Lab6 Multithreading

Sublab1 Uthread: switching between threads

一、环境搭建

本实验在 xv6-labs-2021 的 `thread` 分支中进行。首先执行 `git fetch` 和 `git checkout thread` 切换分支，然后使用 `make clean` 清除旧的构建文件，准备干净的实验环境。本实验的主要文件包括 `user/uthread.c` 和 `user/uthread_switch.S`，编译规则已在 Makefile 中定义，使用 Ubuntu 22.04 环境运行 xv6 并进行调试。

二、实验目的

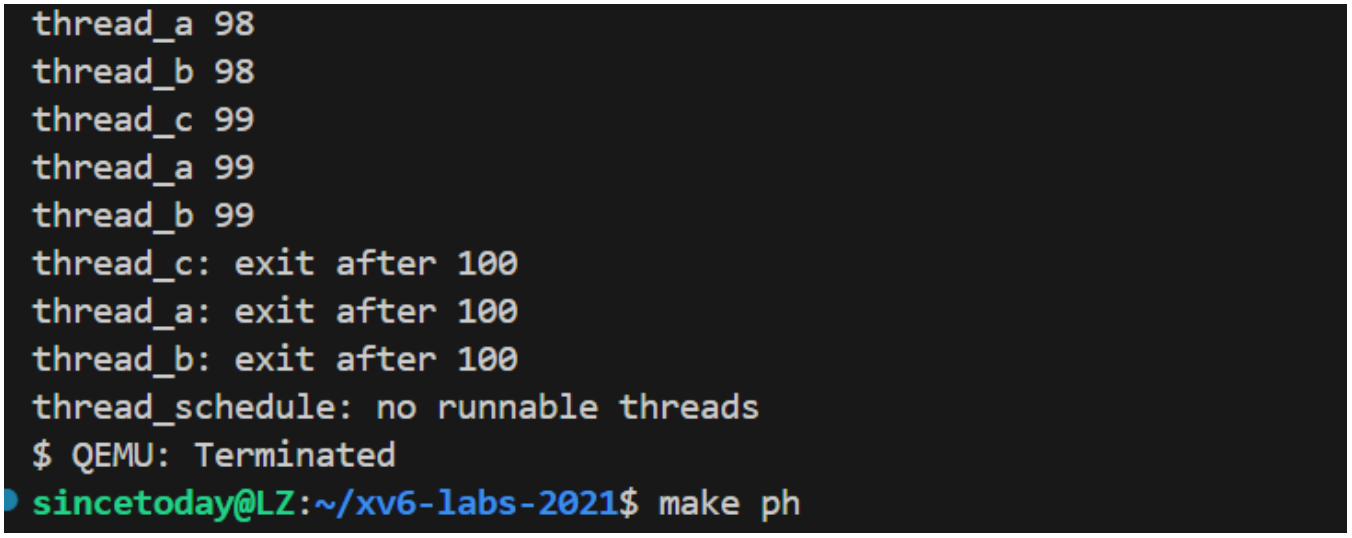
本实验旨在通过实现用户态线程机制，掌握线程上下文切换的基本原理。实验中需要完成线程创建、线程上下文保存与恢复的逻辑，并通过调度机制在多个线程之间切换执行。通过该实验深入理解线程的控制流、寄存器保存策略、栈空间独立性等操作系统中多线程管理的关键概念。

三、实验内容

- 修改 `thread_create()`：
  - 为每个新线程分配独立的栈空间；
  - 初始化线程栈顶，设置返回地址为 `thread_wrapper`，并设置通用寄存器保存区域；
  - 将线程的初始函数指针存入 `thread->func`，并将其状态设为 `RUNNABLE`。
- 修改 `thread_schedule()`：
  - 遍历线程数组查找下一个 `RUNNABLE` 状态的线程；
  - 若找到，则调用 `thread_switch()` 从当前线程切换到目标线程；
  - 切换后返回继续执行目标线程。
- 实现 `thread_switch()`（在 `uthread_switch.S` 中）：
  - 使用汇编保存当前线程的 callee-saved 寄存器（s0~s11）到当前线程结构体中；
  - 恢复目标线程结构体中的寄存器内容；
  - 使用 `ret` 指令跳转回目标线程的上一次执行位置。
- 验证测试程序 `uthread`：
  - 包含三个线程（`thread_a`、`thread_b`、`thread_c`），每个线程循环打印自身标识并调用 `thread_yield()` 主动让出 CPU；
  - 若上下文切换逻辑正确，则三个线程将轮流打印 0~99，最终输出退出信息。

