Lab08: locks

Barrier Implementation Using Pthreads (Moderate)

1) 实验目的

本实验的目标是实现一个线程屏障（barrier），在该屏障点，所有参与的线程必须等待直到所有其他线程也到达该点。你将使用 pthread 条件变量来实现这一点，它们是线程同步的工具。

2) 实验步骤

1. 理解现有代码

1. **查看 `barrier.c` 的代码**：
   - 找到 `struct barrier` 的定义以及 `barrier_init` 函数的实现。
   - 确定 `barrier` 函数需要完成的任务。
2. **分析当前 `barrier` 函数的缺陷**：
   - **断言失败**：如果线程在所有其他线程到达屏障之前就离开了屏障，程序将触发断言错误。这表明屏障的实现存在问题。

2. 实现屏障功能

1. **修改 `barrier.c` 中的 `barrier` 函数**：
   - 你需要确保每个线程在调用 `barrier` 时，直到所有参与的线程都到达屏障后，才允许线程继续执行。
   - 使用 `pthread_cond_wait` 和 `pthread_cond_broadcast` 实现线程同步。

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

struct barrier {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int count;      // 当前到达屏障的线程数
    int threshold;  // 总线程数
    int round;      // 当前轮次
};

void barrier_init(struct barrier *b, int n) {
    pthread_mutex_init(&b->mutex, NULL);
    pthread_cond_init(&b->cond, NULL);
    b->count = 0;
```

```
    b->threshold = n;
    b->round = 0;
}

void barrier(struct barrier *b) {
    pthread_mutex_lock(&b->mutex);

    int round = b->round;
    b->count++;

    if (b->count == b->threshold) {
        b->round++;
        b->count = 0;  // 准备下一个轮次
        pthread_cond_broadcast(&b->cond);  // 唤醒所有线程
    } else {
        while (round == b->round) {
            pthread_cond_wait(&b->cond, &b->mutex);  // 等待其他线程
        }
    }

    pthread_mutex_unlock(&b->mutex);
}
```

2. **重新编译并测试**：
   - 编译并运行程序，检查屏障是否在不同线程数量下工作正常。

```
make barrier
./barrier 1
./barrier 2
./barrier 3
```

3. **确保代码通过测试**：
   - 使用 `make grade` 命令运行所有测试，确保屏障的实现能够通过所有测试用例。

```
make grade
```

3) 实验中遇到的困难和解决办法

1. **处理线程竞争和同步**：
   - **困难描述**：需要确保线程在屏障点正确同步，避免数据竞争。
   - **解决办法**：使用互斥锁和条件变量来管理线程同步，确保所有线程在屏障点正确等待和通知。

2. **轮次管理**：
   - **困难描述**：处理多轮次屏障时，确保线程在正确的轮次同步。

- **解决办法**：在每次屏障到达时更新轮次计数，并确保线程在其轮次结束时被唤醒。

3. **调试和验证**：

- **困难描述**：调试多线程程序可能会遇到难以复现的错误。
- **解决办法**：使用调试工具（如 `gdb`）逐步检查线程状态和屏障实现，确保线程在屏障点正确等待和释放。

4) 实验心得

通过本次实验，我深入理解了线程同步机制和条件变量的使用。实现线程屏障让我掌握了如何管理多个线程在特定点的同步，避免了数据竞争和状态不一致的问题。通过对条件变量和互斥锁的实际应用，我提高了在多线程编程中的同步和协调能力。这些技能在实际开发中对于构建稳定和高效的并发程序至关重要。

Buffer cache (hard)

1) 实验目的

本实验的目标是优化 xv6 操作系统中的缓冲区缓存管理，通过减少锁竞争来提高性能。具体来说，你需要实现一个更细粒度的锁机制，用于管理缓冲区缓存，从而减少对 `bcache.lock` 的争用。

2) 实验步骤

1. 理解现有代码

1. **查看 `bio.c` 代码**：

- 定位 `bcache` 结构体以及 `bget` 和 `brelse` 函数。
- 理解当前 `bcache.lock` 的使用方式及其对性能的影响。

2. **分析性能问题**：

- 运行 `bcachetest`，查看 `bcache.lock` 的高竞争情况。
- 记录锁竞争统计信息，识别主要的瓶颈。

2. 实现高级锁机制

1. **定义桶锁**：

- 实现一个哈希表来管理缓冲区缓存，每个桶使用一个锁。使用质数作为桶的数量（例如：13）来减少哈希冲突。

```
#define NUM_BUCKETS 13

struct bucket_lock {
    pthread_mutex_t lock;
};

struct bucket_lock bucket_locks[NUM_BUCKETS];
```

- 在 `bcache_init()` 函数中初始化这些锁。

```
void bcache_init() {
    for (int i = 0; i < NUM_BUCKETS; i++) {
        pthread_mutex_init(&bucket_locks[i].lock, NULL);
    }
    // 其他初始化代码...
}
```

2. **修改 `bget` 和 `brelse` 函数**:

- 更新 `bget` 函数，使用哈希表来查找适当的桶锁。

```
struct buf* bget(uint dev, uint blockno) {
    struct buf* b;
    int bucket_index = blockno % NUM_BUCKETS;

    pthread_mutex_lock(&bucket_locks[bucket_index].lock);

    // 其他代码逻辑...

    pthread_mutex_unlock(&bucket_locks[bucket_index].lock);
    return b;
}
```

- 在 `brelse` 函数中，更新对桶锁的使用，确保对缓存块的释放不会引发锁竞争。

```
void brelse(struct buf* b) {
    int bucket_index = b->blockno % NUM_BUCKETS;

    pthread_mutex_lock(&bucket_locks[bucket_index].lock);

    // 其他代码逻辑...

    pthread_mutex_unlock(&bucket_locks[bucket_index].lock);
}
```

3. **重新编译并测试**:

- 编译并运行程序，检查 `bcachetest` 是否通过，并确认锁竞争显著减少。

```
make bcache
./bcachetest
```

4. **确保代码通过所有测试**:

- 使用 `make grade` 命令运行所有测试，确保优化后的实现能够通过所有测试用例。

```
make grade
```

3) 实验中遇到的困难和解决办法

1. **处理锁竞争和同步**：
   - **困难描述**：需要确保不同线程对缓冲区块的访问不会引发锁竞争。
   - **解决办法**：使用哈希表和桶锁来管理不同缓冲区块的访问，减少全局锁的竞争。
2. **处理哈希冲突**：
   - **困难描述**：缓冲区块可能哈希到相同的桶中，引发锁冲突。
   - **解决办法**：调整桶的数量和哈希算法，确保哈希冲突尽可能减少。
3. **调试和验证**：
   - **困难描述**：调试多线程程序可能会遇到难以复现的错误。
   - **解决办法**：使用调试工具（如 `gdb`）逐步检查线程状态和锁的实现，确保线程在访问缓冲区块时正确同步。

4) 实验心得

通过本次实验，我深入理解了缓冲区缓存管理和锁机制的优化。实现细粒度锁机制让我掌握了如何有效地减少多线程环境中的锁竞争，提高系统的性能和稳定性。通过对哈希表和桶锁的实际应用，我提高了在复杂并发环境中的编程能力，这些技能在高性能系统的开发中非常重要。