Multithread 实验

20307130340 杨孟颖 计算机

一、实验目的

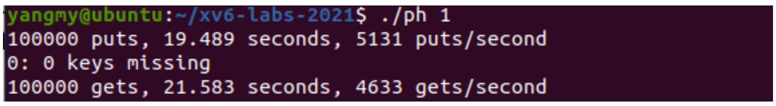
在向 hash 表中插入项时，若使用单线程不会产生遗漏，但是使用多线程会产生遗漏。实验要求向插入过程加入锁，使插入表项在多线程运行的情况下也可以达到正确的结果，并且实现使多线程操作表项的 CPU 效率更高。

二、实验原理

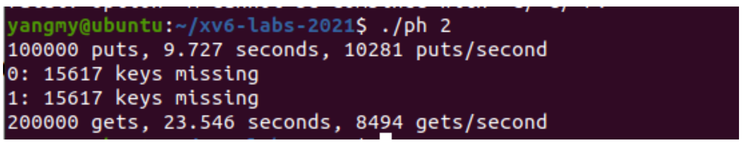
并发编程由于不同线程可以访问同一个变量。在资源竞争的情况下就会产生程序运行结果不确定的后果。在 ph.c 中可以看到，在命中 hash 桶之后采取 insert 操作：

```
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    insert(key, value, &table[i], table[i]);
  }
static void
insert(int key, int value, struct entry **p, struct entry *n)
{
  struct entry *e = malloc(sizeof(struct entry));
  e->key = key;
  e->value = value;
  e->next = n;
  *p = e;
}
```

而 insert 函数并不是原子的。如果有两个 thread 同时 insert，由于 e->next = n 和*p = e 两条语句并不原子，可能会丢 entry。如果在执行 insert 函数的时候时间片结束开始执行另一个线程的 insert 函数，就会导致插入失败。



```
yangmy@ubuntu:~/xv6-labs-2021$ ./ph 1
100000 puts, 19.489 seconds, 5131 puts/second
0: 0 keys missing
100000 gets, 21.583 seconds, 4633 gets/second
```



```
yangmy@ubuntu:~/xv6-labs-2021$ ./ph 2
100000 puts, 9.727 seconds, 10281 puts/second
0: 15617 keys missing
1: 15617 keys missing
200000 gets, 23.546 seconds, 8494 gets/second
```

而 C 语言中 pthread.h 头文件的库中包含的 pthread_mutex_lock(&lock)可以让线程获得一个互斥锁，让资源只能由获得锁的线程访问，保证程序执行的确定性。

在这个实验中，对于 hash 表的每一个桶，再插入时应该为一个原子操作，因此声明 NKEYS 个锁，对应于 hash 表的所有桶。在加上锁之后只有相应 hash 桶的值可以访问资源，这样就防止插入操作被打断。

三、实验内容

1、 上下文进程切换

通过执行 swtch 函数实现该过程：swtch(ctx1, ctx2);

保存 callee 寄存器的内容到 ctx1 中；恢复所有的 ctx2 的内容到 callee 寄存器里。

thread_switch 只需要保存/恢复 callee-save registers。原因是：switch 是按照一个普通函数来调用的，对于有些寄存器，swtch 函数的调用者默认 swtch 函数会修改。所以调用者已经在自己的栈上保存了这些寄存器，当函数返回时这些寄存器会自动回恢复。

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&p->lock))
    panic("sched p->lock");
  if(mycpu()->noff != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(intr_get())
    panic("sched interruptible");

  intena = mycpu()->intena;
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}
```

定义 context switches 存储上下文切换时需保存的 14 个 callee 寄存器的结构体。参照了 kernel/proc.h 中的 struct context 的定义，因为 struct context 的注释中写明了 struct context 也是保存 context switches 的结构体。（ // Saved registers for kernel context switches.）

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

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
```

```
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```
在 thread 结构体中添加 struct context context;用以保存上下文线程切换时需要恢复的寄存器的值。
```
struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct context context;

};
```
在添加函数体值时参考了 kernel/proc.c 中 scheduler(void) 函数的 swtch(&c->context, &p->context);实现上下文线程切换。
```
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) {         /* 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_thread->context);
  } else
    next_thread = 0;
}
```

通过阅读函数体得知 `thread_create(void (*func)())`函数主要进行线程的初始化操作：其先在线程数组中找到一个状态为 FREE 即未初始化的线程，然后设置其状态为 RUNNABLE 等进行初始化。传递的 `thread_create()` 参数 func 需要记录，这样在线程运行时才能运行该函数。这一功能对应于 ra 寄存器。线程进行调度切换时，需要保存和恢复寄存器状态。这一功能对应于 sp 寄存器。

