同济大学操作系统课程设计——Lab6: Multi-threading

2151422武芷朵 Tongji University, 2024 Summer

Lab6: Multi-threading：多线程实验

项目地址：wzd232604/TJOS-xv6-2024-labs: 同济大学操作系统课程设计-xv6实验 (github.com)

综述

- 这个 lab 主要由三部分组成：
  - 实现一个用户级线程的创建和切换
  - 使用 UNIX pthread 线程库实现一个线程安全的 Hash 表
  - 利用 UNIX 的锁和条件变量实现一个 barrier

切换到 `thread` 分支：

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

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ git fetch
wzd@ubuntu:~/Desktop/xv6-labs-2021$ git checkout -f thread
Branch 'thread' set up to track remote branch 'thread' from 'origin'.
Switched to a new branch 'thread'
wzd@ubuntu:~/Desktop/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
        user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln us
er/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie  user/_uthread \
ph barrier
```

1. Uthread: switching between threads (moderate)

1.1 实验目的

用户态进程：设计并实现一个用户级线程系统的上下文切换机制，实现类似于协程的线程切换，而非依赖内核进行调度。需要创建线程、保存/恢复寄存器以在线程之间切换，并且确保解决方案通过测试。

1.2 实验步骤

需要在user/uthread.c 中实现thread_create() 和thread_schedule()，并且在 user/uthread_switch.S 中实现thread_switch 用于切换上下文。

1. 在`uthread.c`中创建适当的数据结构来表示线程。每个线程被定义成一个结构体，一个字节数组用作线程的栈，一个整数用于表示线程的状态：

user > C uthread.c > all_thread

```
12
13  //存储需要保存的内容，即进程运行时的寄存器
14  struct context {
15    uint64 ra;
16    uint64 sp;
17
18    // callee-saved
19    uint64 s0;
20    uint64 s1;
21    uint64 s2;
22    uint64 s3;
23    uint64 s4;
24    uint64 s5;
25    uint64 s6;
26    uint64 s7;
27    uint64 s8;
28    uint64 s9;
29    uint64 s10;
30    uint64 s11;
31  };
32
33
34  struct thread {
35    char       stack[STACK_SIZE]; /* the thread's stack */
36    int        state;             /* FREE, RUNNING, RUNNABLE */
37    struct context context;
38  };
```

2. 创建线程:在 `user/uthread.c` 中修改 `thread_create()`，将 ra 设为我们所要执行的线程的函数地址（`thread_switch` 在保存第一个进程的上下文后会加载第二个进程的上下文， 然后跳至刚刚加载的ra 地址处开始执行）：

```
user > C uthread.c > thread_a(void)
88  }
89
90  void
91  thread_create(void (*func)())
92  {
93    struct thread *t;
94
95    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
96      if (t->state == FREE) break;
97    }
98    t->state = RUNNABLE;
99    // YOUR CODE HERE
100   // 用自己的栈，sp 指向栈顶，向栈底生长
101   t->context.sp = (uint64)t->stack + STACK_SIZE;
102   // 该线程直接返回自己要执行的那个函数的入口地址
103   t->context.ra = (uint64)func;
104 }
105
```

3. 上下文切换: 在`user/uthread_switch.S`文件中的`thread_switch`实现线程切换的汇编代码。函数中按照struct context 各项在内存中的位置，需要实现保存调用者保存的寄存器，切换到下一个线程，然后恢复下一个线程的寄存器状态。在实现上，可以仿照`kernel/trampoline.S`的写法。

user > uthread_switch.S

```
9   thread_switch:
10      /* YOUR CODE HERE */
11      sd ra, 0(a0)
12      sd sp, 8(a0)
13      sd s0, 16(a0)
14      sd s1, 24(a0)
15      sd s2, 32(a0)
16      sd s3, 40(a0)
17      sd s4, 48(a0)
18      sd s5, 56(a0)
19      sd s6, 64(a0)
20      sd s7, 72(a0)
21      sd s8, 80(a0)
22      sd s9, 88(a0)
23      sd s10, 96(a0)
24      sd s11, 104(a0)
25
26      ld ra, 0(a1)
27      ld sp, 8(a1)
28      ld s0, 16(a1)
29      ld s1, 24(a1)
30      ld s2, 32(a1)
31      ld s3, 40(a1)
32      ld s4, 48(a1)
33      ld s5, 56(a1)
34      ld s6, 64(a1)
35      ld s7, 72(a1)
36      ld s8, 80(a1)
37      ld s9, 88(a1)
38      ld s10, 96(a1)
39      ld s11, 104(a1)
40      ret    /* return to ra */
41
```

4. 调度线程,切换上下文: 在 `user/uthread.c` 中添加 `thread_schedule` 函数，调用 `thread_switch` 来实现线程的切换。需要传递一些参数给 `thread_switch`，以便它知道要切换到哪个线程：

user > uthread.c > thread_schedule(void)