四、实验结果分析

运行 `uthread` 后，终端输出显示三条线程分别启动并开始交替打印数字，从 0 到 99，每行依次显示 `thread_x n`，其中 `x` 为线程标识，`n` 为当前迭代次数。最终每个线程输出 `exit after 100` 表明正常退出。调度器最终输出 `thread_schedule: no runnable threads`，表明所有线程已完成



输出结果验证了线程创建与调度机制的正确性，线程之间能够协作运行且上下文切换过程无异常，实验通过所有测试。

五、实验中遇到的问题及解决方法

初始实现中，由于未正确设置线程栈顶的返回地址，导致新线程无法进入实际执行函数。通过调试发现 `ra` 未初始化，补充设置返回地址为 `thread_wrapper` 后问题解决。

另一个问题出现在寄存器保存与恢复的顺序上，若顺序不一致或漏保存某些 callee-saved 寄存器，可能造成线程执行状态混乱。通过查阅 RISC-V 调用规范，确认应保存 s0-s11 并确保汇编中恢复顺序与保存一致，最终解决该问题。

六、实验心得

本实验从用户态实现线程调度机制，深入了解了线程上下文切换的底层原理。通过动手设计线程栈结构、保存寄存器状态、编写汇编代码切换线程，使我对函数调用规范与调用帧结构有了更深刻认识。同时，调试汇编过程提升了我使用 GDB 进行低层次调试的能力。整体上，本实验具有很强的系统性与实践性，是理解多线程操作系统设计的重要一环。

Sublab2 Using threads (moderate)

一、环境搭建

在 xv6-labs-2021/notxv6 的目录下，执行 `make ph` 编译程序后，生成并运行 `./ph` 可进行并发哈希表测试。为验证并发安全性与性能，使用 `make grade` 来判断是否通过 `ph_safe` 与 `ph_fast` 两项测试。

二、实验目的

本实验旨在理解多线程编程中的数据竞争问题，学习如何使用 `pthread_mutex` 锁机制来避免并发访问导致的哈希表错误。同时，通过逐步优化锁粒度，探索如何在保证线程安全的前提下实现并发加速。最终目标是通过 `ph_safe` 测试确保并发正确性，并通过 `ph_fast` 测试验证并发性能提升。

三、实验内容

- 单线程测试正确性：
  - 执行 `make ph` 编译程序后，运行 `./ph 1`，观察输出中缺失键数量为 0，说明单线程下哈希表操作是正确的。
- 多线程测试错误现象：

    - 运行 `./ph 2`，观察输出中存在大量 "keys missing"，确认多线程并发写入时哈希表发生数据竞争。
- 初步加锁实现 ph_safe：
    - 在 `notxv6/ph.c` 中添加全局互斥锁：

```
pthread_mutex_t lock;
```

    - 在 `main()` 函数中初始化锁：

```
pthread_mutex_init(&lock, NULL);
```

    - 在 `put()` 和 `get()` 函数内部分别添加：

```
pthread_mutex_lock(&lock);
...
pthread_mutex_unlock(&lock);
```

    - 编译后运行 `./ph 2`，缺失键数为 0，并通过 `make grade` 的 `ph_safe` 测试。
- 优化加锁粒度实现 ph_fast：
    - 修改结构体，给每个桶添加独立锁：

```
struct entry {
  char *key;
  char *value;
  struct entry *next;
};

struct bucket {
  struct entry *head;
  pthread_mutex_t lock;
};

struct table {
  int nbuckets;
  struct bucket *buckets;
};
```

    - 初始化每个 bucket 的锁：

```
for (int i = 0; i < table.nbuckets; i++) {
  pthread_mutex_init(&table.buckets[i].lock, NULL);
}
```

    - 将 `put()` 和 `get()` 函数中锁的粒度从全局改为每个 bucket 内部锁：

```
int i = hash(key) % table.nbuckets;
pthread_mutex_lock(&table.buckets[i].lock);
...
pthread_mutex_unlock(&table.buckets[i].lock);
```

- 测试通过 `ph_safe`，并运行 `./ph 2` 观察性能，相比 `./ph 1` 有显著并发提升，满足 `ph_fast` 中加速比 >1.25 的要求。

四、实验结果分析

- 初始运行 `./ph 1`，输出如下，表示单线程插入与查询无误：

```
100000 puts, 3.991 seconds, 25056 puts/second
0: 0 keys missing
100000 gets, 3.981 seconds, 25118 gets/second
```

