Lab1/3/6 实习报告

姓名：吴悦欣 学号：1900012946

日期：2021/10/16

目录

内容一：实验总结

一、Lab1

本次实验共有两个部分，都是针对系统调用的实现展开的。

第一部分是实现 **trace 系统调用**，该函数的功能是：当给出的 mask 参数指定的系统调用被调用时，会输出一行包含当前进程的 pid、对应系统调用名称以及该调用的返回值的字符串。原代码提供了 trace 的用户调用代码。该过程的系统调用过程如下：

a. 在用户程序中调用 int trace(int mask)函数后，会把该进程的 a7 寄存器的值赋为对应的系统调用号 SYS_trace（定义在 syscall.h 中），函数调用需要的参数存放在 a0 寄存器中
b. 通过 ecall 指令进入内核【该过程在 usys.pl 中实现】
c. 进程切换后，内核开始执行 syscall，查看 a7 中断的调用号【该过程在 syscall.c 的 syscall 函数中】调用对应的函数 sys_trace()，通过 argint 函数 sys_trace 函数可以获得对应的参数【从 trapframe 结构的 a0 中获得参数】，并返回需要的值，放在 a0 寄存器中返回。

因此如果需要打印对应的结果，只需要在 syscall 的函数中判断函数调用号是否对应 mask 之后输出相关信息即可。下面是完成该函数的具体步骤（可以根据 git 版本管理在左侧显示的颜色条分辨我进行的改动）。

1. 在 user/user.h 中添加 trace 函数的声明

```
user > C user.h > trace(int)
25  int sleep(int);
26  int uptime(void);
27  int trace(int);
28  int sysinfo(struct sysinfo*);
29
```

2. 在 user/usys.pl 中添加 trace 函数系统调用的入口

```
user > usys.pl
38  entry("uptime");
39
40  entry("trace");
41  entry("sysinfo");
```

3. 在 kernel/syscall.h 中添加对应的系统调用号

```
kernel > C syscall.h > SYS_trace
22  #define SYS_close  21
23  #define SYS_trace  22
24  #define SYS_sysinfo 23
```

4. 在 kernel/syscall.c 中添加对应的函数声明，函数指针，用来输出函数名称的字符串数组以及修改 syscall 实现 trace 要求的功能

```
kernel > C syscall.c > syscalls
105  extern uint64 sys_write(void);
106  extern uint64 sys_uptime(void);
107  extern uint64 sys_trace(void);
108  extern uint64 sys_sysinfo(void);
109
```

```c
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    uint64 sys_pipe(void)
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_trace]   sys_trace,
[SYS_sysinfo] sys_sysinfo,
};
```

```c
static char* syscallnames[] = {
  "fork",
  "exit",
  "wait",
  "pipe",
  "read",
  "kill",
  "exec",
  "fstat",
  "chdir",
  "dup",
  "getpid",
  "sbrk",
  "sleep",
  "uptime",
  "open",
  "write",
  "mknod",
  "unlink",
  "link",
  "mkdir",
  "close",
  "trace",
  "sysinfotest"
};
```

kernel > C syscall.c > [@] syscalls

```c
// 根据trapframe中的a7进行对应的系统调用
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    if((1 << num) == p->trace_mask || p->trace_mask == 0x7fffffff){
      printf("%d: syscall %s -> %d\n", p->pid, syscallnames[num - 1], p->trapframe->a0);
    }
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

5. 在 kernel/sysproc.c 中添加对应被调用的函数 sys_trace()，同时在 proc 的进程结构体（kernel/proc.h）中增加一个变量来记录标志了需要打印的函数对象的 mask

kernel > C sysproc.c > sys_trace(void)

```c
sys_trace(void)
{
  int n;
  if(argint(0, &n) < 0)
    return -1;
  myproc()->trace_mask = n;
  return 0;
}
```

kernel > C proc.h > proc > trapframe

