# 同济大学操作系统课程设计 - Xv6 - Lab: Multithreading

## 【实验目的】

多线程允许多个任务同时执行，充分利用多核处理器的能力，从而提高程序的并行性和执行效率。对于计算密集型任务，多线程可以显著减少处理时间。通过将复杂的任务分解为多个子任务并使用多线程同时处理，可以使得程序设计更具模块化，易于管理和维护。这种分工协作的方式也有助于提高程序的可靠性和可扩展性。在某些应用场景中，如服务器和网络应用，多线程技术被用来模拟并发用户请求的处理。通过多线程，服务器可以同时处理多个客户端的请求，提高系统的吞吐量和响应速度。

在本次练习中，需要设计一个用户级线程系统的上下文切换机制，并且实现它。为了入门， xv6 系统中包含了两个文件：`user/uthread.c` 和 `user/uthread_switch.S`，以及在 Makefile 中的一条用于构建 `uthread` 程序的规则。`uthread.c` 包含了大部分用户级线程包的代码，以及三个简单测试线程的代码。不过，该线程包中缺少一些用于创建线程和在线程之间切换的代码

## 【实验环境】

- **虚拟机：**VMare Workstation 17
- **操作系统：**Ubuntu-20.04.6
- **实验系统：**xv6-labs-2021

## 【实验内容】

要启动实验室，需要先切换到 thread分支：

```
$ git fetch
$ git checkout thread
$ make clean
```

### 1 Uthread: switching between threads

1. 在 `uthread.c` 中创建线程的数据结构

   在 `uthread.c` 文件中，我们首先需要定义一个数据结构来表示线程。每个线程由一个结构体表示，该结构体包含以下内容：

   - 一个字节数组用于作为线程的栈；
   - 一个整数变量用于表示线程的状态。
   - 我们可以定义如下的结构体：

```
#define STACK_SIZE 4096  // 每个线程的栈大小

struct thread {
    char stack[STACK_SIZE];     // 线程栈
    int state;                  // 线程状态
    struct context context;     // 线程的上下文信息
};

struct thread threads[MAX_THREADS];  // 线程数组
```

`context` 结构体用于保存寄存器的值，从而实现线程切换时能够正确恢复上下文。

2. 创建线程

在 `user/uthread.c` 文件中修改 `thread_create()` 函数，以创建新的线程并将其初始化。我们需要设置线程的栈顶，并将寄存器 `ra` 设置为线程函数的地址，以便在切换到该线程时执行该函数。以下是 `thread_create()` 的一个简化实现：

```
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;
  t->context.sp = (uint64)t->stack + STACK_SIZE;
  t->context.ra = (uint64)func;
}
```


```
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;
  t->context.sp = (uint64)t->stack + STACK_SIZE;
  t->context.ra = (uint64)func;
}
```


3. 实现上下文切换

在 `user/uthread_switch.S` 文件中实现 `thread_switch` 汇编函数，用于在线程之间切换。我们需要保存当前线程的上下文（寄存器状态），切换到下一个线程的上下文，并恢复其状态。以下是一个 `thread_switch` 的示例实现：

```
thread_switch:
    // 保存当前线程的寄存器状态
    sd ra, 0(a0)       // 保存返回地址
    sd sp, 8(a0)       // 保存栈指针
    // 这里省略其他寄存器保存

    // 加载下一个线程的寄存器状态
    ld ra, 0(a1)       // 恢复返回地址
    ld sp, 8(a1)       // 恢复栈指针
    // 这里省略其他寄存器恢复

    ret  // 跳转到下一个线程的地址
```

- `sd ra, 0(a0)`：将旧线程的 `ra` 寄存器的值存储到旧线程的栈中，偏移量为0。
- `sd sp, 8(a0)`：将旧线程的 `sp` 寄存器的值存储到旧线程的栈中，偏移量为8。这个操作将保存旧线程的栈指针，以便在恢复时知道旧线程的栈位置。
- `ld ra, 0(a1)`：将新线程的 `ra` 寄存器的值从新线程的栈中加载，偏移量为0。
- `ld sp, 8(a1)`：将新线程的 `sp` 寄存器的值从新线程的栈中加载，偏移量为8。这个操作将恢复新线程的栈指针，使其指向新线程的栈。
- `ret`：返回到 `ra` 寄存器保存的地址，实现线程切换。在执行 `ret` 指令后，控制权将从当前线程切换到新线程。

