OS Lab 3

Multithreading

王世炟

PB20151796

November 5, 2022

Part 1: 实验要求

本实验要求我们熟悉并使用**多线程**。在一个用户级线程包中实现线程间的切换，使用多线程来加速程序，并实现一个 barrier。

Part 2: Uthread: switching between threads

2.1 问题简述

本部分要求我们为一个用户级线程系统设计上下文切换机制，然后实现它。
在 xv6 有两个文件 user/uthread.c 和 user/uthread_switch.S，以及 Makefile 中的一条规则，用来构建一个 uthread 程序。
我们需要给 user/uthread.c 中的 thread_create() 和 thread_schedule() 以及 user/uthread_switch.S 中的 thread_switch 添加代码。以确保当 thread_schedule() 第一次运行一个给定的线程时，该线程在自己的堆栈中执行传递给 thread_create() 的函数。还要确保 thread_switch 保存被切换走的线程的寄存器，恢复被切换到的线程的寄存器，并返回到后一个线程的指令中它最后离开的地方。

2.2 知识基础

从一个线程切换到另一个线程，需要保存旧线程的 CPU 寄存器，并恢复新线程之前保存的寄存器；栈指针和 pc 被保存和恢复，意味着 CPU 将切换栈和正在执行的代码。
ra 寄存器指向线程要运行的函数，switch 结束后会返回到 ra 处开始运行；sp 指向线程自己的栈。要注意：压栈是减小栈指针，所以一开始在最高处。

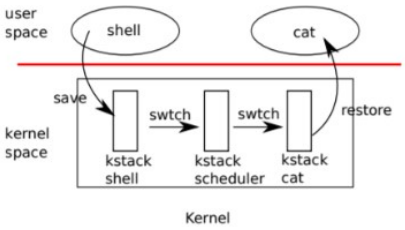

图 1: switch from one user process to another.

2.3 实验步骤

(1) 上下文切换

参考 kernel/switch.S 进行对于 uthread_switch.S 进行编写:

```
        .globl thread_switch
thread_switch:
        /* YOUR CODE HERE */
                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)
```

```
29                ld s8, 80(a1)
30                ld s9, 88(a1)
31                ld s10, 96(a1)
32                ld s11, 104(a1)
33        ret    /* return to ra */
```

(2) **定义上下文字段**

在 uthread.c 中添加 context 结构体作为上下文（可以直接从 kernel/proc.h 中复制）：

```
1  // Saved registers for user context switches.
2  struct context {
3    uint64 ra;
4    uint64 sp;
5
6    // callee-saved
7    uint64 s0;
8    uint64 s1;
9    uint64 s2;
10   uint64 s3;
11   uint64 s4;
12   uint64 s5;
13   uint64 s6;
14   uint64 s7;
15   uint64 s8;
16   uint64 s9;
17   uint64 s10;
18   uint64 s11;
19 };
```

在线程的结构体中进行声明：

```
1  struct thread {
2    char        stack[STACK_SIZE]; /* the thread's stack */
3    int         state;             /* FREE, RUNNING, RUNNABLE */
4    struct context context;        // swtch() here to enter scheduler().
5  };
```

(3) **完成** thread_schedule()

在 thread_schedule 中调用 thread_switch：

```
1  void
2  thread_schedule(void)
3  {
4    ...
5    if (current_thread != next_thread) {         /* switch threads?  */
```

```
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    thread_switch((uint64)(&(t->context)),(uint64)(&(current_t
    hread->context)));
  } else
    next_thread = 0;
}
```

(3) **完成** thread_create()

创建并初始化线程

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

线程栈是从高位到低位，因此初始化时栈指针 sp 应该指向数组底部.

返回地址 ra 直接指向该函数的地址，这样开始调度时，直接执行该函数（线程）。

(4) **结果**

按照实验流程得到以下结果：

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel/k
e virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 2
thread_c 3
thread_a 3
thread_b 3
thread_c 4
thread_a 4
thread_b 4
thread_c 5
thread_a 5
thread_b 5
thread_c 6
thread_a 6
thread_b 6
thread_c 7
thread_a 7
thread_b 7
thread_c 8
thread_a 8
thread_b 8
thread_c 9
thread_a 9
thread_b 9
thread_c 10
```

图 2: switching passed

