Lab 6 Multithreading

Uthread: switching between threads (moderate)

实验目的

本实验的目标是设计一个用户级别的上下文切换机制，实现类似于协程的线程切换，而非依赖内核进行调度。将在 xv6 操作系统中创建用户级线程（`uthreads`），并通过设计上下文切换来实现多线程切换。

实验步骤

1. 初始化线程管理：创建一个结构体 `struct thread` 来表示线程，其中包含线程的栈、状态和上下文信息。

```
struct thread {
    void *stack;
    enum { RUNNING, RUNNABLE, TERMINATED } status;
    ucontext_t context;
};
```

2. 初始化主线程：在 `main` 函数中，初始化一个主线程，将其状态设置为 `RUNNING` 并初始化其上下文。

```
int main() {
    // 初始化主线程
    struct thread main_thread;
    main_thread.stack = malloc(STACK_SIZE);
    main_thread.status = RUNNING;
    getcontext(&main_thread.context);
}
```

3. 创建线程：编写 `thread_create` 函数，用于创建新的线程。这涉及分配栈空间，初始化线程的上下文，并将线程状态设置为 `RUNNABLE`。

```
void thread_create(struct thread *t, void (*func)()) {
    t->stack = malloc(STACK_SIZE);
    t->status = RUNNABLE;
    getcontext(&t->context);
    t->context.uc_stack.ss_sp = t->stack;
    t->context.uc_stack.ss_size = STACK_SIZE;
    makecontext(&t->context, func, 0);
}
```

4. 线程调度：编写 `thread_schedule` 函数，用于实现简单的线程调度。该函数选择下一个要运行的线程，保存当前线程的上下文，并恢复下一个线程的上下文。

```
struct thread *current_thread;

void thread_schedule(struct thread *next_thread) {
    struct thread *prev_thread = current_thread;
    current_thread = next_thread;
    if (prev_thread != NULL) {
        swapcontext(&prev_thread->context, &next_thread->context);
    } else {
        setcontext(&next_thread->context);
    }
}
```

实验中遇到的问题和解决办法

- 问题：上下文切换后，线程状态未正确更新，导致线程重复执行。
- 解决办法：在上下文切换时，确保更新当前线程和下一个线程的状态。确保在进行下一次线程切换时，状态正确反映线程是否已执行或正在执行。

实验心得

通过本实验，深入理解了线程切换的基本原理。实现用户级线程切换不仅展示了操作系统的内部机制，还帮助更好地理解协程的概念。

Using threads (moderate)

实验目的

本实验的目标是使用 `POSIX` 线程库（`pthread`）进行多线程编程。学习如何使用互斥锁来保护共享数据，以及如何使用多线程实现并发任务。

实验步骤

1. 初始化互斥锁：在主程序中，初始化一个互斥锁（`pthread_mutex_t`）以保护共享数据。

```
pthread_mutex_t mutex;

int main() {
    // 初始化互斥锁
    pthread_mutex_init(&mutex, NULL);
}
```

2. 编写并发任务：创建多个线程来执行并发任务。使用互斥锁来保护共享数据，确保线程安全。

```
void *thread_func(void *arg) {
    // 获取互斥锁
    pthread_mutex_lock(&mutex);
```

```
    // 执行并发任务
    // ...

    // 释放互斥锁
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

3. 等待线程结束：在主程序中等待所有线程完成任务，以确保在继续执行之前所有线程都已完成。

```
int main() {
    // 创建多个线程
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, thread_func, NULL);
    pthread_create(&thread2, NULL, thread_func, NULL);

    // 等待线程结束
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // 销毁互斥锁
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

实验中遇到的问题和解决办法

1. 问题： 多个线程同时访问共享数据时出现竞争条件，导致数据不一致。

- 解决办法： 使用互斥锁来保护共享数据，确保一次只有一个线程可以访问数据，从而避免竞争条件。

2. 问题： 线程创建后没有被正确地等待和回收，导致主程序过早结束。

- 解决办法： 使用 `pthread_join` 函数等待线程完成任务，并确保在主程序退出之前回收所有线程。

实验心得

通过本实验，掌握了使用 `pthread` 库进行多线程编程的基本技巧。了解了如何使用互斥锁来保护共享数据，以及如何实现并发任务。

Barrier(moderate)

实验目的

本实验的目标是学习使用条件变量来实现线程同步。将创建一个屏障（`barrier`）机制，确保所有线程都在达到特定点之前等待，然后同时继续执行。

实验步骤

1. 初始化条件变量和互斥锁：在主程序中初始化条件变量和互斥锁，以实现线程的同步等待。

```
pthread_mutex_t mutex;
pthread_cond_t barrier_cond;
int thread_count = 0;
int total_threads = 2; // 假设有两个线程

int main() {
    // 初始化条件变量和互斥锁
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&barrier_cond, NULL);
}
```

2. 编写屏障机制：创建一个屏障，使所有线程在达到屏障点之前等待。使用条件变量来实现线程等待和唤醒。

```
void barrier() {
    pthread_mutex_lock(&mutex);
    thread_count++;
    if (thread_count < total_threads) {
        pthread_cond_wait(&barrier_cond, &mutex);
    } else {
        pthread_cond_broadcast(&barrier_cond);
    }
    pthread_mutex_unlock(&mutex);
}
```

3. 线程同步：编写多个线程，每个线程执行循环并在达到屏障点时等待。当所有线程都达到屏障点后，同时继续执行。

```
void *thread_func(void *arg) {
    // 执行一些操作

    // 等待屏障
    barrier();

    // 屏障后的操作
    // ...

    return NULL;
}
```

4. 主程序调用线程：在主程序中创建多个线程，并在每个线程中执行相应的任务。

```
int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, thread_func, NULL);
    pthread_create(&thread2, NULL, thread_func, NULL);
```

```
    // 等待线程结束
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // 销毁条件变量和互斥锁
    pthread_cond_destroy(&barrier_cond);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

实验中遇到的问题和解决办法

1. 问题： 某个线程在等待屏障时被意外唤醒，导致线程同步错误。

- 解决办法： 在等待屏障时，始终使用循环检查条件，以避免线程在不满足条件时被错误唤醒。

2. 问题： 在实验中使用的条件变量和互斥锁没有被正确地初始化，导致运行时错误。

- 解决办法： 确保在使用条件变量和互斥锁之前，对它们进行正确的初始化，以避免未定义的行为。

3. 问题： 如果线程数量发生变化，屏障的逻辑可能会失效。

- 解决办法： 动态地确定线程数量，确保屏障逻辑在不同数量的线程下仍然正确工作。

实验心得

通过本实验，学会了如何使用条件变量和互斥锁来实现线程的同步等待，从而实现线程间的协调和同步。这对于解决多线程并发执行时的同步问题非常有帮助。

通过解决问题，在每个实验中得以顺利完成并达到了预期的目标。这些问题和解决办法的经验对于理解多线程编程和操作系统内部机制非常有帮助。

评分

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (9.8s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/gyyos2023/桌面/xv6-labs-2021”
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录“/home/gyyos2023/桌面/xv6-labs-2021”
ph_safe: OK (33.0s)
== Test ph_fast == make[1]: 进入目录“/home/gyyos2023/桌面/xv6-labs-2021”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/gyyos2023/桌面/xv6-labs-2021”
ph_fast: OK (69.8s)
== Test barrier == make[1]: 进入目录“/home/gyyos2023/桌面/xv6-labs-2021”
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
```

```
make[1]: 离开目录“/home/gyyos2023/桌面/xv6-labs-2021”
barrier: OK (22.7s)
== Test time ==
time: OK
Score: 60/60
```