Lab07: Multithreading

Uthread: Switching Between Threads (Moderate)

1) 实验目的

本实验的目的是设计和实现用户级线程系统中的上下文切换机制。通过完成以下任务，我们将实现一个简单的线程调度系统，支持线程的创建、切换和退出：

1. **在 `uthread.c` 中实现线程创建和调度功能。**
2. **在 `uthread_switch.S` 中实现线程上下文切换功能。**

2) 实验步骤

1. 设计线程切换机制

上下文切换：切换线程时需要保存当前线程的上下文（寄存器状态）并恢复新线程的上下文。因为只有调用 `thread_switch` 时需要保存和恢复寄存器的状态，所以只需处理调用者保存的寄存器（如 `ra`, `s0-s11`）即可。

线程创建：在创建新线程时，需要为线程分配栈空间并将线程函数的入口地址设置为该线程的栈顶。

2. 实现 `thread_create()`

- **在 `user/uthread.c` 中实现线程创建**：

```
#include "uthread.h"
#include "types.h"
#include "riscv.h"
#include "defs.h"

void thread_create(struct thread *t, void (*start_func)(), void *arg) {
    // Allocate stack for the new thread
    t->stack = (char *)malloc(PGSIZE);
    if (t->stack == 0) {
        panic("thread_create: out of memory");
    }

    // Initialize stack and setup thread's stack frame
    t->stack += PGSIZE; // Move stack pointer to the top of the stack
    *(--t->stack) = (char)arg; // Argument to the start function
    *(--t->stack) = (char)start_func; // Return address to start function
    t->stack -= sizeof(struct trapframe) / sizeof(char); // Allocate space
for trapframe

    // Setup initial context for the new thread
    t->tf = (struct trapframe *)t->stack;
    t->tf->epc = (uint64)start_func; // Entry point of the thread
```

```
    t->tf->sp = (uint64)t->stack; // Stack pointer
}
```

3. 实现 `thread_schedule()`

- **在 `user/uthread.c` 中实现线程调度**：

```
void thread_schedule() {
    struct thread *next_thread = pick_next_thread(); // Function to pick the
next thread
    if (next_thread == 0) {
        printf("thread_schedule: no runnable threads\n");
        return;
    }

    struct thread *current_thread = current_thread(); // Function to get
current thread
    thread_switch(current_thread, next_thread);
}
```

4. 实现 `thread_switch`（汇编代码）

- **在 `user/uthread_switch.S` 中实现线程上下文切换**：

5. 测试和验证

1. **编译和运行 `uthread` 测试程序**：

```
make qemu
```

2. **检查输出**：
    - 确保所有线程正确运行并按预期输出。
    - 验证线程的创建、调度和切换功能是否正常。

3) 实验中遇到的困难和解决办法

1. **上下文切换的实现**：
    - **困难描述**：正确保存和恢复寄存器状态以实现线程切换。
    - **解决办法**：确保在 `thread_switch` 汇编代码中正确保存和恢复寄存器，使用调试工具检查寄存器的值。
2. **线程调度的实现**：
    - **困难描述**：正确选择和调度线程。

- **解决办法**：实现线程选择机制，如基于时间片的轮转调度，确保 `pick_next_thread` 函数选择正确的线程。

3. **线程栈管理**：
   - **困难描述**：分配和管理线程栈空间。
   - **解决办法**：确保在 `thread_create` 中正确分配和初始化线程栈，避免栈溢出或非法访问。

4) 实验心得

通过本次实验，我深入理解了用户级线程的上下文切换机制和线程调度的实现。实现线程切换和调度让我对操作系统中的线程管理有了更深刻的认识。调试和验证线程的正确性是确保系统稳定运行的关键，掌握了这些技能对实际开发中的多线程系统有很大帮助。

Using Threads: Parallel Hash Table with Pthreads (Moderate)

