lab thread

司若童 19300240024

实验准备

首先clone 2021年的xv6 labs，切换到thread branch。

Uthread

实验要求

实现用户级的线程，是制定一个计划来创建线程并保存/恢复寄存器以在线程之间切换。

实验步骤

定义一个context结构体，用于保存进程的上下文，包括被调用者保存的寄存器sp和s0~s11寄存器，用于保存返回值的ra寄存器:

```c
#define RUNNABLE     0x2

#define STACK_SIZE   8192
#define MAX_THREAD   4

struct thread_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;
  char          stack[STACKSIZE];
};
```

修改thread结构体，添加context：

```c

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

修改thread_create，对寄存器初始化，ra为函数入口，sp和s0指向栈底：

```c
     * thread_switch(??, ??);
     */
  } else
    next_thread = 0;
}

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 - 1];
  t->context.s0 = (uint64)&t->stack[STACK_SIZE - 1];
}
```

修改thread_schedule：

uthread.c ~/Desktop/xv6/xv6-labs-2021/user

```c
    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;
```

修改uthread_switch.s：

sd=store doubleword，存储双字

ld=load doubleword，加载双字

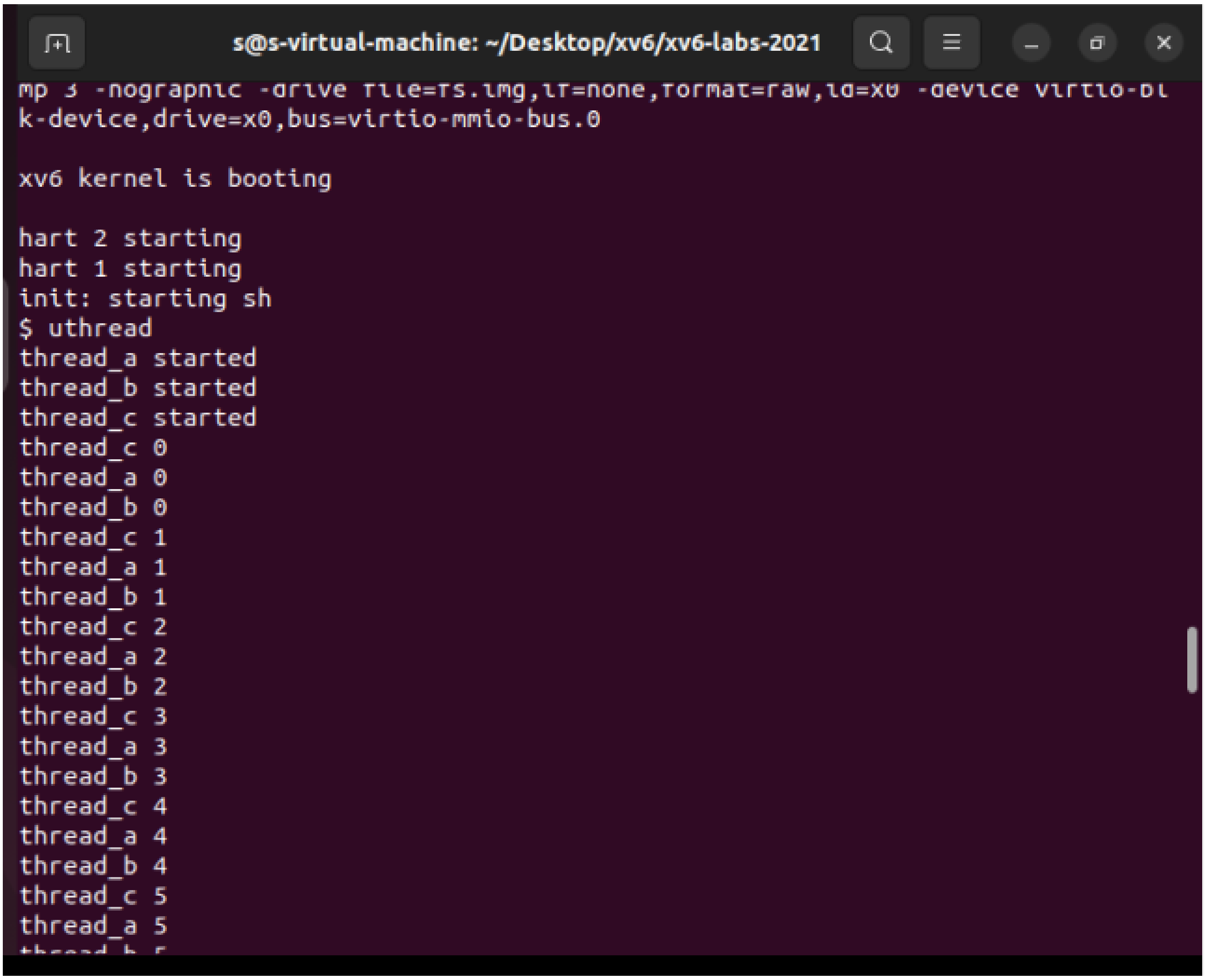
根据：

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