```c
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table 原型是uint64的指针
  struct trapframe *trapframe; // data page for trampoline.S--汇编文件，主要工作是进入实模式读取内存磁
  struct context context;      // swtch() here to run process，上下文环境，是一些环境切换时需要保存的寄
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int trace_mask;              // number to record the mask for the trace function
};
```

6. 修改 fork()函数（kernel/proc.c）使得子进程继承父进程的 trace_mask，打印对应的调用过程

```
kernel > C proc.c > fork(void)
279     np->trace_mask = p->trace_mask;
```

在第二个任务中是需要完成 **sysinfo** 系统调用，该函数的主要功能是：将当前进程剩余空间的字节数以及非空闲的进程数赋值给对应结构体 sysinfo 的属性。在实现系统调用的过程类似于第一个 trace 的实现，需要添加对应的入口、系统调用号、声明以及对应的系统调用函数 sys_sysinfo，不同的在于需要另外实现对剩余空间和非空闲进程的计数：

1. kernel/kalloc.c 中对剩余空间的字节数：kmem 是内核内存的结构体，其中 freelist 属性指向第一个空闲的页，因此只需要计算剩下的空闲页的个数*页大小即可。

```
kernel > C kalloc.c > freemem_size(void)
84  int
85  freemem_size(void)
86  {
87    struct run *r;
88    int num = 0;
89
90    r = kmem.freelist;
91    while(r){
92      num++;
93      r = r->next;
94    }
95    return num * PGSIZE;
96  }
```

2. kernel/proc.c 中对非空闲空间的计数：proc 为所有进程的集合，遍历所有进程对象判断其状态进行计数即可。

```
kernel > C proc.c > proc_num(void)
696
697 int
698 proc_num(void)
699 {
700   struct proc *p;
701   int num = 0;
702   for(p = proc; p < &proc[NPROC]; p++){
703     if(p->state != UNUSED){
704       num++;
705     }
706   }
707   return num;
708 }
```

3. 在 kernel/sysproc.c 中书写 sys_sysinfo 函数，调用 1，2 中的函数给传入的指针对应的 sysinfo 结构体赋值，返回进程信息。

```
kernel > C sysproc.c > sys_trace(void)
114 uint64
115 sys_sysinfo(void)
116 {
117   struct proc *p = myproc();
118   struct sysinfo sysinfo_item;
119   uint64 addr;
120
121   if(argaddr(0, &addr) < 0)
122     return -1;
123   sysinfo_item.freemem = freemem_size();
124   sysinfo_item.nproc = proc_num();
125   if(copyout(p->pagetable, addr, (char *)&sysinfo_item, sizeof(sysinfo_item)) < 0)
126     return -1;
127   return 0;
128 }
```

根据评分要求添加了 time.txt 并写入用时 5h，lab1 最终的测试结果如下：