4. 调度线程并切换上下文

   在 `user/uthread.c` 文件中实现 `thread_schedule` 函数，用于调度线程并调用 `thread_switch` 实现线程的切换。我们需要根据当前线程的状态决定下一个运行的线程，并传递合适的参数给 `thread_switch`。以下是 `thread_schedule` 的一个简化实现：

```
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) {
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    thread_switch((uint64)&t->context, (uint64)&next_thread->context);
  } else
    next_thread = 0;
}
```

5. 运行 xv6：使用 `make qemu` 指令来编译并运行 xv6 系统。在编译通过后，启动 QEMU 虚拟机，等待 xv6 系统启动。

6. 测试 `uthread` 命令

   在 xv6 的命令行中输入 `uthread`，可以运行我们所编写的线程库，并验证其功能是否正确。该命令会启动多个线程，并进行简单的上下文切换操作，以测试我们的实现：

```
thread_a 92
thread_b 92
thread_c 93
thread_a 93
thread_b 93
thread_c 94
thread_a 94
thread_b 94
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
$
```

## 2 Using threads

本实验旨在通过使用 `POSIX` 线程库（`pthread`）来实现多线程编程，并在多线程环境下处理哈希表。通过该实验，您将学习如何使用线程库创建和管理线程，并了解在多线程环境下如何使用加锁技术保护共享资源。

1. 编译与运行

   首先，我们通过运行 `make ph` 来编译位于 `notxv6/ph.c` 中的程序。在完成编译后，运行命令 `./ph 1`，观察单线程环境下哈希表的行为。

   ```
   $ make ph
   $ ./ph 1
   ```

   在单线程环境下，程序将数据写入哈希表，并随后读取所有数据。可以看到，所有键值对都被成功地写入和读取，没有出现遗漏或数据丢失的情况。

```
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab6$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab6$ ./ph 1
100000 puts, 7.701 seconds, 12986 puts/second
0: 0 keys missing
100000 gets, 7.708 seconds, 12973 gets/second
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab6$
```

2. 多线程测试

   接下来，运行命令 `./ph 2`，测试多线程环境下的哈希表操作。

   ```
   $ ./ph 2
   ```

在多线程环境中，多个线程同时执行读写操作。测试结果显示，由于线程之间的竞争访问，部分键值对未能正确写入哈希表，导致出现“missing keys”的问题。尽管如此，由于并行化的效果，程序的总插入速率达到了每秒 26174 次，这约是单线程运行速度的两倍，表现出预期的并行加速效果。

3. 多线程问题分析

   在多线程环境下出现“missing keys”的问题，通常是由于竞争条件引起的。以下是可能导致问题的情景分析：

   当多个线程同时向哈希表插入键值对时，如果它们插入的键值对的哈希值不同，那么这些键值对会被成功插入到不同的哈希桶中。然而，当两个键的哈希值相同时，它们将被插入到相同的位置。此时，可能会发生竞争条件，其中一个键值对将另一个覆盖，导致插入失败，从而出现键的丢失。

4. 使用互斥锁解决多线程竞争问题

   为了解决在多线程环境下的竞争条件问题，我们可以为每个哈希桶添加一个互斥锁，以确保在任何时刻只有一个线程可以访问或修改特定的哈希桶。这样可以保证插入操作的原子性，从而避免键的丢失。

   首先，我们在程序中定义一个互斥锁，用于保护哈希表的访问。以下是初始化互斥锁的示例代码：

```
#include <pthread.h>

// 定义互斥锁
pthread_mutex_t lock;

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

    // 其余代码
}
```

   在进行哈希表插入操作时，我们需要确保插入操作是原子性的。因此，在 `insert` 函数的前后，我们使用互斥锁进行加锁和解锁：

```
void put(int key, int value) {

    // 加锁
    pthread_mutex_lock(&lock);

    // 执行插入操作
    ...
    // 解锁
    pthread_mutex_unlock(&lock);
}
```