```
thread_switch:
    /* YOUR CODE HERE */
    sd ra, 0(a0)

    sd sp, 8(a0)
    sd fp, 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 sp, 8(a1)
    ld fp, 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)

    ld ra, 0(a1) /* set return address to next thread */
    ret    /* return to ra */
```

实验结果



s@s-virtual-machine: ~/Desktop/xv6/xv6-labs-2021

```
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
$
```

using thread

实验要求

使用hash表探索并行，ph.c中包含了一个单线程可正确使用的hash表，将其修改成多线程使用正确的。

实验步骤

测试ph

```
s@s-virtual-machine:~/Desktop/xv6-labs-2021/notxv6$ make ph
cc     ph.c    -o ph
s@s-virtual-machine:~/Desktop/xv6-labs-2021/notxv6$ ./ph 1
100000 puts, 5.607 seconds, 17834 puts/second
0: 0 keys missing
100000 gets, 5.557 seconds, 17994 gets/second
```

单线程使用：ph调用put()函数添加了大量的key，print每秒put的数量；调用get()函数，从hash表中获取keys，print每秒get的数量。

```
s@s-virtual-machine:~/Desktop/xv6-labs-2021/notxv6$ ./ph 2
100000 puts, 2.738 seconds, 36526 puts/second
1: 112 keys missing
0: 112 keys missing
200000 gets, 5.438 seconds, 36776 gets/second
```

多线程使用：发现存在keys丢失，说明多线程并行时发生错误，需要对put操作使用lock

修改ph.c

暴力lock

定义互斥锁lock，并动态初始化：

```c
struct entry *table[NBUCKET];
int keys[NKEYS];
int nthread = 1;
pthread_mutex_t lock;
```

```c
  pthread_mutex_init(&lock,NULL);
```

critical session：

```c
static void *
put_thread(void *xa)
{
  int n = (int) (long) xa; // thread number
  int b = NKEYS/nthread;

  for (int i = 0; i < b; i++) {
    put(keys[b*n + i], n);
  }

  return NULL;
}
```

加锁：

```
for (int i = 0; i < b; i++) {
  pthread_mutex_lock(&lock);
  put(keys[b*n + i], n);
  pthread_mutex_unlock(&lock);
}
```

运行：

```
s@s-virtual-machine:~/Desktop/xv6-labs-2021/notxv6$ ./ph 2
100000 puts, 5.729 seconds, 17456 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 5.734 seconds, 34882 gets/second
```

没有missing keys

尝试将lock放在for循环外：

```
put_thread(void *xa)
{
  int n = (int) (long) xa; // thread number
  int b = NKEYS/nthread;
  pthread_mutex_lock(&lock);
  for (int i = 0; i < b; i++) {

    put(keys[b*n + i], n);