```
56  thread_schedule(void)
71    }
72    // 没有可运行的线程
73    if (next_thread == 0) {
74      printf("thread_schedule: no runnable threads\n");
75      exit(-1);
76    }
77    // 存在可运行的线程，且不是自己
78    if (current_thread != next_thread) {         /* switch threads?  */
79      next_thread->state = RUNNING;
80      t = current_thread;
81      current_thread = next_thread;
82      /* YOUR CODE HERE
83       * Invoke thread_switch to switch from t to next_thread:
84       * thread_switch(??, ??);
85       */
86      thread_switch((uint64)&t->context, (uint64)&next_thread->context);
87    } else
```

5. 利用make qemu指令运行xv6：
6. 在命令行中输入`uthread`：

```
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

7. 在终端里运行 `./grade-lab-thread uthread` 可进行评分：

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./grade-lab-thread uthread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (2.2s)
```

1.3 实验中遇到的问题和解决办法

1. 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。

2. 问题： 在创建线程时，需要如何为每个线程分配独立的堆栈空间。

- 解决办法：利用寄存器各自的功能特性，比如在 RISC-V 架构中，寄存器 ra 是返回地址寄存器（Return Address Register），它存储了函数调用后的返回地址。而寄存器 sp 则是栈指针寄存器（Stack Pointer Register），它存储了当前栈的顶部地址。因此struct结构体将这两个寄存器包含其中，并在创建线程的时候对其进行的分配。

1.4 实验心得

- 通过这次实验，更深入地理解了线程的概念以及线程切换的底层实现机制，充分利用寄存器的特性和调试工具。
- 通过理解 RISC-V 架构中寄存器的功能特性，能够选择适当的寄存器来存储必要的信息，比如函数指针和栈顶指针。当线程被调度执行时，它能够正确跳转到函数起始位置，并且在自己的独立栈上执行，避免与其他线程的干扰。

2. Using threads (moderate)

2.1 实验目的

本实验旨在通过使用使用 `POSIX` 线程库（`pthread`）实现多线程编程，以及在多线程环境下处理哈希表。学习如何使用线程库创建和管理线程，以及如何通过加锁来实现一个线程安全的哈希表，使用锁来保护共享资源，以确保多线程环境下的正确性和性能。

2.2 实验步骤

1. 运行 `make ph` 编译 `notxv6/ph.c`，运行 `./ph 1`，

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./ph 1
100000 puts, 8.336 seconds, 11997 puts/second
0: 0 keys missing
100000 gets, 8.352 seconds, 11973 gets/second
```

写入哈希表的数据被完整地读出，没有遗漏

2. 运行 `./ph 2`

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./ph 2
100000 puts, 3.821 seconds, 26174 puts/second
1: 16474 keys missing
0: 16474 keys missing
200000 gets, 9.647 seconds, 20732 gets/second
```

在多线程同时读写的情况下，部分数据由于竞争访问，出现了缺少的键（missing keys）问题，即一些数据没有被正确写入到哈希表中。

当两个线程同时向哈希表中添加条目时，它们的总插入速率为每秒 26174次。这大约是运行 ph 1 的单线程速度的两倍。约 2 倍的出色 "并行加速" 是我们所期望的（即两倍的内核在单位时间内产生两倍的工作量）。

3. 分析多线程问题：在多线程环境中，出现缺少的键问题可能是由于竞争条件引起的。尝试分析多线程情况下的序列事件，找出可能导致缺少键的情况。

notxv6 > C ph.c > insert(int, int, entry **, entry *)

