Lab6 Multithreading

2351882 王小萌

Tongji University,2025 Summer

代码仓库链接：https://github.com/wxmxmw/Tongji-University-xv6-labs-2021.git

1 Uthread: switching between threads

1.1 实验目的

设计并实现一个用户级线程系统的上下文切换机制。补充完成一个用户级线程的创建和切换上下文的代码。需要创建线程、保存/恢复寄存器以在线程之间切换。

1.2 实验步骤

1. user/uthread.c 中的 struct thread 增加成员 struct context ， 用于线程切换时保存/恢复寄存器信息。

```
34  struct thread {
35    char       stack[STACK_SIZE]; /* the thread's stack */
36    int        state;             /* FREE, RUNNING, RUNNABLE */
37    struct context context;
38  };
```

Context 定义如下：

```
struct context{
uint64 ra;
uint64 sp;

uint64 s0;
uint64 s1;
uint64 s2;
uint64 s3;
uint64 s4;
uint64 s5;
uint64 s6;
uint64 s7;
uint64 s8;
uint64 s9;
uint64 s10;
uint64 s11;
};
```

2. 在 user/uthread.c 的 thread_create() 函数中添加代码，该函数的主要任务是进行线程的初始化操作。别注意的是，传递给 thread_create() 函数的参数 func 必须记录下来，以便在线程运行时能 够执行该函数。此外，线程有独立的栈结构，函数运行时需要在该线程的栈上进行，因此需要初始 化线程的栈指针。

```
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;

  //该线程直接返回要执行的函数得入口地址
  t->threadContext.ra = (uint64)func;
  //sp指向栈顶，向栈底生长
  t->threadContext.sp = (uint64)(t->stack)+STACK_SIZE;
}
```

3. 在 thread_schedule() 函数中添加代码。该函数负责用户多线程间的调度，通过函数主动调用进 行线程切换。其主要任务是从当前线程在线程数组中的位置开始，寻找一个状态为 RUNNABLE 的线程进行运行。

```
  }

  if (current_thread != next_thread) {         /* switch threads?  */
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    thread_switch((uint64)&t->threadContext,(uint64)&next_thread->threadContext);
  } else
    next_thread = 0;
}
```

4. 在 user/uthread_switch.S 中添加 thread_switch 的代码。正如上文所述，该函数的功 能与 kernel/swtch.S 中的 swtch 函数一致。

uthread.c | uthread_switch.S | swtch.S

user > uthread_switch.S

```
    .text

# Context thread_switch
#
#   extern void thread_switch(uint64, uint64);
#
# Save current registers in old. Load from new.


.globl thread_switch
thread_switch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret /* return to ra */
```

5. 编译运行，运行 uthread，结果如下：

```
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
$
```

6. 单项评分：

```
vale@Puppyyoo:~/xv6-1abs-2021$ ./grade-1ab-thread uthread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.6s)
vale@Puppyyoo:~/xv6-1abs-2021$
```

1.3 实验中遇到的问题和解决办法

问题 ： 上下文切换失败。在 thread_switch() 函数中，无法正确保存和恢复寄存器状态，导致线程 切换失败或程序崩溃。

解决办法 ： 检查 thread_switch 的汇编代码，确保寄存器的保存和恢复操作正确无误。可以对照 kernel/swtch.S 中的 swtch 函数，逐行检查代码的正确性。此外， 可

以通过打印寄存器值或使用调试器，验证寄存器的正确保存和恢复。

1.4 实验心得

通过本次实验，我们成功实现了用户级线程系统的上下文切换机制，并通过了相关测试。实验过程中，我们深入了解了线程的创建、上下文保存与恢复等关键技术，并通过实际编码加 深了对用户级线程系统的理解。通过进一步的优化和改进，可以使用户级线程系统在实际应用中发挥更大的作用。

2 Using threads

2.1 实验目的

通过使用线程和锁实现并行编程，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 命令 编译 notxv6/ph.c ， 并运行 ./ph 1。

```
vale@Puppyyoo:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
vale@Puppyyoo:~/xv6-labs-2021$ ./ph 1
100000 puts, 4.252 seconds, 23520 puts/second
0: 0 keys missing
100000 gets, 4.195 seconds, 23837 gets/second
vale@Puppyyoo:~/xv6-labs-2021$
```

2. 运行 ./ph 2 即使用两个线程运行该哈希表，输出如下，可以看到其 put 速度近乎先前 2 倍，但是有 16721 个键丢失，也说明了该哈希表非线程安全。

```
vale@Puppyyoo: /xv6-labs-2021$ ./ph 2
100000 puts, 1.731 seconds, 57772 puts/second
0: 16721 keys missing
1: 16721 keys missing
200000 gets, 4.061 seconds, 49252 gets/second
```

3. 在 ph.c 中增加一个互斥锁数组，每个 bucket 对应一个互斥锁。

```
19
20    //lock for bucket
21    pthread_mutex_t lock[NBUCKET];
22
```

4. 在 notxv6/ph.c 中实现 init 函数 ， 初始化锁：

```
24  void
25  Init(){
26    for(int i=0;i<NBUCKET;i++){
27      pthread_mutex_init(&lock[i],NULL);
28    }
29  }
30
```

Main（）函数使用 put 调用 insert 之前，使用 init 来初始化锁：

```
20      keys[i] = random();
21    }
22    Init();
23    //
24    // first the puts
25    //
```