   程序结束时，需要销毁互斥锁以释放系统资源：

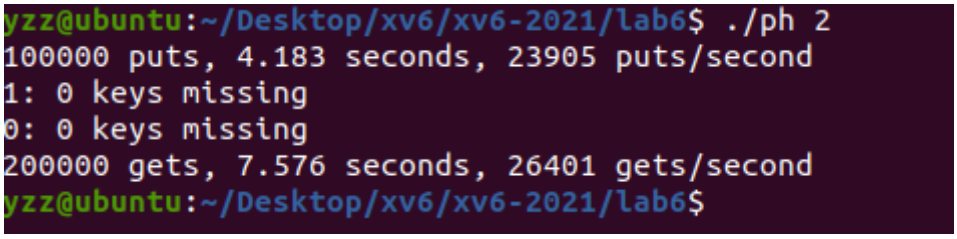
```
int main() {
    // 程序其他部分

    // 销毁互斥锁
    pthread_mutex_destroy(&lock);

    return 0;
}
```

5. 重新编译与测试

   完成加锁机制的实现后，再次运行 `make ph` 进行编译，然后运行 `./ph 2` 进行多线程测试。

```
$ make ph
$ ./ph 2
```

   经过修改后的程序，在多线程环境下能够正确地写入和读取哈希表中的所有键值对，“missing keys” 的问题已被完全解决。




## 3 Barrier

在此作业中实现一个barrier： 一个点应用程序，所有参与线程都必须等待，直到所有其他参与线程到达这一点也是如此。使用 pthread 条件变量，它是一个序列协调技术类似于 XV6 的 Sleep 和 Wakeup。

1. 了解 `struct barrier` 结构体

   在本实验中，我们将使用 `struct barrier` 结构体来实现多线程的屏障同步机制。`struct barrier` 是一个用于同步多个线程的结构体，其成员变量包括：

```
struct barrier {
    pthread_mutex_t barrier_mutex; // 用于保护对共享数据的访问
    pthread_cond_t barrier_cond;   // 用于管理线程的等待与唤醒
    int nthread;                   // 已经到达屏障的线程数
    int round;                     // 当前屏障轮次
} bstate;
```

   - `barrier_mutex`：互斥锁，用于保护对 `bstate` 结构体中共享数据的访问，确保线程之间的同步。
   - `barrier_cond`：条件变量，管理线程的等待与唤醒。当所有线程都到达屏障时，将通过条件变量唤醒所有等待中的线程。
   - `nthread`：表示当前已达到屏障的线程数量。
   - `round`：表示当前屏障的轮次，每次所有线程通过屏障后，轮次会加一。

   在 `barrier()` 函数中，通过比较 `bstate.nthread` 和全局变量 `nthread` 来判断是否应让线程等待或唤醒其他线程。`pthread_cond_wait()` 函数使线程释放 `mutex` 锁并进入等待状态，直到条件满足被唤醒。`pthread_cond_broadcast()` 用于唤醒所有等待条件的线程。

2. 在 `barrier()` 函数中添加同步逻辑

在 `barrier.c` 文件中的 `barrier()` 函数中，我们需要实现屏障机制。当某个线程到达屏障时，它会获取互斥锁 `barrier_mutex`，并修改 `nthread` 的值。只有当所有线程都到达屏障时，`nthread` 才会被清零，同时 `round` 加一，并唤醒所有等待中的线程。

以下是实现屏障同步逻辑的代码框架：

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

  ++ bstate.nthread;
  if (bstate.nthread != nthread) {
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  } else {
    ++ bstate.round;
    bstate.nthread = 0;
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
```

在这个实现中：

- 当一个线程到达屏障时，`bstate.nthread` 会加一，记录当前到达屏障的线程数量。
- 如果当前线程不是最后一个到达的线程，则调用 `pthread_cond_wait()` 使其等待在条件变量 `barrier_cond` 上，同时释放互斥锁 `barrier_mutex`，以允许其他线程进入屏障。
- 当最后一个线程到达屏障时，`bstate.nthread` 等于 `nthread`，这意味着所有线程都已到达屏障。此时，`bstate.nthread` 被重置为 0，并且 `bstate.round` 加一，表示进入了新的轮次。接着，使用 `pthread_cond_broadcast()` 唤醒所有在 `barrier_cond` 上等待的线程。

3. 编译与测试

在完成 `barrier()` 函数的逻辑后，我们需要编译并测试该实现。首先，使用 `make barrier` 命令编译 `notxv6/barrier.c` 文件：