```
27  }
28
29  static void
30  insert(int key, int value, struct entry **p, struct entry *n)
31  {
32    struct entry *e = malloc(sizeof(struct entry));
33    e->key = key;
34    e->value = value;
35    e->next = n;
36    *p = e;
37  }
38
```

同时有多个线程工作，所以当一个进程往哈希表中填入一个键时，可能有其他进程也希望填入一个键。如果两个键的哈希值不相等，那么插入是没有问题的，两个键都可以成功在不同的哈希筒中出现。但是如果两个键哈希值相同，那么其插入的位置一样，就会导致其中一个（相对后写入的）将另一个新插入的键覆盖，从而导致了键的丢失。

4. 加锁（对于每个哈希筒，每次只允许一个进程访问），保护共享资源，以防止多线程竞争引起的问题：

定义一个保护哈希表的互斥锁：

notxv6 > C ph.c > ...

```
19
20  // 每个桶一个锁
21  pthread_mutex_t lock[NBUCKET];
22
```

在main函数使用put调用insert之前，使用`pthread_mutex_init`来初始化锁：

notxv6 > C ph.c > main(int, char * [])

```
113  main(int argc, char *argv[])
131  }
132
133  // 初始化锁
134  Init();
```

```
notxv6 > C ph.c > put(int, int)
22
23
24  // 初始化锁
25  void
26  Init()
27  {
28      for (int i = 0; i < NBUCKET; ++ i)
29          pthread_mutex_init(&lock[i], NULL);
30  }
31
```

在使用insert前后，使用lock上锁开锁，保证insert操作的原子性：

```
notxv6 > C ph.c > get(int)
51  void put(int key, int value)
52  {
53    int i = key % NBUCKET;
54    pthread_mutex_lock(&lock[i]);       /* held the mutex */
55    // is the key already present?
56    struct entry *e = 0;
57    for (e = table[i]; e != 0; e = e->next) {
58      if (e->key == key)
59        break;
60    }
61    if(e){
62      // update the existing key.
63      e->value = value;
64    } else {
65      // the new is new.
66      insert(key, value, &table[i], table[i]);
67    }
68    pthread_mutex_unlock(&lock[i]);    /* release mutex */
69
70  }
71
```

使用完毕之后，除了释放锁，还需要销毁，防止占用资源：

```
notxv6 > C ph.c > main(int, char * [])
113  main(int argc, char *argv[])
165
166    for (int i = 0; i < NBUCKET; ++ i)
167        pthread_mutex_destroy(&lock[i]);
168  }
```

5. 再使用`make ph`编译`notxv6/ph.c`，运行`./ph 2`

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./ph 1
100000 puts, 9.115 seconds, 10971 puts/second
0: 0 keys missing
100000 gets, 8.297 seconds, 12053 gets/second
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./ph 2
100000 puts, 6.518 seconds, 15342 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 7.617 seconds, 26258 gets/second
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

重新测试之后可以发现，keys missing在单线程和多线程下都为0。

2.3 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： 错误地使用锁可能会导致死锁，即所有线程都在等待锁，无法继续执行。

- 解决办法： 确保在使用锁时遵循正确的获取和释放顺序，以避免死锁情况的发生。同时，在最后尽管已经释放了锁，但是创建了锁会占用一定的资源和内存，这影响了程序的性能，因此我又添加上了`pthread_mutex_destroy`来销毁锁，从而避免在程序中反复创建锁而没有销毁它们，导致最终耗尽系统的资源，使得其他程序或操作受到影响。

2.4 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。
- 使用锁可以确保在访问共享资源时的线程安全性，防止竞争条件和数据不一致的问题。我学会了如何使用 `pthread_mutex_t` 类型的锁，并且确保在使用锁时始终遵循正确的获取和释放顺序，以避免死锁情况的发生。此外，还了解到在锁的创建和销毁方面需要注意性能问题，因为未销毁的锁可能会影响系统的资源和性能。

3 Barrier(moderate)

3.1 实验目的

本实验旨在通过使用条件变量实现一个线程屏障（barrier），即每个线程都要在 barrier 处等待，直到所有线程到达 barrier 之后才能继续运行，加深对多线程编程中同步和互斥机制的理解。在多线程应用中，线程屏障可以用来确保多个线程在达到某一点后都等待，直到所有其他参与的线程也达到该点。通过使用pthread条件变量，我们将学习如何实现线程屏障，解决竞争条件和同步问题。

3.2 实验步骤

1. 熟悉`struct barrier`结构体的成员变量。