```
bettywu@ubuntu:~/xv6-labs-2020$ ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.4s)
== Test trace all grep == trace all grep: OK (0.9s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (12.4s)
== Test sysinfotest == sysinfotest: OK (2.3s)
== Test time ==
time: OK
Score: 35/35
```

二、Lab3

本次实验是关于 trap 陷阱的实现。在做实验前，先根据课本对应内容阅读了 kernel/trampoline.S 部分的代码，这是一个切换用户/内核空间的汇编文件，其中 uservec 部分在进入内核空间前保存了用户进程的寄存器状态，并将进程的 trapframe 中关于内核的部分进行对应的赋值，切换到内核进程；之后调用 kernel/trap.c 中的 usertrap 函数，对于陷阱的类别进行区分后进行对应的处理——系统的调用（syscall）或是时钟的中断或是其他的异常（exit）；最后再回到 trampoline.S 中的 userret 部分恢复用户空间。通过这一步，我对于 trap 的机制有了大致初步的了解。

实验的**第一部分**是根据一个测试代码的汇编结果回答对应的问题，以下是问题及对应的回答：

1. Which registers contain arguments to the functions? For example, which register holds 13 in main's call to printf?
   函数调用通过 a0, a1, a2...a8 传递参数，返回值放在 a0 或 a1 寄存器。在 printf 中，由 a2 传递 13 这个参数。
2. Where is the call to function f in the assembly code for main? Where is the call to g?
   在 main 函数中，f 的调用被优化了，在 printf 之前直接把 12 这个结果赋值给了对应的寄存器。而对 g 的调用是在 f 函数中，在汇编中也被内联优化，没有调用的过程，直接将 g 对应的汇编实现加入到 f 的汇编中。
3. At what address is the function printf located? 0x640
4. What value is in the register ra just after the jar to printf in main? 0x30
5. What is the output? If big -endian, what is the proper value for i? And if the change to 57616 be needed?
   Output: He110 World
   大端：高字节低地址；
   小端：低字节低地址
   都不需要改变，57616 的十六进制打印不会被大小端影响，而 i 的值的存储方式被强制类型转换成字符串之后和字符的存储顺序也是一致的，所以不需要变换。
6. What will be printed after 'y='? Why?
   -280324136；应该是寄存器 a2 中的值，printf 会在 a2 获取 y 的值

主要解决的问题是函数调用过程中的传参方式、内联优化、函数跳转的定位、大小端的区别。

第二部分是实现函数 backtrace，该函数在被调用的时候打印在目前调用的函数栈帧地址空间之上的栈中的函数（即经过层层调用到达当前的函数，希望得到整个调用的路径）。

主要的修改是在 kernel/printf.c 中增加 backtrace 函数以及在 kernel/riscv.h 中增加内联汇编将当前栈帧所在寄存器 s0 的值返回，相关代码和运行结果如下：

kernel > C printf.c > ...

```
137  void
138  backtrace(void)
139  {
140    uint64 frame_p = r_fp();
141    uint64 bottom_stack = PGROUNDUP(frame_p);
142    printf("backtrace:\n");
143    while(frame_p < bottom_stack){
144      printf("%p\n", *((uint64 *)(frame_p - 8)));
145      frame_p = *((uint64 *)(frame_p - 16));
146    }
147    return ;
148  }
```

kernel > C riscv.h > r_fp()

```
322  // return the frame pointer of the currently executing
323  // function for the backtrace function.
324  static inline uint64
325  r_fp()
326  {
327    uint64 x;
328    asm volatile("mv %0, s0" : "=r" (x) );
329    return x;
330  }
```

```
betty@ubuntu:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -d
rive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bu
s.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002d92
0x0000000080002c04
0x00000000800028a0
$ QEMU: Terminated
betty@ubuntu:~/xv6-labs-2020$ addr2line -e kernel/kernel
0x0000000080002d92
/home/betty/xv6-labs-2020/kernel/sysproc.c:73
0x0000000080002c04
/home/betty/xv6-labs-2020/kernel/syscall.c:144
0x00000000800028a0
/home/betty/xv6-labs-2020/kernel/trap.c:76
```

第三部分是实现一个系统调用 alarm 函数，其功能是通过两个参数——整数指定间隔时间，函数指针指定调用的函数——每每隔一段特定的系统的时间就调用一次函数。系统调用的增加和 Lab1 的过程相同，此处不再赘述，只关注具体的函数实现：

1. 修改 kernel/proc.h 中 proc 的结构体，增加成员变量来记录时间间隔、已经过去的时间、函数指针、调用对应函数时保存原进程的寄存器的 trapframe。

kernel > C proc.h > proc

```
105    char name[16];               // Process name (debugging)
106    int ticks_sum;               // ticks for calling the sigalarm
107    int ticks_cnt;               // the passed ticks
108    uint64 handler;              // handler in sigalarm call
109    struct trapframe *alarm_trapframe;   // save context when calling handler
110  };
111
```

2. 撰写 sigalarm 和 sigreturn 对应的系统函数，传参的过程与 Lab1 相似，同时为对应进程的成员变量赋值。

kernel > C sysproc.c > sys_sigalarm(void)