```
$ make barrier
```

编译成功后，运行 `./barrier 2` 进行测试：

```
$ ./barrier 2
```

此命令将启动两个线程，并在它们之间执行屏障同步操作。屏障确保两个线程在每一轮都等待对方完成工作，从而同步它们的执行过程。

通过这个测试，我们可以验证屏障机制的正确性，确保所有线程在每个屏障点都能正确同步。所有线程在屏障处等待，直到最后一个线程到达时，才会继续执行下一轮操作。

```
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab6$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab6$ ./barrier 2
OK; passed
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab6$
```

# 【分析讨论】

在本次实验中，我通过多个具体任务深入学习和实践了多线程编程中的关键概念和技术，包括线程管理、线程同步以及在多线程环境中保障数据一致性的方法。这些任务不仅让我更好地理解了理论知识，还培养了我在实际编程中解决复杂问题的能力。

1. **多线程与上下文切换:**

在第一个任务中，我实现了一个简单的用户级线程库，包括线程的创建、上下文切换以及调度机制。通过这个任务，我掌握了如何在应用程序级别模拟线程的创建和调度，并且理解了上下文切换的实现细节。这个过程让我深刻认识到，线程的管理和调度不仅仅是操作系统的职责，也可以通过合理的编程手段在用户空间中实现。更重要的是，这一任务让我意识到，线程的创建和切换需要消耗资源，因此在实际应用中需要慎重设计以避免不必要的性能开销。

2. **多线程中的竞争问题与解决:**

   在第二个任务中，我深入探讨了多线程环境下的哈希表操作。通过对缺少键问题的分析，我了解了多线程编程中的竞争条件问题，以及这些问题如何导致数据不一致。通过为每个哈希桶引入互斥锁，我成功解决了竞争条件导致的数据丢失问题。这一任务让我深刻体会到，在共享资源的多线程操作中，同步机制是至关重要的。互斥锁的引入虽然解决了竞争问题，但也带来了潜在的性能瓶颈。因此，在实际开发中，需要在确保数据一致性与程序性能之间找到一个平衡点。

3. **屏障同步机制的实现:**

   在第三个任务中，我实现了一个多线程的屏障同步机制。通过使用互斥锁和条件变量，我确保了所有线程在屏障处能够正确地同步。这一过程让我深入理解了条件变量在线程同步中的重要作用，并且学习了如何通过条件变量来协调多个线程的执行顺序。这一任务让我意识到，在多线程编程中，合理的同步机制不仅可以避免数据竞争，还能确保线程之间的协调工作，从而提升整个程序的稳定性和可靠性。

通过本次实验，我在多线程编程方面取得了显著的进步。从线程的创建与上下文切换，到多线程环境中的数据竞争，再到线程的同步与屏障机制，每个环节都让我更加深刻地理解了多线程编程的复杂性与挑战性。通过这些实践，我不仅掌握了基本的多线程编程技能，也学会了如何在复杂的并发环境中设计高效且安全的程序。

# 【实验验证】

新建`time.txt`，输入自己做实验的用时，运行`make grade`进行评分：

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (4.2s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/yzz/Desktop/xv6/xv6-2021/
lab6'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/yzz/Desktop/xv6/xv6-2021/lab6'
ph_safe: OK (11.4s)
== Test ph_fast == make[1]: Entering directory '/home/yzz/Desktop/xv6/xv6-2021/
lab6'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/yzz/Desktop/xv6/xv6-2021/lab6'
ph_fast: OK (25.6s)
== Test barrier == make[1]: Entering directory '/home/yzz/Desktop/xv6/xv6-2021/
lab6'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/yzz/Desktop/xv6/xv6-2021/lab6'
barrier: OK (10.3s)
== Test time ==
time: OK
Score: 60/60
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab6$
```