  }
  pthread_mutex_unlock(&lock);
  return NULL;
}
```

运行：

```
s@s-virtual-machine:~/Desktop/xv6-labs-2021/notxv6$ ./ph 2
100000 puts, 5.700 seconds, 17544 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 5.709 seconds, 35032 gets/second
```

但是两种方法都没办法通过make grade，不满足ph_fast：

```
== Test ph_safe == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-20
21'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
ph_safe: OK (11.7s)
== Test ph_fast == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-20
21'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
ph_fast: FAIL (23.3s)
    Parallel put() speedup is less than 1.25x
```

粒度更细的lock

重新观察，如果多个key映射到同一个bucket，hash表的做法是在bucket后面链接一串链表。

```c
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;
}
```

insert操作导致了race condition，所以对insert加锁：

```c
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);
  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);
}
```

操作：

```
s@s-virtual-machine:~/Desktop/xv6-labs-2021/notxv6$ ./ph 2
100000 puts, 2.732 seconds, 36608 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 5.239 seconds, 38175 gets/second
```

成功！

```
== Test ph_safe == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-20
21'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
ph_safe: OK (8.3s)
== Test ph_fast == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-20
21'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
ph_fast: OK (19.3s)
```

再尝试根据提示：每个hash bucket都加锁，尝试用多个lock：

```c
  pthread_mutex_lock(&locks[i]);
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    insert(key, value, &table[i], table[i]);
  }
  pthread_mutex_unlock(&locks[i]);
}
```

运行：

```
s@s-virtual-machine:~/Desktop/xv6-labs-2021/notxv6$ make ph
cc     ph.c   -o ph
s@s-virtual-machine:~/Desktop/xv6-labs-2021/notxv6$ ./ph 2
100000 puts, 2.685 seconds, 37241 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 5.355 seconds, 37347 gets/second
```

make grade：