```
 99
100  uint64
101  sys_sigalarm(void)
102  {
103    int ticks;
104    uint64 handler;
105    struct proc *p = myproc();
106
107    if(argint(0, &ticks) < 0)
108      return -1;
109    if(argaddr(1, &handler) < 0)
110      return -1;
111
112    p->ticks_sum = ticks;
113    p->handler = handler;
114    p->ticks_cnt = 0;
115    p->alarm_trapframe = 0;
116
117    return 0;
118  }
```

kernel > C sysproc.c > sys_sigreturn(void)

```
120  uint64
121  sys_sigreturn(void)
122  {
123    struct proc *p = myproc();
124    if(p->alarm_trapframe != 0){
125      memmove(p->trapframe, p->alarm_trapframe, PGSIZE);
126      kfree(p->alarm_trapframe);
127      p->alarm_trapframe = 0;
128    }
129    return 0;
130  }
```

3. 在每个 tick（系统的时间单位）的中断中增加当前进程的 ticks 计数，并且判断是否进行对应的函数调用。这一部分在 kernel/trap.c 的 usertrap 函数中实现，时钟中断对应 which_dev（trap 原因的序号）为 2 的情况

kernel > C trap.c > devintr()

```
79   // give up the CPU if this is a timer interrupt.
80   if(which_dev == 2){
81     if(p->ticks_sum != 0){
82       p->ticks_cnt += 1;
83       if(p->ticks_sum == p->ticks_cnt && p->alarm_trapframe == 0){
84         p->ticks_cnt = 0;
85         p->alarm_trapframe = kalloc();
86         memmove(p->alarm_trapframe, p->trapframe, PGSIZE);
87         p->trapframe->epc = p->handler;
88       }
89       else{
90         yield();
91       }
92     }
93     else{
94       yield();
95     }
96   }
```

判断alarm_trapframe是否为0是要检查上一次使用handler是否已经返回

Lab3 最终的测试结果如下

```
betty@ubuntu:~/xv6-labs-2020$ ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (1.6s)
== Test running alarmtest == (3.8s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests == usertests: OK (117.7s)
== Test time ==
time: OK
Score: 85/85
```

三、Lab6

Lab6 是关于多线程的实验，在实验之前阅读了 xv6 说明的第七章和相关的代码，主要是关于 xv6 中进程切换的实现、sleep 和 wakeup 机制的实现及应用的介绍。其中进程切换主要是依靠 kernel/proc.c 中的 schedular 和 sched 函数的上下文切换、调用 swtch 函数实现；后续 sleep，wakeup 函数还有相关的 pipe，wait，exit，kill 函数的介绍中强调了进程锁的重要性和其中微妙的顺序。

第一个实验 Uthread 是实现用户层面的线程切换，这个 task 比较简单，但是需要注意的是栈的增长方向应当是高地址向低地址增长，所以在使用数组定义的栈 thread->stack 的时候要注意栈底是在&thread->stack[STACK_SIZE]。实验具体的实现如下：

1. 在 user/uthread.c 完善 thread 结构体，增加存储上下文（寄存器）的结构 context

user > C uthread.c > thread_schedule(void)

```
13  struct context {
14    uint64 ra;
15    uint64 sp;
16
17    // callee-saved
18    uint64 s0;
19    uint64 s1;
20    uint64 s2;
21    uint64 s3;
22    uint64 s4;
23    uint64 s5;
24    uint64 s6;
25    uint64 s7;
26    uint64 s8;
27    uint64 s9;
28    uint64 s10;
29    uint64 s11;
30  };
31
32  struct thread {
33    char       stack[STACK_SIZE]; /* the thread's stack */
34    int        state;             /* FREE, RUNNING, RUNNABLE */
35    struct context context;       /* the registers need saved */
36  };
```

2. 在 user/uthread.c 完善 thread_create 函数，将参数中的函数指针作为开始该线程时先执行的函数，并让栈指针指向栈底