```
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;
}
```

```
thread_c 89
thread_a 89
thread_b 89
thread_c 90
thread_a 90
thread_b 90
thread_c 91
thread_a 91
thread_b 91
thread_c 92
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、pthread_mutex_t lock[NKEYS]将所声明为全局变量，并在 main 函数中将其初始化：pthread_mutex_init(lock, NULL)。对竞争资源 table 加上锁，如图所示：

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

  // is the key already present?
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  pthread_mutex_lock(&lock[i]);       // acquire lock
  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]);     // release lock
}

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

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

结果如图所示：

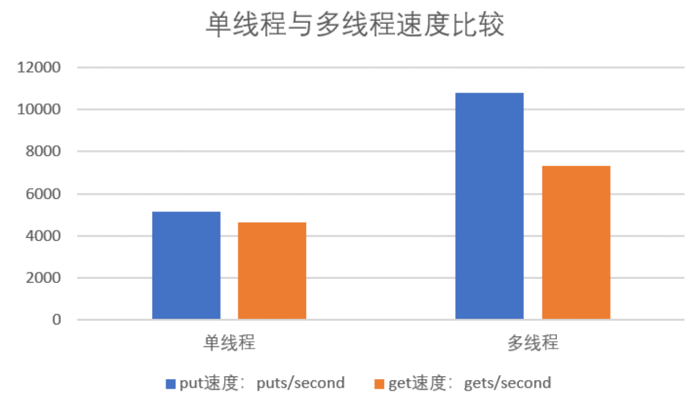
```
yangmy@ubuntu:~/xv6-labs-2021$ ./ph 2
100000 puts, 9.273 seconds, 10784 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 27.268 seconds, 7335 gets/second
```

运行 make grade 的结果如下：

```
make[1]: Leaving directory '/home/yangmy/xv6-labs-2021'
ph_safe: OK (19.8s)
== Test ph_fast == make[1]: Entering directory '/home/yangmy/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/yangmy/xv6-labs-2021'
ph_fast: OK (56.4s)
```

成功通过准确性和速度测试。

将单线程和多线程的运行结果的速度进行比较，做出速度对比图如下：



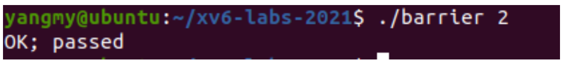
可见单线程速度远低于多线程。

3、barrier

通过阅读实验要求和代码得知 barrier.c 文件的要求是使创建的线程循环 loop 次，每次循环需要使所有线程都进行过一遍之后再开始下一轮循环。因此可知，播报条件为 bstate.nthread==nthread，此时 pthread_cond_broadcast(&bstate.barrier_cond)。否则线程在此等待，并将 bstate.nthread 自增 1，表示进行过 barrier 的线程数量增加 1.

```
static void
barrier()
{
  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);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

执行结果：



实验结果：

```
$ make qemu-gdb
uthread: OK (5.7s)
== Test answers-thread.txt == answers-thread.txt: FAIL
    Cannot read answers-thread.txt
== Test ph_safe == make[1]: Entering directory '/home/yangmy/lab5/xv6-labs-2022'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/yangmy/lab5/xv6-labs-2022'
ph_safe: OK (12.4s)
== Test ph_fast == make[1]: Entering directory '/home/yangmy/lab5/xv6-labs-2022'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/yangmy/lab5/xv6-labs-2022'
ph_fast: OK (31.6s)
== Test barrier == make[1]: Entering directory '/home/yangmy/lab5/xv6-labs-2022'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/yangmy/lab5/xv6-labs-2022'
barrier: OK (12.3s)
```

四、实验中遇到的问题

1、 由于需要存储线程切换时的状态，所以需要在

```
struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
};
```

结构体中添加存储寄存器的结构 struct context context; 从 kernel/proc.h 中可以找到功能一样的结构体 struct context，因此刚开始的想法是直接使用 #include "kernel/proc.h" 从而使用已经定义好的 struct context。但是会产生以下报错信息：

