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;
}
```

4. **实现** `thread_switch`：在汇编中保存当前线程的寄存器，恢复下一个线程的寄存器，完成上下文切换。

```
swtch:
    sd ra, 0(a0)          // 保存返回地址寄存器
    sd sp, 8(a0)          // 保存栈指针寄存器
    sd s0, 16(a0)         // 保存s0寄存器
    sd s1, 24(a0)         // 保存s1寄存器
    sd s2, 32(a0)         // 保存s2寄存器
    sd s3, 40(a0)         // 保存s3寄存器
    sd s4, 48(a0)         // 保存s4寄存器
    sd s5, 56(a0)         // 保存s5寄存器
    sd s6, 64(a0)         // 保存s6寄存器
    sd s7, 72(a0)         // 保存s7寄存器
    sd s8, 80(a0)         // 保存s8寄存器
    sd s9, 88(a0)         // 保存s9寄存器
    sd s10, 96(a0)        // 保存s10寄存器
    sd s11, 104(a0)       // 保存s11寄存器

    ld ra, 0(a1)          // 恢复返回地址寄存器
    ld sp, 8(a1)          // 恢复栈指针寄存器
    ld s0, 16(a1)         // 恢复s0寄存器
    ld s1, 24(a1)         // 恢复s1寄存器
    ld s2, 32(a1)         // 恢复s2寄存器
    ld s3, 40(a1)         // 恢复s3寄存器
    ld s4, 48(a1)         // 恢复s4寄存器
    ld s5, 56(a1)         // 恢复s5寄存器
    ld s6, 64(a1)         // 恢复s6寄存器
    ld s7, 72(a1)         // 恢复s7寄存器
    ld s8, 80(a1)         // 恢复s8寄存器
    ld s9, 88(a1)         // 恢复s9寄存器
    ld s10, 96(a1)        // 恢复s10寄存器
    ld s11, 104(a1)       // 恢复s11寄存器

    ret                   // 返回到ra指示的地址
```

5. 测试

运行`make qemu`后执行`uthread`：

```
thread_c 95
thread_a 95
thread_b 95
thread_c 96
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
```

运行`make grade`：

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (3.9s)
```

实验中遇到的问题和解决办法

本次实验的难点在于编写汇编语言实现寄存器的保存和恢复过程。需要清楚理解在上下文切换过程中哪些寄存器需要保存和恢复。通过阅读材料可以知道，在RISC-V架构中，保存和恢复的是callee-saved寄存器（即`ra`、`sp`、`s0`到`s11`），`sd`指令用于保存寄存器内容到指定内存位置，`ld`指令用于从指定内存位置恢复寄存器内容，`ret`指令用于返回到保存的返回地址，完成上下文切换。同时还要注意传入参数时哪个是要保存和恢复的线程。

实验心得

在这次实验中，我最大的收获是深入理解了上下文切换的实现及其复杂性。以前在学习操作系统时，知道上下文切换是多线程和多任务系统中的核心机制，但这次通过实际编写`thread_switch`汇编代码，真正体会到了实现它的细节和挑战。特别是实现寄存器保存和恢复的部分，让我意识到即便是看似简单的汇编代码，也需要充分考虑执行顺序和内存布局。

最初由于对某些寄存器的作用和保存顺序不够熟悉，导致几次程序崩溃，经过仔细阅读文档和反复调试，才逐步找出问题所在并加以修正。这个过程虽然有些挫折，但最终看到线程正确切换并输出预期结果时，感到非常有成就感。通过这次实验，我不仅掌握了具体的上下文切换实现方法，更体会到细致和严谨在系统编程中的重要性。

Using threads

实验目的

- 理解并行编程的基本概念
- 分析多线程对哈希表操作的影响
- 解决多线程环境下的数据一致性问题，优化并行性能

实验步骤

1. 学习`pthreads`的使用方法, 主要学习以下的四个函数

   ```
   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. 学习`ph.c`文件中哈希表的实现

   `ph.c`文件实现了一个简单的哈希表，并且使用多线程进行并行操作

   - 数据结构

```
#define NBUCKET 5 // 哈希桶的数量
#define NKEYS 100000 // 存储键的数组大小

struct entry {
  int key;
  int value;
  struct entry *next;
}; // 链表节点的结构体，存储键值对和指向下一个节点的指针
struct entry *table[NBUCKET]; // 哈希表数组
int keys[NKEYS]; // 存储要插入哈希表的键的数组
int nthread = 1; // 用于记录线程数量的全局变量
```

- 插入函数