```
user > C uthread.c > thread_create(void(*)())
89  void
90  thread_create(void (*func)())
91  {
92    struct thread *t;
93
94    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
95      if (t->state == FREE) break;
96    }
97    t->state = RUNNABLE;
98    // YOUR CODE HERE
99    t->context.ra = (uint64)func;
100   t->context.sp = (uint64)&(t->stack[STACK_SIZE - 1]);
101 }
```

3. 在 user/uthread.c 完善 thread_schedule 函数，进行线程的上下文切换

```
user > C uthread.c > thread_schedule(void)
79    current_thread = next_thread;
80    /* YOUR CODE HERE
81     * Invoke thread_switch to switch from t to next_thread:
82     * thread_switch(??, ??);
83     */
84    thread_switch((uint64)&t->context, (uint64)&current_thread->context);
```

第二部分是观察 notxv6/ph.c 代码，思考多线程可能带来的问题并且通过 pthreads 提供的 pthread_mutex 加锁的机制来避免该错误并保持多核带来的高效率。在原本 ph.c 文件中多核可能带来问题原因是：

线程共享全局变量，所以它们拥有同一个 table。在只有一个线程的时候该线程会按照顺序依次存入表项，但是多个线程（这里以两个为例）时，就有可能出错。按照如下的执行顺序就会丢失表项：thread1 执行到 put 函数中找到了 key1 可以放置的空项，即将执行 insert 函数此时 cpu 调度到 thread2，thread2 也需要放 key2 项，而且找到了和 thread1 一样的空槽，而由于 thread1 尚未插入表项，因此在 thread2 看来这是一个可以插入新表项的位置，由此即会导致 thread1 或者 thread2 的表项丢失（如果先调度 thread1 进行 insert，那么 thread1 的 entry 丢失，否则 thread2 的丢失）

为了解决这个问题添加的代码如下：

1. 为每个桶加锁，并进行初始化

```
notxv6 > C ph.c > lock
16  struct entry *table[NBUCKET];
17  pthread_mutex_t lock[NBUCKET];
```

```
notxv6 > C ph.c > main(int, char * [])
121   for (int i = 0; i < NBUCKET; i++){
122     pthread_mutex_init(&lock[i], NULL);
123   }
```

2. 在 put 中每个线程检查桶是否有对应的表项前加锁，防止其他线程修改该桶的状态；同时在该线程插入/修改项完成后归还锁。

```
notxv6 > C ph.c > put(int, int)
38  // 在hash table中插入项
39  static
40  void put(int key, int value)
41  {
42    int i = key % NBUCKET;
43
44    // is the key already present?
45    struct entry *e = 0;
46    pthread_mutex_lock(&lock[i]);
47    for (e = table[i]; e != 0; e = e->next) {
48      if (e->key == key){
49        break;
50      }
51    }
52    if(e){
53      // update the existing key.
54      e->value = value;
55    } else {
56      // the new is new.
57      insert(key, value, &table[i], table[i]);
58    }
59    pthread_mutex_unlock(&lock[i]);
60  }
```

第三部分是完善 notxv6/barrier.c 中的 barrier 函数，使之完成所有线程都要等到其他线程到达 barrier 函数之后才能离开的功能。主要的想法是使用 pthread_cond_wait 和 pthread_cond_broadcast 函数，前者的功能是在指定的 cond 条件上等待，并且释放原本持有的锁，在返回的时候重新取得锁；还是要注意使用互斥量，防止在对 bstate.uthread 计数增加的时候产生 race。下图是函数的具体实现。

```
notxv6 > C barrier.c > barrier()
25  static void
26  barrier()
27  {
28    // YOUR CODE HERE
29    //
30    // Block until all threads have called barrier() and
31    // then increment bstate.round.
32    //
33    pthread_mutex_lock(&bstate.barrier_mutex);
34    bstate.nthread += 1;
35    if(bstate.nthread == nthread){
36      bstate.nthread = 0;
37      bstate.round += 1;
38      pthread_cond_broadcast(&bstate.barrier_cond);
39    }
40    else{
41      pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
42    }
43    pthread_mutex_unlock(&bstate.barrier_mutex);
44  }
```

Lab6 最终的结果如下