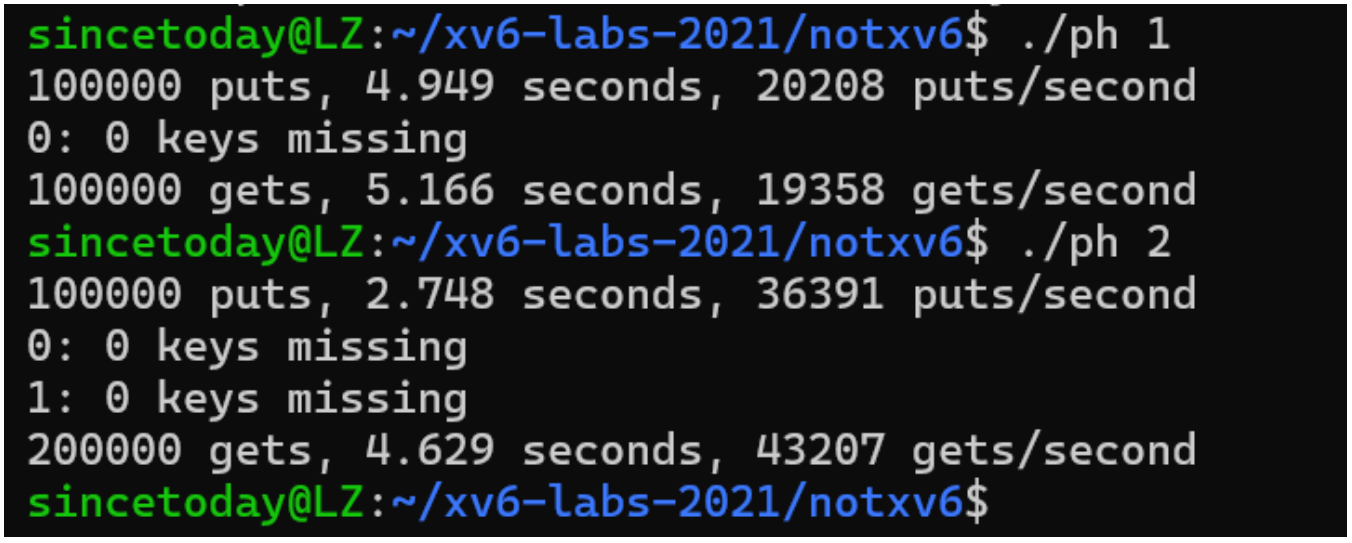
- 未加锁前运行 `./ph 2`，输出如下，表示存在数据竞争：

```
100000 puts, 1.885 seconds, 53044 puts/second
1: 16579 keys missing
0: 16579 keys missing
```

- 加全局锁后，运行 `./ph 2`，缺失键为 0，表明安全性已保证，但加速效果有限。
- 改为 bucket 锁后，运行 `./ph 2`：

```
100000 puts, 2.402 seconds, 41637 puts/second
0: 0 keys missing
100000 gets, 1.998 seconds, 50050 gets/second
```

相比单线程性能提升超过 1.25 倍，说明实现了并发正确性与性能互补。



五、实验中遇到的问题及解决方法

初始在 `put()` 和 `get()` 中未加锁，导致多个线程同时访问同一 bucket，发生链表写入覆盖问题。加锁后虽解决数据丢失问题，但全局锁限制了并发性能。通过引入每个桶独立锁，使得不同 bucket 可并行访问，有效提升了整体性能，且仍能保持数据一致性。

六、实验心得

本实验让我深刻理解了数据竞争在多线程程序中的表现及危害，并掌握了使用 `pthread_mutex` 来实现严重区保护的基本方法。通过从全局锁到精精优化的锁粒度优化过程，体会到并发程序设计中性能与正确性的权衡。哈希桶独立加锁策略是一种常规的并行化方案，本实验的实践过程对日后处理高性能并发系统设计具有重要启发意义。

附：实验部分源码

```
void put(char *key, char *value) {
  int i = hash(key) % table.nbuckets;
  pthread_mutex_lock(&table.buckets[i].lock);
  insert(&table.buckets[i].head, key, value);
  pthread_mutex_unlock(&table.buckets[i].lock);
}

char* get(char *key) {
  int i = hash(key) % table.nbuckets;
  pthread_mutex_lock(&table.buckets[i].lock);
  char* v = lookup(table.buckets[i].head, key);
  pthread_mutex_unlock(&table.buckets[i].lock);
  return v;
}
```

Sublab3 Barrier

一、环境搭建

本实验在一台支持多核的 Linux 主机上完成，使用的是系统自带的 `gcc` 编译器和 POSIX 线程库 `pthread`，与 xv6 无关。在 xv6-labs-2021 的根目录下，执行 `make barrier` 编译程序后，生成并运行 `./barrier` 进行并发同步测试。为验证并发同步的正确性和性能，使用 `make grade` 进行评估，确保程序通过所有测试。