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

```
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
```

```
48
49  static
50  void put(int key, int value)
51  {
52    int i = key % NBUCKET;
53    pthread_mutex_lock(&lock[i]); //held the mutex
54    // is the key already present?
55    struct entry *e = 0;
56    for (e = table[i]; e != 0; e = e->next) {
57      if (e->key == key)
58        break;
59    }
60    if(e){
61      // update the existing key.
62      e->value = value;
63    } else {
64      // the new is new.
65      insert(key, value, &table[i], table[i]);
66    }
67    pthread_mutex_unlock(&lock[i]); //release the mutex
68  }
69
```

使用完毕之后，销毁锁防止占用资源：

```
158     printf("%d gets, %.3f seconds, %.0f gets/second\n",
159             NKEYS*nthread, t1 - t0, (NKEYS*nthread) / (t1 - t0));
160   for(int i=0;i<NBUCKET;++i){
161     pthread_mutex_destroy(&lock[i]);
162   }
163 }
164
```

5. 终端使用 make ph 编译 notxv6/ph.c ，再次运行 ./ph 2 命令结果:

```
$ QEMU: Terminated
vale@Puppyyoo:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
vale@Puppyyoo:~/xv6-labs-2021$ ./ph 1
100000 puts, 4.213 seconds, 23737 puts/second
0: 0 keys missing
100000 gets, 4.259 seconds, 23480 gets/second
vale@Puppyyoo:~/xv6-labs-2021$ ./ph 2
100000 puts, 2.454 seconds, 40747 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 3.869 seconds, 51697 gets/second
```

此时，无键丢失，实现了线程安全。

2.3 实验中遇到的问题和解决办法

问题：性能降低幅度过大。

解决办法：在加锁过程中，需要确保锁的粒度尽可能小，以提高并发性能。通过将加锁范围 缩小至 insert() 函数，我们减少了锁的争用，提升了程序性能

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。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 Barrie

3.1 实验目的

本实验的目的是实现一个线程屏障（barrier），即每个线程在到达屏障时等待，直到所有线程都到达屏 障后才能继续运行，从而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，线程屏障用于确 保多个线程在达到某个点后都等待，直到所有其他线程也到达该点。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. 先观察 barrier 结构体。然后在 notxv6/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。

```
struct barrier {
  pthread_mutex_t barrier_mutex;
  pthread_cond_t barrier_cond;
  int nthread;      // Number of threads that have reached this round of the barrier
  int round;     // Barrier round
} bstate;
```

```
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //
  //lock
  pthread_mutex_lock(&bstate.barrier_mutex);

  //increase nthread
  ++ bstate.nthread;
  //look up condition
  if(bstate.nthread!=nthread){
    //abandon the lock and sleep,until broadcast
    pthread_cond_wait(&bstate.barrier_cond,&bstate.barrier_mutex);
  }else{
    //increase round
    ++bstate.round;
    bstate.nthread = 0;
    //broadcast
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
   pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

2. 在终端里执行 make barrier 编译 notxv6/barrier.c ， 分别运行 ./barrier 2 及./barrier 5 结果：

```
vale@Puppyyoo:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
vale@Puppyyoo:~/xv6-labs-2021$ ./barrier 2
OK; passed
```

```
vale@Puppyyoo:~/xv6-labs-2021$ ./barrier 5
OK; passed
```

多线程运行该程序，输出如上，运行成功。

3.3 实验中遇到的问题和解决办法

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。

3.4 实验心得

通过本实验，深入理解了线程屏障机制的实现原理和具体实现过程。通过引入 pthread 条件变量和互 斥锁，我们成功实现了一个线程安全的屏障机制，确保多线程环境下的同步。实验中遇到的问题和解决方 案，使我进一步掌握了多线程编程的技巧和方法。这些经验将为未来的并行编程实践提供宝贵的参考。

4 实验检验得分

1. 在实验目录下创建 time.txt，填写完成实验时间数。

2. 创建 answers-thread.txt 文件,将程序运行结果填入。

3. 在终端中执行 make grade,得到 lab6 总分:

```
make[1]: Leaving directory '/home/vale/xv6-labs-2021'
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.1s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/vale/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/vale/xv6-labs-2021'
ph_safe: OK (6.7s)
== Test ph_fast == make[1]: Entering directory '/home/vale/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/vale/xv6-labs-2021'
ph_fast: OK (15.2s)
== Test barrier == make[1]: Entering directory '/home/vale/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/vale/xv6-labs-2021'
barrier: OK (3.0s)
== Test time ==
time: OK
Score: 60/60
```

4. Lab6 代码提交

保存并提交到本地分支:

```
vale@Puppyyoo:~/xv6-labs-2021$ git add .
vale@Puppyyoo:~/xv6-labs-2021$ git status
On branch thread
Your branch is up to date with 'origin/thread'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   answers-thread.txt
        modified:   notxv6/barrier.c
        modified:   notxv6/ph.c
        new file:   time.txt
        modified:   user/uthread.c
        modified:   user/uthread_switch.S

vale@Puppyyoo:~/xv6-labs-2021$ git commit -m"uthread finished"
[thread eeb6943] uthread finished
 6 files changed, 437 insertions(+), 6 deletions(-)
 create mode 100644 answers-thread.txt
 create mode 100644 time.txt
```