```
betty@ubuntu:~/xv6-labs-2020$ ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (2.6s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (11.6s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (24.1s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (2.7s)
== Test time ==
time: OK
Score: 60/60
```

内容二：遇到的困难以及收获

1. Lab1

在第一个 Lab 中主要的困难是对于系统调用具体实现的不熟悉，加上内核代码体量较大，因此理解起来比较困难，但是根据官方的指南逐个文件修改，做完第一个 trace 函数之后，重新梳理了一遍系统调用的顺序，做第二个的时候就明了很多了。

2. Lab3

最开始切换 branch 的时候遇到了因为没有 commit 而不能直接切换的问题（从 Lab0 到 Lab1 的时候没碰到），查阅了一些资料觉得需要保留之前的一些修改，所以采用的 stash 和 stash pop 的方法。但是写完 backtrace 函数后发现不能运行，因为切换分支的时候覆盖了 MakeFile 文件中对 Lab1 设置的命令，而一些文件例如 sysinfo.h 没有了，所以要手动删掉多余的文件，后来才意识到应该之前直接 commit 让它覆盖/删除掉不需要的文件，这里折腾了比较久。

在 backtrace 的实现中，因为在取返回地址的时候没有弄清楚原本得到的只是地址所在的位置的指针，因此打印出来的地址在 addr2line 中一直不对，也卡了很久。完成试验后对于 trap 的寄存器的存储过程以及其中重要的一些量的替换（例如 trapframe->epc）以及为什么最终的实现可以完成我们以前宏观看到的那种效果有了更多原理上的了解。

3. Lab6

还是碰到了地址和实际的值直接没分太清楚的情况，在第一部分中一开始完全没发现在 thread_create 函数中给 sp 寄存器赋初值时用的是 t->stack[STACK_SIZE-1]，而没有使用取地址符，所以一直报下图的错误。

```
$ uthread
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x0000000000000120 stval=0xfffffffffffffff8
```

第三部分的完成过程中主要遇到的是对 pthread_cond_wait 一开始的理解不太正确，对于所有的线程都一刀切的在增加 uthread 都调用这个函数，后来才理解到说这样会导致所有的进程都 wait，就不存在一个可以继续执行 uthread 判断后进行 broadcast 的 wakeup 操作的线程了。

在本次线程相关的实验中，我对线程的运作原理还有 xv6 中线程切换的过程都有了更多的了解，同时，对互斥量的使用也更加熟悉了；其中感觉理解起来最困难的是 sleep 和 wakeup 函数的实现部分。在第一部分其实就有的“pipe 如何实现没有可以 read 的内容就阻塞等待”这个问题，也在后续的实验操作和学习中得到了答案。

内容三：对课程或 Lab 的意见和建议

可以增加一些小问题上的提示，比如说做完之后要 commit。有些同学在看到中文的 guide 之后就不太看英文的，但是原本的英文 guide 其实要详细很多，纠结在中文的 guide 上容易浪费时间，也许可以完善一下中文的 guide 或者在上面注释让大家去原本的 guide 上查看更多详细的步骤。

内容四：参考文献

[1] pthread_cond_wait 函数的定义：
https://baike.baidu.com/item/pthread_cond_wait/3011997?fr=aladdin
[2] 大小端：
https://www.cnblogs.com/yinheyi/p/5580789.html
[3] 函数指针：
https://blog.csdn.net/wangqingchuan92/article/details/77863373
[4] git commit 报错：
https://blog.csdn.net/weixin_34248023/article/details/88831183?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_title~default-0.no_search_link&spm=1001.2101.3001.4242
[5] auipc+jal 指令：
https://zhuanlan.zhihu.com/p/354654279
[6] 内联汇编：
https://blog.csdn.net/goaqnfv59125/article/details/70767097
[7] exec 族函数
https://blog.csdn.net/zhengqijun_/article/details/52852074