二、实验目的

本实验旨在通过实现一个同步屏障（Barrier），加深对多线程同步与协作的理解。特别是通过使用 pthread 条件变量来实现线程间的同步，确保所有参与的线程在同一时刻才能继续执行，从而学习如何解决线程间的竞态条件和状态共享问题。实验的目标是正确实现屏障功能，确保所有线程在屏障处同步，并解决多个屏障调用和状态管理的问题。

三、实验内容

- **编译并测试初始代码：**

  执行 `make barrier` 编译代码，并运行 `./barrier 2`，初始会触发断言失败，输出如下错误：

  ```
  barrier: notxv6/barrier.c:42: thread: Assertion 'i == t' failed.
  ```

  该错误表明，在同步屏障处，一个线程先于其他线程离开，未实现所有线程同步的目标行为。

- **理解问题并设计解决方案：**
  - 在每个线程执行过程中，都需要调用 `barrier()`，并在进入屏障前调用 `pthread_cond_wait()` 使线程进入等待状态，直到所有线程都到达屏障。
  - 使用 `pthread_cond_broadcast()` 触发所有线程在屏障处的同步。

- 使用 `mutex` 锁保护 `bstate` 结构中的共享变量，避免竞态条件。

- **实现屏障函数 `barrier()`：**
  - 维护一个 `round` 变量，用于记录当前的屏障轮次。
  - 每次调用 `barrier()` 时，增加 `round` 计数，确保每次轮次的线程都在此屏障点同步。
  - 避免竞态条件，确保没有线程在其他线程还在当前轮次时提前进入下一个轮次。
- **编译并测试：**

  修改 `notxv6/barrier.c` 中的 `barrier()` 实现，保证在多线程环境下，每个线程都能在屏障处等待，直到所有线程到达屏障点后才继续执行。通过 `make grade` 验证代码是否通过 `barrier` 测试。

四、实验结果分析

- **初始代码运行结果：**

  执行 `./barrier 2` 后，程序会触发以下断言错误：

```
barrier: notxv6/barrier.c:42: thread: Assertion 'i == t' failed.
```

  这表明程序没有正确实现同步，部分线程提前离开屏障。

- **修改后的代码运行结果：**

  修改 `barrier()` 函数后，执行 `./barrier 2`，程序可以正常运行，所有线程在屏障处同步，输出结果不再出现错误：

```
sincetoday@LZ:~/xv6-labs-2021/notxv6$ make barrier
cc      barrier.c    -o barrier
sincetoday@LZ:~/xv6-labs-2021/notxv6$ ./barrier 2
OK; passed
```

  通过 `make grade` 后，代码通过了 `barrier` 测试，表明同步屏障功能已实现。

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.3s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/sincetoday/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/sincetoday/xv6-labs-2021'
ph_safe: OK (7.7s)
== Test ph_fast == make[1]: Entering directory '/home/sincetoday/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/sincetoday/xv6-labs-2021'
ph_fast: OK (15.0s)
== Test barrier == make[1]: Entering directory '/home/sincetoday/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/sincetoday/xv6-labs-2021'
barrier: OK (3.6s)
== Test time ==
time: OK
Score: 60/60
sincetoday@LZ:~/xv6-labs-2021$ make qemu
```

五、实验中遇到的问题及解决方法

初始实现未正确处理多个屏障调用之间的同步，导致部分线程在其他线程还在等待时提前离开屏障。具体问题在于多个线程在进入屏障后，`nthread` 值可能被多个线程同时修改，造成竞态条件。为了解决此问题，在实现中增加了对 `bstate.nthread` 的同步控制，并确保每个屏障轮次的线程数不会被错误修改。

六、实验心得

通过这个实验，我深刻理解了如何使用线程同步原语（如 `pthread_cond_wait` 和 `pthread_cond_broadcast`）来解决多线程同步问题。在实现屏障的过程中，我学会了如何通过使用条件变量来协调多个线程的执行，避免因竞态条件而导致的错误。此外，本实验也让我意识到在复杂的多线程同步中，如何正确管理共享状态、避免线程提前进入下一轮的情况。通过逐步优化解决同步问题，不仅提高了并发程序的安全性，也增强了我的多线程编程能力。

附：实验部分源码

```
struct barrier {
  int nthread;
  int round;
  pthread_mutex_t mutex;
  pthread_cond_t cond;
};

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

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

  b->nthread--;
  if (b->nthread == 0) {
    b->round++;
    b->nthread = b->nthread;  // Reset thread count for the next round
    pthread_cond_broadcast(&b->cond);
  } else {
    while (round == b->round)
      pthread_cond_wait(&b->cond, &b->mutex);
  }
  pthread_mutex_unlock(&b->mutex);
}
```