```
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;
}
//该函数用于在链表中插入一个新的键值对节点。
//key是键，value是值，p是指向要插入位置的指针，n是下一个节点的指针。
```

- put函数

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

  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  if(e){
    e->value = value;
  } else {
    insert(key, value, &table[i], table[i]);
  }
}
// 该函数用于将键值对插入到哈希表中。如果键已经存在，则更新其值；
// 否则插入新节点。
```

- get函数

```
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;
  }

  re
turn e;
}
// 用于从哈希表中获取指定键的节点。
```

- 线程函数

```
static void *put_thread(void *xa)
{
  int n = (int) (long) xa;
  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;
  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;
}
```

- `put_thread`函数：用于线程并行执行插入操作。
- `get_thread`函数：用于线程并行执行获取操作，并统计缺失的键数。

3. 执行`make ph`编译`ph`程序

   - 使用单线程运行`./ph 1`，观察输出结果，记录每秒插入和获取的操作数以及缺失的键数

     ```
     (base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./ph 1
     100000 puts, 9.502 seconds, 10524 puts/second
     0: 0 keys missing
     100000 gets, 9.483 seconds, 10546 gets/second
     ```

   - 使用多线程运行`./ph 2`，观察输出结果，记录每秒插入和获取的操作数以及缺失的键数。发现产生了错误，键本该生成在哈希表中，但是发生了丢失。

     ```
     (base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./ph 2
     100000 puts, 4.090 seconds, 24448 puts/second
     0: 16825 keys missing
     1: 16825 keys missing
     200000 gets, 9.766 seconds, 20479 gets/second
     ```

4. 多线程运行出现错误的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]; // 每个散列桶一个锁
   ```

5. 在程序初始化时调用`pthread_mutex_init(&lock, NULL);`初始化锁

   ```
   for (int i = 0; i < NBUCKET; i++) {
       //散列桶内锁初始化
       pthread_mutex_init(&lock[i], NULL);
   }
   ```

6. 在`put()`函数中添加`pthread_mutex_lock(&lock);`和`pthread_mutex_unlock(&lock);`语句，确保在进行哈希表操作时持有锁，以避免多线程冲突。

```
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.
        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. 重新编译并执行 `./ph 2`

```
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./ph 2
100000 puts, 4.931 seconds, 20280 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 9.662 seconds, 20700 gets/second
```

8. 测试

运行make grade:

```
== Test ph_safe == make[1]: Entering directory '/ro
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/p
make[1]: Leaving directory '/root/workspace/persona
ph_safe: OK (12.5s)
== Test ph_fast == make[1]: Entering directory '/ro
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/root/workspace/persona
ph_fast: OK (29.3s)
```

实验中遇到的问题和解决办法

为每个哈希桶添加完锁之后运行`./ph 2`发现还是会出现丢失的情况，回顾了步骤一中的4个函数，发现在主函数中没有初始化锁，初始化之后发现还是有丢失的情况，再反复检查后才意识到需要重新编译`make ph`,两个错误解决后便得到了正确的结果。

实验心得

在完成这个并行编程实验的过程中，我学到了很多关于多线程和数据同步的知识。首先，通过初始代码的测试，我清楚地看到了多线程环境下数据一致性问题的严重性。在单线程模式下，程序运行正常且没有键缺失，但在多线程模式下，键缺失现象非常明显。这让我意识到，在并行编程中，数据同步是一个至关重要的环节。于是，我在代码中添加了锁机制来保护共享数据结构。这样可以允许某些插入操作并行进行，从而提高了整体性能。总的来说，这次实验让我深刻体会到了并行编程的复杂性和重要性。掌握多线程编程和数据同步的技巧，将对我今后的编程实践有很大的帮助。

Barrier

实验目的

- 理解和应用线程同步机制
- 实现屏障功能，处理多轮屏障调用
- 防止线程竞态

实验步骤

1. 理解屏障：屏障(barrier)，即一个应用程序中的同步点，所有参与的线程必须在该点等待，直到所有其他参与线程也达到该点。应对连续的屏障调用，每一轮都应该确保所有线程同步到达屏障并正确处理。
2. 理解`notxv6/barrier.c`中的实现，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。