```
notxv6 > C barrier.c > main(int, char * [])
 9
10  struct barrier {
11    pthread_mutex_t barrier_mutex;
12    pthread_cond_t barrier_cond;
13    int nthread;      // Number of threads that have reached this round of the barrier
14    int round;     // Barrier round
15  } bstate;
```

通过比较 `bstate.nthread` 与全局变量 `nthread`，可以分别判断是否使进程等待或者唤醒其他进程。

库函数 pthread_cond_wait(&cond, &mutex); 使进程释放mutex 锁并进入睡眠，等待 cond 将其唤醒。pthread_cond_broadcast(&cond); 将所有等待 cond 的进程唤醒。

2. 在`barrier.c`中的`barrier()`函数中添加逻辑。在某个线程到达 barrier() 时，需要获取互斥锁进而修改 nthread。当 nthread 与预定的值相等时，将 nthread 清零，轮数加一，并唤醒所有等待中的线程。

使用`pthread_cond_wait()`来等待条件满足，`pthread_cond_broadcast()`来唤醒等待的线程。

```
notxv6 > C barrier.c > barrier()
28  barrier()
34  //
35  // 上锁
36  pthread_mutex_lock(&bstate.barrier_mutex);
37
38  // 来了一个线程，累计 ++
39  ++ bstate.nthread;
40
41  // 查看条件变量
42  if (bstate.nthread != nthread) {
43      // 放弃锁并且睡眠，等待条件变量来唤醒
44      pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
45  } else {
46      // 增加 barrier 里的计数（只计一次）
47      ++ bstate.round;
48      // 清零
49      bstate.nthread = 0;
50      // 当最后一个线程到达，直接广播给其他的线程
51      pthread_cond_broadcast(&bstate.barrier_cond);
52  }
```

- 在这个函数中，`bstate.nthread`表示当前已经达到屏障的线程数量。当一个线程进入屏障时，会将这个计数值加一，以记录达到屏障的线程数量。
- 如果还有线程未达到屏障，这个线程就会调用`pthread_cond_wait()`来等待在条件变量`barrier_cond`上。在调用这个函数之前，线程会释放之前获取的锁`barrier_mutex`，以允许其他线程在这个锁上等待。
- 当最后一个线程到达屏障，`bstate.nthread`的值会等于`nthread`，就会进入`else`分支。在这里，首先重置`bstate.nthread`为 0，以便下一轮的屏障计数。然后，增加`bstate.round`表示进入了下一个屏障的轮次。最后，通过调用`pthread_cond_broadcast()`向所有在条件变量上等待的线程发出信号，表示可以继续执行。

3. 保存后在终端里执行`make barrier`编译`notxv6/barrier.c`，运行`./barrier 2`：

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ make barrier
make: 'barrier' is up to date.
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./barrier 2
OK; passed
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

3.3 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：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。

3. pthread_cond_wait与pthread_cond_broadcast的配合使用：

- 解决办法：当一个线程需要等待某个条件满足时，它可以调用pthread_cond_wait函数，这会导致线程进入等待状态，同时释放它持有的互斥锁。线程在等待期间会一直阻塞，直到其他线程调用pthread_cond_broadcast来唤醒它。

所有线程在某个共享资源或条件未满足的情况下，都调用pthread_cond_wait来等待。这个等待会自动释放互斥锁，让其他线程可以访问共享资源。

当某个线程改变了共享资源或条件，使得其他线程可以继续执行时，它调用pthread_cond_broadcast来通知其他线程。这些线程会被唤醒，重新尝试获取互斥锁，然后继续执行。

3.4 实验心得

- 通过本次实验，我深入了解了多线程编程中的同步机制，特别是条件变量和互斥锁的使用。我学会了如何设计和实现屏障同步，以保证多个线程在特定点同步等待和唤醒，从而实现了程序的并发控制。

4 实验检验得分

1. 在实验目录下创建创建`answers-thread.txt`，填入1的测试结果
2. 在实验目录下创建`time.txt`，填写完成实验时间数
3. 在终端中执行`make grade`