```
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
$ QEMU: Terminated
ubuntu@VM5153-OSVM:~/桌面/xv6-labs-2020$ make clean
```

图 3: switching passed

Part 3: Using threads

3.1 问题简述

在 notxv6/ph.c 中实现了一个哈希表，要求我们将它改成线程安全的。

3.2 问题分析

在多线程并发时，会产生竞争条件，引起运行错误，解决办法是加锁。
A sequence of events with 2 threads that can lead to a key being missing：
[假设键 k1、k2 属于同个 bucket]
thread 1: 尝试设置 k1
thread 1: 发现 k1 不存在，尝试在 bucket 末尾插入 k1 — scheduler 切换到 thread 2
thread 2: 尝试设置 k2
thread 2: 发现 k2 不存在，尝试在 bucket 末尾插入 k2
thread 2: 分配 entry，在桶末尾插入 k2

— scheduler 切换回 thread 1

thread 1: 分配 entry，没有意识到 k2 的存在，在其认为的“桶末尾”（实际为 k2 所处位置）插入 k1

[k1 被插入，但是由于被 k1 覆盖，k2 从桶中消失了，引发了键值丢失]

题目中给出了关于锁的操作：

```
pthread_mutex_t lock;              // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock);         // acquire lock
pthread_mutex_unlock(&lock);       // release lock
```

为了多线程能够比单线程运行的更快，我们选择加小锁，即每个桶加一个锁。

3.3 实验步骤

首先在全局声明锁：

```
pthread_mutex_t lock[NBUCKET];
```

然后在 main 函数中初始化锁：

```
int
main(int argc, char *argv[])
{
  pthread_t *tha;
  void *value;
  double t1, t0;

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

最后，在 put() 和 get() 函数中加锁：

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;

```

```
  pthread_mutex_lock(&lock[i]);
  // is the key already present?
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    insert(key, value, &table[i], table[i]);
  }
  pthread_mutex_unlock(&lock[i]);
}
```

```
static struct entry*
get(int key)
{
  int i = key % NBUCKET;
  pthread_mutex_lock(&lock[i]);

  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key) break;
  }
  pthread_mutex_unlock(&lock[i]);
  return e;
}
```

```
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/ubuntu/桌面/xv6-labs-2020”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
ph_safe: OK (21.2s)
== Test ph_fast == make[1]: 进入目录“/home/ubuntu/桌面/xv6-labs-2020”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
ph_fast: OK (46.8s)
```

图 4: ph passed

Part 4: Barrier

条件变量：条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待”条件变量的条件成立”而挂起；另一个线程使”条件成立”（给出

条件成立信号)。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

原理：线程进入同步屏障 barrier 时，将已进入屏障的线程数量增加 1，然后再判断是否已经达到总线程数。如果未达到，则进入睡眠，等待其他线程。如果已经达到，则唤醒所有在 barrier 中等待的线程，所有线程继续执行；屏障轮数 + 1；

```
1  static struct entry*
2  static void
3  barrier()
4  {
5    // YOUR CODE HERE
6    //
7    // Block until all threads have called barrier() and
8    // then increment bstate.round.
9    //
10   pthread_mutex_lock(&bstate.barrier_mutex);
11   if (++bstate.nthread == nthread)
12   {
13     bstate.nthread = 0;
14     bstate.round++;
15     pthread_cond_broadcast(&bstate.barrier_cond);
16   }
17   else
18   {
19     pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
20   }
21   pthread_mutex_unlock(&bstate.barrier_mutex);
22 }
```

Part 5: 实验结果

运行 make grade，检验结果：

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (6.0s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/ubuntu/桌面/xv6-labs-2020”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
ph_safe: OK (21.4s)
== Test ph_fast == make[1]: 进入目录“/home/ubuntu/桌面/xv6-labs-2020”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
ph_fast: OK (46.1s)
== Test barrier == make[1]: 进入目录“/home/ubuntu/桌面/xv6-labs-2020”
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
barrier: OK (2.7s)
== Test time ==
time: OK
Score: 60/60
ubuntu@VM5153-OSVM:~/桌面/xv6-labs-2020$
```

图 5: Result

本次实验完成了 Multithreading 某些功能的实现和实际应用。在实验过程中得到了以下收获：

- 对于进程切换的上下文的认知不是很清晰，通过本次实验让我对于这个知识点有了深刻的认识
- 对于竞争条件的发生有了直观的理解，观察到了此问题实际带来的后果
- 认识到了加锁等操作的合理性以及正确性