1) 实验目的

本实验旨在通过多线程编程与同步锁机制，改进一个简单的哈希表，使其在多线程环境中能正确工作并获得良好的性能。实验包括以下几个目标：

1. **识别并修复哈希表在多线程环境下的错误。**
2. **通过添加锁机制来解决数据竞争问题。**
3. **优化锁机制以提高性能，允许并行操作。**

2) 实验步骤

1. 识别问题

1. **运行初始测试**：
   - 编译并运行程序 `ph` 来检查当前哈希表在单线程和多线程环境中的表现。

```
make ph
./ph 1
./ph 2
```

2. **分析多线程环境下的错误**：
   - 观察多线程运行时输出中的 "keys missing" 行，找出导致键丢失的原因。常见原因是数据竞争，多个线程同时对哈希表进行操作，可能会造成数据丢失或错误。

3. **提交问题分析**：
   - 记录可能导致键丢失的事件序列，并将其写入 `answers-thread.txt` 文件。描述如何在两个线程同时插入数据时，导致键丢失的具体情况。

2. 添加锁以修复哈希表

1. **在 `notxv6/ph.c` 中添加互斥锁**：

- **声明和初始化锁**：

```
pthread_mutex_t lock;

void init_hash_table() {
    pthread_mutex_init(&lock, NULL);
}
```

- **在 `put` 和 `get` 函数中添加锁**：

```
void put(int key, int value) {
    pthread_mutex_lock(&lock);
    // 插入数据的代码
    pthread_mutex_unlock(&lock);
}

int get(int key) {
    pthread_mutex_lock(&lock);
    // 获取数据的代码
    pthread_mutex_unlock(&lock);
}
```

2. **重新编译并测试**：

- 确保程序在添加锁后，能够正确处理多线程操作，并且 `ph_safe` 测试通过。

```
make ph
./ph 1
./ph 2
```

3. **优化性能**：

- **使用每个哈希桶的锁**：
  - **修改哈希表结构**：

```
#define BUCKET_COUNT 100
pthread_mutex_t locks[BUCKET_COUNT];

void init_hash_table() {
    for (int i = 0; i < BUCKET_COUNT; i++) {
        pthread_mutex_init(&locks[i], NULL);
    }
}
```

- **在 `put` 和 `get` 中使用桶级别的锁**：

```
void put(int key, int value) {
    int bucket = hash(key) % BUCKET_COUNT;
    pthread_mutex_lock(&locks[bucket]);
    // 插入数据的代码
    pthread_mutex_unlock(&locks[bucket]);
}

int get(int key) {
    int bucket = hash(key) % BUCKET_COUNT;
    pthread_mutex_lock(&locks[bucket]);
    // 获取数据的代码
    pthread_mutex_unlock(&locks[bucket]);
}
```

4. **重新编译并测试性能**：
   - 运行 `ph` 测试程序，确保在多线程环境中能够实现正确性和性能的双重优化：

```
make ph
./ph 1
./ph 2
```

3) 实验中遇到的困难和解决办法

1. **数据竞争问题**：
   - **困难描述**：在多线程操作中，哈希表的操作可能导致数据丢失。
   - **解决办法**：使用互斥锁（`pthread_mutex_t`）来确保线程安全，避免数据竞争。
2. **性能优化**：
   - **困难描述**：使用全局锁会降低并行性能。
   - **解决办法**：实现每个哈希桶的锁，从而允许多个线程同时对不同桶进行操作，提高性能。
3. **测试和验证**：
   - **困难描述**：确保程序在多线程环境中既能正确运行又能达到预期性能。
   - **解决办法**：通过多次测试和性能评估，确保程序在多线程环境中表现正常，检查是否有任何锁竞争或性能瓶颈。

4) 实验心得

通过本次实验，我深入了解了多线程编程中的数据竞争问题及其解决方案。实现互斥锁来保证线程安全，并通过优化锁机制提高性能，使我对多线程编程中的挑战和优化方法有了更深刻的认识。通过这次实验，我也提高了对线程安全和性能优化的理解，这在实际开发中非常重要。

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。