```
/uthread.c
In file included from user/uthread.c:4:
./kernel/proc.h:29:24: error: 'NCPU' undeclared here (not in a function)
   29 | extern struct cpu cpus[NCPU];
      |                        ^~~~
./kernel/proc.h:86:19: error: field 'lock' has incomplete type
   86 |   struct spinlock lock;
      |                   ^~~~
./kernel/proc.h:101:3: error: unknown type name 'pagetable_t'
  101 |   pagetable_t pagetable;       // User page table
      |   ^~~~~~~~~~~
./kernel/proc.h:104:22: error: 'NOFILE' undeclared here (not in a function); did you mean 'T_FILE'?
  104 |   struct file *ofile[NOFILE];  // Open files
      |                      ^~~~~~
      |                      T_FILE
user/uthread.c: In function 'thread_schedule':
user/uthread.c:68:20: error: passing argument 1 of 'thread_switch' makes integer from pointer without a cast [-Werror=int-conversion]
   68 |     thread_switch(&t->context, &current_thread->context);
      |                   ^~~~~~~~~~~
      |                   |
      |                   struct context *
user/uthread.c:23:27: note: expected 'uint64' {aka 'long unsigned int'} but argument is of type 'struct context *'
   23 | extern void thread_switch(uint64, uint64);
      |                           ^~~~~~
user/uthread.c:68:33: error: passing argument 2 of 'thread_switch' makes integer from pointer without a cast [-Werror=int-conversion]
   68 |     thread_switch(&t->context, &current_thread->context);
      |                                ^~~~~~~~~~~~~~~~~~~~~~~~
      |                                |
      |                                struct context *
user/uthread.c:23:35: note: expected 'uint64' {aka 'long unsigned int'} but argument is of type 'struct context *'
   23 | extern void thread_switch(uint64, uint64);
      |                                   ^~~~~~
cc1: all warnings being treated as errors
make: *** [<builtin>: user/uthread.o] Error 1
```

于是不引入 kernel/proc.h，直接在 uthread.c 中再定义 context 结构体。成功解决问题。

2、 在修改 thread_schedule(void)函数时，刚开始使用 thread_switch 的方式是：thread_switch(&t->context, &current_thread->context);从逻辑上没有问题，但是会出现以下报错：

```
user/uthread.c: In function 'thread_schedule':
user/uthread.c:85:20: error: passing argument 1 of 'thread_switch' makes integer from pointer without a cast [-Werror=int
-conversion]
   85 |       thread_switch(&t->context, &current_thread->context);
      |                     ^~~~~~~~~~~
      |                     |
      |                     struct context *
user/uthread.c:40:27: note: expected 'uint64' {aka 'long unsigned int'} but argument is of type 'struct context *'
   40 | extern void thread_switch(uint64, uint64);
      |                           ^~~~~~
user/uthread.c:85:33: error: passing argument 2 of 'thread_switch' makes integer from pointer without a cast [-Werror=int
-conversion]
   85 |       thread_switch(&t->context, &current_thread->context);
      |                                  ^~~~~~~~~~~~~~~~~~~~~~~~
      |                                  |
      |                                  struct context *
user/uthread.c:40:35: note: expected 'uint64' {aka 'long unsigned int'} but argument is of type 'struct context *'
   40 | extern void thread_switch(uint64, uint64);
      |                                   ^~~~~~
cc1: all warnings being treated as errors
```

通过阅读报错信息得知是输入参数的类型与定义不符。因此使用强制类型转换：`thread_switch((uint64)&t->context, (uint64)&current_thread->context);` 修改后成功解决问题。

3、 由于书上所给的例子都为对线程加锁，所以一开始也尝试对线程加锁：

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

  // is the key already present?
  pthread_mutex_lock(&lock[value]);       // acquire lock
  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[value]);     // release lock
}
```

Put 函数的第二个参数为线程数。运行结果如图：

```
yangmy@ubuntu:~/xv6-labs-2021$ ./ph 2
100000 puts, 3.530 seconds, 28328 puts/second
1: 15048 keys missing
0: 15048 keys missing
200000 gets, 10.019 seconds, 19962 gets/second
```

发现依然出现未命中的情况。这是由于加锁应该保护的是访问的变量而不是线程。后改为给每个桶加上锁。

2、在实验中给桶枷锁时，还尝试了加在遍历桶之前并在 get 函数中也加入锁。这时实验结果是正确的，但是速度比加在 insert 函数中慢。这是因为遍历桶环节和 get 函数对键值插入 hash 桶没有影响，因为在退出线程时寄存器会记录线程运行状态。在遍历之前加锁会使遍历环节无法并行执行，因此速度变慢。代码和执行情况如图：

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

  // is the key already present?
  pthread_mutex_lock(&lock[i]);       // acquire lock
  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]);     // release lock
}

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

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

```
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/yangmy/xv6-labs-2021'
ph_safe: OK (35.5s)
== Test ph_fast == make[1]: Entering directory '/home/yangmy/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/yangmy/xv6-labs-2021'
ph_fast: OK (83.2s)
```

五、实验总结

在实验过程中，我对锁的使用条件和使用方式有了更深刻的理解，了解了加锁对多线程程序正确进行的重要作用，并学着分析应当在那些地方加锁。