   `pthread_cond_wait(&cond, &mutex);`

   - 这个函数让线程进入睡眠状态，等待条件变量`cond`满足，同时释放互斥锁`mutex`
   - 当条件变量满足时，线程被唤醒并重新获取互斥锁`mutex`，然后继续执行。

   `pthread_cond_broadcast(&cond);`

- 这个函数唤醒所有正在等待条件变量 `cond` 的线程。

```
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
```

3. 对于一些了的barrier调用，所有线程的一次调用为一轮，当所有线程都到达屏障时需要将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;
```

4. 执行 `make barrier` 编译 `barrier` 程序, 使用2个线程运行 `./barrier 2` ，观察输出结果和错误信息，确认程序在断言失败时终止。

```
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./barrier 2
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
Aborted (core dumped)
```

5. 实现 `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 `bstate.round`

```
static void barrier()
{
    pthread_mutex_lock(&bstate.barrier_mutex); // 上锁

    bstate.nthread++;                          // 增加已到达屏障的线程数

    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_condceste.round;
        while (current_round == bstate.round) {
            pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
        }
    }

    pthread_mutex_unlock(&bstate.barrier_mutex); // 解锁
}
```

6. 测试

运行`make barrier`编译，首先进行单线程测试`./ barrier 1`：

```
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./barrier 1
OK; passed
```

进行多线程测试：

```
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./barrier 2
OK; passed
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./barrier 4
OK; passed
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./barrier 6
OK; passed
```

运行`make grade`测试：

```
== Test barrier == make[1]: Entering directory '/root/workspace/personal_data/zwc/Xv6/xv6-labs-2022'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/root/workspace/personal_data/zwc/Xv6/xv6-labs-2022'
barrier: OK (2.7s)
```

实验中遇到的问题和解决办法

当线程调用barrier()中，对bstate结构体中的共享变量（`nthread`和`round`）进行访问修改时，需要加锁，多线程同时访问和修改这些变量会导致竞态条件，导致数据不一致。通过互斥锁，可以确保同一时间只有一个线程能够访问和修改这些变量。起初没有加锁导致测试无法通过，加锁之后即可解决该问题。

实验心得

在完成这个屏障(barrier)实验的过程中，我对多线程编程和同步机制有了更深刻的理解。实验的主要任务是实现一个屏障点，所有参与的线程必须在该点等待，直到所有线程都到达为止。这涉及到pthread库中的条件变量和互斥锁的使用。由于多个线程并发修改共享变量`nthread`和`round`，需要确保这些操作是原子性的。通过使用互斥锁，成功解决了这个问题。这个实验让我更深入地理解了线程同步的概念和实践。我还认识到细致的测试和调试在多线程编程中的重要性，稍有不慎就可能导致难以察觉的同步错误。

总结

本实验旨在通过实现和测试多线程机制，加深对多线程编程、数据同步以及线程调度的理解。具体目标包括实现线程之间的上下文切换、分析多线程对数据结构操作的影响、解决多线程环境下的数据一致性问题，并实现线程同步机制。

- Uthread: switching between threads

- **目的**:
  - 设计和实现上下文切换机制，创建和管理线程。
  - 实现线程之间的上下文切换，包括保存和恢复寄存器状态。
- **心得**： 通过实现线程上下文切换，理解了多线程的调度和寄存器管理的复杂性。掌握了具体的上下文切换实现方法，并体会到系统编程的细致和严谨的重要性。

- Using threads（1种锁）
  - **目的**:
    - 理解并行编程的基本概念，分析多线程对哈希表操作的影响。
    - 解决多线程环境下的数据一致性问题，优化并行性能。
  - **心得**： 通过添加锁机制保护共享数据结构，解决了多线程环境下的数据一致性问题。掌握了多线程编程和数据同步的技巧，深刻理解了并行编程的复杂性和重要性。

- Barrier （1种锁）
  - **目的**:
    - 理解和应用线程同步机制，防止线程竞态。
    - 实现屏障功能，确保所有线程同步到达屏障。
  - **心得**： 通过实现屏障机制，理解了线程同步和竞态条件的处理。掌握了条件变量和互斥锁的使用，认识到细致测试和调试在多线程编程中的重要性。