```
== Test ph_safe == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-20
21'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
ph_safe: OK (8.2s)
== Test ph_fast == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-20
21'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
ph_fast: OK (20.1s)
```

结束！

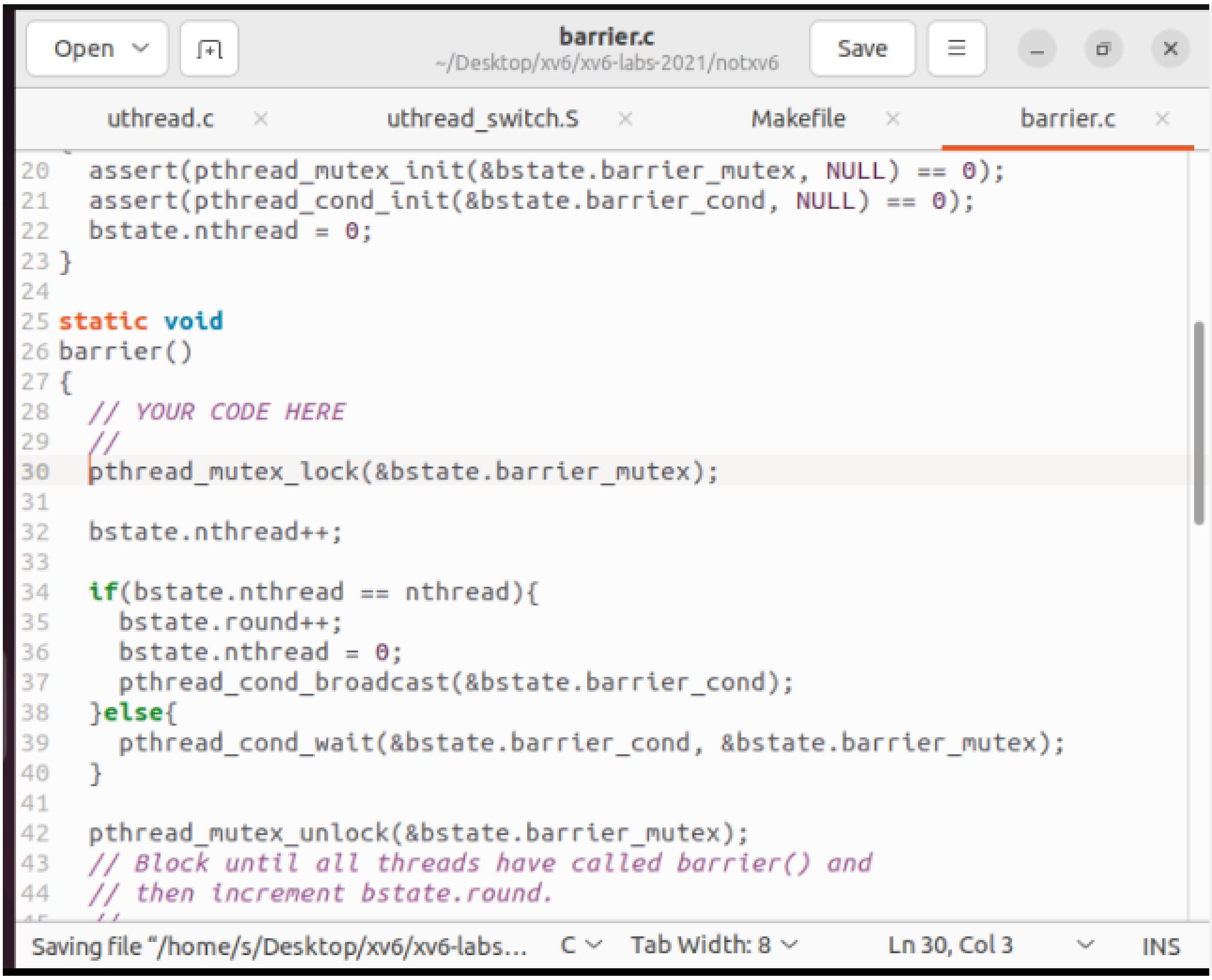
barrier

实验目的

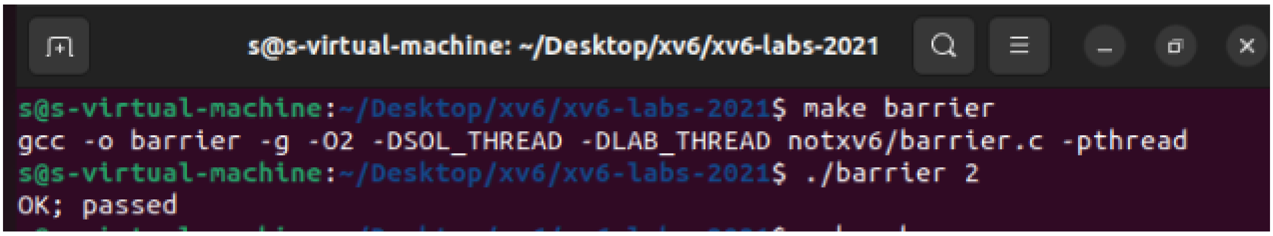
设置一个barrier，线程全部到达barrier后才能继续执行。

实验步骤

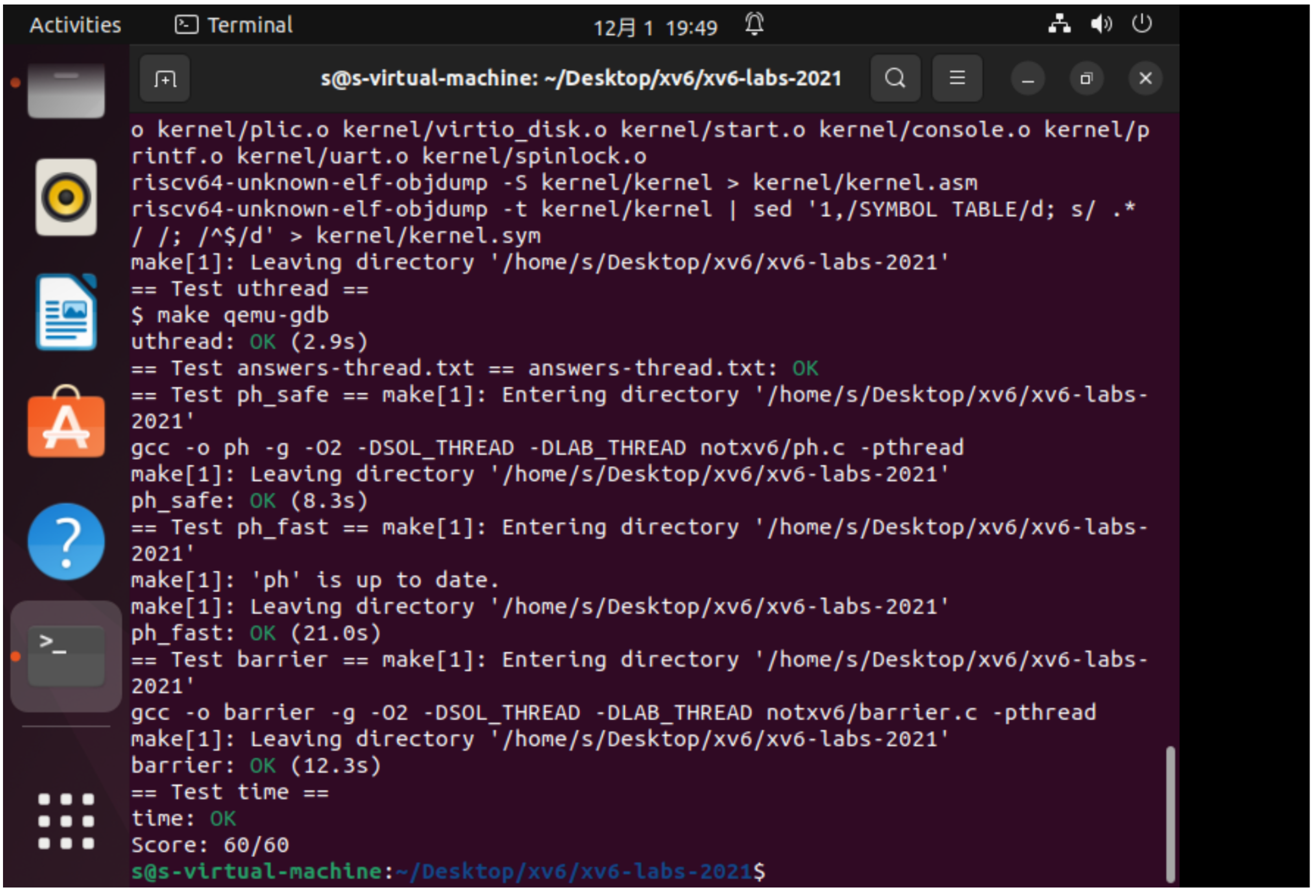
barrier.c：



实验结果



make grade


Activities
Terminal
12月 1 19:49
s@s-virtual-machine: ~/Desktop/xv6/xv6-labs-2021
o kernel/plic.o kernel/virtio_disk.o kernel/start.o kernel/console.o kernel/p
rintf.o kernel/uart.o kernel/spinlock.o
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .*
/ /; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.9s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-
2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
ph_safe: OK (8.3s)
== Test ph_fast == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-
2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
ph_fast: OK (21.0s)
== Test barrier == make[1]: Entering directory '/home/s/Desktop/xv6/xv6-labs-
2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/s/Desktop/xv6/xv6-labs-2021'
barrier: OK (12.3s)
== Test time ==
time: OK
Score: 60/60
s@s-virtual-machine:~/Desktop/xv6/xv6-labs-2021$